Freescale Semiconductor
Application Note

Document Number: AN2115
Rev. 4, 04/2013

Complex Floating Point Fast Fourier Transform

This document compares the performance of fast Fourier transform (FFT) with and without AltiVec™ technology to demonstrate how mathematically-intensive code can be adapted for use with AltiVec and how AltiVec increases code performance. AltiVec is supported on PowerPC™ microprocessors such as the MPC74XX, MC86XX, T4XXX, and B4XXX.

To locate published updates for this document, see the website listed on the last page of this document.

Contents

1 Overview

Fourier transforms convert a signal to and from the frequency domain. Just as a glass prism may display the spectrum of an incoming light wave, Fourier transforms break a signal down into its frequency components. This process involves writing a signal as a summation of sines and cosines. Equation 1 shows a typical algorithm used for this purpose, the discrete Fourier transform (DFT), and Equation 2 calculates the DFT of a discrete signal using a set of symmetric points around a unit circle, W_N.

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

Equation 1. DFT Equation

where

```
x[n] = discrete-time signal
X[k] = frequency domain components
N = Number of Points
k = 0,1,2,..., N-1
W_N is a multiplicand factor and is shown in Equation 2
```

$$W_N[n] = e^{-j\left(\frac{2\Pi}{N}\right)}$$

Equation 2. W_N Value

The DFT is of the order $O(N^2)$ for a signal of length N (an algorithm is said to have an order of N, O(N), if it computes in a scalar multiple of N iterations). The fast Fourier transform (FFT) reduces the number of calculations of the DFT by dividing the initial function into repeated subfunctions and continues this process until the subfunction is no longer divisible. For a detailed description of the derivation of the FFT formula and the various types of FFTs available, see *Inside The FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*, by Eleanor Chu and Alan George.

Equation 3 shows the decimation of the DFT algorithm from Equation 1.

$$X[k] = \left(\sum_{n=0}^{\frac{N}{2}-1} \left(x[n] + x\left[n + \frac{N}{2}\right] \right) \cdot W_{\frac{N}{2}}^{kn} \right) + \left(\sum_{n=0}^{\frac{N}{2}-1} \left(\left(x[n] - x\left[n + \frac{N}{2}\right] \right) W_N^n \right) \cdot W_{\frac{N}{2}}^{kn} \right)$$

Equation 3. DFT Subdivided into the Radix-2 Decimation in Frequency (DIF)

In Equation 3, the Radix-2 Decimation in Frequency (DIF) FFT divides the DFT problem into two subproblems, each of which equals half the original sum. Note that, in this example, the FFT is a DIF because it decimates the frequency components (X[k]) of the DFT problem. In comparison, if the FFT is a DIT, it decimates the time components (x[n]).

Note that the Radix-2 DIF FFT is not the fastest algorithm. If faster algorithms are desired, check on a higher radix algorithm, such as Radix-4, which divides the equation into four subproblems.

2 Signal Flow Graph for Scalar and Vector FFTs

This section details two functions that perform the FFT. The first performs a single-precision, complex Radix-2 DIF FFT using PowerPC instructions. The second function performs the same transform using the AltiVec instructions provided to work with the PowerPC instructions. These two functions are written as similarly as possible to provide the best comparison.

When coding FFTs, the Signal Flow Graph (SFG) of the equations resemble a butterfly. The butterfly equivalent to Equation 3 is shown in Figure 1. Note how the original x[n] values can be replaced to constitute the new sets of coefficients.

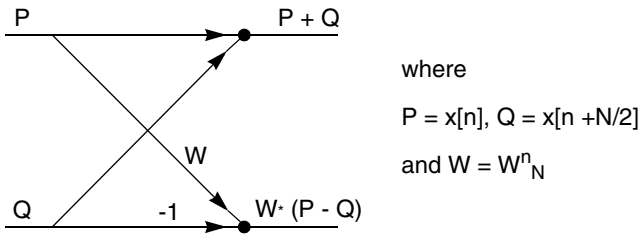

Figure 1. Single Butterfly Representation of a Signal Flow Graph

Figure 2 shows an SFG for a data set of size **N** = 8. Inputs are on the left and calculations flow from left to right. This DIF FFT takes the input signal in order and produces the output in bit-reversed order. In bit-reversed order, each output index is represented as a binary number and the indices' bits are reversed. For example, in an eight-point DIF FFT, sequential order of the indices' bits is 000, 001, 010, 011, and so on. Reversing these bits yields 000, 100, 010, 110, and so on. This sequence corresponds to 0, 4, 2, 6, and so on in decimal notation, which is the output order shown in Figure 2.

