## 5.5 MULTIRATE FILTERS

Multirate filters are digital filters that change the sampling rate of a digitally-represented signal. These filters convert a set of input samples to another set of data that represents the same analog signal sampled at a different rate. A system incorporating multirate filters (a multirate system) can process data sampled at various rates.

Some examples of applications for multirate filters are:

- Sample-rate conversion between digital audio systems
- Narrow-band low-pass and band-pass filters
- Sub-band coding for speech processing in vocoders
- Transmultiplexers for TDM (time-division multiplexing) to FDM (frequency-division multiplexing) translation
- Quadrature modulation
- Digital reconstruction filters and anti-alias filters for digital audio, and
- Narrow-band spectra calculation for sonar and vibration analysis.

For additional information on these topics, see *References* at the end of this chapter.

The two types of multirate filtering processes are decimation and interpolation. Decimation reduces the sample rate of a signal. It eliminates redundant or unnecessary information and compacts the data, allowing more information to be stored, processed, or transmitted in the same amount of data. Interpolation increases the sample rate of a signal. Through calculations on existing data, interpolation fills in missing information between the samples of a signal. Decimation reduces a sample rate by an integer factor M, and interpolation increases a sample rate by an integer factor L. Non-integer rational (ratio of integers) changes in sample rate can be achieved by combining the interpolation and decimation processes.

The ADSP-2100 programs in this chapter demonstrate decimation and interpolation as well as efficient rational changes in sample rate. Cascaded stages of decimation and interpolation, which are required for large rate changes (large values of L and M) and are useful for implementing narrow-band low-pass and band-pass filters, are also demonstrated.

### 5.5.1 Decimation

Decimation is equivalent to sampling a discrete-time signal. Continuous-time (analog) signal sampling and discrete-time (digital) signal sampling are analogous.

#### *5.5.1.1 Continuous-Time Sampling*

Figure 5.5 shows the periodic sampling of a continuous-time signal, $x_c(t)$, where t is a continuous variable. To sample $x_c(t)$ at a uniform rate every T seconds, we modulate (multiply) $x_c(t)$ by an impulse train, s(t):

$$s(t) = \sum_{n=-\infty}^{+\infty} \partial(t-nT)$$

The resulting signal is a train of impulses, spaced at intervals of T, with amplitudes equal to samples of $x_c(t)$. This impulse train is converted to a set of values x(n), where n is a discrete-time variable and $x(n)= x_c(nT)$. Thus, $x_c(t)$ is quantized both in time and in amplitude to form digital values x(n). The modulation process is equivalent to a track-and-hold circuit, and the quantization process is equivalent to an analog-to-digital (A/D) converter.

Figure 5.6 shows a frequency-domain interpretation of the sampling process. $X_c(w)$ is the spectrum of the continuous-time signal $x_c(t)$. S(w), a train of impulses at intervals of the sampling frequency, $F_s$ or 1/T, is the frequency transform of s(t). Because modulation in the time domain is equivalent to convolution in the frequency domain, the convolution of $X_c(w)$ and S(w) yields the spectrum of x(n). This spectrum is a sequence of periodic repetitions of $X_c(w)$, called *images* of $X_c(w)$, each centered at multiples of the sampling frequency, $F_s$.

The frequency that is one-half the sampling frequency ($F_s/2$) is called the Nyquist frequency. The analog signal $x_c(t)$ must be bandlimited before sampling to at most the Nyquist frequency. If $x_c(t)$ is not bandlimited, the images created by the sampling process overlap each other, mirroring the spectral energy around $nF_s/2$, and thus corrupting the signal representation. This phenomenon is called aliasing. The input $x_c(t)$ must pass through an analog anti-alias filter to eliminate any frequency component above the Nyquist frequency.

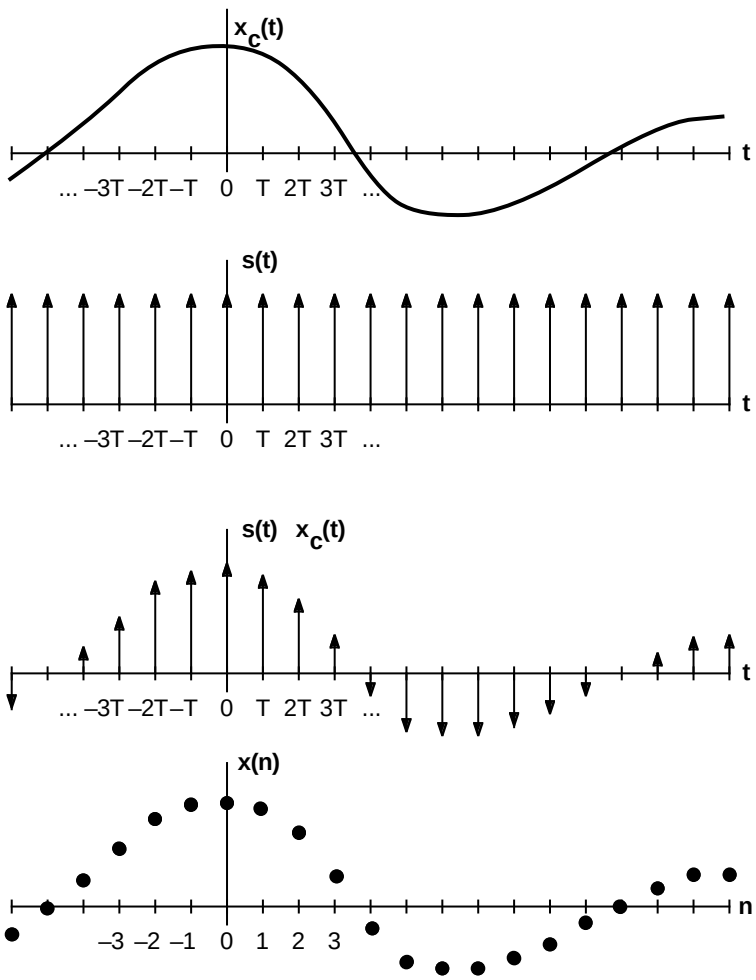


**Figure 5.5 Sampling Continuous-Time Signal**

### *5.5.1.2 Discrete-Time Sampling*

Figure 5.7 shows the sampling of a discrete-time signal, x(n). The signal x(n) is multiplied by s(n), a train of impulses occurring at every integer multiple of M. The resulting signal consists of every Mth sample of x(n) with all other samples zeroed out. In this example, M is 4; the decimated version of x(n) is the result of discarding three out of every four samples. The original sample rate, $F_s$, is reduced by a factor of 4; $F_s' = F_s / 4$.

# 5 Digital Filters

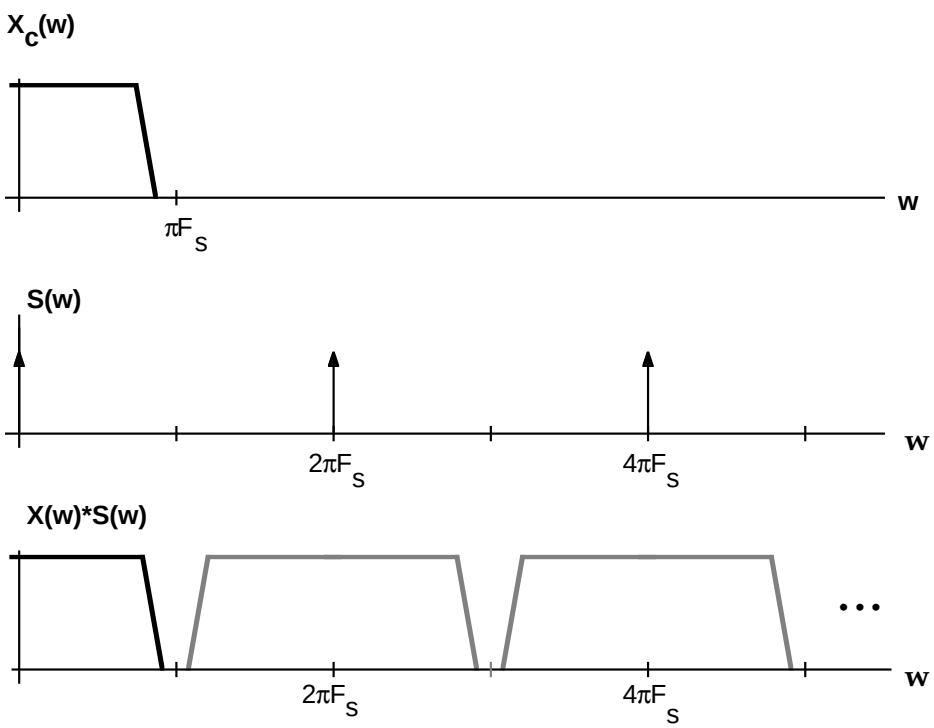


**Figure 5.6 Spectrum of Continuous-Time Signal Sampling**

Figure 5.8 shows the frequency-domain representation of sampling the discrete-time signal. The spectrum of the waveform to be sampled is X(w). The original analog signal was bandlimited to $w_s/2$ before being sampled, where $w_s$ is the original sample rate. The shaded lines indicate the images of X(w) above $w_s/2$. Before decimation, X(w) must be bandlimited to one-half the *final* sample rate, to eliminate frequency components that could alias. H(w) is the transfer function of the low-pass filter required for a decimation factor (M) of four. Its cutoff frequency is $w_s/8$. This digital anti-alias filter performs the equivalent function as the analog anti-alias filter described in the previous section.

W(w) is a version of X(w) filtered by H(w), and W(w)*S(w) is the result of convolving W(w) with the sampling function S(w), the transform of s(n). The shaded lines in W(w)*S(w) represent the images of W(w) formed by this convolution. These images show the energy in the original signal that would alias if we decimated X(w) without bandlimiting it. X(w') is the result of decimating W(w)*S(w) by four. Decimation effectively spreads out the energy of the original sequence and eliminates unwanted information located above $w_s/M$.

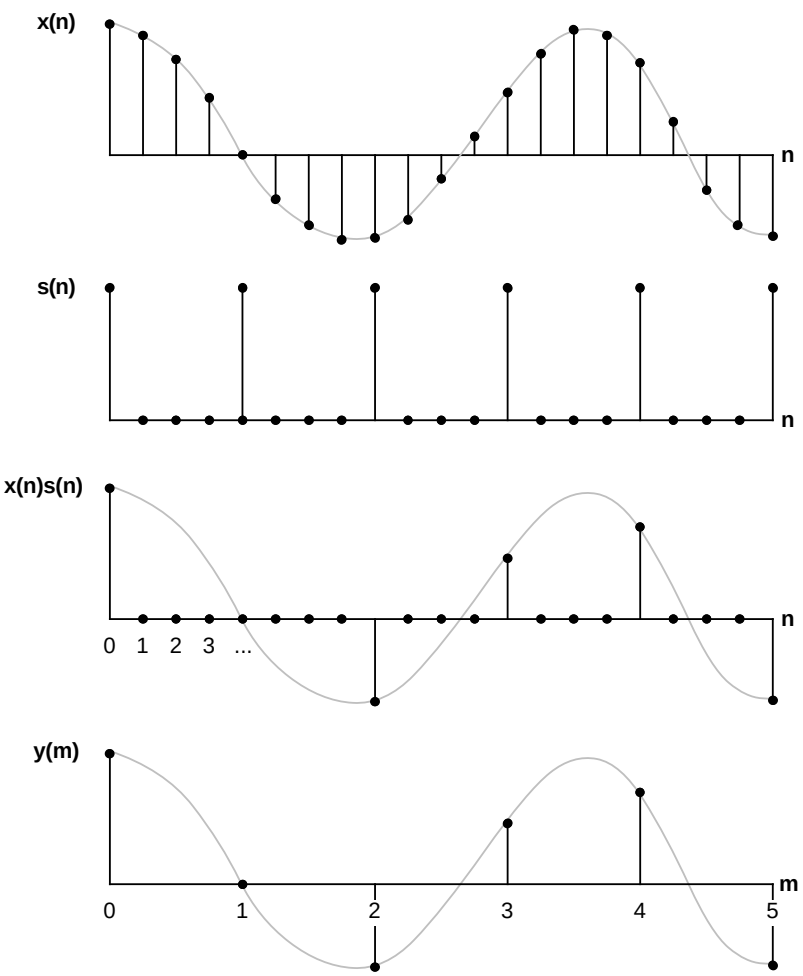


**Figure 5.7 Sampling Discrete -Time Signal**

The decimation and anti-alias functions are usually grouped together into one function called a decimation filter. Figure 5.9 shows a block diagram of a decimation filter. The input samples x(n) are put through the digital anti-alias filter, h(k). The box labeled with a down-arrow and M is the sample rate compressor, which discards M–1 out of every M samples. Compressing the filtered input w(n) results in y(m), which is equal to w(Mm).

