# Digital Filters 5

## 5.1 OVERVIEW

The digital computation of filter transfer functions has always been an important area of digital signal processing. Apart from the obvious advantages of virtually eliminating errors in the filter associated with voltage and temperature drift, component aging, and EMI-induced power supply noise, digital filters are capable of performance specifications that would, at best, be extremely difficult to achieve with an analog implementation. Digital filters are able to realize sharp cutoff characteristics, tight passband and stopband specifications, exactly linear phase responses, and even arbitrary magnitude responses.

Many of the routines in this chapter make use of circular buffers for storing data and coefficients. To implement circular addressing, the length register (Ln) that corresponds to the circular buffer pointer register (In) must be set to the buffer length. See the discussion in Chapter 2 of the *ADSP-2100 User's Manual* for more information.

The bibliography at the end of this manual provides several excellent sources of introductions to digital filter theory, design, and implementation.

## 5.2 FINITE IMPULSE RESPONSE (FIR) FILTERS

A finite impulse response (FIR) filter is a discrete linear time-invariant system whose output is based on the weighted summation of a finite number of past inputs. FIR filters, unlike infinite impulse response (IIR) filters, are nonrecursive and require no feedback loops in their computation. This property allows simple analysis and implementation on microprocessors such as the ADSP-2100. A graphic representation of an FIR filter is shown in Figure 5.1, on the next page.

# 5 Digital Filters

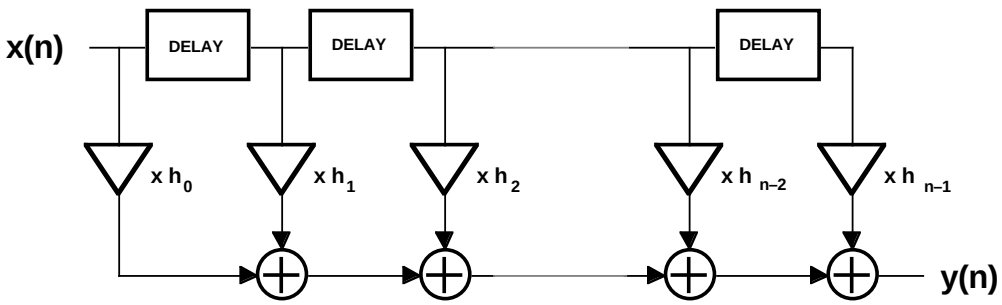


**Figure 5.1 FIR Filter**

### 5.2.1 Single-Precision FIR Transversal Filter

The realization of an FIR filter can take many forms, although the most useful in practice are generally the transversal and lattice structures. The FIR lattice filter is described later in this chapter. Another implementation of the transversal filter is given in Chapter 13. An FIR transversal filter structure can be obtained directly from the equation for discrete-time convolution.

$$y(n) = \sum_{k=0}^{N-1} h_k(n)\, x(n-k)$$

In this equation, x(n) and y(n) represent the input to and output from the filter at time *n*. The output y(n) is formed as a weighted linear combination of the current and past input values of x, x(n–k). The weights, $h_k(n)$, are the transversal filter coefficients at time *n*. (For a nonadaptive filter, the coefficients do not change with *n*. Adaptive filters are described later in this chapter.) In the equation, x(n–k) represents the past value of the input signal "contained" in the (k+1)th tap of the transversal filter. For example, x(n), the present value of the input signal, would correspond to the first tap, while x(n–42) would correspond to the forty-third filter tap.

The subroutine that realizes the sum-of-products operation used in computing the transversal filter is shown in Listing 5.1. The first instruction sets up the computation by clearing MR and loading MX0 and MY0 with the first data and coefficient values from data and program memory. The multiply/accumulate with dual data fetch in the *sop* loop is then executed N–1 times in N cycles to compute the sum of the first N–1 products. The final multiply/accumulate instruction is performed with the rounding mode enabled to round the result to the upper 24 bits of MR. MR1 is then conditionally saturated to its most positive or negative value based on the status of the overflow flag MV. In this manner, results are

accumulated to the full 40-bit resolution of MR, with saturation of the output only if the final result overflowed beyond the least significant 32 bits of MR.

The limit on the number of filter taps attainable for a real-time implementation of the transversal filter subroutine is determined primarily by the processor cycle time, the sampling rate, and the number of other computations required. The transversal filter subroutine presented here requires a total of N+6 cycles for a filter of length N; at an 8 kHz sampling rate and an instruction cycle time of 125 nanoseconds, this permits a filter of 900 taps with 94 instruction cycles for other operations.

```
.MODULE fir_sub;


{
   FIR Transversal Filter Subroutine

   Calling Parameters
        I0 –> Oldest input data value in delay line
        L0 = Filter length (N)
        I4 –> Beginning of filter coefficient table
        L4 = Filter length (N)
        M1,M5 = 1
        CNTR = Filter length - 1 (N-1)

   Return Values
        MR1 = Sum of products (rounded and saturated)
        I0 –> Oldest input data value in delay line
        I4 –> Beginning of filter coefficient table

   Altered Registers
        MX0,MY0,MR

   Computation Time
        N - 1 + 5 + 2 cycles

   All coefficients and data values are assumed to be in 1.15 format.
}

.ENTRY  fir;

fir:    MR=0, MX0=DM(I0,M1), MY0=PM(I4,M5);
        DO sop UNTIL CE;
sop:       MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M5);
        MR=MR+MX0*MY0(RND);
        IF MV SAT MR;
        RTS;
.ENDMOD;
```

