SPECTRUM ANALYZER: PROCESSOR EXERCISE USING C LANGUAGE WITH C
INTRODUCTION (55x)*

Thomas Shen

Based on Spectrum Analyzer: Processor Exercise Using C Language with C Introduction† by
Matt Kleffner

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License ‡

Abstract
This is a processor exercise in which students implement a spectrum analyzer using mixed C and
assembly code. Students are to acquire a block of 1024 samples, apply a Hamming window, compute
a length-1024 Discrete Fourier Transform using provided Fast Fourier Transform code, and display the
magnitude-squared spectrum on an oscilloscope. Students will also compile and explore a reference im-
plementation of an autocorrelation-based power spectral density (PSD) estimator. This implementation
estimates the PSD of an IIR-filtered pseudo-noise generator.

1 Implementation
As this is your first experience with the C environment, you will have the option to add most of the required
code in C or assembly. A C skeleton will provide access to input samples, output samples, and interrupt
handling code. You will add code to transfer the inputs and outputs (in blocks at a time), apply a hamming
window, compute the magnitude-squared spectrum, and include a trigger pulse. After the hamming window
is created, either an assembly or C module that bit-reverses the input and performs an FFT calculation is
called.
As your spectrum analyzer works on a block of samples at a time, you will need to use interrupts to
pause your processing while samples are transferred from/to the CODEC (A/D and D/A) buffer. For-
fortunately, the interrupt handling routines have been written for you in a C shell program available at
\v:\ece420\55x\1ab4\main.c. For this lab, you will be working with the code available at \v:\ece420\55x\1ab4.

†http://cnx.org/content/m13044/1.2/
‡http://creativecommons.org/licenses/by/2.0/
1.1 Interrupt Basics

Interrupts are an essential part of the operation of any microprocessor. They are particularly important in embedded applications where DSPs are often used. Hardware interrupts provide a way for interacting with external devices while the processor executes code. For example, in a key entry system, a key press would generate a hardware interrupt. The system code would then jump to a specified location in program memory where a routine could process the key input. Interrupts provide an alternative to polling. Instead of checking for key presses at a predetermined rate (requires a clock), the system could be busy executing other code. On the TI-C55x DSP, interrupts provide a convenient way to transfer blocks of data to/from the CODEC in a timely fashion.

1.2 Interrupt Handling

The `main.c`, `dma.c`, and `lab4.c` code are intended to make your interaction with the hardware much simpler. These files handle the buffering of data using interrupts from the CODEC (A/D and D/A) and the USB port. Here, we will describe the important aspects of the code necessary to complete the assignment.

In the first few labs, data was processed on a sample-by-sample basis, so no buffering was necessary. However, the spectrum analyzer to be implemented in this lab works over a block of \( N = 1024 \) samples. If it were possible to compute a 1024-point FFT in the sample time of one sample, then no additional interrupt handling routines would be necessary. Samples could be collected in a 1024-length buffer and a 1024-point FFT could be computed uninterrupted while the buffer fills. Unfortunately, the DSP is not fast enough to accomplish this task.

We now provide an explanation of the shell C program `main.c`. The `main.c` file contains the main function that sets up the McBSP (Multi Channel Buffered Serial Port), DMA (Direct Memory Access), and interrupts. Then it returns and allows the DSP/BIOS scheduler to take over.

Many of the important interrupt routines are defined in the DSP/BIOS scheduler. The settings can be viewed by expanding the DSP/BIOS Config folder and double-clicking on the `.cdb` file. Changes do not have to be made, but it is good to know that the most important parts are the Scheduling and the Chip Support Library. Under Scheduling, there is the Hardware Interrupt Manager and Software Interrupt Manager. The Chip Support Library includes the DMA controller and the MCBSP. Various settings for these and many other modules can be set using a graphical interface instead of straight up coding.

Code for the DMA is defined in `dma.c`. The buffers are defined in this file, as well as the hardware interrupts. The hardware interrupts are initialized by calling the function `init_DMA` in the main function. When the hardware interrupts are triggered, they call the `HWI_DMA0_Transmit()` and `HWI_DMA1_Receive()` functions. At the end of these two functions, the software interrupt `SWI_Process()` is posted with different variables. Posting `SWI_Process()` will call the `SWI_ProcessBuffer()` function which will require modification.

