Application Report
SPRA291 - August 2001

Implementing Fast Fourier Transform Algorithms of Real-Valued Sequences With the TMS320 DSP Platform

Robert Matusiak *Digital Signal Processing Solutions*

ABSTRACT

The Fast Fourier Transform (FFT) is an efficient computation of the Discrete Fourier Transform (DFT) and one of the most important tools used in digital signal processing applications. Because of its well-structured form, the FFT is a benchmark in assessing digital signal processor (DSP) performance.

The development of FFT algorithms has assumed an input sequence consisting of complex numbers. This is because complex phase factors, or twiddle factors, result in complex variables. Thus, FFT algorithms are designed to perform complex multiplications and additions. However, the input sequence consists of real numbers in a large number of real applications.

This application report discusses the theory and usage of two algorithms used to efficiently compute the DFT of real-valued sequences as implemented on the Texas Instruments TMS320C6000™.

The first algorithm performs the DFT of two *N*-point real-valued sequences using one *N*-point complex DFT and additional computations.

The second algorithm performs the DFT of a *2N*-point real-valued sequence using one *N*-point complex DFT and additional computations. Implementations of these additional computations, referred to as the split operation, are presented both in C and C6000™ assembly language. For implementation on the C6000, optimization techniques in both C and assembly are covered.

Contents

List of Figures

List of Tables

List of Examples

1 Introduction

TI's C6000 platform of high-performance, fixed-point DSPs provides architectural and speed improvements that makes the FFT computation faster and easier to program than other fixed-point DSPs.

The C6000 platform devices are based on an advanced Very Long Instruction Word (VLIW) central processing unit (CPU) with eight functional units that include two multipliers and six arithmetic logic units (ALUs). The CPU can execute up to eight instructions per cycle. Complementing the architecture is a very efficient C compiler that increases performance and reduces code development time. The C6000 architecture and development tools featured are discussed in this application report along with the following topics:

- Theory of DFTs of real-valued sequences
- Algorithm implementation
- C6000 CPU features
- C6000 development tools
- Optimizing C code for the C6000
- C-callable assembly language functions for the C6000

2 Basics of the DFT and FFT

Methods of performing the DFT of real sequences involve complex-valued DFTs. This section reviews the basics of the DFT and FFT.

The DFT is viewed as a frequency domain representation of the discrete-time sequence $x(n)$. The N-point DFT of finite-duration sequence $x(n)$ is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \qquad l = 0, 1, \ldots, N-1 \tag{1}$$

and the inverse DFT (IDFT) is defined as

$$X(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)W_N^{-kn} \qquad n = 0,1,\ldots,N-1 \tag{2}$$

where

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

The W_N^{kn} factor is also referred to as the twiddle factor. Observation of the above equations shows that the computational requirements of the DFT increase rapidly as the number of samples in the sequence *N* increases. Because of the large computational requirements, direct implementation of the DFT of large sequences has not been practical for real-time applications. However, the development of fast algorithms, known as FFTs, has made implementation of DFT practical in real-time applications.

The definition of FFT is the same as DFT, but the method of computation differs. The basics of FFT algorithms involve a divide-and-conquer approach in which an *N*-point DFT is divided into successively smaller DFTs. Many FFT algorithms have been developed, such as radix-2, radix-4, and mixed radix; in-place and not-in-place; and decimation-in-time and decimation-in-frequency.

In most FFT algorithms, restrictions may apply. For example, a radix-2 FFT restricts the number of samples in the sequence to a power of two.

In addition, some FFT algorithms require the input or output to be re-ordered. For example, the radix-2 decimation-in-frequency algorithm requires the output to be bit-reversed. It is up to implementers to choose the FFT algorithm that best fits their application.

Table 1 compares the number of math computations involved in direct computation of the DFT versus the radix-2 FFT algorithm. As you can see, the speed improvement of the FFT increases as *N* increases. Detailed descriptions of the DFT and FFT can be found in the references.[1,2,3] The following sections describe methods of efficiently computing the DFT of real-valued sequences using complex-valued DFTs/IDFTs.

Table 1. Comparison of Computational Complexity for Direct Computation of the DFT Versus the Radix-2 FFT Algorithm

	Direct Computation of the DFT		Radix-2 FFT	
Number of Points	**Complex Multiplies**	**Complex Additions**	**Complex Multiplies**	**Complex Additions**
N	N^2	N^2–*N*	$(N/2)\log_2 N$	$N\log_2 N$
4	16	12	4	8
16	256	240	32	64
64	4096	4032	192	384
256	65536	65280	1024	2048
1024	1048576	1047552	5120	10240

3 Efficient Computation of the DFT of Real Sequences

In many real applications, the data sequences to be processed are real-valued. Even though the data is real, complex-valued DFT algorithms can still be used. One simple approach creates a complex sequence from the real sequence; that is, real data for the real components and zeros for the imaginary components. The complex DFT can then be applied directly. However, this method is not efficient. This section shows you how to use the complex-valued DFT algorithms to efficiently process real-valued sequences.

3.1 Efficient Computation of the DFT of Two Real Sequences

Suppose $x_1(n)$ and $x_2(n)$ are real-valued sequences of length N, and $x(n)$ is a complex-valued sequence defined as

$$x(n) = x_1(n) + jx_2(n) \qquad 0 \le n \le N-1 \tag{4}$$

The DFT of the two N-length sequences $x_1(n)$ and $x_2(n)$ can be found by performing a single N-length DFT on the complex-valued sequence and some additional computation. These additional computations are referred to as the split operation, and are shown below.

$$\begin{aligned} X_1(k) &= \frac{1}{2}[X(k) + X^*(N-k)] \\ X_2(k) &= \frac{1}{2j}[X(k) - X^*(N-k)] \end{aligned} \qquad k = 0, 1, \ldots, N-1 \tag{5}$$

As you can see from the above equations, the transforms of $x_1(n)$ and $x_2(n)$, $X_1(k)$ and $X_2(k)$, respectively, are solved by computing one complex-valued DFT, $X(k)$, and some additional computations.

Now assume we want to get back $x_1(n)$ and $x_2(n)$ from $X_1(k)$ and $X_2(k)$, respectively. As with the forward DFT, the IDFT of $X_1(k)$ and $X_2(k)$ is found using a single complex-valued DFT. Because the DFT operation is linear, the DFT of equation (4) can be expressed as

$$X(k) = X_1(k) + jX_2(k) \tag{6}$$

This shows that $X(k)$ can be expressed in terms of $X_1(k)$ and $X_2(k)$; thus, taking the inverse DFT of $X(k)$, we get $x(n)$, which gives us $x_1(n)$ and $x_2(n)$.

The above equations require complex arithmetic not directly supported by DSPs; thus, to implement these complex-valued equations, it is helpful to express the real and imaginary terms in real arithmetic. The forward DFT of the equations shown in (5) can be written as follows:

$$\begin{aligned} X_1 r(k) &= \frac{1}{2}[Xr(k) + Xr(N-k)] \text{ and } X_1 i(k) = \frac{1}{2}[Xi(k) - Xi(N-k)] \\ X_2 r(k) &= \frac{1}{2}[Xi(k) + Xi(N-k)] \text{ and } X_2 i(k) = \frac{-1}{2}[Xr(k) - Xr(N-k)] \end{aligned} \qquad k = 0, 1, \ldots, N-1 \tag{7}$$

In addition, because the DFT of real-valued sequences has the properties of complex conjugate symmetry and periodicity, the number of computations in (7) can be reduced. Using the properties, the equations in (7) can be rewritten as follows:

$$X_1r(0) = Xr(0) \qquad X_1i(0) = 0$$

$$X_2r(0) = Xi(0) \qquad X_2i(0) = 0$$

$$X_1r(N/2) = Xr(N/2) \qquad X_1i(N/2) = 0$$

$$X_2r(N/2) = Xi(N/2) \qquad X_2i(N/2) = 0 \tag{8}$$

$$X_1\,r(k) = \frac{1}{2}[Xr\,(k) + Xr(N-k)] \qquad X_1\,i(k) = \frac{1}{2}[Xi\,(k) - Xi(N-k)]$$

$$X_2\,r(k) = \frac{1}{2}[Xi\,(k) + Xi(N-k)] \qquad X_2\,i(k) = \frac{-1}{2}[Xr\,(k) - Xr(N-k)]$$

$$X_1r(N-k) = X_1r(k) \qquad X_1i(N-k) = -X_1i(k)$$

$$X_2r(N-k) = X_2r(k) \qquad X_2i(N-k) = -X_2i(k)$$

Similarly, the additional computation involved in computing the IDFT can be written as follows:

$$Xr(k) = X_1r(k) - X_2i(k)$$

$$k = 0, 1, \ldots, N-1 \tag{9}$$

$$Xi(k) = X_1i(k) + X_2r(k)$$

See Appendix A for a detailed derivation of these equations.

Now that we have the equations for the split operation used in computing the DFT of two real-valued sequences, we turn to the following steps, which outline how to use the equations. The forward DFT is outlined as follows.

Step 1: Form the N-point complex-valued sequence $x(n)$ from the two N-length sequences $x_1(n)$ and $x_2(n)$.

for n=0, …, N–1
$x_r(n) = x_1(n)$
$x_i(n) = x_2(n)$

Step 2: Compute the N-length complex DFT of $x(n)$.

X(k) = DFT[x(n)]

NOTE: The DFT can be any efficient DFT algorithm (such as one of the various FFT algorithms), but the output must be in normal order.

Step 3: Compute the split operation equations.

$X_1r(0) = Xr(0)$ $\quad X_1i(0) = 0$
$X_2r(0) = Xi(0)$ $\quad X_2i(0) = 0$
$X_1r(N/2) = Xi(N/2)$ $\quad X_1i(N/2) = 0$
$X_2r(N/2) = Xi(N/2)$ $\quad X_2i(N/2) = 0$

for k=1, …, N/2–1

$X_1r(k) = 0.5 * [Xr(k) + Xr(N - k)]$ $\quad X_1i(k) = 0.5 * [Xi(k) - Xi(N - k)]$
$X_2r(k) = 0.5 * [Xi(k) + Xi(N - k)]$ $\quad X_2i(k) = 0.5 * [Xr(k) - Xr(N - k)]$
$X_1r(N - k) = X_1r(k)$ $\quad X_1i(N - k) = - X_1i(k)$
$X_2r(N - k) = X_2r(k)$ $\quad X_2i(N - k) = - X_2i(k)$

For two frequency domain sequences, $X_1(k)$ and $X_2(k)$, derived from real-valued sequences, perform the following steps to take the IDFT of $X_1(k)$ and $X_2(k)$.

Step 1: Form a single complex-valued sequence $X(k)$ from $X_1(k)$ and $X_2(k)$ using the IDFT split equations.

for k=0, …, N–1

$Xr(k) = X_1r(k) - X_2i(k)$
$Xi(k) = X_1i(k) + X_2r(k)$

Step 2: Compute the N-length IDFT of $X(k)$.

$x(n) = \text{IDFT}[X(k)]$

As with the forward DFT, the IDFT can be any efficient IDFT algorithm. The IDFT can be computed using the forward DFT and some conjugate operations.

$x(n) = [\text{DFT}\{X^*(k)\}]^*$

where:

* is the complex conjugate operator

Step 3: From $x(n)$, form $x_1(n)$ and $x_2(n)$.

for n = 0, 1, …. N–1

$x_1(n) = xr(n)$
$x_2(n) = xi(n)$

Appendix C contains C implementation of the outlined DFT and IDFT algorithms.

3.2 Efficient Computation of the DFT of a 2*N*-Point Real Sequence

Assume $g(n)$ is a real-valued sequence of 2N points. We outline the equations involved in obtaining the 2N-point DFT of *g(n)* from the computation of one N-point complex-valued DFT. First, we subdivide the 2N-point real sequence into two N-point sequences as follows:

And define $x(n)$ to be the N-point complex-valued sequence:

$$x_1(n) = g(2n)$$
$$x_2(n) = g(2n + 1) \qquad 0 \le n \le N - 1 \qquad (10)$$

The DFT of g(n), G(k), can be computed using

$$x(n) = x_1(n) + jx_2(n) \qquad 0 \le n \le N-1 \tag{11}$$

where

$$G(k) = X(k)A(k) + X^*(N-k)B(k) \qquad k = 0,1,\ldots,N-1 \quad \text{with } X(N) = X(0) \tag{12}$$

$$A(k) = \frac{1}{2}\left(1 - jW_{2N}^{k}\right) \quad \text{and} \quad B(k) = \frac{1}{2}\left(1 + jW_{2N}^{k}\right) \tag{13}$$

As you can see, we have computed the DFT of a 2*N*-point sequence from one *N*-point DFT and additional computations, which we call the split operation.

Similarly, if we have a frequency domain 2*N*-point sequence, which was derived from a real-valued sequence, we can use an *N*-point IDFT to obtain the time domain 2*N*-point real-valued sequence using the following equation:

$$X(k) = G(k)A^*(k) + G^*(N-k)B^*(k) \qquad k = 0,1,\ldots,N-1 \quad \text{with } G(N) = G(0) \tag{14}$$

The equations shown in (12) and (14) are of the same form. Equation (14) can be obtained from equation (12) if G(k) is swapped with X(k), and A(k) and B(k) are complex conjugated. Thus, equations (12) and (14) can be implemented with one common split function.

NOTE: In implementing these equations, *A*(*k*), *A**(*k*), *B*(*k*), and *B**(*k*) can be pre-computed and stored in a table. Their values can thus be obtained by table look-up as opposed to arithmetic computation. The result is a large computational savings because the sine and cosine functions required by twiddle factors do not need to be computed when performing the split. (A detailed derivation of these equations is provided in Appendix B.)

As in the previous section, Efficient Computation of the DFT of Two Real Sequences, when implementing the above equations, it is useful to express them in their real and imaginary terms.

Only *N* points of *G(k)* are computed in equation (12) because other *N* points can be found using the complex conjugate property. This is applied to the following equation.

$$\begin{aligned} &Gr(k) = Xr(k)\,Ar(k) - Xi(k)\,Ai(k) + Xr(N-k)\,Br(k) + Xi(N-k)\,Bi(k) \\ &Gi(k) = Xi(k)\,Ar(k) + Xr(k)\,Ai(k) + Xr(N-k)\,Bi(k) - Xi(N-k)\,Br(k) \\ &Gr(2N-k) = Gr(k) \\ &Gi(2N-k) = -Gi(k) \end{aligned} \qquad k = 0,1,\ldots,N-1 \quad \text{with } X(N) = X(0) \tag{15}$$

$$Gr(N) = Gr(0) - Gi(0)$$
$$Gi(N) = 0$$

As with the forward DFT, the equations for the IDFT can be expressed in their real and imaginary terms as follows.

$$Xr(k) = Gr(k)\,Ar(k) - Gi(k)\,(-Ai(k)) + Gr(N-k)\,Br(k) + Gi(N-k)\,Bi(k)$$
$$Xi(k) = Gi(k)\,Ar(k) + GXr(k)\,(-Ai(k)) + Gr(N-k)\,(-Bi(k)) - Gi(n-k)\,Br(k) \quad (16)$$
$$k = 0, 1, \ldots, N-1$$
$$\text{with } G(N) = G(0)$$

Now that we have the equations for the split operation to compute the DFT of a real-valued sequence, the steps for using these equations are outlined. The forward DFT is outlined first.

Step 1: Initialize $A(k)$s and $B(k)$s.

Real applications usually perform this only once during a power-up or initialization sequence. These values can be pre-stored in a boot ROM or computed. In either case, once they are generated, this step is no longer needed when performing the DFT. The pseudo code for generating them is given below.

for k = 0, 1, …., N–
$Ai(k) = -\cos(\pi k/N)$
$Ar(k) = -\sin(\pi k/N)$
$Bi(k) = \cos(\pi k/N)$
$Br(k) = \sin(\pi k/N)$

Step 2: Let $g(n)$ be a $2N$-point real sequence. From $g(n)$, form the N-point complex-valued sequence.

$x(n) = x_1(n) + jx_2(n)$

where

$x_1(n) = g(2n)$
$x_2(n) = g(2n + 1)$

for n = 0, 1, …., N–1
$xr(n) = g(2n)$
$xi(n) = g(2n + 1)$

Step 3: Perform an N-point complex FFT on the complex-valued sequence $x(n)$.

$X(k) = DFT[x(n)]$

NOTE: The FFT can be any DFT method, such as radix-2, radix-4, mixed radix, direct implementation of the DFT, etc. However, the DFT output must be in normal order.

Step 4: Implement the split operation equations.

$X(N) = X(0)$
$Gr(N) = Gr(0) – Gr(0)$
$Gi(N) = 0$

for k=0,1,…,N–1
$Gr(k) = Xr(k)\ Ar(k) – Xi(k)\ Ai(k) + Xr(N–k)\ Br(k) + Xi(N–k)\ Bi(k)$
$Gi(k) = Xi(k)\ Ar(k) + Xr(k)\ Ai(k) + Xr(N–k)\ Bi(k) – Xi(N–k)\ Br(k)$
$Gr(2N–k) = Gr(k)$
$Gi(2N–k) = –Gi(k)$

For a 2N-point frequency domain sequences $G(k)$ derived from a 2N-point real-valued sequences, perform the following steps for the IDFT of $G(k)$.

Step 1: Initialize $A^*(k)$s and $B^*(k)$s.

As with the forward DFT, this step is usually performed only once during a power-up or initialization sequence. The values can be pre-stored in a boot ROM or computed. In either case, once the values are generated, this step is no longer needed when performing the DFT.

Because $A^*(k)$ and $B^*(k)$ are the complex conjugates of A(k) and B(k), respectively, each can be derived from the $A(k)$s and $B(k)$s. The following pseudo code is used to generate them.

for k = 0, 1, …., N–1
$A^*i(k) = \cos(\pi k/N)$
$A^*r(k) = 1 – \sin(\pi k/N)$
$B^*i(k) = – \cos(\pi k/N)$
$B^*r(k) = 1 + \sin(\pi k/N)$

or, if $A(k)$ and $B(k)$ are already generated, you can use the following pseudo code:

for k = 0, 1, …., N–1
$A^*i(k) = –Ai(k)$
$A^*r(k) = Ar(k)$
$B^*i(k) = – Bi(k)$
$B^*r(k) = Br(k)$

Step 2: Let $G(k)$ be a 2N-point complex-valued sequence derived from a real-valued sequence $g(n)$.

We want to get back $g(n)$ from $G(k) \rightarrow g(n)$ = IDFT[$G(k)$]. However, we want to apply the same techniques we applied with the forward DFT, that is, use an N-point IFFT. This can be accomplished using the following equations.

$G(N) = G(0)$

for k = 0,1,…,N–1
$Xr(k) = Gr(k)\ Ar(k) – Gi(k)(–Ai(k)) + Gr(N–k)\ Br(k) + Gi(N–k)\ (–Bi(k))$
$Xi(k) = Gi(k)\ Ar(k) + Gr(k)\ (–Ai(k)) + Gr(N–k)\ (–Bi(k)) – Gi(N–k)Br(k)$

Step 3: Perform the *N*-point inverse DFT of $X(k)$.

$$x(n) = x_1(n) + jx_2(n) = \text{IDFT}[X(k)]$$

NOTE: The IDFT can be any method but must have an output in normal order.

Step 4: $g(n)$ can then be found from $x(n)$.

for n = 0, 1, …., N

$$g(2n) = x_1(n)$$
$$g(2n+1) = x_2(n)$$

Appendix C contains C implementations of the outlined DFT and IDFT algorithms.

4 TMS320C62x™ Architecture and Tools Overview

Before we discuss how to efficiently implement the real-valued FFT algorithms on the C62x™, it is helpful to take a brief look at the C62x architecture and code development tools. The TMS320C62x fixed-point processors are based on a 256-bit advanced VLIW CPU with eight functional units, including two multipliers and six arithmetic and logic units (ALUs). The CPU can execute up to eight 32-bit instructions per cycle. With an instruction clock frequency of 200 MHz and greater, C62x peak performance starts at 1600 million instructions per second (MIPs).

The C62x processor consists of three main parts:

- CPU
- Peripherals
- Memory

Figure 1 shows a block diagram of the first device in this generation, the TMS320C6201 DSP.