**Listing 5.1 FIR Filter Single-Precision**

### 5.2.2 Double-Precision FIR Transversal Filter

Many digital filters require a sum-of-products computation using operands that are greater than 16 bits in magnitude. The following subroutine implements a sum-of-products calculation using coefficients and data that are both represented in double precision. On the ADSP-2100, this is accomplished through the use of the mixed-mode multiply instructions, in much the same manner as described in Chapter 2.

The subroutine that realizes the sum-of-products operation used in computing the transversal filter is shown in Listing 5.2. First, the sum of the products of the low halves of the coefficients and the high halves of the data values is computed; this sum is accumulated with the sum of the products of the high halves of the coefficients and the low halves of the data values. This sum is then shifted right 16 bits and then accumulated with the sum of the products of the high halves of the coefficients and the high halves of the data values. A conditional saturation is then performed on the final 32-bit result before storage to data memory. Note that because the result is only the most significant 32 bits, the products of the low-order coefficients and the low-order data affect only the least significant bit of the result and are therefore not computed.

The above routine is easily extended to applications requiring other multiprecision formats or even those requiring mixed precision. For example, to use 32-bit coefficients and 16-bit data values, you would eliminate the *lhloop* loop and make corresponding changes in the data memory pointer values and the size of the circular buffer. Chapter 2 describes the basic techniques for performing multiprecision multiplications, which are directly applicable to multiprecision multiply/accumulate operations.

```
.MODULE dfir_sub;

{
   Double-Precision Transversal Filter Subroutine

   Calling Parameters
        I0 —> Oldest input data value in delay line
        L0 = 2 × Filter length (N)
        I4 —> 2nd location (LSW of 1st value) of filter coefficient table
        L4 = 2 × Filter length (N)
        M0,M4 = 1
        M1,M5 = 2
        M2,M6 = 3
        AX0 = Filter length - 2 (N-2)
        CNTR = Filter length - 2 (N-2)

   Return Values
        MR1,MR0 = sum of products
        (conditionally saturated to 32 bits)
        I0 —> Oldest input data value in delay line
        I4 —> 2nd location (LSW of 1st value) of filter coefficient table

   Altered Registers
        MX0,MY0,MR

   Computation Time
        3 × (N - 2) + 16 + 9

   All coefficients and data values are assumed to be in 1.15 format.
}

.ENTRY  dfir;

dfir:   MR=0, MX0=DM(I0,M1), MY0=PM(I4,M5);
        DO hlloop UNTIL CE;
hlloop:    MR=MR+MX0*MY0(SU), MX0=DM(I0,M1), MY0=PM(I4,M5);
        MR=MR+MX0*MY0(SU), MX0=DM(I0,M2), MY0=PM(I4,M4);
        MR=MR+MX0*MY0(SU), MX0=DM(I0,M1), MY0=PM(I4,M5);
        CNTR=AX0;
        DO lhloop UNTIL CE;
lhloop:    MR=MR+MX0*MY0(US), MX0=DM(I0,M1), MY0=PM(I4,M5);
        MR=MR+MX0*MY0(US), MX0=DM(I0,M0), MY0=PM(I4,M5);
        MR=MR+MX0*MY0(US), MX0=DM(I0,M1), MY0=PM(I4,M5);
        MR0=MR1;                          {downshift 16 places}
        MR1=MR2;
        CNTR=AX0;
        DO hhloop UNTIL CE;
hhloop:    MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M5);
        MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M6);
        MR=MR+MX0*MY0(SS);
        IF MV SAT MR;
        RTS;
.ENDMOD;
```

**Listing 5.2 Double-Precision FIR Filter**

# 5 Digital Filters

### 5.2.3 Two-Dimensional FIR Filter

The two-dimensional FIR filter is used in a variety of applications, including smoothing and edge detection in image processing, in which the input is a matrix that represents a digitized image. The routine presented in this section is a two-dimensional version of the single-precision FIR filter presented earlier in this chapter. Instead of performing a sum-of-products operation on a one-dimensional input signal, it convolves a two-dimensional (QxR) coefficient matrix with a two-dimensional (SxT) input matrix using the equation:

$$G(x,y) = \sum_{i=0}^{Q-1} \sum_{j=0}^{R-1} [H(i,j)\, F(x-i,\, y-j)] \qquad \text{(Oppenheim, 1978)}$$

The two-dimensional FIR filter is computed by multiplying and accumulating a section of each row of the input matrix by each row of the coefficient matrix. The value of a point in the output matrix is equal to a sum-of-products operation of the input matrix with the coefficient matrix.

