Implementing Radix-2 FFT Algorithms on the TMS470R1x

ABSTRACT

This application report describes implementing Radix-2 FFT algorithms on the TMS470R1x. The FFT is implemented to work with complex input data. The key objective is to get a fast execution time, with obtaining a small code size secondary.

Contents

List of Figures

List of Tables

1 Introduction

In many applications, specific signals are measured with sensors. These signals contain information necessary for the application to perform its tasks. The signal has to be transformed by special algorithms before the application can get the information from it.

In most cases, it is useful to convert the signal from its time domain into the frequency domain to determine the spectrum of the signal and the different frequencies it is made up of. To complete this conversion, a Fourier transform has to be performed. Different algorithms were developed for this task, such as discrete Fourier transforms (DFT) or fast Fourier transforms (FFT).

This application report explains a Radix-2 FFT algorithm to convert a signal into the frequency domain. It works on complex input data, where the real and imaginary parts are stored in two separate arrays.

Other algorithms, such as the L-Shaped Butterfly, Hadamard Transform, etc., have a better execution speed. However, they have certain limitations, for example, resolution. As the Radix-2 is the most common transform used and therefore offers a good basis for performance comparison, we limit ourselves to this FFT. The other algorithms are not explained in this application report.

2 TMS470R1x Architecture

The TMS470R1x contains a 16/32bit RISC CPU with a von Neumann architecture. All arithmetic operations have to be performed in registers (of a load/store architecture). Therefore, all the parts of a calculation have to be loaded into registers first. It also has a 32x8 hardware-multiplier implemented. The instruction cycle time of the hardware multiplier depends on the format of the input data. It varies between 2 and 5 cycles for a multiplication. All other data processing instructions need 1 or 2 cycles. This difference in the cycle time leads to the conclusion that the implemented algorithm should use as few multiplications as possible.

3 Fourier Transform

With the Fourier transform, a function is split up in a sum of sine functions with different frequencies. To get the original signal, the sine functions have to be overlaid. The time signal is transformed into the frequency domain.

Fourier transform of signal s(t):

$$S(f) = \int_{-\infty}^{\infty} s(t) e^{-12\pi f t}\, dt \tag{1}$$

s(t) = Magnitude of the signal

f = Frequency of the signal

To do this transform with a CPU this equation has to be numerically integrated as shown in Equation 2:

$$S(f_k) = \sum_{i=0}^{N-1} s(t_i) e^{-12\pi f_k t_i} (t_i + 1 - t_i) \qquad k = 0, 1, \ldots N-1 \tag{2}$$

If we look closely at this equation, we can see that the time for the calculation of the N sine components is proportional to N2. This is a long computation time, which is not useful for real-time applications. This lengthy computation time led to a special implementation of the Fourier transform, the fast Fourier transform (FFT).

4 FFT

The FFT takes advantage of the cyclic features of the exponential function.

$$W_{N = e}{}^{-j2\pi / N} \tag{3}$$

The equation for the discrete Fourier transform can be written as:

$$X(k) = \sum_{n=0}^{N-1} W_N^{mm} X[n] \qquad m = 0, 1, \ldots, N-1 \qquad [16, \text{p } 106] \tag{4}$$

Figure 1 explains the cyclic features (twiddle factors) of the exponential function.

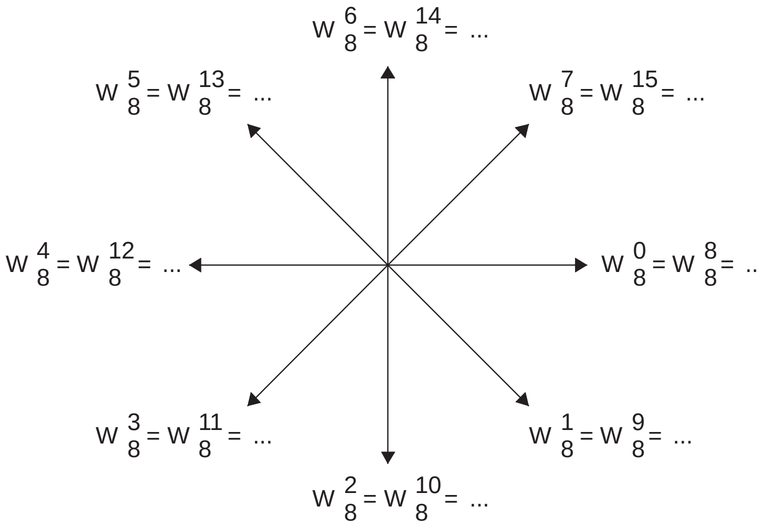

Figure 1. Twiddle Factors (W8)