Data acquisition systems such as the digital audio tape recorder can take advantage of decimation filters to avoid using expensive high-

# 5 Digital Filters

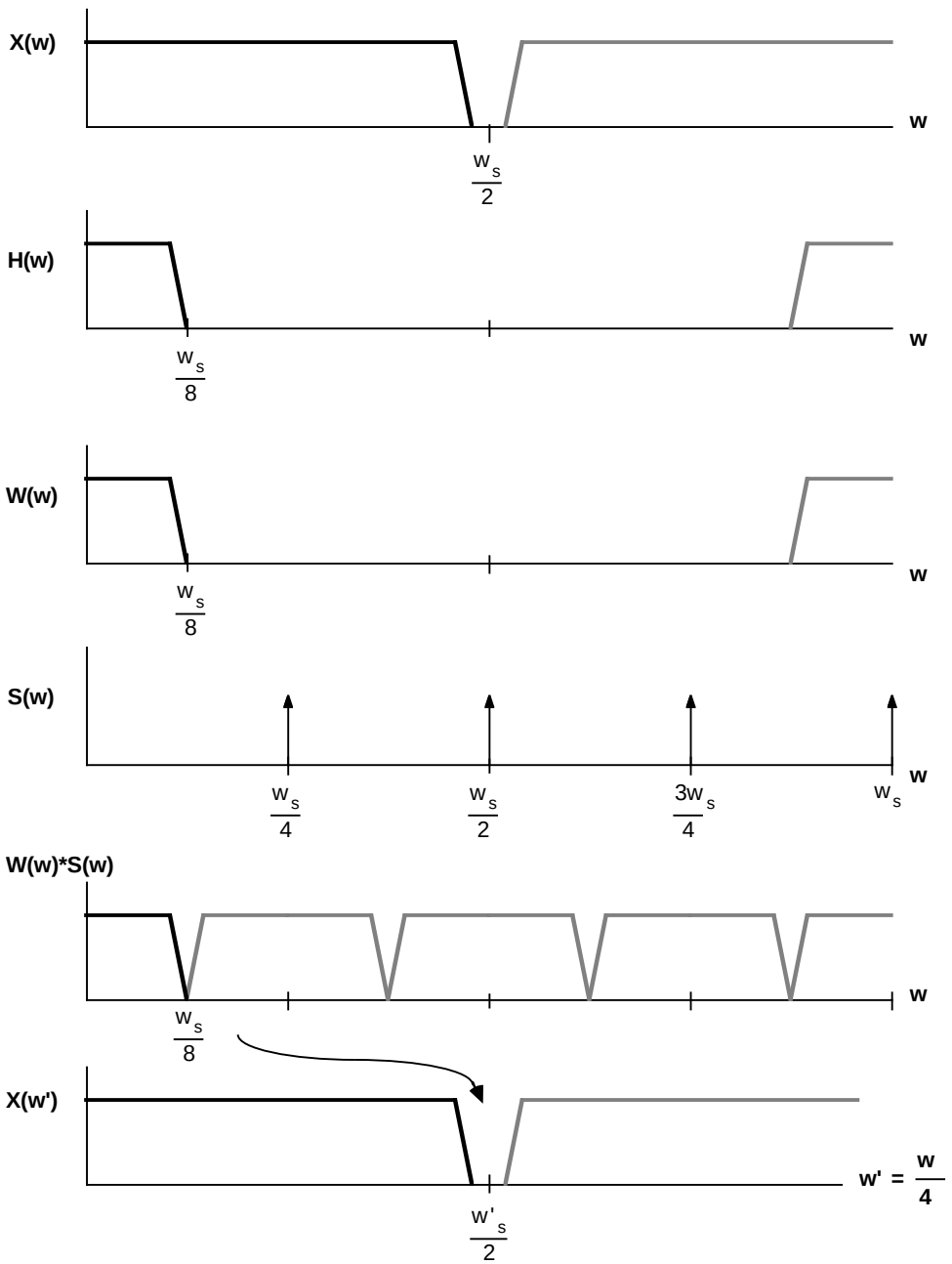


**Figure 5.8 Spectrum of Discrete-Time Signal Sampling**

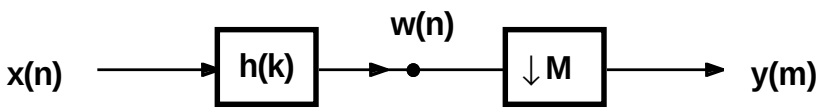


**Figure 5.9 Decimation Filter Block Diagram**

performance analog anti-aliasing filters. Such a system over-samples the input signal (usually by a factor of 2 to 8) and then decimates to the required sample rate. If the system over-samples by two, the transition band of the front end filter can be twice that required in a system without decimation, thus a relatively inexpensive analog filter can be used.

### *5.5.1.3 Decimation Filter Structure*

The decimation algorithm can be implemented in an FIR (Finite Impulse Response) filter structure. The FIR filter has many advantages for multirate filtering including: linear phase, unconditional stability, simple structure, and easy coefficient design. Additionally, the FIR structure in multirate filters provides for an increase in computational efficiency over IIR structures. The major difference between the IIR and the FIR filter is that the IIR filter must calculate all outputs for all inputs. The FIR multirate filter calculates an output for every Mth input. For a more detailed description of the FIR and IIR filters, refer to Crochiere and Rabiner, 1983; see *References* at the end of this chapter.

The impulse response of the anti-imaging low-pass filter is h(n). A time-series equation filtering x(n) is the convolution

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

where N is the number of coefficients in h(n). N is the order, or number of taps, in the filter. The application of this equation to implement the filter response $H(e^{jw})$ results in an FIR filter structure.

Figure 5.10a, on the next page, shows the signal flowgraph of an FIR decimation filter. The N most recent input samples are stored in a delay line; z–1 is a unit sample delay. N samples from the delay line are multiplied by N coefficients and the resulting products are summed to form a single output sample w(n). Then w(n) is down-sampled by M using the rate compressor.

# 5 Digital Filters

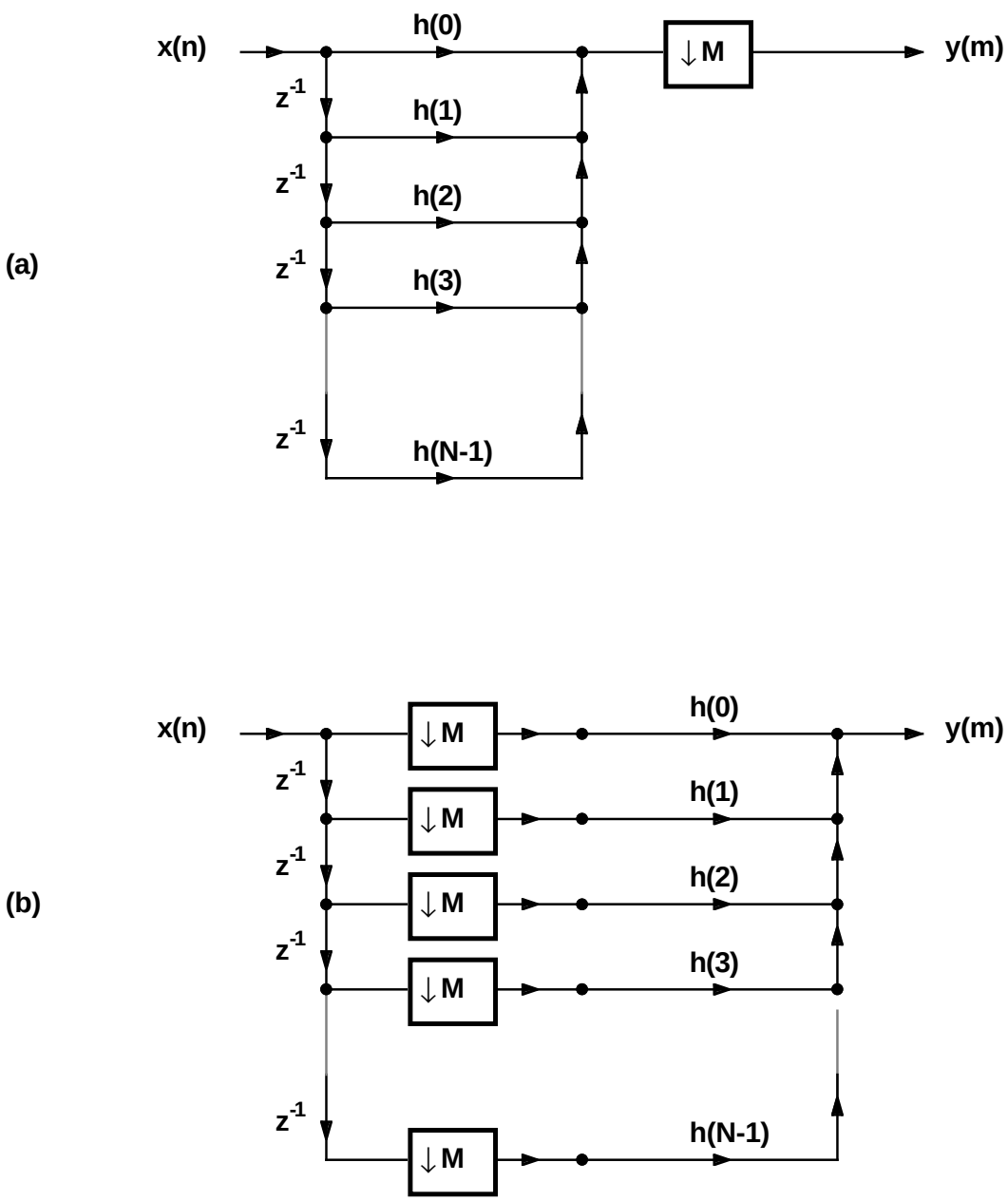


Figure 5.10 FIR Form Decimation Filter

It is not necessary to calculate the samples of w(n) that are discarded by the rate compressor. Accordingly, the rate compressor can be moved in front of the multiply/accumulate paths, as shown in Figure 5.10b. This change reduces the required computation by a factor of M. This filter structure can be implemented by updating the delay line with M inputs before each output sample is calculated.

Substitution of the relationship between w(n) and y(m) into the convolution results in the decimation filtering equation

$$y(m) = \sum_{k=0}^{N-1} h(k)\, x(Mm-k)$$

Some of the implementations shown in textbooks on digital filters take advantage of the symmetry in transposed forms of the FIR structure to reduce the number of multiplications required. However, such a reduction of multiplications results in an increased number of additions. In this application, because the ADSP-2100 is capable of both multiplying and accumulating in one cycle, trading off multiplication for addition is a useless technique.

#### 5.5.1.4 ADSP-2100 Decimation Algorithm

Figure 5.11 shows a flow chart of the decimation filter algorithm used for the ADSP-2100 routine. The decimator calculates one output for every M inputs to the delay line.

External hardware causes an interrupt at the input sample rate $F_s$, which triggers the program to fetch an input data sample and store it in the *data* circular buffer. The index register that points into this data buffer is then incremented by one, so that the next consecutive input sample is written to the next address in the buffer. The counter is then decremented by one and compared to zero. If the counter is not yet zero, the algorithm waits for another input sample. If the counter has decremented to zero, the algorithm calculates an output sample, then resets the counter to M so that the next output will be calculated after the next M inputs.

The output is the sum of products of N data buffer samples in a circular buffer and N coefficients in another circular buffer. Note that M input samples are written into the data buffer before an output sample is calculated. Therefore, the resulting output sample rate is equal to the input rate divided by the decimation factor: $F_s' = F_s / M$.

# 5 Digital Filters

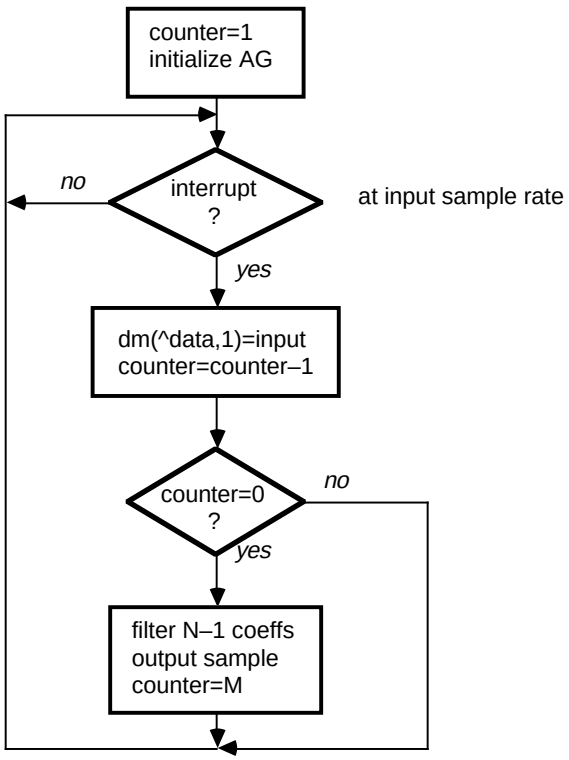


**Figure 5.11 Decimator Flow Chart**

For additional information on the use of the ADSP-2100's address generators for circular buffers, see the *ADSP-2100 User's Manual,* Chapter 2.

The ADSP-2100 program for the decimation filter is shown in Listing 5.8. Inputs to this filter routine come from the memory-mapped port *adc*, and outputs go to the memory-mapped port *dac*. The filter's I/O interfacing hardware is described in more detail later in this chapter.

```
{DECIMATE.dsp
Real time Direct Form FIR Filter, N taps, decimates by M for a
decrease of 1/M times the input sample rate.

   INPUT: adc
   OUTPUT: dac
}

.MODULE/RAM/ABS=0   decimate;
.CONST              N=300;
.CONST              M=4;                  {decimate by factor of M}
.VAR/PM/RAM/CIRC    coef[N];
.VAR/DM/RAM/CIRC    data[N];
.VAR/DM/RAM         counter;
.PORT               adc;
.PORT               dac;
.INIT               coef:<coef.dat>;

                RTI;                {interrupt 0}
                RTI;                {interrupt 1}
                RTI;                {interrupt 2}
                JUMP sample;        {interrupt 3= input sample rate}

initialize:     IMASK=b#0000;       {disable all interrupts}
                ICNTL=b#01111;      {edge sensitive interrupts}
                SI=M;               {set decimation counter to M}
                DM(counter)= SI;    {for first input data sample}
                I4=^coef;           {setup a circular buffer in PM}
                L4=%coef;
                M4=1;               {modifier for coef is 1}
                I0=^data;           {setup a circular buffer in DM}
                L0=%data;
                M0=1;               {modifier for data is 1}
                IMASK=b#1000;       {enable interrupt 3}
wait_interrupt: JUMP wait_interrupt;      {infinite wait loop}

{__________Decimator, code executed at the sample rate_________________}
sample:         AY0=DM(adc);
                DM(I0,M0)=AY0;      {update delay line with newest}
                AY0=DM(counter);
                AR=AY0-1;           {decrement and update counter}
                DM(counter)=AR;
                IF NE RTI;          {test and return if not M times}
```

***(listing continues on next page)***

# 5 Digital Filters

```
{_________code below executed at 1/M times the sample rate_____________}
do_fir:         AR=M;                    {reset the counter to M}
                DM(counter)=AR;
                CNTR=N - 1;
                MR=0, MX0=DM(I0,M0), MY0=PM(I4,M4);
                DO taploop UNTIL CE;     {N-1 taps of FIR}
taploop:           MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
                MR=MR+MX0*MY0(RND);      {last tap with round}
                IF MV SAT MR;            {saturate result if overflow}
                DM(dac)=MR1;             {output data sample}
                RTI;
.ENDMOD;
```

**Listing 5.8 Decimation Filter**

The routine uses two circular buffers, one for data samples and one for coefficients, that are each N locations long. The *coef* buffer is located in program memory and stores the filter coefficients. Each time an output is calculated, the decimator accesses all these coefficients in sequence, starting with the first location in *coef*. The I4 index register, which points to the coefficient buffer, is modified by one (from modify register M0) each time it is accessed. Therefore, I4 is always modified back to the beginning of the coefficient buffer after the calculation is complete.

The FIR filter equation starts the convolution with the most recent data sample and accesses the oldest data sample last. Delay lines implemented with circular buffers, however, access data in the opposite order. The oldest data sample is fetched first from the buffer and the newest data sample is fetched last. Therefore, to keep the data/coefficient pairs together, the coefficients must be stored in memory in reverse order.