The routine, shown in Listing 5.3, assumes that the first (data memory) address of the input matrix is stored in I0, the first output matrix address in I1, and the first coefficient matrix address in I4. The length registers L0 and L1 should each be set to zero, and L4 should be set to the length of (total number of elements in) the QxR coefficent matrix. The number of rows of the output buffer (S–Q) should be stored in the CNTR, and the number of columns of the output buffer (T–R) should be stored in AX0. AX1 should store Q, the number rows in the coefficient matrix. AY0 should store R–2. The modify registers M0 and M4 should both be set to one. M1 should store T–R+1, M2 should be set to –(Q × T+1), and M3 should store R–1. All of these values must be initialized before the routine is called.

Convolution at the edges of the input matrix can yield meaningless results because the edge values do not have valid adjacent data. This situation can be remedied in several ways; one way is to set any value outside the input matrix to zero. Another way, used in this example, is to perform the convolution only if the coefficient matrix is completely enclosed by the input matrix. To use a coefficient matrix that is not enclosed by the input matrix, you must call the routine with CNTR set to S (the number of input matrix rows), AX0 set to T (the number input matrix columns), and M3 set

to zero.

The routine begins by reading the first coefficient into MY0. The *in_row* loop is executed once for each row of the output matrix. In this loop, the CNTR register is loaded and the *in_col* loop is executed, generating a column of the output matrix on each pass. The *row_loop* loop executes the *col_loop* loop once for each column of the coefficient matrix.

The last two multiply/accumulate and data read instructions are removed from the *col_loop* loop in order to provide efficient pointer manipulation. The first multiply/accumulate operation outside the loop is performed in parallel with moving I0 to point to the first element of the next convolution row. The second multiply/accumulate operation is performed in parallel with reading in the values that will be used for the next sequence of execution of the *row_loop* loop. When all iterations of the *row_loop* loop have been executed, the value in MR1 is stored in the appropriate location of the output matrix. The last instruction in the *in_row* loop modifies I0 to point to the first element of the next row of input data.

The output matrix, stored by rows, is smaller than the input matrix if the input matrix fully encloses the coefficient matrix. In this case, the number of cycles the routine requires is:

$$(((R-2+4) \times Q+5) \times (T-R)+3) \times (S-Q)+3+4$$

If the coefficient matrix is not fully enclosed by the input matrix, the number is:

$(((R-2+4) \times Q+5) \times T+3) \times S+3+4$

```
.MODULE two_dimensional_FIR_filter;

{           Q-1 R-1
   G(x,y) =  Σ   Σ  [H(i,j) F(x-i, y-j)]
             i=0 j=0

   Calling Parameters
        I0 —> F, SxT Input Matrix stored by rows                    L0 = 0
        I1 —> G, (S-Q)x(T-R) Output Matrix stored by rows           L1 = 0
        I4 —> H, QxR Coefficient Matrix stored by rows              L4 = Q × R
        M0,M4 = 1
        M1 = T-R+1
        M2 = -(Q × T+1)                         AX1 = Q
        M3 = R-1                                AY0 = R-2
        CNTR = S-Q                              AX0 = T-R

   Return Values
        G(x,y) filled [Output Matrix]

   Altered Registers
        MX0,MY0,MR,I0,I1,I4,L4

   Computation Time
        (((R-2+4) × Q + 5) × (T-R) + 3) × (S-Q) + 3 + 4 cycles
}

.ENTRY  tdfir;

tdfir:  MY0=PM(I4,M4);                          {Get first coefficient}
        DO in_row UNTIL CE;                     {Loop through output rows}
           CNTR=AX0;
           DO in_col UNTIL CE;                  {Loop through output columns}
              CNTR=AX1;
              MR=0, MX0=DM(I0,M0);              {Clear MR, get input data}
              DO row_loop UNTIL CE;             {Loop through coefficient rows}
                 CNTR=AY0;
                 DO col_loop UNTIL CE;          {Loop through coefficient cols}
col_loop:           MR=MR+MX0*MY0 (SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
                 MR=MR+MX0*MY0 (SS), MX0=DM(I0,M1), MY0=PM(I4,M4);
                 {Move pointer to next convolution window row}
row_loop:        MR=MR+MX0*MY0 (SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
                 {Read values for next loop}
              DM(I1,M0)=MR1;                    {Save output points}
in_col:       MODIFY(I0,M2);                    {Get next conv. start same row}
in_row:     MODIFY(I0,M3);                      {Point to next input row}
        RTS;
.ENDMOD;
```

**Listing 5.3 Two-Dimensional FIR Filter**

## 5.3 INFINITE IMPULSE RESPONSE (IIR) FILTERS

Compared to the FIR filter, an IIR filter can often be much more efficient in terms of attaining certain performance characteristics with a given filter order. This is because the IIR filter incorporates feedback and is capable of realizing both poles and zeroes of a system transfer function, whereas the FIR filter is only capable of realizing the zeroes (although the FIR filter is still more desirable in many applications, because of features such as stability and the ability to realize exactly linear phase responses).