Explicitly, the twiddle factors translate into the following:

$$W_8^0 = W_8^8 = \cos(0°) - j\sin(0°) = 1 - j0$$

$$W_8^1 = W_8^9 = \cos(45°) - j\sin(45°) = 0.7 - j0.7$$

$$W_8^2 = W_8^{10} = \cos(90°) - j\sin(90°) = 0 - j1$$

$$W_8^3 = W_8^{11} = \cos(135°) - j\sin(135°) = 0.7 - j0.7$$

$$W_8^4 = W_8^{12} = \cos(180°) - j\sin(180°) = -1 - j0$$

$$W_8^5 = W_8^{13} = \cos(225°) - j\sin(225°) = -0.7 - j0.7$$

$$W_8^6 = W_8^{14} = \cos(270°) - j\sin(45°) = 0 + j1$$

$$W_8^7 = W_8^{15} = \cos(315°) - j\sin(315°) = 0.7 - j0.7$$

The measured samples can be split up in an even and an odd part.

Equation 4 can be rewritten as follows:

$$X(k) = \sum_{n=0}^{P-1} x[2n] W_N^{2mn} + W_N^m \sum_{P=0}^{P-1} x[2n+1] W_N^{2mn} \qquad (5)$$

$$X_a(k) = \sum_{n=0}^{P-1} a[n] W_P^{mn} \qquad m = 0, 1, \ldots, P-1 \qquad (6)$$

$$X_b(k) = \sum_{n=0}^{P-1} b[n] W_P^{mn} \qquad m = 0, 1, \ldots, P-1 \tag{7}$$

(a[n] = x[2n] and b[n] = x[2n+1])

With $W_N = e^{-j2\pi/N}$ it can be seen that $W_P = W_N^2$ and $W_N^{P2mn} = W_P^{mn}$.

Equation 5 can be written in another way as follows:

$$X[k] = X_a[k] + W_N^m X_b[k] \tag{8}$$

4.1 Radix-2 FFT

To understand the basics of a FFT, it is often useful to look to a special flow diagram.

Figure 2 shows a diagram for an 8-point radix-2 DIT-FFT (decimation in time-FFT). There are several ways to calculate a radix-2 FFT because the derivation from the DFT can be performed differently. Finally, we end up with the distinction of *decimation in time* and *decimation in frequency*, depending on how the twiddle factors are arranged in the butterfly. In addition, we can have bit-reversed inputs or outputs. The scrambling caused by the bit-reversal can be corrected in the first or the last stage of the FFT.

