Chapter 3 Digital Filters

Contents

Slide 3-1  Discrete-Time Convolution
Slide 3-2  Sinusoidal Steady-State Response
Slide 3-3  FIR Filters
Slide 3-4  Type 1 Direct Form Realization
Slide 3-5  Design Program WINDOW.EXE
Slide 3-6  Design Program REMEZ.EXE
Slide 3-8  Using Circular Buffers
Slide 3-9  Circular Buffers Using C
Slide 3-10  Circular Buffers in C (cont.)
Slide 3-11  Circular Buffers of Length $L = 2^K$
             with C
Slide 3-12  Circular Buffers of Length $L = 2^K$
             with C (cont.)
Slide 3-13  Experiment 3.1 C FIR Filter
Slide 3-14  Experiment 3.1 (cont. 1)
Slide 3-15  Experiment 3.1 (cont. 2)
Slide 3-16  Experiment 3.1 (cont. 3)
Slide 3-17  Experiment 3.1 (cont. 4)
Slide 3-18  Experiment 3.1 (cont. 5)
Slide 3-19  Measuring Clock Cycles for a Code
Segment with CCS v5.1

Slide 3-20  Profiling by Using a Timer
Slide 3-21  Profiling by Using a Timer (cont.)

Slide 3-22  Circular Buffers Using C6713
             Hardware
Slide 3-23  Hardware Circular Buffers (cont. 1)
Slide 3-24  Hardware Circular Buffers (cont. 2)
Slide 3-25  Indirect Addressing with Registers
Slide 3-26  Writing in C vs. Assembly
Slide 3-27  Calling Assembly Functions from C
Slide 3-28  Calling Assembly Functions from C
Slide 3-29  How a Function Makes a Call
Slide 3-30  How a Called Function Responds
Slide 3-31  Using Assembly Functions with C
Slide 3-32  Linear Assembly and the Optimizer
Slide 3-33  Linear Assembly and the Optimizer
             (cont. 1)
Slide 3-34  Linear Assembly and the Optimizer
             (cont. 2)
Slide 3-35  Invoking the Assembly Optimizer
Slide 3-36  A Simple C-Callable Linear Assembly
             Convolution Function
Slide 3-37  convol1.sa (cont. 1)
Slide 3-38  convol1.sa (cont. 2)
Slide 3-39  convol1.sa (cont. 3)
IIR Filter Design Example (cont.)
Sample Output Listing for Example
A Cascade of 2nd Order Type 1 Direct Form Sections
Measuring the Phase Response
Phase Differences by Lissajous Figures
Lissajous Figures (cont.)
Phase Differences by Time Delay
Measuring Just the IIR Filter Phase Shift
Measuring Just the IIR Filter Phase Shift (cont.)

Experiment 3.3 IIR Filters
Experiment 3.3 (cont.)
Experiment 3.3 (cont.)
Experiment 3.3 (cont.)
Measuring Phase with the Scope
CHAPTER 3
DIGITAL FILTERS

Discrete-Time Convolution

The output $y[n]$ of an LTI system with impulse response $h[n]$ is related to its input $x[n]$ by

$$y[n] = \sum_{k=-\infty}^{\infty} x[k] h[n-k] = \sum_{k=-\infty}^{\infty} h[k] x[n-k]$$

The $z$-Transform of a Convolution

$$Y(z) = \sum_{n=-\infty}^{\infty} y[n] z^{-n} = X(z)H(z)$$
Sinusoidal Steady-State Response

Input

\[ x[n] = Ce^{j\omega nT} \]

Output

\[ y[n] = \sum_{k=-\infty}^{\infty} h[k]Ce^{j\omega(n-k)T} \]
\[ = Ce^{j\omega nT} \sum_{k=-\infty}^{\infty} h[k]e^{-j\omega kT} \]
\[ = x[n]H(z)|_{z=e^{j\omega T}} \]

Frequency Response

\[ H^*(\omega) = H(z)|_{z=e^{j\omega T}} = A(\omega)e^{j\theta(\omega)} \]

Amplitude Response

\[ A(\omega) = |H^*(\omega)| \quad \text{or} \quad \alpha(\omega) = 20 \log_{10} |H^*(\omega)| \quad \text{dB} \]

Phase Response

\[ \theta(\omega) = \arg H^*(\omega) \]

Notice that they have period \( \omega_s = 2\pi/T \).
The output can be expressed as

\[ y[n] = CA(\omega)e^{j[\omega n T + \theta(\omega)]} \]

When the input is the real sinusoid

\[ x[n] = C \cos(\omega n T + \phi) = \Re\{Ce^{j\phi}e^{j\omega n T}\} \]

the output is

\[
\begin{align*}
    y[n] &= \Re\{H^*(\omega)Ce^{j\phi}e^{j\omega n T}\} \\
    &= CA(\omega)\cos[\omega n T + \theta(\omega) + \phi]
\end{align*}
\]

**Finite Duration Impulse Response (FIR) Filters**

Output of an \( N \)-Tap FIR Filter

\[
y[n] = \sum_{k=0}^{N-1} h[k]x[n - k] = \sum_{k=n-N+1}^{n} x[k]h[n - k]
\]
Type 1 Direct Form Realization
Design Program
C:\DIGFIL\WINDOW.EXE

21-tap bandpass filter, Passband 1000 - 3000 Hz

ENTER NAME OF LISTING FILE: junk.lst
ENTER FILENAME FOR COEFFICIENTS: junk.cof
ENTER SAMPLING FREQUENCY IN HZ: 8000

WINDOW TYPES
1  RECTANGULAR WINDOW
2  TRIANGULAR WINDOW
3  HAMMING WINDOW
   \[ 0.54 + 0.46 \cos(\theta) \]
4  GENERALIZED HAMMING WINDOW
   \[ \alpha + (1-\alpha) \cos(\theta) \]
5  HANNING WINDOW \[ 0.5 + 0.5 \cos(\theta) \]
6  KAISER (I0-SINH) WINDOW
7  CHEBYSHEV WINDOW

FILTER TYPES
1  LOWPASS FILTER
2  HIGHPASS FILTER
3  BANDPASS FILTER
4  BANDSTOP FILTER
5  BANDPASS HILBERT TRANSFORM
6  BANDPASS DIFFERENTIATOR
ENTER FILTER LENGTH, WINDOW TYPE, FILTER TYPE: 21,3,3
SPECIFY LOWER, UPPER CUTOFF IN HZ: 1000,3000
CREATE (FREQUENCY,RESPONSE) FILE (Y OR N)? y
ENTER FILENAME: junk.dat
LINEAR (L) OR DB (D) SCALE ?: d

Design Program C:\DIGFIL\REMEZ.EXE

ENTER LISTING FILENAME: junk.lst
ENTER COEFFICIENT STORAGE FILENAME: junk.cof
LINEAR OR DB AMPLITUDE SCALE FOR PLOTS? (L OR D): d
ENTER SAMPLING FREQUENCY (HZ): 8000
ENTER START AND STOP FREQUENCIES IN HZ FOR
     RESPONSE CALCULATION (FSTART,FSTOP): 0,4000

FILTER TYPES AVAILABLE:
   1 MULTIPLE PASSBAND/STOPBAND FILTER
   2 DIFFERENTIATOR
   3 HILBERT TRANSFORM

ENTER: FILTER LENGTH, TYPE, NO. OF BANDS,
       GRID DENSITY: 21,1,3,32
ENTER THE BAND EDGES (FREQUENCIES IN HERTZ)
0,500,1000,3000,3500,4000
SPECIAL USER DEFINED AMPLITUDE RESPONSE(Y/N)? n
SPECIAL USER DEFINED WEIGHTING FUNCTION(Y/N)? n
ENTER (SEPARATED BY COMMAS):