The relationship between the address and the contents of the two circular buffers (after N inputs have occurred) is shown in the table below. The *data* buffer is located in data memory and contains the last N data samples input to the filter. Each pass of the filter accesses the locations of both buffers sequentially (the pointer is modified by one), but the first address accessed is not always the first location in the buffer, because the decimation filter inputs M samples into the delay line before starting each filter pass. For each pass, the first fetch from the data buffer is from an address M greater than for the previous pass. The data delay line moves forward M samples for every output calculated.

| *Data* | | *Coefficient* | |
|---|---|---|---|
| DM(0) | = x(n–(N–1)) oldest | PM(0) | = h(N–1) |
| DM(1) | = x(n–(N–2)) | PM(1) | = h(N–2) |
| DM(2) | = x(n–(N–3)) | PM(2) | = h(N–3) |
| | • | | • |
| | • | | • |
| | • | | • |
| DM(N–3) | = x(n–2) | PM(N–3) | = h(2) |
| DM(N–2) | = x(n–1) | PM(N–2) | = h(1) |
| DM(N–1) | = x(n–0) newest | PM(N–1) | = h(0) |

A variable in data memory is used to store the decimation counter. One of the processor's registers could have been used for this counter, but using a memory location allows for expansion to multiple stages of decimation (described in *Multistage Implementations*, later in this chapter).

The number of cycles required for the decimation filter routine is shown below. The ADSP-2100 takes one cycle to calculate each tap (multiply and accumulate), so only 18+N cycles are necessary to calculate one output sample of an N-tap decimator. The 18 cycles of overhead for each pass is just six cycles greater than the overhead of a non-multirate FIR filter.

| | |
|---|---|
| Interrupt response | 2 cycles |
| Fetch input | 1 cycle |
| Write input to data buffer | 1 cycle |
| Decrement and test counter | 4 cycles |
| Reload counter with M | 2 cycles |
| FIR filter pass | 7 + N cycles |
| Return from interrupt | 1 cycle |
| | |
| Maximum total | 18 + N cycles/output |

#### 5.5.1.5 A More Efficient Decimator

The routine in Listing 5.8 requires that the 18+N cycles needed to calculate an output occur during the first of the M input sample intervals. No calculations are done in the remaining M–1 intervals. This limits the number of filter taps that can be calculated in real time to:

$$N = \frac{1}{F_s\, t_{CLK}} - 18$$

where $t_{CLK}$ is the instruction cycle time of the processor.

An increase in this limit by a factor of M occurs if the program is modified so that the M data inputs overlap the filter calculations. This more efficient version of the program is shown in Listing 5.9.

In this example, a circular buffer *input_buf* stores the M input samples. The code for loading *input_buf* is placed in an interrupt routine to allow the input of data and the FIR filter calculations to occur simultaneously.

A loop waits until the input buffer is filled with M samples before the filter output is calculated. Instead of counting input samples, this program determines that M samples have been input when the input buffer's index register I0 is modified back to the buffer's starting address. This strategy saves a few cycles in the interrupt routine.

After M samples have been input, a second loop transfers the data from *input_buf* to the data buffer. An output sample is calculated. Then the program checks that at least one sample has been placed in *input_buf*. This check prevents a false output if the output calculation occurs in less than one sample interval. Then the program jumps back to wait until the next M samples have been input.

This more efficient decimation filter spreads the calculations over the output sample interval $1/F_s'$ instead of the input interval $1/F_s$. The number of taps that can be calculated in real time is:

$$N = \frac{M}{F_s\, t_{CLK}} - 20 - 2M - 6(M-1)$$

which is approximately M times greater than for the first routine.

```
{DEC_EFF.dsp
Real time Direct Form FIR Filter, N taps, decimates by M for a decrease of 1/M times
the input sample rate. This version uses an input buffer to allow the filter
computations to occur in parallel with inputs. This allows larger order filter for a
given input sample rate. To save time, an index register is used for the input buffer
as well as for a decimation counter.

   INPUT: adc
   OUTPUT: dac
}

.MODULE/RAM/ABS=0   eff_decimate;
.CONST              N=300;
.CONST              M=4;                  {decimate by factor of M}
.VAR/PM/RAM/CIRC    coef[N];
.VAR/DM/RAM/CIRC    data[N];
.VAR/DM/RAM/CIRC    input_buf[M];
.PORT               adc;
.PORT               dac;
.INIT               coef:<coef.dat>;

        RTI;                {interrupt 0}
        RTI;                {interrupt 1}
        RTI;                {interrupt 2}
        JUMP sample;        {interrupt 3= input sample rate}

initialize:  IMASK=b#0000;          {disable all interrupts}
             ICNTL=b#01111;         {edge sensitive interrupts}
             I4=^coef;              {setup a circular buffer in PM}
             L4=%coef;
             M4=1;            {modifier for coef is 1}
             I0=^data;              {setup a circular buffer in DM}
             L0=%data;
             M0=1;            {modifier for data is 1}
             I1=^input_buf;         {setup a circular buffer in DM}
             L1=%input_buf;
             IMASK=b#1000;          {enable interrupt 3}

wait_M:      AX0=I1;                {wait for M inputs}
             AY0=^input_buf;
             AR=AX0-AY0;      {test if pointer is at start}
             IF NE JUMP wait_M;
```

*(listing continues on next page)*

```
{____________code below executed at 1/M times the sample rate___________}
             CNTR=M;
             DO load_data UNTIL CE;
                  AR=DM(I1,M0);
load_data:        DM(I0,M0)=AR;

fir:         CNTR=N - 1;
             MR=0, MX0=DM(I0,M0), MY0=PM(I4,M4);
             DO taploop UNTIL CE;    {N-1 taps of FIR}
taploop:          MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
             MR=MR+MX0*MY0(RND);    {last tap with round}
             IF MV SAT MR;          {saturate result if overflow}
             DM(dac)=MR1;           {output data sample}

wait_again:  AX0=I1;
             AY0=^input_buf;
             AR=AX0-AY0;      {test and wait if i1 still}
             IF EQ JUMP wait_again; {points to start of input_buf}
             JUMP wait_M;

{___________sample input, code executed at the sample rate_____________}
sample:      ENA SEC_REG;           {so no registers will get lost}
             AY0=DM(adc);           {get input sample}
             DM(I1,M0)=AY0;         {load in M long buffer}
             RTI;
.ENDMOD;
```

**Listing 5.9 Efficient Decimation Filter**

### 5.5.2 Decimator Hardware Configuration

Both decimation filter programs assume an ADSP-2100 system with the I/O hardware configuration shown in Figure 5.12. The processor is interrupted by an interval timer at a frequency equal to the input sample rate $F_s$ and responds by inputting a data value from the A/D converter. The track/hold (sampler) and the A/D converter (quantizer) are also clocked at this frequency. The D/A converter on the filter output is clocked at a rate of $F_s/M$, which is generated by dividing the interval timer frequency by M.

To keep the output signal jitter-free, it is important to derive the D/A converter's clock from the interval timer and not from the ADSP-2100. The sample period of the analog output should be disassociated from writes to the D/A converter. If an instruction-derived clock is used, any conditional

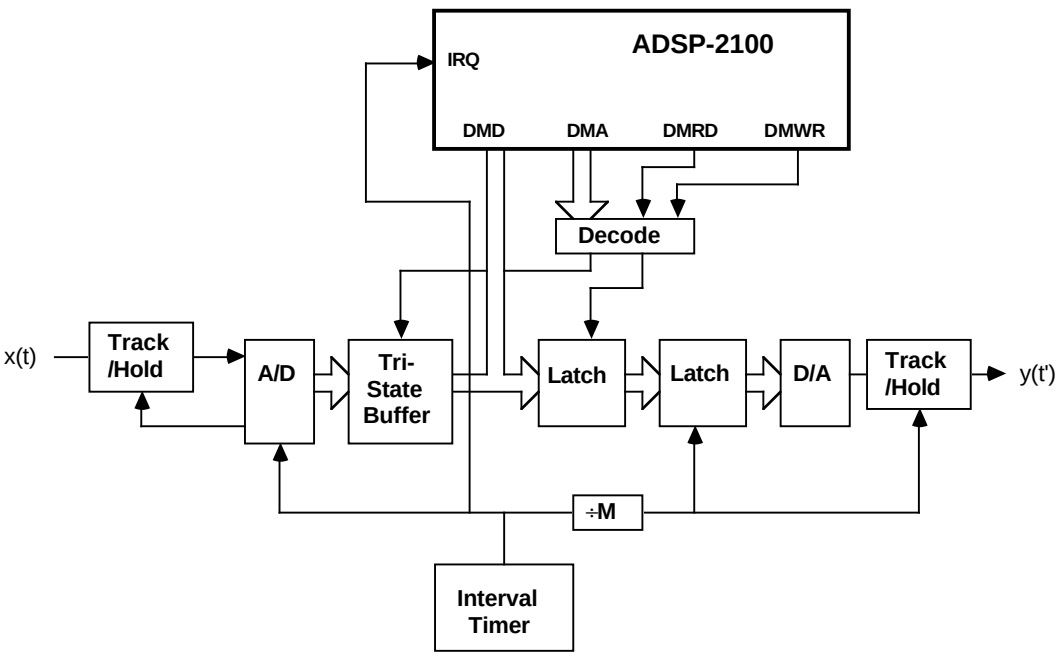


**Figure 5.12 Decimator Hardware**

instructions in the program could branch to different length program paths, causing the output samples to be spaced unequally in time. The D/A converter must be double-buffered to accommodate the interval-time-derived clock. The ADSP-2100 outputs data to one latch. Data from this latch is fed to a second latch that is controlled by an interval-timer-derived clock.

### 5.5.3 Interpolation

The process of recreating a continuous-time signal from its discrete-time representation is called reconstruction. Interpolation can be thought of as the reconstruction of a discrete-time signal from another discrete-time signal, just as decimation is equivalent to sampling the samples of a signal. Continuous-time (analog) signal reconstruction and discrete-time (digital) signal reconstruction are analogous.

Figure 5.13, on the following page, illustrates the reconstruction of a continuous-time signal from a discrete-time signal. The discrete-time signal x(n) is first made continuous by forming an impulse train with amplitudes at times nT equal to the corresponding samples of x(n). In a

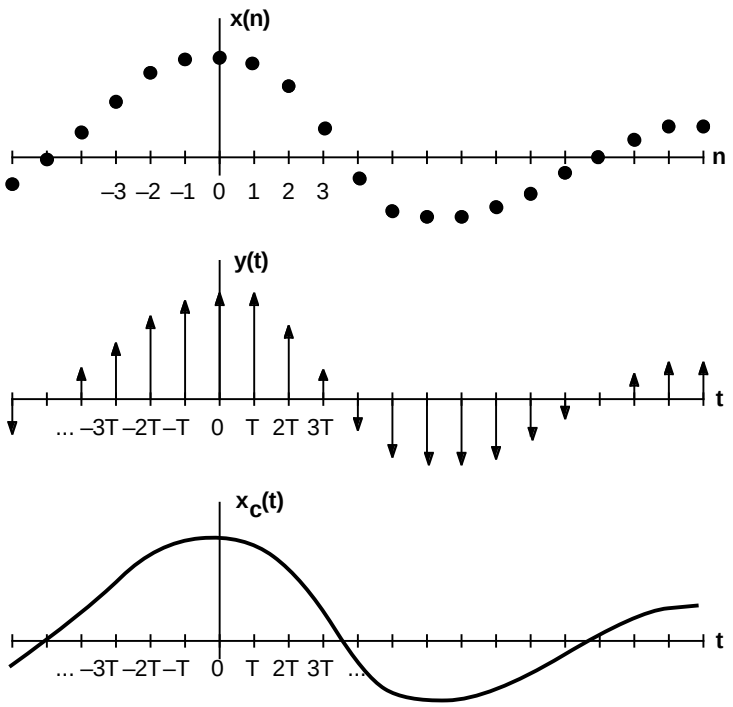


**Figure 5.13 Reconstruction of a Continuous-Time Signal**

real system, a D/A converter performs this operation. The result is a continuous signal y(t), which is smoothed by a low-pass anti-imaging filter (also called a reconstruction filter), to produce the reconstructed analog signal $x_c(t)$. The frequency domain representation of y(t) in Figure 5.14 shows that images of the original signal appear in the discrete- to continuous-time conversion. These images are eliminated by the anti-imaging filter, as shown in the spectrum of the resulting signal $X_c(w)$.

#### *5.5.3.1 Reconstruction of a Discrete-Time Signal*

Figure 5.15 shows the interpolation by a factor of L (4 in this example) of the discrete-time signal x(n). This signal is expanded by inserting L–1 zero-valued samples between its data samples. The resulting signal w(m) is low-pass filtered to produce y(m). The insertion of zeros and the smoothing filter fill in the data missing between the samples of x(n). Because one sample of x(n) corresponds to L samples of y(m), the sample rate is increased by a factor of L.