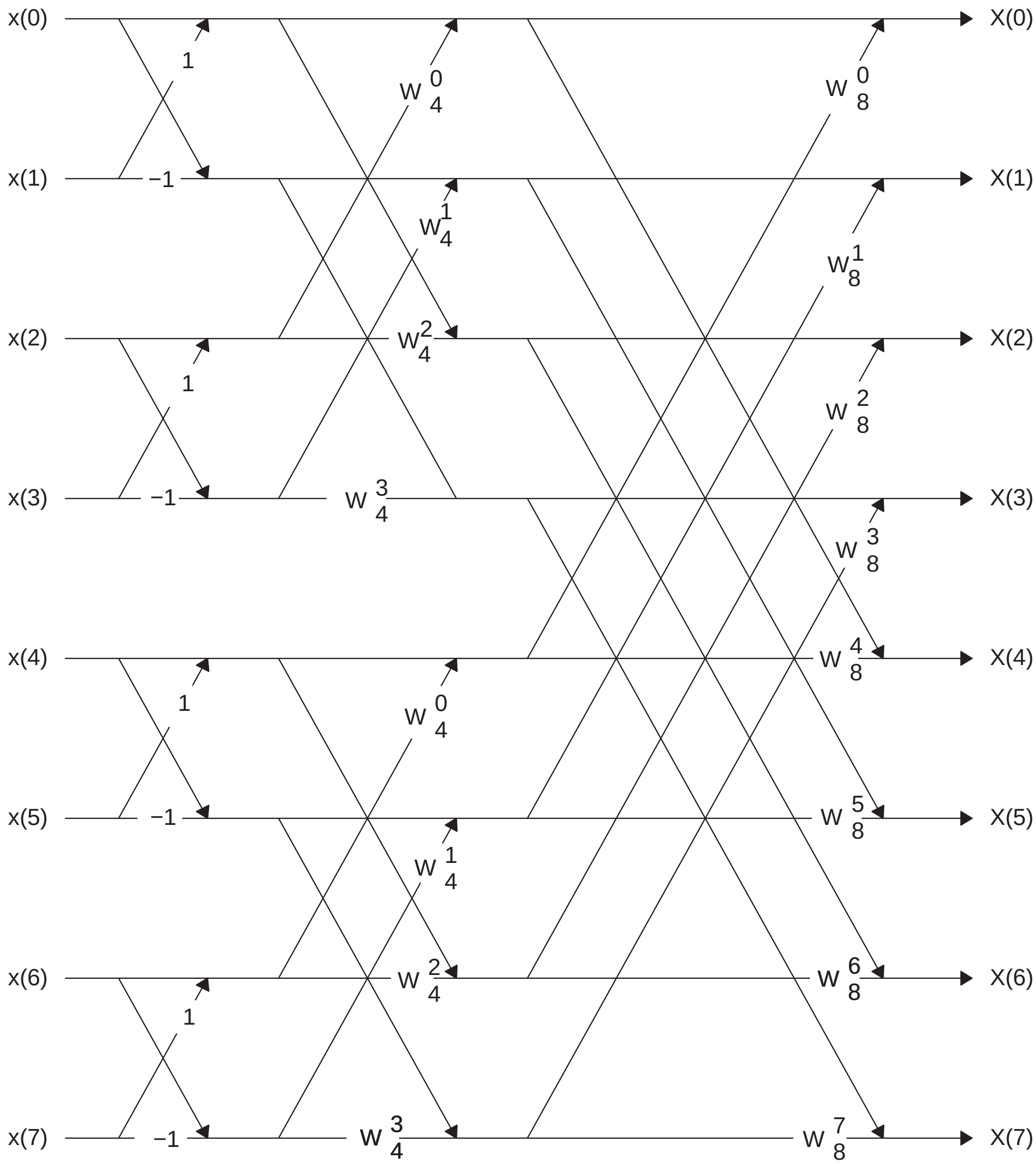

Figure 2. 8-Point DIT-Radix2-FFT

In stage 1, no multiplication is needed, since the twiddle factor

$$w_N^0$$

is always 0 for sine and 1 for cosine. The real and imaginary parts of the butterfly can be calculated with the following equations:

$$x\_re(n,s+1) = x\_re(n,s) + x\_re(n+t,s) * \cos(x) + x\_im(n+t,s) * \sin(x) \tag{9}$$

$$x\_im(n,s+1) = x\_im(n,s) + x\_im(n+t,s) * \cos(x) - x\_re(n+t,s) * \sin(x) \tag{10}$$

$$x\_re(n+t,s+1) = x\_re(n,s) - x\_re(n+t,s) * \cos(x) - x\_im(n+t,s) * \sin(x) \tag{11}$$

$$x\_im(n+t,s+1) = x\_im(n,s) - x\_im(n+t,s) * \cos(x) + x\_re(n+t,s) * \sin(x) \tag{12}$$

The cosine and sine values are normally implemented as lookup tables because the calculation takes too long. The amount of ROM required depends on the number of points, which have to be calculated. This means that for an N-point FFT, 2N values need to be stored. To minimize these requirements, a single table with two pointers (one for cosine, one for sine) can be implemented.

4.2 Bit Reversal

Bit reversal is necessary to reorder the results or respectively the input data.

Discrete transforms are the main users of bit-reverse and digit-reverse routines. Discrete transforms take discrete inputs in one domain and convert them to discrete outputs in another. For example, an FFT takes a discrete time domain input and transforms it into the discrete frequency domain output (i.e., x(t) → X(jwt).)

Bit-reverse and digit-reverse routines are routines in which the data is reordered based on its index value from *0* to *–1*, where *N* is the number of points to be bit-reversed.

Many discrete transforms (FFT, DCT, IDCT, DST, etc.) are executed in place using the same memory locations for both the input and output. This reduces both data size and algorithmic complexity. Bit-reversing routines are needed to take full advantage of in-place execution. For example, if the input is in normal order but the output is in bit-reverse order, then you have to do bit reversal during the last stage of the FFT to view the resulting output in normal order.

