AN2768
Rev. 0, 7/2004

Implementation of a 128-Point FFT on the MRC6011 Device

by Zhao Li, Hirokazu Higa, and Ed Martinez

The Fast Fourier Transform (FFT) is an efficient way to compute the Discrete-time Fourier Transform (DFT) by exploiting symmetry and periodicity in the DFT. Because it is so efficient, the algorithm is implemented on many DSPs and hardware platforms for real-time applications. FFT applications include not only DSP but also spectrum analysis, speech processing, and filter designs where filter coefficients are determined according to the frequency response of the filter. The frequency response of a filter can be obtained by taking the Discrete Fourier Transform of its impulse response. Conversely, given the frequency response samples in the frequency domain, the time domain impulse response can be computed by taking the inverse DFT. For any discrete time sequences, the frequency components or spectral components can be obtained by taking the FFT on the discrete time sequences.

This application note describes how a 128-point FFT is implemented on the Freescale Semiconductor MRC6011 reconfigurable compute fabric (RCF) device, which has an array of processors working in parallel. FFT butterflies are normally performed sequentially. However, the MRC6011 device has 16 processors that can perform 16 simultaneous butterflies. The RCF programming techniques to achieve this parallelism are the subject of this application note. We begin with a look at the parallel RCF architecture, which is the basis for the parallel butterfly operations on the RC array. Also, we describe the effects of finite-precision arithmetic relative to the MRC6011 device and the FFT results.

CONTENTS

1 Basics of the Fast Fourier Transform

The DFT is the basis of the fast Fourier transform. The DFT of a finite-length sequence of length *N* is defined as follows:

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}, k = 0,1,..., N-1,$$

Equation 1

Where $W_N = e^{-j(2\pi/N)}$. The Inverse Discrete Fourier Transform is given by

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{-kn}, n = 0,1,..., N-1.$$

Equation 2

In these two equations, both *x[n]* and *X[k]* can be complex, so *N* complex multiplications and $(N-1)$ complex additions are required to compute each value of the DFT if we use **Equation 1** directly. Computing all *N* values of the frequency components requires a total of N^2 complex multiplications and $N(N-1)$ complex additions. To improve efficiency in computing the DFT, the properties of symmetry and periodicity of W_N^{kn} are exploited, and they are described as follows:

1. $W_N^{k[N-n]} = W_N^{-kn} = (W_N^{kn})^*$ (Complex conjugate symmetry)
2. $W_N^{kn} = W_N^{k(n+N)} = W_N^{(k+N)n}$ (Periodicity in n and k)

To simplify the notation of the two preceding expressions, we let *r=(kn)mod (N)* so that the property of symmetry becomes $W_N^{r+N/2} = -W_N^r$ and the property of periodicity is $W_N^{r+N} = W_N^r$. The W_N^r are often referred to as the twiddle factors in FFT computation, where $W_N = e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N)$ is the N^{th} root of unity.

1.1 Decimation in Time and Frequency

FFT algorithms decompose the DFT of a time domain sequence of length *N* into successively smaller DFTs—a divide and conquer strategy. Among the variety of divide and conquer algorithms are decimation in time (DIT) and decimation in frequency (DIF). DIT decomposes the time sequence x[n] into successively smaller sub-sequences until there are only two elements in the sequences for a radix-2 DFT. **Figure 1** and **Figure 2** show the decimation process for a time sequence of eight elements.

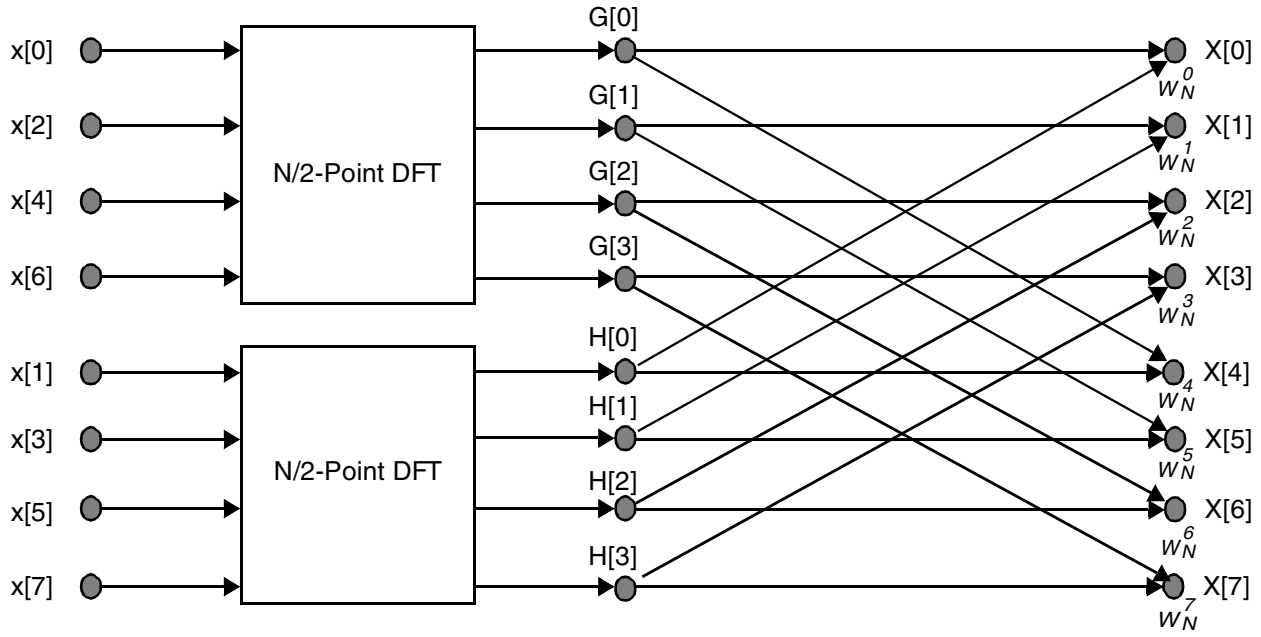


Figure 1. Decimation in Time of an N-Point DFT into Two (N/2)-Point DFT (N = 8)

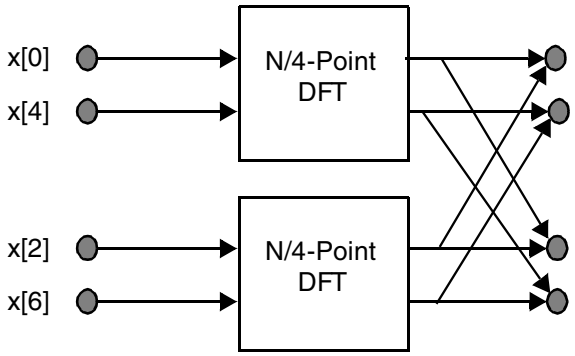


Figure 2. Decimation in Time of an (N/2)-Point DFT into Two (N/4)-Point DFT (N = 8)

Equation 3 demonstrates the decimation in time principle and is derived from the **Equation 1**, with the help of the symmetry and periodicity properties.

$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r]W_{N/2}^{rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1]W_{N/2}^{rk}$$
$$= G[k] + W_N^k H[k], \qquad k = 0,1,...,N-1.$$

Equation 3

DIF algorithms decompose the sequence of DFT coefficients *X[k]* into successively smaller sub-sequences. **Figure 3** illustrates the decimation process in the discrete frequency sequence of eight. Alternatively, the N-point DFT can be represented in terms of successively smaller sequences with N/2 frequency samples. **Figure 3** illustrates the decimation process for a discrete frequency of eight.

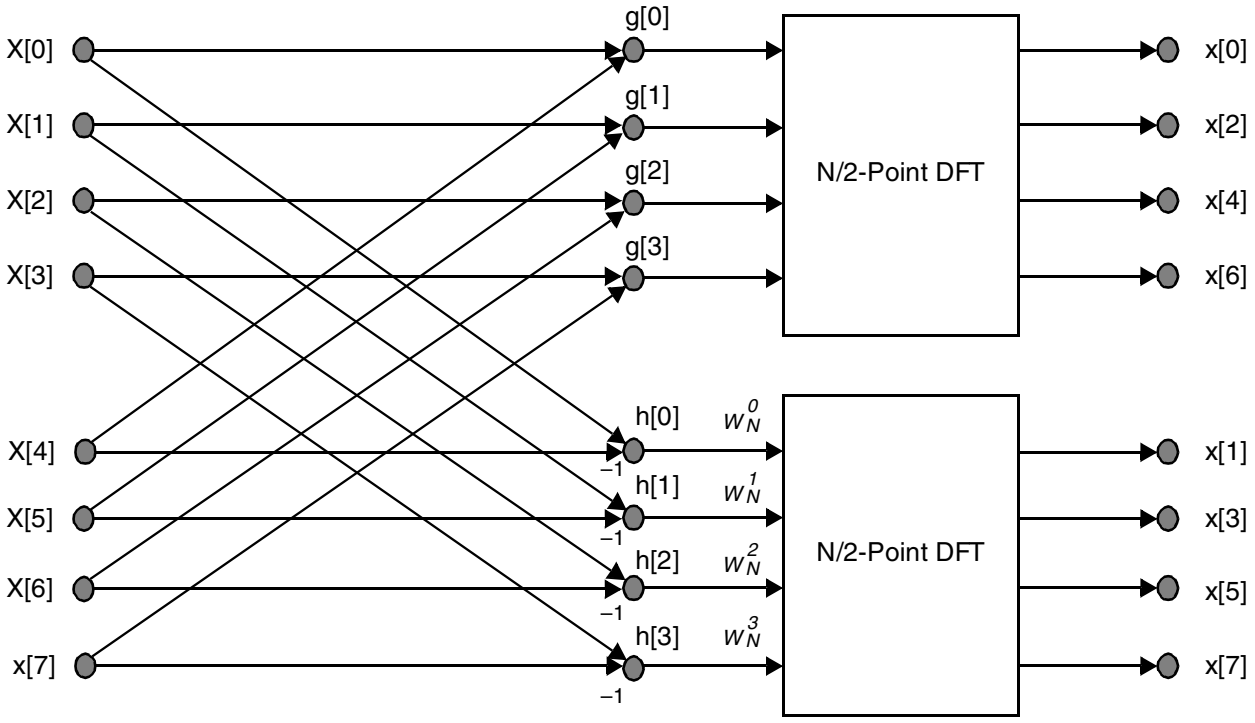


Figure 3. Decimation in Frequency of an N-Point DFT into Two (N/2)-Point DFT (N = 8)

Similar to DIT, the DIF principle is illustrated in the following equations, also originating from **Equation 1**.

$$X[2r] = \sum_{n=0}^{(N/2)-1} (x[n] + x[n+N/2]) W_{N/2}^{rn}, r = 0,1,...,(N/2)-1.$$

Equation 4

$$X[2r+1] = \sum_{n=0}^{(N/2)-1} (x[n] - x[n+(N/2)]) W_N^n W_{N/2}^{nr}, r = 0,1,...,(N/2)-1.$$

Equation 5

1.2 Radix-2 and Radix-4

The radix-2 FFT algorithm breaks the DFT calculation down into several 2-point DFTs, each consisting of a multiply-and-accumulate (MAC) operation called a bufferfly, as shown in **Figure 4**.

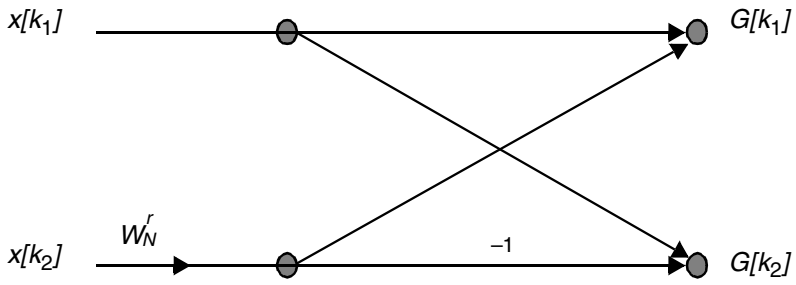


Figure 4. Radix-2 Butterfly

Figure 5 shows a radix-4 butterfly.