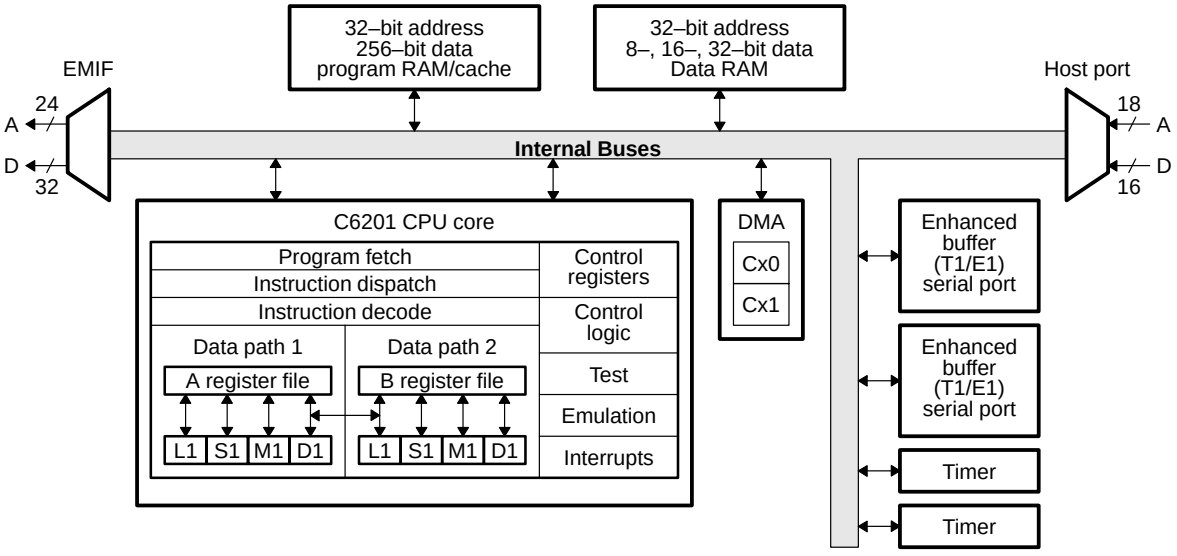

Figure 1. TMS320C6201 DSP Block Diagram

TMS320C62x and C62x are trademarks of Texas Instruments.

This discussion focuses on the CPU, or *core*, of the device. The C62x CPU is the central building block of all TMS320C62x devices and features two data paths where processing occurs. Each data path has four functional units (.L, .S, .M, .D), along with a register file containing 16 32-bit general-purpose registers.

The C62x is a load-store architecture in which all functional units obtain operands from a register file, rather than directly from memory.

The .D units load/store data from and to memory from the register file with an address reach of 32 bits. The C62x architecture is also byte addressable: the .D units can load or store data in either 8 bits (byte), 16 bits (half-word), or 32 bits (word). In addition, the .D units can perform 32-bit addition and subtraction and address calculations.

The .M units perform multiplication, featuring a 16-bit by 16-bit multiplier that produces a 32-bit result. Additional multiplier features include the ability to select either the 16 most significant bits (MSBs) or 16 least significant bits (LSBs) of a register operand, and optionally left-shift the multiplier output by one with saturation.

The .S units perform branches and shifting primarily, but also perform bit field operations such as extract, set and clear bit fields, as well as 32-bit logical operations and 32-bit addition and subtraction. Another advanced feature of each .S unit is the ability to split its ALU to perform two 16-bit adds or subtracts in a single cycle.

Although the .S and .D units perform ALU functions, the .L unit is the main ALU for the CPU, performing both 32-bit and 40-bit integer arithmetic. The .L unit also features saturation logic, comparison instructions, and bit counting and can perform 32-bit logical operations. In support of the eight functional units, the CPU has a program fetch unit; instruction dispatch unit; instruction decode unit; control registers; control logic; and test, emulation and interrupt logic.

The C62x features a state of the art software development environment. A very efficient C compiler, along with a linear assembly optimizer, allows fast time to market through ease of use. Its orthogonal reduced instruction set computing (RISC)-like CPU architecture makes the C62x CPU a very good C-compiler target. Combined with TI's compiler expertise, these features make the C62x compiler the most efficient DSP compiler on the market today.

Because of its efficiency, most C62x coding can be done in C. However, as with many other DSPs, some tasks or routines require assembly coding to achieve the highest performance possible.

As a result, TI has developed a new tool called the assembly optimizer that makes assembly language coding easier and faster. The assembly optimizer allows you to write linear assembly code (no parallel instructions) without assigning registers to operands.

The assembly optimizer accepts this input syntax and generates an assembly language output that parallelizes the linear instructions and assigns registers to operands. This relieves the assembly language programmer of the following responsibilities:

- Determining which instructions can be executed in parallel
- Knowing how to position code to avoid delay slot conflicts
- Keeping track of which registers are live or free

The C62x assembler is also included in the code development tool set.

Figure 2 shows the process flow to develop code for the C62x.

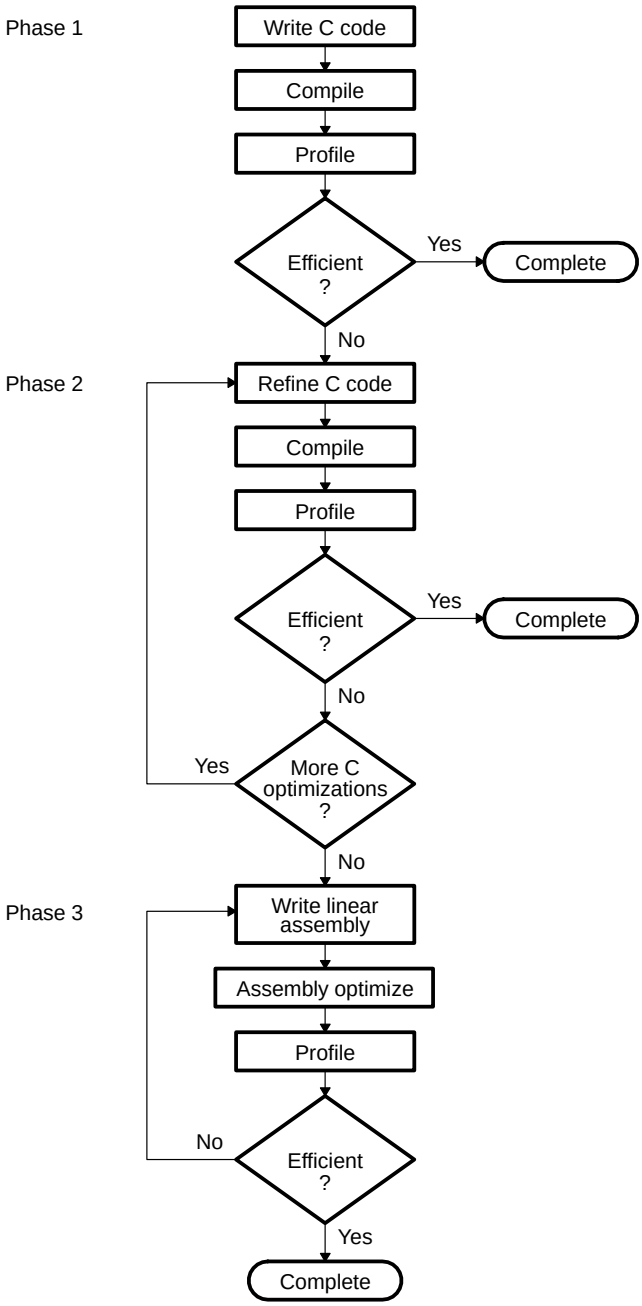

Figure 2. Code Development Flow Chart

In phase 1 of the code development process, TI recommends that the algorithm first be implemented in C, which serves the following purposes:

- Provides an easy way to verify the functionality of an algorithm
- Provides a working model to verify results of optimized versions
- May meet your efficiency requirements, and thus completes your implementation

If phase 1 fails to meet your performance requirements, you may need to proceed to phase 2 to refine and optimize your C code. The process includes modifying your C code for efficiency using the C code optimization methods. This section offers a brief overview of the C-code optimization methods. For a more detailed explanation, see the *TMS320C6000 Programmers Guide* (SPRU198).

One of the easiest methods used to optimize your C code is the C compilers' optimizer, evoked using compiler options. Some of the most commonly used optimizer options are: *–o3*, *–pm*, *–mt*, and *–x2*. See the *TMS320C6000 Optimizing C Compiler User's Guide* (SPRU187) for a list of available compiler optimization options and usage.

Other methods to optimize C code for efficiency involve modifying your C code. One very effective method uses compiler intrinsics – special functions that map directly to inlined C62x instructions. Intrinsic functions typically allow you to use a C62x specific feature that is not directly expressible in C, such as .L unit saturation.

Other effective optimization techniques include:

- Loop unrolling
- Software pipelining
- Trip count specification
- Using the *const* keyword to eliminate memory dependencies

All of these methods produce very efficient C code. Nevertheless, the compiler still may not produce the efficiency required. In this case, phase 3 may be required. Phase 3 uses the assembly optimizer and/or the assembler to generate C62x assembly code. By far, the easiest and recommended route is the assembly optimizer. The assembly optimizer usage is outlined in detail in the *TMS320C6000 Optimizing C Compiler User's Guide* (SPRU187). In addition, the *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186) outlines assembler usage.

A recommended approach for using either assembly method is to implement assembly routines as C-callable assembly functions.

NOTE: Use caution when implementing C-callable assembly routines so that you do not disrupt the C environment and cause a program to fail. The TI *TMS320C6000 Optimizing C Compiler User's Guide* (SPRU187) details the register, stack, calling, and return requirements of the C62x run-time environment. TI recommends that you read the material covering these requirements before implementing a C-callable assembly language function.

5 Implementation and Optimization of Real-Valued DFTs

Appendix C contains the source code listings for C implementation of the two efficient methods for performing the DFT of real-valued sequences outlined in this application report. Each implementation fits into phase 1 of the code development flow chart shown in Figure 2.

The primary purpose of this particular implementation is to verify the functionality of split operation algorithm implementations and provide a known good model to compare against optimized versions. Another benefit is that this implementation is generic C code and thus can be easily ported to other DSPs or CPUs featuring C compilers.

Because the primary focus of this application report is the split operations used in the efficient computation of DFTs, the C implementation is not efficient with respect to the other operations involved in the computation of real-valued DFTs. For example, the direct form of the DFT is implemented rather than a more computationally efficient FFT. Optimizing the C code to yield better performance is addressed in Appendix D.

Example 1 and Example 2 show the compiler usage for building the executable files for these implementations.

Example 1. Efficient Computation of the DFT of a 2N-Point Real Sequence

```
cl6x –g vectors.asm realdft1.c split1.c data1.c dft.c –z –o test1.out –l
rts6201.lib lnk.cmd
```

Example 2. Efficient Computation of the DFT of Two Real Sequences

```
cl6x –g vectors.asm realdft2.c split2.c data2.c dft.c –z –o test2.out –l
rts6201.lib lnk.cmd
```

The example compiler usage results in two executable files that can be loaded into the C62x device simulator and run:

- test1.out
- test2.out

The –g option used in the above compiler usage tells the compiler to build the code with debug information. This means that the compiler does not use the optimizer but allows the code to be easily viewed by the debugger.

First-time users of the C62x are encouraged to try different compiler options and compare the effects of each on code performance. For benchmarking code on the C62x debugger, see the *TMS320C6x C Source Debugger User's Guide* (SPRU188).

Appendix D contains the source code listings for optimized C implementations of the two efficient methods for performing the DFT of real-valued sequences outlined in this application report. These implementations apply to phase 2 of the code development flowchart shown in .

For this implementation, the C code is refined to yield better performance. Not all C optimization techniques outlined in this application report have been implemented. This is so that the C code remains generic and can be ported easily to other DSPs. However, you can easily apply other C62x C optimization techniques to increase performance. The following optimizations are implemented in Appendix D.

- The DFT is replaced with a radix-4 FFT, yielding a large computational savings as the number of data samples to be transformed increases. The radix-4 FFT restricts the size to a power of 4.
- Split operation tables and FFT twiddle factors are generated using pre-generated look-up tables instead of the run-time support functions sin() and cos(). This reduces the number of cycles required for the setup code.
- The code is organized as a series of functions to separate the independent tasks so they could be easily and independently optimized.

Example 3 and Example 4 show the compiler usage for building executable files for these implementations.

Example 3. Efficient Computation of the DFT of a 2N-Point Real Sequence

```
cl6x –g vectors.asm data1.c digitgen.c digit.c radix4.c realdft3.c split1.c
splitgen.c –z –o test3c.out –l rts6201.lib lnk.cmd
```

Example 4. Efficient Computation of the DFT of Two Real Sequences

```
cl6x –g vectors.asm data2.c realdft4.c split2.c radix4.c digit.c digitgen.c –z
–o test4c.out –l rts6201.lib lnk.cmd
```

The result of the above compiles is two executables:

- test3c.out
- test4c.out

The executables can be loaded into the C62x device simulator and run. The same restrictions that apply to the executables in Appendix C apply to these executables.

Appendix E contains C62x assembly language source code listings. These implementations fit into phase 3 of the code development flowchart shown in Figure 2. Each assembly listing contains a C62x C-callable assembly language function that replaces an equivalent C function shown in Appendix D. The following list includes functions implemented in assembly.

split1.asm The C-callable assembly language function that implements the split routine for the efficient computation of the DFT of two real sequences algorithm.

split2.asm The-callable assembly language function that implements the split routine for the efficient computation of the DFT of 2N-point real sequence.

radix4.asm Replaces radix4.c. A C-callable assembly language function that implements the radix-4 FFT.

digit.asm Replaces digit.c. A C-callable assembly language function that implements the digit reversal for the radix-4 FFT

Because each of the above routines is functionally equivalent in C and assembly, no modification of other functions in Appendix D is required to use them. All that must be changed to use these functions is the way in which we build the executables. Example 5 and Example 6 show how to build the executables with the assembly versions.

Example 5. Efficient Computation of the DFT of a 2N-Point Real Sequence

```
cl6x –g vectors.asm data1.c digitgen.c digit.asm radix4.asm realdft3.c
split1.asm splitgen.c –z –o test3a.out –l rts6201.lib lnk.cmd
```

Example 6. Efficient Computation of the DFT of Two Real Sequences

```
cl6x –g vectors.asm data2.c realdft4.c split2.asm radix4.asm digit.asm digit-
gen.c –z –o test4a.out –l rts6201.lib lnk.cmd
```

The result of the compiles shown in Example 5 and Example 6 is two executables:

- test3a.out
- test4a.out

These can be loaded into the C62x device simulator and run. The same restrictions that apply to the executables in Appendix C apply to these executables.

6 Summary

This application report examined the theory and implementation of two efficient methods for computing the DFT of real-valued sequences. The implementation was presented in both C and C62x assembly language. As this application report reveals, a large computational savings can be achieved using these methods on real-valued sequences rather using complex-valued DFTs or FFTs. Moreover, the TMS320C62x CPU performs well when implementing these algorithms in either C or assembly.

7 References

1. Burrus, C.S., and Parks, T.W. *DFT/FFT and Convolution Algorithms*, John Wiley and Sons, New York,1985.
2. Manolakis, D.G., and Proakis, J.G. *Introduction to Digital Signal Processing*, Macmillan Publishing Company, 1988.
3. *Digital Signal Processing Applications with the TMS320 Family, Theory, Algorithms and Implementations, Volume 3* (SPRA017).
4. *TMS320C6000 Technical Brief* (SPRU197).
5. Burrus, C.S., Heideman, M.T., Jones, D.L., Sorensen, H.V. “Real-Valued Fast Fourier Transform Algorithms*”, IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-35, No. 6,pp. 849–863, June 1987.
6. *TMS320C6000 Programmer’s Guide* (SPRU198).
7. *TMS320C6000 Optimizing C Compiler User’s Guide* (SPRU187).
8. *TMS320C6000 Assembly Language Tools User’s Guide* (SPRU186).
9. *TMS320C6x C Source Debugger User’s Guide* (SPRU188).

Appendix A Derivation of Equation Used to Compute the DFT/IDFT of Two Real Sequences

This appendix provides a detailed derivation of the equations used to compute the FFT/IDFT of two real sequences using one complex DFT/IDFT.

A.1 Forward Transform

Assume $x_1(n)$ and $x_2(n)$ are real-valued sequences of length N, and let $x(n)$ be a complex-valued sequence defined as

$$x(n) = x_1(n) + jx_2(n) \qquad 0 \le n \le N-1 \tag{17}$$

The DFT operation is linear, thus the DFT of $x(n)$ may be expressed as:

$$X(k) = X_1(k) + jX_2(k) \qquad 0 \le k \le N-1 \tag{18}$$

We can express the sequences $x_1(n)$ and $x_2(n)$ in terms of $x(n)$ as follows:

$$x_1(n) = \frac{x(n) + x^*(n)}{2}$$

where * is the complex conjugate operator (19)

$$x_2(n) = \frac{x(n) - x^*(n)}{2j}$$

The following shows that these equalities are true:

$$\frac{x(n) + x^*(n)}{2} = \frac{x_1(n) + jx_2(n) + x_1(n) - jx_2(n)}{2} = x_1(n)$$

$$\frac{x(n) - x^*(n)}{2j} = \frac{x_1(n) + jx_2(n) - x_1(n) + jx_2(n)}{2j} = x_2(n) \tag{20}$$

Therefore, we can express the DFT of $x_1(n)$ and $x_2(n)$ in terms of $x(n)$ as shown below:

$$X_1(k) = DFT[x_1(n)] = \frac{1}{2}\{DFT[x(n)] + DFT[x^*(n)]\}$$

$$X_2(k) = DFT[x_2(n)] = \frac{1}{2j}\{DFT[x(n)] - DFT[x^*(n)]\} \tag{21}$$

From the complex property of the DFT, we know the following is true:

$$\text{If } x(n) \overset{DFT}{\underset{N}{\longleftrightarrow}} X(k), \text{ then } x^*(n) \overset{DFT}{\underset{N}{\longleftrightarrow}} X^*(N-k)$$

Thus, we can express $X_1(k)$ and $X_2(k)$ as follows:

$$X_1(k) = \frac{1}{2}[x(k) + X^*(N-k)]$$

$$X_2(k) = \frac{1}{2j}[X(k) - X^*(N-k)] \tag{22}$$

From the above equations, we can see that by performing a single DFT on the complex-valued sequence $x(n)$, we have obtained the DFT of two real-valued sequences with only a small amount of additional computation in calculating $X_1(k)$ and $X_2(k)$ from $X(k)$.