A disadvantage of in-place bit reversal is that the original input data is lost, which is why we implemented an out-of-place algorithm. The bit-reversal is implemented in the first stage of the FFT. This results in larger RAM requirements and additional code. The extra code needed can easily be separated from the FFT calculation itself and therefore creates no overhead in the cycle count for the FFT routine.

Since the TMS470 has no bit-reversed addressing mode, we created a bit-reversed offset table. Using this table we were able to implement the bit-reversed addressing mode using the normal load commands.

The bit reversal is quite simple. If we take the binary format of the address the sample is stored in and mirror it, we get the bit-reversed address. See examples below in Figure 3.

Odd number of bits

0 0 0		0 0 0
0 0 1		1 0 0
0 1 0		0 1 0
0 1 1	→	1 1 0
1 0 0		0 0 1
1 0 1		1 0 1
1 1 0		0 1 1
1 1 1		1 1 1

Bit 0 and bit 2 are exchanged.
Bit 1 is in the middle and remains unchanged.

Even number of bits

0 0 0 0		0 0 0 0
0 0 0 1		1 0 0 0
0 0 1 0		0 1 0 0
0 0 1 1	→	1 1 0 0
0 1 0 0		0 0 1 0
0 1 0 1		1 0 1 0
0 1 1 0		0 1 1 0
0 1 1 1		1 1 1 0

Bit 0 and bit 3 are exchanged.
Bit 1 and bit 2 are exchanged.

Figure 3. Examples of Bit-Reversed Address

5 Implementation

The algorithm is implemented in assembler and is configurable. This means the number of points can be configured at compile time. The range is $8 \le N \le 128$ points.

By defining with -dN=x at the command line (compiler option), the number of points can be chosen. The first, second and last stages are not implemented with a macro as the other stages are. When compiling, a macro expansion is performed, depending on how N is configured. This method eliminates unnecessary branches and the code size is reduced because only the used code gets inlined.

The number of points is limited to 128 because of register indexing constraints from the TMS470R1x CPU. If a higher number is needed, a different implementation scheme has to be used.

The assembler routine can be called from assembler or C. The algorithm uses different arrays for the input data, the temporary values and the result. The arguments to the function are passed via registers and the stack. The arrays for the real and imaginary parts should be consecutive.

R0 → pointer to real part of an temporary array

R1 → pointer to bitreversal offset table

R2 → pointer to sine table (twiddle factors)

R3 → pointer to real part of the input array

arg5 → pointer to imaginary part of the input array (Stack)

arg6 → pointer to result (Stack)

C-Call example:

Rad2fft(&x2_re[0], &brev[0], &sine[0], &x1_re[0], &x1_im[0], &result[0]);

Where:

&x2_re[0] is a pointer to the first element of the real part of the temporary array

&brev[0] is a pointer to the first element of the bitreversal offset table

&sine[0] is a pointer to the first element of the sine table

&x1_re[0] is a pointer to the first element of the real part of the input array

&x1_im[0] is a pointer to the first element of the imaginary part of the input array

&result[0] is a pointer to the first element of the result array

The complete assembler listing and a sample in C how to call the function are shown in Appendix A. The results are shown in Section 6.

6 Results

The results of the Radix-2 FFT algorithm are shown in Table 1.

Table 1. Radix-2 FFT Algorithm Results

	8-Point FFT	16-Point FFT	64-Point FFT	128-Point FFT
Code size (1)	396 Byte	580 Byte	948 Byte	1132 Byte
Table size (2)	36 Byte	72 Byte	144 Byte	288 Byte
Array size (3)	80 Byte	160 Byte	320 Byte	640 Byte
Cycle count (4)	592	1573	9563	22508

(1) Code size means the size of the Rad2fft function.
(2) Table size is the size of the constant tables. In the example code they are named sine and brev. For testing reasons, they are defined as variables, not as constants.
(3) Array size is the size of the input stream, and intermediate and output arrays. The cycle count is based on the input of the fundamental frequency.
(4) If other input signals are used, the cycle count may vary slightly, because of the multiplier of the ARM7.