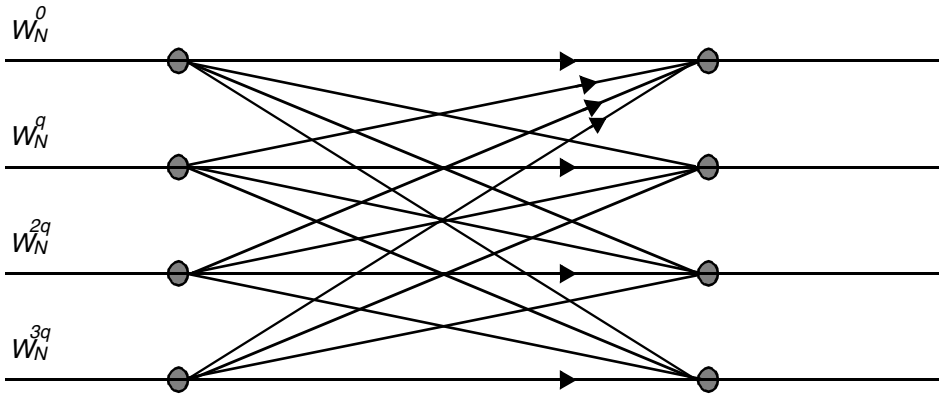


Figure 5. Radix-4 Butterfly

If the FFT number of points is a power of 2, the successive decomposition results in 2-point butterflies. If the number of points in an FFT is a power of 4, the FFT can be broken down into several 4-point DFTs. The twiddle factors of a radix-4 FFT possess the unique property that they belong to the set of $\{1,-1,j,-j\}$. Therefore, the radix-4 FFT requires fewer complex multiplications but more additions than the radix-2 FFT for the same number of points. The radix-4 FFT is more efficient than the radix-2 FFT, so its hardware implementation is simpler. The implementation of the radix-4 FFT is beyond the scope of this application note, which focuses on a radix-2 decimation in time implementation.

For an N-point FFT, the FFT algorithm decomposes the DFT into $\log_2 N$ stages, each of which consists of *N/2 butterfly* computations. Each butterfly takes two complex numbers $x_{re}[k_1]+jx_{im}[k_1]$ and $x_{re}[k_2]+jx_{im}[k_2]$ and computes from them two other numbers,

$$G[k_1]=x_{re}[k_1]+jx_{im}[k_1]+W_N^r\{x_{re}[k_2]+jx_{im}[k_2]\} \quad \textbf{Equation 6}$$

$$G[k_2]=x_{re}[k_1]+jx_{im}[k_1]-W_N^r\{x_{re}[k_2]+jx_{im}[k_2]\} \quad \textbf{Equation 7}$$

Where

$$W_N^r = e^{\frac{-j2\pi r}{N}} = \cos\left(\frac{2\pi r}{N}\right)-j\sin\left(\frac{2\pi r}{N}\right) \quad \textbf{Equation 8}$$

Figure 6 shows the 2-point butterfly operation.

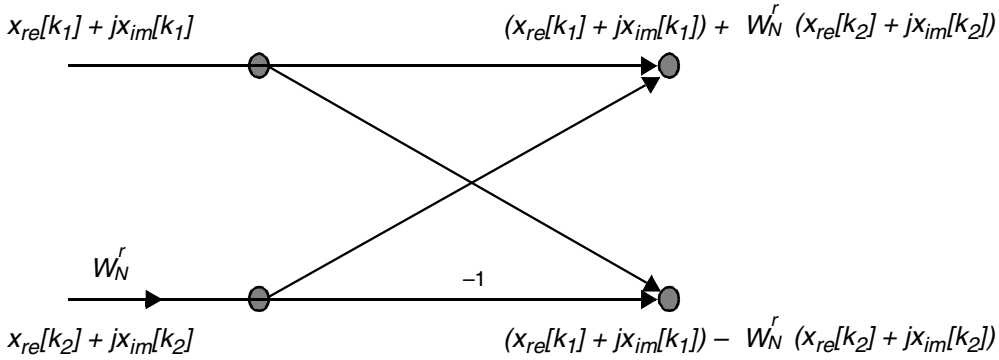


Figure 6. Radix-2 Butterfly Operation

If we define

$$C = \cos(2\pi r / N) \quad \text{and} \quad S = \sin(2\pi r / N) \qquad \textbf{Equation 9}$$

The 2-point butterfly output in **Equation 6** and **Equation 7** can be formulated as follows:

$$\begin{aligned} G[k_1] &= x_{re}[k_1] + x_{re}[k_2]C + x_{im}[k_2]S \\ &\quad + j\{x_{im}[k_1] + x_{im}[k_2]C - x_{re}[k_2]S\} \\ &= G_{re}[k_1] + jG_{im}[k_1] \end{aligned} \qquad \textbf{Equation 10}$$

$$\begin{aligned} G[k_2] &= x_{re}[k_1] - \{x_{re}[k_2]C + x_{im}[k_2]S\} \\ &\quad + j\{x_{im}[k_1] - x_{im}[k_2]C + x_{re}[k_2]S\} \\ &= G_{re}[k_2] + jG_{im}[k_2] \end{aligned} \qquad \textbf{Equation 11}$$

Where

$$\begin{aligned} G_{re}[k_2] &= 2x_{re}[k_1] - G_{re}[k_1] \\ G_{im}[k_2] &= 2x_{im}[k_1] - G_{im}[k_1] \end{aligned} \qquad \textbf{Equation 12}$$

This relationship in **Equation 12** can be used to simplify the computation of the second output.

1.3 Bit Reversal

The radix-2 FFT needs bit-reversed data ordering; that is, the MSBs become LSBs and the LSBs become MSBs. **Table 1** shows an example of bit-reversal with an 8-point input sequence.

Table 1. Bit Reversal with 8-Point Input Sequence

Decimal Number	0	1	2	3	4	5	6	7
Binary Equivalent	000	001	010	011	100	101	110	111
Bit reversed Binary	000	100	010	110	001	101	011	111
Decimal Equivalent	0	4	2	6	1	5	3	7

Bit reversal is convenient for in-place computation of an FFT. With bit reversal of the input sequence, the output sequence is ordered normally from 0 to N –1.

2 MRC6011 Architecture Overview

The MRC6011 processor is the first Freescale device based on the RCF core. It is a highly integrated system-on-a-Chip (SoC) that combines six reconfigurable RCF cores into a homogeneous compute node. The six RCF cores reside in two RC modules containing three RCF cores each. The RCF cores on the MRC6011 device instantiate the 2×8 processing array depicted in **Figure 17**. Both RC modules are accessible via an antenna slave interface and two slave I/O Interfaces. Each antenna interface can interact with up to 16 antennas, and each RCF core can manipulate the data from two antennas. The processed data goes either to one of the two slave I/O bus interfaces (industry-wide DSP device-compatible) or to another RCF core in the same or the adjacent module. At 250 MHz,

the six-core MRC6011 device delivers a peak performance of 24 giga complex correlations per second with a sample resolution of 8 bits for I and Q inputs each–or up to 48 giga complex correlations per second at a resolution of 4 bits.

The Freescale RCF technology is designed to meet the processing needs of computationally intensive tasks such as the FFT. The Freescale MRC6011 RCF device is a highly effective device on which to implement the FFT. The RCF core technology of the MRC6011 device is based on an array of processing elements that combines efficient parallel computing with fast and flexible reconfiguration and data routing. **Figure 7** shows the highly parallel architecture of the RCF core. The array of processors perform DSP-like operations for compute intensive algorithms in many applications. The RCF core contains the array of reconfigurable cells (RCs), the RC controller, a context memory that acts as the RC program memory, and a frame buffer that is the data memory of the RC array.

The data and instruction memory of the RC controller resides outside the RCF core and is associated with an ICache and a DCache. The sequence generator and interleaver interact only with other internal core components and are designed for CDMA operations. The input buffer receives data from outside with a 32-bit input bus, and the DMA controller handles all other data and RC array program transfers from external memory space. Multiple cores can interconnect via internal and external buses to expand the RCF parallelism with arbitrators and glue logic. Multiple cores can perform more parallel computations with careful programming of the shared resources.

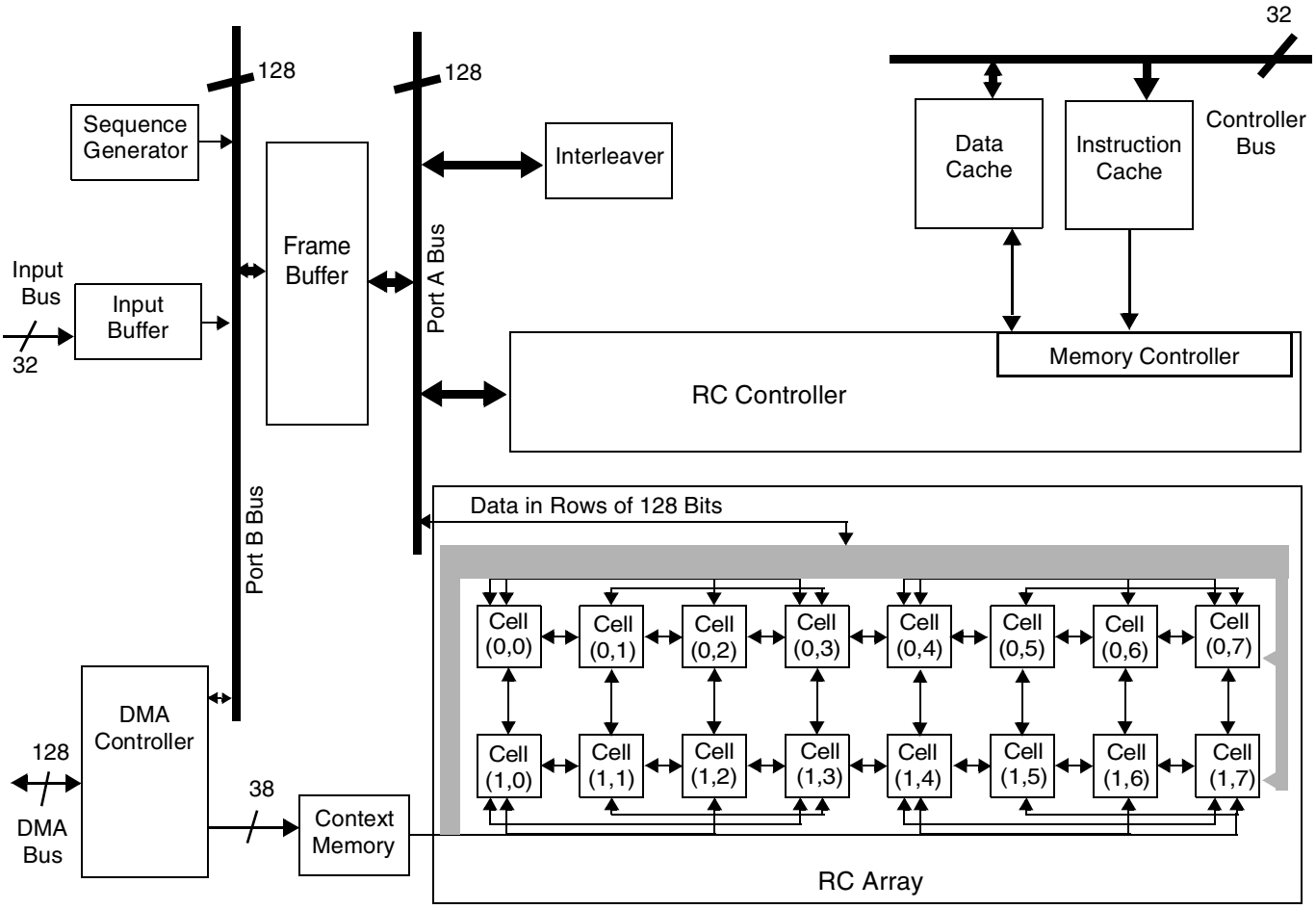


Figure 7. RCF Core with 8 × 2 Reconfigurable Cells

The RC controller executes the main control process of an application and schedules every execution cycle of the processing array. The actual array instructions reside in the context memory. The wide interconnect path between the context memory and the RC array allows single-cycle reconfiguration of the processing units. The DMA controller is the interface to the main system bus. The input buffer connects to the antenna input ports and appropriately interleaves and combines the data for efficient writes into the frame buffer. The following sections describe the frame buffer and RC array to clarify how parallel FFT butterflies are performed on this architecture.

2.1 Frame Buffer

The frame buffer is a dual-port RAM that connects one side to the RC controller, RC array, and interleaver, and the other side to the DMA controller, input buffer, and sequence generator via a separate 128-bit bus. The frame buffer is organized in rows (horizontally) and banks (vertically) to facilitate easy data transfer to and from the RC array. Since the RC array is organized into two rows of eight RCs each, there are eight two-byte banks in the frame buffer.

As shown in **Figure 8**, an Omega network between the frame buffer and the RC array routes data into the RC array for computation in each RC. The Omega network can broadcast a byte or a 16-bit word into all RCs so that all RCs process the same data. Alternatively, the Omega network can transfer a row of data into the RCs column-wise so that each RC processes different data. The 128-point FFT application employs both of these data routing mechanisms.