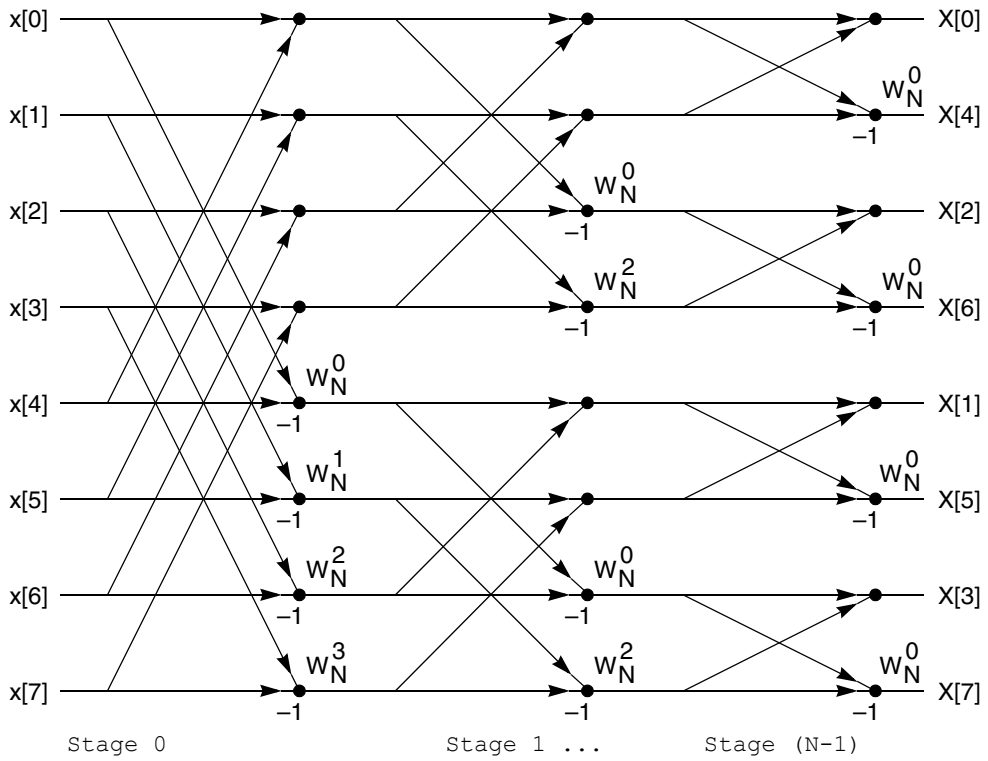

Figure 2. Radix-2, DIF, FFT Computational Lattice Structure for N = 8

Each butterfly involves one complex addition and one complex subtraction followed by a complex multiply (the value W involved in the complex multiply is commonly called a twiddle value). One advantage of the butterfly structure is that result values can safely overwrite the input values in memory. This allows the Radix-2 FFT to be complete as an in-place computation. A single iteration of in-place computation forms a stage. As Figure 2 shows, within a stage, there are **N**/2 butterflies. There are $N*\log_2(N)$ stages; therefore, the Radix-2 FFT is an $O(N*\log_2(N))$ operation.

The illustrated implementation uses three nested “for” loops. The outer loop iterates over the stages. The middle loop iterates over blocks (where butterflies intersect). As Figure 2 shows, Stage 0 has one block, Stage 1 has two blocks, Stage 2 has four blocks, and so on. Finally, the innermost loop iterates over individual butterflies in a block.

Since the element size is a power of two, there are $\log_2 N$ stages. The size information is described as **P** where:

$N = 2^P$ or more easily calculated as N = (1 << P).

2.1 Outer Loop

2.1.1 Scalar

The outer loop iterates **P** times, once for each stage. For AltiVec code, however, the last two stages are done separately because, unlike prior stages, the two last stages have smaller blocks and both butterfly points end up on the same vector. Furthermore, the first stage in the vector code is done separately because

the twiddle values, which reside in the vectors as pairs, are consecutive and only two of the vectors need to be loaded. To be consistent with the vector code, the scalar code's last two stages are done separately. This ensures an accurate comparison. The first stage, however, is still processed in the three-nested loops.

The code below illustrates the scalar outer loop.

Outer Scalar Loop

```
for ( stage=0; stage < (P-2); stage++ )
{
/* Calculations for each block go here */
}
```

2.1.2 Vector

The stage loop variable for the vector code starts from 1 because stage 0 is handled separately. The code below shows the vector equivalent of the code shown above.

Outer Vector Loop

```
for ( stage=0; stage < (P-2); stage++ )
{
/* Calculations for each block go here */
}
```

2.2 Inner Loop

2.2.1 Scalar

Within a stage, two values must be calculated: the block offset and the stride. The block offset in the stage block always points at the top of the current butterfly. The stride is the distance between the top of a butterfly and its bottom. Incrementing to the next block within a stage requires knowing the size of a block. This is a simple calculation: block size is always twice the value of stride. This relationship is used in the "for" loop to increment through the blocks within a stage. Because the block pointer should never point outside the dataset, block<N is the stopping condition for the loop. Stride is initialized to N/2. The code below includes the second inner loop.

Inner Scalar Loop

```
for( stage=0; stage < (P-2); stage++ )
{
        for ( block=0; block<N; block += stride*2 )
        {
                /* work on the butterflies here */
        }
}
```

2.2.2 Vector

In the vector code the block variable points at the top of a block. However, the data is stored as two floating-point complex numbers per vector and whenever block++ is performed the variable steps over two sets of data. Therefore, the stopping condition has to be block<N/2. For similar reasons,

the increment value is now stride instead of stride*2. Stride is initialized to N/4 in this case. This code is displayed below.

Inner Vector Loop

```
for ( stage=0; stage < (P-2); stage++ )
{
         for ( block=0; block<N/2; block+= stride )
         {
                  /* work on the butterflies here */
         }
}
```

2.3 Innermost Loop

2.3.1 Scalar

The innermost loop calculates the butterflies. Two butterflies (vertically) are processed because symmetry in the unit circle allows twiddle values loaded for one butterfly to be used again, with some manipulation, for calculation in another butterfly. The number of butterflies in a stage is always equal to the stride. Because two butterflies are processed in a loop, the stopping condition is stride/2. Two pairs of variables are needed to point at the two butterflies and these pairs are a stride amount apart. One pair of variables, pa and pb, point at the top and bottom of one butterfly, while variables qa and qb point to the top and bottom of the second butterfly. The code below demonstrates the complete FFT scalar loops.