Appendix A Creating a COFF

A.1 Files Needed

The files shown in Table A-1 are needed to create an executable common object file format (COFF):

Table A-1. Files Needed to Create a COFF

File Name	Description
c32.bat	Batch file that launches code translation
files.cmd	List of files to be compiled/assembled
Main.c	Contains C-routine calling the FFT
Rad2.asm	Contains Assembler code for FFT
intvecs.asm	Interrupt vector table
startup.c	Sets up system environment for SE-chip
result.txt	Contains input signal and FFT result

These files are available as sources.

A.2 Example Program

The following is an example program.

```
Main.c
/*********************************************************************************/
/*                                                                               */
/*  PROJECT:    TEST OF RADIX-2 FFT                                              */
/*                                                                               */
/*********************************************************************************/
#include "stdio.h"
#include "math.h"
volatile int     a,b;
short                           x1_re[N];
short                           x1_im[N];
short                           x2_re[N];
short                           x2_im[N];
short                           sine[N+N/4];
unsigned short    result[N];
short                           brev[N];
extern void sineinit();
void bit_rev_init(short *br)
{
volatile int bit,rev,tmp1,tmp2,maskl,maskh,m,n,shift;
volatile float x,y;
   x = log(N);
   y = log(2);
   bit = (int)(x/y);
   rev = 0;
   for (n=0;n<N;n++)
    {
     for (m=0;m<(bit/2)+0.5;m++)
         shift = (bit-((m*2)+1));
         maskl = 1<<m;
         tmp1 = (n & maskl) << shift;
                      maskh = N/2>>m;
         tmp2 = (n & maskh) >> shift;
        rev = rev | tmp1 | tmp2;
      }
       br[n] = rev * 2;
       rev = 0;
    }
```

```
}
void sineinit(short *si)
{
    double  rad;
    int I;
    I = 0;
    while (++I < (N + N/4) )
    {
             rad = (double)(i*2.0*3.141592654/N);
                     si[i] = (int)(sin(rad)*32768);
    }
}
void Print()
{
        FILE    *f;
        f = fopen("result.txt","w");

        if (f == NULL)
               printf("Error in opening file 'result.txt'\n");
        for(a=0;a<N;a++)
               fprintf(f,"\%d\n",x1_re[a]);
        fprintf(f,"\n");
        for(a=0;a<N;a++)
                {
                fprintf(f,"\%d\n",result[a]);
                }
        fclose(f);
}
main()
{
double          rad;
volatile                int A = 305;
volatile                int f = 1;
    sineinit(&sine[0]);
    bit_rev_init(&brev[0]);
       for(a=0;a<N;a++)
                x1_re[a] = x1_im[a] = x2_re[a] = x2_im[a] = 0;
    for(f=1;f<N;f++)
        {
                //-----test-----
                for(a=0;a<N;a++)
                {
                        rad = (double)((f*a*2.0*3.141592654)/(N));
                        x1_re[a] = (int)(sin(rad)*A);
                        /*
                        if (a != N/2)
                                x1_re[a] = (((int)(sin(rad)*A+0.5))-1)/2;
                        else
                                x1_re[a] = ((int)(sin(rad)*A+0.5))/2;
                        */
                }
                //----test end--
                //---calculate radix-2 FFT-----
                Rad2fft(&x2_re[0], &brev[0], &sine[0], &x1_re[0], &x1_im[0], &result[0]);

                //---Write result to file----
               Print();
        }
    for(;;);
}
Rad2.asm
;******************************************************************************
;*                                                                            *
;*  Optimized assembler program for Radix-2 FFT algorithm                     *
;*                                                                            *
;******************************************************************************
```

```
                  .global _Rad2fft

      ; R0          -> pointer to x2_re
      ; R1          -> pointer to brev
      ; R2          -> pointer to sine
      ; R3          -> pointer to x1_re
      ; arg5        -> pointer to x1_im
      ; arg6        -> pointer to result