The `SWI_ProcessBuffer()` function is defined in `lab4.c`. It is called every time the software interrupt `SWI_Process` is posted, which is set to happen every \( N = 1024 \) samples. As given, the function will simply copy the inputs to the outputs. (After commenting out some lines and uncommenting others.) Follow the example and comments to modify the code to perform the necessary operations.

Although each of these operations may be performed in C or assembly, we suggest you follow the guidelines suggested.

1. Transfer inputs (C)
2. Apply a Hamming Window (C/assembly)
3. Bit-reverse the input (C and assembly) (Already done)
4. Apply an \( N \)-point FFT (C and assembly) (Already done)
5. Compute the magnitude-squared spectrum and place in output (C/assembly)
6. Include a trigger pulse (C/assembly)

Note: Bit-reversing and application of the FFT may be done in reverse order depending on implementation.

http://cnx.org/content/m13809/1.12/
Near the beginning of the SWI_ProcessBuffer function, the input samples need to be copied to specific buffers for processing. In C, pointers may be used as array names so that pdest[0] is the first word pointed to by pdest. The input samples are not in consecutive order and must be accessed with offsets. The four channels of input are offset from psrc respectively by $4i$ and $4i + 2$, $i = 0, \ldots, \text{BlockLen} - 1$. The four output channels are accessed consecutively as offsets from pdest. On channel 1 of the output, the input is echoed out. You are to fill channel 2 with the windowed magnitude-squared FFT values by performing the operations listed above. For the first step, take a look at the way we make the DSPLIB cfft call to find out where to transfer the inputs to. (You may change the function call cfft to pass in different values if you like. Just remember that bit_rev() expects its input in a specific location.) Likewise, take a look at the C FFT code (declared in lab4fft.c) to find out where to copy the inputs to.

1.3 Assembly FFT Routine

As the list of operations indicates, bit-reversal and FFT computation are to be done in both C and assembly. For the assembly version, make sure that the line defining C_FFT is commented out in lab4.c. We are providing you with a shell assembly file, available at v:\ece420\55x\lab4\c_fft_given.asm and shown in Appendix B (Section 4: Appendix B), containing many useful declarations and some code. The code for performing bit-reversal and other declarations needed for the FFT routine are also provided in this section. However, we would like you to enter this code manually, as you will be expected to understand its operation.

The assembly file c_fft_given.asm contains two main parts, the data section starting with .sect ".data" and the program section starting with .sect ".text". Every function and variable accessed in C must be preceded by a single underscore _ in assembly and a .global _name must be placed in the assembly file for linking. In this example, bit_rev_fft is an assembly function called from the C program with a label _bit_rev_fft in the text portion of the assembly file and a .global _bit_rev_fft declaration. In each assembly function, the macro ENTER_ASM is called upon entering and LEAVE_ASM is called upon exiting. These macros are defined in v:\ece420\55x\macro.asm. The ENTER_ASM macro saves the status registers and AR1, AR6, and AR7 when entering a function as required by the register use conventions. The ENTER_ASM macro also sets the status registers to the assembly conventions we have been using (i.e, FRCT = 1 for fractional arithmetic and CPL = 0 for DP referenced addressing). The LEAVE_ASM macro just restores the saved registers.

1.3.1 Parameter Passing

The parameter passing convention between assembly and C places the parameters into registers depending on the size of the parameters. Data pointers (16 or 23 bit) will be placed in (X)AR0 through (X)AR4 in that order. 16-bit data will be placed in T0, T1, and AR0 through AR4 in that order. 32-bit data will be placed in accumulators AC0 through AC2. If there are no available registers of the correct type, the parameters will be passed onto the stack. For more details, see page 6-16 of the Optimizing C/C++ Compiler User’s Guide (spru281e1).

1.3.2 Registers Modified

When entering and leaving an assembly function, the ENTER_ASM and LEAVE_ASM macros ensure that certain registers are saved and restored. Since the C program may use any and all registers, the state of a register cannot be expected to remain the same between calls to assembly function(s). Therefore, any information that needs to be preserved across calls to an assembly function must be saved to memory!