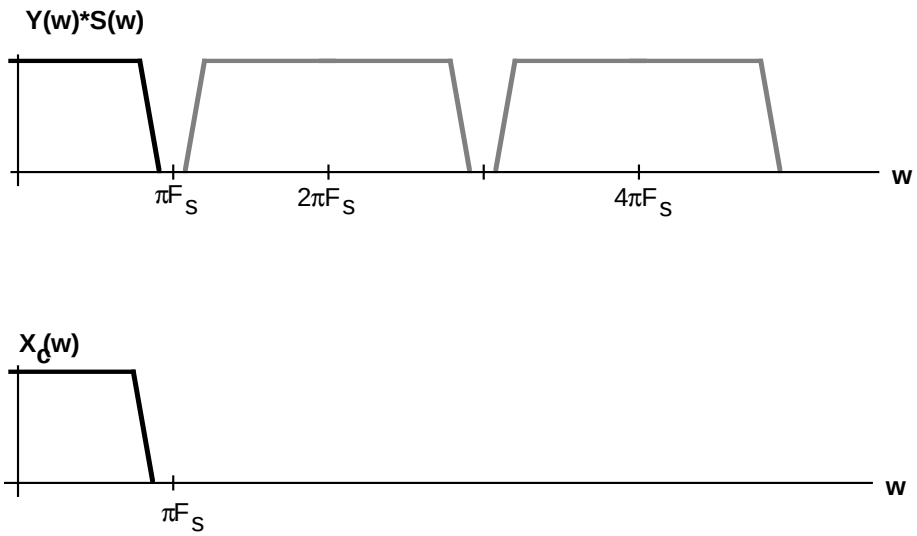


**Figure 5.14 Spectrum of Continuous-Time Signal Reconstruction**

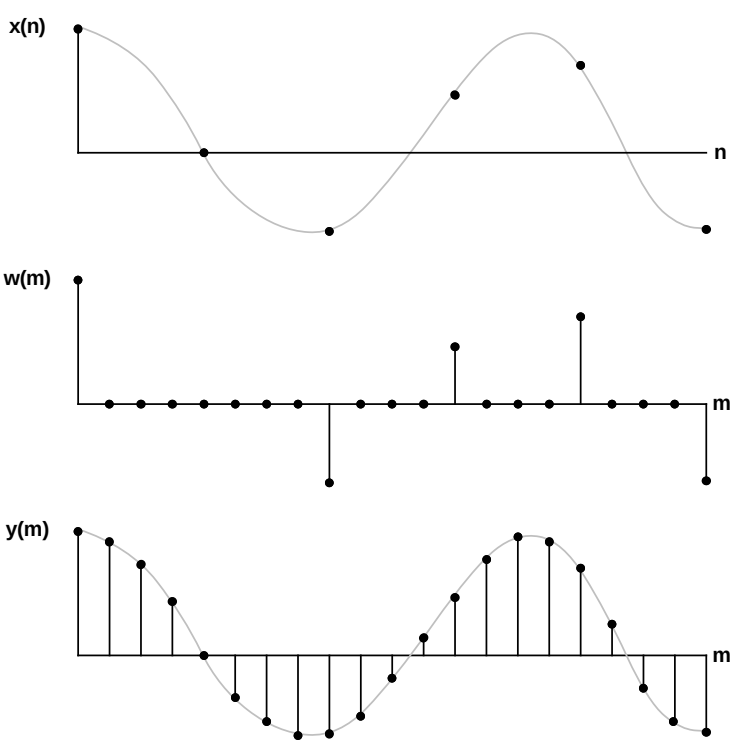


**Figure 5.15 Interpolation of Discrete-Time Signal**

# 5 Digital Filters

Figure 5.16 shows the frequency interpretation of interpolation. $F_s'$ is the input sample frequency and $F_s''$ is the output sample frequency. $F_s''$ is equal to $F_s'$ multiplied by the interpolation factor L (3 in this example). H(w) is the response of the filter required to eliminate the images in W(w). The lower stopband frequency of H(w) must be less than $F_s''/2L$, which is the Nyquist frequency of the original signal. Thus filtering by H(w) accomplishes the function of a digital anti-imaging filter.

Digital audio systems such as compact disk and digital audio tape players frequently use interpolation (oversampling) techniques to avoid using costly high performance analog reconstruction filters. The anti-imaging function in the digital interpolator allows these systems to use inexpensive low-order analog filters on the D/A outputs.

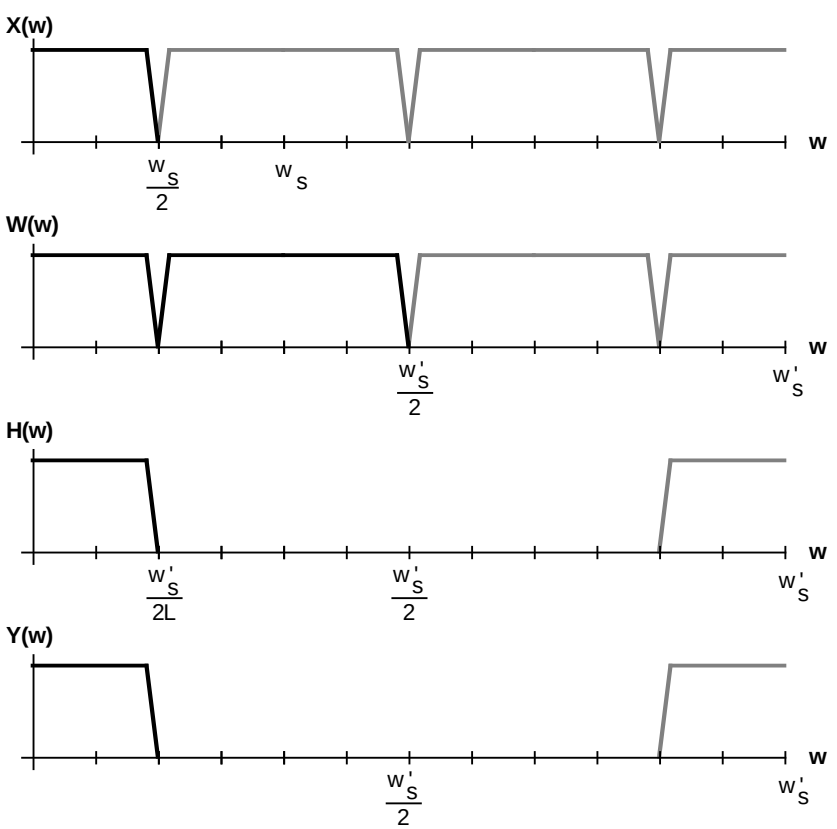


**Figure 5.16 Spectrum of Interpolation**

### 5.5.3.2 Interpolation Filter Structure

Figure 5.17a shows a block diagram of an interpolation filter. The two major differences from the decimation filter are that the interpolator uses a sample rate expander instead of the sample rate compressor and that the interpolator's low-pass filter is placed after the rate expander instead of before the rate compressor. The rate expander, which is the block labeled with an up-arrow and L, inserts L–1 zero-valued samples after each input sample. The resulting w(m) is low-pass filtered to produce y(m), a smoothed, anti-imaged version of w(m). The transfer function of the interpolator H(k) incorporates a gain of 1/L because the L–1 zeros inserted by the rate expander cause the energy of each input to be spread over L output samples.

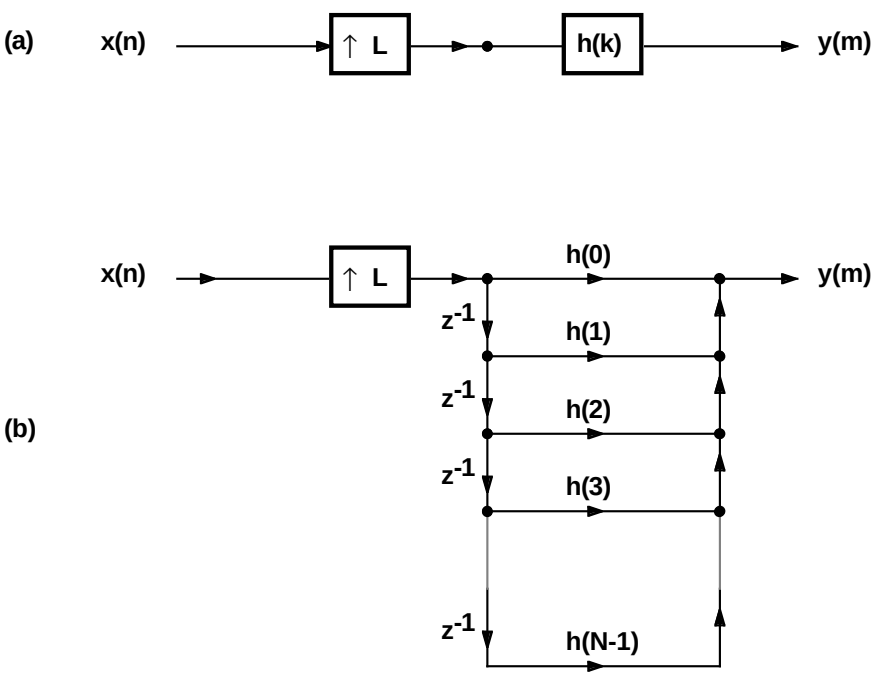


**Figure 5.17 Interpolation Filter Block Diagram**

The low-pass filter of the interpolator uses an FIR filter structure for the same reasons that an FIR filter is used in the decimator, notably computational efficiency. The convolution equation for this filter is

$$y(m) = \sum_{k=0}^{N-1} h(k)\, w(m-k)$$

N–1 is the number of filter coefficients (taps) in h(k), w(m–k) is the rate expanded version of the input x(n), and w(m–k) is related to x(n) by

$$w(m-k) = \begin{cases} x((m-k)/L) & \text{for } m-k = 0, \pm L, \pm 2L, \ldots \\ 0 & \text{otherwise} \end{cases}$$

The signal flow graph that represents the interpolation filter is shown in Figure 5.17b, on previous page. A delay line of length N is loaded with an input sample followed by L–1 zeros, then the next input sample and L–1 zeros, and so on. The output is the sum of the N products of each sample from the delay line and its corresponding filter coefficient. The filter calculates an output for every sample, zero or data, loaded into the delay line.

An example of the interpolator operation is shown in the signal flowgraph in Figure 5.18. The contents of the delay line for three consecutive passes of the filter are highlighted. In this example, the interpolation factor L is 3. The delay line is N locations long, where N is the number of coefficients of the filter; N=9 in this example. There are N/L or 3 data samples in the delay line during each pass. The data samples x(1), x(2), and x(3) in the first pass are separated by L–1 or 2 zeros inserted by the rate expander. The zero-valued samples contribute (L–1)N/L or 6 zero-valued products to the output result. These (L–1)N/L multiplications are unnecessary and waste processor capacity and execution time.

A more efficient interpolation method is to access the coefficients and the data in a way that eliminates wasted calculations. This method is accomplished by removing the rate expander to eliminate the storage of the zero-valued samples and shortening the data delay line from N to N/L locations. In this implementation, the data delay line is updated only after L outputs are calculated. The same N/L (three) data samples are accessed for each set of L output calculations. Each output calculation

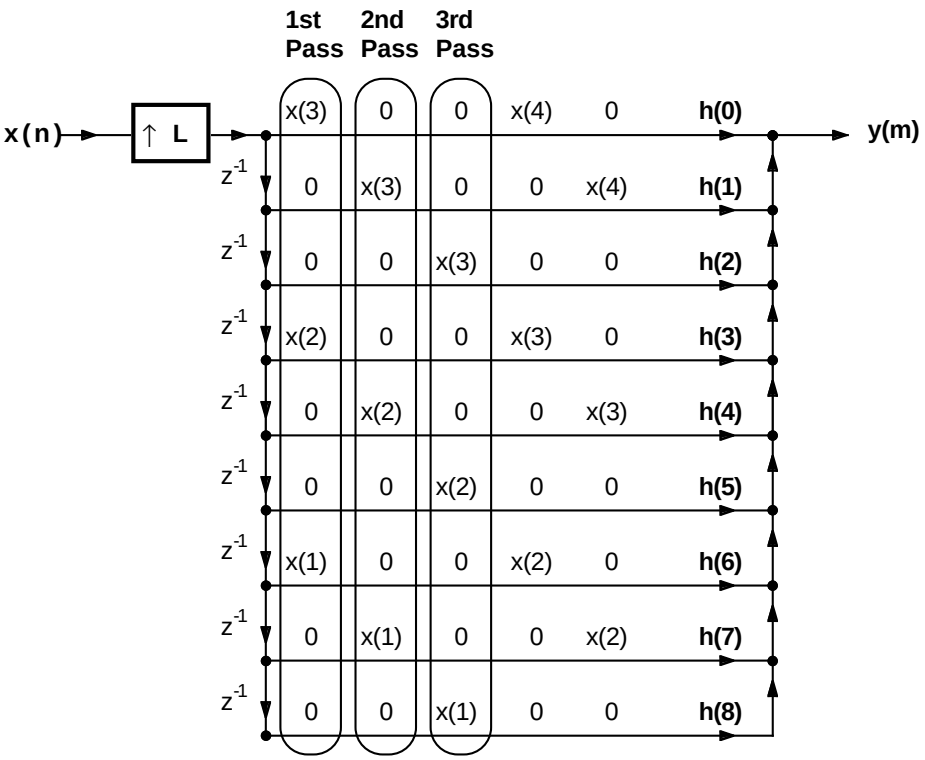


**Figure 5.18 Example Interpolator Flowgraph**

accesses every Lth (third) coefficient, skipping the coefficients that correspond to zero-valued data samples.

Crochiere and Rabiner (see *References* at the end of this chapter) refer to this efficient interpolation filtering method as *polyphase filtering*, because a different phase of the filter function h(k) (equivalent to a set of interleaved coefficients) is used to calculate each output sample.

### *5.5.3.3 ADSP-2100 Interpolation Algorithm*

A circular buffer of length N/L located in data memory forms the data delay line. Although the convolution equation accesses the newest data sample first and the oldest data sample last, the ADSP-2100 fetches data samples from the circular buffer in the opposite order: oldest data first, newest data last. To keep the data/coefficient pairs together, the coefficients are stored in program memory in reverse order, e.g., h(N–1) in PM(0) and h(0) in PM(N–1).

Figure 5.19 shows a flow chart of the interpolation algorithm. The processor waits in a loop and is interrupted at the output sample rate (L times the input sample rate). In the interrupt routine, the coefficient address pointer is decremented by one location so that a new set of interleaved coefficients will be accessed in the next filter pass. A counter tracks the occurrence of every Lth output; on the Lth output, an input sample is taken and the coefficient address pointer is set forward L locations, back to the first set of interleaved coefficients. The output is then calculated with the coefficient address pointer incremented by L locations to fetch every Lth coefficient. One restriction in this algorithm is that the number of filter taps must be an integral multiple of the interpolation factor; N/L must be an integer.