Complete FFT Scalar Loops

```
          for( stage=0; stage < (P-2); stage++ )
          {
                   for( block=0; block<N; block +=
stride*2 )
                   {
                            pa = block;
                            pb = block + stride/2;
                            qa = block + stride;
                            qb = blockblock + stride +
stride/2;
                            for( j = 0; j < stride/2;
j++)
                            {
                             /* work on two butterflies
here */
```

2.3.2 Vector

The vector code is similar to the scalar code, except that all strides are divided by two because there are two complex values per vector. The processing of two vectors within the inner loop corresponds to the processing of four butterflies within the loop. The code below shows the equivalent.

Complete FFT Vector Loops

```
for( stage=1; stage < (P-2); stage++ )
{
         for( block=0; block<N/2; block += stride )
         {
                  pa = block;
                  pb = block + stride/4;
                  qa = block + stride/2;
                  qb = block + stride/2 + stride/4;
                  for( j = 0; j < stride/4; j++)
                  {
                   /* work on four butterflies here */
                  }
         }
}
```

2.3.3 Scalar (detail)

The butterfly calculation is processed in the innermost loop. As mentioned above, pa, pb, qa, and qb are used to point at the tops and bottoms of the two butterflies. pfs is the complex element array of data. Initially, the values are inputs; but, because operated on in-place, they store the outputs as well. The twiddle values are pre-calculated and stored in pfw. As Figure 2 shows, the twiddle values for the first stage are accessed from consecutive places. For the second stage, the twiddle

values are accessed as every other value from a table and so forth. Shown below is the corresponding scalar code.

Code Scalar Calculations

```
/* complex add */
ft1a.re = pfs[pa+j].re + pfs[qa+j].re;
ft1a.im = pfs[pa+j].im + pfs[qa+j].im;
ft1b.re = pfs[pb+j].re + pfs[qb+j].re;
ft1b.im = pfs[pb+j].im + pfs[qb+j].im;

/* complex sub */
ft2a.re = pfs[pa+j].re - pfs[qa+j].re;
ft2a.im = pfs[pa+j].im - pfs[qa+j].im;
ft2b.re = pfs[pb+j].re - pfs[qb+j].re;
ft2b.im = pfs[pb+j].im - pfs[qb+j].im;
pfs[pa+j] = ft1a;       /* store adds */
pfs[pb+j] = ft1b;

/* complex multiply */
pfs[qa+j].re = ft2a.re * pfw[iw].re -
                  ft2a.im * pfw[iw].im;
pfs[qa+j].im = ft2a.re * pfw[iw].im +
                  ft2a.im * pfw[iw].re;

/* twiddled complex multiply */
pfs[qb+j].re = ft2b.re * pfw[iw].im +
                  ft2b.im * pfw[iw].re;
pfs[qb+j].im = -ft2b.re * pfw[iw].re +
                  ft2b.im * pfw[iw].im;
iw += edirts;
```

2.3.4 Vector (detail)

As mentioned earlier, the vector innermost loop processes four butterflies at a time. By doing more calculations within the inner loop, expensive memory transactions are minimized. Because complex numbers are represented as interleavings of real and imaginary values, the vec_perm instruction is used to extract the twiddle values as well as the data. A vector logic instruction, which involves using xor with a

-0.0, is used in a single-cycle negation of the floating point values. Below is the vector code for the innermost loop.

Core Vector Calculations

```
/* PREP THE TWIDDLES */
/* transpose-conjugate and its negate */
vtfw0 = vec_perm(
        pw[j*2*edirts],
        pw[j*2*edirts+edirts],
        vcaraibrbi );
vtfw1 = vec_perm(
        pw[j*2*edirts],
        pw[j*2*edirts+edirts],
        vcaiarbibr );
vtfw2 = vec_xor( vtfw1, vcfnegeven );
vtfw3 = vec_xor( vtfw1, vcfnegodd  );

/* FOUR BUTTERFLIES */
/* complex add and subtract */
vtf10 = vec_add( pvf[pa+j], pvf[qa+j] );
vtf11 = vec_sub( pvf[pa+j], pvf[qa+j] );
pvf[pa+j]= vtf10;

vtf20 = vec_add( pvf[pb+j], pvf[qb+j] );
vtf21 = vec_sub( pvf[pb+j], pvf[qb+j] );
pvf[pb+j] = vtf20;

/* complex multiply (apply twiddle) */
vtf12 = vec_perm( vtf11, vtf11, vcprm0022 );
vtf12 = vec_madd( vtf12, vtfw0, vcfzero );
vtf13 = vec_perm( vtf11, vtf11, vcprm1133 );
pvf[qa+j] = vec_madd( vtfw2, vtf13, vtf12 );

vtf22 = vec_perm( vtf21, vtf21, vcprm1133 );
vtf22 = vec_madd( vtf22, vtfw0, vcfzero );
vtf23 = vec_perm( vtf21, vtf21, vcprm0022 );
pvf[qa+j] = vec_madd( vtfw3, vtf23, vtf22 );
```

As mentioned earlier, three stages are removed from the iterations over the SGF to gain performance. The first stage is removed because different vector permute constants must be used due to the twiddles being adjacent in memory. The last two stages are removed because the final twiddle values (1,0) and (0,-1) do not require a complex multiply. Instead, vector-permute and vector-negate instructions are used.