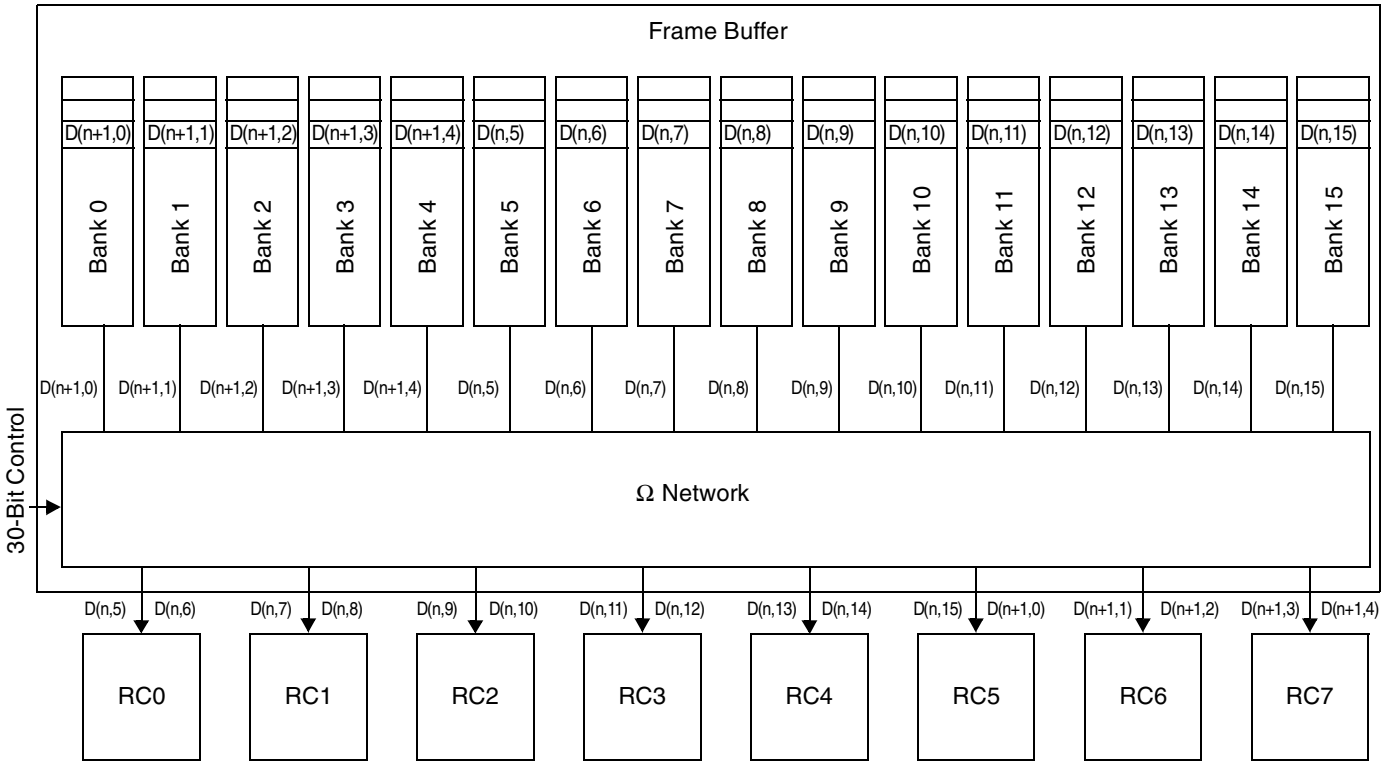


Figure 8. Omega Network Location

The 40 KB frame buffer is organized into 2560 rows × 16 bytes per row. In the 128-point FFT computation, input data and twiddle factors are both 16-bit integers or 16-bit fixed-point numbers (–1.0 to +1.0), so there are eight input data or twiddle factors in each frame buffer row. **Section 3** describes how the entire 128 complex data input samples and the various twiddle factors are stored to support parallel butterfly operations on the RC array.

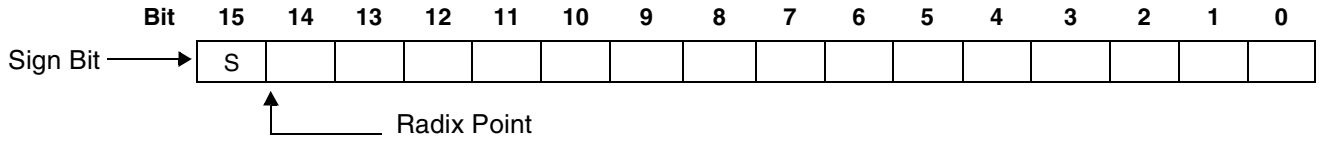
2.2 RC Array

The 16-element RC array operates using the same clock as the RC controller that schedules array operations. Each RC can complete a MAC operation, so the array performs a maximum of 16 MACs on each core clock cycle. The RC array takes data directly from the frame buffer through the Omega network to perform operations on data already in the RC registers. Depending on the application, the intermediate results are either temporarily stored in one of the sixteen registers in each RC or they are output back to the frame buffer rows for subsequent computing.

Sometimes, data exchange between two or more cells is needed. Sophisticated inter-cell connections enable the cells to exchange data items in selected rule-governed ways to feed data to locations where it is needed on the next clock cycle. In the 128-point FFT application, with decimation in time, the bit reversal of the input data requires this type of operation to generate the 128-point complex frequency components in normal order.

3 The FFT on the MRC6011 Device

The 128-point complex input samples are 16-bit two's complement fixed-point real and imaginary numbers with a value between –1.0 and +1.0 and in the following format:



Since the frame buffer is organized into 128-bit rows, each row can store eight 16-bit real or eight 16-bit imaginary numbers. For complex numbers, the first row usually stores the real parts and the next row stores the imaginary parts, as shown in **Table 2**.

Table 2. Conventional Data Storage in the Frame Buffer for Complex Numbers

Bytes 0/1	2/3	4/5	6/7	8/9	10/11	12/13	14/15
Data[0].re	Data[1].re	Data[2].re	Data[3].re	Data[4].re	Data[5].re	Data[6].re	Data[7].re
Data[0].im	Data[1].im	Data[2].im	Data[3].im	Data[4].im	Data[5].im	Data[6].im	Data[7].im

Alternatively, the real parts and the imaginary parts can be grouped together and occupy consecutive rows if they are to be accessed one after the other. We choose this type of storage over the one shown in **Table 2** for the following reasons:

- Reading consecutive rows of real or imaginary numbers into RC registers is simple. Conveniently, the input data is transposed for bit reversal in the RC registers.
- The twiddle factors are not loaded into the RC registers for the butterfly operations. It is more efficient if real parts of the twiddle factors are collocated in consecutive rows of the frame buffer.

3.1 Input Data Storage in the Frame Buffer

The 128 complex input samples are stored so that 128 real numbers occupy 16 rows of the frame buffer and 128 imaginary numbers occupy another 16 rows of the frame buffer, as illustrated in **Figure 9**.

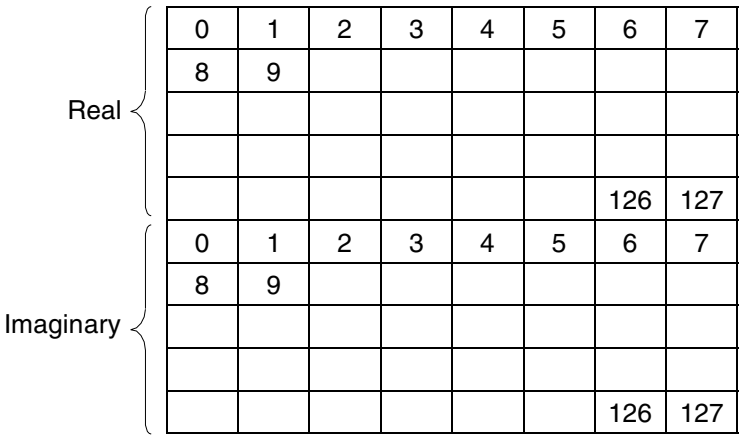


Figure 9. Input Data Storage in Frame Buffer

Since the input data is transposed to produce the bit-reversed input for a decimation in time FFT, the initial data organization in the frame buffer is not crucial. The input data are signed 2's complement 16-bit fixed-point numbers between –1.0 and +1.0.

3.2 Twiddle Factor Storage in Frame Buffer

The organization of the twiddle factors in the frame buffer is important for the butterfly operations because the MAC operation takes the twiddle factor as an argument directly from the frame buffer (see **Figure 10**). Both real and imaginary twiddle factors for every stage of the butterfly operation are accessed with two pointers pointing to the two parts of the twiddle factors, respectively. The twiddle factors are 2's complement 16-bit fixed-point numbers between –1.0 and +1.0. For easy access during the butterfly operations, the stage 1 twiddle factor is a single twiddle factor that coincides with the first of the stage 2 twiddle factors.

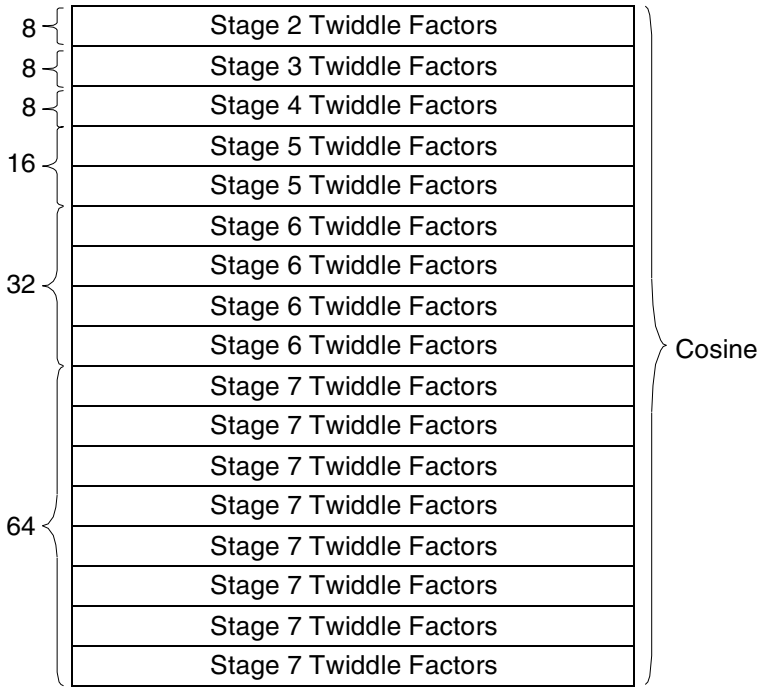


Figure 10. Real Twiddle Factor Storage in the Frame Buffer

The imaginary (or sine) part of the twiddle factors is stored immediately after the real (or cosine) part of the twiddle factors. Each part of the twiddle factors has an associated pointer that is updated during the butterfly operations only when the frame buffer offset exceeds the limit of addressability of the instruction, which is 64.

3.3 Transposition for Bit Reversal

The bit reversal of 128 complex input data requires two 128-element transpositions. Each 128-element transposition has two simultaneous 64-element transpositions on the top and bottom rows of the RC array. The transposition is performed for the 128 real numbers and then for the 128 imaginary numbers. The steps in the transposition are as follows:

1. Set up circular buffer 0.
2. Load 64 real numbers into the R0, R4, R2, R6, R1, R5, R3, and R7 registers of the first row of RCs in the array.
3. Load the next 64 real numbers into the R0, R4, R2, R6, R1, R5, R3, and R7 registers of the second row of RCs in the array.
4. Transpose both rows to shuffle the 128 real numbers:

   R0–R7 → R8–R15
   R0–R7 → R0–R7
5. Push the G5–G8 data to the frame buffer and reshuffle the G1–G4 data for the butterfly.

Before the transposition begins, circular buffer 0 is selected and initialized to point to the beginning of the input data in the frame buffer so that data can be loaded into the RC registers without incurring pointer updates. The code for circular buffer set-up and consecutive frame buffer row loading is listed in **Example 1**.

Example 1. RCF Circular Buffer Initialization and Frame Buffer Row Loading

```
_DEC_CIRCULAR_BUFFER_SELECT = 0;
_DEC_AUTOINCREMENT0 = (unsigned long)psiFBInputData;

MORPHO_ASM( psiFBInputData )

            /** LOAD INPUT DATA **/
            /***** Row0 *****/
            /* Load input data(Re0-63)*/
            CELL{0,*}:OUT_REG R0=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{0,*}:OUT_REG R4=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{0,*}:OUT_REG R2=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{0,*}:OUT_REG R6=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{0,*}:OUT_REG R1=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{0,*}:OUT_REG R5=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{0,*}:OUT_REG R3=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{0,*}:OUT_REG R7=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            /***** Row1 *****/
            /* Load input data(Re64-127)*/
            CELL{1,*}:OUT_REG R0=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{1,*}:OUT_REG R4=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{1,*}:OUT_REG R2=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{1,*}:OUT_REG R6=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{1,*}:OUT_REG R1=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{1,*}:OUT_REG R5=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{1,*}:OUT_REG R3=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
            CELL{1,*}:OUT_REG R7=BYP{FB++{16,OMEGA_RT, COL_BUS, WORD}};
```

At the end of the real input data loading, the input data are interleaved and are ready for the final shuffling to complete the transpose operation. **Figure 11** shows the layout of the data at this stage.

