Chapter 3 Digital Filters

Contents

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

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

Frequency Response

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

Amplitude Response

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

Phase Response

$$\theta(\omega) = \arg H^*(\omega)$$

Notice that they have period $\omega_s = 2\pi/T$.

The output can be expressed as

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

When the input is the real sinusoid

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

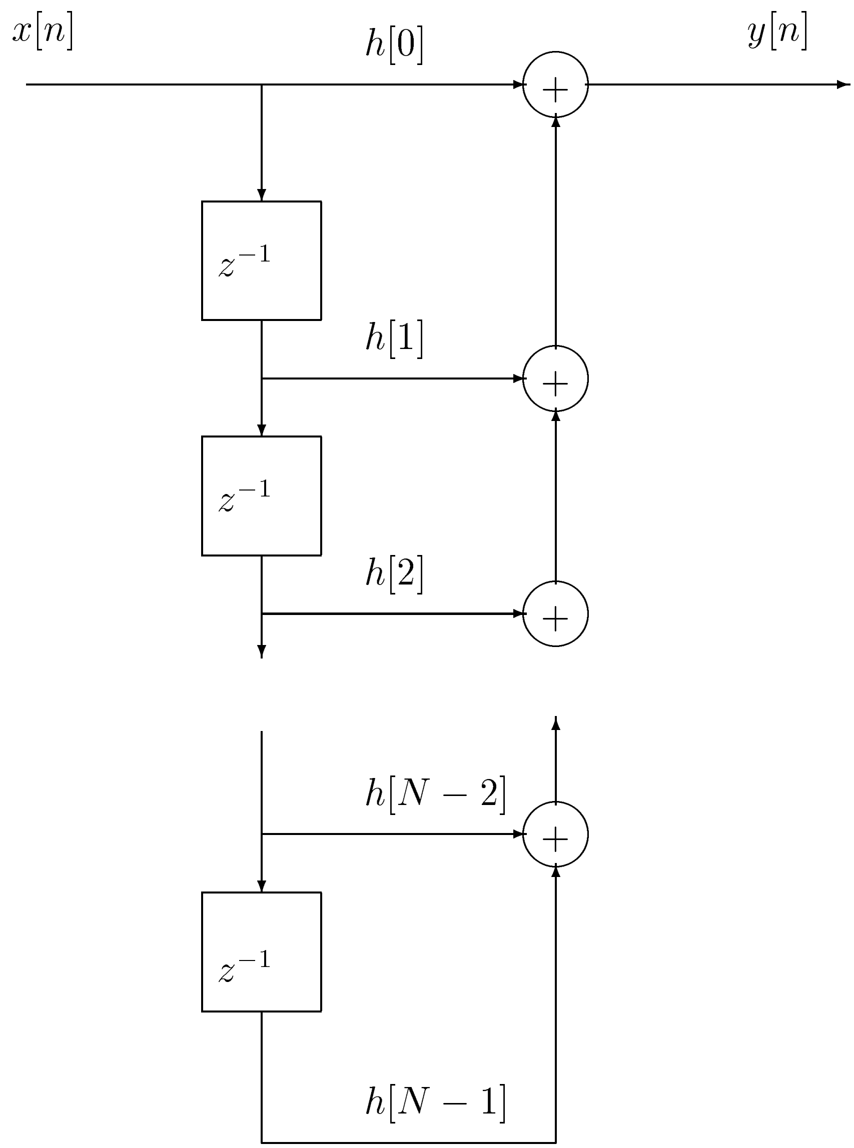
the output is

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

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\REMEZ87.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

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

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

array index	filter coefficient array `h[]`	circular buffer array `xcirc[]`
0	$h[0]$	$x[n-newest]$
1	$h[1]$	$x[n-newest+1]$
$\vdots$	$\vdots$	$\vdots$
		$x[n-1]$
$newest$		$x[n]$
$oldest$		$x[n-N+1]$
		$x[n-N+2]$
$\vdots$	$\vdots$	$\vdots$
$N-2$	$h[N-2]$	$x[n-newest-2]$
$N-1$	$h[N-1]$	$x[n-newest-1]$

$$y[n] = \sum_{k=0}^{N-1} \mathtt{h}[k]\mathtt{xcirc}[(newest - k) \bmod N]$$

Circular Buffers Using C

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

```
main()
{
   int x_index = 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 Buffer in C (cont.)