In the source code the loop for the final two stages is manually unrolled so that 16 butterflies are calculated in one iteration. This number of butterflies is found to be optimal. Fewer butterflies results in more overhead due to branching. More butterflies results in a shortage of vector registers.

After all iterations are done, the FFT of the original signal is found in the same memory location as the original signal. The element at index zero of the result is the DC component of the signal. Lower index values correspond to lower frequency bins of the FFT after bit reversal.

3 Fast Fourier Transform: Example 1

Table 1 shows the above concepts.

Table 1. Values for x[n]

n	Real	Imag
0	2.1	0.0
1	3.0	2.1
2	1.3	2.1
3	4.2	3.4
4	0.9	2.1
5	3.2	0.1
6	1.0	1.1
7	2.3	0.2

Note: Where N = 8

First the twiddle values are calculated using Equation 2. For a DFT calculation, eight twiddle values across the unit circle are needed. For the FFT discussed in this paper, symmetry is used to reduce these numbers to two values (see Figure 3).

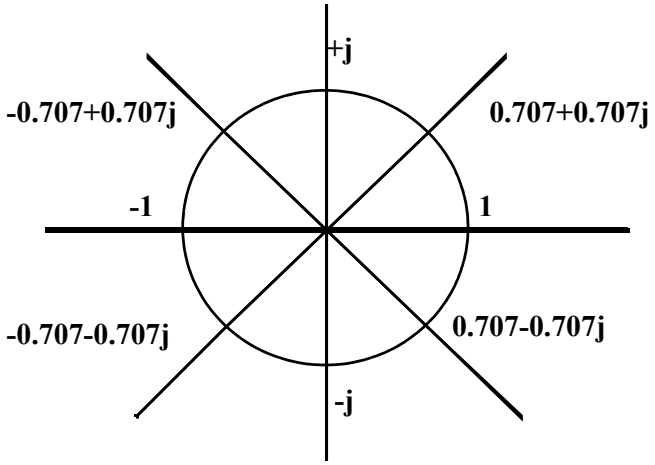

Figure 3. DFT Twiddle Values for N = 8

The two selected values {(1,0), (0.707, -0.707)} are W_0 and W_1in Equation 2. Each W_n starts from an angle position of 0 and increases in the negative direction. Therefore, W_2 and W_3 (the only ones needed for N = 8 FFT—see Figure 2) can be determined from the following:

- Real(W_0) = -Imaginary(W_2)
- Real(W_1) = Imaginary(W_3)
- Imaginary(W_1) = - Real(W_3)

Using these twiddle values, the FFT is calculated. Table 2 shows the result.

Table 2. Values for x[n] and X[k]

	INPUT			OUTPUT	
n	Real	Imag	K	Real	Imag
0	2.1	0.0	0	18	11.1
1	3.0	2.1	1	-7.4	-0.5
2	1.3	2.1	2	-0.7	-0.8
3	4.2	3.4	3	2.1	-1.4
4	0.9	2.1	4	4.39	-4.45
5	3.2	0.1	5	0.01	-0.35
6	1.0	1.1	6	5.6	-2.15
7	2.3	0.2	7	-4.96	-1.44

4 Fast Fourier Transform: Example 2

For larger data sets, the input and output data are best described in plots. Such an example is demonstrated next. For this example Equation 4 represents the signal considered.

$$x(t) = 5\sin(2\Pi x 2t) + \sin(2\Pi x 20t)$$

Equation 4. Signal with Two Frequency Components

To change the above into a discrete signal, one needs to sample it at a certain frequency. For example, if the signal is sampled at 256 samples per second, the discrete signal obtained is as shown in Equation 5. This is obtained by noting that discrete time variable *n* is equivalent to time domain variable *t* multiplied by the sampling frequency. Figure 4 shows the plot of the signal.

$$x[n] = 5\sin\left(\frac{2\Pi x 2n}{256}\right) + \sin\left(\frac{2\Pi x 20n}{256}\right)$$

Equation 5. Discrete Signal with Two Frequency Components

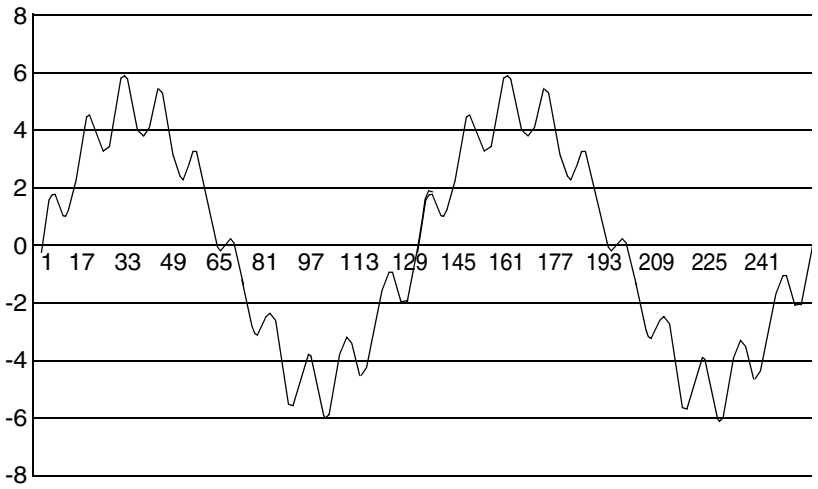

Figure 4. Plot of Equation 5

The sinusoid of greater magnitude in the figure corresponds to the first sine term in Equation 5. The nested sinusoids correspond to the second sine term in Equation 5. When this signal is transformed through the FFT function, the two frequency components are isolated. The output plot (after correcting for address bit reversal) is shown in Figure 5.