	Cell 0	Cell 1	Cell 2	Cell 3	Cell 4	Cell 5	Cell 6	Cell 7
Register 0	0	1	2	3	4	5	6	7
Register 1	32	33	34	35	36	37	38	39
Register 2	16	17	18	19	20	21	22	23
Register 3	48	49	50	51	52	53	54	55
Register 4	8	9	10	11	12	13	14	15
Register 5	40	41	42	43	44	45	46	47
Register 6	24	25	26	27	28	29	30	31
Register 7	56	57	58	59	60	61	62	63
Register 0	64	65	66	67	68	69	70	71
Register 1	96	97	98	99	100	101	102	103
Register 2	80	81	82	83	84	85	86	87
Register 3	112	113	114	115	116	117	118	119
Register 4	72	73	74	75	76	77	78	79
Register 5	104	105	106	107	108	109	110	111
Register 6	88	89	90	91	92	93	94	95
Register 7	120	121	122	123	124	125	126	127

Figure 11. 128-Point Input Data Loaded into the RC Registers in Top and Bottom Rows of the RC Array

Figure 12 shows that the transposition of the two 8×8 matrices simultaneously completes in the top and bottom rows of the RC array.

	Cell 0	Cell 1	Cell 2	Cell 3	Cell 4	Cell 5	Cell 6	Cell 7
Register 0	0	32	16	48	8	40	24	56
Register 1	1	33	17	49	9	41	25	57
Register 2	2	34	18	50	10	42	26	58
Register 3	3	35	19	51	11	43	27	59
Register 4	4	36	20	52	12	44	28	60
Register 5	5	37	21	53	13	45	29	61
Register 6	6	38	22	54	14	46	30	62
Register 7	7	39	23	55	15	47	31	63
Register 0	64	96	80	112	72	104	88	120
Register 1	65	97	81	113	73	105	89	121
Register 2	66	98	82	114	74	106	90	122
Register 3	67	99	83	115	75	107	91	123
Register 4	68	100	84	116	76	108	92	124
Register 5	69	101	85	117	77	109	93	125
Register 6	70	102	86	118	78	110	94	126
Register 7	71	103	87	119	79	111	95	127

Figure 12. 128-Point Input Data Completed Transpose in Top and Bottom Rows of the RC Array

The shuffling that brings the data format from that in **Figure 11** to that in **Figure 12** uses the connectivity of the RCs within quadrants and the available fast lanes between the quadrants of the RCs. Eight cycles are required to complete this final shuffling operation. The code segment for this operation is listed in **Example 2**.

Example 2. RCF Code to Transpose Two 8 × 8 Matrices with One on Each Row

```
/** 8X8 TRANSPOSE OF SYMBOLS [R0-R7]->[R8-R15] **/
       MORPHO{//1.cycle
              CELL{*,0}:R0  R8 =BYP{R0{*,0}};
              CELL{*,1}:R1  R9 =BYP{R1{*,1}};
              CELL{*,2}:R4  R12=BYP{R2{*,4}};//Expr lane right->left
              CELL{*,3}:R5  R13=BYP{R3{*,5}};//Expr lane right->left
              CELL{*,4}:R2  R10=BYP{R4{*,2}};//Expr lane left->right
              CELL{*,5}:R3  R11=BYP{R5{*,3}};//Expr lane left->right
              CELL{*,6}:R6  R14=BYP{R6{*,6}};
              CELL{*,7}:R7  R15=BYP{R7{*,7}};
       }

       MORPHO{//2.cycle
              CELL{*,0}:R1  R9 =BYP{R0{*,1}};
              CELL{*,1}:R0  R8 =BYP{R1{*,0}};
              CELL{*,2}:R5  R13=BYP{R2{*,5}};//Expr lane right->left
              CELL{*,3}:R4  R12=BYP{R3{*,4}};//Expr lane right->left
              CELL{*,4}:R3  R11=BYP{R4{*,3}};//Expr lane left->right
              CELL{*,5}:R2  R10=BYP{R5{*,2}};//Expr lane left->right
              CELL{*,6}:R7  R15=BYP{R6{*,7}};
              CELL{*,7}:R6  R14=BYP{R7{*,6}};
       }

       MORPHO{//3.cycle
              CELL{*,0}:R6  R14=BYP{R0{*,6}};//Expr lane right->left
              CELL{*,1}:R7  R15=BYP{R1{*,7}};//Expr lane right->left
              CELL{*,2}:R2  R10=BYP{R2{*,2}};
              CELL{*,3}:R3  R11=BYP{R3{*,3}};
              CELL{*,4}:R4  R12=BYP{R4{*,4}};
              CELL{*,5}:R5  R13=BYP{R5{*,5}};
              CELL{*,6}:R0  R8 =BYP{R6{*,0}};//Expr lane left->right
              CELL{*,7}:R1  R9 =BYP{R7{*,1}};//Expr lane left->right
       }

       MORPHO{//4.cycle
              CELL{*,0}:R7  R15=BYP{R0{*,7}};//Expr lane right->left
              CELL{*,1}:R6  R14=BYP{R1{*,6}};//Expr lane right->left
              CELL{*,2}:R3  R11=BYP{R2{*,3}};
              CELL{*,3}:R2  R10=BYP{R3{*,2}};
              CELL{*,4}:R5  R13=BYP{R4{*,5}};
              CELL{*,5}:R4  R12=BYP{R5{*,4}};
              CELL{*,6}:R1  R9 =BYP{R6{*,1}};//Expr lane left->right
              CELL{*,7}:R0  R8 =BYP{R7{*,0}};//Expr lane left->right
       }
```

```
MORPHO{//5.cycle
       CELL{*,0}:R4  R10=BYP{R0{*,2}};
       CELL{*,1}:R5  R11=BYP{R1{*,3}};
       CELL{*,2}:R0  R14=BYP{R2{*,6}};//Expr lane right->left
       CELL{*,3}:R1  R15=BYP{R3{*,7}};//Expr lane right->left
       CELL{*,4}:R6  R8 =BYP{R4{*,0}};//Expr lane left->right
       CELL{*,5}:R7  R9 =BYP{R5{*,1}};//Expr lane left->right
       CELL{*,6}:R2  R12=BYP{R6{*,4}};
       CELL{*,7}:R3  R13=BYP{R7{*,5}};
}

MORPHO{//6.cycle
       CELL{*,0}:R5  R11=BYP{R0{*,3}};
       CELL{*,1}:R4  R10=BYP{R1{*,2}};
       CELL{*,2}:R1  R15=BYP{R2{*,7}};//Expr lane right->left
       CELL{*,3}:R0  R14=BYP{R3{*,6}};//Expr lane right->left
       CELL{*,4}:R7  R9 =BYP{R4{*,1}};//Expr lane left->right
       CELL{*,5}:R6  R8 =BYP{R5{*,0}};//Expr lane left->right
       CELL{*,6}:R3  R13=BYP{R6{*,5}};
       CELL{*,7}:R2  R12=BYP{R7{*,4}};
}

MORPHO{//7.cycle
  CELL{*,0}:R2  R12=BYP{R0{*,4}};//Expr lane right->left
  CELL{*,1}:R3  R13=BYP{R1{*,5}};//Expr lane right->left
  CELL{*,2}:R6  R8 =BYP{R2{*,0}};
  CELL{*,3}:R7  R9 =BYP{R3{*,1}};
  CELL{*,4}:R0  R14=BYP{R4{*,6}};
  CELL{*,5}:R1  R15=BYP{R5{*,7}};
  CELL{*,6}:R4  R10=BYP{R6{*,2}};//Expr lane left->right
  CELL{*,7}:R5  R11=BYP{R7{*,3}};//Expr lane left->right
}

MORPHO{//8.cycle
       CELL{*,0}:R3  R13=BYP{R0{*,5}};//Expr lane right->left
       CELL{*,1}:R2  R12=BYP{R1{*,4}};//Expr lane right->left
       CELL{*,2}:R7  R9 =BYP{R2{*,1}};
       CELL{*,3}:R6  R8 =BYP{R3{*,0}};
       CELL{*,4}:R1  R15=BYP{R4{*,7}};
       CELL{*,5}:R0  R14=BYP{R5{*,6}};
       CELL{*,6}:R5  R11=BYP{R6{*,3}};//Expr lane left->right
       CELL{*,7}:R4  R10=BYP{R7{*,2}};//Expr lane left->right
}
```

After the 128 real numbers are transposed and transferred into the R[8–15] registers of the RC array, the 128 imaginary numbers are read from the frame buffer to repeat the same transpose operations in registers R[0–7]. At this time, both the real and imaginary parts have completed the bit reversal. However, the 128 × 2 = 256 data items occupy all 16 (registers/cell) × 16 cells = 256 registers in the RC array, leaving no room for further operations. To continue with subsequent butterfly operations, we can keep only half of the transposed (or bit reversed) data in the RC array and move half of the data out to the frame buffer for temporary storage, making 128 registers available for butterfly operations.

The bit reversed data is organized into groups of 16 complex samples to form eight groups of data. Groups 1 to 4 are selected to stay in the RC registers, and Groups 5 to 8 are temporarily placed into the frame buffer.

3.4 Post-Transpose Data Organization

The 128 complex data are divided into eight groups, each with 16 complex numbers, as shown in **Figure 13**. Simultaneous butterfly operations are performed in stages within groups and between groups to reach the final FTT results.

	G1	G2	G3	G4	G5	G6	G7	G8
Cell 0	0	4	2	6	1	5	3	7
	64	68	66	70	65	69	67	71
Cell 1	32	36	34	38	33	37	35	39
	96	100	98	102	97	101	99	103
Cell 2	16	20	18	22	17	21	19	23
	80	84	82	86	81	85	83	87
Cell 3	48	52	50	54	49	53	51	55
	112	116	114	118	113	117	115	119
Cell 4	8	12	10	14	9	13	11	15
	72	76	74	78	73	77	75	79
Cell 5	40	44	42	46	41	45	43	47
	104	108	106	110	105	109	107	111
Cell 6	24	28	26	30	25	29	27	31
	88	92	90	94	89	93	91	95
Cell 7	56	60	58	62	57	61	59	63
	120	124	122	126	121	125	123	127

Figure 13. Data Partitioned into Groups for Parallel Butterflies

3.5 Parallel Butterfly Operations

An FFT with 128-point input data is composed of seven stages of butterfly operations. In the first four stages, 16 data samples from the same group are used in the butterfly computation. No data exchange is necessary between different groups. However, in stage 5, data exchange between groups is needed. For example, G1 data and G2 data form butterflies in stage 5. The butterfly span increases as the number of stages increases. In stage 7, G1 data work with G5 data to complete butterflies of the final stage.

If we partition the data of the 8-point DIT FFT into groups of 2, stage 1 of this particular FFT needs data only from its own group, and stage 2 requires data with increased distance—that is, from G1 and G3 or G2 and G4. The last stage of this FFT requires data with a distance of half the FFT length, as shown in **Figure 14**.

The first half of the 128-point input data is independent of the second half of the data until stage 7 of the FFT. Taking this into consideration, as well as the number of registers available in RCs for the FFT computations, we divide the 128-point FFT into three steps. In the first step, G1–G4 are register-resident while the first six stages of their butterfly computations are performed. Then the intermediate results from the first six stages of G1–G4 data are sent to the frame buffer for temporary storage. In the second step, G5–G8 are moved into the RC registers to perform the first six stages of the butterfly operations and the intermediate results from these groups stay in the RC. In the last step, the partial results of G5–G8 in RC the registers are paired with those of G1–G4 in the frame buffer to form the butterflies of stage 7 and complete the 128-point FFT.