1. VALUE FOR EACH BAND FOR MULTIPLE PASS/STOP BAND FILTERS
2. SLOPES FOR DIFFERENTIATOR (GAIN = Ki*f -> SLOPE = Ki
   WHERE Ki = SLOPE OF i-TH BAND, f IN HERTZ)
3. MAGNITUDE OF DESIRED VALUE FOR HILBERT TRANSFORM 0,1,0

ENTER WEIGHT FOR EACH BAND. (FOR A DIFFERENTIATOR THE WEIGHT FUNCTION GENERATED BY THE PROGRAM FOR THE i th BAND IS WT(i)/f WHERE WT(i) IS THE ENTERED BAND WEIGHT AND f IS IN HERTZ.) 1,1,1

STARTING REMEZ ITERATIONS
DEVIATION = .159436E-03

CALCULATING IMPULSE RESPONSE

CALCULATING FREQUENCY RESPONSE

CREATE (FREQ,RESPONSE) FILE (Y OR N)? y
ENTER FILENAME: junk.dat
Using Circular Buffers to Implement FIR Filters

\[ y[n] = \sum_{k=0}^{N-1} h[k]x[n - k] \]

\[ = h[0]x[n] + h[1]x[n - 1] + \cdots + h[N - 1]x[n - N + 1] \]

<table>
<thead>
<tr>
<th>array index</th>
<th>filter coefficient array ( h[] )</th>
<th>circular buffer array ( xcirc[] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( h[0] )</td>
<td>( x[n - \text{newest}] )</td>
</tr>
<tr>
<td>1</td>
<td>( h[1] )</td>
<td>( x[n - \text{newest} + 1] )</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>( \text{newest} )</td>
<td></td>
<td>( x[n - 1] )</td>
</tr>
<tr>
<td>( \text{oldest} )</td>
<td></td>
<td>( x[n] )</td>
</tr>
<tr>
<td>( N - 2 )</td>
<td>( h[N - 2] )</td>
<td>( x[n - \text{newest} - 2] )</td>
</tr>
<tr>
<td>( N - 1 )</td>
<td>( h[N - 1] )</td>
<td>( x[n - \text{newest} - 1] )</td>
</tr>
</tbody>
</table>
\[ y[n] = \sum_{k=0}^{N-1} h[k] x_{\text{circ}}[(\text{newest} - k) \mod N] \]

**Circular Buffers Using C**

A sample code segment for an FIR filter using a circular buffer for the input sample array is shown below.

```c
main()
{
    int x_index = 0, newest = 0;
    float y, xcirc[N];
    ...
    ...
    /*--------------------------------------------*/
    /* circularly increment newest */
    ++newest;
    if(newest == N) newest = 0;
    /*-------------------------------------------*/
    /* Put new sample in delay line. */
    xcirc[newest] = newsample;
    /*-------------------------------------------*/
    /* Do convolution sum */
    ...
    Go on to the next slide
}```
Circular Buffers in C (cont.)

\[
y = 0;
\]
\[
x\_index = \text{newest}
\]
for \(k = 0; k < N; k++\)
\[
\{
\]
\[
y += h[k]*xcirc[x\_index];
\]
/*-------------------------------------*/
\[
/* circularly decrement x\_index */
--x\_index;
\]
\[
if(x\_index == -1) x\_index = N-1;
\]
/*-------------------------------------*/
\[
}\}
\]
\]
...}

**Warning:** DSK6713_AIC23_read() and MCBSP_read() each return a 32-bit unsigned int. Convert the returned value to an int before shifting right 16 bits to knock off the right channel and get the left channel with sign extension. Shifting an unsigned int right fills the MSB’s with 0’s so the sign is not extended.

**Note:** C has the mod operator, \%, but its implementation by the compiler is very inefficient because the compiler must account for all general cases. Therefore, you should implement the mod operation as shown in the code sement above.
Circular Buffers of Length $L = 2^K$ with C

The circular buffer can be any convenient length, $L$, greater than or equal to the number of filter taps, $N$. If $L > N$, then $L - N$ extra memory locations are needed. However, memory is very cheap now, so this is of negligible concern.

A convenient choice for $L$ is the smallest power of 2 greater than or equal to $N$. Let $L = 2^K$. Then reducing the “newest” pointer and summation index modulo $L$ can be very efficiently performed in C by anding these integers with $2^K - 1$ which in binary form has:

- 1’s in bits 0 through $K - 1$
- zeros in bits $K$ through 31.
Circular Buffers of Length $L = 2^K$ with C (cont.)

Suppose a signed 32 bit integer, $I$, is represented as

$$I = [i_{31}, i_{30}, \ldots, i_0] = \sum_{n=0}^{30} i_n2^n - i_{31}2^{31}$$

$$= \sum_{n=0}^{K-1} i_n2^n + \left( \sum_{n=K}^{30} i_n2^n - i_{31}2^{31} \right)$$

$$= \sum_{n=0}^{K-1} i_n2^n + 2^K \left( \sum_{n=K}^{30} i_n2^{n-K} - i_{31}2^{31-K} \right)$$

Then

$$I = \sum_{n=0}^{K-1} i_n2^n \mod 2^K$$

Thus reducing $I$ modulo $2^K$ can be accomplished by setting $i_K$ through $i_{31}$ to 0 which is easily done by bitwise anding $I$ with $2^K - 1$. 
Chapter 3, Experiment 1
FIR Filter Using C

Perform the following tasks for an FIR filter using a circular buffer and C:

1. Initialize McBSP0, McBSP1, and the AIC23 codec as before and set the sampling rate to 16000 Hz.

2. Measure the amplitude response of the DSK left channel analog path. We will assume the right channel is the same. Apply a sine wave from the signal generator to the left channel of the line input and loop the samples internally in the DSP back to the line output. Vary the frequency and record the values of the output amplitude divided by the input amplitude. Use enough frequencies to get an accurate plot of the response. In particular, be sure to use enough points in the transition region from the passband to the stopband. Plot the response using your favorite plotting program. You should use the set of frequencies chosen here in the rest of Chapter 3.
Experiment 3.1 (cont. 1)

3. Design a 25-tap bandpass FIR filter for a sampling rate of 16 kHz using WINDOW.EXE, REMEZ.EXE, or MATLAB. The passband should extend from 2,000 Hz to 5,000 Hz. Plot the amplitude response in dB of the filter you designed using the output from the design program.

4. Write a C program to implement the filter using a circular sample buffer of length 25. Convert the input samples to floating point format before putting them into the circular buffer. The left channel is the upper 16 bits. So, arithmetically shift the received word 16 bits right to extend the sign and lop off the lower 16 bits (right DAC channel) and then convert the result to a float. Declare and initialize the coefficient array h[] and sample array xcirc[] in main( ), not as globals. The start of each iteration should be controlled by synchronizing it to the McBSP1 XRDY flag. Each time a sample is transmitted, a new input sample can be read because the transmit and receive frame syncs are identical.
5. First compile your program without optimization. Look at the assembly code generated by the compiler to get some idea of how the C source code is implemented by the ’C6713. You can make CCS generate a file with C statements interlisted with assembly language as follow:

Open the “Show Build Settings” menu and click on “C6000 Compiler.” Expand “Advanced Options.” Select “Assembler Options.” Check the box labeled “Keep the generated assembly language ... .” On the next line down labeled “Source interlist” select “Generate C source interlisted assembly file” in the drop-down menu to the right. Then build your project and you should find the interlisted file in the “Debug” folder with a “.asm” extension.

6. Use the profiling capabilities of Code Composer Studio to measure the number of cycles required to generate one output sample. (Do not include the time spent polling the XRDY flag!) Slides 3-19, 3-20, and 3-21 explain how to measure the clock cycles required to execute a code segment.
Experiment 3.1 (cont. 3)

7. Browse through Chapter 3 Optimizing Your Code in the *TMS320C6000 Optimizing Compiler User’s Guide*, SPRU1871. In the compiler build options, go to “Advanced Optimizations” under “Advanced Options” and set the box at the line “Optimize for speed (--opt_for_speed, -mf)” to 5. Then compile your program using the four optimization levels o0, o1, o2, and o3. Look at the assembly code generated for each optimization level. Measure and record the number of cycles required to generate one output sample for each optimization level.

8. Measure the amplitude response of the filtering system from the line input to line output jack and plot the results on a dB scale after correcting for the DSK analog response. Compare your measured result with the theoretical response.

9. Estimate the maximum number of filter taps that could be used based on results of Item 7, a DSP clock of 225 MHz, and a sampling rate of 16 kHz.

10. Copy `C:\c6713dsk\dsk6713.cmd` to a folder other than your project one. Map .bss, .data, .cinit,
Experiment 3.1 (cont. 4)

.pinit, .sysmem, .cio, and .csldata to the external memory BMEM. Change the heap to 0x2000 and the stack to 0x10000. This will insure that internal memory, heap, and stack do not overflow as you increase the filter length in the next item. Give it a different name with the “.cmd” extension. Warning: Do not “add” this modified linker command file to your project because you will get build errors. Instead, select “Project Build Options,” then “C6000 Linker,” then “File Search Path” and point it to your modified dsk6713.cmd.

Alternatively, you can put the modified linker command file in your project directory but not point to it with the build options linker command file search path. Code Composer will automatically recognize your modified linker command file and use it for linking. You can check that CCS is using the correct linker command file by looking near the bottom of the “General” build options screen.
Experiment 3.1 (cont. 5)

11. Increase the number of filter taps from 25 to find the largest number of taps that can be used without running out of time and report the result. Do this for no optimization and o3 optimization. (Note: You do not have to redesign the filter for each length. Multiplying by 0 takes the same time as multiplying by a non-zero value. So, just set the taps beyond $N - 1 = 24$ to zero. If an array size is larger than the size of the initialization list, C automatically sets the remaining elements to 0.)

12. Modify your C program to use a circular buffer of length $L = 32 = 2^5$. Perform the modulo 32 operations using the method described in slides 3-11 and 3-12. Repeat items 6 through 11.
Measuring Clock Cycles for a Code Segment with CCS v5.5

1. Load your project and start a debug session by clicking on the green bug.

2. Click on the main tool bar item, Run, and then Clock.

3. From the Clock menu, click on Enable and a Profile Clock icon will appear at the bottom of the CCS window.

4. From the Clock menu, click on Setup. Set the Count box to Clock Cycles and the Reset Option to Automatic and click OK. The clock Show option will automatically get checked.

5. Put breakpoints at the start and end of the code segment you want to profile. Run the program to the start breakpoint. Then run to the end breakpoint. The number next to the Profile Clock icon is the number of clocks between the breakpoints.
Profiling by Using a Timer

The following code segment shows how to use CSL library functions to initialize a C6713 timer and use it to count CPU cycles in a code segment. It is a little bit more accurate than using the profile clock.