### 5.3.1 Direct Form IIR Filter

The IIR filter can realize both the poles and zeroes of a system because it has a rational transfer function, described by polynomials in *z* in both the numerator and the denominator:

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}}$$

The difference equation for such a system is described by the following:

$$y(n) = \sum_{k=0}^{M} b_k x(n-k) + \sum_{k=1}^{N} a_k y(n-k)$$

In most applications, the order of the two polynomials M and N are the same.

The roots of the denominator determine the pole locations of the filter, and the roots of the numerator determine the zero locations. There are, of course, several means of implementing the above transfer function with an IIR filter structure. The "direct form" structure presented in Listing 5.4 implements the difference equation above.

Note that there is a single delay line buffer for the recursive and non-recursive portions of the filter (Oppenheim and Schafer's Direct Form II). The sum-of-products of the *a* values and the delay line values are first computed, followed by the sum-of-products of the *b* values and the delay line values.

```
.MODULE diriir_sub;

{
   Direct Form II IIR Filter Subroutine

   Calling Parameters
        MR1 = Input sample (x[n])
        MR0 = 0
        I0 —> Delay line buffer current location (x[n-1])
        L0 = Filter length
        I5 —> Feedback coefficients (a[1], a[2], ... a[N])
        L5 = Filter length - 1
        I6 —> Feedforward coefficients (b[0], b[1], ... b[N])
        L6 = Filter length
        M0 = 0
        M1,M4 = 1
        CNTR = Filter length - 2
        AX0 = Filter length - 1

   Return Values
        MR1 = output sample (y[n])
        I0 —> delay line current location (x[n-1])
        I5 —> feedback coefficients
        I6 —> feedforward coefficients

   Altered Registers
        MX0,MY0,MR

   Computation Time
        (N - 2) + (N - 1)) + 10 + 4 cycles  (N = M = Filter order)

   All coefficients and data values are assumed to be in 1.15 format.
}

.ENTRY  diriir;

diriir: MX0=DM(I0,M1), MY0=PM(I5,M4);
        DO poleloop UNTIL CE;
poleloop:  MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I5,M4);
        MR=MR+MX0*MY0(RND);
        CNTR=AX0;
        DM(I0,M0)=MR1;
        MR=0, MX0=DM(I0,M1), MY0=PM(I6,M4);
        DO zeroloop UNTIL CE;
zeroloop:  MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I6,M4);
        MR=MR+MX0*MY0(RND);
        MODIFY (I0,M2);
        RTS;
```

**Listing 5.4 Direct Form IIR Filter**

### 5.3.2 Cascaded Biquad IIR Filter

A second-order biquad IIR filter section is shown on Figure 5.2. Its transfer function in the z-domain is:

$$H(z) = Y(z)/X(z) = ( B_0 + B_1z^{-1} + B_2z^{-2} )/( 1 + A_1z^{-1} + A_2z^{-2} )$$

where $A_1$, $A_2$, $B_0$, $B_1$ and $B_2$ are coefficients that determine the desired impulse response of the system H(z). Furthermore, the corresponding difference equation for a biquad section is:

$$Y(n) = B_0X(n) + B_1X(n–1) + B_2X(n–2) – A_1Y(n–1) – A_2Y(n–2)$$

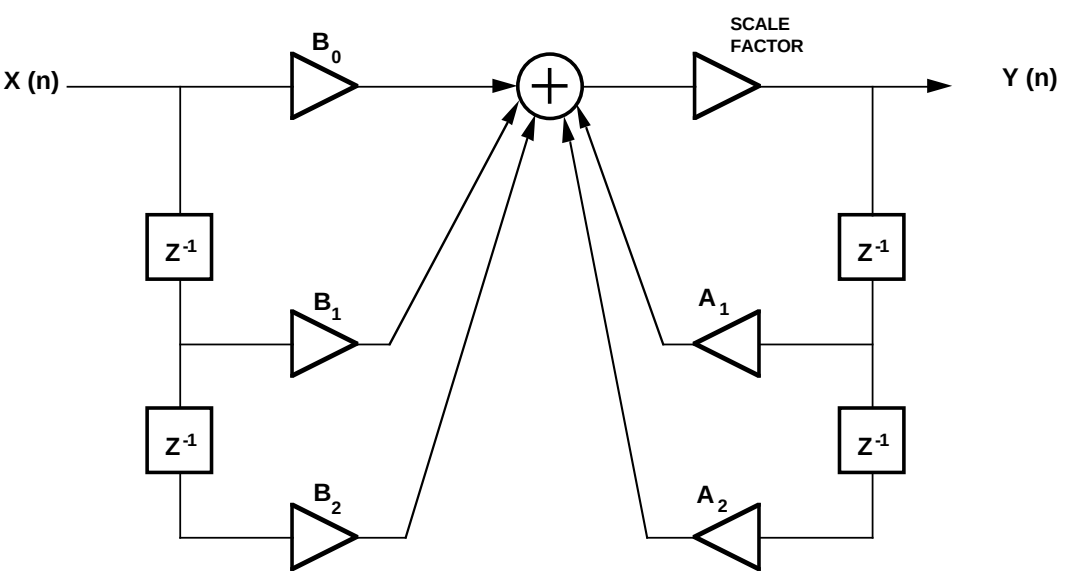


**Figure 5.2 Second-order Biquad IIR Filter Section**

Higher-order filters can be obtained by cascading several biquad sections with appropriate coefficients. Another way to design higher-order filters is to use only one complicated single section. This approach is called the direct form implementation. The biquad implementation executes slower but generates smaller numerical errors than the direct form implementation. The biquads can be scaled separately and then cascaded in order to minimize the coefficient quantization and the recursive