      ;-------------------------------------------------------------------------------
      ; macro definition of stage
      ;          off    defines the offset of the two values used to calculate the butterfly
      ;          butt    defines how often a butterfly is executed inside a block
      ;          block    defines how often blocks are executed
      ;          sioff    defines the offset of the sine table values used
      ;          alsi    defines the alignment of the sine table pointer after each
      ;          block
      ;-------------------------------------------------------------------------------
      stage      .macro    off, butt, block, sioff, alsi
            mov      R1,#block
            mov      R3,#butt
            add      R0,R0,#(off-2)         ;align address
            sub         R2,R2,#sioff*2         ;align twiddle
      stage1_?:
            ldrsh       R4,[R0,#-(off-2)]!        ;x_re(m)
            ldrsh       R5,[R0,LR]             ;x_im(m)
            ldrsh       R6,[R0,#off]!          ;x_re(m+4)
            ldrsh       R7,[R0,LR]                         ;x_im(m+4)
            ldrsh       R8,[R2,#sioff*2]!            ;wi sin(k+8)
            ldrsh       R9,[R2,#(N/4)*2]             ;wr cos(k+8)
      stage2_?:
               mul         R12,R9,R6                      ;x_re(m+4) * cos(k+8)
               mul         R11,R8,R7                      ;x_im(m+4) * sin(k+8)
               add         R10,R11,R12                    ;x_re(m+4) * cos(k+8) + x_im(m+4) * sin(k+8)
               mov         R10,R10, LSL#1
               mov         R10,R10, ASR#16
               add         R10,R4,R10                     ;x_re(m) + x_re(m+4) * cos(k+8) + x_im(m+4)
      * sin(k+8)
               strh        R10,[R0,#-off]!          ;x_re(m)= x_re(m) + [x_re(m+4)*cos(k+8) +
      x_im(m+4)*sin(k+8)]
               add         R10,R12,R11                    ;x_re(m+4) * cos(k+8) + x_im(m+4) * sin(k+8)
               mov         R10,R10, LSL#1
               mov         R10,R10, ASR#16
               sub         R4,R4,R10                      ;x_re(m) - [x_re(m+4) * cos(k+8) - x_im(m+4)
      * sin(k+8)]
               mul         R12,R9,R7                      ;x_im(m+4) * cos(k+8)
               mul         R11,R8,R6                      ;x_re(m+4) * sin(k+8)
               sub         R10,R12,R11                    ;x_im(m+4) * cos(k+8) - x_re(m+4) * sin(k+8)
               mov         R10,R10, LSL#1
               mov         R10,R10, ASR#16
               add         R10,R5,R10                     ;x_im(m) + x_im(m+4) * cos(k+8) - x_re(m+4)
      * sin(k+8)

               strh        R10,[R0,LR]                 ;x_im(m) = x_im(m) + [x_im(m+4) * cos(k+8) -
      x_re(m+4) * sin(k+8)]
               strh        R4,[R0,#off]!            ;x_re(m) = x_re(m) - [x_re(m+4) * cos(k+8) -
      x_im(m+4) * sin(k+8)]
               sub         R10,R12,R11                 ;x_im(m+4) * cos(k+8) - x_re(m+4) * sin(k+8)
               mov         R10,R10, LSL#1
               mov         R10,R10, ASR#16
               sub         R10,R5,R10                  ;x_im(m) - x_im(m+4) * cos(k+8) - x_re(m+4) *
      sin(k+8)
               strh        R10,[R0,LR]                 ;x_im(m) = x_im(m) - [x_im(m+4) * cos(k+8) -
      x_re(m+4) * sin(k+8)]
               subs        R3,R3,#1
               bne         stage1_?
               subs        R1,R1,#1
```

```
          movne       R3,#butt
          ldrnesh        R4,[R0,#2]!                      ;x_re(m+8)
          ldrnesh        R5,[R0,LR]                       ;x_im(m+8)
          ldrnesh        R6,[R0,#off]!                 ;x_re(m+12)
          ldrnesh        R7,[R0,LR]                       ;x_im(m+12)
          ldrnesh        R8,[R2,#-alsi]!            ;wi sin(k+8)
          ldrnesh        R9,[R2,#(N/4)*2]           ;wr cos(k+8)
          bne       stage2_?
         .endm
;----end of macro definition
_Rad2fft:
      stmfd   SP!,{R4-R12,LR}        ; save used registers

          ldr            R4,[SP,#40]                        ; &x1_im[0]
          stmfd          SP!,{R0-R2}                        ; store registers for later use
;-------------------------------------------------------------------------------
; STAGE 1 (BITREVERSAL IS ALREADY PERFORMED IN THIS STAGE)