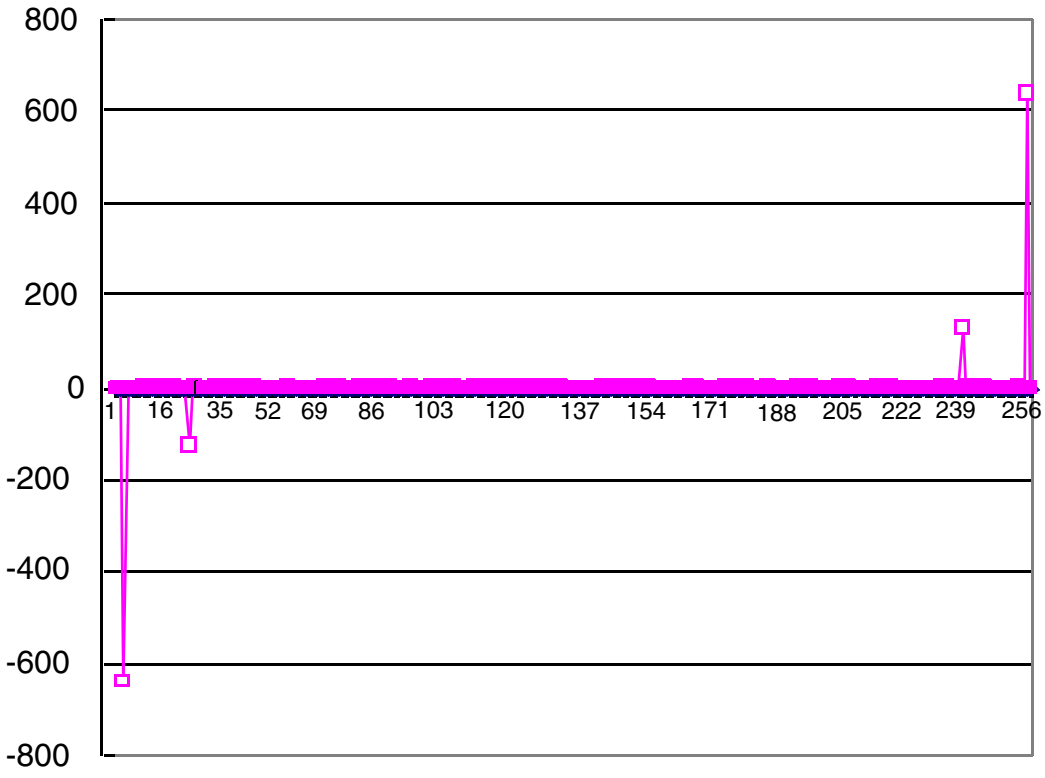

Figure 5. FFT Output for the Signal in Figure 4

The even part of the input signal corresponds to the imaginary part of the output. The odd part of the input signal correspond to real output. Because the input was entirely composed of sine waves (which are pure odd signals), the output is entirely composed of imaginary components. According to the Fourier transform properties (Oppenheim and Schafer, p52), the output of a real signal has an odd-symmetric imaginary component. Figure 5 confirms this property. The spikes at positions 2 and 254 correspond to the first term of the input signal. The spikes at positions 20 and 246 correspond to the second term in the input signal.

Equation 6 shows the mathematically-derived solution. This equation represents the first two spikes in Figure 5 and their reflections on the y-axis. In the equation, δ is a signal input to extract the system's impulse response. To read more about FFT functions and other signal processing techniques and terms, see Section 7, "References."

$$X_K = -\frac{5}{2}j[256\delta[K-2] - 256\delta[K+2]] - \frac{1}{2}j[256\delta[K-20] - 256\delta[K+20]]$$

Equation 6. Mathematical Representation of the FFT output

To relate the discrete-time frequency and the continuous-time frequency, Equation 7 illustrates the relation used.

$$W(\text{discrete}) = W(\text{continuous}) \times \text{SamplingPeriod}$$

Equation 7. Discrete and Continuous Frequency

For the discrete time-frequency components, (k) is used, as in 2πk/N, where N is the number of points displayed (see *DSP First* p340 - 343). Therefore, the first two spikes in the discrete frequency domain correspond to 2π(2)/256 and 2π(20)/256. Because the sampling frequency is 256 samples per second, the continuous-time frequency is obtained by multiplying 2π(2)/256 and 2π(20)/256 by 256. This gives continuous-time radial frequencies of 2π(2) and 2π(20), or 2Hz and 20Hz, which Figure 4 confirms.

5 Performance

Performance results are given in clock cycles for a Freescale T4240 microprocessor. To obtain the execution time, divide clock cycles by Megahertz. Performance for other AltiVec implementations may vary. Also, performance can vary depending on the C compiler used and can improve as the quality of a compiler improves from release to release. It is assumed that all required instructions and data are present in the L1 cache.