Now, we explain how to use the FFT routine provided by TI for the C55x. TI provides a library of commonly used functions. These functions include FFT, FIR, IIR, and some math operations. More information can be found in the DSP Library Programmer’s Reference2. The CFFT function will be used

\footnotesize{\begin{itemize}
\end{itemize}}

http://cnx.org/content/m13809/1.12/
for the DSPLIB implementation. Refer to the reference guide to figure out the correct syntax in calling the function.

The length of the FFT is defined as a label K_FFT_SIZE and the bit-reversing algorithm assumes that the input starts at data memory location _fft_data. To have your code assemble for an N-point FFT, you will have to include the following label definitions in your assembly code.

```
N .set 1024
K_FFT_SIZE .set N ; size of FFT
```

The FFT provided by TI requires that the input be in normal order, with alternating real and imaginary components. The output will be in bit-reversed order, so bit-reversing needs to be done after the FFT. Bit-reversed addressing is a convenient way to order input/output \( x[n] \) into a FFT so that the output/input \( X(k) \) is in sequential order (i.e., \( X(0), X(1), \ldots, X(N-1) \) for an \( N \)-point FFT). The following table illustrates the bit-reversed order for an eight-point sequence.

<table>
<thead>
<tr>
<th>Input Order</th>
<th>Binary Representation</th>
<th>Bit-Reversed Representation</th>
<th>Output Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 1

The following routine performs the bit-reversed reordering of the FFT output data. The routine assumes that the output is stored in data memory starting at the location labeled _bit_rev_data, which must be aligned to the least power of two greater than the input buffer length, and consists of alternating real and imaginary parts. Because our input data is going to be purely real in this lab, you will have to make sure that you set the imaginary parts to zero by zeroing out every other memory location.

```
_bit_rev

ENTER_ASM

MOV #_fft_data, AR3 ; AR3 -> original input
MOV #_bit_rev_data, AR7 ; AR7 -> data processing buffer
MOV AR7, AR2 ; AR2 -> bit-reversed data
MOV #K_FFT_SIZE-1, BRC0
MOV #K_FFT_SIZE, T0 ; T0 = 1/2 size of circ buffer
RPTB bit_rev_end-1
```

http://cnx.org/content/m13809/1.12/
MOV dbl(*AR3), AC0
MOV AC0, dbl(*AR2+)
AMAR *(AR3+T0B)
bit_rev_end:

LEAVE_ASM
RET

As mentioned, in the above code _fft_data is a label indicating the start of the input data and _bit_rev_data is a label indicating the start of a circular buffer where the bit-reversed data will be written.

In general, to have a pointer index memory in bit-reversed order, the T0 register needs to be set to one-half the length of the circular buffer; a statement such as ARx+T0B is used to move the ARx pointer to the next location. For more information regarding the bit-reversed addressing mode, refer to Chapter 6 in the DSP Programmer’s Reference Guide[?]. Is it possible to bit-reverse a buffer in place?

1.4 C FFT Routine

A bit-reversing and FFT routine have also been provided in lab4fft.c, listed in Appendix C (Section 5: Appendix C:). **Again, make sure you understand how the bit reversal is taking place.** In main.c, the line defining C_FFT must not be commented for use of the C FFT routine. The sine tables (twiddle factors) are located in sinetables.h. This fft requires its inputs in two buffers, the real buffer real and the imaginary buffer imag, and the output is placed in the same buffers. The length of the FFT, N, and log(N) are defined in lab4.h, which is also listed in Appendix C (Section 5: Appendix C:). When experimenting with the C FFT make sure you modify these length values instead of the ones in the assembly code and main.c!

1.5 Creating the Window

As mentioned, you will be using the FFT to compute the spectrum of a windowed input. For your implementation you will need to create a 1024-point Hamming window. First, create a Hamming window in matlab using the function hamming. Use the matlab function write_inntvector_headerfile with name set to ’window’ and elempertline set to 8 to create the header file that is included in lab4.c. The window has already been created for you.