;-------------------------------------------------------------------------------
          mov            LR,#N*2                               ;offset to x1_im
          mov            R12,#N/2
          sub            R0,R0,#2
stage1:
          ldrsh          R8,[R1],#2                         ;load offset from bitreversal table
          ldrsh          R9,[R1],#2                         ;load offset from bitreversal table
          ldrsh          R10,[R3,R8]                        ;load x1_re[m]
          ldrsh          R5,[R3,R9]                         ;load x1_re[m+offs]
          ldrsh          R11,[R4,R8]                        ;load x1_im[m]
          ldrsh          R7,[R4,R9]                         ;load x1_im[m+offs]
          add            R6,R10,R5
          strh           R6,[R0,#2]!                        ;x2_re(m) = x1_re[m] + x1_re[m+offs]
          add            R6,R11,R7
          strh           R6,[R0,LR]                         ;x2_im(m) = x1_im[m] + x1_im[m+offs]

          sub            R6,R10,R5
          strh           R6,[R0,#2]!                        ;x2_re(m+offs) = x1_re[m] -
x1_re[m+offs]
          sub            R6,R11,R7
          strh           R6,[R0,LR]                         ;x2_im(m+offs) = x1_im[m] -
x1_im[m+offs]
          subs           R12,R12,#1
          bne            stage1
;-------------------------------------------------------------------------------
; STAGE 2
;-------------------------------------------------------------------------------
          ldmfd    SP,{R0-R2}
          mov      R12,#N/4
          sub            R0,R0,#2
stage2:
          ldrsh    R4,[R0,#2]!                        ;load x2_re[m]
          ldrsh          R7,[R0,LR]              ;load x2_im[m]
          ldrsh          R5,[R0,#4]!                        ;load x2_re[m+2]
          ldrsh          R8,[R0,LR]                         ;load x2_im[m+2]
          add            R6,R4,R5
          strh           R6,[R0,#-4]!                   ;x2_re(m) = x2_re[m] + x2_re[m+2]
          add            R6,R7,R8
          strh           R6,[R0,LR]                         ;x2_im(m) = x2_im[m] + x2_im[m+2]
          sub      R6,R4,R5
          strh           R6,[R0,#4]!                        ;x2_re(m+2) = x2_re[m] - x2_re[m+2]
          sub            R6,R7,R8
          strh           R6,[R0,LR]                     ;x2_im(m+2) = x2_im[m] - x2_im[m+2]
          ldrsh          R4,[R0,#-2]!               ;load x2_re[m+1]
          ldrsh          R6,[R0,LR]                 ;load x2_im[m+1]
          ldrsh          R7,[R0,#4]!                    ;load x2_re[m+3]
          ldrsh          R5,[R0,LR]                     ;load x2_im[m+1]

          add            R8,R4,R5
```

```
        strh        R8,[R0,#-4]!                ;x2_re(m+1) = x2_re[m+1] + x2_re[m+3]
        sub         R8,R6,R7
        strh        R8,[R0,LR]                  ;x2_im(m+1) = x2_im[m+1] - x2_im[m+3]
        sub         R8,R4,R5
        strh        R8,[R0,#4]!                    ;x2_re(m+3) = x2_re[m+1] - x2_re[m+3]
        add         R8,R6,R7
        strh        R8,[R0,LR]                     ;x2_im(m+1) = x2_im[m+1] + x2_im[m+3]
        subs        R12,R12,#1
        bne         stage2
        ldmfd   SP,{R0-R2}
;-------------------------------------------------------------------------------
; STAGE 3
;-------------------------------------------------------------------------------
        .if N>8
        stage    8, 4, N/8, N/8, (N-N/4)

        ldmfd   SP,{R0-R2}
        .endif
;-------------------------------------------------------------------------------
; STAGE 4
;-------------------------------------------------------------------------------
       .if N>16
        stage    16, 8, N/16, N/16, (N-N/8)

       ldmfd   SP,{R0-R2}
       .endif
;-------------------------------------------------------------------------------
; STAGE 5
;-------------------------------------------------------------------------------
      .if N>32
      stage    32, 16, N/32, N/32, (N-N/16)

      ldmfd   SP,{R0-R2}
      .endif
;-------------------------------------------------------------------------------
; STAGE 6
;-------------------------------------------------------------------------------