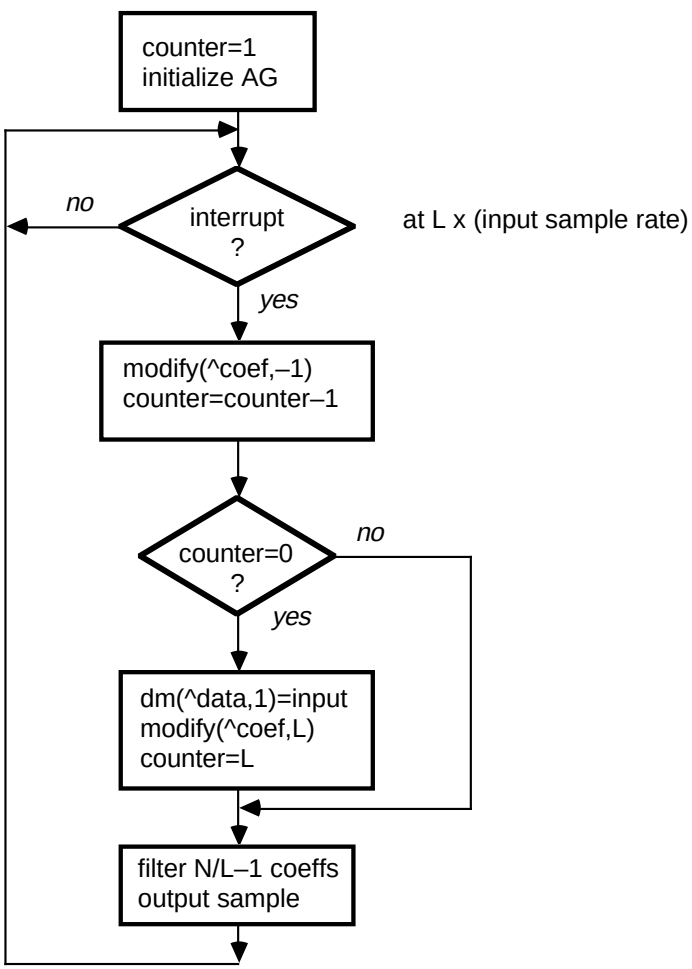


**Figure 5.19 Interpolation Flow Chart**

Listing 5.10 is an ADSP-2100 program that implements this interpolation algorithm. The ADSP-2100 is capable of calculating each filter pass in ((N/L)+17) processor instruction cycles. Each pass must be calculated within the period between output samples, equal to $1/F_sL$. Thus the maximum number of taps that can be calculated in real time is:

$$N = \frac{1}{F_s\, t_{CLK}} - 17L$$

where $t_{CLK}$ is the processor cycle time and $F_s$ is the input sampling rate.

```
{INTERPOLATE.dsp
Real time Direct Form FIR Filter, N taps, uses an efficient algorithm
to interpolate by L for an increase of L times the input sample rate. A
restriction on the number of taps is that N/L be an integer.

   INPUT: adc
   OUTPUT: dac
}

.MODULE/RAM/ABS=0 interpolate;
.CONST            N=300;
.CONST            L=4;                    {interpolate by factor of L}
.CONST            NoverL=75;
.VAR/PM/RAM/CIRC  coef[N];
.VAR/DM/RAM/CIRC  data[NoverL];
.VAR/DM/RAM       counter;
.PORT             adc;
.PORT             dac;
.INIT             coef:<coef.dat>;

        RTI;                        {interrupt 0}
        RTI;                        {interrupt 1}
        RTI;                        {interrupt 2}
        JUMP sample;                {interrupt 3 at (L*input rate)}

initialize:     IMASK=b#0000;       {disable all interrupts}
                ICNTL=b#01111;      {edge sensitive interrupts}
                SI=1;               {set interpolate counter to 1}
                DM(counter)=SI;     {for first data sample}
                I4=^coef;           {setup a circular buffer in PM}
                L4=%coef;
```

*(listing continues on next page)*

```
                M4=L;               {modifier for coef is L}
                M5=-1;              {modifier to shift coef back -1}
                I0=^data;           {setup a circular buffer in DM}
                L0=%data;
                M0=1;
                IMASK=b#1000;       {enable interrupt 3}
wait_interrupt: JUMP wait_interrupt;{infinite wait loop}

{_____________________Interpolate_____________________________________}

sample:     MODIFY(I4,M5);          {shifts coef pointer back by -1}
            AY0=DM(counter);
            AR=AY0-1;               {decrement and update counter}
            DM(counter)=AR;
            IF NE JUMP do_fir;      {test and input if L times}

{________input data sample, code executed at the sample rate__________}

do_input:   AY0=DM(adc);            {input data sample}
            DM(I0,M0)=AY0;          {update delay line with newest}
            MODIFY(I4,M4);          {shifts coef pointer up by L}
            DM(counter)=M4;         {reset counter to L}

{_________filter pass, occurs at L times the input sample rate________}

do_fir:     CNTR=NoverL - 1;        {N/L-1 since round on last tap}
            MR=0, MX0=DM(I0,M0), MY0=PM(I4,M4);
            DO taploop UNTIL CE;    {N/L-1 taps of FIR}
taploop:        MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
            MR=MR+MX0*MY0(RND);     {last tap with round}
            IF MV SAT MR;           {saturate result if overflowed}
            DM(dac)=MR1;            {output sample}
            RTI;
.ENDMOD;
```

**Listing 5.10 Efficient Interpolation Filter**

The interpolation filter has a gain of 1/L in the passband. One method to attain unity gain is to premultiply (offline) all the filter coefficients by L. This method requires the maximum coefficient amplitude to be less than 1/L, otherwise the multiplication overflows the 16-bit coefficient word length. If the maximum coefficient amplitude is not less than 1/L, then you must multiply each output result by 1/L instead. The code in Listing

5.11 performs the 16-by-32 bit multiplication needed for this gain correction. This code replaces the saturation instruction in the interpolation filter program in Listing 3.3. The MY1 register should be initialized to L at the start of the routine, and the last multiply/accumulate of the filter should be performed with (SS) format, not the rounding option. This code multiplies a filter output sample in 1.31 format by the gain L, in 16.0 format, and produces in a 1.15 format corrected output in the SR0 register.

```
MX1= MR1;
MR= MR0*MY1 (UU);
MR0= MR1;
MR1= MR2;
MR= MR+MX1*MY1 (SU);
SR= LSHIFT MR0 BY -1 (LO);
SR= SR OR ASHIFT MR1 BY -1 (HI);
```

**Listing 5.11 Extended Precision Multiply**

#### *5.5.3.4 Interpolator Hardware Configuration*

The I/O hardware required for the interpolation filter is the same as that for the decimation filter with the exception that the interval timer clocks the output D/A converter, and the input A/D converter is clocked by the interval counter signal divided by L. The interval timer interrupts the ADSP-2100 at the output sample rate. This configuration is shown in Figure 5.20, on the following page.

### 5.5.4 Rational Sample Rate Changes

The preceding sections describe processes for decreasing or increasing a signal's sample rate by an integer factor (M for decreasing, L for increasing). In real systems, the integer factor restriction is frequently unacceptable. For instance, if two sequences of different sample rates are analyzed in operations such as cross-correlation, cross-spectrum, and transfer function determination, the sequences must be converted to a common sample rate. However, the two sample rates are not generally related by an integer factor. Another instance in which the integer factor restriction is unacceptable is in transferring data between two storage media that use different sampling rates, such as from a compact disk at 44.1kHz to a digital audio tape at 48.0kHz. The compact disk data must have its sampling rate increased by a factor of 48/44.1.

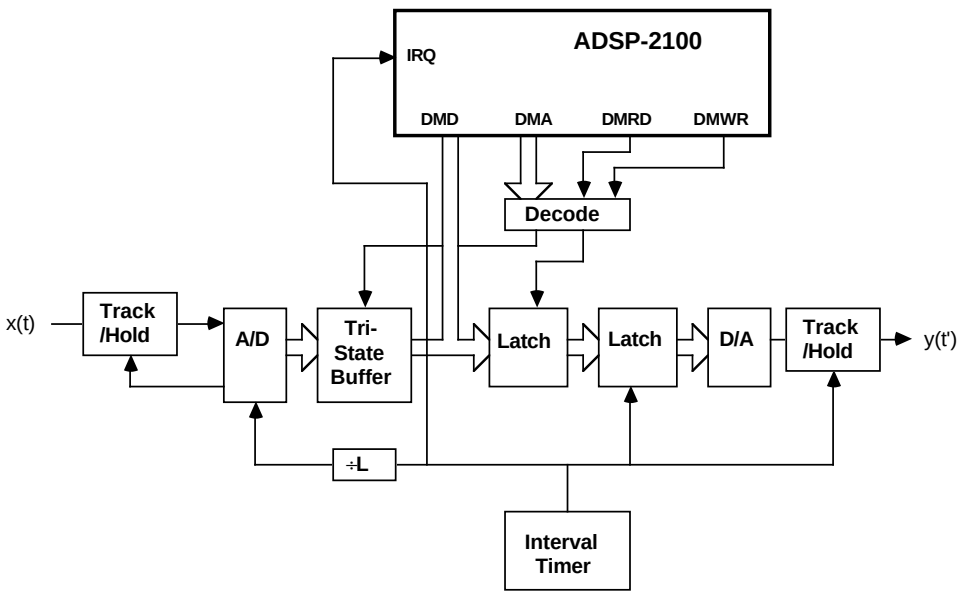


**Figure 5.20 Interpolation Filter Hardware**

### *5.5.4.1 L/M Change in Sample Rate*

A noninteger rate change factor can be approximated by a rational number closest to the desired factor. This number should be the least common multiple of the two sample rates. A rational number is the ratio of two integers and can be expressed as the fraction L/M. For an L/M sample rate change, the signal is first interpolated to increase its sample rate by L. The resulting signal is decimated to decrease its sample rate by M. The overall change in the sample rate is L/M. Thus, a rational rate change can be accomplished by cascading interpolation and decimation filters. For example, to increase a signal's sample rate by a ratio of 48/44.1 requires interpolation by L=160 and decimation by M=147.

Figure 5.21a shows the cascading of the interpolation and decimation processes. The cascaded filters are the interpolation and decimation filters discussed in the two previous sections of this chapter. The rate expander and the low-pass filter h'(k) make up the interpolator, and the low-pass filter h"(k) and the rate compressor make up the decimator. The interpolator has the same restriction that N/L must be an integer. The input signal x(n) is interpolated to an intermediate sample rate that is a common multiple of the input and output sample rates. To maintain the maximum bandwidth in the intermediate signal x(k), the interpolation must precede the decimation; otherwise, some of the desired frequency content in the original signal would be filtered out by the decimator.

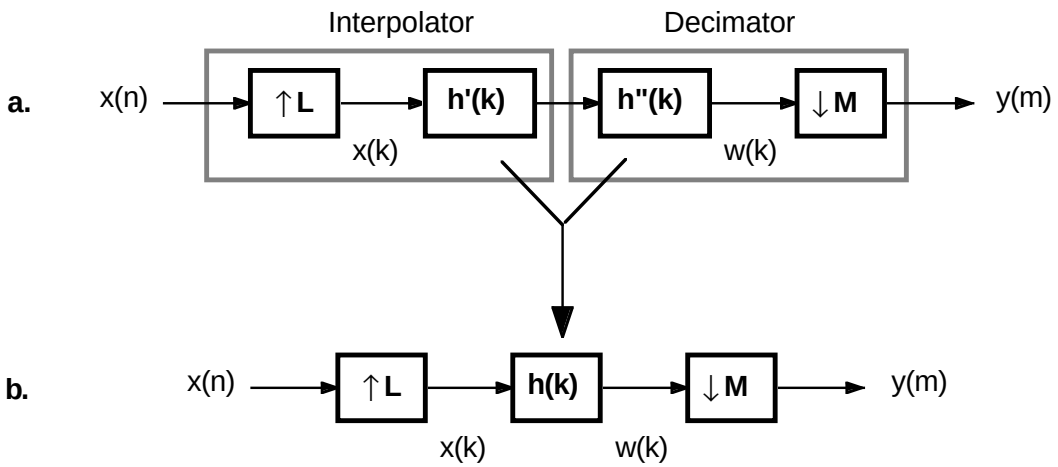


**Figure 5.21 Combined Interpolation and Decimation**

A significant portion of the computations can be eliminated because the two filters, h'(k) and h"(k), are redundant. Both filters have low-pass transfer functions, and thus the filter with the highest cutoff frequency is unnecessary. The interpolation and decimation functions can be combined using one low-pass filter, h(k), as shown in Figure 5.21b. This rate changer incorporates a gain of L to compensate for the 1/L gain of the interpolation filter, as described earlier in this chapter.

Figure 5.22, on the next page, shows the frequency representation for a sample rate increase of 3/2. The input sample frequency of 4kHz is first increased to 12kHz which is the least common multiple of 4 and 6kHz. This intermediate signal X(k) is then filtered to eliminate the images caused by the rate expansion and to prevent any aliasing that the rate compression could cause. The filtered signal W(k) is then rate compressed by a factor of 2 to result in the output Y(k) at a sample rate of 6kHz. Figure 5.23, also on the next page, shows a similar example that decreases the sample rate by a factor of 2/3.

Figure 5.24 shows the relationship between the sample periods used in the 3/2 and 2/3 rate changes. The intermediate sample period is one-Lth of the input period, $1/F_s$, and one-Mth of the output period.

### *5.5.4.2 Implementation of Rate Change Algorithm*

The rational rate change algorithm applies the same calculation-saving techniques used in the decimation and interpolation filters. In the interpolation, the rate expander is incorporated into the filter so that all

