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
1 Introduction ......................................................................................................................... 1
2 TMS470R1x Architecture .................................................................................................... 2
3 Fourier Transform ................................................................................................................ 2
4 FFT .................................................................................................................................... 2
5 Implementation ................................................................................................................... 6
6 Results ................................................................................................................................. 7
Appendix A Creating a COFF ................................................................................................ 8

List of Figures
1 Twiddle Factors (W8) ......................................................................................................... 3
2 8-Point DIT-Radix2-FFT .................................................................................................... 5
3 Examples of Bit-Reversed Address .................................................................................... 6

List of Tables
1 Radix-2 FFT Algorithm Results ....................................................................................... 7
A-1 Files Needed to Create a COFF ....................................................................................... 8

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):

\[ S(f) = \int_{-\infty}^{\infty} s(t) e^{-j2\pi ft} \, dt \]  

(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:

\[ S(k) = \sum_{i=0}^{N-1} s(t_i) e^{-j2\pi ik} \left( t_i + 1 - t_i \right) \quad k = 0, 1, \ldots, N - 1 \]  

(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 N^2. 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.

\[ W_N = e^{-j2\pi i/N} \]  

(3)

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

\[ X(k) = \sum_{n=0}^{N-1} W_N^{mn} X[n] \quad m = 0, 1, \ldots, N - 1 \]  

[16, p 106]  

(4)

Figure 1 explains the cyclic features (twiddle factors) of the exponential function.
Explicitly, the twiddle factors translate into the following:

\[ W_{0/8} = W_{8}^{8} = \cos(0^\circ) - j \sin(0^\circ) = 1 - j0 \]

\[ W_{1/8} = W_{9}^{9} = \cos(45^\circ) - j \sin(45^\circ) = 0.7 - j0.7 \]

\[ W_{2/8} = W_{10}^{10} = \cos(90^\circ) - j \sin(90^\circ) = 0 - j1 \]

\[ W_{3/8} = W_{11}^{11} = \cos(135^\circ) - j \sin(135^\circ) = 0.7 - j0.7 \]

\[ W_{4/8} = W_{12}^{12} = \cos(180^\circ) - j \sin(180^\circ) = -1 - j0 \]

\[ W_{5/8} = W_{13}^{13} = \cos(225^\circ) - j \sin(225^\circ) = -0.7 - j0.7 \]

\[ W_{6/8} = W_{14}^{14} = \cos(270^\circ) - j \sin(270^\circ) = 0 + j1 \]

\[ W_{7/8} = W_{15}^{15} = \cos(315^\circ) - j \sin(315^\circ) = 0.7 - j0.7 \]

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

**Equation 4** can be rewritten as follows:

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

(5)

\[ X_m(k) = \sum_{n=0}^{P-1} a[n]W_P^{mn} \quad m = 0, 1, ..., P - 1 \]

(6)
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:

\[
x_{\text{re}}(n,s+1) = x_{\text{re}}(n,s) + x_{\text{re}}(n+t,s) \cdot \cos(x) + x_{\text{im}}(n+t,s) \cdot \sin(x)
\]

(9)

\[
x_{\text{im}}(n,s+1) = x_{\text{im}}(n,s) + x_{\text{im}}(n+t,s) \cdot \cos(x) - x_{\text{re}}(n+t,s) \cdot \sin(x)
\]

(10)

\[
x_{\text{re}}(n+t,s+1) = x_{\text{re}}(n,s) - x_{\text{re}}(n+t,s) \cdot \cos(x) - x_{\text{im}}(n+t,s) \cdot \sin(x)
\]

(11)

\[
x_{\text{im}}(n+t,s+1) = x_{\text{im}}(n,s) - x_{\text{im}}(n+t,s) \cdot \cos(x) + x_{\text{re}}(n+t,s) \cdot \sin(x)
\]

(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., \( x(t) \rightarrow X(j\omega) \))

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.

<table>
<thead>
<tr>
<th>Odd number of bits</th>
<th>Even number of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

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

Bit 1 is in the middle and remains unchanged.

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 \( 8 \leq N \leq 128 \) 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>
<thead>
<tr>
<th></th>
<th>8-Point FFT</th>
<th>16-Point FFT</th>
<th>64-Point FFT</th>
<th>128-Point FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code size(1)</td>
<td>396 Byte</td>
<td>580 Byte</td>
<td>948 Byte</td>
<td>1132 Byte</td>
</tr>
<tr>
<td>Table size(2)</td>
<td>36 Byte</td>
<td>72 Byte</td>
<td>144 Byte</td>
<td>288 Byte</td>
</tr>
<tr>
<td>Array size(3)</td>
<td>80 Byte</td>
<td>160 Byte</td>
<td>320 Byte</td>
<td>640 Byte</td>
</tr>
<tr>
<td>Cycle count(4)</td>
<td>592</td>
<td>1573</td>
<td>9563</td>
<td>22508</td>
</tr>
</tbody>
</table>

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

These files are available as sources.

A.2 Example Program

The following is an example program.

```c
#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;
    }
}
```
Example Program

```c
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();
    }
}
```

Rad2.asm

;**************************************************************
;*  Optimized assembler program for Radix-2 FFT algorithm*
;**************************************************************
Example Program

.globl _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
; ali defines the alignment of the sine table pointer after each block

stage .macro off, butt, block, sioff, ali
move R1, #block
move 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)

* sin(k+8)

strh R10, [R0, #-(off-2)] ;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)]

* 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-3] ;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

Implementing Radix-2 FFT Algorithms on the TMS470R1x

SPNA071A–November 2006
Submit Documentation Feedback
Example Program

```
movne R3,#but
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_?

;----end of macro definition
.endm

.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/4
sub R0,R0,#2

stagel:

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) = x1_im[m] - x1_im[m+offs]

sub R12,R12,#1
bne stagel

;------------------------------------------------------------------------------
; 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
```
Example Program

```
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

.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)]
```

Implementing Radix-2 FFT Algorithms on the TMS470R1x

SPNA071A—November 2006
Submit Documentation Feedback
Example Program

```
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)
sin(k+1)
sub R8, R7, R9 ; x_im(m+32) * cos(k+1) - x_re(m+32) *
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 ; store result[n]
mla R6, R6, R8, R6
mov R6, R6, LSR#15
strh R6, [R1, #2]! ; 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 ; store result[n+32]
mla R6, R6, R7, R6
mov R6, R6, LSR#15
strh R6, [R1, #N] bne lstage
subs R3, R3, 1
lmdfd SP!, (R0-R2)
lmdfd SP!, (R4-R12, PC)^

Intvecs.asm

.state32
.global _c_int00
.sect "*.intvecs"
b _c_int00 ; RESET INTERRUPT
b _c_int00 ; RESET INTERRUPT
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
```
IMPORTANT NOTICE

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Products</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amplifiers</td>
<td>amplifier.ti.com</td>
</tr>
<tr>
<td>Data Converters</td>
<td>dataconverter.ti.com</td>
</tr>
<tr>
<td>DSP</td>
<td>dsp.ti.com</td>
</tr>
<tr>
<td>Interface</td>
<td>interface.ti.com</td>
</tr>
<tr>
<td>Logic</td>
<td>logic.ti.com</td>
</tr>
<tr>
<td>Power Mgmt</td>
<td>power.ti.com</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Copyright © 2006, Texas Instruments Incorporated