In addition, because $x_1(n)$ and $x_2(n)$ are real-valued sequences, $X_1(k)$ and $X_2(k)$ has complex conjugate symmetry – $X_1(N–k) = X_1^*(k)$ and $X_2(N–k) = X_2^*(k)$; thus, we only need to compute $X_1(k)$ and $X_2(k)$ for k = 0,1,2, ..., N/2.

$$X_1(k) = \frac{1}{2}\{X(k) + X^*(N-k)\}$$

$$X_1(N-k) = X_1^*(k)$$

$$k = 0, 1, \ldots, N/2 \qquad \text{with } X(N) = X(0) \tag{23}$$

$$X_2(k) = \frac{1}{2j}\{X(k) - X^*(N-k)\}$$

$$X_2(N-k) = X_2^*(k)$$

To implement these equations, it is helpful if we express them in terms of their real and imaginary terms.

$$X_1(k) = \frac{1}{2}\{Xr(k) + jXi(k) + Xr(N-k) - jXi(N-k)\}$$

$$= \frac{1}{2}\{(Xr(k) + Xr(N-k)) + j(Xi(k) - Xi(N-k))\}$$

or

$$k = 0, 1, \ldots, N/2 \qquad \text{with } X(N) = X(0) \tag{24}$$

$$X_1 r(k) = \frac{1}{2}\{Xr(k) + Xr(N-k)\}$$

$$X_1 i(k) = \frac{1}{2}\{Xi(k) - Xi(N-k)\}$$

Similarly, it can be shown that

$$X_2 r(k) = \frac{1}{2}\{Xi(k) + Xi(N-k)\}$$

$$k = 0, 1, \ldots, N/2 \qquad \text{with } X(N) = X(0) \tag{25}$$

$$X_2 i(k) = \frac{-1}{2}\{Xr(k) - Xr(N-k)\}$$

There are two special cases with the above equations, k = 0 and k = N/2. For k = 0:

$$X_1 r(0) = \frac{1}{2}\{Xr(0) + Xr(N)\}$$

$$X_1 i(0) = \frac{1}{2}\{Xi(0) - Xi(N)\} \tag{26}$$

$$X_2 r(0) = \frac{1}{2}\{Xi(0) + Xi(N)\}$$

$$X_2 i(0) = \frac{-1}{2}\{Xr(0) - Xr(N)\}$$

Because of the periodicity property of the DFT, we know $X(k + N) = X(k)$. Therefore, $X_r(0) = X_r(N)$ and $X_i(0) = X_i(N)$. Using this property, the above equations can be expressed as follows:

$$\begin{aligned} X_1\, r(0) &= Xr(0) \\ X_1\, i(0) &= 0 \\ X_2\, r(0) &= Xi(0) \\ X_2\, i(0) & \end{aligned} \tag{27}$$

For $k = N/2$:

$$\begin{aligned} X_1\, r(N/2) &= \frac{1}{2}\{Xr(N/2) + Xr(N/2)\} \\ X_1\, i(N/2) &= \frac{1}{2}\{Xi(N/2) - Xi(N/2)\} \\ X_2\, r(N/2) &= \frac{1}{2}\{Xi(N/2) + Xi(N/2)\} \\ X_2\, i(N/2) &= \frac{-1}{2}\{Xr(N/2) - Xr(N/2)\} \end{aligned} \tag{28}$$

or

$$\begin{aligned} X_1\, r(N/2) &= Xr(N/2) \\ X_1\, i(N/2) &= 0 \\ X_2\, r(N/2) &= Xr(N/2) \\ X_2\, i(N/2) &= 0 \end{aligned} \tag{29}$$

Thus, (24) and (25) must be computed only for $k = 1,2, \ldots N/2 - 1$.

A.2 Inverse Transform

We can use a similar method to obtain the IDFT. We know $X_1(k)$ and $X_2(k)$. We want to express $X(k)$ in terms of $X_1(k)$ and $X_2(k)$. Recall, the relationship between $x_1(n)$, $x_2(n)$ and $x(n)$ is $x(n) = x_1(n) + jx_2(n)$. Since the DFT operator is linear, $X(k) = X_1(k) + jX_2(k)$. Thus, $X(k)$ can be found by the following equations:

$$\begin{aligned} Xr(k) &= X_1 r(k) - X_2 i(k) \\ Xi(k) &= X_1 i(k) + X_2 r(k) \end{aligned}$$

x(n) can then be found by taking the inverse transform of X(k).

x(n) = IDFT[*X(k)*]

From $x(n)$, we can get $x_1(n)$ and $x_2(n)$.

$$\begin{aligned} X_1(n) &= xr(n) \\ X_2(n) &= xi(n) \end{aligned}$$

Appendix B Derivation of Equations Used to Compute the DFT/IDFT of a Real Sequence

This appendix details the derivation of the equations used to compute the DFT/IDFT of a 2*N*-length real-valued sequence using an *N*-length complex DFT/IDFT.

B.1 Forward Transform

Assume $g(n)$ is a real-valued sequence of $2N$ points. The following shows how to obtain the $2N$-point DFT of $g(n)$ using an N-point complex DFT.

Let

$$\begin{aligned} X_1(n) &= g(2n) \\ X_2(n) &= g(2n+1) \end{aligned} \tag{30}$$

We have subdivided a $2N$-point real sequence into two N-point sequences. We now can apply the same method shown in Appendix A.

Let $x(n)$ be the N–point complex–valued sequence.

$$x(n) = x_1(n) + jx_2(n) \qquad 0 \le n \le N-1 \tag{31}$$

From the results shown in Appendix A, we have

$$\begin{aligned} X_1(k) &= \frac{1}{2}\{X(k) + X^*(N-k)\} \\ & \qquad\qquad k = 0,1,\ldots,N-1 \\ X_2(k) &= \frac{1}{2j}\{X(k) - X^*(N-k)\} \end{aligned} \tag{32}$$

We now express the $2N$-point DFT in terms of two N-point DFTs.

$G(k) = DFT[g(n)] = DFT[g(2n) + g(2n+1)] = DFT[g(2n)] + DFT[g(2n+1)]$

$$\begin{aligned} &= \sum_{n=0}^{N-1} g(2n) W_{2N}^{2nk} + \sum_{n=0}^{N-1} g(2n+1) W_{2N}^{(2n+1)k} \qquad k = 0,1,\ldots,N-1 \\ &= \sum_{n=0}^{N-1} x_1(n) W_N^{nk} + W_{2N}^{k} \sum_{n=0}^{N-1} x_2(n) W_N^{nk} \end{aligned} \tag{33}$$

Thus,

$$G(k) = X_1(k) + W_{2N}^{k} X_2(k) \qquad k = 0,1,\ldots,N-1 \tag{34}$$

Using equation (32), we can express $G(k)$ in terms of $X(k)$.

$$\begin{aligned} G(k) &= \frac{1}{2}\{X(k) + X^*(N-k)\} + W_{2N}^{k} \frac{1}{2j}\{X(k) - X^*(N-k)\} \\ & \qquad\qquad k = 0,1,\ldots,N-1 \\ &= X(k)\left[\frac{1}{2}\left(1 - jW_{2N}^{k}\right)\right] + X^*(N-k)\left[\frac{1}{2}\left(1 + jW_{2N}^{k}\right)\right] \end{aligned} \tag{35}$$

Let

$$A(k) = \left[\frac{1}{2}\left(1 - jW_{2N}^{k}\right)\right] \qquad k = 0,1,\ldots,N-1 \tag{36}$$

$$B(k) = \left[\frac{1}{2}\left(1 + jW_{2N}^{k}\right)\right]$$

Thus, $G(k)$ can be expressed as follows:

$$G(k) = X(k)A(k) + X^{*}(N-k)B(k) \qquad k = 0,1,\ldots,N-1 \tag{37}$$

Because $x(n)$ is a real–valued sequence, we know that the DFT transform results will have complex conjugate symmetry. Also, because of the periodicity property of the DFT, we know $X(k+N) = X(k)$; therefore, $X(N) = X(0)$. Using these properties, we can find the other half of the DFT result.

Thus, we have computed the DFT of a $2N$-point real sequence using one N-point complex DFT and additional computations.

To implement these equations, it is helpful to express them in terms of their real and imaginary terms.

$$G(k) = (Xr(k) + j\,Xi(k))(Ar(k) + j\,Ai(k)) + (Xr(N-k) - j\,Xi(N-k))(Br(k) + j\,Bi(k)) \qquad k = 0,1,\ldots,N-1 \tag{38}$$

Carrying out the multiplication, separating the real and imaginary terms, and applying the periodicity and complex conjugate properties, we have the following:

$$Gr(k) = Xr(k)Ar(k) - Xi(k)Ai(k) + Xr(N-k)Br(k) + Xi(N-k)Bi(k) \qquad k = 0,1,\ldots,N-1 \quad \text{with } X(N) = X(0)$$

$$Gi(k) = Xi(k)Ar(k) + Xr(k)Ai(k) + Xr(N-k)Bi(k) - Xi(N-k)Br(k)$$

$$Gr(k) = Xr(0) - Xi(0) \qquad k = N \tag{39}$$

$$Gi(k) = 0$$

$$Gr(2N-k) = Gr(k) \qquad k = 1,2,\ldots,N-1$$

$$Gi(2N-k) = Gi(k)$$

B.2 Inverse Transform

We will now derive the equations for the IDFT of a 2*N*-point complex sequence derived from a real sequence using an *N*-point complex IDFT. We express the *N*-point complex sequence, $X(k)$, in terms of the 2*N*-point complex sequence $G(k)$. Once $X(k)$ is known, $x(n)$ can be found by taking the IDFT of $X(k)$. Once $x(n)$ is known, $g(n)$ follows.

Equation (37) can be rewritten as follows:

$$\begin{aligned} G(k) &= X(k)A(k) + X^*(N-k)B(k) \qquad k = 0,1,\ldots,N-2 \\ G(N-k) &= X(N-k)A(N-k) + X^*(k)B(N-k) \qquad k = 0,1,\ldots,N/2-1 \end{aligned} \tag{40}$$

where

$$\begin{aligned} A(N-k) &= \left[\frac{1}{2}\left(1 - jW_{2N}^{N-k}\right)\right] = \left[\frac{1}{2}\left(1 + jW_{2N}^{-k}\right)\right] = A^*(k) \\ B(N-k) &= \left[\frac{1}{2}\left(1 + jW_{2N}^{N-k}\right)\right] = \left[\frac{1}{2}\left(1 - jW_{2N}^{-k}\right)\right] = B^*(k) \end{aligned} \tag{41}$$

The above equalities can be shown to be true by recalling the following definition and substituting appropriately for *k*.

$$W_{2N}^{k} = e^{-j2\pi k/2N} = \cos(2\pi k/2N) - j\sin(2\pi k/2N) \tag{42}$$

We would like to make the ranges of *k* for $G(k)$ and $G(N–k)$ the same. Look at $G(N/2)$:

$$\begin{aligned} G(N/2) &= X(N/2)A(N/2) + X^*(N/2)B(N/2) \\ &= X(N/2)\left[\frac{1}{2}\left(1 - jW_{2N}^{N/2}\right)\right] + X^*(N/2)\left[\frac{1}{2}\left(1 + jW_{2N}^{N/2}\right)\right] \end{aligned} \tag{43}$$

But from (42), we see that

$$W_{2N}^{N/2} = -j \tag{44}$$

Therefore:

$$G(N/2) = X^*(N/2) \tag{45}$$

Now we can express $G(k)$ and $G(N–k)$ with the same ranges of k, and along with (41) and (43) we have

$$G(k) = X(k)A(k) + X^*(N-k)B(k) \qquad k = 0,1,\ldots,N/2-1 \tag{46}$$

$$\begin{aligned} G(N-k) &= X(N-k)A^*(k) + X^*(k)B(k) \qquad k = 0,1,\ldots,N/2-1 \\ G(N/2) &= X^*(N/2) \end{aligned} \tag{47}$$

From (46) and (47) you can see we have two equations and two unknowns, $X(k)$ and $X(N–k)$. We can use some algebra tricks to come up with equations for $X(k)$ and $X(N–k)$. If we multiply both sides of (46) with $A(k)$, complex conjugate both sides of (47), then multiply both sides by $B(k)$, we have the following:

$$\begin{aligned} G(k)A(k) &= X(k)A(k) + X^*(N-k)B(k)A(k) \\ -G^*(N-k)B(k) &= X(k)B(k)B(k) + X^*(N-k)B(k)A(k) \\ \hline G(k)A(k) - G^*(N-k)B(k) &= X(k)\{A(k)A(k) - B(k)A(k)\} \\ & \qquad k = 0,1,\ldots,N/2-1 \end{aligned} \tag{48}$$

Solving for $X(k)$:

$$X(k) = \frac{G(k)A(k) - G^*(N-k)B(k)}{A(k)A(k) - B(k)B(k)} \qquad k = 0,1,\ldots,N/2-1 \tag{49}$$

Equation (49) can be simplified as follows:

$$A(k)A(k) = \left[\frac{1}{2}\left(1 - jW_{2N}^{k}\right)\right]\left[\frac{1}{2}\left(1 - jW_{2N}^{k}\right)\right] = \frac{1}{4}\left[1 - 2jW_{2N}^{k} - W_{2N}^{k}\right] \tag{50}$$

$$B(k)B(k) = \left[\frac{1}{2}\left(1 + jW_{2N}^{k}\right)\right]\left[\frac{1}{2}\left(1 + jW_{2N}^{k}\right)\right] = \frac{1}{4}\left[1 + 2jW_{2N}^{k} - W_{2N}^{k}\right] \tag{51}$$

$$\begin{aligned} A(k)kA(k) - B(k)B(k) &= -jW_{2N}^{k} \\ X(k) &= \frac{G(k)A(k) - G^*(N-k)B(k)}{-jW_{2N}^{k}} \qquad k = 0,1,\ldots,N/2-1 \end{aligned} \tag{52}$$

Similarly, if we multiply both sides of (47) with $A^*(k)$, conjugate both sides of (46), then multiply both sides by $B^*(k)$, we have the following:

$$\begin{aligned} G^*(k)B^*(k) &= X^*(k)A^*(k)B^*(k) + X(N-k)B^*(k)B^*(k) \\ -G^*(N-k)A^*(k) &= X^*(k)A^*(k)B^*(k) + X(N-k)A^*(k)A^*(k) \\ \hline G^*(k)B^*(k) - G(N-k)A^*(k) &= X(N-k)\{A^*(k)A^*(k) - B^*(k)B^*(k)\} \\ & \qquad k = 0,1,\ldots,N/2-1 \end{aligned} \tag{53}$$

Solving for $X(N–k)$:

$$X(N-k) = \frac{G^*(k)B^*(k) - G(N-k)A^*(k)}{A^*(k)A^*(k) - B^*(k)B^*(k)} \quad k = 0,1,\ldots,N/2-1 \tag{54}$$

Equation (54) can be simplified as follows:

$$A^*(k)A^*(k) = \left[\frac{1}{2}\left(1 + jW_{2N}^{-k}\right)\right]\left[\frac{1}{2}\left(1 + jW_{2N}^{-k}\right)\right] = \frac{1}{4}\left[1 + 2jW_{2N}^{-k} - W_{2N}^{-k}\right] \tag{55}$$

$$B^*(k)B^*(k) = \left[\frac{1}{2}\left(1 - jW_{2N}^{-k}\right)\right]\left[\frac{1}{2}\left(1 - jW_{2N}^{-k}\right)\right] = \frac{1}{4}\left[1 - 2jW_{2N}^{-k} - W_{2N}^{-k}\right] \tag{56}$$

$$A^*(k)A^*(k) - B^*(k)B^*(k) = jW_{2N}^{-k} \tag{57}$$

$$X(N-k) = \frac{G^*(k)B^*(k) - G(N-k)A^*(k)}{-jW_{2N}^{-k}} \quad k = 0,1,\ldots,N/2-1 \tag{58}$$

It can be shown that

$$\frac{A(k)}{-jW_{2N}^{k}} = A^*(k) \quad \text{and} \quad \frac{B(k)}{-jW_{2N}^{k}} = B(k) \tag{59}$$

$$\frac{A^*(k)}{jW_{2N}^{k}} = A^*(k) \quad \text{and} \quad \frac{B^*(k)}{jW_{2N}^{k}} = B(k) \tag{60}$$

Making these substitutions, we get:

$$\begin{aligned} X(k) &= G(k)A^*(k) + G^*(N-k)B^*(k) \\ X(N-k) &= G^*(k)B(k) + G(N-k)A(k) \\ X(N/2) &= G^*(N/2) \end{aligned} \qquad k = 0,1,\ldots,N/2-1 \tag{61}$$

For equation (61), if we make the following substitutions, along with replacing k with $N/2 – k$,

$$A(N-k) = A^*(k) \quad \text{and} \quad B(N-k) = B^*(k) \tag{62}$$

we can see that $X(k)$ can be expressed as a single equation.

$$\begin{aligned} X(k) &= G(k)A^*(k) + G^*(N-k)B^*(k) \qquad k = 0,1,\ldots,N-1 \\ & \qquad\qquad\qquad\qquad\quad G(N) = G(0) \end{aligned} \tag{63}$$

Now, in terms of implementing these equations, it is helpful to express them in terms of their real and imaginary terms.

$$X(k) = (Gr(k) + jGi(k))(Ar(k) - jAi(k)) + (Gr(N-k) - jGi(N-k))(Br(k) - jBi(k)) \quad k = 0,1,\ldots,N-1 \quad \text{with } G(N) = G(0) \tag{64}$$

Carrying out the multiplication and separating the real and imaginary terms, we have the following:

$$\begin{aligned} Xr(k) &= Gr(k)Ar(k) + Gi(k)Ai(k) + Gr(N-k)Br(k) - Gi(N-k)Bi(k) \\ Xi(k) &= Gi(k)Ar(k) - Gr(k)Ai(k) - Gr(N-k)Bi(k) - Gi(N-k)Br(k) \end{aligned} \quad k = 0,1,\ldots,N-1 \quad \text{with } G(N) = G(0) \tag{65}$$

Now we have formed the complex sequence with which we can use an N-point complex DFT to obtain $x(n)$, which we then can use to get $g(n)$.

$$x(n) = xr(n) + jxi(n) = IDFT[X(k)]$$

$$\begin{aligned} g(2n) &= xr(n) \\ g(2n+1) &= xi(n) \end{aligned} \quad n = 0,1,\ldots,N-1 \tag{66}$$

Appendix C C Implementations of the DFT of Real Sequences

This appendix contains C implementations of the efficient methods for performing the DFT of real-valued sequences.

C.1 Implementation Notes

The following lists usage, assumptions, and limitations of the code.

Data format	All data and state variables are 16-bit signed integers (shorts). In this example, the decimal point is assumed to be between bits 15 and 14, thus the Q15 data format. For complex data and variables, the real and imaginary components are both Q15 numbers. From this data format, you can see that this code was developed for a fixed-point processor.
Memory	Complex data is stored in memory in imaginary/real pairs. The imaginary component is stored in the most significant halfword (16 bits) and the real component is stored in the least significant halfword, unless otherwise noted.
Endianess	The code is presented and tested in little endian format. Some modification to the code is necessary for big endian format.
Overflow	No overflow protection or detection is performed.

File	Description
realdft1.c	DFT of a 2*N*-point real sequence main program
split1.c	Split function for the DFT of a 2*N*-point real sequence
data1.c	Sample data
params1.h	Header file, for example
realdft2.c	DFT of a two *N*-point real sequence main program
split2.c	Split function for the DFT of two *N*-point real sequence
data2.c	Sample data
params2.h	Header file, for example
dft.c	Direct implementation of the DFT function
params.h	Header file
vectors.asm	Reset vector assembly source
lnk.cmd	Example linker command file

Example C–1. realdft1.c File

```
/*************************************************************************
 FILE
   realdft1.c – C source for an example implementation of the DFT/IDFT
   of a 2N-point real sequences using one N-point complex DFT/IDFT.


*************************************************************************
```

Description

This program is an example implementation of an efficient way of computing the DFT/IDFT of a real-valued sequence.

In many applications, the input is a sequence of real numbers. If this condition is taken into consideration, additional computational savings can be achieved because the FFT of a real sequence has some symmetrical properties. The DFT of a 2N-point real sequence can be efficiently computed using a N-point complex DFT and some additional computations.

The following steps are required in the computation of the FFT of a real-valued sequence using the split function:

1. Let g(n) be a 2N-point real sequence. From g(n), form the the N-point complex valued sequence, x(n) = x1(n) + jx2(n), where x1(n) = g(2n) and x2(n) = g(2n + 1).

2. Perform an N-point complex FFT on the complex valued sequence x(n) -> X(k) = DFT{x(n)}. Note that the FFT can be any DFT method, such as radix-2, radix-4, mixed radix, direct implementation of the DFT, etc. However, the DFT output must be in normal order.