accumulation errors. The coefficients and data in the direct form implementation must be scaled all at once, which gives rise to larger errors. Another disadvantage of the direct form implementation is that the poles of such single-stage high-order polynomials get increasingly sensitive to quantization errors. The second-order polynomial sections (i.e., biquads) are less sensitive to quantization effects.

An ADSP-2100 subroutine that implements a high-order filter is shown in Listing 5.5. The subroutine is arranged as a module and is labeled *biquad_sub*. There are a number of registers that need to be initialized in order to execute this subroutine. It may be sufficient to do this initialization only once (e.g., at powerup) if other executed algorithms do not need these registers. In most typical cases, however, some of these registers may need to be set every time the *biquad_sub* routine is called. It may sometimes be beneficial, from a modular software point of view, to always initialize all the setup registers as a part of this subroutine.

The *biquad_sub* routine takes its input from the SR1 register. This register must contain the 16-bit input X(n). X(n) is assumed to be already computed before this subroutine is called. The output of the filter is also made available in the SR1 register.

After the initial design of a high order filter, all coefficients must be scaled down in each biquad stage separately. This is necessary in order to conform to the 16-bit fixed-point fractional number format as well as to ensure that overflows will not occur in the final multiply-accumulate operations in each stage. The scaled-down coefficients are the ones that get stored in the processor's memory. The operations in each biquad are performed with scaled data and coefficients and are eventually scaled up before being output to the next one. The choice of a proper scaling factor depends greatly on the design objectives, and in some cases it may even be unnecessary. The filter coefficients are usually designed with a commercial software package in higher than 16-bit precision arithmetic. System performance deviates from ideal when such high precision coefficients are quantized to 16 bits and further scaled down. In systems that require stringent specifications, careful simulations of quantization and scaling effects must be performed.

During the initialization of the *biquad_sub* routine, the index register I0 points to the data memory buffer that contains the previous error inputs and the previous biquad section outputs. This buffer must be initialized to zero at powerup unless some nonzero initial condition is desired. The index register I1 points to another buffer in data memory that contains the

individual scale factors for each biquad. The buffer length register L1 is set to zero if the filter has only one biquad section. In the case of multiple biquads, L1 is initialized with the number of biquad sections. The index register I4, on the other hand, points to the circular program memory buffer that contains the scaled biquad coefficients. These coefficients are stored in the order: $B_2$, $B_1$, $B_0$, $A_2$ and $A_1$ for each biquad. All of the individual biquad coefficient groups must be stored in the same order that the biquads are cascaded in, such as: $B_2$, $B_1$, $B_0$, $A_2$, $A_1$, $B_2$*, $B_1$*, $B_0$*, $A_2$*, $A_1$*,$B_2$**, etc. The buffer length register L4 must be set to the value of (5 x number of biquad sections). Finally, the loop counter register CNTR must be set to the number of biquad sections since the filter code will be executed as a loop.

The core of the *biquad_sub* routine starts its execution at the *biquad* label. The routine is organized in a looped fashion where the end of the loop is the instruction labeled *sections*. Each iteration of the loop executes the computations for one biquad. The number of loops to be executed is determined by the CNTR register contents. The SE register is loaded with the appropriate scaling factor for the particular biquad at the beginning of each loop iteration. After this operation, the coefficients and the data values are fetched from memory in the sequence that they have been stored. These numbers are multiplied and accumulated until all of the values for a particular biquad have been accessed. The result of the last multiply/accumulate is rounded to 16 bits and upshifted by the scaling value. At this point the *biquad* loop is executed again, or the filter computations are completed by doing the final update to the delay line. The delay lines for data values are always being updated within the *biquad* loop as well as outside of it.

The filter coefficients must be scaled appropriately so that no overflows occur after the upshifting operation between the biquads. If this is not ensured by design, it may be necessary to include some overflow checking between the biquads.