```
    y = 0;
    x_index = 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.

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, REMEZ87.EXE, or MATLAB. The passband should extend from 2,000 Hz to 5,000 Hz. Plot the theoretical amplitude response in dB.

4. Write a C program to implement the filter using a circular sample buffer. 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.

   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.

Experiment 3.1 (cont. 2)

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. 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!)

6. Browse through Chapter 3 Optimizing Your Code in the *TMS320C6000 Optimizing Compiler User's Guide*, SPRU1871. 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.

7. 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 EVM response. Compare your measured result with the theoretical response.

8. 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.

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 3-15 for each register that determine the address modes as shown in the table on Slide 3-15.
- 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 Nblock be the value of the BK0 or BK1 field. Then the circular buffer has the size BUF_LEN = $2^{\mathrm{Nblock}+1}$ bytes. So, the circular buffer size can only be a power of 2 bytes.

Address Mode Register (AMR) Fields

31–26	25–21	20–16	15–14	13–12	11–10
Resvd	BK1	BK0	B7 mode	B6 mode	B5 mode

9–8	7–6	5–4	3–2	1–0
B4 mode	A7 mode	A6 mode	A5 mode	A4 mode

AMR Mode Field Encoding

Mode	Addressing Option
00	Linear Mode
01	Circular Mode Using BK0 Size
10	Circular Mode Using BK1 Size
11	Reserved

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

```
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

Register	Preserved By	Special Uses
A0	Parent	
A1	Parent	
A2	Parent	
A3	Parent	Structure register
A4	Parent	Argument 1 or return value
A5	Parent	Argument 1 or return value with A4 for doubles and longs
A6	Parent	Argument 3
A7	Parent	Argument 3 with A6 for doubles and longs
A8	Parent	Argument 5
A9	Parent	Argument 5 with A8 for doubles and longs
A10	Child	Argument 7
A11	Child	Agument 7 with A10 for doubles and longs
A12	Child	Argument 9
A13	Child	Argument 9 with A12 for doubles and longs
A14	Child	
A15	Child	Frame pointer (FP)

Calling Assembly Functions from C

"B" Side Register Usage

Register	Preserved By	Special Uses
B0	Parent	
B1	Parent	
B2	Parent	
B3	Parent	Return address
B4	Parent	Argument 2
B5	Parent	Argument 2 with B4 for doubles and longs
B6	Parent	Argument 4
B7	Parent	Argument 4 with B6 for doubles and longs
B8	Parent	Argument 6
B9	Parent	Argument 6 with B8 for doubles and longs
B10	Child	Argument 8
B11	Child	Agument 8 with B10 for doubles and longs
B12	Child	Argument 10
B13	Child	Argument 10 with B12 for doubles and longs
B14	Child	Data page pointer (DP)
B15	Child	Stack pointer (SP)

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 3-19 and 3-20
   - 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 Optimizing Compiler User's Guide*, SPRU187I, Chapter 4.
- *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 3-28 through 3-31. 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`

```
          .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^{Nblock+1} and in words is 2^{Nblock-1}
;       newest  index pointing to newest sample in buffer
```

convol1.sa (cont. 1)

```
;    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
;      #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 Nblock into the BK0 field of the Address Mode
;     Register (AMR)

    SHL Nblock, 16, Nblock ; Shift Nblock 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 unpredicatable
;         results but you can ignore this.
;
;         Suppose B4 was chosen by the optimizer.
;
    set Nblock, 8,8, Nblock; Set mode circular, BK0, B4
;   set Nblock, 10,10, Nblock; Use this for B5.
    MVC Nblock, AMR        ; load mode into AMR
;-------------------------------------------------------

;  Clear convolution sum registers

    ZERO  sum
```

convol1.sa (cont. 3)

```
;  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

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

          .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
```

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:
;*                                A-side   B-side
;*      .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*
;*
```

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        98,000000000h,0
```

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

Suppose we want to do an Nh = 25 tap filter. The circular buffer must be 32 words or $\text{BUF\_LEN} = 4 \times 32 = 128$ bytes. Since $\text{BUF\_LEN} = 2^{\text{Nblock}+1}$, we need Nblock = 6.