3. The following additional computation are used to get G(k) from
   $X(k)Gr(k) = Xr(k)Ar(k) – Xi(k)Ai(k) + Xr(N–k)Br(k) + Xi(N–k)Bi(k)$
   $k = 0, 1, ..., N–1$
   and $X(N) = X(0)$
   $Gi(k) = Xi(k)Ar(k) + Xr(k)Ai(k) + Xr(N–k)Bi(k) – Xi(N–k)Br(k)$

   Note that only N-points of the 2N-point sequence of G(k) are computed in the above equations. Because the DFT of a real-sequence has symmetric properties, we can easily compute the remaining N points of G(k) with the following equations.
   $Gr(N) = Xr(0) – Xi(0)$
   $Gi(N) = 0$
   $Gr(2N–k) = Gr(k)$
   $k = 1, 2, ..., N–1$
   $Gi(2N–k) = –Gi(k$

   As you can see, the above equations assume that A(k) and B(k), which are sine and cosine coefficients, are pre-computed. The C-code can be used to initialize A(k) and B(k).

```
for(k=0; k<N; k++)
{
   A[k].imag = (short)(16383.0*(–cos(2*PI/(double)(2*N)*(double)k)));
   A[k].real = (short)(16383.0*(1.0 – sin(2*PI/(double)(2*N)*(double)k)));
   B[k].imag = (short)(16383.0*(cos(2*PI/(double)(2*N)*(double)k)));
   B[k].real = (short)(16383.0*(1.0 + sin(2*PI/(double)(2*N)*(double)k)));
}
```

The following steps are required in the computation of the IFFT of a complex valued frequency domain sequence that was derived from a real sequence:

1. Let G(k) be a 2N-point complex valued sequence derived from a real valued sequence g(n). We want to get back g(n) from G(k) –> g(n) = IDFT{G(k)}. However, we want to apply the same techniques as we did with the forward FFT. Use a N-point IFFT. This can be accomplished by the following equations.
   Xr(k) = Gr(k)IAr(k) – Gi(k)IAi(k) + Gr(N–k)IBr(k) + Gi(N–k)IBi(k)
   k = 0, 1, ..., N–1
   and G(N) = G(0)
   Xi(k) = Gi(k)IAr(k) + Gr(k)IAi(k) + Gr(N–k)IBi(k) – Gi(N–k)IBr(k)
2. Perform the N-point inverse DFT of X(k) –> x(n) = x1(n) + jx2(n) = IDFT{X(k)}. Note that the IDFT can be any method, but must have an output that is in normal order.
3. g(n) can then be found from x(n).
   g(2n) = x1(n)
   n = 0, 1, ..., N–1
   g(2n+1) = x2(n)

As you can see, the above equations can be used for both the forward and inverse FFTs, however, the pre-computed coefficients are slightly different. The following C-code can be used to initialize IA(k) and IB(k).

```
   for(k=0; k<N; k++)
   {
      IA[k].imag = –(short)(16383.0*(–cos(2*PI/(double)(2*N)*(double)k)));
      IA[k].real = (short)(16383.0*(1.0 – sin(2*PI/(double)(2*N)*(double)k)));
      IB[k].imag = –(short)(16383.0*(cos(2*PI/(double)(2*N)*(double)k)));
      IB[k].real = (short)(16383.0*(1.0 + sin(2*PI/(double)(2*N)*(double)k)));
   }
Note, IA(k) is the complex conjugate of A(k) and IB(k) is the complex conjugate of
B(k).
 ************************************************************************/
#include <math.h>
#include ”params1.h”
#include ”params.h”
extern short g[];
void dft(int, COMPLEX *);
void split(int, COMPLEX *, COMPLEX *, COMPLEX *, COMPLEX *);
main()
{
   int   n, k;

   COMPLEX x[NUMPOINTS+1];       /* array of complex DFT data */
   COMPLEX  A[NUMPOINTS];        /* array of complex A coefficients */
   COMPLEX  B[NUMPOINTS];        /* array of complex B coefficients */
   COMPLEX  IA[NUMPOINTS];       /* array of complex A* coefficients */
   COMPLEX  IB[NUMPOINTS];       /* array of complex B* coefficients */
   COMPLEX G[2*NUMPOINTS];       /* array of complex DFT result */
```

```
                                /* Initialize A,B, IA, and IB arrays */

   for(k=0; k<NUMPOINTS; k++)
   {
   A[k].imag = (short)(16383.0*(–cos(2*PI/(double)(2*NUMPOINTS)*(double)k)));
   A[k].real = (short)(16383.0*(1.0 – sin(2*PI/(double)(2*NUMPOINTS)*(double)k)));
   B[k].imag = (short)(16383.0*(cos(2*PI/(double)(2*NUMPOINTS)*(double)k)));
   B[k].real = (short)(16383.0*(1.0 + sin(2*PI/(double)(2*NUMPOINTS)*(double)k)));
   IA[k].imag = –A[k].imag;
   IA[k].real = A[k].real;
   IB[k].imag = –B[k].imag;
   IB[k].real = B[k].real;
   }

/* Forward DFT */
/* From the 2N point real sequence, g(n), for the N-point complex sequence, x(n) */
   for (n=0; n<NUMPOINTS; n++)
   {
      x[n].imag = g[2*n + 1]; /* x2(n) = g(2n + 1) */
      x[n].real = g[2*n];     /* x1(n) = g(2n)     */
   }
/* Compute the DFT of x(n) to get X(k) –> X(k) = DFT{x(n)}  */

      dft(NUMPOINTS, x);
/* Because of the periodicity property of the DFT, we know that X(N+k)=X(k). */
      x[NUMPOINTS].real = x[0].real;
      x[NUMPOINTS].imag = x[0].imag;
/* The split function performs the additional computations required to get
   G(k) from X(k). */
      split(NUMPOINTS, x, A, B, G);
/* Use complex conjugate symmetry properties to get the rest of G(k) */
      G[NUMPOINTS].real = x[0].real – x[0].imag;
      G[NUMPOINTS].imag = 0;

      for (k=1; k<NUMPOINTS; k++)
   {
         G[2*NUMPOINTS–k].real = G[k].real;
         G[2*NUMPOINTS–k].imag = –G[k].imag;
   }
/* Inverse DFT – We now want to get back g(n).  */
```

```
/* The split function performs the additional computations required to get
   X(k) from G(k). */
      split(NUMPOINTS, G, IA, IB, x);
/* Take the inverse DFT of X(k) to get x(n).  Note the inverse DFT could be any
   IDFT implementation, such as an IFFT. */
/* The inverse DFT can be calculated by using the forward DFT algorithm directly by
complex conjugation – x(n) = (1/N)(DFT{X*(k)})*, where * is the complex conjugate
operator.  */
/* Compute the complex conjugate of X(k). */
      for (k=0; k<NUMPOINTS; k++)
      {
         x[k].imag = –x[k].imag; /* complex conjugate X(k) */

      }
/* Compute the DFT of X*(k). */
      dft(NUMPOINTS, x);
/* Complex conjugate the output of the DFT and divide by N to get x(n). */
      for (n=0; n<NUMPOINTS; n++)
      {
         x[n].real = x[n].real/16;
         x[n].imag = (–x[n].imag)/16;
      }
/* g(2n) = xr(n) and g(2n + 1) = xi(n) */
      for (n=0; n<NUMPOINTS; n++)
      {
         g[2*n] = x[n].real;
         g[2*n + 1] = x[n].imag;
      }
      return(0);
 }
```

Example C–2. split1.c File

```
/*************************************************************************
 FILE
   split1.c – This is the C source code for the implementation of the
   split routine, which is the additional computation in computing the
   DFT of an 2N-point real-valued sequences using a N-point complex DFT.
*************************************************************************
```

Description

Computation of the DFT of 2N-point real-valued sequences can be efficiently computed using one N-point complex DFT and some additional computations. This function implements these additional computations, which are shown below.

$$Gr(k) = Xr(k)Ar(k) – Xi(k)Ai(k) + Xr(N–k)Br(k) + Xi(N–k)Bi(k)$$

$$k = 0, 1, ..., N–1$$

$$\text{and } X(N) = X(0)$$

$$Gi(k) = Xi(k)Ar(k) + Xr(k)Ai(k) + Xr(N–k)Bi(k) – Xi(N–k)Br(k)$$

```
*************************************************************************/
#include ”params1.h”
#include ”params.h”
void split(int N, COMPLEX *X, COMPLEX *A, COMPLEX *B, COMPLEX *G)
{
      int k;
      int Tr, Ti;
      for (k=0; k<N; k++)
      {
         Tr = (int)X[k].real * (int)A[k].real – (int)X[k].imag * (int)A[k].imag +
         (int)X[N–k].real * (int)B[k].real + (int)X[N–k].imag * (int)B[k].imag;
         G[k].real = (short)(Tr>>15);

         Ti = (int)X[k].imag * (int)A[k].real + (int)X[k].real * (int)A[k].imag +
         (int)X[N–k].real * (int)B[k].imag – (int)X[N–k].imag * (int)B[k].real;
         G[k].imag = (short)(Ti>>15);
      }
}
```

Example C–3. data1.c File

```
/*************************************************************************
 FILE
   data1.c – Sample data used in realdft1.c

 *************************************************************************
/* array of real–valued input sequence, g(n)      */
short g[] = {255, –35, 255, –35, 255, 255, 255, 255,
      255, 255, 255, 20, 255, 255, 255, 255,
      0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0};
```

Example C–4. params1.h File

```
/*************************************************************************
 FILE
   params1.h – This is the C header file for example real FFT
   implementations.
 *************************************************************************/
#define     NUMDATA     32          /* number of real data samples */
#define     NUMPOINTS   NUMDATA/2   /* number of point in the DFT  */
```

Example C–5. realdft2.c File

```
/*****************************************************************************
 FILE
   realdft2.c – C source for an example implementation of the DFT/IDFT of two
   N-point real sequences using one N-point complex DFT/IDFT.

 *****************************************************************************
```

Description

This program is an example implementation of an efficient way of computing the DFT/IDFT of two real-valued sequences.

Assume we have two real-valued sequences of length N – x1[n] and x2[n]. The DFT of x1[n] and x2[n] can be computed with one complex-valued DFT of length N, as shown above, by following this algorithm.

1. Form the complex-valued sequence x[n] from x1[n] and x2[n]
   xr[n] = x1[n] and xi[n] = x2[n], 0,1, ..., N–1

   Note, if the sequences x1[n] and x2[n] are coming from another algorithm or a data acquisition driver, this step may be eliminated if these put the data in the complex-valued format correctly.
2. Compute X[k] = DFT{x[n]}

This can be the direct-form DFT algorithm or an FFT algorithm. If using an FFT algorithm, make sure the output is in normal order – bit reversal is performed.

3. Compute the following equations to get the DFTs of x1[n] and x2[n].

```
X1r[0] = Xr[0]
X1i[0] = 0

X2r[0] = Xi[0]
X2i[0] = 0

X1r[N/2] = Xr[N/2]
X1i[N/2] = 0

X2r[N/2] = Xi[N/2]
X2i[N/2] = 0
for k = 1,2,3, ...., N/2–1
        X1r[k] = (Xr[k] + Xr[N–k])/2
        X1i[k] = (Xi[k] – Xi[N–k])/2
        X1r[N–k] = X1r[k]
        X1i[N–k] = X1i[k]

        X2r[k] = (Xi[k] + Xi[N–k])/2
        X2i[k] = (Xr[N–k] – Xr[k])/2
        X2r[N–k] = X2r[k]
        X2i[N–k] = X2i[k]
```

4. Form X[k] from X1[k] and X2[k]

   for k = 0,1, ..., N–1
   Xr[k] = X1r[k] – X2i[k]
   Xi[k] = X1i[k] + X2r[k]

5. Compute x[n] = IDFT{X[k]}

   This can be the direct form IDFT algorithm, or an IFFT algorithm. If using an IFFT algorithm, make sure the output is in normal order – bit reversal is performed

```
*******************************************************************************/
#include <math.h> /* include the C RTS math library         */
#include ”params2.h” /* include file with parameters        */
#include ”params.h”  /* include file with parameters        */
extern short x1[];
extern short x2[];
void dft(int, COMPLEX *);
extern void split2(int, COMPLEX *, COMPLEX *, COMPLEX *);
main()
{
   int   n, k;

   COMPLEX X1[NUMDATA];    /* array of real-valued DFT output sequence, X1(k)   */
   COMPLEX X2[NUMDATA];    /* array of real-valued DFT output sequence, X2(k)   */
   COMPLEX x[NUMPOINTS+1]; /* array of complex DFT data, X(k)                    */


/* Forward DFT */
/* From the two N-point real sequences, x1(n) and x2(n), form the N-point complex
   sequence, x(n) = x1(n) + jx2(n) */
      for (n=0; n<NUMDATA; n++)
      {
         x[n].real = x1[n];
         x[n].imag = x2[n];
      }

/* Compute the DFT of x(n), X(k) = DFT{x(n)}. Note, the DFT can be any
   DFT implementation such as FFTs. */
      dft(NUMPOINTS, x);
/* Because of the periodicity property of the DFT, we know that X(N+k)=X(k). */
      x[NUMPOINTS].real = x[0].real;
      x[NUMPOINTS].imag = x[0].imag;
```

```
/* The split function performs the additional computations required to get
   X1(k) and X2(k) from X(k). */
      split2(NUMPOINTS, x, X1, X2);
/* Inverse DFT – We now want to get back x1(n) and x2(n) from X1(k) and X2(k) using
   one complex DFT */
/* Recall that x(n) = x1(n) + jx2(n).  Since the DFT operator is linear,
   X(k) = X1(k) + jX2(k).  Thus we can express X(k) in terms of X1(k) and X2(k). */
      for (k=0; k<NUMPOINTS; k++)
      {
         x[k].real = X1[k].real – X2[k].imag;
         x[k].imag = X1[k].imag + X2[k].real;
      }
/* Take the inverse DFT of X(k) to get x(n).  Note the inverse DFT could be any
   IDFT implementation, such as an IFFT. */
/* The inverse DFT can be calculated by using the forward DFT algorithm directly
   by complex conjugation – x(n) = (1/N)(DFT{X*(k)})*, where * is the complex
   conjugate operator.  */
/* Compute the complex conjugate of X(k). */
      for (k=0; k<NUMPOINTS; k++)
      {
         x[k].imag = –x[k].imag;

      }
/* Compute the DFT of X*(k). */
      dft(NUMPOINTS, x);
/* Complex conjugate the output of the DFT and divide by N to get x(n). */
      for (n=0; n<NUMPOINTS; n++)
      {
         x[n].real = x[n].real/16;
         x[n].imag = (–x[n].imag)/16;
      }
/* x1(n) is the real part of x(n), and x2(n) is the imaginary part of x(n). */
      for (n=0; n<NUMDATA; n++)
      {
         x1[n] = x[n].real;
         x2[n] = x[n].imag;
      }
      return(0);
 }
```

Example C–6. split2.c File

```
/*************************************************************************
 FILE
   split2.c – This is the C source code for the implementation of the
   split routine, which is the additional computations in computing the
   DFT of two N-point real-valued sequences using one N-point complex DFT.
 *************************************************************************

Description

    Computation of the DFT of two N-point real-valued sequences can be efficiently computed using
    one N-point complex DFT and some additional computations. This function implements these
    additional computations, which are shown below.

        X1r[0] = Xr[0]
        X1i[0] = 0

        X2r[0] = Xi[0]
        X2i[0] = 0

        X1r[N/2] = Xr[N/2]
        X1i[N/2] = 0

        X2r[N/2] = Xi[N/2]
        X2i[N/2] = 0

        for k = 1,2,3, ...., N/2–1
              X1r[k] = (Xr[k] + Xr[N–k])/2
              X1i[k] = (Xi[k] – Xi[N–k])/2
              X1r[N–k] = X1r[k]
              X1i[N–k] = X1i[k]
              X2r[k] = (Xi[k] + Xi[N–k])/2
              X2i[k] = (Xr[N–k] – Xr[k])/2
              X2r[N–k] = X2r[k]
              X2i[N–k] = X2i[k]
 *************************************************************************/
#include ”params.h”
void split2(int N, COMPLEX *X, COMPLEX *X1, COMPLEX *X2)
{
      int k;

      X1[0].real = X[0].real;
      X1[0].imag = 0;

      X2[0].real = X[0].imag;
      X2[0].imag = 0;

      X1[N/2].real = X[N/2].real;
      X1[N/2].imag = 0;

      X2[N/2].real = X[N/2].imag;
      X2[N/2].imag = 0;

      for (k=1; k<N/2; k++)
   {
         X1[k].real = (X[k].real + X[N–k].real)/2;
```

```
        X1[k].imag = (X[k].imag – X[N–k].imag)/2;

        X2[k].real = (X[k].imag + X[N–k].imag)/2;
        X2[k].imag = (X[N–k].real – X[k].real)/2;

        X1[N–k].real = X1[k].real;
        X1[N–k].imag = –X1[k].imag;

        X2[N–k].real = X2[k].real;
        X2[N–k].imag = –X2[k].imag;

   }
}
```

Example C–7. data2.c File

```
/*************************************************************************

 FILE
   data2.c – Sample data used in realdft2.c

*************************************************************************

/* array of real-valued input sequence, x1(n)      */
short x1[] = {255, 255, 255, 255, 255, 255, 255, 255,
      0, 0, 0, 0, 0, 0, 0, 0};
/* array of real-valued input sequence, x2(n)      */

short x2[] = {–35, –35, –35, –35, –35, –35, –35, –35,
      0, 0, 0, 0, 0, 0, 0, 0};
```

Example C–8. params2.h File

```
/************************************************************************

 FILE
   params2.h – This is the C header file for example real FFT
   implementations.
 ************************************************************************/

#define     NUMDATA     16       /* number of real data samples */
#define     NUMPOINTS   NUMDATA  /* number of point in the DFT  */
```

Example C–9. dft.c File

```
/************************************************************************

 FILE
   dft.c – This is the C source code for the direct implementation of the
   Discrete Fourier Transform (DFT) algorithm.
 ************************************************************************
```

Description

This function computes the DFT of an N-length complex-valued sequence. Note, N cannot exceed 1024 without modification to this code.

The N point DFT of a finite-duration sequence x(n) of length L<–N is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) * \exp(-j2\pi kn/N) \quad k = 0,1,2, \ldots, N-1$$

It is always helpful to express the above equation in its real and imaginary terms for implementation.

```
   exp(–j2*pi*n*k/N) = cos(2*pi*n*k/N) – jsin(2*pi*n*k/N) –> several
   identities used here
            e(jb) = cos(b) + j sin(b)
            e(–jb) = cos(–b) + j sin(–b)
            cos(–b) = cos(b) and sin(–b) = –sin(b)
            e(–jb) = cos(b) – j sin(b)

           N–1
   X(k) = SUM {[xr(n) + j xi(n)][cos(2*pi*n*k/N) – jsin(2*pi*n*k/N)]}
           n=0
                                                k=0,1,2, ... ,N–1
                                  OR
            N–1
   Xr(k) = SUM {[xr(n) * cos(2*pi*n*k/N)] + [xi(n) * sin(2*pi*n*k/N)]}
            n=0
                                                k=0,1,2, ... ,N–1
          N–1
 Xi(k) = SUM {[xi(n) * cos(2*pi*n*k/N)] – [xr(n) * sin(2*pi*n*k/N)]}
          n=0
*************************************************************************/
#include <math.h>
#include ”params.h”

void dft(int N, COMPLEX *X)
{
      int n, k;
      double arg;
      int Xr[1024];
      int Xi[1024];
      short Wr, Wi;
```

```
      for(k=0; k<N; k++)
      {
         Xr[k] = 0;
         Xi[k] = 0;
      for(n=0; n<N; n++)
      {
            arg =(2*PI*k*n)/N;
            Wr = (short)((double)32767.0 * cos(arg));
            Wi = (short)((double)32767.0 * sin(arg));
            Xr[k] = Xr[k] + X[n].real * Wr + X[n].imag * Wi;
            Xi[k] = Xi[k] + X[n].imag * Wr – X[n].real * Wi;
      }
   }
   for (k=0;k<N;k++)
   {
      X[k].real = (short)(Xr[k]>>15);
      X[k].imag = (short)(Xi[k]>>15);
   }
}
```

Example C–10. params.h File

```
/*************************************************************************
 FILE
   params.h – This is the C header file for example real FFT
   implementations.
 *************************************************************************/
#define     TRUE     1
#define     FALSE    0
#define     BE       TRUE
#define     LE       FALSE
#define     ENDIAN   LE       /* selects proper endianess.  If
                                 building code in Big Endian,
                                 use BE, else use LE */
#define     PI       3.141592653589793 /* definition of pi */
/* Some functions used in the example implementations use word loads which make
   the code endianess dependent.  Thus, one of the below definitions need to be
   used depending on the endianess you are using to build your code */
/* BIG Endian */
#if ENDIAN == TRUE
   typedef struct {
      short imag;
```

```
      short real;
      } COMPLEX;
#else
/* LITTLE Endian */
   typedef struct {
      short real;
      short imag;
      } COMPLEX;
#endif
```

Example C–11. vectors.asm

```
/*************************************************************/
/*  vectors.asm – reset vector assembly                      */
/*************************************************************/
               .def    RESET
               .ref    _c_int00
               .sect   ”.vectors”
RESET:
     mvk       .s2     _c_int00, B2
     mvkh      .s2     _c_int00, B2
     b         .s2     B2
     nop
     nop
     nop
     nop
     nop
```

Example C–12. lnk.cmd

```
/*************************************************************/
/*  lnk.cmd – example linker command file                    */
/*************************************************************/
–c
–heap  0x2000
–stack 0x8000
MEMORY
{
    VECS:   o = 00000000h       l=00200h /* reset & interrupt vectors*/
    IPRAM:  o = 00000200h       l=0FE00h /* internal program memory  */
    IDRAM:  o = 80000000h       l=10000h /* internal data memory     */
}
```

```
SECTIONS
{
    vectors     >       VECS
    .text       >       IPRAM
    .tables     >       IDRAM
    .data       >       IDRAM
    .stack      >       IDRAM
    .bss        >       IDRAM
    .sysmem     >       IDRAM
    .cinit      >       IDRAM
    .const      >       IDRAM
    .cio        >       IDRAM
    .far        >       IDRAM
}
```

Appendix D Optimized C Implementation of the DFT of Real Sequences

This appendix contains optimized C implementations of the efficient methods for performing the DFT of real-valued sequences outlined in this application report.