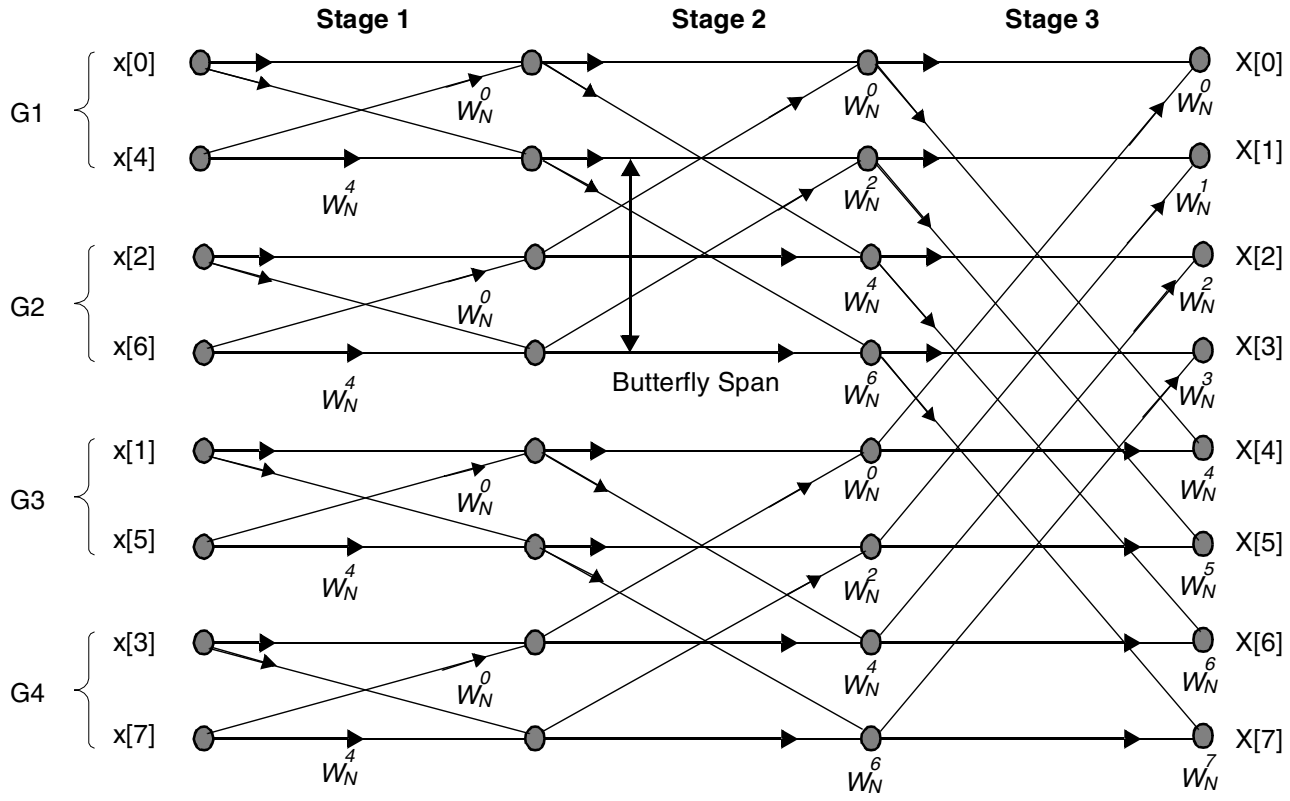


Figure 14. 8-point DIT FFT Butterfly Diagram

The procedure for computing the 128-point FFT on the RCF architecture is summarized as follows:

1. Groups 1–4 are in stage 1 for six butterfly operations.
2. The intermediate results are of groups 1–4 are sent to the frame buffer.
3. Groups 5–8 are in stage 1 for six butterfly operations.
4. (G1, G5), (G2, G6), (G3, G7), (G4, G8) are paired to complete stage 7 butterfly operations.
5. The final results are output to the frame buffer.

Since each group has 16 data items, eight butterflies can be formed. One row of RCs can perform eight butterflies simultaneously, so the 2×8 RC array can perform 16 simultaneous butterfly operations. In step one, for example, the group 1 and group 2 butterfly operations are simultaneously performed on the first and second rows of the RC array. Next are the group 3 and group 4 butterfly operations for stage 1. The following sections explain how the basic 2-point butterfly operation is performed on the RCs. Examples are presented from stage to stage.

3.5.1 2-point Butterfly Operation on RC

In **Example 3**, the group 1 (G1) butterfly operations are performed on the top row of the RC array, and the G2 butterfly operations are performed on the bottom row. Each cell gets two complex samples. The results of the butterfly operation are inputs to the butterfly operation at the next stage. A total of 16 butterfly operations are performed in parallel, and each cell computes one butterfly at a time. The code segment in **Example 3** shows how the 16 butterflies are simultaneously performed on the RC array.

Example 3. Parallel Butterfly Operations without Scaling Down

```
_DEC_CIRCULAR_BUFFER_SELECT = 0;
   _DEC_AUTOINCREMENT0 = (unsigned long)psiFBInputData;

      /* Stage1 */
      /* G1,G2[Re,Im] = k1[R4,R0], k2[R5,R1], tmp = R8,R9 */
      CELL{*,*} R13 = MULSIH{FB{psiFBInputTwiddleCos, 0, OMEGA_BR2, COL_BUS, WORD},R5, R14} << 1;
      CELL{*,*} R12 = MULSIH{FB{psiFBInputTwiddleCos, 0, OMEGA_BR2, COL_BUS, WORD},R1, R14} << 1;
      CELL{*,*} R11 = MULSIH{FB{psiFBInputTwiddleSin, 0, OMEGA_BR2, COL_BUS, WORD},R1, R14} << 1;
      CELL{*,*} R10 = MULSIH{FB{psiFBInputTwiddleSin, 0, OMEGA_BR2, COL_BUS, WORD},R5, R14} << 1;
      CELL{*,*} NOP{};
      CELL{*,*} R13 = ADD{R13, R11};
      CELL{*,*} R12 = SUB{R12, R10};
      CELL{*,*} R8 = ADD{R4, R13};
      CELL{*,*} R9 = ADD{R0, R12};
      CELL{*,*} R4 = MULSIL{R15,R4,R8};
      CELL{*,*} R0 = MULSIL{R15,R0,R9};
```

With reference to **Equation 10**, the first four cycles compute $x_{re}[k_2]C$, $x_{im}[k_2]C$, $x_{im}[k_2]S$ and $x_{re}[k_2]S$ with rounding, respectively. All R14 registers hold a value of 0x8000. They are first sign extended to 0xFFFF8000 and then subtracted from the products so that the sixteenth bit is rounded to improve precision in the 128-point FFT.

The NOP operation is for MAC pipeline delay when an ADD instruction follows a MAC instruction. The four cycles after the NOP compute the real and imaginary parts $G_{re}[k_1]$ and $G_{im}[k_1]$ of $G[k_1]$. The last two cycles compute $G_{re}[k_2]$ and $G_{im}[k_2]$ to complete the stage 1 butterfly operations for G1 and G2. **Table 3** provides the notation for the input/output and intermediate results.

Table 3. Symbols of Input/Output and Intermediate Results of the 128-point FFT

	Symbols
Butterfly Input	*X*[k]
Stage 1 Output	G[k]
Stage 2 Output	H[k]
Stage 3 Output	I [k]
Stage 4 Output	J[k]
Stage 5 Output	K[k]
Stage 6 Output	L[k]
Stage 7 Output	*Y*[k]

Figure 15 shows a complete picture of the stage 1 butterfly operation on the RC array for G1 and G2 data, which matches the code segment described in **Example 3**. The top portion of the figure shows the eight parallel butterfly operations for G1 data, and the lower portion shows another eight parallel butterfly operations for G2 data. In each cell are two complex samples or four 16-bit values stored in registers R4, R0, R5, R1, respectively. The R4{0,*} notation indicates the R4 register in every cell in the first row of the RC array. The R4 registers hold the real portions of half of the G1 samples. The corresponding imaginary portions are in R0{0,*} of these cells.

As indicated in the code segment of **Example 3**, the outputs of the butterflies go into R8, R9, R4, R0 of all 16 cells of the RC array, which are addressed as R8{*,*}. Stage 1 butterfly operations are formed within individual cells. As shown in **Figure 14**, the butterfly span increases with each stage. To continue with the 16 parallel butterfly computations as in stage 1 for subsequent stages, data regrouping is needed. Data regrouping is handled via register exchanges on the RC array.

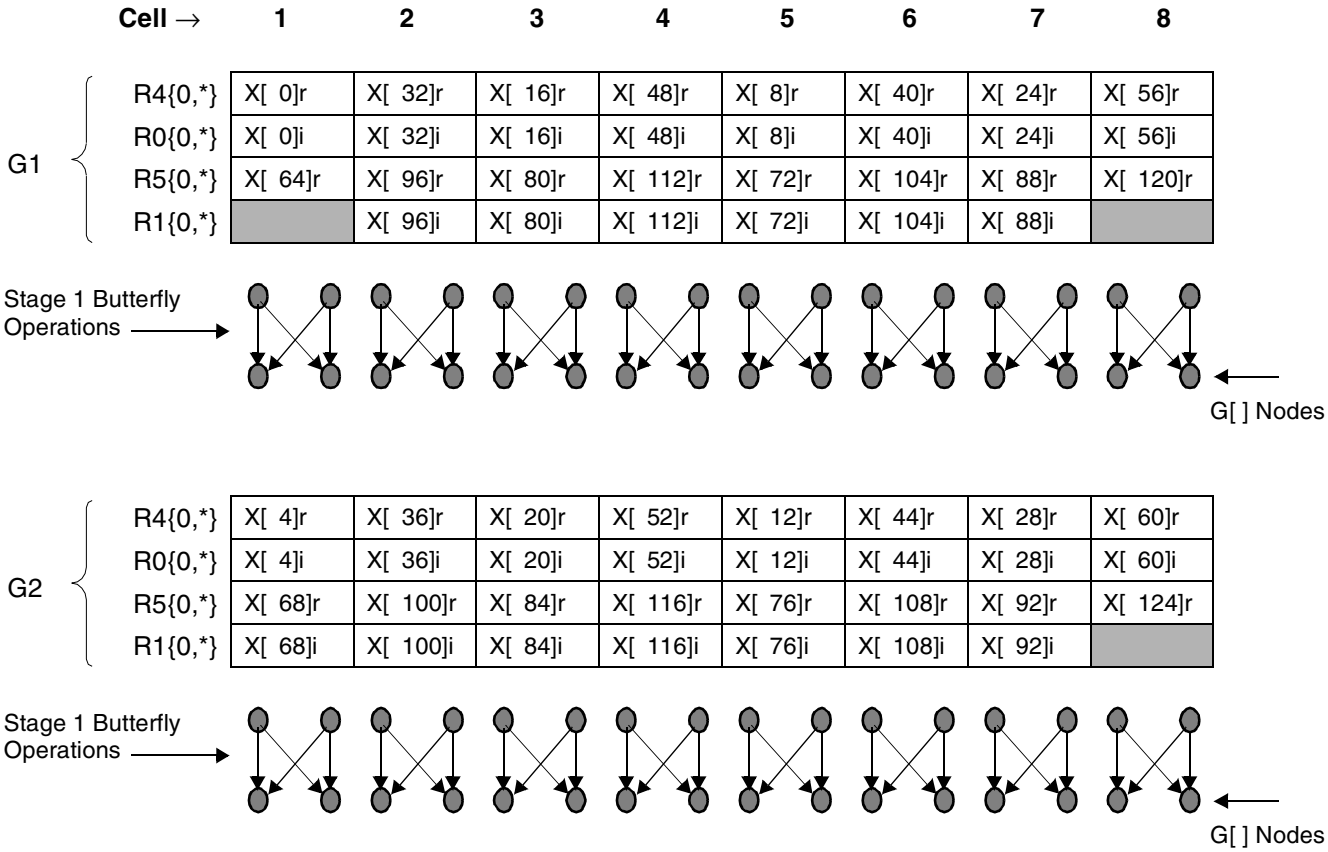


Figure 15. Sixteen Butterfly Operations in Parallel for the Group 1 and Group 2 Data

3.5.2 Data Regrouping

From stage 1 to stage 6, data regrouping is required at the end of each stage because of the increased butterfly spans. Data regrouping changes the original butterfly data flow as the intermediate nodes are regrouped. The change is not a problem if the first four stages of the butterfly operations are regrouped since butterflies are formed from within the groups. It is not a problem either when the group boundary is crossed to form the butterflies for stages 5 through 7 since all groups perform the same exchanges. **Figure 16** shows an example of data regrouping for stage 2 of Group 1 data. The output nodes are stored in registers R8, R9, R0, and R4. If no regrouping occurred for this data, the butterfly operations would have been performed across the cell boundary, which is not efficient on an RC array. The data samples used in a particular computation (in general) are better aligned column-wise for efficient computation. Data regrouping simplifies the next butterfly operation. Data regrouping for stage 2 swaps data items with neighboring cells to transform inter-cell butterflies to intra-cell butterflies.