```c
#include <csl_timer.h>

TIMER_Handle hTimer;
Uint32 start, stop, overhead;
Uint32 t;

/* Open a free timer */
hTimer = TIMER_open(TIMER_DEVANY,0);

/* Set the HLD (bit 7) and GO (bit 6) in CTL register to 0 to halt timer. Set bit 9 in CTL to select CPU CLK/4. Set PRD to maximum to allow large count and clear the count. */

TIMER_configArgs(hTimer, 0x00000200, 0xFFFFFFFF, 0x00000000);

/* Determine cycles required to get timer start stop counts */
```
Profiling by Using a Timer (cont.)

/* Timer_getCount() called twice to avoid L1D miss */
start = TIMER_getCount(hTimer);
start = TIMER_getCount(hTimer);
stop = TIMER_getCount(hTimer);
overhead = stop - start; /*cycles to get "start" and "stop" */

<code not being profiled>

/* begin profile area */
TIMER_start(hTimer);
start = TIMER_getCount(hTimer);
<code lines to be profiled>
/* end profile area */
stop = TIMER_getCount(hTimer);
TIMER_pause(hTimer);

/* Compute cycles for profile area. Multiply by 4 since C6713 timers increment once for 4 CPU clocks. */

t = (stop-start-overhead) * 4;
printf("# cycles in profile area: %d \n", t);
Circular Buffers Using the TMS320C6713 Hardware

The TMS320C6000 family of DSP’s has built-in hardware capability for circular buffers.

- The eight registers, A4–A7 and B4–B7, can be used for linear or circular indirect addressing.

- The Address Mode Register (AMR) contains 2-bit fields shown in the figure on Slide 3-23 for each register that determine the address modes as shown in the table on Slide 3-23.

- Then number of words in the buffer is called the block size. The block size is determined by either the BK0 or BK1 5-bit fields in the AMR. The choice between them is determined by the 2-bit mode fields.
Circular Buffers Using the TMS320C6713 Hardware (cont. 1)

- Let $N_{\text{block}}$ be the value of the BK0 or BK1 field. Then the circular buffer has the size $\text{BUF}_\text{LEN} = 2^{N_{\text{block}}+1}$ bytes. So, the circular buffer size can only be a power of 2 bytes.

**Address Mode Register (AMR) Fields**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AMR Mode Field Encoding</strong></td>
<td>Resvd</td>
<td>BK1</td>
<td>BK0</td>
<td>B7 mode</td>
<td>B6 mode</td>
<td>B5 mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B4 mode</td>
<td>A7 mode</td>
<td>A6 mode</td>
<td>A5 mode</td>
<td>A4 mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mode</th>
<th>Addressing Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Linear Mode</td>
</tr>
<tr>
<td>01</td>
<td>Circular Mode Using BK0 Size</td>
</tr>
<tr>
<td>10</td>
<td>Circular Mode Using BK1 Size</td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Circular Buffers Using the TMS320C6713 Hardware (cont. 2)

- The buffer must be aligned on a byte boundary that is a multiple of the block size BUF_LEN. Therefore, the Nblock+1 lsb’s of the buffer base address must all be 0. This can be done in a C program by using the DATA_ALIGN pragma. Suppose the buffer is an array x[]. The alignment command is:

  #pragma DATA_ALIGN(x, BUF_LEN)

The array x[] must be a global array.

It can also be done by creating a named section in the assembly program and using the linker to align the section properly.

How the Circular Buffer is Implemented
Circular addressing is implemented by inhibiting carries or borrows between bits Nblock and Nblock+1 in the address calculations. Therefore, bits Nblock+1 through 31 do not change as the address is incremented or decremented by an amount less than the buffer size.
Indirect Addressing Through Registers

Hardware circular addressing cannot be performed in C. It must be carried out by assembly instructions. Circular addressing is accomplished by indirect addressing through one of the eight allowed registers using the auto-increment/decrement and indexed modes.

A typical circular buffering instruction is

\[ \text{LDW } *A5--, A8 \]

where the A5 field in the AMR has been set for circular addressing. LDW is the mnemonic for “load a word.” The word is loaded into the destination register A8 from the address pointed to by A5 and the address is decremented by 4 bytes according the mode in the AMR after being used (post decremented).
Writing in C vs. Assembly

Because of the tremendous advances in DSP hardware capabilities and software code generation tools, it is becoming standard practice to implement applications almost entirely in a higher level language like C. Some advantages are:

- Rapid software development using a high level language.
- Can use powerful optimizing compilers.
- Application can be easily ported to different DSP’s.
- Profiling tools can find time intensive code segments which can then be written in optimized assembly code.

Generating efficient assembly code for the ’C6000 family by hand is very difficult because:

- there are the multiple execution units
- there is a multi-level pipeline
- different instructions take different times to execute
# Calling Assembly Functions from C

## “A” Side Register Usage

<table>
<thead>
<tr>
<th>Register</th>
<th>By</th>
<th>Special Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>Parent</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>Parent</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>Parent</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>Parent</td>
<td>Structure register</td>
</tr>
<tr>
<td>A4</td>
<td>Parent</td>
<td>Argument 1 or return value</td>
</tr>
<tr>
<td>A5</td>
<td>Parent</td>
<td>Argument 1 or return value with A4 for doubles and longs</td>
</tr>
<tr>
<td>A6</td>
<td>Parent</td>
<td>Argument 3</td>
</tr>
<tr>
<td>A7</td>
<td>Parent</td>
<td>Argument 3 with A6 for doubles and longs</td>
</tr>
<tr>
<td>A8</td>
<td>Parent</td>
<td>Argument 5</td>
</tr>
<tr>
<td>A9</td>
<td>Parent</td>
<td>Argument 5 with A8 for doubles and longs</td>
</tr>
<tr>
<td>A10</td>
<td>Child</td>
<td>Argument 7</td>
</tr>
<tr>
<td>A11</td>
<td>Child</td>
<td>Argument 7 with A10 for doubles and longs</td>
</tr>
<tr>