D.1 Implementation Notes

The following lists usage, assumption, and limitations of the code.

Data format	All data and state variables are 16-bit signed integers (shorts). In this example, the decimal point is assumed to be between bits 15 and 14, thus the Q15 data format. For complex data and variables, the real and imaginary components are both Q15 numbers. From this data format, you can see that this code was developed for a fixed-point processor.
Memory	Complex data is stored in memory in imaginary/r,eal pairs. The imaginary component is stored in the most significant halfword (16 bits) and the real component is stored in the least significant halfword, unless otherwise noted.
Endianess	The code is presented and tested in little endian format. Some modification to the code is necessary for big endian format.
Overflow	No overflow protection or detection is performed.

File	Description
realdft3.c	DFT of a 2*N*-point real sequence main program
realdft4.c	DFT of a two *N*-point real sequence main program
radix4.c	Radix-4 FFT C function
digit.c	Radix-4 digit reversal C function
digitgen.c	C function used to initialize digit reversal table used by the function in digit .c
splitgen.c	C function used to initialize the split tables used by the split1 routines

Example D–1. realdft3.c File

```
/*************************************************************************
 FILE
   realdft3.c – C source for an example implementation of the DFT/IDFT
   of a 2N-point real sequence, using one N-point complex DFT/IDFT.

*************************************************************************
```

D.2 Description

This program is an example implementation of an efficient way of computing the DFT/IDFT of a real-valued sequence.

In many applications, the input is a sequence of real numbers. If this condition is taken into consideration, additional computational savings can be achieved because the FFT of a real sequence has some symmetrical properties. The DFT of a 2N-point real sequence can be efficiently computed using a N-point complex DFT and some additional computations.

The following steps are required in the computation of the FFT of a real-valued sequence using the split function:

1. Let g(n) be a 2N-point real sequence. From g(n), form the the N-point complex-valued sequence, x(n) = x1(n) + jx2(n), where x1(n) = g(2n) and x2(n) = g(2n + 1).
2. Perform an N-point complex FFT on the complex valued sequence x(n) –> X(k) = DFT{x(n)}. Note that the FFT can be any DFT method, such as radix-2, radix-4, mixed radix, direct implementation of the DFT, etc. However, the DFT output must be in normal order.
3. The following additional computation are used to get G(k) from X(k)
   Gr(k) = Xr(k)Ar(k) – Xi(k)Ai(k) + Xr(N–k)Br(k) + Xi(N–k)Bi(k)
   k = 0, 1, ..., N–1
   and X(N) = X(0)
   Gi(k) = Xi(k)Ar(k) + Xr(k)Ai(k) + Xr(N–k)Bi(k) – Xi(N–k)Br(k)

Note that only N-points of the 2N-point sequence of G(k) are computed in the above equations. Because the DFT of a real–sequence has symmetric properties, we can easily compute the remaining N points of G(k) with the following equations.
Gr(N) = Xr(0) – Xi(0)
Gi(N) = 0

Gr(2N–k) = Gr(k)
k = 1, 2, ..., N–1
Gi(2N–k) = –Gi(k)

As you can see, the above equations assume that A(k) and B(k), which are sine and cosine coefficients, are pre–computed. The C–code can be used to initialize A(k) and B(k).

```
for(k=0; k<N; k++)
{
   A[k].imag = (short)(16383.0*(–cos(2*PI/(double)(2*N)*(double)k)));
   A[k].real = (short)(16383.0*(1.0 – sin(2*PI/(double)(2*N)*(double)k)));
   B[k].imag = (short)(16383.0*(cos(2*PI/(double)(2*N)*(double)k)));
   B[k].real = (short)(16383.0*(1.0 + sin(2*PI/(double)(2*N)*(double)k)));
}
```

The following steps are required in the computation of the IFFT of a complex-valued frequency domain sequence that was derived from a real sequence:

1. Let G(k) be a 2N-point complex valued sequence derived from a real-valued sequence g(n). We want to get back g(n) from G(k) –> g(n) = IDFT{G(k)}. However, we want to apply the same techniques as we did with the forward FFT, using an N-point IFFT. This can be accomplished by the following equations.
   Xr(k) = Gr(k)IAr(k) – Gi(k)IAi(k) + Gr(N–k)IBr(k) + Gi(N–k)IBi(k)
   k = 0, 1, ..., N–1
   and G(N) = G(0)
   Xi(k) = Gi(k)IAr(k) + Gr(k)IAi(k) + Gr(N–k)IBi(k) – Gi(N–k)IBr(k)
2. Perform the N-point inverse DFT of X(k) –> x(n) = x1(n) + jx2(n) = IDFT{X(k)}. Note that the IDFT can be any method, but must have an output that is in normal order.
3. g(n) can then be found from x(n).
   g(2n) = x1(n)
   n = 0, 1, ..., N–1
   g(2n+1) = x2(n)

As you can see, the above equations can be used for both the forward and inverse FFTs; however, the pre-computed coefficients are slightly different. The following C code can be used to initialize IA(k) and IB(k).

```
for(k=0; k<N; k++)
{
   IA[k].imag = –(short)(16383.0*(–cos(2*PI/(double)(2*N)*(double)k)));
   IA[k].real = (short)(16383.0*(1.0 – sin(2*PI/(double)(2*N)*(double)k)));
   IB[k].imag = –(short)(16383.0*(cos(2*PI/(double)(2*N)*(double)k)));
   IB[k].real = (short)(16383.0*(1.0 + sin(2*PI/(double)(2*N)*(double)k)));
}
```

Note that IA(k) is the complex conjugate of A(k), and IB(k) is the complex conjugate of B(k).

```
*************************************************************************/
typedef struct {  /* define the data type for the radix-4 twiddle factors */
   short imag;
   short real;
   } COEFF;
#include ”params1.h”       /* header files with parameters */
#include ”params.h”
#include ”splittbl.h”      /* header file that contains tables used to generate
                              the split tables */
#include ”sinestbl.h”      /* header file that contains the FFT twiddle factors */
#pragma DATA_ALIGN(x,64);  /* radix-4 routine requires x to be
                              aligned to a 4*NUMPOINTS boundry */
COMPLEX x[NUMPOINTS+1];    /* array of complex DFT data */
extern short g[];          /* real-valued input sequence */
/* functions defined externally */
void FftSplitTableGen(int N, COMPLEX *W, COMPLEX *A, COMPLEX *B);
void R4DigitRevIndexTableGen(int, int *, unsigned short *, unsigned short *);
void split1(int, COMPLEX *, COMPLEX *, COMPLEX *, COMPLEX *);
void digit_reverse(int *, unsigned short *, unsigned short *, int);
void radix4(int, short[], short[]);
main()
{
   int   n, k;
   COMPLEX  A[NUMPOINTS];     /* array of complex A coefficients */
   COMPLEX  B[NUMPOINTS];     /* array of complex B coefficients */
   COMPLEX  IA[NUMPOINTS];    /* array of complex A* coefficients */
   COMPLEX  IB[NUMPOINTS];    /* array of complex B* coefficients */
   COMPLEX G[2*NUMPOINTS];    /* array of complex DFT result */
   unsigned short IIndex[NUMPOINTS], JIndex[NUMPOINTS];
```

```
   int count;
/* Initialize A,B, IA, and IB arrays */
   FftSplitTableGen(NUMPOINTS, W, A, B);

/* Split tables for the IDFT are the complex conjugate of the split
   tables of the DFT */
   for(k=0; k<NUMPOINTS; k++)
   {
      IA[k].imag = –A[k].imag;
      IA[k].real = A[k].real;
      IB[k].imag = –B[k].imag;
      IB[k].real = B[k].real;
   }
/* Initialize tables for FFT digit reversal function */
   R4DigitRevIndexTableGen(NUMPOINTS, &count, IIndex, JIndex);

/* Forward DFT */
/* From the 2N point real sequence, g(n), for the N-point complex sequence, x(n) */
   for (n=0; n<NUMPOINTS; n++)
   {
      x[n].imag = g[2*n + 1];    /* x2(n) = g(2n + 1) */
      x[n].real = g[2*n];        /* x1(n) = g(2n)     */
   }
/* Compute the DFT of x(n) to get X(k) –> X(k) = DFT{x(n)}  */

   radix4(NUMPOINTS, (short *)x, (short *)W4);
   digit_reverse((int *)x, IIndex, JIndex, count);
/* Because of the periodicity property of the DFT, we know that X(N+k)=X(k). */
   x[NUMPOINTS].real = x[0].real;
   x[NUMPOINTS].imag = x[0].imag;
/* The split function performs the additional computations required to get
   G(k) from X(k). */
   split1(NUMPOINTS, x, A, B, G);
/* Use complex conjugate symmetry properties to get the rest of G(k) */

   G[NUMPOINTS].real = x[0].real – x[0].imag;
   G[NUMPOINTS].imag = 0;

   for (k=1; k<NUMPOINTS; k++)
   {
```

```
      G[2*NUMPOINTS–k].real = G[k].real;
      G[2*NUMPOINTS–k].imag = –G[k].imag;
   }
/* Inverse DFT – We now want to get back g(n).  */
/* The split function performs the additional computations required to get
   X(k) from G(k). */
   split1(NUMPOINTS, G, IA, IB, x);
/* Take the inverse DFT of X(k) to get x(n).  Note the inverse DFT could be any
   IDFT implementation, such as an IFFT. */
/* The inverse DFT can be calculated by using the forward DFT algorithm directly
   by complex conjugation – x(n) = (1/N)(DFT{X*(k)})*, where * is the complex
   conjugate operator.  */
/* Compute the complex conjugate of X(k). */
   for (k=0; k<NUMPOINTS; k++)
   {
      x[k].imag = –x[k].imag;    /* complex conjugate X(k) */

   }
/* Compute the DFT of X*(k). */
   radix4(NUMPOINTS, (short *)x, (short *)W4);
   digit_reverse((int *)x, IIndex, JIndex, count);
 /* Complex conjugate the output of the DFT and divide by N to get x(n). */
   for (n=0; n<NUMPOINTS; n++)
   {
      x[n].real = x[n].real/16;
      x[n].imag = (–x[n].imag)/16;
   }
/* g(2n) = xr(n) and g(2n + 1) = xi(n) */
   for (n=0; n<NUMPOINTS; n++)
   {
      g[2*n] = x[n].real;
      g[2*n + 1] = x[n].imag;
   }
   return(0);
 }
```

Example D–2. realdft4.c File

```
/*************************************************************************
 FILE
   realdft4.c – C source for an example implementation of the DFT/IDFT
   of two N-point real sequences using one N-point complex DFT/IDFT.


*************************************************************************
```

Description

This program is an example implementation of an efficient way of computing the DFT/IDFT of two real-valued sequences.

Assume we have two real-valued sequences of length N – x1[n] and x2[n]. The DFT of x1[n] and x2[n] can be computed with one complex–valued DFT of length N, as shown above, by following this algorithm.

1. Form the complex–valued sequence x[n] from x1[n] and x2[n]
   r[n] = x1[n] and xi[n] = x2[n], 0,1, ..., N–1

   Note, if the sequences x1[n] and x2[n] are coming from another algorithm or a data acquisition driver, this step may be eliminated if these put the data in the complex–valued format correctly.

2. Compute X[k] = DFT{x[n]}

   This can be the direct form DFT algorithm, or an FFT algorithm. If using an FFT algorithm, make sure the output is in normal order – bit reversal is performed.

3. Compute the following equations to get the DFTs of x1[n] and x2[n].

   X1r[0] = Xr[0]
   X1i[0] = 0

   X2r[0] = Xi[0]
   X2i[0] = 0

   X1r[N/2] = Xr[N/2]
   X1i[N/2] = 0

   X2r[N/2] = Xi[N/2]
   X2i[N/2] = 0

   for k = 1,2,3, ..., N/2–1
   X1r[k] = (Xr[k] + Xr[N–k])/2
   X1i[k] = (Xi[k] – Xi[N–k])/2
   X1r[N–k] = X1r[k]
   X1i[N–k] = X1i[k]

X2r[k] = (Xi[k] + Xi[N–k])/2
X2i[k] = (Xr[N–k] – Xr[k])/2
X2r[N–k] = X2r[k]
X2i[N–k] = X2i[k]

4. Form X[k] from X1[k] and X2[k]

   for k = 0,1, ..., N–1
   Xr[k] = X1r[k] – X2i[k]
   Xi[k] = X1i[k] + X2r[k]

5. Compute x[n] = IDFT{X[k]}
   This can be the direct form IDFT algorithm or an IFFT algorithm. If using an IFFT algorithm, make sure the output is in normal order – bit reversal is performed.

```
*************************************************************************/
typedef struct {  /* define the data type for the radix-4 twiddle factors */
   short imag;
   short real;
   } COEFF;
#include ”params2.h”       /* include file with parameters */
#include ”params.h”        /* include file with parameters */
#include ”sinestbl.h”      /* header file that contains the FFT twiddle factors */
#pragma DATA_ALIGN(x,64);  /* radix-4 routine requires x to be
                              aligned to a 4*NUMPOINTS boundary */
COMPLEX x[NUMPOINTS+1];    /* array of complex DFT data, X(k) */
extern short x1[];
extern short x2[];
void R4DigitRevIndexTableGen(int, int *, unsigned short *, unsigned short *);
extern void split2(int, COMPLEX *, COMPLEX *, COMPLEX *);
void digit_reverse(int *, unsigned short *, unsigned short *, int);
void radix4(int, short[], short[]);
main()
{
   int   n, k;

   COMPLEX X1[NUMDATA];    /* array of real-valued DFT output sequence, X1(k) */
   COMPLEX X2[NUMDATA];    /* array of real-valued DFT output sequence, X2(k) */
   unsigned short IIndex[NUMPOINTS], JIndex[NUMPOINTS];
   int count;
/* Initialize tables for FFT digit reversal function */
   R4DigitRevIndexTableGen(NUMPOINTS, &count, IIndex, JIndex);
/* Forward DFT */
/* From the two N-point real sequences, x1(n) and x2(n), form the N-point complex
   sequence, x(n) = x1(n) + jx2(n) */
```

```
   for (n=0; n<NUMDATA; n++)
   {
      x[n].real = x1[n];
      x[n].imag = x2[n];
   }

/* Compute the DFT of x(n), X(k) = DFT{x(n)}.  Note, the DFT can be any
   DFT implementation such as FFTs. */
   radix4(NUMPOINTS, (short *)x, (short *)W4);
   digit_reverse((int *)x, IIndex, JIndex, count);
/* Because of the periodicity property of the DFT, we know that X(N+k)=X(k). */
   x[NUMPOINTS].real = x[0].real;
   x[NUMPOINTS].imag = x[0].imag;
/* The split function performs the additional computations required to get
   X1(k) and X2(k) from X(k). */
   split2(NUMPOINTS, x, X1, X2);
/* Inverse DFT – We now want to get back x1(n) and x2(n) from X1(k) and X2(k) using
   one complex DFT */
/* Recall that x(n) = x1(n) + jx2(n). Since the DFT operator is linear,
   X(k) = X1(k) + jX2(k).  Thus we can express X(k) in terms of X1(k) and X2(k). */
   for (k=0; k<NUMPOINTS; k++)
   {
      x[k].real = X1[k].real – X2[k].imag;
      x[k].imag = X1[k].imag + X2[k].real;
   }
/* Take the inverse DFT of X(k) to get x(n).  Note the inverse DFT could be any
   IDFT implementation, such as an IFFT. */
/* The inverse DFT can be calculated by using the forward DFT algorithm directly
   by complex conjugation – x(n) = (1/N)(DFT{X*(k)})*, where * is the complex
   conjugate operator.  */
/* Compute the complex conjugate of X(k). */
   for (k=0; k<NUMPOINTS; k++)
   {
      x[k].imag = –x[k].imag;
   }
/* Compute the DFT of X*(k). */
   radix4(NUMPOINTS, (short *)x, (short *)W4);
   digit_reverse((int *)x, IIndex, JIndex, count);
/* Complex conjugate the output of the DFT and divide by N to get x(n). */
   for (n=0; n<NUMPOINTS; n++)
```

```
   {
      x[n].real = x[n].real/16;
      x[n].imag = (–x[n].imag)/16;
   }
/* x1(n) is the real part of x(n), and x2(n) is the imaginary part of x(n). */
   for (n=0; n<NUMDATA; n++)
   {
      x1[n] = x[n].real;
      x2[n] = x[n].imag;
   }
   return(0);
 }
```

Example D–3. radix4.c File

```
/*************************************************************************
 FILE
   radix4.c – radix-4 FFT function based on Burrus, Parks p .113
 *************************************************************************/
void radix4(int n, short x[], short w[])
{
   int             n1, n2, ie, ia1, ia2, ia3, i0, i1, i2, i3, j, k;
   short           t, r1, r2, s1, s2, co1, co2, co3, si1, si2, si3;
   n2 = n;
   ie = 1;
   for (k = n; k > 1; k >>= 2) {
      n1 = n2;
      n2 >>= 2;
      ia1 = 0;
      for (j = 0; j < n2; j++) {
         ia2 = ia1 + ia1;
         ia3 = ia2 + ia1;
         co1 = w[ia1 * 2 + 1];
         si1 = w[ia1 * 2];
         co2 = w[ia2 * 2 + 1];
         si2 = w[ia2 * 2];
         co3 = w[ia3 * 2 + 1];
         si3 = w[ia3 * 2];
         ia1 = ia1 + ie;
         for (i0 = j; i0 < n; i0 += n1) {
            i1 = i0 + n2;
            i2 = i1 + n2;
```

```
            i3 = i2 + n2;
            r1 = x[2 * i0] + x[2 * i2];
            r2 = x[2 * i0] – x[2 * i2];
            t = x[2 * i1] + x[2 * i3];
            x[2 * i0] = r1 + t;
            r1 = r1 – t;
            s1 = x[2 * i0 + 1] + x[2 * i2 + 1];
            s2 = x[2 * i0 + 1] – x[2 * i2 + 1];
            t = x[2 * i1 + 1] + x[2 * i3 + 1];
            x[2 * i0 + 1] = s1 + t;
            s1 = s1 – t;
            x[2 * i2] = (r1 * co2 + s1 * si2) >> 15;
            x[2 * i2 + 1] = (s1 * co2–r1 * si2)>>15;
            t = x[2 * i1 + 1] – x[2 * i3 + 1];
            r1 = r2 + t;
            r2 = r2 – t;
            t = x[2 * i1] – x[2 * i3];
            s1 = s2 – t;
            s2 = s2 + t;
            x[2 * i1] = (r1 * co1 + s1 * si1)  >>15;
            x[2 * i1 + 1] = (s1 * co1–r1 * si1)>>15;
            x[2 * i3] = (r2 * co3 + s2 * si3)  >>15;
            x[2 * i3 + 1] = (s2 * co3–r2 * si3)>>15;
         }
      }
      ie <<= 2;
   }
}
```

Example D–4. digit.c File

```
/*************************************************************************
 FILE
   digit.c – This is the C source code for a digit reversal function for
   a radix-4 FFT.
 *************************************************************************/
void digit_reverse(int *yx, unsigned short *JIndex, unsigned short *IIndex, int
count)
{
   int i;
   unsigned short I, J;
   int YXI, YXJ;
```

```
   for (i = 0; i<count; i++)
   {
      I = IIndex[i];
      J = JIndex[i];
      YXI = yx[I];
      YXJ = yx[J];
      yx[J] = YXI;
      yx[I] = YXJ;
   }
}
```

Example D–5. digitgen.c File

```
/*************************************************************************
 FILE
   digitgen.c – This is the C source code for a function used to generate
   index tables for a digit reversal function for a radix-4 FFT.
 *************************************************************************/

void R4DigitRevIndexTableGen(int n, int *count, unsigned short *IIndex, unsigned
short *JIndex)
{
   int j, n1, k, i;
   j = 1;
   n1 = n – 1;
   *count = 0;
   for(i=1; i<=n1; i++)
   {
      if(i < j)
      {
         IIndex[*count] = (unsigned short)(i–1);
         JIndex[*count] = (unsigned short)(j–1);
         *count = *count + 1;
      }

      k = n >> 2;
      while(k*3 < j)
      {
         j = j – k*3;
         k = k >> 2;
      }
      j = j + k;
   }
}
```

Example D–6. splitgen.c File

```
/*************************************************************************
 FILE
   splitgen.c – This is the C source code for a function used to generate
   tables for a split routine used to efficiently compute the DFT of a 2N-point
   real-valued sequence.
 *************************************************************************/

#include ”params.h”
void FftSplitTableGen(int N, COMPLEX *W, COMPLEX *A, COMPLEX *B)
{
   int k;
   for(k=0; k<N/2; k++)
   {

      A[k].real = 16383 – W[k].imag;
      A[k].imag = –W[k].real;
      A[k + N/2].real = 16383 – W[k].real;
      A[k + N/2].imag = W[k].imag;

      B[k].real = 16383 + W[k].imag;
      B[k].imag = W[k].real;
      B[k + N/2].real = 16383 + W[k].real;
      B[k + N/2].imag = –W[k].imag;
   }

}
```

Appendix E Optimized C-Callable ’C62xx Assembly Language Functions Used to Implement the DFT of Real Sequences

This appendix contains optimized C-callable ‘C62xx assembly language functions used to implement the DFT of real sequences.