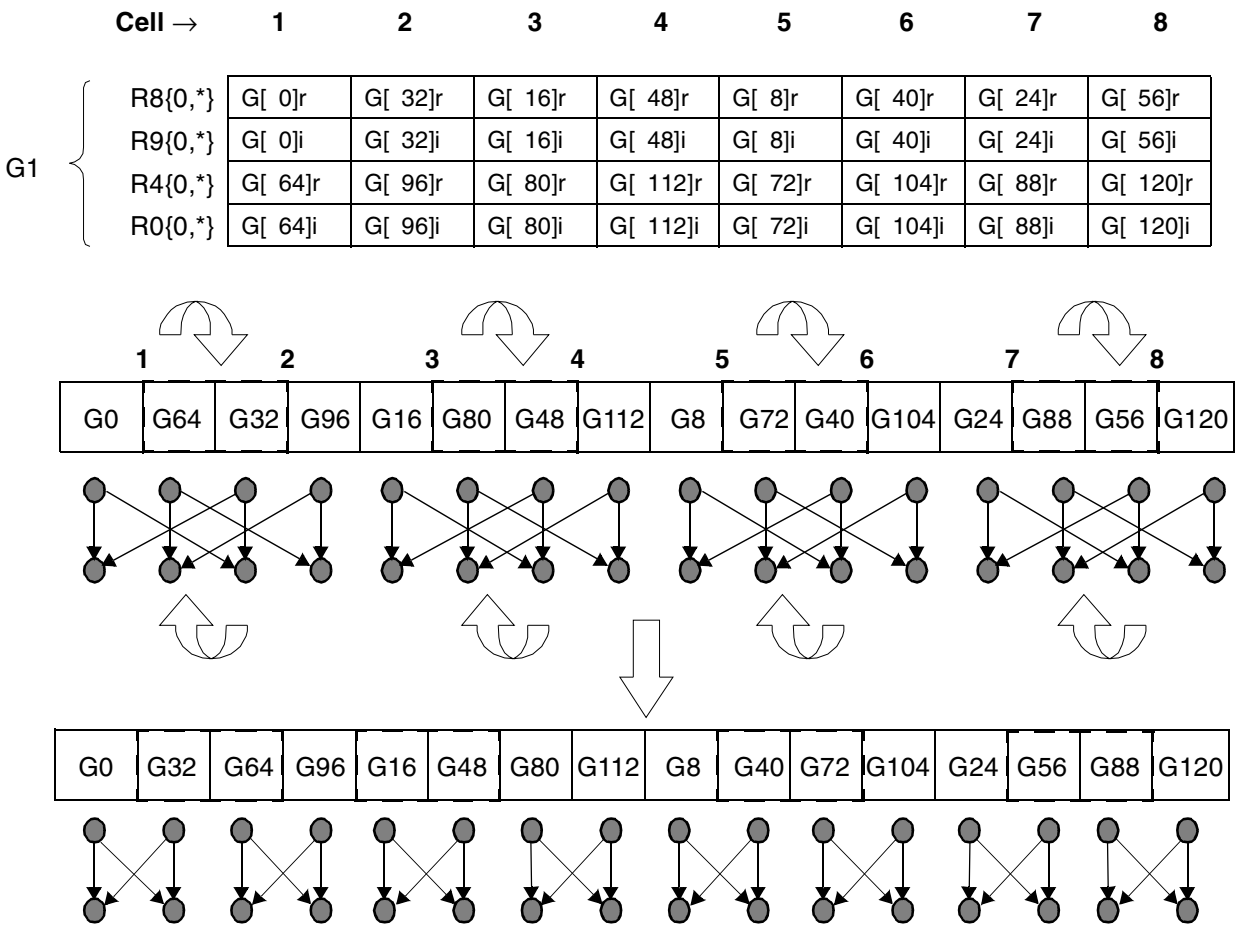


Figure 16. Data Regrouping for Stage 2 of Butterfly Operations for Group 1 Data

Figure 17 shows data regrouping for stage 3 of G1 data, which has a larger butterfly span than the previous stage. The data from cell 1 must be swapped with data in cell 3 and also for cells (2,4), (5,7), (6,8). These data exchanges convert the butterflies in row 2 of **Figure 17** into intra-cell butterflies in row 3 of the figure.

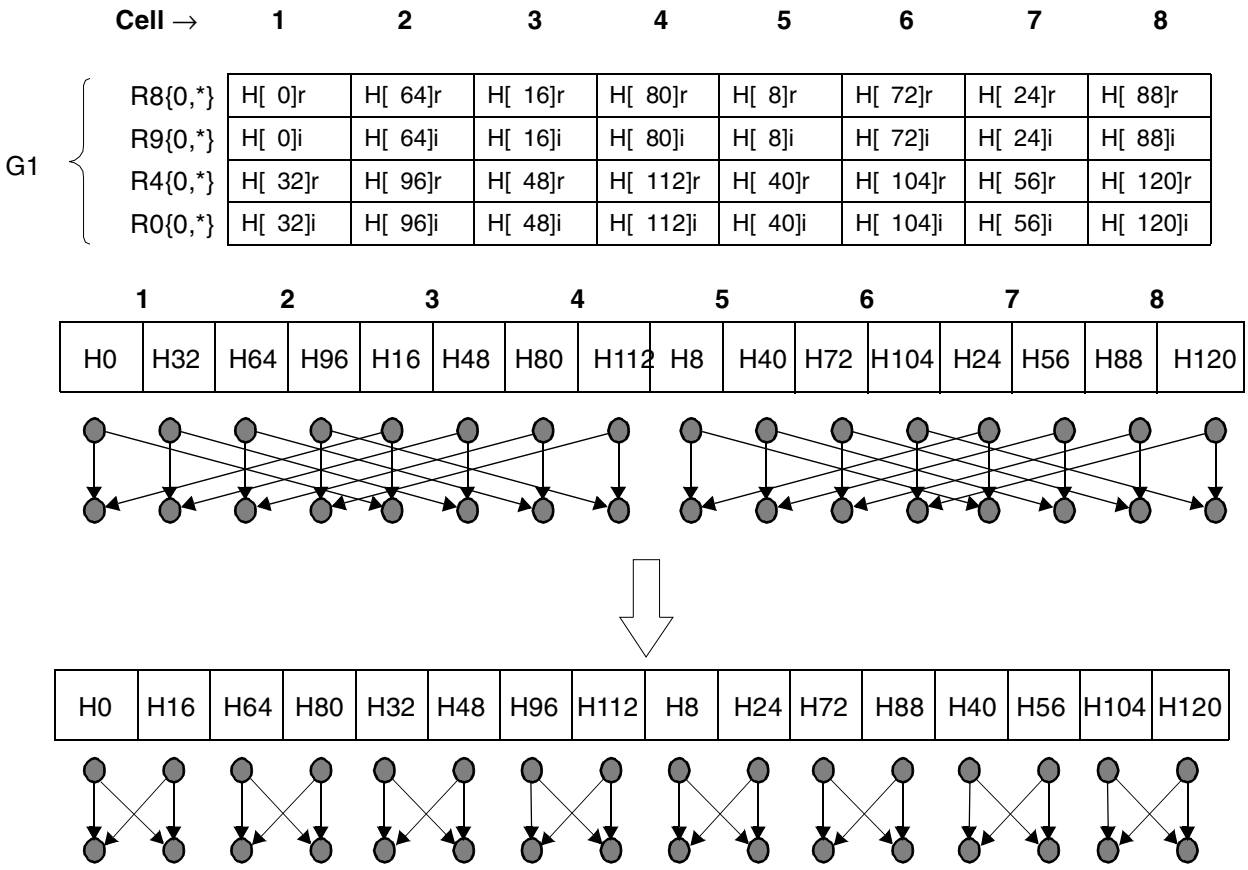


Figure 17. Data Regrouping for Stage 3 of Butterfly Operations for Group 1 Data

Figure 18 shows data regrouping for stage 4 of G1 data, with data exchanges among cell pairs (1,5), (2,6), (3,7), (4,8). A comparison of **Figure 16** through **Figure 18** demonstrates that data regrouping effectively shrinks the spans of the butterflies at each stage so that the parallel butterflies shown in **Figure 15** can continue for subsequent stages. At the end of stage 4, data regrouping causes an exchange of data between the two rows of RCs. At the end of stage 5, data regrouping causes an exchange of data by swapping registers so that Group 1 and Group 3 are paired to form butterflies. For the last stage, the register-resident groups (G5 to G8) must pair with the groups in the frame buffer (G1 to G4) to form butterflies. Continuously regrouping data causes the 16 parallel butterflies at each stage to be performed in virtually the same way within respective cells. **Example 4** shows a code segment for data regrouping in Group 1 data for stage 2. The data is regrouped in four RC cycles to prepare it for the next stage of FFT butterfly operations.

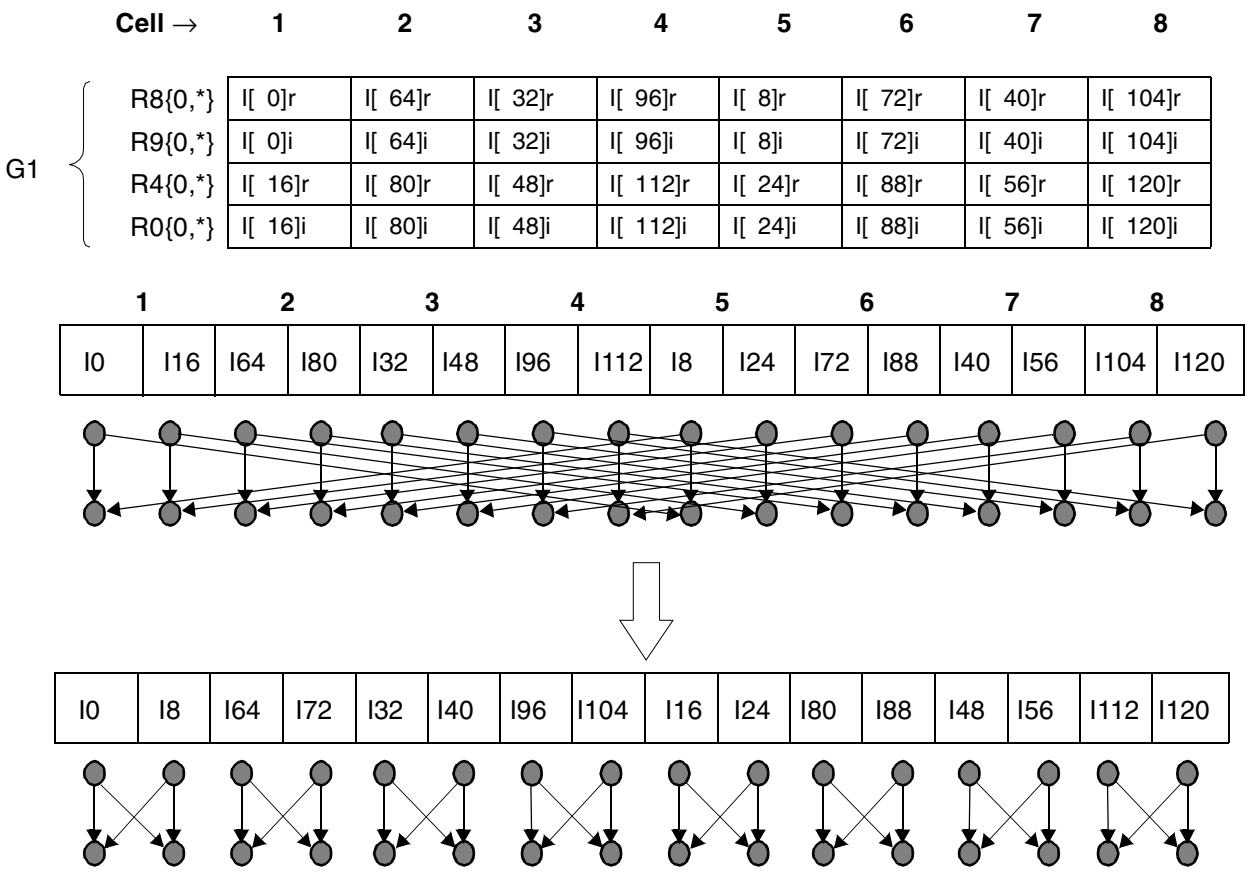


Figure 18. Data Regrouping for Stage 4 of Butterfly Operations for Group 1 Data

Example 4. Code Segment for Data Regroup of Group 1 Data

```
/* Result of G1,G2[Re,Im] = k1[R8,R9], k2[R4,R0] */
        /* data(G1,G2) regroup for stage 2 */
        MORPHO{
                CELL{*,0} R5 = BYP{R8{*,$+1}};
                CELL{*,2} R5 = BYP{R8{*,$+1}};
                CELL{*,4} R5 = BYP{R8{*,$+1}};
                CELL{*,6} R5 = BYP{R8{*,$+1}};
                CELL{*,1} R5 = BYP{R4};
                CELL{*,3} R5 = BYP{R4};
                CELL{*,5} R5 = BYP{R4};
                CELL{*,7} R5 = BYP{R4};
                }

        MORPHO{
                CELL{*,0} R1 = BYP{R9{*,$+1}};
                CELL{*,2} R1 = BYP{R9{*,$+1}};
                CELL{*,4} R1 = BYP{R9{*,$+1}};
                CELL{*,6} R1 = BYP{R9{*,$+1}};
                CELL{*,1} R4 = BYP{R4{*,$-1}};
                CELL{*,3} R4 = BYP{R4{*,$-1}};
                CELL{*,5} R4 = BYP{R4{*,$-1}};
                CELL{*,7} R4 = BYP{R4{*,$-1}};
                }
```

```
MORPHO{
        CELL{*,0} R4 = BYP{R8};
        CELL{*,2} R4 = BYP{R8};
        CELL{*,4} R4 = BYP{R8};
        CELL{*,6} R4 = BYP{R8};
        CELL{*,1} R1 = BYP{R0};
        CELL{*,3} R1 = BYP{R0};
        CELL{*,5} R1 = BYP{R0};
        CELL{*,7} R1 = BYP{R0};
        }
MORPHO{
        CELL{*,0} R0 = BYP{R9};
        CELL{*,2} R0 = BYP{R9};
        CELL{*,4} R0 = BYP{R9};
        CELL{*,6} R0 = BYP{R9};
        CELL{*,1} R0 = BYP{R0{*,$-1}};
        CELL{*,3} R0 = BYP{R0{*,$-1}};
        CELL{*,5} R0 = BYP{R0{*,$-1}};
        CELL{*,7} R0 = BYP{R0{*,$-1}};
        }
```

The data exchange operation for this group of data is shown in **Figure 19**. After data regrouping, intermediate data is transferred into the R4, R0, R5 and R1 registers, which are the same input registers for stage 1 butterflies. The stage 2 butterfly can then reuse the code from stage 1.