<td>A12</td>
<td>Child</td>
<td>Argument 9</td>
</tr>
<tr>
<td>A13</td>
<td>Child</td>
<td>Argument 9 with A12 for doubles and longs</td>
</tr>
<tr>
<td>A14</td>
<td>Child</td>
<td></td>
</tr>
<tr>
<td>A15</td>
<td>Child</td>
<td>Frame pointer (FP)</td>
</tr>
</tbody>
</table>
## Calling Assembly Functions from C

### “B” Side Register Usage

<table>
<thead>
<tr>
<th>Register</th>
<th>By</th>
<th>Special Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>Parent</td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>Parent</td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>Parent</td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>Parent</td>
<td>Return address</td>
</tr>
<tr>
<td>B4</td>
<td>Parent</td>
<td>Argument 2</td>
</tr>
<tr>
<td>B5</td>
<td>Parent</td>
<td>Argument 2 with B4 for doubles and longs</td>
</tr>
<tr>
<td>B6</td>
<td>Parent</td>
<td>Argument 4</td>
</tr>
<tr>
<td>B7</td>
<td>Parent</td>
<td>Argument 4 with B6 for doubles and longs</td>
</tr>
<tr>
<td>B8</td>
<td>Parent</td>
<td>Argument 6</td>
</tr>
<tr>
<td>B9</td>
<td>Parent</td>
<td>Argument 6 with B8 for doubles and longs</td>
</tr>
<tr>
<td>B10</td>
<td>Child</td>
<td>Argument 8</td>
</tr>
<tr>
<td>B11</td>
<td>Child</td>
<td>Argument 8 with B10 for doubles and longs</td>
</tr>
<tr>
<td>B12</td>
<td>Child</td>
<td>Argument 10</td>
</tr>
<tr>
<td>B13</td>
<td>Child</td>
<td>Argument 10 with B12 for doubles and longs</td>
</tr>
<tr>
<td>B14</td>
<td>Child</td>
<td>Data page pointer (DP)</td>
</tr>
<tr>
<td>B15</td>
<td>Child</td>
<td>Stack pointer (SP)</td>
</tr>
</tbody>
</table>
How a Function Makes a Call

1. Passed arguments are placed in registers or on the stack. By convention, argument 1 is the left most argument.
   - The first ten arguments are passed in A and B registers as shown in Slides 3-27 and 3-28
   - Additional arguments are passed on the stack.

2. The calling function (parent) must save A0 through A9 and B0 through B9 if needed after the call, by pushing them on the stack.

3. The caller branches to the function (child).

4. Upon returning, the caller reclaims stack space used for arguments.

See: *TMS320C6000 Optimizing Compiler User’s Guide*, SPRU1871, Sections 8.4 and 8.5 for complete details.
How a Called Function Responds

1. The called function allocates space on the stack for local variables, temporary storage, and arguments to functions this function might call. The frame pointer (FP) is used to access arguments on the stack.

2. If the called function calls another, the return address must be saved on the stack. Otherwise it is left in B3.

3. If the called function modifies A10 through A15 or B10 through B15, it must save them in other registers or on the stack.

4. The called function code is executed.

5. The called function returns an int, float, or pointer in A4. Double or long double are returned in the A5:A4 pair.

6. A10–A15 and B10–B15 are restored if used.

7. The frame and stack pointers are restored.

8. The function returns by branching to the value in B3.
Using Assembly Functions with C

• C variable names are prefixed with an underscore by the compiler when generating assembly code. For example, a C variable named `x` is called `_x` in the assembly code.

• The caller must put the arguments in the proper registers or on the stack for arguments beyond number 10.

• A10–A15 and B10–B15, B3 and, possibly, A3 must be preserved. You can use all other registers freely.

• You must pop everything you pushed on the stack before returning to the caller.

• Any object or function declared in the assembly function that is accessed or called from C must be declared with a `.def` or `.global` directive in the assembly code. This allows the linker to resolve references to it.
Linear Assembly Code and the Assembly Optimizer

Writing efficient assembly code is difficult. The TI code generation tools allow you to write in a language called *linear assembly code* which is very similar to full assembly code. Linear assembly files should be given the extension `.sa`. Linear assembly code does not include information about parallel instructions, instruction latencies, or register usage.

Symbolic names can be used for registers. The *assembly optimizer* operates on linear assembly files. The tasks it performs include:

- finding instructions that can operate in parallel
- handling pipeline latencies
- assigning register usage
- defining which units to use
- optimizing execution time by software pipelining
- creating entry and exit assembly code for functions to be called by C.
Linear Assembly Code and the Assembly Optimizer (cont. 1)

See the following two references for complete details on linear assembly code and how to use the assembly optimizer and interpret its diagnostic reports.

- *TMS320C6000 Programmer’s Guide*, SPRU198F.

An example of a C-callable linear assembly function for performing one convolution iteration using a hardware circular sample buffer is shown in Slides 3-36 through 3-39. A C-callable linear assembly function must

- declare its entry point to be global
- include `.cproc` and `.endproc` directives to mark the assembly code region to be optimized.
Linear Assembly Code and the Assembly Optimizer (cont. 2)

As an example, you will find the following lines in convol1.sa

```assembly
.global _convolve
_convolve .cproc x_addr, h_addr, Nh, Nblock, newest
    .reg sum, prod, x_value, h_value
    
    
    .return sum ; By C convention, put sum in A4
.endproc
```

- The entry point is `_convolve`.
- The names following `.cproc` are the function’s arguments.
- The `.reg` line lists symbolic variable names that the assembly optimizer should assign to registers or the stack, if necessary.
- The `.return` directive causes the assembly optimizer to return `sum` to the caller by putting it in A4.
Invoking the Assembly Optimizer

The linear assembly file can be processed by the assembly optimizer by using the command prompt shell command