```
     ...
 #define Nh 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 */
   #pragma DATA_ALIGN(x, BUF_LEN)
    ...
 main(){
    ...
   int newest = 0; /* Input pointer for buffer */
   float y = 0; /* filter output sample */
   int iy = 0; /* int output for codec */
   int ix;  /* new input sample */
   float h[Nh] = { Put your filter coefficients here
                     separated by commas };
```

Segment of a C Program for Calling the .asm Convolution Function (cont.)

```
/* Prototype the convolution function. */
 extern float convolve(float x[], float h[],
             int N_taps, int N_block, int newest);
/*  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 and make it an int.   */
     while(!DSK6713_AIC23_read(hCodec, &ix));

     ix = ix >> 16;/* Extend sign. Eliminate the   */
                  /*  right channel (16 LSB's).   */
     newest++;  /* Increment input pointer */
     if(newest==BUF_LEN_WORDS) newest = 0;
        /* Reduce modulo buffer size,   */
     x[newest] = ix; /* Put new sample in buffer   */
     y = convolve(x, h, Nh, 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.

Chapter 3, Experiment 2

FIR Filter Using C and Assembly (cont. 1)

4. Use the profiling capabilites 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.sa` 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

$$
\begin{aligned}
H(z) &= \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_M z^{-M}} \\
&= \frac{B(z)}{A(z)}
\end{aligned}
$$

Type 0 Direct Form Realization

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

Cross multiplying gives

$$
\begin{aligned}
Y(z)A(z) &= X(z)B(z) \\
Y(z)\left(1 + \sum_{k=1}^{M} a_k z^{-k}\right) &= X(z)\sum_{k=0}^{N} b_k z^{-k}
\end{aligned}
$$

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*

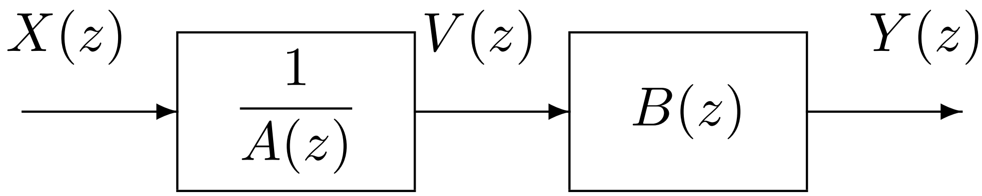
$$y[n] = \sum_{k=0}^{N} b_k x[n-k] - \sum_{k=1}^{M} a_k y[n-k]$$