	Cell →	1	2	3	4	5	6	7	8
G1	R8{0,*}	G[0]r	G[32]r	G[16]r	G[48]r	G[8]r	G[40]r	G[24]r	G[56]r
	R9{0,*}	G[0]i	G[32]i	G[16]i	G[48]i	G[8]i	G[40]i	G[24]i	G[56]i
	R4{0,*}	G[64]r	G[96]r	G[80]r	G[112]r	G[72]r	G[104]r	G[88]r	G[120]r
	R0{0,*}	G[64]i	G[96]i	G[80]i	G[112]i	G[72]i	G[104]i	G[88]i	

G1	R4{0,*}	G[0]r	G[64]r	G[16]r	G[80]r	G[8]r	G[72]r	G[24]r	G[88]r
	R0{0,*}	G[0]i	G[64]i	G[16]i	G[80]i	G[8]i	G[72]i	G[24]i	G[88]i
	R5{0,*}	G[32]r	G[96]r	G[48]r	G[112]r	G[40]r	G[104]r	G[56]r	G[120]r
	R1{0,*}	G[32]i	G[96]i	G[48]i	G[112]i	G[40]i	G[104]i	G[56]i	G[120]i

Figure 19. Group 1 Data Regroup After Stage 1 Butterfly Operation

4 Fixed-Point and Precision Issues

For FFT on fixed-point processors such as the MRC6011 device, finite precision is limited by the number of bits available in the number representation and by the effects of finite arithmetic operations (truncation and rounding). It is important to understand that the magnitude of the complex FFT data changes through the various stages of calculation. **Figure 20** shows how two complex numbers at the input of the DIT butterfly combine to give two outputs. First, Vector B is multiplied by the twiddle factor. Since all twiddle factors in the FFT have the form $e^{-j\theta}$ which has unit magnitude, the magnitude of B´ is the same as that of B. Multiplication by the twiddle factor is nothing more than a rotation of the vector B over the angle θ. The vectors B´ and A are next added and subtracted to give the butterfly ouputs C and D, respectively.

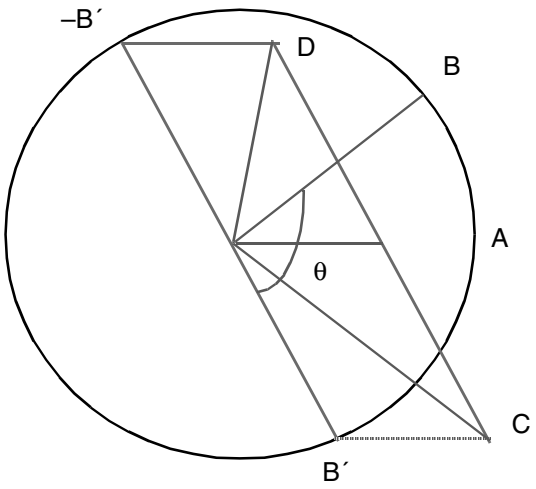


Figure 20. Vector Representation of the DIT Butterfly

Figure 21 shows that the largest magnitude occurs when the vectors B´ and A line up so that the magnitude of C (if B´ and A point in the same direction) or D (B´ and A point in opposite directions) is the sum of the magnitudes of A and B´.

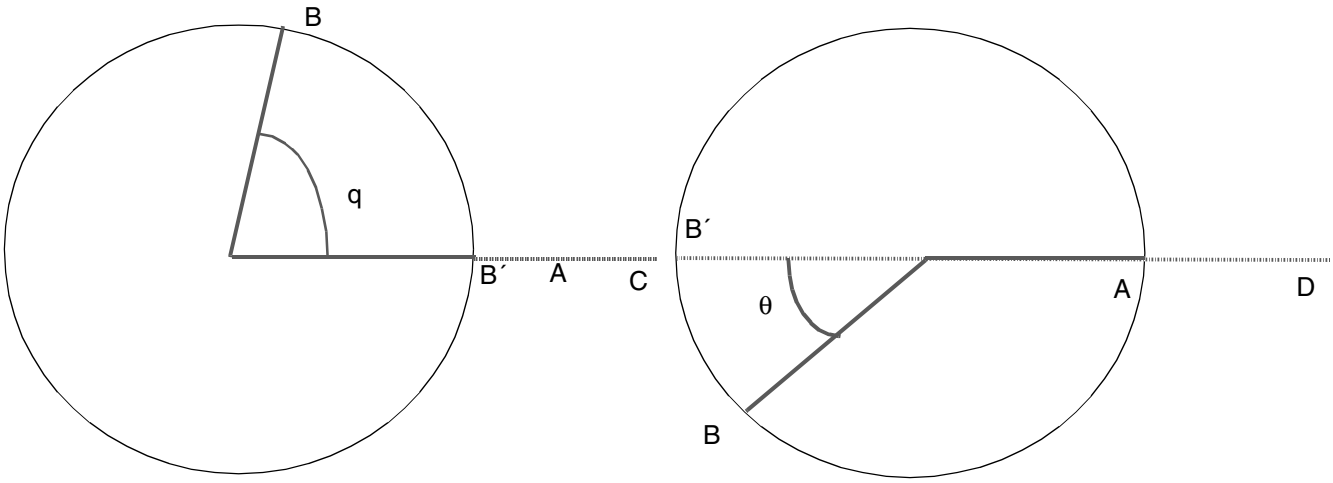


Figure 21. Bounds on Butterfly Output Magnitude

It should also be clear that either the magnitude of C or the magnitude of D is at least equal to the magnitude of the larger of the two vectors A and B. This leads to the relationship:

$$max(|A|, |B|) \leq max(|C|, |D|) \leq 2max(|A|, |B|) \quad \textbf{Equation 13}$$

We can see that the complex numbers at the output stages of the butterflies grow in magnitude from stage to stage. The maximum growth as given by **Equation 13** is a factor of two per stage or *one bit-per-pass.*

4.1 Input Data Analysis

The MRC6011 device is a fixed-point processor that uses 16 bits to represents integers $-1.0 \leq x < 1$. When an FFT is implemented on the MRC6011 device, the internal results of the FFT must not exceed these values at any one time. The previous discussion shows that the magnitude of the complex numbers can grow by a total factor of N or $\log_2 N$ bits in an N-point FFT. Because of the twiddle factor rotation, real and imaginary parts of the complex numbers can

at any time equal their magnitude (they become either purely real or purely imaginary). Therefore, the magnitude of the FFT input data must be limited by $M < 1.0/N$. For a real-input 128-point FFT, the absolute value of the input equals the magnitude and therefore must satisfy $|x(nT)| < \frac{1}{128}$.

We can achieve this bound by scaling the input data so that it resides in the lower 6 bits of the 16-bit word, yielding 10 guard bits to accommodate the worst case growth. For a complex input FFT, we must account for the imaginary part, and the bounds must not become as follows:

$$Real\ Part: -\frac{0.5}{128} \le R|x(nT)| \le \frac{0.5}{128}$$

$$Imaginary\ Part: -\frac{0.5}{128} \le I|x(nT)| \le \frac{0.5}{128}$$

This requirement forces the input data to be scaled so that it resides in the lower five bits of the input data. While scaling satisfies the requirements to avoid overflow, it has a detrimental effect in that precision is lost in the input data because the magnitude of the butterfly increases at every stage. However, this growth can be avoided by scaling the outputs of all of the FFT butterflies by a factor of 0.5 (one bit). Since this scaling is uniform through the FFT (because all points are treated equally), we must apply a common scale factor to the FFT results, as follows:

$$N = 2^{\log_2 N}$$

This scaling is now applied to **Equation 13** to obtain the bounds for the butterfly outputs with scaling:

$$\frac{max(|A|, |B|)}{2} \le max(|C|, |D|) \le max(|A|, |B|) \qquad \textbf{Equation 14}$$

With two's complement fixed-point notation, we ensure that the magnitude of the butterfly inputs is less than one at all times. We use 15 bits for the most possible precision with a signed two's complement fixed-point 16-bit number. The next section discusses the implementation details of this technique.

4.2 Butterfly Output Scaling

The scaling approach is illustrated in **Figure 22**. The two numbers added together at each stage of the butterfly flow are divided by 2 (right shift by 1 bit) to avoid immediate overflow. This approach is very effective and proven to eliminate any overflows. The final FFT outputs are reduced by *N* due to the consecutive right bit shift at each stage, where the *N* is the number of FFT points. In our implementation, the final FFT spectrum is reduced to 1/128 of the original Matlab-computed spectrum. Nonetheless, the shape of the spectrum is the same as that of the floating-point spectrum.

The butterfly output scaling approach is simple to implement on the RCF. The code segment in **Example 5** illustrates the simplicity of the implementation.

Example 5. Scale Down Approach Applied to Stage 1 Butterfly for Group 1 and 2 Data

```
/* Result of G1,G2[Re,Im] = k1[R8,R9], k2[R4,R0] */
/* Stage1 */
/* G1,G2[Re,Im] = k1[R4,R0], k2[R5,R1], tmp = R8,R9 */
CELL{*,*} R13 = MULSIH{FB{psiFBInputTwiddleCos, 0, OMEGA_BR2, COL_BUS, WORD},
                                                        R5, R14} << 1;

CELL{*,*} R12 = MULSIH{FB{psiFBInputTwiddleCos, 0, OMEGA_BR2, COL_BUS, WORD},
                                                        R1, R14} << 1;

CELL{*,*} R11 = MULSIH{FB{psiFBInputTwiddleSin, 0, OMEGA_BR2, COL_BUS, WORD},
                                                        R1, R14} << 1;

CELL{*,*} R10 = MULSIH{FB{psiFBInputTwiddleSin, 0, OMEGA_BR2, COL_BUS, WORD},
                                                     R5, R14} << 1;
CELL{*,*} NOP{};
CELL{*,*} R13 = ADD{R13, R11} >> 1; // scale down;
CELL{*,*} R12 = SUB{R12, R10} >> 1; // scale down;
CELL{*,*} R11 = MULL{R15,R4} >> 1;// scale down for adjustment;
CELL{*,*} R10 = MULL{R15,R0} >> 1;// scale down for adjustment;
CELL{*,*} NOP{};
CELL{*,*} R8 = ADD{R11, R13};
CELL{*,*} R9 = ADD{R10, R12};
CELL{*,*} R4 = MULSIL{R15,R4,R8}; // scale down
CELL{*,*} R0 = MULSIL{R15,R0,R9}; // scale down
```

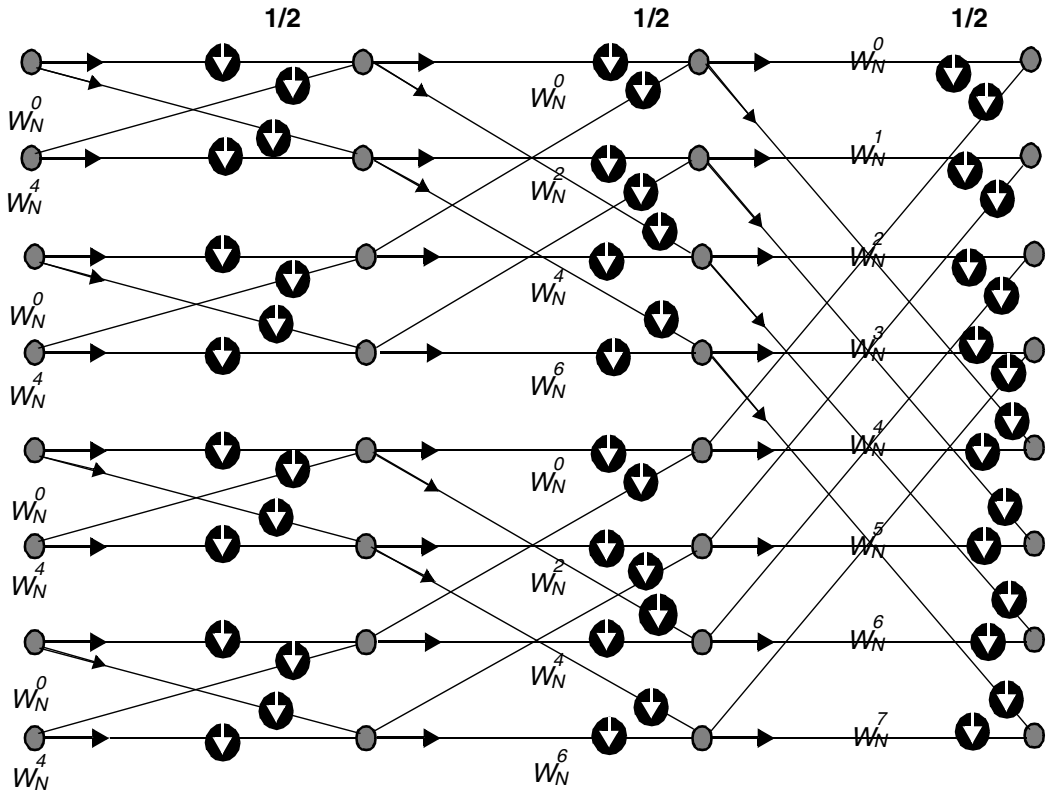


Figure 22. Scale Down By Two Applied to an 8-Point FFT

The scaling step incurs three more cycles at each stage than are shown in **Example 3**. To implement scale down, the code in **Example 3** is changed as follows:

- The right 1 bit shift at the end of the R13, R12 addition.
- Two additional multiplication cycles to scale down R4 and R0 for adjustment.
- One NOP cycle as a result of the MAC delay introduced by the multiplication preceding it.
- Instead of holding the constant of 0x2, the R15 register retains the constant of 0x1 throughout the butterfly stages due to the scale down operation.