```
cl6x -mv6710 -o3 -k convol1.sa
```

- `-mv6710` specifies the floating-point DSP series
- `-o3` specifies optimization level 3. The 3 can be replaced by 0, 1, or 2. The `-o` option can be left out for no optimization.
- `-k` specifies that the `.asm` output file should be kept

You can also use Code Composer Studio to process the file by including it in your project. Set options by clicking on `Project` and then `Options`. Then select the `Compiler` tab and set the desired optimization level. Under `Compiler -> Basic`, set the `Target Version` to 671x (-mv6710).
A Simple Linear Assembly Convolution Function that can be Called from C

;**********************************************************************************************
; File: convol1.sa
; By: S.A. Tretter
;
; Compile using
;
; cl6x -mv6713 -o3 convol1.sa
;
; or by using Code Composer Studio with these options.
;
; This is a C callable assembly function for computing
; one convolution iteration. The circular buffering
; hardware of the C6000 is used. The function
; prototype is:
;
; extern float convolve( float x[ ], float h[ ], int Nh,
;                      int Nblock, int newest );
;
; x[ ]    circular input sample buffer
; h[ ]    FIR filter coefficients
; Nh      number of filter taps
; Nblock  circular buffer size in bytes is
;         $2^{Nbloc+1}$ and in words is $2^{Nbloc-1}$
; newest  index pointing to newest sample in buffer
According to the TI C Compiler conventions, the arguments on entry are found in the following registers:

- &x[0] A4
- &h[0] B4
- Nh A6
- Nblock B6
- newest A8

WARNING: The C calling function must align the circular buffer, x[], on a boundary that is a multiple of the buffer size in bytes, that is, a multiple of BUF_LEN = 2^{Nblock+1} bytes. This can be done by a statement in the C program of the form

```c
#pragma DATA_ALIGN(x, BUF_LEN)
```

Note: x[] must be a global array.

```
.global _convolve
_convolve .cproc x_addr, h_addr, Nh, Nblock, newest
        .reg sum, prod, x_value, h_value

; Compute address of x[newest] and put in x_addr
; Note: The instruction ADDAW shifts the second argument, newest, left 2 bits, i.e., multiplies it by 4, before adding it to the first argument to form the actual byte address of x[newest].

ADDAW x_addr, newest, x_addr ; &x[newest]
```
convol1.sa (cont. 2)

; Set up circular addressing
; Load Nbblock into the BK0 field of the Address Mode Register (AMR)

        SHL Nbblock, 16, Nbblock ; Shift Nbblock to BK0 field

; Note: The assembly optimizer will assign x_addr to some register it likes. You will have to manually look at the assembled and optimized code to see which register it picked and then set up the circular mode using BK0 by writing 01 to the field for that register in AMR. The assembler will give you a warning that changing the AMR can give unpredictable results but you can ignore this.

; Suppose B4 was chosen by the optimizer.

        set Nbblock, 8,8, Nbblock; Set mode circular, BK0, B4
        set Nbblock, 10,10, Nbblock; Use this for B5.
        MVC Nbblock, AMR ; load mode into AMR

; Clear convolution sum registers

        ZERO sum
; Now compute the convolution sum.

loop: .trip 8, 500 ; assume between 8 and 500 taps
    LDW *x_addr--, x_value ; x[newest-k] -> x_value
    LDW *h_addr++, h_value ; h[k] -> h_value
    MPYSP x_value, h_value, prod ; h[k]*x[n-k]
    ADDSP prod, sum, sum ; sum of products

[Nh] SUB Nh, 1, Nh ; Decrement count by 1 tap
[Nh] B loop ; Continue until all taps computed

.return sum ; By C convention, put sum in A4
.endproc
Part of Assembly Optimizer Output for No Optimization

.global _convolve
.sect "text"

;******************************************************************************
;* FUNCTION NAME: _convolve *
;* Regs Modified : A0,A3,A4,B0,B4,B5,B6 *
;* Regs Used : A0,A3,A4,A6,A8,B0,B3,B4,B5,B6 *
;******************************************************************************

_convolve:
  .reg sum, prod, x_value, h_value

  MV .S2X A8,B5 ; [47]
  MV .S2X A4,B4 ; [47]
  MV .S1X B4,A0 ; [47]
  MV .S2X A6,B0 ; [47]

  .line 10
  ADDAW .D2 B4,B5,B4 ; [56] &x[newest]
  .line 17
  SHL .S2 B6,0x10,B6 ; [63] Shift Nblock to BK0 field
  .line 31
  SET .S2 B6,0x8,0x8,B6 ; [77] Set mode circular, BK0, B4
  .line 33
  MVC .S2 B6,AMR ; [79] load mode into AMR
  NOP 1
  .line 38
  ZERO .D1 A4 ; [84]
  .line 42

3-40
Part of Assembly Optimizer Output for No Optimization (cont.)

loop:

.line 43
LDW .D2T2 *B4--,B5 ; |89| x[newest-k] -> x_value
NOP 4

.line 44
LDW .D1T1 *A0++,A3 ; |90| h[k] -> h_value
NOP 4

.line 45
MPYSP .M1X B5,A3,A3 ; |91| h[k]*x[n-k]
NOP 3

.line 46
ADDSP .L1 A3,A4,A4 ; |92| sum of products
NOP 3

[line 48
[B0] ADD .D2 0xffffffff,B0,B0 ; |94| Decrement count by 1 tap

.line 49
[B0] B .S1 loop ; |95| Continue until done
NOP 5
; BRANCH OCCURS ; |95|

;**----------------------------------------------------------------------*

.line 51

.line 52
B .S2 B3 ; |98|
NOP 5
; BRANCH OCCURS ; |98|
.endfunc 98,000000000h,0
Part of Assembly Optimizer Output for -o3 Optimization

```
.asg A15, FP
.asg B14, DP
.asg B15, SP

.global _convolve
.sect 'text'

;**************************************************************************
;* FUNCTION NAME: _convolve *
;*
;* Regs Modified : A0,A1,A2,A3,A4,A5,B0,B4,B5 *
;* Regs Used : A0,A1,A2,A3,A4,A5,A6,A8,B0,B3,B4,B5,B6 *
;**************************************************************************