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



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

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

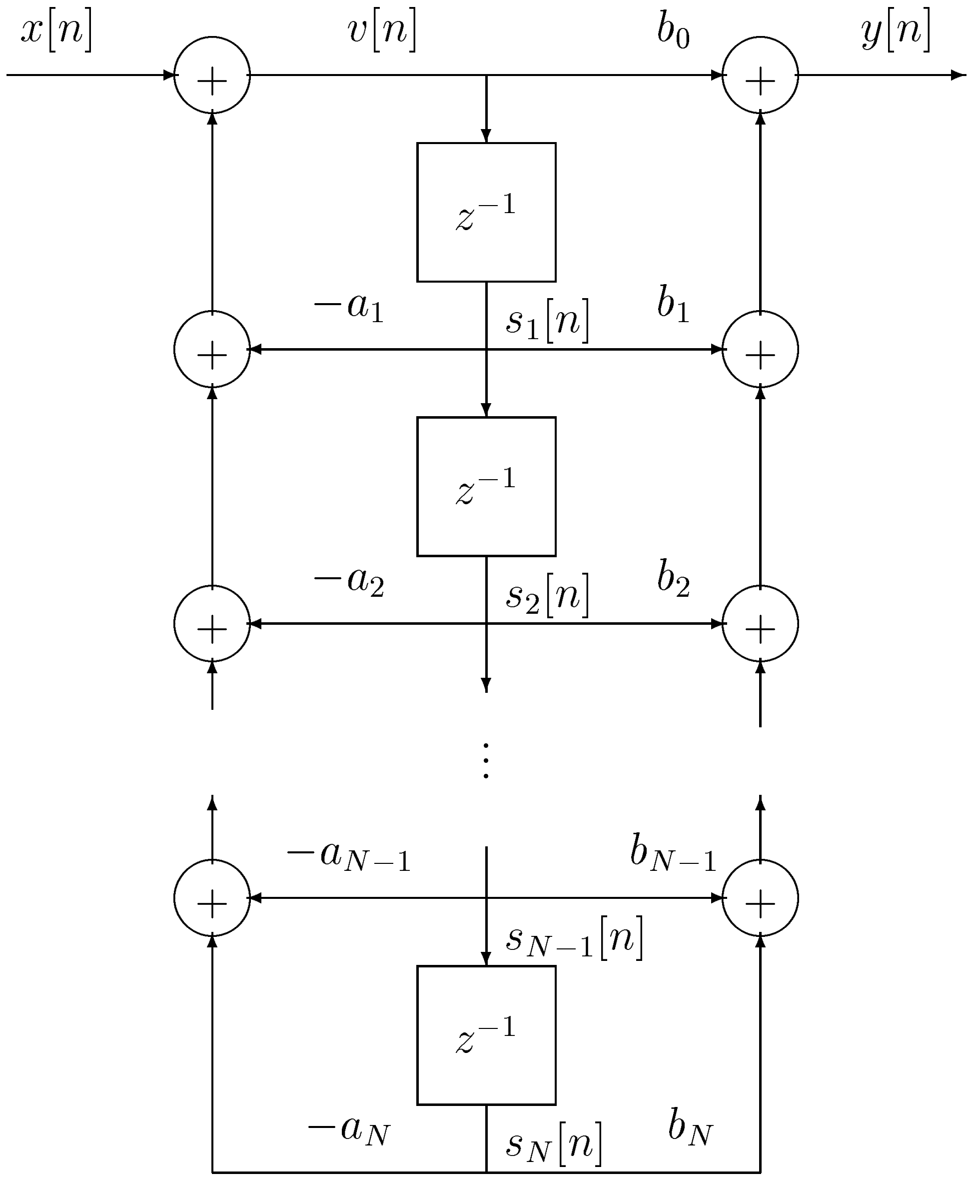
Use the direct form 0 realization to compute:

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

Then, the output can be computed as

$$y[n] = \sum_{k=0}^{N} b_k v[n-k]$$

Type 1 Direct Form Realization



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

$$\begin{aligned}
s_N[n+1] &= s_{N-1}[n] \\
s_{N-1}[n+1] &= s_{N-2}[n] \\
&\vdots \\
s_2[n+1] &= s_1[n] \\
s_1[n+1] &= v[n]
\end{aligned}$$