      .if N>64
      stage    64, 32, N/64, N/64, (N-N/32)

      ldmfd   SP,{R0-R2}
      .endif
;-------------------------------------------------------------------------------
;LAST STAGE (THE SQUARED MAGNITUDE IS ALREADY CALCULATED IN THIS STAGE)
;-------------------------------------------------------------------------------
       mov          R3,#N/2
       add          R0,R0,#-2                    ;align address
       sub          R2,R2,#2                         ;align twiddle
       ldr          R1,[SP,#44]                   ;&result[0]
       sub          R1,R1,#2
lstage:
       ldrsh        R4,[R0,#-(-2)]!            ;x_re(m)
       ldrsh        R5,[R0,LR]                        ;x_im(m)
       ldrsh        R6,[R0,#N]!                       ;x_re(m+32)
       ldrsh        R7,[R0,LR]                        ;x_im(m+32)
       ldrsh        R8,[R2,#2]!                       ;wi sin(k+1)
       ldrsh        R9,[R2,#(N/4)*2]            ;wr cos(k+1)
       mul        R12,R9,R6                 ;x_re(m+32) * cos(k+1)
       mul        R11,R8,R7                 ;x_im(m+32) * sin(k+1)
       add        R10,R11,R12               ;x_re(m+32) * cos(k+1) + x_im(m+32) * sin(k+1)
       mov        R10,R10, LSL#1
       mov        R10,R10, ASR#16
       add        R10,R4,R10                ;x_re(m) + x_re(m+32) * cos(k+1) + x_im(m+32) *
sin(k+1)
       strh       R10,[R0,#-N]!          ;x_re(m) = x_re(m) + [x_re(m+32)*cos(k+1) +
x_im(m+32)*sin(k+1)]
```

```
        add         R12,R12,R11                 ;x_re(m+32) * cos(k+1) + x_im(m+32) * sin(k+1)
        mov         R12,R12, LSL#1
        mov         R12,R12, ASR#16
        sub         R4,R4,R12                   ;x_re(m) - [x_re(m+32) * cos(k+1) - x_im(m+32) *
sin(k+1)]
        mul         R7,R9,R7                       ;x_im(m+32) * cos(k+1)
        mul         R9,R8,R6                       ;x_re(m+32) * sin(k+1)
        sub         R8,R7,R9                       ;x_im(m+32) * cos(k+1) - x_re(m+32) * sin(k+1)
        mov         R8,R8, LSL#1
        mov         R8,R8, ASR#16
        add         R8,R5,R8                       ;x_im(m) + x_im(m+32) * cos(k+1) - x_re(m+32)
* sin(k+1)
        strh        R8,[R0,LR]                  ;x_im(m) = x_re(m) + [x_im(m+32)*cos(k+1) -
x_re(m+32)*sin(k+1)]
        strh        R4,[R0,#N]!              ;x_re(m+32) = x_re(n) - [x_re(m+32)*cos(k+1) -
x_im(m+32)*sin(k+1)]
        mul         R6,R10,R10
        mla         R6,R8,R8,R6
        mov         R6,R6,LSR#15
        strh        R6,[R1,#2]!                    ; store result[n]
        sub         R7,R7,R9                          ;x_im(m+32) * cos(k+1) - x_re(m+32) *
sin(k+1)
        mov         R7,R7, LSL#1
        mov         R7,R7, ASR#16
        sub         R7,R5,R7                    ;x_im(m) - x_im(m+32) * cos(k+1) + x_re(m+32) *
sin(k+1)
sin(k+1)
        strh        R7,[R0,LR]               ;x_im(m+32) = x_im(n) - [x_im(m+32)*cos(k+1) +
x_re(m+32)*sin(k+1)]
        mul         R6,R4,R4
        mla         R6,R7,R7,R6
        mov         R6,R6,LSR#15
        strh        R6,[R1,#N]                     ; store result[n+32]
        subs        R3,R3,#1
        bne         lstage
        ldmfd    SP!,{R0-R2}
        ldmfd    SP!,{R4-R12,PC}^
Intvecs.asm
                    .state32
                    .global _c_int00
                    .sect     ".intvecs"
                    b    _c_int00                        ;  RESET INTERRUPT
                    b    _c_int00                        ;  RESET INTERRUPT          b    #-
8                        ;  UNDEFINED INSTRUCTION INTERRUPT
                    b    #-8                                ;  SOFTWARE INTERRUPT
                    b    #-8                                ;  ABORT (PREFETCH) INTERRUPT
                    b    #-8                                ;  ABORT (DATA) INTERRUPT
                    b    #-8                                ;  RESERVED
                    b    #-8                                ;  IRQ INTERRUPT
                    b    #-8                                ;  FIQ INTERRUPT
                    .end
```

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

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

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