Both versions of the Radix-2 DIF FFT function are executed to find the number of clock cycles necessary for completion. These values are recorded in Table 3. From the clock cycles, the performance gain is calculated as the scalar clock cycles divided by vector clock cycles.

Table 3. Radix-2 DIF FFT Performance (Vector versus Scalar)

N	Scalar Clock Cycles	Vector Clock Cycles	Scalar/Vector
64	4,882	1,953	2.5
128	11,153	3,870	2.9
256	25,700	8,582	3.0
512	58,436	19,010	3.1
1024	132,145	41,862	3.2
2048	294,364	94,043	3.1
4096	666,376	223,447	3.0
8192	1,546,983	537,951	2.9
16384	3,605,714	1,226,338	2.9
32768	8,115,185	2,683,651	3.0
65536	17,498,392	6,663,110	2.6
131072	38,844,895	14,142,322	2.7
262144	88,794,243	35,716,050	2.5

The gain values in Table 3 show that the AltiVec version of the Radix-2 DIF FFT consistently outperforms the scalar version. A performance gain of over 3.0 is seen for larger FFTs where inner-loop computation is a larger part of the execution time than overhead. Performance degrades at N = 8192 and greater because 8192 complex single-precision points consumes the 32K data cache of the T4240. Larger dataset sizes are influenced by hierarchical memory performance, which reduces incremental performance for both scalar and vector routines - though vector remains faster.

To read more about FFT functions and other signal processing techniques and terms, see Section 7, "References."

6 Appendix

6.1 Scalar FFT Function Source Code

```
int sc_fft_dif( cplx *pfs, cplx *pfw, unsigned int n, unsigned int log_n )
{
  unsigned int stage,block,j,iw=0;
  unsigned int pa,pb,qa,qb;
  unsigned int stride,edirts;
  cplx ft1a,ft1b,ft2a,ft2b,ft3a,ft3b;

  //INIT
  stride = n/2;
  edirts = 1;

  //DIF FFT
  for( stage=0; stage<log_n-2; stage++ ) {
```

```
  for( block=0; block<n; block+=stride*2 ) {
    pa = block;
    pb = block + stride/2;
    qa = block + stride;
    qb = block + stride/2 + stride;
    iw = 0;
    for( j=0; j<stride/2; j++ ) {    //2bufflies/loop
     //add
     ft1a.re = pfs[pa+j].re + pfs[qa+j].re;
     ft1a.im = pfs[pa+j].im + pfs[qa+j].im;
     ft1b.re = pfs[pb+j].re + pfs[qb+j].re;
     ft1b.im = pfs[pb+j].im + pfs[qb+j].im;

     //sub
     ft2a.re = pfs[pa+j].re - pfs[qa+j].re;
     ft2a.im = pfs[pa+j].im - pfs[qa+j].im;
     ft2b.re = pfs[pb+j].re - pfs[qb+j].re;
     ft2b.im = pfs[pb+j].im - pfs[qb+j].im;
     pfs[pa+j] = ft1a;       //store adds
     pfs[pb+j] = ft1b;
     //cmul
     pfs[qa+j].re =  ft2a.re * pfw[iw].re -
       ft2a.im * pfw[iw].im;
     pfs[qa+j].im =  ft2a.re * pfw[iw].im +
       ft2a.im * pfw[iw].re;
     //twiddled cmul
     pfs[qb+j].re =  ft2b.re * pfw[iw].im +
       ft2b.im * pfw[iw].re;
     pfs[qb+j].im = -ft2b.re * pfw[iw].re +
       ft2b.im * pfw[iw].im;
     iw += edirts;
    }
  }
  stride = stride>>1;
  edirts = edirts<<1;
}
//last two stages
for( j=0; j<n; j+=4 ) {
  //upper two
  ft1a.re = pfs[j  ].re + pfs[j+2].re;
  ft1a.im = pfs[j  ].im + pfs[j+2].im;
  ft1b.re = pfs[j+1].re + pfs[j+3].re;
  ft1b.im = pfs[j+1].im + pfs[j+3].im;
  ft2a.re = ft1a.re + ft1b.re;
  ft2a.im = ft1a.im + ft1b.im;
  ft2b.re = ft1a.re - ft1b.re;
  ft2b.im = ft1a.im - ft1b.im;

  //lower two
  //notwiddle
  ft3a.re = pfs[j].re - pfs[j+2].re;
  ft3a.im = pfs[j].im - pfs[j+2].im;
  //twiddle
  ft3b.re =  pfs[j+1].im - pfs[j+3].im;
  ft3b.im = -pfs[j+1].re + pfs[j+3].re;
  //store
  pfs[j  ]    = ft2a;
  pfs[j+1]    = ft2b;
  pfs[j+2].re = ft3a.re + ft3b.re;
```

```
    pfs[j+2].im = ft3a.im + ft3b.im;
    pfs[j+3].re = ft3a.re - ft3b.re;
    pfs[j+3].im = ft3a.im - ft3b.im;

  }
  return 0;
}
```