_convolve:
;**************************************************************************
;* SOFTWARE PIPELINE INFORMATION
;*
;* Loop label : loop
;* Known Minimum Trip Count : 8
;* Known Maximum Trip Count : 500
;* Known Max Trip Count Factor : 1
;* Loop Carried Dependency Bound(^) : 4
;* Unpartitioned Resource Bound : 1
;* Partitioned Resource Bound(*) : 1
;* Resource Partition:
;* .L units 1* 0
;* .S units 0 1*
;* .D units 1* 1*
;* .M units 1* 0
;* .X cross paths 1* 0
;* .T address paths 1* 1*
;* Long read paths 0 0
;* Long write paths 0 0
;* Logical ops (.LS) 0 0 (.L or .S unit)
;* Addition ops (.LSD) 0 1 (.L or .S or .D unit)
;* Bound(.L .S .LS) 1* 1*
;* Bound(.L .S .D .LS .LSD) 1* 1*
```

3-42
Part of Assembly Optimizer Output for -o3 Optimization (cont.1)

;* Searching for software pipeline schedule at ...
;* ii = 4 Schedule found with 4 iterations in parallel
;* done
;*
;* Epilog not entirely removed
;* Collapsed epilog stages : 2
;*
;* Prolog not entirely removed
;* Collapsed prolog stages : 2
;*
;* Minimum required memory pad : 0 bytes
;*
;* For further improvement on this loop, try option -mh8
;*
;* Minimum safe trip count : 1
;*------------------------------------------------------------------------*
L1: ; PIPED LOOP PROLOG
    NOP 1
MV  .S2X A6,B0
MV  .S2X A8,B5
    MV  .S2X A4,B4
    MV  .S1X B4,A4
       .line 10
      ADDAW  .D2 B4,B5,B5 ; [56] &x[ newest]
       .line 17
      SHL  .S2 B6,0x10,B4 ; [63] Shift Nblock to BK0 field
       .line 31
      SET  .S2 B4,0x8,0x8,B4 ; [77] Set mode circular, BK0, B4
       .line 33
      MVC  .S2 B4,AMR ; [79] load mode into AMR
       .line 38
      NOP 1
      ZERO  .D1 A3 ; [84]
       .line 42
Part of Assembly Optimizer Output for -o3 Optimization (cont.2)

MV .D2 B5,B4
|| B .S2 loop ; (P) |95| Continue until done
SUB .L1X B0,1,A1
|| MVK .S1 0x2,A2 ; init prolog collapse predicate
|| LDW .D2T2 *B4--,B5 ; (P) |89| x[newest-k] → x_value
|| LDW .D1T1 *A4++,A5 ; (P) |90| h[k] → h_value

;** --------------------------------------------------------------------------*

loop: ; PIPED LOOP KERNEL

[A2] ADDSP .L1 A0,A3,A3 ; ^ |92| sum of products
|| MPYSP .M1X B5,A5,A0 ; @|91| h[k]*x[n-k]

[B0] ADD .D2 0xffffffff,B0,B0 ; @|94| Decrement count by 1 tap

[A2] SUB .D1 A2,1,A2 ;
|| [B0] B .S2 loop ; @|95| Continue until done

[A1] SUB .S1 A1,1,A1 ;
|| [A1] LDW .D2T2 *B4--,B5 ; @@@|89| x[newest-k] → x_value
|| [A1] LDW .D1T1 *A4++,A5 ; @@@|90| h[k] → h_value

;** --------------------------------------------------------------------------*

L3: ; PIPED LOOP EPILOG

ADDSP .L1 A0,A3,A3 ; (E) @@@ ^ |92| sum of products

.line 52
[line 51]
B .S2 B3 ; |98|
NOP 2
MV .D1 A3,A4 ; |97|
NOP 2
; BRANCH OCCURS ; |98|
.endfunc 98,00000000h,0

3-44
Segment of a C Program for Calling the .asm Convolution Function

Suppose we want to do an N = 25 tap filter. The circular buffer must be 32 words or

\[ \text{BUF}_\text{LEN} = 4 \times 32 = 128 \text{ bytes}. \]

Since \( \text{BUF}_\text{LEN} = 2^{\text{Nblock}+1} \), we need Nblock = 6.

... 
```c
#define N 25 /* number of filter taps*/
#define Nblock 6 /*length field in AMR */
#define BUF_LEN 1<<((Nblock+1) /* circular buffer */
              /* size in bytes */
#define BUF_LEN_WORDS 1<<(Nblock-1) /* BUF_LEN/4 */
/*** NOTE: x[ ] must be a global array ***/
    float x[BUF_LEN_WORDS]; /* circular buffer */
/* Align circ. buf. on multiple of block length */
#define DATA_ALIGN(x, BUF_LEN)
/* Prototype the convolution function. */
extern float convolve(float x[], float h[],
                       int N_taps, int N_block, int newest);
...
main(){
...
    int newest = 0; /* Input pointer for buffer */
    float y = 0; /* filter output sample */
    int iy = 0; /* int output for codec */
    Uint32 ix; /* new input sample */
```
Segment of a C Program for Calling the .asm Convolution Function (cont.)

```c
float h[N] = { Put your filter coefficients here separated by commas };  
/* Configure McBSP's and codec */
...
for(;;){
    /* Send last filter output to codec. */
    while(!DSK6713_AIC23_write(hCodec, iy));
    /* NOTE: DSK6713_AIC23_read() returns unsigned int. */
    /* Convert returned value to an 'int' before */
    /* shifting right to extend sign. */