# 5 Digital Filters

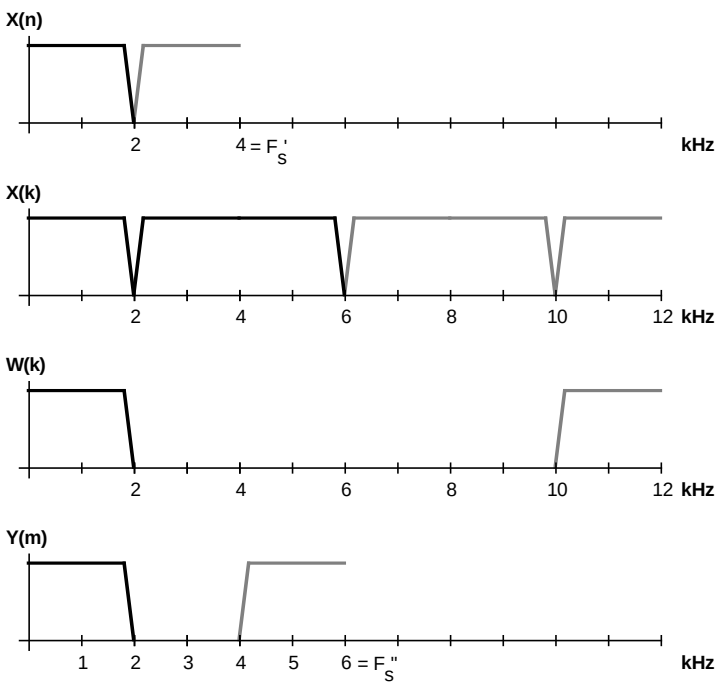


**Figure 5.22 3/2 Sample Rate Change**

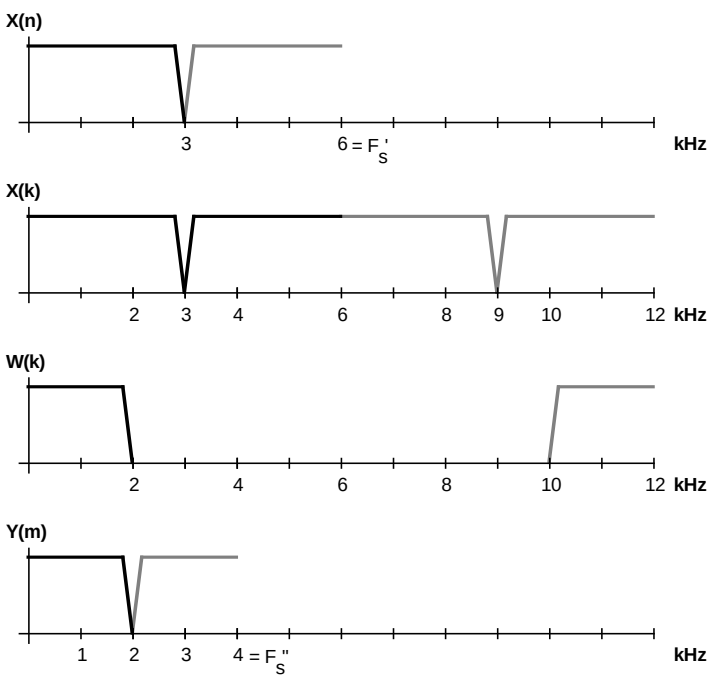


**Figure 5.23 2/3 Sample Rate Change**

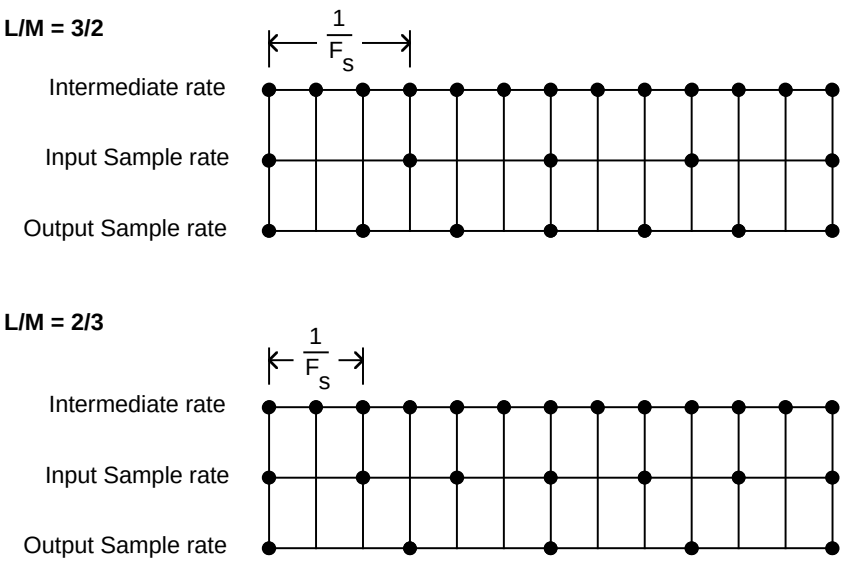


**Figure 5.24 Intermediate, Input and Output Sample Rates**

zero-valued multiplications are skipped. In the decimation, the rate compressor is incorporated into the filter so that discarded results are not calculated, and a data buffer stores input samples that arrive while the filter output is being calculated. Thus, an entire output period is allocated for calculating one output sample.

The flow charts in Figures 5.25 and 5.26 show two implementations of the rate change algorithm. The first one uses software counters to derive the input and output sample rates from the common sample rate. In this algorithm, the main routine is interrupted at the common sample rate. Depending on whether one or both of the counters has decremented to zero, the interrupt routine reads a new data sample into the input buffer and/or sets a flag that causes the main routine to calculate a new output sample.

For some applications, the integer factors M and L are so large that the overhead for dividing the common sample rate leaves too little time for the filter calculations. The second implementation of the rate change algorithm solves this problem by using two external hardware dividers, ÷L and ÷M, to generate the input and output sample rates from the common rate. The ÷L hardware generates an interrupt that causes the processor to input a data sample. The ÷M hardware generates an interrupt (with a lower priority) that starts the calculation of an output sample.

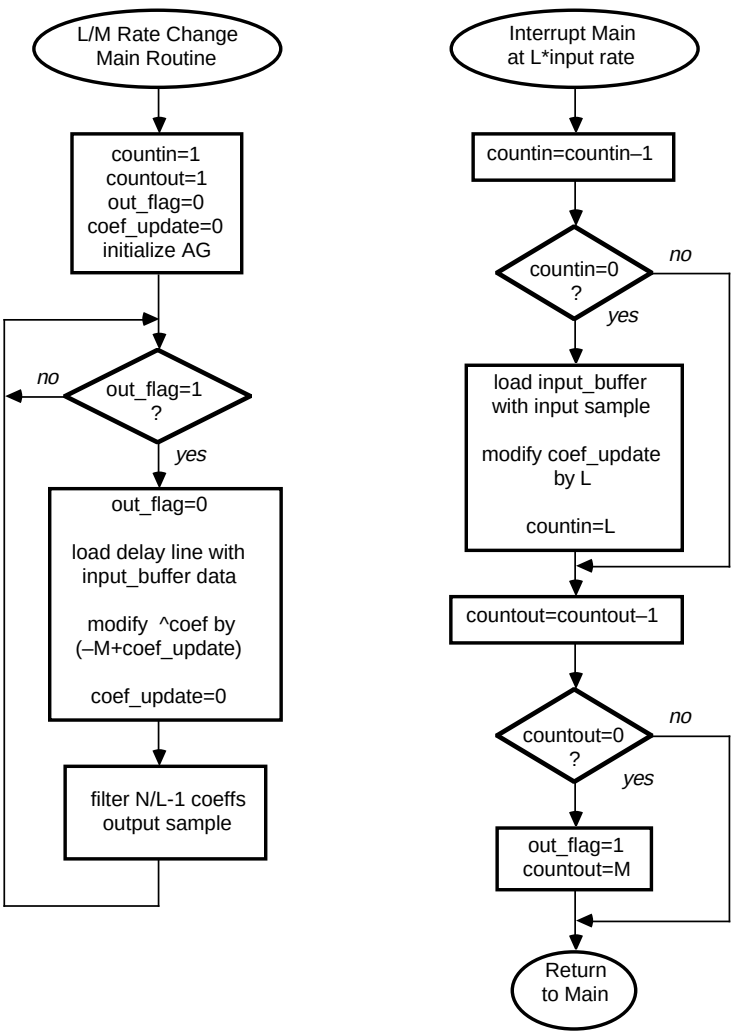


**Figure 5.25 L/M Rate Change with Software Division**

### *5.5.4.3 ADSP-2100 Rational Rate Change Program*

Listings 5.12 and 5.13 contain the ADSP-2100 programs for the two implementations of the rational factor sample rate change discussed above. The programs are identical except that the first uses two counters to derive the input and output sample periods from IRQ1, whereas the second relies on external interrupts IRQ0 and IRQ1 to provide these periods. In the second program, the input routine has the higher priority interrupt so that inputs always precede outputs when both interrupts coincide.

To implement the calculation-saving techniques, the programs must update the coefficient pointer with two different modification values. First, the algorithm must update the coefficient pointer by L each time an input sample is read. This modification moves the coefficient pointer back to the first set of the polyphase coefficients. The variable *coef_update* tracks these updates. The second modification to the coefficient pointer is to set it back by one location for each interpolated output, even the outputs that not calculated because they are discarded in the decimator. The modification constant is –M because M–1 outputs are discarded by the rate compressor. The total value that updates the coefficient pointer is –M + *coef_update*.

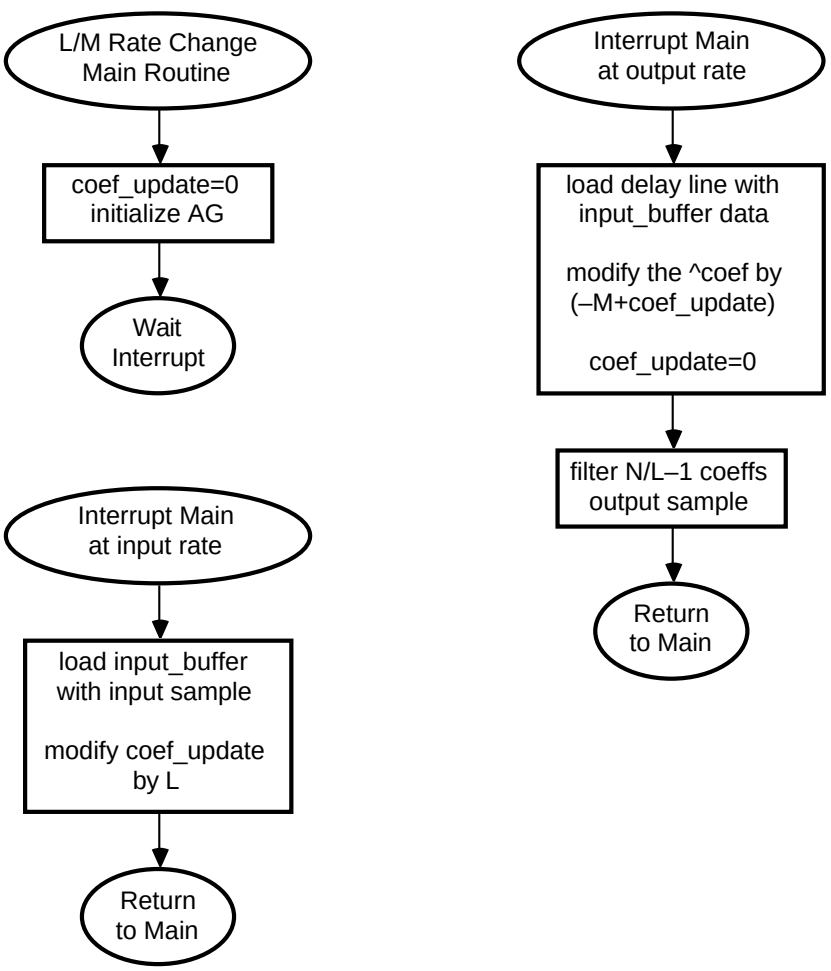


**Figure 5.26 L/M Rate Change with Hardware Division**

# 5 Digital Filters

The rate change program in Listing 5.12 can calculate the following number of filter taps within one output period:

$$N = \frac{M}{F_s\, t_{CLK}} - 30L - 11ML - 6L\lceil M/L \rceil$$

where $t_{CLK}$ is the ADSP-2100 instruction cycle time, and the notation $\lceil u \rceil$ means the smallest integer greater than or equal to $u$. The program in Listing 3.6 can execute

$$N = \frac{M}{F_s\, t_{CLK}} - 22L - 9L\lceil M/L \rceil$$

taps in one output period. These equations determine the upper limit on N, the order of the low-pass filter.

---

```
{RATIO_BUF.dsp
Real time Direct Form FIR Filter, N taps. Efficient algorithm
to interpolate by L and decimate by M for a L/M change in the input sample rate. Uses
an input buffer so that the filter computations to occur in parallel with inputting and
outputting data. This allows a larger number of filter taps for a given input sample
rate.

   INPUT: adc
   OUTPUT: dac
}

.MODULE/RAM/ABS=0 Ratio_eff;
.CONST            N=300;                 {N taps, N coefficients}
.CONST            L=3;                   {decimate by factor of L}
.CONST            NoverL=100;            {NoverL must be an integer}
.CONST            M=2;                   {decimate by factor of M}
.CONST            intMoverL=2;           {smallest integer GE M/L}
.VAR/PM/RAM/CIRC  coef[N];               {coefficient circular buffer}
.VAR/DM/RAM/CIRC  data[NoverL];          {data delay line}
.VAR/DM/RAM       input_buf[intMoverL];  {input buffer is not circular}
.VAR/DM/RAM       countin;
.VAR/DM/RAM       countout;
.VAR/DM/RAM       out_flag;              {true when time to calc. output}
.PORT             adc;
.PORT             dac;
.INIT             coef:<coef.dat>;
```

```
                RTI;              {interrupt 0}
                JUMP interrupt;   {interrupt 1= L * input rate}
                RTI;              {interrupt 2}
                RTI;              {interrupt 3}

initialize:   IMASK=b#0000;       {disable all interrupts}
              ICNTL=b#01111;      {edge sensitive interrupts}
              SI=0;               {variables initial conditions}
              DM(out_flag)=SI;    {true if time to calc. output}
              SI=1;
              DM(countin)=SI;     {input every L interrupts}
              DM(countout)=SI;    {output every M interrupts}
              I4=^coef;           {setup a circular buffer in PM}
              L4=%coef;
              M4=L;               {modifier for coef buffer}
              I5=0;               {i5 tracks coefficient updates}
              L5=0;
              M5=-M;              {coef modify done each output}
              I0=^data;           {setup delay line in DM}
              L0=%data;
              M0=1;               {modifier for data is 1}
              I1=^input_buf;      {setup input data buffer in DM}
              L1=0;               {input buffer is not circular}
              IMASK=b#0010;       {enable interrupt 3}

{___________wait for time to calculate an output sample_____________}
wait_out:     AX0=DM(out_flag);
              AR=PASS AX0;        {test if true}
              IF EQ JUMP wait_out;

{____________code below occurs at the output sample rate____________}
              DM(out_flag)=L5;    {reset the output flag to 0}
              AX0=I1;             {calculate ammount in in buffer}
              AY0=^input_buf;
              AR=AX0-AY0;
              IF EQ JUMP modify_coef;     {skip do loop if buffer empty}
              CNTR=AR;            {dump in buffer into delay line}
```

*(listing continues on next page)*

```
               I1=^input_buf;
               DO load_data UNTIL CE;
                  AR=DM(I1,M0);
load_data:        DM(I0,M0)=AR;
               I1=^input_buf;          {fix pointer to input_buf}

modify_coef:   MODIFY(I5,M5);          {modify coef update by -M}
               M6=I5;
               MODIFY(I4,M6);          {modify ^coef by coef update}
               I5=0;                   {reset coef update}

fir:           CNTR=NoverL-1;
               MR=0, MX0=DM(I0,M0), MY0=PM(I4,M4);
               DO taploop UNTIL CE;    {N/L-1 taps of FIR}
taploop:          MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
               MR=MR+MX0*MY0(RND);     {last tap with round}
               IF MV SAT MR;           {saturate result if overflow}
               DM(dac)=MR1;            {output data sample}
               JUMP wait_out;

{___________interrupt, code executed at L times the input rate_________}
interrupt:     ENA SEC_REG;            {so no registers will get lost}
               AY0=DM(countin);        {test if time for input}
               AR=AY0-1;
               DM(countin)=AR;
               IF NE JUMP skipin;

input:         AY0=DM(adc);            {get input sample}
               DM(I1,M0)=AY0;          {load in M long buffer}
               MODIFY(I5,M4);          {modify the coef update by L}
               DM(countin)=M4;         {reset the input count to L}

skipin:        AY0=DM(countout);       {test if time for output}
               AR=AY0-1;
               DM(countout)=AR;
               IF NE RTI;

output:        DM(out_flag)=M0;        {set output flag to true 1}
               AR=M;                   {reset output counter to M}
               DM(countout)=AR;
               RTI;
.ENDMOD;
```