6.2 Vector FFT Function Source Code

```
int av_fft_dif( vector float *pvf, vector float *pw, unsigned int n, unsigned int log_n )
{
  //local constants
  const vector float vcfzero    = (vector float)( 0., 0., 0., 0.);
  const vector float vcfnegeven = (vector float)(-0., 0.,-0., 0.);
  const vector float vcfnegodd  = (vector float)( 0.,-0., 0.,-0.);
  const vector float vcppnn     = (vector float)( 1., 1.,-1.,-1.);
  const vector float vcpnnp     = (vector float)( 1.,-1.,-1., 1.);
  //SINGLE VECTOR PERM FORMS (V=[0,1,2,3])
  const vector unsigned char vcprm1032 =
    (vector unsigned char)( 4,5,6,7, 0,1,2,3,  12,13,14,15, 8,9,10,11 );
  const vector unsigned char vcprm0022 =
    (vector unsigned char)( 0,1,2,3, 0,1,2,3,  8,9,10,11, 8,9,10,11 );
  const vector unsigned char vcprm1133 =
    (vector unsigned char)( 4,5,6,7, 4,5,6,7,  12,13,14,15, 12,13,14,15 );
  const vector unsigned char vcprm2301 =
    (vector unsigned char)( 8,9,10,11,12,13,14,15, 0,1,2,3,4,5,6,7 );
  const vector unsigned char vcprm0101 =
    (vector unsigned char)( 0,1,2,3,4,5,6,7, 0,1,2,3,4,5,6,7 );
  const vector unsigned char vcprm3232 =
    (vector unsigned char)( 12,13,14,15,8,9,10,11, 12,13,14,15,8,9,10,11 );
  //DOUBLE VECTOR PERM FORMS (V1,V2=[ar,ai,x,x],[br,bi,x,x] x=dontcare)
  const vector unsigned char vcaiarbibr =
    (vector unsigned char)( 4,5,6,7, 0,1,2,3,  20,21,22,23, 16,17,18,19 );
  const vector unsigned char vcaraibrbi =
    (vector unsigned char)( 0,1,2,3,4,5,6,7,  16,17,18,19,20,21,22,23 );

  //VECTOR LOCALS
  vector float vtf10,vtf11,vtf12,vtf13,vtf14,vtf15,
    vtf20,vtf21,vtf22,vtf23,vtf24,vtf25,
    vtf31,vtf32,vtf33,vtf34,vtf35,
    vtf41,vtf42,vtf43,vtf44,vtf45,
    vtfw0,vtfw1,vtfw2,vtfw3;

  //ITERATORS&INDICES
  unsigned int pa,pb,qa,qb,stride,edirts,iw;//indices
  unsigned int stage,block,j;                                  //iterators

  //BEGIN DIF FFT
  //FIRST STAGE
  stride = n/2;   /* in stage 0 stride = n/2 */

  //special case since twiddles are (w1r,w1i,w2r,w2i) in same vec
  //and only one block
  for( j=0; j<stride/4; j++ ) {
    //PREP THE TWIDDLES
    vtfw1 = vec_perm( pw[j], pw[j], vcprm1032 );
    vtfw2 = vec_xor( vtfw1, vcfnegeven );//(-i, r)
    vtfw3 = vec_xor( vtfw1, vcfnegodd  );//( i,-r)
```

```
  //BUTTERFLIES
  //upper two
  vtf10     = vec_add( pvf[j], pvf[n/4+j] );//PA+QA
  vtf11     = vec_sub( pvf[j], pvf[n/4+j] );//PA-QA
  pvf[j] = vtf10;

  vtf20     = vec_add( pvf[n/8+j], pvf[n/8+n/4+j] );//PB+QB
  vtf21     = vec_sub( pvf[n/8+j], pvf[n/8+n/4+j] );//PB-QB
  pvf[n/8+j] = vtf20;

  //lower two (apply twiddle)
  vtf12  = vec_perm( vtf11, vtf11, vcprm0022 );
  vtf12     = vec_madd( vtf12, pw[j], vcfzero );
  vtf13     = vec_perm( vtf11, vtf11, vcprm1133 );
  pvf[n/4+j] = vec_madd( vtfw2, vtf13, vtf12 );

  vtf22     = vec_perm( vtf21, vtf21, vcprm1133 ); //imag
  vtf22     = vec_madd( vtf22, pw[j], vcfzero );   //w.imag
  vtf23     = vec_perm( vtf21, vtf21, vcprm0022 ); //real
  pvf[n/4+n/8+j] = vec_madd( vtfw3, vtf23, vtf22 );//w.real
}
//END FIRST STAGE

stride = n/4;  /* in stage 1 stride = n/4 */

//STAGES iterate over remaining, less the last two
edirts = 1;
for( stage=1; stage<log_n-2; stage++ ) {
  //BLOCKS iterate for the number of blocks
  for( block=0; block<n/2; block+=stride )  {
    pa = block;
    pb = block + stride/4;
    qa = block + stride/2;
    qb = block + stride/4 + stride/2;
    iw = 0;
    //itr(stride) of the block, /4 since 2 bufflies/vec and 2vecs/loop
    for( j=0; j<stride/4; j++ ) {
     //PREP THE TWIDDLES
     vtfw0 = vec_perm( pw[j*2*edirts],
              pw[j*2*edirts+edirts],
              vcaraibrbi );
     vtfw1 = vec_perm( pw[j*2*edirts],
              pw[j*2*edirts+edirts],
              vcaiarbibr );//swap r,i
     vtfw2 = vec_xor( vtfw1, vcfnegeven );//(-i, r)
     vtfw3 = vec_xor( vtfw1, vcfnegodd  );//( i,-r)