1.6 Displaying the Spectrum

Once the DFT has been computed, you must calculate the squared magnitude of the spectrum for display.

\[ |X(k)|^2 = (\Re(X(k)))^2 + (\Im(X(k)))^2 \]  

You may find the assembly instructions sqrm and sqam useful in implementing (1).

Because the squared magnitude is always nonnegative, you can replace one of the magnitude values with a -1.0 as a trigger pulse for display on the oscilloscope. This is easily performed by replacing the DC term (k = 0) with a -1.0 when copying the magnitude values to the output buffer. The trigger pulse is necessary for the oscilloscope to lock to a specific point in the spectrum and keep the spectrum fixed on the scope.

3http://cnx.org/content/m13809/latest/sinetables.h
4http://cnx.org/content/m13809/latest/write_inntvector_headerfile.m

http://cnx.org/content/m13809/1.12/
1.7 Intrinsics

If you are planning on writing some of the code in C, then you may be forced to use intrinsics. Intrinsic instructions provide a way to use assembly instructions directly in C. An example of an intrinsic instruction is `bit_rev_data[0] = _smpyr(bit_rev_data[0], window[0])` which performs the assembly signed multiply round instruction. You may also find the `_lsmpy` instruction useful. For more information on intrinsics, see page 3-29 of the TI-C55x DSP Programmer’s Guide?

The following lines of code were borrowed from the C FFT to serve as an example of arithmetic operations in C. Save this code in a file called mathex.c and create a new project by going to Project->New... In the following window, enter mathex as the name for the project and save it in its own folder on the W: drive. Verify that the Project Type is Executable (.out) and that the target is TMS320C55XX before clicking Finish.

```c
int s1, s2;
int t1, t2;
int i1, i2;
int n1 = 16383, n2 = 16382, n3 = 16381, n4 = 16380;

void main(void)
{
    /* Code for standard 32-bit hardware, */
    /* with x,y limited to 16 bits */
    s1 = (n1*n2 + n3*n4) >> 15;
    s2 = (n1 + n2) >> 1;

    /* Code for TI TMS320C55X series */
    t1 = ((long int)(n1*n2) + (long int)(n3*n4)) >> 15;
    t2 = ((long int)n1 + (long int)n2) >> 1;

    /* Intrinsic code for TMS320C55X series */
    i1 = _sadd(_smpy(n1,n2), _smpy(n3,n4));
    i2 = _ssh1(_sadd(n1, n2),-1);

    while(1);
}
```

Add the mathex.c file to the project by left-clicking on the mathex.pjt file in the left-hand window and selecting Add Files to Project... By default, the generated assembly code is not saved. To save the generated assembly for later comparison, go to Project->Build Options. Under the Compiler tab, click on the Assembly category and make sure Keep Generated .asm Files is selected.

Compile your project before looking at the resulting assembly file and investigating the differences between each block. Be sure to reference page 3-32 of the DSP Programmer’s Guide to find out what the state of the FRCT and OVM bits are. Run this program on the DSP, halt the program, and compare the output values in a memory window. Does each block work properly for all possible values?

1.8 Compiling and Linking

A working program can be produced by compiling the C code and linking assembly modules and the core module. The compiler translates C code to a relocatable assembly form. The linker assigns physical addresses on the DSP to the relocatable data and code segments, resolves `.global` references and links runtime libraries.
Close the mathex project and go back to the original Lab 4 project. In the future if there are additional source code files to include in the project, just follow the above instructions. Once you have completed lab4.c and c_fft_given.asm, select Project->Rebuild All. Load the output file onto the DSP as usual and confirm that valid FFTs are calculated. Once valid output is obtained, measure how many clock cycles it takes to compute both the assembly and C FFT.

2 Quiz Information

From your prelab experiments, you should be able to describe the effect of windowing and zero-padding on FFT spectral analysis. In your DSP system, experiment with different inputs, changing $N$ and the type of window. Can you explain what happens as the input frequency is increased beyond the Nyquist rate? Does the $(|X(k)|^2)$ coincide with what you expect from Matlab? What is the relationship between the observed spectrum and the DTFT? What would happen if the FFT calculation takes longer than it takes to fill inputs with $N$ samples? How long does it take to compute each FFT? What are the tradeoffs between writing code in C versus assembly?