E.1 Implementation Notes

The following lists usage, assumption, and limitations of the code.

Data format	All data and state variables are 16-bit signed integers (shorts). In this example, the decimal point is assumed to be between bits 15 and 14, thus the Q15 data format. For complex data and variables, the real and imaginary components are both Q15 numbers. From this data format, you can see that this code was developed for a fixed-point processor.
Memory	Complex data is stored in memory in imaginary/real pairs. The imaginary component is stored in the most significant halfword (16 bits) and the real component is stored in the least significant halfword, unless otherwise noted.
Endianess	The code is presented and tested in little endian format. Some modification to the code is necessary for big endian format.
Overflow	No overflow protection or detection is performed.

File	Description
split1.asm	C-callable ‘C62xx assembly version of the split function for the DFT of a 2*N*-point real sequence
split2.asm	C-callable ‘C62xx assembly version of the split function for the DFT of a 2*N*-point real sequences.
radix4.asm	Radix-4 FFT C-callable ‘C62xx assembly function.
digit.asm	Radix-4 digit reversal C-callable ‘C62xx assembly function.

Example E–1. split1.asm File

```
*===========================================================================
*
*   TEXAS INSTRUMENTS, INC.
*
*   Real FFT/IFFT split operation
*
*   Revision Date: 5/15/97
*
*   USAGE   This routine is C-callable, and can be called as:
*
*     void split1(int N, COMPLEX *X, COMPLEX *A, COMPLEX *B, COMPLEX *G)
*
*     N = 1/2 the number of samples of the real valued sequence
*     X = pointer to complex input array
*     A = pointer to complex coefficients
*     B = pointer to complex coefficients
*     G = pointer to complex output array
*
```

```
*           If routine is not to be used as a C-callable function,
*           then all instructions relating to stack should be removed.
*           Refer to comments of individual instructions. You will also
*           need to initialize values for all of the values passed, as these
*           are assumed to be in registers as defined by the calling
*           convention of the compiler, (refer to the C compiler reference
*           guide).
*
*   C Code  This is the C equivalent of the assembly code without
*           restrictions. Note that the assembly code is hand-optimized, and
*           restrictions may apply.
*
*           One small, but important note. The split functions uses word
*           loads to read imaginary/real pairs from memory. Because of this,
*           some C definitions may need to be endianess-dependent. Below are
*           the type definitions for COMPLEX for both big and little endian. Also,
*           the split function, as shown below, is written for big endian. See
*           comments in the code to see how to modify, if little endian is desired.
*
*                 LITTLE ENDIAN                   BIG ENDIAN
*           typedef struct {                    typedef struct {
*                 short real;                             short imag;
*                 short imag;                       short real;
*                 } COMPLEX;                     } COMPLEX;
*
*     void split(int N, COMPLEX *X, COMPLEX *A, COMPLEX *B, COMPLEX *G)
*     {
*
*     int k;
*     int Tr, Ti;
*
*           for (k=0; k<N; k++)
*           {
*                 Tr = (int)X[k].real * (int)A[k].real –
*                 (int)X[k].imag * (int)A[k].imag +
*                 (int)X[N–k].real * (int)B[k].real +
*                 (int)X[N–k].imag * (int)B[k].imag;
*
*                 G[k].real = (short)(Tr>>15);
*
*                 Ti = (int)X[k].imag * (int)A[k].real +
*                 (int)X[k].real * (int)A[k].imag +
*                 (int)X[N–k].real * (int)B[k].imag –
*                 (int)X[N–k].imag * (int)B[k].real;
*
*                 G[k].imag = (short)(Ti>>15);
*
*           }
*     }
*
*
```

```
*   DESCRIPTION
*     In many applications, the input is a sequence of real numbers.
*     If this condition is taken into consideration, additional computational
*     savings can be achieved because the FFT of a real sequence has some
*     symmetrical properties. The DFT of a 2N-point real sequence can be
*     efficiently computed using a N-point complex DFT and some additional
*     computations which have been implemented in this split function.
*     Note this split function can be used in the computation of FFTs and IFFTs.
*
*     The following steps are required in the computation of the FFT of
*     a real-valued sequence using the split function:
*
*     1.  Let g(n) be a 2N-point real sequence. From g(n), form the
*         the N-point complex-valued sequence, x(n) = x1(n) + jx2(n),
*         where x1(n) = g(2n) and x2(n) = g(2n + 1).
*
*     2.  Perform an N-point complex FFT on the complex-valued sequence,
*         x(n) –> X(k) = DFT{x(n)}. Note that the FFT can be any DFT method,
*         such as radix–2, radix–4, mixed radix, direct implementation of
*         the DFT, etc. However, the DFT output must be in normal order.
*
*     3.  The following additional computations are used to get G(k) from X(k),
*         and are implemented by the split function.
*           Gr(k) = Xr(k)Ar(k) – Xi(k)Ai(k) + Xr(N–k)Br(k) + Xi(N–k)Bi(k)
*                                             k = 0, 1, ..., N–1
*                                             and X(N) = X(0)
*           Gi(k) = Xi(k)Ar(k) + Xr(k)Ai(k) + Xr(N–k)Bi(k) – Xi(N–k)Br(k)
*
*         Note that only N-points of the 2N-point sequence of G(k) are computed
*         in the above equations. Because the DFT of a real–sequence has
*         symmetric properties, we can easily compute the remaining N points
*         of G(k) with the following equations.
*
*           Gr(N) = Gr(0) – Gi(0)
*           Gi(N) = 0
*
*           Gr(2N–k) = Gr(k)
*                                             k = 1, 2, ..., N–1
*           Gi(2N–k) = –Gi(k)
*
*         As you can see, the split function assumes that A(k) and B(k),
*         which are sine and cosine coefficient, are pre-computed. The
*         C–code can be used to initialize A(k) and B(k).
*
*   for(k=0; k<N; k++)
*   {
*     A[k].imag = (short)(16383.0*(–cos(2*PI/(double)(2*N)*(double)k)));
*     A[k].real = (short)(16383.0*(1.0 – sin(2*PI/(double)(2*N)*(double)k)));
*     B[k].imag = (short)(16383.0*(cos(2*PI/(double)(2*N)*(double)k)));
*     B[k].real = (short)(16383.0*(1.0 + sin(2*PI/(double)(2*N)*(double)k)));
*   }
*
```

```
*     The following steps are required in the computation of the IFFT of
*     a complex-valued frequency domain sequence that was derived from
*     a real sequence using the split function:
*
*     1.  Let G(k) be a 2N-point complex-valued sequence derived from a real
*         valued sequence g(n). We want to get back g(n) from G(k) –>
*         g(n) = IDFT{G(k)}. However, we want to apply the same techniques
*         as we did with the forward FFT, use a N-point IFFT. This can be
*         accomplished by the following equations.
*
*           Xr(k) = Gr(k)IAr(k) – Gi(k)IAi(k) + Gr(N–k)IBr(k) + Gi(N–k)IBi(k)
*                                               k = 0, 1, ..., N–1
*                                               and G(N) = G(0)
*           Xi(k) = Gi(k)IAr(k) + Gr(k)IAi(k) + Gr(N–k)IBi(k) – Gi(N–k)IBr(k)
*
*     2.  Perform the N-point inverse DFT of X(k) –> x(n) = x1(n) + jx2(n) =
*         IDFT{X(k)}. Note that the IDFT can be any method, but must have an
*         output that is in normal order.
*
*     3.  g(n) can then be found from x(n).
*
*           g(2n) = x1(n)
*                                               n = 0, 1, ..., N–1
*           g(2n+1) = x2(n)
*
*     As you can see, the split function can be used for both the forward and
*     inverse FFTs; however, the pre–computed coefficients are slightly
      different.
*     The following C code can be used to initialize IA(k) and IB(k).
*
*   for(k=0; k<N; k++)
*   {
*     IA[k].imag = –(short)(16383.0*(–cos(2*PI/(double)(2*N)*(double)k)));
*     IA[k].real = (short)(16383.0*(1.0 – sin(2*PI/(double)(2*N)*(double)k)));
*     IB[k].imag = –(short)(16383.0*(cos(2*PI/(double)(2*N)*(double)k)));
*     IB[k].real = (short)(16383.0*(1.0 + sin(2*PI/(double)(2*N)*(double)k)));
*   }
*
*     Note that IA(k) is the complex conjugate of A(k), and IB(k) is the complex
*     conjugate of B(k).
*
*   TECHNIQUES
*     32–bit loads are used to load two 16–bit loads.
*
*   ASSUMPTIONS
*
*     A, B, X, and G are stored as imaginary/real pairs.
*
*     Big endian is used. If little endian is desired, modification to
*     the code is required. See comments in the code for which instructions
*     require modification for little endian use.
*
```

```
*   MEMORY NOTE
*     A, B, X and G arrays should be aligned to word boundaries. Also,
*     A and B should be aligned such that they do not generate a memory
*     hit with X.
*
*     A, B, X, and G are all complex data and are required to be stored
*     as imaginary/real pairs in memory, regardless of endianess. In other
*     words, a load word from any of these arrays should result in the imaginary
*     component in the upper 16 bits of a register, and the real component in the
*     lower 16 bits.
*
*   CYCLES  4*N + 32
*
*============================================================================
N           .set  a4    ; Argument 1 – number of points in the FFT
XPtr        .set  b4    ; Argument 2 – pointer to complex data
APtr        .set  a6    ; Argument 3 – pointer to A complex coefficients
BPtr        .set  b6    ; Argument 4 – pointer to B complex coefficients
GPtr        .set  a8    ; Argument 5 – pointer to output buffer
aI_aR       .set  a0    ; Coefficient value loaded from APtr
XNPtr       .set  a1    ; Pointer to the bottom of the data buffer
                        ; This pointer gets decremented through the loop.
x2I_x2R     .set  a2    ; Data value loaded from XNPtr
xRaR        .set  a3    ; Product
xIaI        .set  a5
x2RbR       .set  a7
x2IbI       .set  a9
re1         .set  a10
re2         .set  a11
real        .set  a12
CNT         .set  b0    ; Counter for looping
xI_xR       .set  b1
bI_bR       .set  b2
xIaR        .set  b5
xRaI        .set  b7
x2IbR       .set  b8
x2RbI       .set  b9
im1         .set  b10
im2         .set  b11
imag        .set  b12
            .global _split1
_split1:
    sub     .d2   B15,24,B15        ; Allocate space on the stack.
    stw     .d2   A10,*B15++[1]     ; Push A10 onto the stack.
    stw     .d2   A11,*B15++[1]     ; Push A11 onto the stack.
    stw     .d2   A12,*B15++[1]     ; Push A12 onto the stack.
    stw     .d2   B10,*B15++[1]     ; Push B10 onto the stack.
||  sub     .l2x  N,1,CNT           ; Initialize loop count register.
    stw     .d2   B11,*B15++[1]     ; Push B11 onto the stack.
||  shl     .s1   N,2,N             ; Calculate offset to initialize
                                    ; a pointer to the bottom of the
                                    ; input data buffer.
    stw     .d2   B12,*B15++[1]     ; Push B12 onto the stack.
```

```
||  add     .l1x  N,XPtr,XNPtr      ; XNPtr –> yx[N]

    ; Because there are delay slots in loads, we will begin by
    ; SW pipelining the split operations – in other words, while
    ; we are finishing the current loop iteration, we will be
    ; beginning the next.
    ldw     .d1   *APtr++[1],aI_aR        ; Load a coefficient pointed by APtr.
||  ldw     .d2   *XPtr++[1],xI_xR        ; Load a data value pointed by XPtr.
    nop                                   ; Fill a delay slot.
    ldw     .d1   *XNPtr--[1],x2I_x2R     ; Load a data value pointed by XNPtr.
||  ldw     .d2   *BPtr++[1],bI_bR        ; Load a coefficient pointed by BPtr.
    nop                                   ; Fill a delay slot.

    ldw     .d1   *APtr++[1],aI_aR        ; Load the next value pointed by APtr.
                                          ; (Note that it will not overwrite
                                          ; the current value of aI_aR until
                                          ; 4 delay slots later).
||  ldw     .d2   *XPtr++[1],xI_xR        ; Load the next value pointed by XPtr.
    ; for performing the multiplies, we take advantage of the feature
    ; feature that allows you to choose the operands from either the upper
    ; or lower halves of the register.

    mpy     .m1x  xI_xR,aI_aR,xRaR        ; xRaR = xR * aR – mpy lower * lower
||  mpyhl   .m2x  xI_xR,aI_aR,xIaR        ; xIaR = xI * aR – mpy upper * lower
    mpylh   .m2x  xI_xR,aI_aR,xRaI        ; xRaI = xR * aI – mpy lower * upper
||  mpyh    .m1x  xI_xR,aI_aR,xIaI        ; xIaI = xI * aI – mpy upper * upper
||  ldw     .d1   *XNPtr--[1],x2I_x2R     ; load a data value pointed by XNPtr
||  ldw     .d2   *BPtr++[1],bI_bR        ; load a coefficient pointed by BPtr
    mpy     .m1x  x2I_x2R,bI_bR,x2RbR     ; x2RbR = x2R * bR – mpy lower * lower
||  mpyhl   .m2x  x2I_x2R,bI_bR,x2IbR     ; x2IbR = x2I * bR – mpy upper * lower
    mpylh   .m2x  x2I_x2R,bI_bR,x2RbI     ; x2RbI = x2R * bI – mpy lower * upper
||  mpyh    .m1x  x2I_x2R,bI_bR,x2IbI     ; x2IbI = x2I * bI – mpy upper * upper
||  sub     .l1   xRaR,xIaI,re1           ; re1 = xRaR – xIaI
||  add     .l2   xRaI,xIaR,im1           ; im1 = xRaI + xIaR
||  ldw     .d1   *APtr++[1],aI_aR        ; 3rd load of aI_aR
||  ldw     .d2   *XPtr++[1],xI_xR        ; 3rd load of xI_xR
    ; the second loads of xI_xR and aI_aR are now avaiable, thus we can use
    ; them to begin the 2nd iteration of X's and A's multiplies

    mpy     .m1   xI_xR,aI_aR,xRaR        ; xRaR = xR * aR – mpy lower * lower
||  mpyhl   .m2x  xI_xR,aI_aR,xIaR        ; xIaR = xI * aR – mpy upper * lower
    mpylh   .m2   xI_xR,aI_aR,xRaI        ; xRaI = xR * aI – mpy lower * upper
||  mpyh    .m1x  xI_xR,aI_aR,xIaI        ; xIaI = xI * aI – mpy upper * upper
||  add     .l1   x2RbR,x2IbI,re2         ; re2 = x2RbR + x2IbI
||  sub     .l2   x2RbI,x2IbR,im2         ; im2 = x2RbI – x2IbR
||  ldw     .d1   *XNPtr--[1],x2I_x2R     ; 3rd load of x2I_x2R
||  ldw     .d2   *BPtr++[1],bI_bR        ; 3rd load of bI_bR
    ; The second loads of x2I_x2R and bI_bR are now available, thus we can use
    ; them to begin the 2nd iteration of X2's and B's multiplies.

    mpy     .m1   x2I_x2R,bI_bR,x2RbR     ; x2RbR = x2R * bR – mpy lower * lower
||  mpyhl   .m2x  x2I_x2R,bI_bR,x2IbR     ; x2IbR = x2I * bR – mpy upper * lower
||  add     .l1   re1,re2,real            ; real = re1 + re2
||  add     .l2   im1,im2,imag            ; imag = im1 + im2
```

```
||  b .s2   LOOP                        ; Branch to LOOP – Note that this is the
                                        ; branch for the first time through
                                        ; the loop. Because of this, we only
                                        ; need to do count–1 branches to LOOP
                                        ; within LOOP.
    mpylh   .m2   x2I_x2R,bI_bR,x2RbI   ; x2RbI = x2R * bI – mpy lower * upper
||  mpyh    .m1x  x2I_x2R,bI_bR,x2IbI   ; x2IbI = x2I * bI – mpy upper * upper
||  sub     .l1   xRaR,xIaI,re1         ; rel = xRaR – xIaI
||  add     .l2   xRaI,xIaR,im1         ; im1 = xRaI + xIaR
||  shr     .s1   real,15,real          ; real = real >> 15
||  shr     .s2   imag,15,imag          ; imag = imag >> 15
||  ldw     .d1   *APtr++[1],aI_aR      ; 4th load of aI_aR
||  ldw     .d2   *XPtr++[1],xI_xR      ; 4th load of xI_xR
      ; CAUTION – because of SW pipelining, we actually load more values
      ; of aI_aR, xI_xR, bI_bR, and x2I_x2R than we actually use. Thus,
      ; make sure these arrays are NOT aligned to a boundary close to the
      ; edge of illegal memory.
LOOP: ; this loop is executed N times
    mpy     .m1   xI_xR,aI_aR,xRaR      ; xRaR = xR * aR – mpy lower * lower
||  mpyhl   .m2x  xI_xR,aI_aR,xIaR      ; xIaR = xI * aR – mpy upper * lower
;|| sth     .d1   imag,*GPtr++[1]       ; Store imag in output buffer.
                                        ; CAUTION – Big Endian specific code
                                        ; If Little Endian is desired,
                                        ; replace this line with:
||  sth     .d1   real,*GPtr++[1]
||  [CNT]   sub   .l2   CNT,1,CNT       ; If (CNT != 0), CNT = CNT – 1.

    mpylh   .m2   xI_xR,aI_aR,xRaI      ; xRaI = xR * aI – mpy lower * upper
||  mpyh    .m1x  xI_xR,aI_aR,xIaI      ; xIaI = xI * aI – mpy upper * upper
||  add     .l1   x2RbR,x2IbI,re2       ; re2 = x2RbR + x2IbI
||  sub     .l2   x2RbI,x2IbR,im2       ; im2 = x2RbI – x2IbR
||  ldw     .d1   *XNPtr––[1],x2I_x2R   ; Next load of x2I_x2R
||  ldw     .d2   *BPtr++[1],bI_bR      ; Next load of bI_bR

    mpy     .m1   x2I_x2R,bI_bR,x2RbR   ; x2RbR = x2R * bR – mpy lower * lower
||  mpyhl   .m2x  x2I_x2R,bI_bR,x2IbR   ; x2IbR = x2I * bR – mpy upper * lower
||  add     .l1   re1,re2,real          ; real = re1 + re2
||  add     .l2   im1,im2,imag          ; imag = im1 + im2
;|| sth     .d1   real,*GPtr++[1]       ; Store real in output buffer.
                                        ; CAUTION – Big Endian specific code
                                        ; If Little Endian is desired,
                                        ; replace this line with:
||  sth     .d1   imag,*GPtr++[1]
||  [CNT]   b     .s2   LOOP            ; If (CNT != 0), branch to LOOP.
    mpylh   .m2x  x2I_x2R,bI_bR,x2RbI   ; x2RbI = x2R * bI – mpy lower * upper
||  mpyh    .m1x  x2I_x2R,bI_bR,x2IbI   ; x2IbI = x2I * bI – mpy upper * upper
||  sub     .l1   xRaR,xIaI,re1         ; rel = xRaR – xIaI
||  add     .l2   xRaI,xIaR,im1         ; im1 = xRaI + xIaR
||  shr     .s1   real,15,real          ; real = real >> 15
||  shr     .s2   imag,15,imag          ; imag = imag >> 15
||  ldw     .d1   *APtr++[1],aI_aR      ; next load of aI_aR
```

```
||  ldw     .d2   *XPtr++[1],xI_xR        ; next load of xI_xR
 ; end of LOOP
    ldw     .d2   *--B15[1],B12           ; Pop B12 from the stack.
    ldw     .d2   *--B15[1],B11           ; Pop B11 from the stack.
    ldw     .d2   *--B15[1],B10           ; Pop B10 from the stack.
    ldw     .d2   *--B15[1],A12           ; Pop A12 from the stack.
    ldw     .d2   *--B15[1],A11           ; Pop A11 from the stack.
    ldw     .d2   *--B15[1],A10           ; Pop A10 from the stack.
    b .s2   B3                            ; Function return add   .d2
B15,24,B15        ; Deallocate space from the stack.
    nop           4                 ; Fill delay slots.
```