The execution time for an Nth order *biquad_sub* routine can be calculated as follows (assuming that the appropriate registers have been initialized and N is a power of 2):

ADSP-2101/2102 : (8 x N/2) + 4 processor cycles
ADSP-2100/2100A : (8 x N/2) + 4 + 5 processor cycles

It may take up to a maximum of 12 cycles to initialize the appropriate registers every time the filter is called, but typically this number will be lower.

```
.MODULE     biquad_sub;

{           Nth order cascaded biquad filter subroutine

            Calling Parameters:

               SR1=input X(n)
               I0 —> delay line buffer for X(n-2), X(n-1),
                   Y(n-2), Y(n-1)
               L0 = 0
               I1 —> scaling factors for each biquad section
               L1 = 0  (in the case of a single biquad)
               L1 = number of biquad sections
                   (for multiple biquads)
               I4 —> scaled biquad coefficients
               L4 = 5 x [number of biquads]
               M0, M4 = 1
               M1 = -3
               M2 = 1 (in the case of multiple biquads)
               M2 = 0 (in the case of a single biquad)
               M3 = (1 - length of delay line buffer)

            Return Value:
               SR1 = output sample Y(n)

            Altered Registers:
               SE, MX0, MX1, MY0, MR, SR

            Computation Time (with N even):
               ADSP-2101/2102: (8 x N/2) + 5 cycles
               ADSP-2100/2100A: (8 x N/2) + 5 + 5 cycles

            All coefficients and data values are assumed to be in 1.15 format
}

.ENTRY      biquad;

biquad:     CNTR = number_of_biquads
            DO sections UNTIL CE;
               SE=DM(I1,M2);
               MX0=DM(I0,M0), MY0=PM(I4,M4);
               MR=MX0*MY0(SS), MX1=DM(I0,M0), MY0=PM(I4,M4);
               MR=MR+MX1*MY0(SS), MY0=PM(I4,M4);
               MR=MR+SR1*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
               MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M4);
               DM(I0,M0)=MX1, MR=MR+MX0*MY0(RND);
sections:      DM(I0,M0)=SR1, SR=ASHIFT MR1 (HI);
            DM(I0,M0)=MX0;
            DM(I0,M3)=SR1;
            RTS;
.ENDMOD;
```

**Listing 5.5 Cascaded Biquad IIR Filter**

```
.ENDMOD;
```

## 5.4 LATTICE FILTERS

The lattice filter is used often in the analysis and synthesis of speech, most commonly to simulate the vocal tract. Its physical analogue is a series of cylinders of different radii; each of the filter coefficients represents the amount of energy reflected at a boundary of two cylinders. The all-pole lattice filter is used in voice synthesis (see Chapter 10).

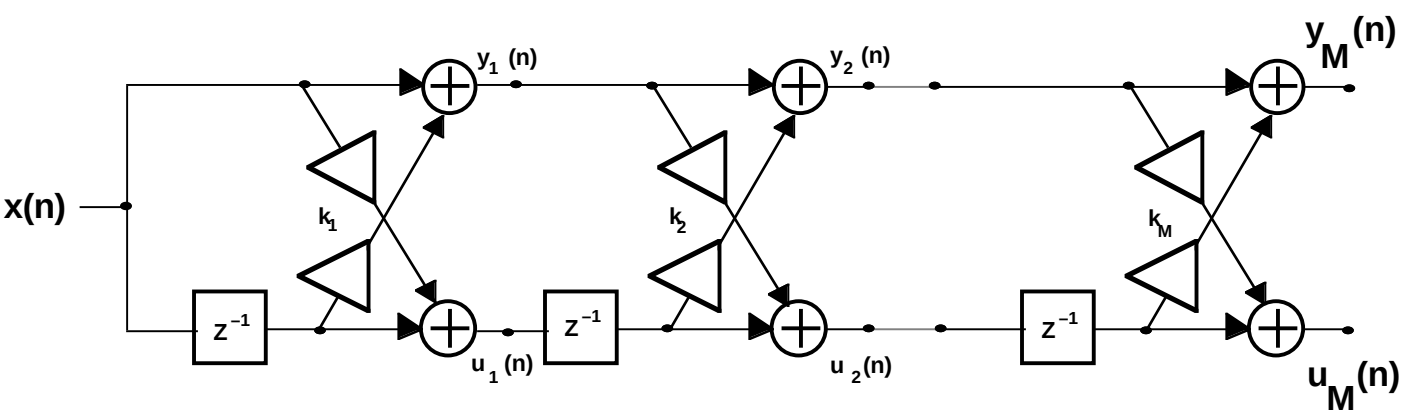


**Figure 5.3 All-Zero Lattice Filter**

### 5.4.1 All-Zero Lattice Filter

The all-zero lattice filter (Bellanger, 1984) is the FIR representation of the lattice filter whose structure is shown in Figure 5.3.

Each stage of the filter has an input and output that are related by the equations

$$y_z(n) = y_{z-1}(n) + k_z u_{z-1}(n-1)$$

$$u_z(n) = k_z y_{z-1}(n) + u_{z-1}(n-1)$$

The initial values of $y_z(n)$ and $u_z(n)$ are both the value of the filter input, x(n).

$y_0(n) = x(n)$

$u_0(n) = x(n)$

For example

$y_1(n) = x(n) + k_1\, x(n–1)$

$u_1(n) = k_1\, x(n) + x(n–1)$

and

$y_2(n) = x(n) + k_1\, (1+k_2)\, x(n–1) + k_2\, x(n–2)$

$u_2(n) = k_2\, x(n) + k_1 (1+k_2)\, x(n–1) + x(n–2)$

The filter output is the output of the last stage.