3 Appendix A:

lab4.c\(^5\)

```c
#include "dsk5510_dual3006cfg.h"
#include "dsk5510.h"
#include "swi_process.h"
#include "dsplib.h"

#define N 1024
#define logN 10
#include "window.h"

/* comment the next line to use DSPLIB fft */
//#define C_FFT

#ifdef C_FFT /* Use C FFT */
/* function defined in lab4fft.c */
void fft(void);

/* FFT data buffers */
int real[N]; /* Real part of data */
int imag[N]; /* Imaginary part of data */
#include "lab4fft.c"
#else /* Use DSPLIB FFT */
/* Function defined by c_fft_given.asm */
void bit_rev(void);

/* FFT data buffers (declared in c_fft_given.asm) */
extern int bit_rev_data[N*2]; /* Data output for bit-reverse function */
extern int fft_data[N*2]; /* In-place FFT & Output array */
```

\(^5\)http://cnx.org/content/m13809/latest/lab4.c

http://cnx.org/content/m13809/1.12/
#ifdef /* C_FFT */

// all data processing should be done in SWI_ProcessBuffer

void SWI_ProcessBuffer()
{
    static unsigned int mbox_value = 0;
    short *psrc, *pdest;
    unsigned int i;

    mbox_value |= SWI_getmbox();

    // buffers are only processed when both transmit and receive are ready
    if((mbox_value & DMA_RECEIVE_DONE) && (mbox_value & DMA_TRANSMIT_DONE)) {
        mbox_value = 0;
    }

    // get buffer pointers
    psrc = receive_buffer[receive_buffer_to_process_index];
    pdest = transmit_buffer[transmit_buffer_to_fill_index];

    // samples are interleaved in input buffer 3-4-1-2
    // output buffer is organized 3-4-1-2-3-4-1-2
    // The following code would copy input from each input channel to the
    // respective output channel:
    /*
    for (i = 0; i < 1024; i++)
    {
        pdest[4*i] = psrc[4*i]; //channel 3 output is channel 3 input
        pdest[4*i+1] = psrc[4*i+1]; //channel 4 output is channel 4 input
        pdest[4*i+2] = psrc[4*i+2]; //channel 1 output is channel 1 input
        pdest[4*i+3] = psrc[4*i+3]; //channel 2 output is channel 2 input
    }
    */

#else /* Use DSPLIB FFT */

    // Insert code to fill assembly FFT buffers*
#endif /* C_FFT */


#else /* Use C FFT */

    /* Insert code to fill */
    /* C FFT buffers */

#endif /* C_FFT */

/* Multiply the input signal by the Hamming window. */

http://cnx.org/content/m13809/1.12/
/* Bit-reverse and compute FFT in C */
fft();
/* Now, take absolute value squared of FFT */
/* Last, set the DC coefficient to -1 for a trigger pulse */
/* done, wait for next time around! */
#endif /* Use DSPLIB FFT */

/* Multiply the input signal by the Hamming window. */
/* Compute FFT using DSPLIB function */
cfft((DATA *)fft_data,N, SCALE);
/* Bit reverse using assembly function */
bit_rev();
/* Now, take absolute value squared of FFT */
/* Last, set the DC coefficient to -1 for a trigger pulse */
/* done, wait for next time around! */
#endif /* C_FFT */

receive_buffer_processed = 1; // flag receive buffer as processed
transmit_buffer_filled = 1; // flag output buffer as full

4 Appendix B:
c_fft_given.asm

.http://cnx.org/content/m13809/latest/c_fft_given.asm

http://cnx.org/content/m13809/1.12/
.CPL_ on ; enable assembler for CPL=1
.mmregs ; enable mem mapped register names

.global _bit_rev_data
.global _fft_data
.global _window
.global _bit_rev

.copy "macro.asm"

.sect ".data"

N .set 1024
K_FFT_SIZE .set 1024

.align 4*N
_bit_rev_data .space 16*2*N ; Output of bit reversing function

.align 4*N
_fft_data .space 16*2*N ; FFT output buffer

.sect ".text"