Example E–2. split2.asm File

```
*==============================================================================
*
*   TEXAS INSTRUMENTS, INC.
*
*   Real FFT/IFFT split operation
*
*   Revision Date: 6/4/97
*
*   USAGE   This routine is C-Callable and can be called as:
*
*           void split2(int N, COMPLEX *X, COMPLEX *X1, COMPLEX *X2)
*
*           N = the number of samples of each real valued sequence
*           X = pointer to complex input array
*           X1 = pointer to complex array DFT result of sequence 1
*           X2 = pointer to complex array DFT result of sequence 2
*
*           If routine is not to be used as a C-callable function,
*           then all instructions relating to stack should be removed.
*           Refer to comments of individual instructions. You will also
*           need to initialize values for all of the values passed, as these
*           are assumed to be in registers as defined by the calling
*           convention of the compiler, (refer to the C compiler reference
*           guide).
*
*   C Code  This is the C equivalent of the assembly code without
*           restrictions. Note that the assembly code is hand optimized, and
*           restrictions may apply.
*
*           One small, but important note. The split functions uses word loads
*           to read imaginary/real pairs from memory. Because of this, some C
*           definitions may need to be endianess-dependent. Also,
*           the split function as shown below is written for little endian.
*
*     typedef struct {
*           short real;
*           short imag;
*           } COMPLEX;
*
*     void split2(int N, COMPLEX *X, COMPLEX *X1, COMPLEX *X2)
*     {
*
```

```
*           int k;
*
*           X1[0].real = X[0].real;
*           X1[0].imag = 0;
*
*           X2[0].real = X[0].imag;
*           X2[0].imag = 0;
*
*           X1[N/2].real = X[N/2].real;
*           X1[N/2].imag = 0;
*
*           X2[N/2].real = X[N/2].imag;
*           X2[N/2].imag = 0;
*
*
*           for (k=1; k<N/2; k++)
*           {
*                 X1[k].real = (X[k].real + X[N–k].real)/2;
*                 X1[k].imag = (X[k].imag – X[N–k].imag)/2;
*
*                 X2[k].real = (X[k].imag + X[N–k].imag)/2;
*                 X2[k].imag = (X[N–k].real – X[k].real)/2;
*
*                 X1[N–k].real = X1[k].real;
*                 X1[N–k].imag = –X1[k].imag;
*
*                 X2[N–k].real = X2[k].real;
*                 X2[N–k].imag = –X2[k].imag;
*           }
*
*     }
*
*
*    DESCRIPTION
*      In many applications, the input is a sequence of real numbers.
*      If this condition is taken into consideration, additional computational
*      savings can be achieved because the FFT of a real sequence has some
*      symmetrical properties. The DFT of a two N-point real sequence can be
*      efficiently computed using one N-point complex DFT and some additional
*      computations which have been implemented in this split function.
*      Note that this split function can be used in the computation of FFTs and IFFTs.
*
*      The following steps are required in the computation of the FFT of
*      two real-valued sequence using the split function:
*
*      Assume we have two real-valued sequences of length N – x1[n] and x2[n].
*      The DFT of x1[n] and x2[n] can be computed with one complex–valued DFT of
*      length N, as shown above, by following this algorithm.
*
*      1.  Form the complex-valued sequence x[n] from x1[n] and x2[n]
*
*            xr[n] = x1[n]  and   xi[n] = x2[n],   0,1, ..., N–1
*
```

```
*     Note that if the sequences x1[n] and x2[n] are coming from another algorithm
*     or a data acquisition driver, this step may be eliminated, if these put the
*     data in the complex-valued format correctly.
*
*     2.  Compute X[k] = DFT{x[n]}
*
*     This can be the direct form DFT algorithm or an FFT algorithm. If using an
*     FFT algorithm, make sure the output is in normal order – bit reversal is
*     performed.
*
*     3.  Compute the following equations to get the DFTs of x1[n] and x2[n].
*         These are the equations that this file implements.
*
*           X1r[0] = Xr[0]
*           X1i[0] = 0
*
*           X2r[0] = Xi[0]
*           X2i[0] = 0
*
*           X1r[N/2] = Xr[N/2]
*           X1i[N/2] = 0
*
*           X2r[N/2] = Xi[N/2]
*           X2i[N/2] = 0
*
*           for k = 1,2,3, ...., N/2–1
*                 X1r[k] = (Xr[k] + Xr[N–k])/2
*                 X1i[k] = (Xi[k] – Xi[N–k])/2
*                 X1r[N–k] = X1r[k]
*                 X1i[N–k] = X1i[k]
*
*                 X2r[k] = (Xi[k] + Xi[N–k])/2
*                 X2i[k] = (Xr[N–k] – Xr[k])/2
*                 X2r[N–k] = X2r[k]
*                 X2i[N–k] = X2i[k]
*
*
*     4. Form X[k] from X1[k] and X2[k]
*
*                 for k = 0,1, ..., N–1
*                 Xr[k] = X1r[k] – X2i[k]
*                 Xi[k] = X1i[k] + X2r[k]
*
*     5.  Compute x[n] = IDFT{X[k]}
*
*     This can be the direct-form IDFT algorithm, or an IFFT algorithm. If using
*     an IFFT algorithm, make sure the output is in normal order – bit reversal is
*     performed.
*
```

```
*   TECHNIQUES
*     32-bit loads are used to load two 16-bit loads.
*
*   ASSUMPTIONS
*
*     X, X1, and X2 are stored as imaginary/real pairs.
*
*     Little endian is used. If little endian is desired, modification to
*     the code is required.
*
*   MEMORY NOTE
*     X must be aligned to a 32-bit boundary.
*
*   CYCLES  5*(N/2–1) + 29
*
*============================================================================
N     .set  a4
XPtr        .set  b4
X1Ptr       .set  a6
X2Ptr       .set  b6
CNT         .set  b0
XNmkPtr     .set  a3
N4          .set  a0
XiXr        .set  b2
XNiXNr      .set  a2
X1NmkPtr    .set  a1
X2NmkPtr    .set  b1
X2rX1r      .set  a8
X1iX2i      .set  b8
X1r         .set  a9
X1i         .set  b9
X2r         .set  a10
X2i         .set  b10
X1Nr        .set  a12
X1Ni        .set  b12
X2Nr        .set  a13
X2Ni        .set  b13
nullA       .set  a14
nullB       .set  b5
            .global_split2
_split2:
    subaw   .d2   B15,10,B15             ; Allocate space on the stack
    ldh     .d2   *XPtr,X1r              ; X1r = Xr[0]
    add     .l    B15,4,A15              ; A15 points to the stack as well
||  ldh     .d2   *+XPtr[1],X2r          ; X2r = Xi[0]
    stw     .d1   A10,*A15++[2]          ; Push A10 onto the stack.
||  stw     .d2   B10,*B15++[2]          ; Push B10 onto the stack.
    stw     .d1   A11,*A15++[2]          ; Push A11 onto the stack.
||  stw     .d2   B11,*B15++[2]          ; Push B11 onto the stack.
    stw     .d1   A12,*A15++[2]          ; Push A12 onto the stack.
||  stw     .d2   B12,*B15++[2]          ; Push B12 onto the stack.
    stw     .d1   A13,*A15++[2]          ; Push A13 onto the stack.
||  stw     .d2   B13,*B15++[2]          ; Push B13 onto the stack.
    stw     .d1   A14,*A15++[2]          ; Push A14 onto the stack.
```

```
||  stw     .d2   B14,*B15++[2]           ; Push B14 onto the stack.
    sth     .d1   X1r,  *X1Ptr            ; X1r[0]=Xr[0]
    shl     .s1   N,    2,    N4          ; N4 = 4*N
||  sth     .d2   X2r,  *X2Ptr            ; X2r[0]=Xi[0]
||  zero    .l2   nullB                   ; nullA = 0
||  zero    .l1   nullA                   ; nullB = 0
    sub     .l1   N4,   4,    N4          ; N4 = N4 – 1
||  sth     .d1   nullA,      *+X1Ptr[1]  ; X1i[0]=0
||  sth     .d2   nullB,      *+X2Ptr[1]  ; X2i[0]=0
    add     .l1x  N4,   XPtr, XNmkPtr     ; XNmkPtr –> X[N–1]
||  add     .l2   XPtr, 4,    XPtr        ; XPTR –> X[1]

    ldw     .d2   *XPtr++[1], XiXr        ; Load X[k].real and X2[k].imag.
||  ldw     .d1   *XNmkPtr––[1],    XNiXNr      ; Load X[N–k].real and X[N–k].imag.
    shr     .s2x  N,    1,    CNT             ; CNT = N/2
    sub     .s2   CNT,  1,    CNT             ; CNT = N/2 – 1
    add     .l1   N4,   X1Ptr,      X1NmkPtr   ; X1NmkPtr –> X1[N–1]
||  add     .l2x  N4,   X2Ptr,      X2NmkPtr   ; X2NmkPtr –> X2[N–1]
    add     .l    X1Ptr,      4,    X1Ptr      ; X1Ptr –> X1[1]
||  add     .l    X2Ptr,      4,    X2Ptr      ; X2Ptr –> X2[1]

    add2    .s1x  XiXr, XNiXNr, X2rX1r    ; X2[k].real = X[k].imag + X[N–k].imag
                                          ; (upper 16 bits)
                                          ; X1[k].real = X[k].real + X[N–k].real
                                          ; (lower 16 bits)
||  sub2    .s2x  XiXr, XNiXNr, X1iX2i    ; X1[k].imag = X[k].imag – X[N–k].imag
                                          ; (upper 16 bits)
                                          ; X2[k].imag = X[k].real – X[N–k].real
                                          ; (lower 16 bits)
||  ldw     .d2   *XPtr++[1], XiXr        ; load X[k].real and X2[k].imag
||  ldw     .d1   *XNmkPtr––[1], XNiXNr   ; load X[N–k].real and X[N–k].imag
    shr     .s1   X2rX1r,     17,   X2r   ; X2[k].real = (X[k].imag +
                                          ; X[N–k].imag)/2
                                          ; X2r = X2rX1r>>17
||  sub2    .s2   nullB, X1iX2i, X2i      ; X2[k].imag = X[N–k].real – X[k].real
                                          ; (lower 16 bits)
                                          ; upper 16 bits are don’t cares
    shr     .s2   X1iX2i, 17, X1i         ; X1[k].imag = (X[k].imag +
                                          ; X[N–k].imag)/2
                                          ; X1i = X1iX2i>>17
||  b .s1   LOOP                          ; Branch for the first time through loop.
LOOP:
    ext     .s1   X2rX1r, 16,17, X1r      ; X1[k].real = X1[k].real/2
||  ext     .s2   X2i,  16,17,      X2i   ; X2[k].imag = X2[k].imag/2
||  sth     .d1   X1i,  *+X1Ptr[1]        ; Store X1[k].imag.
||  sth     .d2   X2r,  *X2Ptr++[1]       ; Store X2[k].real.
    mv.l1   X1r,  X1Nr                    ; X1[N–k].real = X1[k].real
||  mv.s1   X2r,  X2Nr                    ; X2[N–k].real = X2[k].real
||  sub     .l2   nullB,      X1i,  X1Ni  ; X1[N–k].imag = –X1[k].imag
||  sub     .s2   nullB,      X2i,  X2Ni  ; X2[N–k].imag = –X2[k].imag
||  sth     .d2   X2i,  *X2Ptr++[1]       ; Store X2[k].imag.
```

```
||  sth     .d1   X1r,  *X1Ptr++[2]       ; Store X1[k].real.
    add2    .s1x  XiXr, XNiXNr, X2rX1r    ; X2[k].real = X[k].imag + X[N–k].imag
                                          ; (upper 16 bits)
                                          ; X1[k].real = X[k].real + X[N–k].real
                                          ; (lower 16 bits)
||  sub2    .s2x  XiXr, XNiXNr, X1iX2i    ; X1[k].imag = X[k].imag – X[N–k].imag
                                          ; (upper 16 bits)
                                          ; X2[k].imag = X[k].real – X[N–k].real
                                          ; (lower 16 bits)
||  ldw     .d2   *XPtr++[1], XiXr        ; Load X[k].real and X2[k].imag.
||  ldw     .d1   *XNmkPtr––[1], XNiXNr   ; Load X[N–k].real and X[N–k].imag.
    shr     .s1   X2rX1r,     17,   X2r   ; X2[k].real = (X[k].imag +
                                          ; X[N–k].imag)/2
                                          ; X2r = X2rX1r>>17
||  sub2    .s2   nullB, X1iX2i, X2i      ; X2[k].imag = X[N–k].real – X[k].real
                                          ; (lower 16 bits)
                                          ; Upper 16 bits are don’t cares.
||  sth     .d1   X1Ni, *+X1NmkPtr[1]     ; Store X1[N–k].imag.
||  sth     .d2   X2Nr, *X2NmkPtr––[2]    ; Store X2[N–k].real.
||  [CNT]   sub   .l2   CNT,  1,    CNT   ; decrement loop counter
    shr     .s2   X1iX2i,     17,   X1i   ; X1[k].imag = (X[k].imag +
                                          ; X[N–k].imag)/2
                                          ; X1i = X1iX2i>>17
||  sth     .d1   X1Nr, *X1NmkPtr––[2]    ; Store X1[N–k].real.
||  sth     .d2   X2Ni, *+X2NmkPtr[3]     ; Store X2[N–k].imag.
||  [CNT]   b     .s1   LOOP              ; Conditional branch.
; LOOP END
    ldw     .d1   *––A15[2],A14           ; Pop A14 from the stack.
||  ldw     .d2   *––B15[2],B14           ; Pop B14 from the stack.
    ldw     .d1   *––A15[2],A13           ; Pop A13 from the stack.
||  ldw     .d2   *––B15[2],B13           ; Pop B13 from the stack.

    ldw     .d2   *––B15[2],B12           ; Pop B12 from the stack.
||  ldw     .d1   *––A15[2],A12           ; Pop A12 from the stack.
    ldw     .d1   *––A15[2],A11           ; Pop A11 from the stack.
||  ldw     .d2   *––B15[2],B11           ; Pop B11 from the stack.
||  b .s2   B3                            ; Function return.
    ldw     .d2   *––B15[2],B10           ; Pop B10 from the stack.
||  ldw     .d1   *––A15[2],A10           ; Pop A10 from the stack.
    addaw   .d2   B15,10,B15              ; Deallocate space from the stack.
    nop           3                       ; Fill delay slots.
```