The ADSP-2100 implementation of the all-zero lattice filter is shown in Listing 5.6. Various registers must be preloaded before this routine is called. The index register I0 should contain the starting address of the input buffer, and I2 should hold the starting address of the output buffer. I3 should contain the starting address of the filter delay line, and I4 should contain the starting address of the coefficient buffer. The length registers L0 and L2 should be set to zero, but L3 and L4 should be set to the order of the filter (number of sections) to make the delay line and coefficient buffers circular. The modify register M3 should be set to one, and the SE register should contain the value needed to maintain a valid output data format. (For example, if two 4.12 numbers are multiplied, the product is a 7.23 number. To obtain a product in 9.21 format, the SE register must be set to –2.) MF, the multiplier feedback register, should contain the value one in the output format. Multiplication by MF is an alternative method of converting output to the correct format. The CNTR register should contain the number of locations in the output buffer.

The *out_loop* loop is executed once for each output data point. CNTR is loaded with the order of the filter, and the first input data point is loaded into MX0. The *latt_loop* loop performs the filtering operation on the input data point.

The first multiplication in the *latt_loop* loop formats the $y_{z–1}(n)$ value into the MR register and also reads in values for $u_{z–1}(n–1)$ and $k_z$. These values are then multiplied and accumulated to produce $y_z(n)$, at the same time

the value $u_{z-1}(n)$ is stored in the delay line for the next pass. The value $y_z(n)$ is reformatted in the shifter for use by the multiplier in the next pass of the *latt_loop* loop.

Next, $u_{z-1}(n–1)$ is formatted into the multiplier to compute the value of $u_z(n)$. This value is then accumulated with the product of $k_z$ and $y_z(n)$. Again, the shifter reformats the value before storage.

```
.MODULE all_zero_lattice_filter;

{
   All Zero Lattice

   Calling Parameters
        CNTR = Length of Output Buffer
        I0 –> Input Buffer                        L1 = 0
        I2 –> Output Buffer                       L2 = 0
        I3 –> Delay Line Buffer (circular)        L3 = Filter Order
        I4 –> Coefficient Buffer (circular)       L4 = Filter Order
        M0 = 1
        M2 = 0
        M3 = 1
        M4 = 1
        SE = Appropriate Scale Factor
        MF = Formatted 1

   Return Values
        Output Buffer Filled

   Altered Registers
        MX0,MX1,MY0,MF,MR,SR,I2,I3,I4

   Computation Time
        (8 × Filter Order + 4) × Output Buffer Length + 3 + 1 cycles
}

.ENTRY z_latt;

z_latt: SR1=0;                                    {Clear SR1 for first pass}
        DO out_loop UNTIL CE;                     {Loop output length}
           CNTR=L3;
           MX0=DM(I0,M0);                         {Get excitation signal}
           DO latt_loop UNTIL CE;                 {Loop through filter}
              MR=MX0*MF (SS), MX1=DM(I3,M2), MY0=PM(I4,M4);   {Get U,K}
              MR=MR+MX1*MY0 (SS), DM(I3,M3)=SR1;              {Compute Yz store U}
              SR=ASHIFT MR1 (HI);                 {Reformat Yz}
              SR=SR OR LSHIFT MR0 (LO);
              MR=MX1*MF (SS);                     {Format Uz-1}
              MX0=SR1, MR=MR+MX0*MY0 (SS);        {Compute Uz and Hold Yz}
              SR=ASHIFT MR1 (HI);                 {Reformat Uz}
latt_loop:    SR=SR OR LSHIFT MR0 (LO);
out_loop:  DM(I2,M0)=MX0;                         {Save output}
        RTS;
```

```
.ENDMOD;
```

**Listing 5.6 All-Zero Lattice Filter**

### 5.4.2 All-Pole Lattice Filter

The all-pole lattice filter, shown in Figure 5.4, relates the variables $x_z(n)$, $u_z(n)$, and $y(n)$ by the following equations (Bellanger, 1984):