4.3 Rounding on RC

Although rounding on RCF is not automatic, it is achieved by adding 0x00008000 to or subtracting 0xFFFF8000 from the number to be rounded. In **Figure 23**, the truncation and rounding operations are directly compared. The net effect of the truncation introduces bias or error in the final result. However, proper rounding generally yields an unbiased result or a near zero bias in the final result. An apparent cost of rounding is the need for locations to hold the rounding constant as well as the intentional multiplication with a constant (one) and add operation if no multiplication is necessary in the original operations.

```
CELL{*,*} R14 = INV{ZERO} << 15;          /* move 0x8000 into R14 */
CELL{*,*} R2 = MULSIH{R0,R1,R14} << 1;    /* multiply R0,R1 and round */
```

$$R2 = MAC\_REG\ [\ (\ (R0 \times R1)\ <<\ 1) - sign\ extend\ (R14)\ ]\ _{[31\ \ldots\ 16]}$$
$$MAC\_REG\ [\ (\ (R0 \times R1)\ <<\ 1) - 0xFFFF8000\ ]\ _{[31\ \ldots\ 16]}$$
$$MAC\_REG\ [\ (\ (R0 \times R1)\ <<\ 1) + 0x00008000\ ]\ _{[31\ \ldots\ 16]}$$

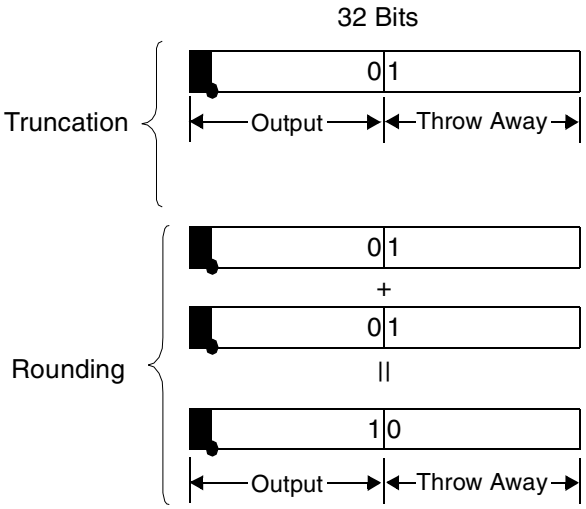


Figure 23. Comparison of Truncation and Rounding Operations on RCF

5 Performance and Error Analysis

This section presents performance data of the implementation, including RCF cycles counts, frame buffer memory requirements, context memory usage, and averaged errors of the FFT spectra. We used both white noise and sine wave as test vectors to test the implementation. The errors associated with each test vector are stable and acceptable with improved precision. With 8-bit inputs, the errors would have been much larger.

Table 4 lists the cycle counts for different runs of the FFT. FFT execution time on RCF is calculated as 3659cyc × 4ns = 14.636 µs, 684 cyc × 4 ns = 2.736 µs at 250 MHz RCF core frequency.

Table 4. RCF Cycle Counts for Separate Runs of the 128-Point FFT

Kernel Name	Cycles First Run	Cycles Second Run
FFT128_Transpose	634	139
FFT128_Stage1_6	1941	297
FFT128_Stage7	1084	248
TOTAL	**3659**	**684**

Table 5 lists the frame buffer memory requirements of the code. The RCF frame buffer size (40 KB) is sufficient to meet the requirements.

Table 5. Memory Requirements for the 128-Point FFT Implementation on RCF

Variable Description	Variable Name	Bytes Required
Input data	psiInputData	512
Twiddle factor	psiInputTwiddle	544
Temporary area for intermediate result	psiOutTmp	256
Output data	psiOutput	512
TOTAL		**1824 bytes**

Table 6 shows how the context memory in the RCF core is used in the FFT implementation. The FFT128 computation is divided into three kernels to avoid limiting the number of contexts (contexts of a kernel cannot exceed 64). However, all contexts of the kernels can be stored in context memory, which has 128 context memory locations for row and column contexts.

Table 6. Context Memory Usage for the 128-Point FFT on RCF

Kernel Name	Number of Row Contexts	Number of Column Contexts
FFT128_Transpose	39	16
FFT128_Stage1_6	58	32
FFT128_Stage7	12	2
TOTAL	**109**	**50**

Four 128-point white noise random number sequences were used to generate the averaged errors of the FFT spectra. The input numbers were clipped to the range of (–32768, +32767) and the errors were computed as follows:

$$Error(i) = sqrt([Y_{re}^{flt}(i) - 128 * Y_{re}^{fix}(i)]^2 + [Y_{im}^{flt}(i) - 128 * Y_{im}^{fix}(i)]^2) \quad \textbf{Equation 15}$$

Where *flt* indicates a MATLAB® floating-point FFT result and *fix* indicates an RCF result. The multiplier of 128 is needed because of the scale down by 2 operation at each stage of the butterfly operation. **Figure 24** shows the point-wise error magnitude for 128 points across the frequency scale for the white noise input.

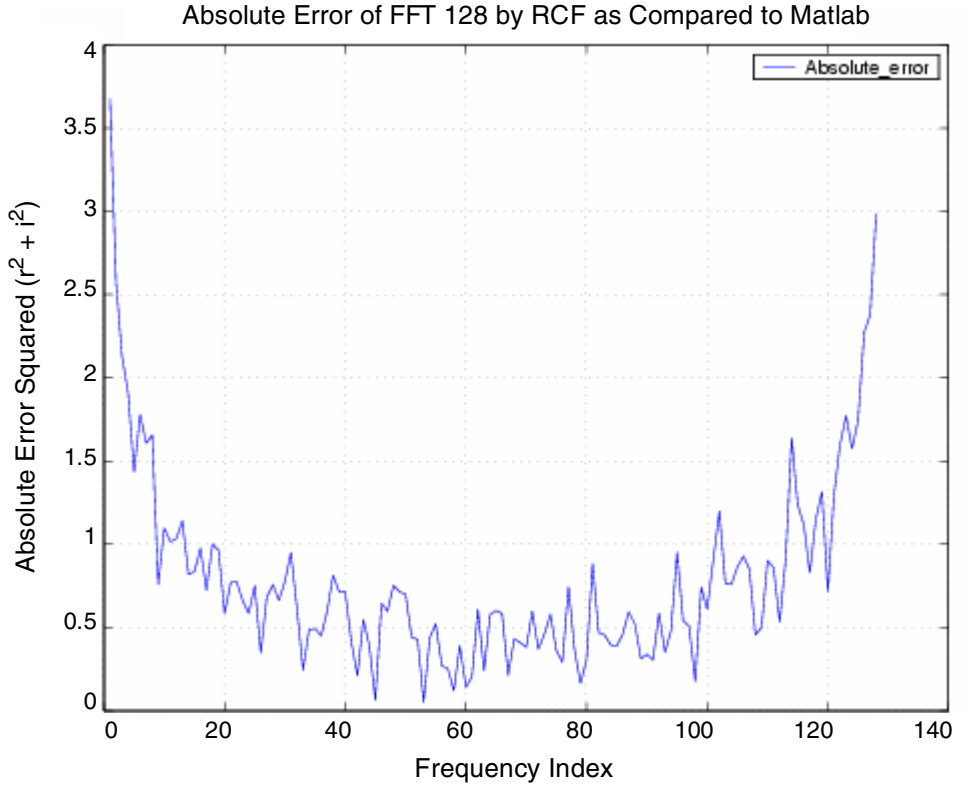


Figure 24. Absolute Error Performance of the 128-Point FFT with White Noise Input

The 128-point FFT implementation was also tested with a sine wave of 0.1 Hz normalized frequency. The input range of the sine wave was set to (–16384,+16383). **Figure 25** shows how the original sine wave and the error performance are plotted. The absolute errors across the frequency scale for the sine wave input are on the same magnitude as the white noise errors. The vertical scale must be multiplied by 128. The shape of the error curves indicates that the low-frequency components contain larger errors than the higher-frequency components.

6 Summary

This application note describes the implementation of a 128-point DIT FFT algorithm on the Freescale MRC6011 RCF device. Because of the limited number of registers, the first half of the 128 input data is independent of the second half of the data, until stage 7. The 128-point FFT butterfly flow is divided into three parts. Between the stages, data regrouping is performed to facilitate the butterfly operations of the next stage. Data regrouping is performed on RCF via register content swaps. The operation effectively converts inter-cell butterflies into intra-cell butterflies so that the parallel RC array is used more efficiently in the FFT computation. Results from different input test vectors have shown the effectiveness of these economic approaches.

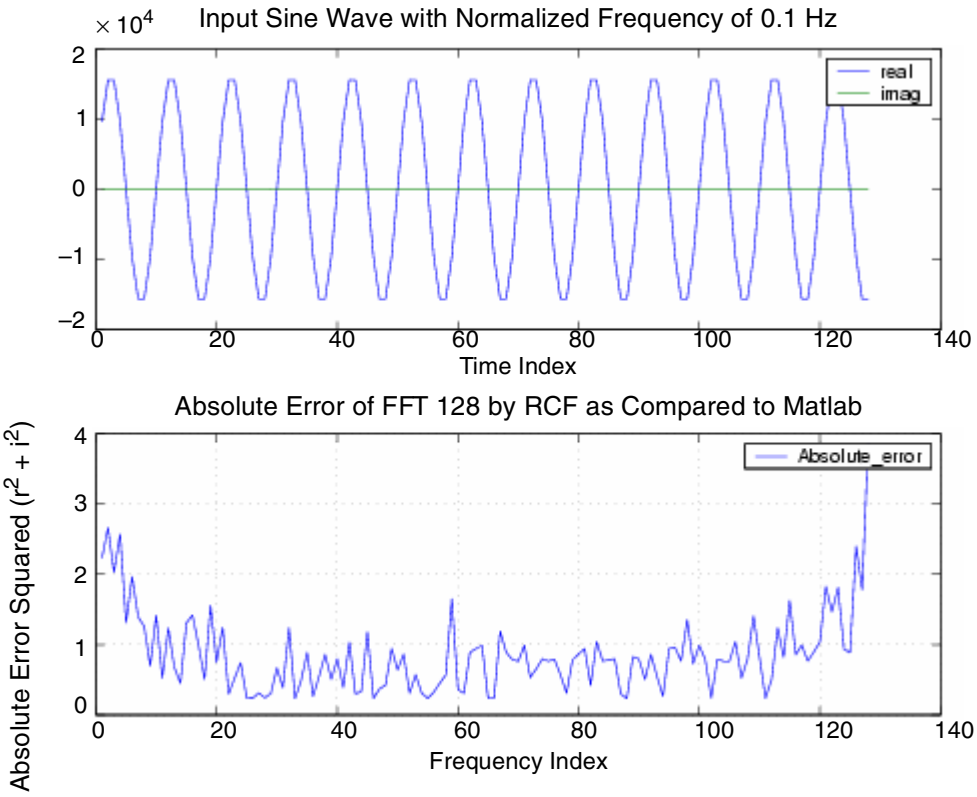


Figure 25. Input Sine Wave and Absolute Error Comparison

7 References

[1] *MRC6011 Reference Manual* (MRC6011RM), Rev. 0, Freescale Semiconductor, 2004.

[2] A.V. Oppenheim and R W. Schafer, *Digital Signal Processing*. Englewood Cliffs, N.J.: Prentice Hall, 1975.

[3] E. Oran Brigham, *The Fast Fourier Transform*, Englewood Cliffs, N.J.: Prentice Hall, 1974.

NOTES:

How to Reach Us:

USA/Europe/Locations not listed:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

Japan:
Freescale Semiconductor Japan Ltd.
Technical Information Center
3-20-1, Minami-Azabu. Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

Learn More:
For more information about Freescale Semiconductor products, please visit
http://www.freescale.com

AN2768
Rev. 0
6/2004