Type 2 Direct Form Realization

Let $M = N$. Then

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

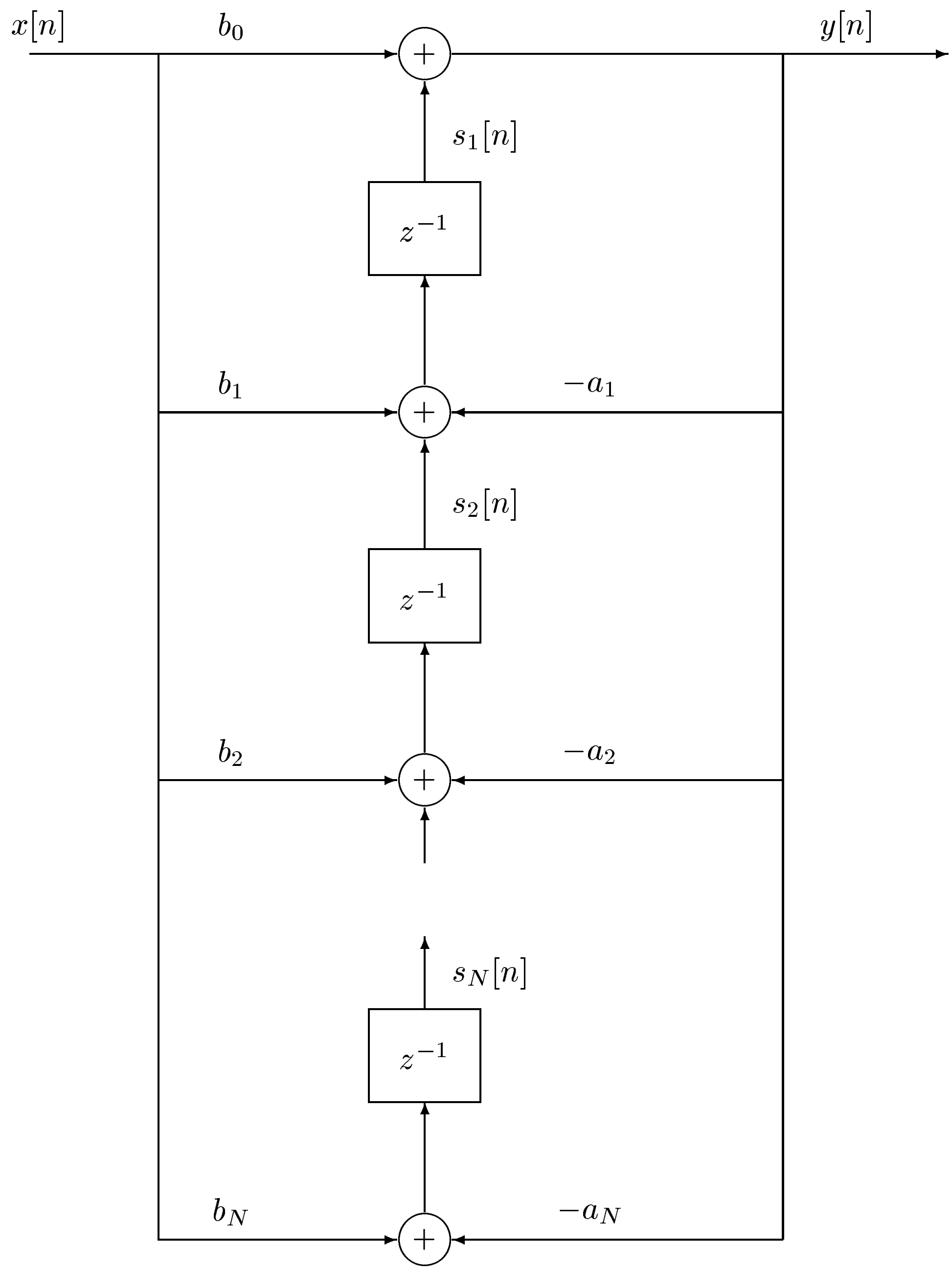
with $a_0 = 1$.