$$x_{z-1}(n) = x_z(n) - k_z u_{z-1}(n-1)$$

$$u_z(n) = k_z x_{z-1}(n) + u_{z-1}(n-1)$$

Therefore

$$y(n) = x_1(n) - k_1 y(n-1)$$

$$u_1(n) = k_1 y(n) + y(n-1)$$

and

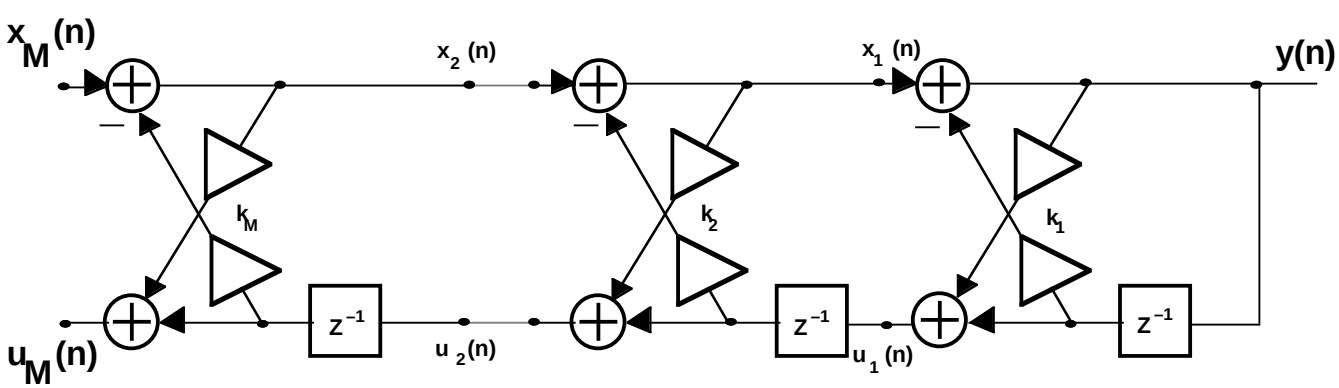


**Figure 5.4 All-Pole Lattice Filter Structure**

$x_1(n) = x_2(n) - k_2 u_1(n-1)$

$u_2(n) = k_2 x_1(n) + u_1(n-1)$

The all-pole lattice filter routine is shown in Listing 5.7. Various registers must be preloaded before the routine is called. The index register I0 should point to the start of the input buffer, I1 to the start of the coefficient buffer, I2 to the start of the output buffer, and I4 to the start of the filter delay line. The length registers L0 and L2 should both be set to zero, and L1 and L4 should be set to the filter order to make the coefficient and delay line buffers circular. The modify registers M0 and M4 should both be set to one; M1 and M5 should both be set to –1. M6 should be set to three and M7 to –2. The SE register, which controls data scaling, should be set to an appropriate value, and AX0 should be set to the order of the filter less one.

The routine loads the first input data value into MY0. The *outloop* loop is executed once for each output data value. The MR register is loaded with the scaled value of $x_z(n)$ at the same time the coefficient $k_z$ and delay line value $u_{z-1}(n-1)$ are loaded. The next instruction computes the value $x_{z-1}(n)$ and also loads the next multiplier operands. The *dataloop* loop performs the remainder of the filtering operation on the data point.

In the *dataloop* loop, $x_{z-1}(n)$ is computed and then shifted to the proper format for the next multiplication. Then the value of $u_z(n)$ is computed

and stored in the delay line. After the *dataloop* loop has been executed, the pointers to the delay line and coefficient buffer are moved to the tops of their buffers at the same time the output of the filter and the last delayed point $u_1(n)$ are saved.

```
.MODULE all_pole_lattice_filter;

{  All-Pole Lattice Filter Routine

   Calling Parameters
        CNTR = Length of Excitation Signal
        I0 —> Excitation Signal                 L0 = 0
        I1 —> Coefficient Buffer (circular)     L1 = Filter Order
        I2 —> Output Buffer                     L2 = 0
        I4 —> Delay Line Buffer (circular)      L4 = Filter Order
        AR = Formatted 1
        M0, M4 = 1                  M1,M5 = -1
        M6 = 3                      M7 = -2
        SE = Appropriate scale value
        AX0 = Filter Order - 1

   Return Values
        Output Buffer Filled

   Altered Registers
        MX0,MY0,MY1,MR,SR,I0,I1,I2,I4
.ENTRY  p_latt;
   Computation Time
        (6 × (Filter Length-1) +8) × Output Buffer Length + 3 + 6 cycles
p_latt: MX0=DM(I0,M0);                          {Get Input data}
        DO outloop UNTIL CE;                    {Loop through output}
}
           CNTR=AX0;
           MR=AR*MY0 (SS), MX0=DM(I1,M0), MY0=PM(I4,M4);          {Get U,K}
           MR=MR-MX0*MY0 (SS), MX0=DM(I1,M0), MY0=PM(I4,M4);      {MR=X10}
           DO dataloop UNTIL CE;                {Loop through filter}
              MR=MR-MX0*MY0 (SS);               {Compute Xz}
              SR=ASHIFT MR1 (HI);               {Reformat Xz}
              MY1=SR1, MR=AR*MY0 (SS);          {Format Uz+1}
              MR=MR+MX0*MY1 (SS), MX0=DM(I1,M0),MY0=PM(I4,M7);    {MR=Uz}
              SR=ASHIFT MR1 (HI);               {Reformat Uz}
dataloop:     PM(I4,M6)=SR1, MR=AR*MY1 (SS);    {Save Uz format Xz}
           MY0=PM(I4,M7), MX0=DM(I1,M1);        {Reset Pointers}
           MY0=DM(I0,M0), SR=ASHIFT MR1 (HI);   {Get new data point}
           DM(I2,M0)=MY1, SR=SR OR LSHIFT MR0 (LO);{Store output}
outloop:   PM(I4,M4)=SR1;                       {Save Y}
        RTS;
.ENDMOD;
```

**Listing 5.7 All-Pole Lattice Filter**