_bit_rev

ENTER_ASM

MOV #_fft_data, AR3
MOV #_bit_rev_data, AR7
MOV AR7, AR2
MOV #K_FFT_SIZE-1, BRC0
MOV #K_FFT_SIZE, T0
RPTB bit_rev_end-1
MOV dbl(*AR3), AC0
MOV AC0, dbl(*AR2+);
AMAR *(AR3+T0B)
bit_rev_end:

LEAVE_ASM

RET

; If you need any more assembly subroutines, make sure you name them
; _name, and include a ".global _name" directive at the top. Also,
; don't forget to use ENTER_ASM at the beginning, and LEAVE_ASM
; and RET at the end!
5 Appendix C:
lab4fft.c

```c
#include "sinetables.h"

void fft(void)
{
    int i,j,k,n1,n2,n3;
    int c,s,a,t,Wr,Wi;

    #include "sinetables.h"
```
j = 0;           /* bit-reverse */
n2 = N \gg 1;
for (i=1; i < N - 1; i++)
{
    n1 = n2;
    while ( j >= n1 )
    {
        j = j - n1;
        n1 = n1 \gg 1;
    }
    j = j + n1;
    if (i < j)
    {
        t = real[i];
        real[i] = real[j];
        real[j] = t;
        t = imag[i];
        imag[i] = imag[j];
        imag[j] = t;
    }
}

/* FFT */
n2 = 1; n3 = Nt;
for (i=0; i < \log N; i++)
{
    n1 = n2; /* n1 = 2**i */
    n2 = n2 + n2; /* n2 = 2**(i+1) */
    n3 = n3 \gg 1; /* cos/sin arg of -6.283185307179586/n2 */
    a = 0;
    for (j=0; j < n1; j++)
    {
        c = costable[a];
        s = sintable[a];
        a = a + n3;
        for (k=j; k < N; k=k+n2)
        {
            /* Code for standard 32-bit hardware, */
            /* with real, imag limited to 16 bits */
            /*
            Wr = (c*real[k+n1] - s*imag[k+n1]) \gg 15;
            Wi = (s*real[k+n1] + c*imag[k+n1]) \gg 15;
            real[k+n1] = (real[k] - Wr) \gg 1;
            imag[k+n1] = (imag[k] - Wi) \gg 1;
            real[k] = (real[k] + Wr) \gg 1;
            imag[k] = (imag[k] + Wi) \gg 1;
            */
        }
    }
}
/* End standard 32-bit code */

/* Code for TI TMS320C54X series */

Wr = ((long int)(c*real[k+n1]) - (long int)(s*imag[k+n1])) >> 15;
Wi = ((long int)(s*real[k+n1]) + (long int)(c*imag[k+n1])) >> 15;
real[k+n1] = ((long int)real[k] - (long int)Wr) >> 1;
imag[k+n1] = ((long int)imag[k] - (long int)Wi) >> 1;
real[k] = ((long int)real[k] + (long int)Wr) >> 1;
imag[k] = ((long int)imag[k] + (long int)Wi) >> 1;

/* End code for TMS320C54X series */

/* Intrinsic code for TMS320C55X series */

Wr = _ssub(_smpy(c, real[k+n1]), _smpy(s, imag[k+n1]));
Wi = _sadd(_smpy(s, real[k+n1]), _smpy(c, imag[k+n1]));
real[k+n1] = _sshl(_ssub(real[k], Wr),-1);
imag[k+n1] = _sshl(_ssub(imag[k], Wi),-1);
real[k] = _sshl(_sadd(real[k], Wr),-1);
imag[k] = _sshl(_sadd(imag[k], Wi),-1);

/* End intrinsic code for TMS320C55X series */

/* Intrinsic code for TMS320C54X series */

Wr = _ssub(_smpy(c, real[k+n1]), _smpy(s, imag[k+n1]));
Wi = _sadd(_smpy(s, real[k+n1]), _smpy(c, imag[k+n1]));
real[k+n1] = _sshl(_ssub(real[k], Wr),-1);
imag[k+n1] = _sshl(_ssub(imag[k], Wi),-1);
real[k] = _sshl(_sadd(real[k], Wr),-1);
imag[k] = _sshl(_sadd(imag[k], Wi),-1);

/* End intrinsic code for TMS320C54X series */