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



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

$$\begin{aligned}
s_1[n+1] &= b_1 x[n] - a_1 y[n] + s_2[n] \\
s_2[n+1] &= b_2 x[n] - a_2 y[n] + s_3[n] \\
&\vdots \\
s_{N-1}[n+1] &= b_{N-1} x[n] - a_{N-1} y[n] + s_N[n] \\
s_N[n+1] &= b_N x[n] - a_N y[n]
\end{aligned}$$

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 know 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: $F1 < F2 < F3 < F4$

$F1 =$ upper edge of lower stopband

$F2 =$ lower edge of passband

$F3 =$ upper edge of passband

$F4 =$ lower edge of upper stopband

Sample Output Listing from IIR.EXE

```
                 DIGITAL BANDPASS ELLIPTIC FILTER
                      FILTER ORDER =  8
                       Z PLANE

      ZEROS                          POLES
    .977149 +- j  .212554      .173365 +- j  .761580
    .902015 +- j  .431705     -.028463 +- j  .919833
   -.538154 +- j  .842847      .683010 +- j  .651915
   -.873779 +- j  .486323      .482595 +- j  .656484

    RADIUS       FREQUENCY         RADIUS       FREQUENCY
  .100000E+01     .272712E+03     .781063E+00     .171502E+04
  .100000E+01     .568352E+03     .920273E+00     .203939E+04
  .100000E+01     .272351E+04     .944190E+00     .970348E+03
  .100000E+01     .335335E+04     .814782E+00     .119288E+04

        4 CASCADE STAGES, EACH OF THE FORM:

F(z) = ( 1 + B1*z**(-1) + B2*z**(-2) ) / ( 1 + A1*z**(-1) + A2*z**(-2) )

    B1           B2              A1           A2
 -1.954298     1.000000         -.346731      .610059
 -1.804029     1.000000          .056927      .846903
  1.076307     1.000000       -1.366019      .891495
  1.747559     1.000000         -.965191      .663870
 SCALE FACTOR FOR UNITY GAIN IN PASSBAND:   1.8000479016654E-002

                FREQUENCY RESPONSE
  FREQUENCY       GAIN         GAIN (dB)      PHASE      DELAY (SEC)
       .0000   2.1048E-03  -5.3536E+01      .00000   .13458E-03
     40.0000   2.0557E-03  -5.3741E+01     -.03385   .13493E-03
     80.0000   1.9093E-03  -5.4382E+01     -.06789   .13600E-03
    120.0000   1.6681E-03  -5.5556E+01     -.10228   .13780E-03
       .
       .
       .
```

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 $x_{max} = A$. 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

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

and so

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

The Lissajous figures form an interesting display but it is difficult to make accurate measurements of θ 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

$$\theta = \omega_0 d = 2\pi \frac{d}{T_0} \quad \text{radians}$$

Therefore, the phase difference can be easily found by multiplying the relative time delay by the frequency in radians/sec or by multiplying 2π 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.

Setting Break and Profile Points in Assembly Programs Called from C

In evaluating filter implementations, you are asked to measure the time required to generate a filter output sample. If you try to set break or profile points in an ASM routine called from C by displaying the ASM source code and setting these points on displayed source lines, the source lines will be highlighted as if the point was set. But when you run the program, Code Composer will tell you that it can not set the break or profile point and it has disabled the point.

Fortunately, there is a solution. First set a break point on the C source line that calls the ASM function. Restart your program and run it to this break point. Then single step into the ASM routine. The Dis-Assembly window should rise to the surface. Then set the desired break or profile points in the Dis-Assembly window. Go to the Profiler menu and enable the clock and display the statistics. Delete the break point on the C line that calls the ASM routine. Finally, restart and run your program and the profiling statistics should be displayed.

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.

   Plot the theortical amplitude response generated by the filter design program on a dB scale. Plot the phase response also. Explain any discontinuities in the phase response.

2. Write a program to implement your filter on the DSK. Use type 1 direct forms for the filter sections. You can use C or assembly.

Experiment 3.3 (cont.)

3. Use the signal generator and oscilloscope to measure the amplitude response and plot it in dB. Also measure the phase response and plot the results. Be sure to adjust the measured responses for the responses of the analog paths in the DSK. Compare your theoretical and measured responses.

4. Use the profiling capability of Code Composer Studio to measure the number of clock cycles and time required to process one sample, and record the result. Do this for the two cases where the program is compiled without optimization and with level -o3 optimization.