Example E–3. radix4.asm File

```
*************************************************************************
*
*   TEXAS INSTRUMENTS INC.
*
*   COMPLEX FFT (Radix 4)
*
*   Revision Data: 04/28/97
*
*   USAGE This routine is C-callable and can the called as
*
*   void radix4(int n, short x[], short w[])
*
*     n    --- FFT size (power of 4) (input)
*     x[]  --- input and output sequences (dim–n) (input/output)
*     w[]  --- FFT coefficients (dim–n) (input)
*
*     If the routine is not to be used as a C-callable function,
*     then all instructions relating to dummy should be removed.
*     Refer to comments of individual instructions. You will also
*     need to initialize values for all the values passed as these
*     are assumed to be in registers as defined by the calling
*     convention of the compiler, (refer to the C compiler reference
*     guide.)
*
*   C CODE
*
*     This is the C equivalent of the assembly code, without the
*     assumptions listed below. Note that the assembly code is hand-
*     optimized and assumptions apply.
*
*     SOURCE:Burrus, Parks p .113
*
*   void radix4(int n, short x[], short w[])
*   {
*     int             n1, n2, ie, ia1, ia2, ia3, i0, i1, i2, i3, j, k;
*     short           t, r1, r2, s1, s2, co1, co2, co3, si1, si2, si3;
*
*     n2 = n;
*     ie = 1;
*     for (k = n; k > 1; k >>= 2) {
*           n1 = n2;
*           n2 >>= 2;
*           ia1 = 0;
*           for (j = 0; j < n2; j++) {
*                 ia2 = ia1 + ia1;
*                 ia3 = ia2 + ia1;
*                 co1 = w[ia1 * 2 + 1];
*                 si1 = w[ia1 * 2];
*                 co2 = w[ia2 * 2 + 1];
*                 si2 = w[ia2 * 2];
*                 co3 = w[ia3 * 2 + 1];
*                 si3 = w[ia3 * 2];
*                 ia1 = ia1 + ie;
```

```
*                 for (i0 = j; i0 < n; i0 += n1) {
*                       i1 = i0 + n2;
*                       i2 = i1 + n2;
*                       i3 = i2 + n2;
*                       r1 = x[2 * i0] + x[2 * i2];
*                       r2 = x[2 * i0] – x[2 * i2];
*                       t = x[2 * i1] + x[2 * i3];
*                       x[2 * i0] = r1 + t;
*                       r1 = r1 – t;
*                       s1 = x[2 * i0 + 1] + x[2 * i2 + 1];
*                       s2 = x[2 * i0 + 1] – x[2 * i2 + 1];
*                       t = x[2 * i1 + 1] + x[2 * i3 + 1];
*                       x[2 * i0 + 1] = s1 + t;
*                       s1 = s1 – t;
*                       x[2 * i2] = (r1 * co2 + s1 * si2) >> 15;
*                       x[2 * i2 + 1] = (s1 * co2–r1 * si2)>>15;
*                       t = x[2 * i1 + 1] – x[2 * i3 + 1];
*                       r1 = r2 + t;
*                       r2 = r2 – t;
*                       t = x[2 * i1] – x[2 * i3];
*                       s1 = s2 – t;
*                       s2 = s2 + t;
*                       x[2 * i1] = (r1 * co1 + s1 * si1)  >>15;
*                       x[2 * i1 + 1] = (s1 * co1–r1 * si1)>>15;
*                       x[2 * i3] = (r2 * co3 + s2 * si3)  >>15;
*                       x[2 * i3 + 1] = (s2 * co3–r2 * si3)>>15;
*                 }
*           }
*           ie <<= 2;
*     }
*   }
*
*   DESCRIPTION
*
*     This routine is used to compute FFT of a complex sequence
*     of size n, a power of 4, with ”decimation-in-frequency
*     decomposition” method. The output is in digit-reversed
*     order. Each complex value is with interleaved 16-bit real
*     and imaginary parts.
*
*   TECHNIQUES
*        1. Loading input x as well as coefficient w in word.
*        2. Both loops j and i0 shown in the C code are placed in the
*           INNERLOOP of the assembly code.
*
*   ASSUMPTIONS
*     4 <= n <= 65536
*     Both input x and coefficient w should be aligned on word
*     boundary.
*
```

```
*   MEMORY NOTE
*     Align x and w on different word boundaries to minimize
*     memory bank hits. There are N/4 memory bank hits total
*
*   CYCLES
*     LOGBASE4(N) * (10 * N/4 + 33) + 7 + N/4
*
*******************************************************************************
      .global dummy
      .global _radix4
      .bss    dummy,52                    ; Reserve space for dummy.
      .text
_radix4:
      MVK   .S1   dummy,      A0          ; New dummy pointer in A0 and B1
||    MVK   .S2   dummy,      B1
      MVKH  .S1   dummy,      A0          ; New dummy pointer in A0 and B1
||    MVKH  .S2   dummy,      B1.
      STW   .D2   B3,         *B1         ; Push return address on dummy.
      STW   .D1   A10,        *+A0[1]     ; Push A10 on dummy.
||    STW   .D2   B10,        *+B1[2]     ; Push B10 on dummy.
*** BEGIN Benchmark Timing ***
B_START:
      MVK   .S1   32,   A1                ; A1 = 32
||    LMBD  .L1   1,    A4,   A2          ; 31 – log2(n)
||    SHR   .S2X  A4,   2,    B6          ; n2 = n / 4
||    ZERO  .L2   B7                      ; Mask
||    STW   .D1   A11,  *+A0[3]           ; Push A11 on dummy.
||    STW   .D2   B11,  *+B1[4]           ; Push B11 on dummy.
      SUB   .L1   A1,   A2,   A4          ; log2(n)+1 (circ buff size in bytes)
||    SHR   .S1   A4,   1,    A7          ; 2 * n2 = n / 2, a–side
||    SHR   .S2X  A4,   1,    B9          ; 2 * n2 = n / 2, b–side
||    MV    .L2   B6,   B0                ; n / 4
||    STW   .D1   A12,  *+A0[5]           ; Push A12 on dummy.
||    STW   .D2   B12,  *+B1[6]           ; Push B12 on dummy.
      SHL   .S1   A4,   16,   A4          ; Shift into BK0 field.
||    MVC   .S2   B4,   IRP               ; Save off x.
||    STW   .D1   A13,  *+A0[7]           ; Push A13 on dummy.
||    STW   .D2   B13,  *+B1[8]           ; Push B13 on dummy.
      ADDK  .S1   0404h,A4                ; A5, B5 set circular mode on BK0
||    MVK   .S2   1,    B8                ; ie = 1
||    STW   .D1   A14,  *+A0[9]           ; Push A14 on dummy.
||    STW   .D2   B14,  *+B1[10]          ; Push B14 on dummy.
      MVC   .S2X  A4,   AMR               ; Load AMR.
||    STW   .D1   A15,  *+A0[11]          ; Push A15 on dummy.
||    STW   .D2   B15,  *+B1[12]          ; Push B15 on dummy.
||    SUB   .L2   B0,   1,    B0          ; Loop coutner = n / 4 – 1
K_LOOP:
      MV    .L2   B4,   B5                ; Reset X load pointer.
||    MV    .L1X  B4,   A5                ; Reset X store pointer.
||    ADD   .D2   B0,   1,    B1          ; i = loop counter + 1
||    MV    .D1   A6,   A1                ; Setup twiddle factor pointer
      ZERO  .S1   A4                      ; j = 0
||    SUBAW .D1   A5,   A7,   A5          ; Setup for first preincrement
||    AND   .S2   B1,   B7,   B1          ; J loop twiddle reload test
      SUBAW .D1   A5,   A7,   A5          ; Setup for first preincrement
```

```
||    MPY   .M2   B1,   1,    B2          ; J loop twiddle reload test
      LDW   .D2   *B5++[B6],  B10         ; xi0=xt[0*n2],yi0 = yt[0*n2+1]
      LDW   .D2   *B5++[B6],  A8          ; xi1=xt[2*n2],yi1 = yt[2*n2+1]
||    [!B2] LDW   .D1   *++A1[A4],  B15   ; si1 = w[2 * j], co1 = w[2*j+1]
      LDW   .D2   *B5++[B6],  B11         ; xi2=xt[4*n2],yi2 = yt[4*n2+1]
||    [!B2] LDW   .D1   *++A1[A4],  A3    ; si2 = w[4*j], co2 = w[4*j+1]
      LDW   .D2   *B5++[B6],  A9          ; xi3=xt[6*n2],yi3 = yt[6*n2+1]
      NOP         2
      [!B2] LDW   .D1   *++A1[A4],  A13   ; si3 = w[6*j], co3 = w[6*j+1]
||    [!B2] ADD   .L1X  A4,   B8,   A4    ; j += ie
      SUB2  .S2   B10,  B11,  B3          ; r2a=xi0 – xi2,s2a = yi0 – yi2
||    ADD   .L2   B0,   0,    B1          ; * i = loop counter
||    MV    .L1   A6,   A1                ; reset w
      SUB2  .S1   A8,   A9,   A10         ; t3=xi1 – xi3,  t1 = yi1 – yi3
||    AND   .S2   B1,   B7,   B1          ; * j loop twiddle reload test
      ADD2  .S1   A8,   A9,   A8          ; t0=xi1 + xi3,  t2 = yi1 + yi3
||    ADD2  .S2   B10,  B11,  B1          ; r1a=xi0 + xi2,s1a = yi0 + yi2
||    MPY   .M2   B1,   1,    B2          ; * j loop twiddle reload test
||    [!B1] ADDAW .D2   B5,   1,    B5    ; * reset x input, (circular)
||    [!B2] SUBAW .D1   A5,   1,    A5    ;
||    ADD   .L2   B0,   1,    B0          ;
||    ZERO  .L1   A2                      ; First pass cond. init to zero.
LOOP:
      SHR   .S1X  B3,   16,   A9          ; * extract s2a
||    SHR   .S2X  A10,  16,   B10         ; * extract t1
||    [!B2] ADDAW .D1   A5,   1,    A5    ; * reset x output, (circular)
||    LDW   .D2   *B5++[B6],  B10         ; ** xi0=xt[0*n2], yi0=yt[0*n2+1]
||    MV    .L1   A6,   A1                ; ** reset w
    ADD     .L2   B3,   B10,  B11         ; * r1c = r2a + t1
||  SUB     .L1   A9,   A10,  A12         ; * s1c = s2a – t3
||  SUB2    .S2X  B1,   A8,   B1          ; * r1b=r1a – t0, s1b = s1a – t2
||  ADD2    .S1X  B1,   A8,   A8          ; * xo0=r1a + t0, yo0 = s1a + t2
||  LDW     .D2   *B5++[B6],  A8          ; ** xi1=xt[2*n2], yi1=yt[2*n2+1]
||  [!B2]   LDW   .D1   *++A1[A4],  B15   ; ** si1 = w[2*j], co1 = w[2*j+1]
    [A2]    ADD   .S2X  A11,  2,    B3    ; copy B–side x store pointer
||  [A2]    SHR   .S1   A14,  15,   A14   ; xo2 = xa2 >> 15
||  SUB     .L2   B3,   B10,  B12         ; * r2c = r2a – t1
||  ADD     .L1   A9,   A10,  A9          ; * s2c = s2a + t3
||  MPY     .M1X  A12,  B15,  A10         ; * ss1 = s1c * si1
||  MPYLH   .M2   B11,  B15,  B10         ; * rc1 = r1c * co1
||  LDW     .D2   *B5++[B6],  B11         ; ** xi2=xt[4*n2], yi2=yt[4*n2+1]
||  [!B2]   LDW   .D1   *++A1[A4],  A3    ; ** si2 = w[4*j], co2 = w[4*j+1]
    [A2]    SHR   .S2   B13,  15,   B13   ; yo1 = ya1 >> 15
||  [A2]    SHR   .S1   A15,  15,   A15   ; xo3 = xa3 >> 15
||  MPYLH   .M1X  B1,   A3,   A10         ; * rc2 = r1b * co2
||  MPYLH   .M2X  A12,  B15,  B11         ; * sc1 = s1c * co1
||  LDW     .D2   *B5++[B6],  A9          ; ** xi3=xt[6*n2], yi3=yt[6*n2+1]
||  ADDAW   .D1   A5,   A7,   A5          ;
    [A2]    SHR   .S2   B14,  15,   B14   ; yo2 = ya2 >> 15
||  [B0]    B     .S1   LOOP              ; for i
||  MPY     .M1   A9,   A13,  A12         ; * ss3 = s2c * si3
||  ADD     .L1X  B10,  A10,  A8          ; * xa1 = rc1 + ss1
||  MPY     .M2   B11,  B15,  B13         ; * rs1 = r1c * si1
||  [B0]    STW   .D1   A8,   *++A5[A7]   ; * xt[0*n2]=xo0,  yt[0*n2+1]=yo0
```

```
    [A2]    STH   .D2   B13,  *B3++[B9]   ; yt[2 * n2 + 1] = yo1
||  [A2]    SHR   .S2   B4,   15,   B4    ; yo3 = ya3 >> 15
||  [A2]    STH   .D1   A0,   *A11++[A7]  ; xt[2 * n2] = xo1
||  SHR     .S1   A8,   15,   A0          ; * xo1 = xa1 >> 15
||  MPYH    .M2X  B1,   A3,   B14         ; * sc2 = s1b * co2
||  MPYHL   .M1X  B1,   A3,   A9          ; * ss2 = s1b * si2
    [A2]    STH   .D2   B14,  *B3++[B9]   ; yt[4 * n2 + 1] = yo2
||  MPYLH   .M1   A9,   A13,  A1          ; * sc3 = s2c * co3
||  MPY     .M2X  B1,   A3,   B12         ; * rs2 = r1b * si2
||  SUB     .L2   B11,  B13,  B13         ; * ya1 = sc1 – rs1
||  [!B2]   LDW   .D1   *++A1[A4],  A13   ; ** si3 = w[6*j], co3 = w[6*j+1]
||  [!B2]   ADD   .L1X  A4,   B8,   A4    ; ** j += ie
||  SUB     .S2   B0,   1,    B0          ; *** generate loop counter
    [A2]    STH   .D2   B4,   *B3         ; yt[6 * n2 + 1] = yo3
||  [A2]    STH   .D1   A14,  *A11++[A7]  ; xt[4 * n2] = xo2
||  MPY     .M2X  B12,  A13,  B12         ; * rs3 = r2c * si3
||  MPYLH   .M1X  B12,  A13,  A11         ; * rc3 = r2c * co3
||  ADD     .L1   A10,  A9,   A14         ; * xa2 = rc2 + ss2
||  SUB2    .S2   B10,  B11,  B3          ; ** r2a = xi0–xi2, s2a = yi0–yi2
||  SUB     .L2   B0,   1,    B1          ; *** i = loop counter – 1
    [A2]    STH   .D1   A15,  *A11        ; xt[6 * n2] = xo3
||  SUB     .L2   B14,  B12,  B14         ; * ya2 = sc2 – rs2
||  SUB2    .S1   A8,   A9,   A10         ; ** t3=xi1 – xi3,  t1 = yi1–yi3
||  AND     .S2   B1,   B7,   B1          ; *** j loop twiddle reload test
||  [!A2]   ADD   .L1   A2,   1,    A2    ; First Pass Done Set Cond. Reg
    ADDAH   .D1   A5,   A7,   A11         ; * copy A-side x store pointer
||  SUB     .L2X  A1,   B12,  B4          ; * ya3 = sc3 – rs3
||  ADD     .L1   A11,  A12,  A15         ; * xa3 = rc3 + ss3
||  ADD2    .S1   A8,   A9,   A8          ; ** t0=xi1 + xi3,  t2 = yi1+yi3
||  ADD2    .S2   B10,  B11,  B1          ; ** r1a = xi0+xi2, s1a = yi0+yi2
||  MPY     .M2   B1,   1,    B2          ; *** j loop twiddle reload test
||  [!B1]   ADDAW .D2   B5,   1,    B5    ; *** reset x input, (circular)
; LOOP ends here
    SHL     .S2   B7,   2,    B7          ; mask   <<= 2
||  MPY     .M2   B6,   B8,   B0          ; n/4 = n2 * ie
    SHR     .S1   A7,   2,    A7          ; 2 * n2 >>= 2
||  SHR     .S2   B9,   2,    B9          ; 2 * n2 >>= 2
||  ADD     .L2   B7,   3,    B7          ; mask    += 3
    SHR     .S2   B6,   2,    B6          ; n2     >>= 2
||  SUB     .L2   B0,   1,    B0          ; loop counter = n/4 – 1
    CMPGT   .L2   B7,   B0,   B1          ; kcond    = mask > n / 4 – 1
    [!B1]   B     .S1   K_LOOP            ; if (!kcond) do loop
||  SHL     .S2   B8,   2,    B8          ; ie     <<= 2
    MVC     .S2   IRP,  B4                ; Reload x.
    NOP     4
; K_LOOP ends here
B_END:
*** END Benchmark Timing ***
        MVK     .S1     dummy,  A0            ; New dummy pointer in A0 and B0.
||      MVK     .S2     dummy,  B0            ; New dummy pointer in A0 and B0.
        MVKH    .S1     dummy,  A0            ; New dummy pointer in A0 and B0.
||      MVKH    .S2     dummy,  B0            ; New dummy pointer in A0 and B0.
        LDW     .D2     *B0,    B3            ; Pop return address off dummy.
```

```
||  ZERO    .L2   B2
        LDW     .D1     *+A0[1],        A10     ; Pop A10 off dummy.
||      LDW     .D2     *+B0[2],        B10     ; Pop B10 off dummy.
||  MVC     .S2   B2,   AMR                    ; Reset AMR.
        LDW     .D1     *+A0[3],        A11     ; Pop A11 off dummy.
||      LDW     .D2     *+B0[4],        B11     ; Pop B11 off dummy.
        LDW     .D1     *+A0[5],        A12     ; Pop A12 off dummy.
||      LDW     .D2     *+B0[6],        B12     ; Pop B12 off dummy.
        LDW     .D1     *+A0[7],        A13     ; Pop A13 off dummy.
||      LDW     .D2     *+B0[8],        B13     ; Pop B13 off dummy.
        LDW     .D1     *+A0[9],        A14     ; Pop A14 off dummy.
||      LDW     .D2     *+B0[10],       B14     ; Pop B14 off dummy.
||  B .S2   B3
        LDW     .D1     *+A0[11],       A15     ; Pop A15 off dummy.
||      LDW     .D2     *+B0[12],       B15     ; Pop B15 off dummy.
    NOP           4                            ; Wait 4 cycles for the last pop
                                                ; to occur before returning.
```

Example E–4. digit.asm File

```
;*************************************************************************
; FILE
;   digit.asm – C62xx assembly source for a C-callable FFT digit reversal
;   function.
;
;*************************************************************************
; DESCRIPTION
;
;   This functions implements, by table lookup, digit/bit reversal for FFT
;   algorithms. The function assumes that index tables which contain the
;   indexes of data pairs that get swapped are pre-computed, and stored as
;   two separate arrays. Since this is a table lookup method, this is a
;   generic routine. It can be used for bit-reversal of radix-2 FFTs, or
;   digit-reversal of radix-4 FFTs etc.
;
;*************************************************************************
; POTOTYPE
;     void digit_reverse(int *yx, unsigned short *IIndex,
;     unsigned short *JIndex, int count)
;
;
;*************************************************************************
; IMPLEMENTATION
;
;   The following C code is functional equivalent to this assembly version.
;
;
;   void digit_reverse(int *yx, unsigned short *JIndex,
;   unsigned short *IIndex, int count)
;   {
;
;     int i;
;     unsigned short I, J;
;     int YXI, YXJ;
;
```

```
;     for (i = 0; i<count; i++)
;     {
;     I = IIndex[i];
;     J = JIndex[i];
;     YXI = yx[I];
;     YXJ = yx[J];
;     yx[J] = YXI;
;     yx[I] = YXJ;
;     }
;
;   }
;
;*************************************************************************
      .global     _digit_reverse

AXPtr       .set  a4    ; arg1 passed by calling function
                        ; Pointer to FFT data. This is a static
                        ; pointer. Data to be reversed is accessed
                        ; using indexes. Also this an A register,
                        ; thus it is used in the .d1 unit.
JIndexPtr   .set  b4    ; arg2 passed by calling function
                        ; pointer to digit reversal index
IIndexPtr   .set  a6    ; arg3 passed by calling function
                        ; pointer to other digit reversal index
count       .set  b6    ; arg4 passed by calling function
                        ; Number of points to reverse
J     .set  a0          ; index loaded using JIndex pointer
I     .set  b0          ; Index loaded using IIndex pointer
TJ    .set  a7          ; Temporary copy of J. This is needed
                        ; because the next value of J is loaded
                        ; before the current one is finished being used.
                        ; It is used to store the data value
                        ; loaded by the I index.
TI    .set  b7          ; Temporary copy of I. This is needed
                        ; because the next value of I is loaded
                        ; before the current one is finished being used.
                        ; It is used to store the data value
                        ; loaded by the J index.
XI    .set  a5          ; Data value loaded using the I index.
XJ    .set  b5          ; Data value loaded using the J index.
BXPtr       .set  b2    ; Pointer to FFT data, points to the same
                        ; memory location as AXPtr. It is a B register,
                        ; so it can be used in the .d2 unit.
CNT   .set  b1          ; Count register, used for looping
      .text
_digit_reverse:
    ldh     .d1   *IIndexPtr++[1], I      ; Load an I index.
||  ldh     .d2   *JIndexPtr++[1], J      ; Load a J index.
||  mv.l2x  AXPtr, BXPtr                  ; Copy AXPtr to BXPtr.
    nop           2                       ; Fill the delay slots.
    ldh     .d1   *IIndexPtr++[1], I      ; Load the next I index.
||  ldh     .d2   *JIndexPtr++[1], J      ; Load the next J index.
```

```
||  sub     .l2   count,1,CNT          ; Decrement the count by
                                       ; one, and put into a register
                                       ; that can be used as a
                                       ; condition register.
    nop           1                    ; Fill a delay slot.
    ldw     .d1   *+AXPtr[J],XJ        ; Load the value pointed by
                                       ; the first J index loaded.
||  ldw     .d2   *+BXPtr[I],XI        ; Load the value pointed by
                                       ; the first I index loaded.
||  b       LOOP                       ; Branch for the first time
                                       ; through the loop.
    nop                                ; Fill a delay slot.
    mv.l1   J,TJ                       ; Make a copy of J so that
                                       ; the value is not lost due
                                       ; to the reloading of J.
||  mv      .l2   I,TI                 ; Make a copy of I so that
                                       ; the value is not lost due
                                       ; to the reloading of I.

LOOP:
    ldw     .d1   *+AXPtr[J],XJ        ; load the value pointed by J
||  ldw     .d2   *+BXPtr[I],XI        ; load the value pointed by I
||[CNT]     b     .s1   LOOP           ; conditional branch, branch
                                       ; if CNT != 0
;||[!CNT]b  .s2   B3                   ; having the return
                                       ; here may be a bug,
                                       ; we can try it when
                                       ; we get everything else
                                       ; working

    ldh     .d1   *IIndexPtr++[1], I   ; Load the next I index.
||  ldh     .d2   *JIndexPtr++[1], J   ; Load the next J index.
||[CNT]     sub   .l2   CNT,1,CNT      ; Decrement the loop counter.


    stw     .d1   XI,*+AXPtr[TJ]       ; Data loaded from the I index
                                       ; is stored at the location
                                       ; pointed by the J index.
||  stw     .d2   XJ,*+BXPtr[TI]       ; Data loaded from the J index
                                       ; is stored at the location
                                       ; pointed by the I index.
                                       ; Note that TJ and TI have the I
                                       ; and J values, 3 iterations back.
||  mv.l1   J,TJ                       ; Make a copy of J so that
                                       ; the value is not lost due
                                       ; to the reloading of J.
||  mv      .l2   I,TI                 ; Make a copy of I so that
                                       ; the value is not lost due
                                       ; to the reloading of I.
;loop end

    b .s2   B3                         ; Function return
    nop           5
```

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.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265