       //BUTTERFLIES
     //upper two
     vtf10     = vec_add( pvf[pa+j], pvf[qa+j] );//PA+QA
     vtf11     = vec_sub( pvf[pa+j], pvf[qa+j] );//PA-QA
     pvf[pa+j] = vtf10;

     vtf20     = vec_add( pvf[pb+j], pvf[qb+j] );//PB+QB
     vtf21     = vec_sub( pvf[pb+j], pvf[qb+j] );//PB-QB
     pvf[pb+j] = vtf20;

     //lower two (apply twiddle)
     vtf12  = vec_perm( vtf11, vtf11, vcprm0022 );
     vtf12     = vec_madd( vtf12, vtfw0, vcfzero );
     vtf13     = vec_perm( vtf11, vtf11, vcprm1133 );
     pvf[qa+j] = vec_madd( vtfw2, vtf13, vtf12 );
```

```
      vtf22     = vec_perm( vtf21, vtf21, vcprm1133 );//imag
      vtf22     = vec_madd( vtf22, vtfw0, vcfzero );  //w.imag

      vtf23     = vec_perm( vtf21, vtf21, vcprm0022 ); //real
      pvf[qb+j] = vec_madd( vtfw3, vtf23, vtf22 );   //w.real
     }
     //end BUTTERFLIES
   }

   //stride halves, block count (twiddle separation) doubles
   stride = stride >> 1;
   edirts = edirts << 1;

 }
 //end STAGES (third-to-last stage, actually)

 //Last two stages (quad unrolled; forces n>=16)
 for( block=0; block<n/2; block+=8 ) { //(n/2 since 1Vec=2pts)
   vtf13 = vec_sub( pvf[block  ], pvf[block+1] );//PB1
   vtf11 = vec_add( pvf[block  ], pvf[block+1] );//PA1
   vtf23 = vec_sub( pvf[block+2], pvf[block+3] );//QB1
   vtf21 = vec_add( pvf[block+2], pvf[block+3] );//QA1
   vtf33 = vec_sub( pvf[block+4], pvf[block+5] );//PB1
   vtf31 = vec_add( pvf[block+4], pvf[block+5] );//PA1
   vtf43 = vec_sub( pvf[block+6], pvf[block+7] );//QB1
   vtf41 = vec_add( pvf[block+6], pvf[block+7] );//QA1

   vtf15 = vec_perm( vtf13, vtf13, vcprm3232 );//PB2b
   vtf14 = vec_perm( vtf13, vtf13, vcprm0101 );//PB2a
   vtf12 = vec_perm( vtf11, vtf11, vcprm2301 );//PA2
   vtf25 = vec_perm( vtf23, vtf23, vcprm3232 );//QB2b
   vtf24 = vec_perm( vtf23, vtf23, vcprm0101 );//QB2a
   vtf22 = vec_perm( vtf21, vtf21, vcprm2301 );//QA2
   vtf35 = vec_perm( vtf33, vtf33, vcprm3232 );//PB2b
   vtf34 = vec_perm( vtf33, vtf33, vcprm0101 );//PB2a
   vtf32 = vec_perm( vtf31, vtf31, vcprm2301 );//PA2
   vtf45 = vec_perm( vtf43, vtf43, vcprm3232 );//QB2b
   vtf44 = vec_perm( vtf43, vtf43, vcprm0101 );//QB2a
   vtf42 = vec_perm( vtf41, vtf41, vcprm2301 );//QA2

   pvf[block]   = vec_madd( vtf11, vcppnn, vtf12 );//PA3
   pvf[block+1] = vec_madd( vtf15, vcpnnp, vtf14 );//PB3
   pvf[block+2] = vec_madd( vtf21, vcppnn, vtf22 );//QA3
   pvf[block+3] = vec_madd( vtf25, vcpnnp, vtf24 );//QB3
   pvf[block+4] = vec_madd( vtf31, vcppnn, vtf32 );//PA3
   pvf[block+5] = vec_madd( vtf35, vcpnnp, vtf34 );//PB3
   pvf[block+6] = vec_madd( vtf41, vcppnn, vtf42 );//QA3
   pvf[block+7] = vec_madd( vtf45, vcpnnp, vtf44 );//QB3
 }
 //end last stage
 //END DIF FFT
 return 0;
}
```

7 References

1. Eleanor Chu and Alan George. *Inside The FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. ISBN 0-8493-0270-b, CRC Press, 2000.
2. *The FFT Demystified*. http://www.eptools.com/tn/T0001/INDEX.HTM, Engineering Productivity Tools Ltd., 1999.
3. J. H. McClellan, R. W. Schafer and M. A. Yoder. *DSP First*. ISBN 0-13-243171-8, Prentice-Hall, 1998.
4. A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing*. p599, ISBN 0-130216292-X, Prentice-Hall, 1989.

8 Revision History

Table 4 summarizes the revision history of this document.

Table 4. Revision History

Rev. Number	Date	Substantive Change(s)
4	04/2013	Clarified Equation 2 Corrected Figure 1 Corrected Table 2 Clarified Equation 7 Removed erroneous MPC7410 performance and updated Table 3 to the T4240
3		Rebranding for Freescale; not-technical formatting.
2.1	2003	Nontechnical reformatting
2	2002	Modified Table 3. Updated scalar FFT function and vector FFT function source code. Added revision history.
0–1	2001	Initial release.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN2115
Rev. 4
04/2013