**Listing 5.12 Rational Rate Change Program with Software Division**

```
{RATIO_2INT.dsp
Real time Direct Form FIR Filter, N taps.  Efficient algorithm to interpolate by L and
decimate by M for a L/M change in the input sample rate. Uses an input buffer so that
the filter computations to occur in parallel with inputting and outputting data. This
allows a larger number of filter taps for a given input sample rate. This version uses
two interrupts and external divide by L and divide by M to eliminate excessive overhead
for large values of M and L.

   INPUT: adc
   OUTPUT: dac
}

.MODULE/RAM/ABS=0 Ratio_2int;
.CONST            N=300;                 {N taps, N coefficients}
.CONST            L=3;                   {decimate by factor of L}
.CONST            NoverL=100;            {NoverL must be an integer}
.CONST            M=2;                   {decimate by factor of M}
.CONST            intMoverL=2;           {smallest integer GE M/L}
.VAR/PM/RAM/CIRC  coef[N];               {coefficient circular buffer}
.VAR/DM/RAM/CIRC  data[NoverL];          {data delay line}
.VAR/DM/RAM       input_buf[intMoverL];  {input buffer is not circular}
.PORT             adc;
.PORT             dac;
.INIT             coef:<coef.dat>;

        JUMP output;               {interrupt 0= L*Fin/M}
        JUMP input;                {interrupt 1= L*Fin/L}
        RTI;                       {interrupt 2}
        RTI;                       {interrupt 3}

initialize: IMASK=b#0000;          {disable all interrupts}
            ICNTL=b#11111;         {edge sens. nested interrupts}
            I4=^coef;              {setup a circular buffer in PM}
            L4=%coef;
            M4=L;                  {modifier for coef buffer}
            I5=0;                  {i5 tracks coefficient updates}
            L5=0;
            M5=-M;                 {coef modify done each output}
            I0=^data;              {setup delay line in DM}
            L0=%data;
            M0=1;                  {modifier for data is 1}
            I1=^input_buf;         {setup input data buffer in DM}
            L1=0;                  {input buffer is not circular}
            IMASK=b#0011;          {enable interrupt 3}
```

*(listing continues on next page)*

```
{___________wait for time to output or input a sample_________________}
wait_out:    JUMP wait_out;

{______interrupt code below executed at the output sample rate________}
output:      AX0=I1;                {calculate ammount in in buffer}

             AY0=^input_buf;
             AR=AX0-AY0;
             IF EQ JUMP modify_coef;{skip do loop if buffer empty}
             CNTR=AR;               {dump in buffer into delay line}

             I1=^input_buf;
             DO load_data UNTIL CE;
                AR=DM(I1,M0);
load_data:      DM(I0,M0)=AR;
             I1=^input_buf;         {fix pointer to input_buf}
modify_coef: MODIFY(I5,M5);         {modify coef update by -M}
             M6=I5;
             MODIFY(I4,M6);         {modify ^coef by coef update}
             I5=0;                  {reset coef update}

fir:         CNTR=NoverL-1;
             MR=0, MX0=DM(I0,M0), MY0=PM(I4,M4);
             DO taploop UNTIL CE;   {N/L-1 taps of FIR}
taploop:        MR=MR+MX0*MY0(ss), MX0=DM(I0,M0), MY0=PM(I4,M4);
             MR=MR+MX0*MY0(RND);    {last tap with round}
             IF MV SAT MR;          {saturate result if overflow}
             DM(dac)=MR1;           {output data sample}
             RTI;

{_______interrupt code below executed at the input sample rate_________}

input:       ENA SEC_REG;           {context save}
             AY0=DM(adc);           {get input sample}
             DM(I1,M0)=AY0;         {load in M long buffer}
             MODIFY(I5,M4);         {modify the coef update by L}
             RTI;
.ENDMOD;
```

**Listing 5.13 Rational Rate Change Program with Hardware Division**

### 5.5.5 Multistage Implementations

The previous examples of sample rate conversion in this chapter use a single low-pass filter to prevent the aliasing or imaging associated with the rate compression and expansion. One method for further improving the efficiency of rate conversion is to divide this filter into two or more cascaded stages of decimation or interpolation. Each successive stage reduces the sample rate until the final sample rate is equal to twice the bandwidth of the desired data. The product of all the stages' rate change factors should equal the total desired rate change, M or L.

Crochiere and Rabiner (see *References* at the end of this chapter) show that the total number of computations in a multi-stage design can be made substantially less than that for a single-stage design because each stage has a wider transition band than that of the single-stage filter. The sample rate at which the first stage is calculated may be large, but because the transition band is wide, only a small number of filter taps (N) is required. In the last stage, the transition band may be small, but because the sample rate is small also, fewer taps and thus a reduced number of computations are needed. In addition to computational efficiency, multistage filters have the advantage of a lower round-off noise due to the reduced number of taps.

Figure 5.27, on the following page, shows the frequency spectra for an example decimation filter implemented in two stages. The bandwidth and transition band of the desired filter is shown in Figure 5.27a and the frequency response of the analog anti-alias filter required is shown in Figure 5.27b. The shaded lines indicate the frequencies that alias into the interstage transition bands. These frequencies are sufficiently attenuated so as not to disturb the final pass or transition bands. The frequency responses of the first and final stage filters are shown in Figure 5.27c and d. The example in Figure 5.28 is the same except that aliasing is allowed in the final transition band. This aliasing is tolerable, for instance, when the resulting signal is used for spectral analysis and the aliased band can be ignored. All the transition bands are wide and therefore the filter stages require fewer taps than a single-stage filter.

Crochiere and Rabiner (see *References* at the end of this chapter) contains some design curves that help to determine optimal rate change factors for the intermediate stages. A two- or three-stage design provides a substantial reduction in filter computations; further reductions in computations come from adding more stages. Because the filters are cascaded, each stage must have a passband ripple equal to final passband

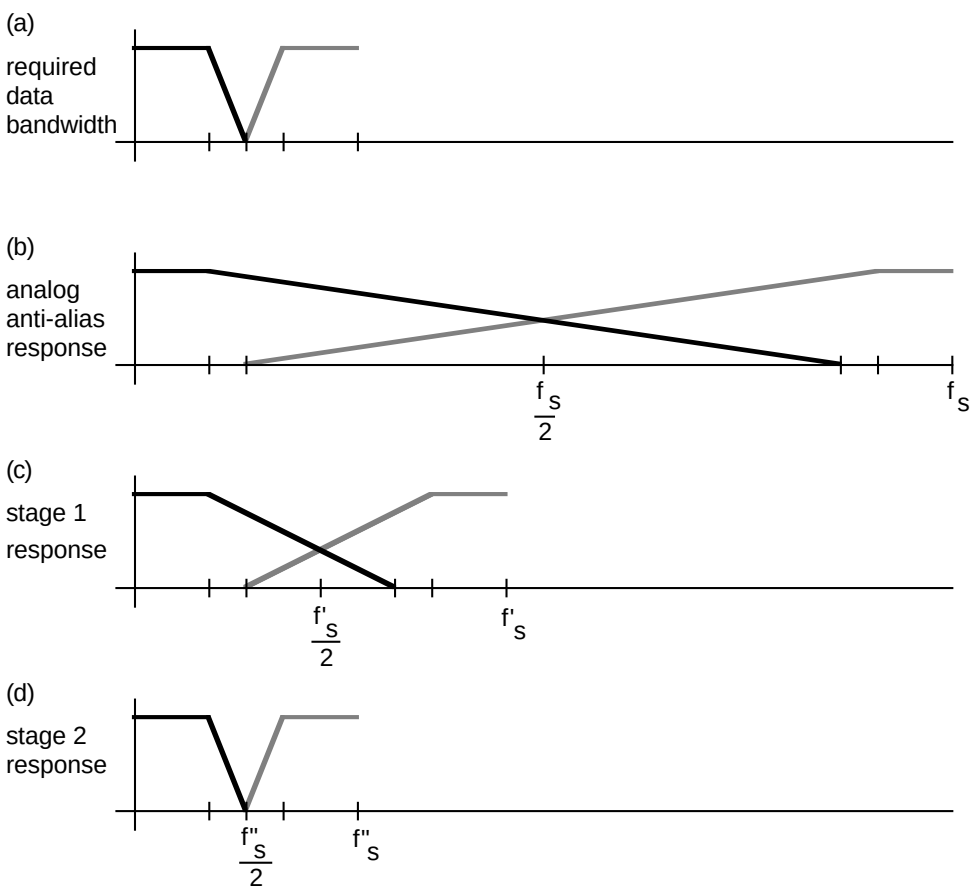


**Figure 5.27 Two-Stage Decimation with No Transition Band Aliasing**

ripple divided by the number of stages used. The stopband ripple does not have to be less than the final stopband ripple because each successive stage attenuates the stopband ripple of the previous stage.

Listing 5.14 is the ADSP-2100 program for a two-stage decimation filter. This two-stage decimation program is similar to the program for the buffered decimation listed in *Decimation*, earlier in this chapter. The main difference is that two *ping-ponged* buffers are used to store input samples. While one buffer is filled with input data, the other is dumped into the delay line of the first filter stage. The result of the first stage filter is written to the second stage's delay line. After M samples have been input and one final result has been calculated, the two input buffers swap functions, or *ping-pong*.

Two buffers are needed because only a portion (M–1 samples) of the input buffer can be dumped into the first stage delay line at once. The single-stage decimation algorithm used only one buffer because it could dump all input data into the buffer at once. The two ping-ponged buffers are implemented as one double-length circular buffer, 2M locations long, indexed by two pointers offset from each other by M. Because the pointers follow each other and wrap around the buffer, the two halves switch between reading and writing (ping-pong) after every M inputs.

Listing 5.15 is the ADSP-2100 program for a two-stage interpolation filter. This routine is essentially a cascade of the program for the single-stage interpolator. The required interrupt rate is the product of the interpolation factors of the individual stages, (L–1)(L–2). The program can be expanded easily if more than two stages of interpolation or decimation are required.

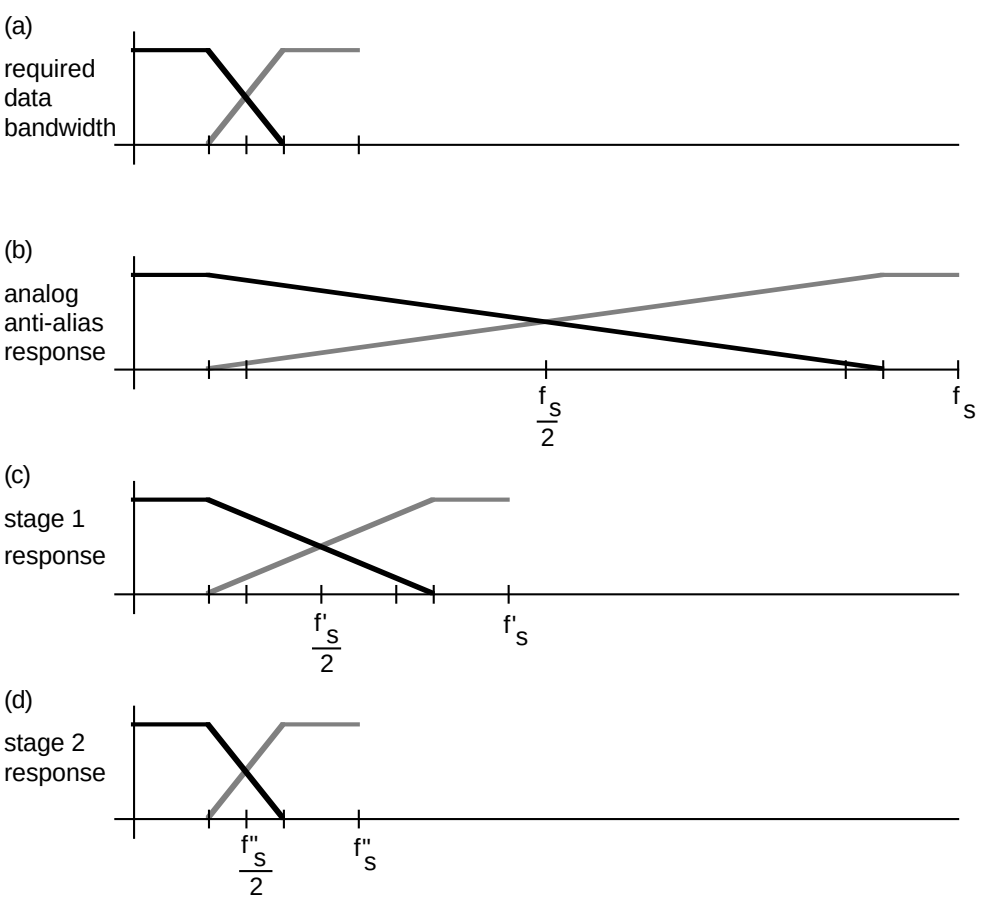


**Figure 5.28 Two-Stage Decimation with Transition Band Aliasing**

```
{DEC2STAGEBUF.dsp
Real time Direct Form Finite Impulse Filter, N1,N2 taps, uses two cascaded stages to
decimate by M1 and M2 for a decrease of 1/(M1*M2) times the input sample rate. Uses an
input buffer to increase efficiency.

   INPUT: adc
   OUTPUT: dac
}

.MODULE/RAM/ABS=0 dec2stagebuf;
.CONST            N1=32;
.CONST            N2=148;
.CONST            M_1=2;                  {decimate by factor of M}
.CONST            M_2=2;                  {decimate by factor of M}
.CONST            M1xM2=4;                {(M_1 * M_2)}
.CONST            M1xM2x2=8;              {(M_1 * M_2 * 2)}
.VAR/PM/RAM/CIRC  coef1[N1];
.VAR/PM/RAM/CIRC  coef2[N2];
.VAR/DM/RAM/CIRC  data1[N1];
.VAR/DM/RAM/CIRC  data2[N2];
.VAR/DM/RAM/CIRC  input_buf[m1xm2x2];
.VAR/DM/RAM       input_count;
.PORT             adc;
.PORT             dac;
.INIT             coef1:<coef1.dat>;
.INIT             coef2:<coef2.dat>;

        RTI;                        {interrupt 0}
        RTI;                        {interrupt 1}
        RTI;                        {interrupt 2}
        JUMP sample;                {interrupt 3 at input rate}

initialize: IMASK=b#0000;           {disable all interrupts}
            ICNTL=b#01111;          {edge sensitive interrupts}
            SI=M1xM2;
            DM(input_count)=SI;
            I4=^coef1;              {setup a circular buffer in PM}
            L4=%coef1;
            M4=1;                   {modifier for coef is 1}
            I5=^coef2;              {setup a circular buffer in PM}
            L5=%coef2;
```

```
            I0=^data1;              {setup a circular buffer in DM}
            L0=%data1;
            M0=1;                   {modifier for data is 1}
            I1=^data2;              {setup a circular buffer in DM}
            L1=%data2;
            I2=^input_buf;          {setup input buffer in DM}
            L2=%input_buf;
            I3=^input_buf;          {setup second in buffer in DM}
            L3=%input_buf;
            IMASK=b#1000;           {enable interrupt 3}

wait_full:  AX0=DM(input_count);    {wait until input buffer is full}
            AR=PASS AX0;
            IF NE JUMP wait_full;

            AR=M1xM2;               {reinitialize input counter}
            DM(input_count)=AR;
            CNTR=M_2;
            DO stage_1 UNTIL CE;
               CNTR=M_1;
               DO dump_buf UNTIL CE;
                  AR=DM(I3,M0);
dump_buf:         DM(I0,M0)=AR;
               CNTR=N1 - 1;
               MR=0, MX0=DM(I0,M0), MY0=PM(I4,M4);
               DO taploop1 UNTIL CE;{N-1 taps of FIR}
taploop1:         MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
               MR=MR+MX0*MY0(RND);  {last tap with round}
               IF MV SAT MR;        {saturate result if overflow}
stage_1:       DM(I1,M0)=MR1;       {pass to next filter stage}

            CNTR=N2 - 1;
            MR=0, MX0=DM(I1,M0), MY0=PM(I5,M4);
            DO taploop2 UNTIL CE;   {N-1 taps of FIR}
taploop2:      MR=MR+MX0*MY0(SS), MX0=DM(I1,M0), MY0=PM(I5,M4);
            MR=MR+MX0*MY0(RND);     {last tap with round}
            IF MV SAT MR;           {saturate result if overflow}
            DM(dac)=MR1;            {output data sample}
            JUMP wait_full;
```