    /* Get new sample pair from DRR */
    while(!DSK6713_AIC23_read(hCodec, &ix));
    newest++; /* Increment input pointer */
    if(newest==BUF_LEN_WORDS) newest = 0;
    /* Reduce modulo buffer size */
    /* Put new sample in buffer */
    x[newest] = ((int)ix)>>16;
    /* Do convolution */
    y = convolve(x, h, N, Nblock, newest);
    iy = ( (int) y) << 16;
}
```

Chapter 3, Experiment 2
FIR Filter Using C and Assembly

Perform the following tasks for a C program that calls an assembly convolution routine:

1. Complete the C program that calls the assembly function `convolve()` in the file `convol1.sa`. Use the 25-tap filter you designed for Experiment 3.1.

2. Build the complete executable module using level `-o3` optimization for both the C and linear assembly programs.

3. Attach the signal generator to the input jack and observe the output on the oscilloscope. Sweep the input frequency to check that the frequency response is correct. You do not have to do a detailed frequency response measurement.

Note: You may have to click on Debug → Reset CPU to get the program to run properly.
4. Use the profiling capabilities of Code Composer Studio to measure the number of cycles required for one call to the convolution function with and without optimization. Compare the results to those for the Experiment 3.1 implementation totally in C.

5. Get the file `convolve.s` from our web site. It unrolls the convolution sum loop once to compute the contributions from two taps in each iteration of the summation loop. The number of filter taps must be an even number. However, a filter with an odd number of taps can be implemented by adding one dummy tap which is zero. The idea is to improve efficiency by eliminating branching overhead and by allowing the optimizer to schedule use of the execution units more optimally.
Experiment 3.2
FIR Filter Using C and Assembly (cont. 2)

Rebuild your FIR filter implementation using this new assembly function and level -o3 optimization. Compare the execution time for one call this convolution routine with that of the function in `convol1.sa`

The variable, ii, reported by the assembly optimizer indicates the number of cycles required by the convolution loop kernel. With level -o2 or -o3 optimization it reports ii = 4 for `convol1.sa` and `convolve.sa`, and that 4 instructions are executing in parallel. Therefore, the kernel for `convol1.sa` requires 4 cycles per tap while the kernel for `convolve.sa` requires only 2 cycles per tap.

Notice the `convol1.asm` only uses multiplier .M1 while `convolve.asm` use both .M1 and .M2.
Infinite Duration Impulse Response (IIR) Filters

Transfer Function

\[ H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_M z^{-M}} \]

\[ = \frac{B(z)}{A(z)} \]

Type 0 Direct Form Realization

\[ \frac{Y(z)}{X(z)} = H(z) = \frac{B(z)}{A(z)} \]

Cross multiplying gives

\[ Y(z)A(z) = X(z)B(z) \]

\[ Y(z) \left( 1 + \sum_{k=1}^{M} a_k z^{-k} \right) = X(z) \sum_{k=0}^{N} b_k z^{-k} \]
IIR Filters (cont. 1)

\[ Y(z) = \sum_{k=0}^{N} b_k X(z) z^{-k} - \sum_{k=1}^{M} a_k Y(z) z^{-k} \]

Time domain equivalent is the difference equation

\[ y[n] = \sum_{k=0}^{N} b_k x[n - k] - \sum_{k=1}^{M} a_k y[n - k] \]

It is called a direct form because the coefficients in the transfer function appear directly in the difference equation.

It is called a recursive filter because past outputs as well as the present and \( N \) past inputs are used in computing the current output.

The filter requires \( N + M + 1 \) storage elements for \( x(n), \ldots, x(n - N) \) and \( y(n - 1), \ldots, y(n - M) \).
Type 1 Direct Form Realization

\[ V(z) = X(z) \frac{1}{A(z)} \]

\[ Y(z) = \frac{X(z)}{A(z)} B(z) = V(z) B(z) \]

Use the direct form 0 realization to compute:

\[ v[n] = x[n] - \sum_{k=1}^{M} a_k v[n - k] \]

Then, the output can be computed as

\[ y[n] = \sum_{k=0}^{N} b_k v[n - k] \]
Type 1 Direct Form Realization

Let $M = N$ without loss of generality.
Computing the Direct Form 1 Output

Step 1: Compute $v[n]$

$$v[n] = x[n] - \sum_{k=1}^{N} a_k s_k[n]$$

Step 2: Compute the output $y[n]$

$$y[n] = b_0 v[n] + \sum_{k=1}^{N} b_k s_k[n]$$

Step 3: Update the state variables

$$s_N[n + 1] = s_{N-1}[n]$$

$$s_{N-1}[n + 1] = s_{N-2}[n]$$

$$\vdots$$

$$s_2[n + 1] = s_1[n]$$

$$s_1[n + 1] = v[n]$$
Type 2 Direct Form Realization

Let $M = N$. Then

$$\sum_{k=0}^{N} a_k z^{-k} Y(z) = \sum_{k=0}^{N} b_k z^{-k} X(z)$$

with $a_0 = 1$.

Taking everything except $Y(z)$ to right-hand side gives

$$Y(z) = b_0 X(z) + \sum_{k=1}^{N} [b_k X(z) - a_k Y(z)] z^{-k}$$

This is the key equation for the type 2 direct form realization shown in the following figure.
Type 2 Direct Form Realization

\[ x[n] \quad b_0 \quad + \quad s_1[n] \quad y[n] \]

\[ b_1 \quad + \quad -a_1 \quad s_2[n] \]

\[ b_2 \quad + \quad -a_2 \quad s_N[n] \]

\[ b_N \quad + \quad -a_N \]
Computing the Direct Form 2
Output

Step 1: Compute the output $y[n]$

$$y[n] = b_0 x[n] + s_1[n]$$

Step 2: Update the state variables

$$s_1[n + 1] = b_1 x[n] - a_1 y[n] + s_2[n]$$
$$s_2[n + 1] = b_2 x[n] - a_2 y[n] + s_3[n]$$
$$\vdots$$
$$s_{N-1}[n + 1] = b_{N-1} x[n] - a_{N-1} y[n] + s_N[n]$$
$$s_N[n + 1] = b_N x[n] - a_N y[n]$$
A Program for Designing IIR Filters

C:\DIGFIL\IIR\IIR.EXE

Uses the bilinear transformation with a Butterworth, Chebyshev, inverse Chebyshev, or elliptic analog prototype filter.

It can design lowpass, highpass, bandpass, or bandstop filters.

The form of the resulting filter is a cascade (product) of sections, each with a second order numerator and denominator with the leading constant terms normalized to 1, possibly a first order section normalized in the same way, and an overall scale factor. These second order sections are also known as biquads.
IIR Filter Design Example

Design a bandpass filter based on an elliptic analog prototype filter.

The nominal lower stopband extends from 0 to 600 Hz.
The passband extends from 1000 to 2000 Hz.
The upper stopband extends from 3000 to 4000 Hz.

SAVE RESULTS IN A FILE (Y OR N): y
ENTER LISTING FILENAME: junk.lst
ENTER 1 FOR ANALOG, 2 FOR DIGITAL: 2
ENTER SAMPLING RATE IN HZ: 8000
ENTER NUMBER OF FREQS TO DISPLAY: 100
ENTER STARTING FREQUENCY IN HZ: 0
ENTER STOPPING FREQUENCY IN HZ: 4000
ENTER 1 FOR BW, 2 FOR CHEBY, 3 FOR ICHEBY,
    4 FOR ELLIPTIC: 4
IIR Filter Design Example (cont.)

ENTER 1 FOR LOWPASS, 2 FOR HP, 3 FOR BP,
OR 4 FOR BR: 3

ENTER F1,F2,F3,F4 FOR BP OR BR FREQS:
600,1000,2000,3000

ENTER PASSBAND RIPPLE AND STOPBAND ATTENUATION
IN +DB: 0.2,40

ELLiptIC FILTER ORDER = 4

CREATE FREQ, LINEAR GAIN FILE (Y,N)? n
CREATE FREQ, DB GAIN FILE (Y,N)? Y
ENTER FILENAME: junkdb.dat
CREATE FREQ, PHASE FILE (Y,N)? n
CREATE FREQ, DELAY FILE (Y,N)? y
ENTER FILENAME: JUNKDEL.DAT

Note: $F_1 < F_2 < F_3 < F_4$

$F_1 =$ upper edge of lower stopband
$F_2 =$ lower edge of passband
$F_3 =$ upper edge of passband
$F_4 =$ lower edge of upper stopband
Sample Output Listing from IIR.EXE

DIGITAL BANDPASS ELLIPTIC FILTER
FILTER ORDER = 8
Z PLANE

<table>
<thead>
<tr>
<th>ZEROS</th>
<th>POLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>.977149 +- j .212554</td>
<td>.173365 +- j .761580</td>
</tr>
<tr>
<td>.902015 +- j .431705</td>
<td>-.028463 +- j .919833</td>
</tr>
<tr>
<td>-.538154 +- j .842847</td>
<td>.683010 +- j .651915</td>
</tr>
<tr>
<td>-.873779 +- j .486323</td>
<td>.482595 +- j .656484</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RADIUS</th>
<th>FREQUENCY</th>
<th>RADIUS</th>
<th>FREQUENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>.100000E+01</td>
<td>.272712E+03</td>
<td>.781063E+00</td>
<td>.171502E+04</td>
</tr>
<tr>
<td>.100000E+01</td>
<td>.568352E+03</td>
<td>.920273E+00</td>
<td>.203939E+04</td>
</tr>
<tr>
<td>.100000E+01</td>
<td>.272351E+04</td>
<td>.944190E+00</td>
<td>.970348E+03</td>
</tr>
<tr>
<td>.100000E+01</td>
<td>.335335E+04</td>
<td>.814782E+00</td>
<td>.119288E+04</td>
</tr>
</tbody>
</table>

4 CASCADE STAGES, EACH OF THE FORM:

\[
F(z) = \frac{(1 + B1*z^{-1} + B2*z^{-2})}{(1 + A1*z^{-1} + A2*z^{-2})}
\]

<table>
<thead>
<tr>
<th>B1</th>
<th>B2</th>
<th>A1</th>
<th>A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.954298</td>
<td>1.000000</td>
<td>-.346731</td>
<td>.610059</td>
</tr>
<tr>
<td>-1.804029</td>
<td>1.000000</td>
<td>.056927</td>
<td>.846903</td>
</tr>
<tr>
<td>1.076307</td>
<td>1.000000</td>
<td>-1.366019</td>
<td>.891495</td>
</tr>
<tr>
<td>1.747559</td>
<td>1.000000</td>
<td>-.965191</td>
<td>.663870</td>
</tr>
</tbody>
</table>

SCALE FACTOR FOR UNITY GAIN IN PASSBAND: 1.8000479016654E-002

<table>
<thead>
<tr>
<th>FREQUENCY RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREQUENCY</td>
</tr>
<tr>
<td>.0000</td>
</tr>
<tr>
<td>40.0000</td>
</tr>
<tr>
<td>80.0000</td>
</tr>
<tr>
<td>120.0000</td>
</tr>
</tbody>
</table>


3-61
A Cascade of 2nd Order Type 1 Direct Form Sections

Scale Factor for Unity Gain

\[ x[n] \quad c \quad v_1[n] \quad s_{1,0}[n] \quad z^{-1} \quad s_{1,1}[n] \quad -a_{1,1} \quad b_{1,1} \quad \quad s_{1,2}[n] \quad z^{-1} \quad s_1[n] \quad v_1[n] \quad y_1[n] \quad v_2[n] \quad s_{2,0}[n] \quad z^{-1} \quad s_{2,1}[n] \quad -a_{2,1} \quad b_{2,1} \quad s_{2,2}[n] \quad s_2[n] \quad y_2[n] \]

\[ y[n] \quad v_L[n] \quad y_{L-1}[n] \quad \ldots \quad y[n] \quad v_L[n] \quad y_{L-1}[n] \]

\[ s_{L,0}[n] \quad z^{-1} \quad s_{L,1}[n] \quad b_{L,1} \quad -a_{L,1} \quad s_{L,2}[n] \quad z^{-1} \quad b_{L,2} \quad -a_{L,2} \]
Measuring the Phase Response

Suppose the input to a system is

\[ x(t) = A \sin \omega_0 t \]

and the output is

\[ y(t) = B \sin(\omega_0 t + \theta) \]

Phase Differences by Lissajous Figures

If \( x(t) \) is applied to the horizontal input of an oscilloscope and \( y(t) \) is applied to the vertical input, the following ellipse will be observed:

\[
\left( \frac{y}{B} \right)^2 - 2 \left( \frac{x}{A} \right) \left( \frac{y}{B} \right) \cos \theta + \left( \frac{x}{A} \right)^2 = \sin^2 \theta
\]

If \( \theta = 0 \) the ellipse becomes the straight line

\[ y = \frac{B}{A} x \]

When \( \theta = \pi/2 \), the principal axes are aligned with the x and y axes.
Phase Differences by Lissajous Figures (cont.)

The maximum value for $x$ is $x_{\text{max}} = A$. The ellipse crosses the $x$-axis when $y = 0$ or $\omega_0 t + \theta = \pi$. The corresponding value for $x$ is

$$x_0 = A \sin(\pi - \theta) = A \sin \theta$$

Thus

$$\frac{x_0}{x_{\text{max}}} = \sin \theta$$

and so

$$\theta = \sin^{-1} \frac{x_0}{x_{\text{max}}}$$

The Lissajous figures form an interesting display but it is difficult to make accurate measurements of $\theta$ this way.
Measuring Phase Differences by Relative Time Delay

The output can also be expressed as

\[ y(t) = B \sin[\omega_0(t + d)] = B \sin(\omega_0 t + \theta) \]

where

\[ \theta = \omega_0 d = 2\pi \frac{d}{T_0} \] radians

Therefore, the phase difference can be easily found by multiplying the relative time delay by the frequency in radians/sec or by multiplying \(2\pi\) by the ratio of the time delay and the period of the sinewave.

Students have found it much easier and more accurate to use this method for measuring the phase response.
Measuring Just the IIR Filter Phase Shift

The figure above shows the situation when you measure the phase shift from Left Line In to Left Line Out. The input, \( s(t) \), passes through the AIC left input filter with phase shift \( \theta_{IL}(\omega) \), the IIR DSP filter with phase shift \( \theta_{DSP}(\omega) \), and the AIC left output filter with phase shift \( \theta_{OL}(\omega) \). The phase from Left Line In to Left Line Out is

\[
\theta_L(\omega) = \theta_{IL}(\omega) + \theta_{DSP}(\omega) + \theta_{OL}(\omega)
\]
Measuring Just the IIR Filter Phase Shift (cont.)

When the input samples are passed directly to the AIC right output channel, the phase shift from Left Line In to Right Line Out is

\[ \theta_R(\omega) = \theta_{IL}(\omega) + \theta_{OR}(\theta) \]

The phase shift from Left Line Out to Right Line Out is

\[ \theta_{LR}(\omega) = \theta_L(\omega) - \theta_R(\omega) \]
\[ = \theta_{DSP}(\omega) + [\theta_{OL}(\omega) - \theta_{OR}(\omega)] \]

The left and right output filters are nearly identical, so

\[ \theta_{OL}(\omega) - \theta_{OR}(\omega) \simeq 0 \]

and

\[ \theta_{LR}(\omega) \simeq \theta_{DSP}(\omega) \]
Chapter 3, Experiment 3
IIR Filter Experiments

Perform the following tasks for IIR filters:

1. Design an IIR bandpass filter based on an elliptic lowpass analog prototype. Use a 16 kHz sampling rate. The lower stopband should extend from 0 to 800 Hz, the passband from 2000 to 5000 Hz, and the upper stopband from 7000 to 8000 Hz. The passband ripple should be no more than 0.3 dB and the stopband attenuation should be at least 40 dB.

For the filter you designed, plot the amplitude response generated by the filter design program using a dB scale. Plot the phase response also. Explain any discontinuities in the phase response.

2. Implement your filter on the DSK taking the input from the left channel and sending the filtered output to the left channel. Use type 1 direct forms for the filter sections.
Experiment 3.3 (cont. 1)

3. Use the signal generator and oscilloscope to measure the amplitude response and plot it in dB. Compare your theoretical and measured responses.

4. Plot the filter output versus input on the oscilloscope and observe the Lissajous figure ellipse as you vary the input frequency. To generate the Lissajous figure,

- First make sure the volts per division on channel 1 and 2 are the same, that the probes on both channels are set to 1:1, and that both channels are DC coupled.

- Then press the “Horiz” button at the top in the “Horizontal” section. Turn the knob in the middle on the left with the circular arrow icon to show “XY” on the screen. Press the knob in to select XY.
Experiment 3.3 (cont. 2)

Capture the Lissajous figures at several frequencies and include them in your lab report. In theory, you can measure the phase response from the Lissajous figure. In practice, it is hard to measure the voltages accurately on the oscilloscope and to track the rotations of the ellipse when it is changing rapidly with frequency. Do not bother trying to measure the phase response using the Lissajous figures.

5. Measure the phase response from Line In to Line Out by using the phase measurement function of the Agilent oscilloscope and plot the result. The oscilloscope uses the time delay method. Compare this measured phase response with the theoretical response of your filter. Explain why they differ significantly. Slide 3-72 explains how to use the oscilloscope to measure the phase response.
Experiment 3.3 (cont. 3)

6. Send the input samples directly to the Right Line Out as well as passing them through your IIR filter to the Left Line Out as shown in the figure on Slide 3-66. Use the method explained in Slides 3-66 and 3-67 to measure just the phase response of your IIR filter. Plot your measured IIR phase response and compare it with the theoretical one.

7. Use the profiling capability of Code Composer Studio to measure the number of clock cycles and time required to process one sample with the IIR filter and record the result. Do this for no optimization and with -o3 optimization. Compare your results with the 25-tap FIR filter results.
Using the Agilent Oscilloscope to Measure Phase Difference

(a) Press the “Meas” button.

(b) Under the screen display, use the “Type” button to select “Phase.”

(c) Press the “Settings” button.

(d) Set “Source1” and “Source2” to your two oscilloscope channels.

(e) Press the “Meas” button and below the screen press “Add Measurement.” The phase difference should now be displayed in degrees on the right side of the screen.

The scope phase measurements may jitter around. Use averaging to get more stable measurements as follows:

(a) Press the “Acquire” button.

(b) Below the screen press “Acq Mode” and select “Averaging.”

(c) Select “#Avgs.”

(d) Set the mode back to “Normal” when done.