*(listing continues on next page)*

```
{_____________interrupt code executed at the sample rate_____________}
sample:     ENA SEC_REG;            {context save}
            AY0=DM(adc);            {input data sample}
            DM(I2,M0)=AY0;          {load input buffer}
            AY0=DM(input_count);
            AR=AY0-1;               {decrement and update counter}
            DM(input_count)=AR;
            RTI;
.ENDMOD;
```

**Listing 5.14 ADSP-2100 Program for Two-Stage Decimation**

```
{INT2STAGE.dsp
Two stage cascaded real time Direct Form Finite Impulse Filter, N1, N2 taps, uses an
efficient algorithm to interpolate by L1*L2 for an increase of L1*L2 times the input
sample rate. A restriction on the number of taps is that N divided by L be an integer.

   INPUT: adc
   OUTPUT: dac
}

.MODULE/RAM/ABS=0 interpolate_2stage;
.CONST            N1=32;
.CONST            N2=148;
.CONST            L_1=2;                {stage one factor is L1}
.CONST            L_2=2;                {stage two factor is L2}
.CONST            N1overL1=16;
.CONST            N2overL2=74;
.VAR/PM/RAM/CIRC  coef1[N1];
.VAR/PM/RAM/CIRC  coef2[N2];
.VAR/DM/RAM/CIRC  data1[N1overL1];
.VAR/DM/RAM/CIRC  data2[N2overL2];
.VAR/DM/RAM       counter1;
.VAR/DM/RAM       counter2;
```

```
.PORT             adc;
.PORT             dac;
.INIT             coef1:<coef1.dat>;
.INIT             coef2:<coef2.dat>;

                  RTI;              {interrupt 0}
                  RTI;              {interrupt 1}
                  RTI;              {interrupt 1}
                  JUMP sample;      {interrupt 3= L1*L2 output rate}

initialize:     IMASK=b#0000;       {disable all interrupts}
                ICNTL=b#01111;      {edge sensitive interrupts}
                SI=1;               {set interpolate counters to 1}
                DM(counter1)=SI;    {for first data sample}
                DM(counter2)=SI;
                I4=^coef1;          {setup a circular buffer in PM}
                L4=%coef1;
                M4=L_1;             {modifier for coef is L1}
                M6=-1;              {modifier to shift coef back -1}

                I5=^coef2;          {setup a circular buffer in PM}
                L5=%coef2;
                M5=L_2;             {modifier for coef is L2}
                I0=^data1;          {setup a circular buffer in DM}
                L0=%data1;
                M0=1;
                I1=^data2;          {setup a circular buffer in DM}
                L1=%data2;
                IMASK=b#1000;       {enable interrupt 3}
wait_interrupt: JUMP wait_interrupt;      {infinite wait loop}


{_____________________Interpolate_____________________________________}
sample:     MODIFY(I5,M6);          {shifts coef pointer back by -1}

            AY0=DM(counter2);
            AR=AY0-1;               {decrement and update counter}
            DM(counter2)=AR;
            IF NE JUMP do_fir2;     {test, do stage 1 if L_2 times}
```

*(listing continues on next page)*

```
            MODIFY(I4,M6);          {shifts coef pointer back by -1}
            AY0=DM(counter1);
            AR=AY0-1;               {decrement and update counter.}
            DM(counter1)=AR;
            IF NE JUMP do_fir1;     {test and input if L_1 times}

{_________input data sample, code occurs at the input sample rate____}
do_input:   AY0=DM(adc);            {input data sample}
            DM(I0,M0)=AY0;          {update delay line with newest}
            MODIFY(I4,M4);          {shifts coef1 pointer by L1}
            DM(counter1)=M4;        {reset counter1}

{_________filter pass, occurs at L1 times the input sample rate_______}
do_fir1:    CNTR=N1overL1 - 1;      {N1/L1-1 because round last tap}

            MR=0, MX0=DM(I0,M0), MY0=PM(I4,M4);
            DO taploop1 UNTIL CE;   {N1/L_1-1 taps of FIR}
taploop1:      MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
            MR=MR+MX0*MY0(RND);     {last tap with round}
            IF MV SAT MR;           {saturate result if overflow}
            DM(I1,M0)=MR1;          {update delay line with newest}
            MODIFY(I5,M5);          {shifts coef2 pointer by L2}
            DM(counter2)=M5;        {reset counter2}

{_______filter pass, executed at (L1*L2) times the input sample rate__}
do_fir2:    CNTR=N2overL2 - 1;      {N2/L2-1 because round last tap}

            MR=0, MX0=DM(I1,M0), MY0=PM(I5,M5);
            DO taploop2 UNTIL CE;   {N2/L_2-1 taps of FIR}
taploop2:      MR=MR+MX0*MY0(SS), MX0=DM(I1,M0), MY0=PM(I5,M5);
            MR=MR+MX0*MY0(RND);     {last tap with round}
            IF MV SAT MR;           {saturate result if overflow}
            DM(dac)=MR1;            {output sample}
            RTI;
.ENDMOD;
```

**Listing 5.15 ADSP-2100 Program for Two-Stage Interpolation**

### 5.5.6 Narrow-Band Spectral Analysis

The computation of the spectrum of a signal is a fundamental DSP operation that has a wide range of applications. The spectrum can be calculated efficiently with the fast Fourier transform (FFT) algorithm. An N-point FFT results in N/2 bins of spectral information spanning zero to the Nyquist frequency. The frequency resolution of this spectrum is $F_s/N$ Hz per bin, where $F_s$ is the sample rate of the data. The number computations required is on the order of $N\log_2 N$. Often, applications such as sonar, radar, and vibration analysis need to determine only a narrow band of the entire spectrum of a signal. The FFT would require calculating the entire spectrum of the signal and discarding the unwanted frequency components.

Multirate filtering techniques let you translate a frequency band to the baseband and reduce the sample rate to twice the width of the narrow band. An FFT performed on reduced-sample-rate data allows either greater resolution for about the same amount of computations or an equivalent resolution for a reduced amount of computations. Thus, the narrow band can be calculated more efficiently. In addition to computation savings, this frequency translation technique has the advantage of eliminating the problem of an increased noise floor caused by the bit growth of data in a large-N FFT.

One method of frequency translation takes advantage of the aliasing properties of the rate compressor. As discussed in *Decimation*, earlier in this chapter, rate compression (discrete-time sampling) in the frequency domain results in images, a sequence of periodic repetitions of the baseband signal. These images are spaced at harmonics of the sampling frequency.

The modulation and the sample rate reduction can be done simultaneously, as shown in Figure 5.29a, on the next page. The input signal is band-pass-filtered to eliminate all frequencies but the narrow band of interest (between $w_1$ and $w_2$). This signal is rate-compressed by a factor of M to get the decimated and frequency-translated output. An (N/M)-point FFT is performed on the resulting signal, producing the spectrum of the signal, which now contains only the narrow band.

Figure 5.29b shows the modulation of the band-pass signal. The modulating impulses are spaced at intervals of $2\pi/M$. Figure 5.29c shows that the narrow band is translated to baseband because it is forced to alias. The spectrum of the final signal y(m) is shown in Figure 5.29d. Zero corresponds to $w_1$, and $\pi$ or the Nyquist frequency corresponds to $w_2$. If an

# 5 Digital Filters

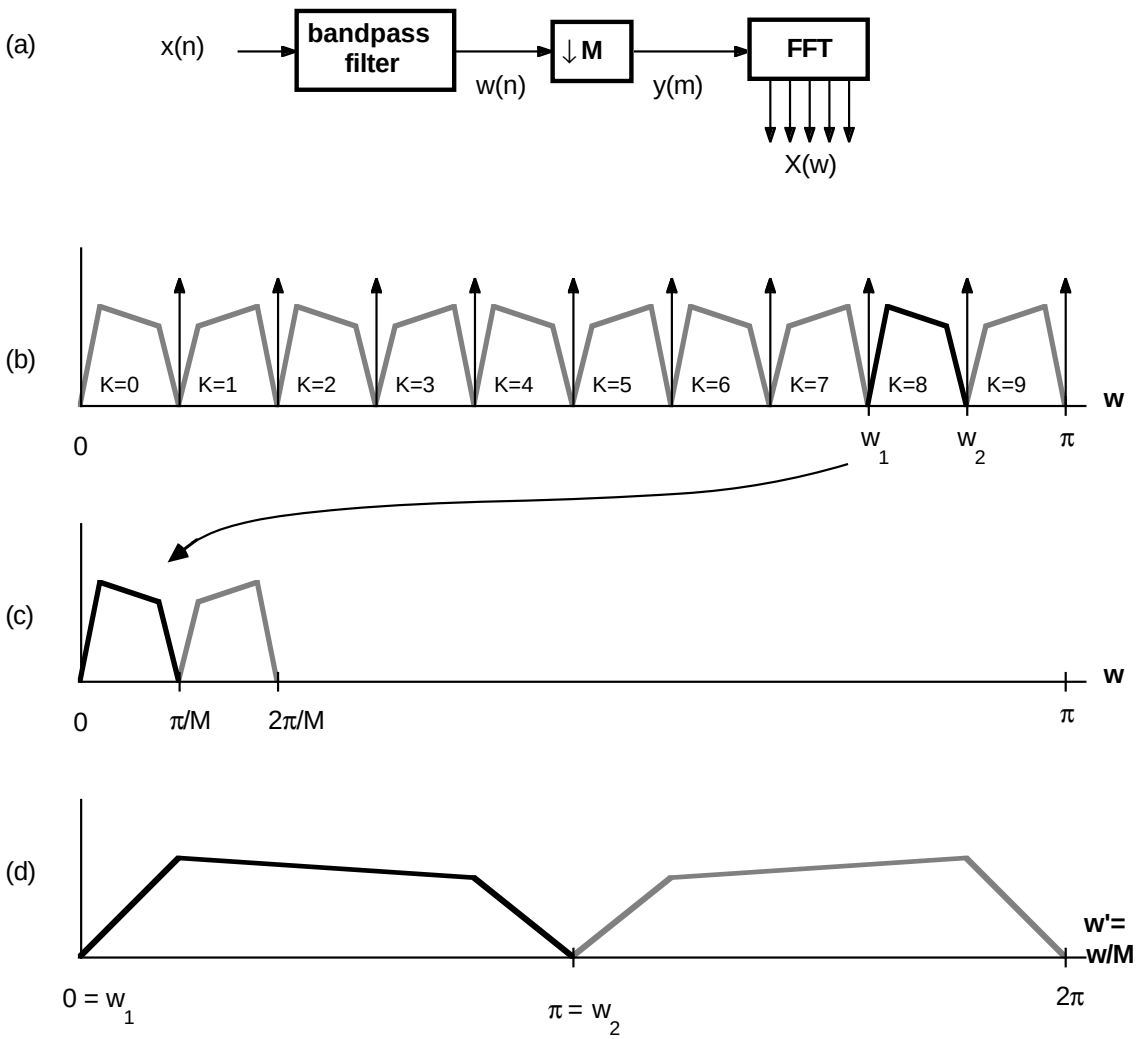


**Figure 5.29 Integer Band Decimator for High Resolution Spectral Analysis**

odd integer band is chosen for the band-pass filter, e.g., K=9, then the translated signal is inverted in frequency. This situation can be corrected by multiplying each output sample y(m) by $(-1)^m$, i.e., inverting the sign of all odd samples.

The entire narrow band filtering process can be accomplished using the single- or multi-stage decimation program listed in this chapter. A listing of an ADSP-2100 FFT implementation can be found in Chapter 6.

## 5.6 ADAPTIVE FILTERS

The stochastic gradient (SG) adaptive filtering algorithm, developed in its present form by Widrow and Hoff (Widrow and Hoff, 1960), provides a powerful and computationally efficient means of realizing adaptive filters. It is used to accomplish a variety of applications, including

- echo cancellation in voice or data signals,
- channel equalization in data communication to minimize the effect of intersymbol interference, and,
- noise cancellation in speech, audio, and biomedical signal processing.

The SG algorithm is the most commonly used adaptation algorithm for transversal filter structures. Honig and Messerschmitt, 1984, provides an excellent and thorough treatment of the SG transversal filter as well as other algorithms and filter structures. Using the notation developed therein, the estimation error $e_c(T)$ between the two input signals y(T) and d(T) of a joint process estimator implemented with a transversal filter structure is given by the following equation:

$$e_c(T) = d(T) - \sum_{j=1}^{n} c_j(T)\, y(T - j + 1)$$

The estimation error of the joint process estimator is thus formed by the difference between the signal it is desired to estimate, d(T), and a weighted linear combination of the current and past input values y(T). The weights, $c_j(T)$, are the transversal filter coefficients at time T. The adaptation of the jth coefficient, $c_j(T)$, is performed according to the following equation:

$$c_j(T + 1) = c_j(T) + \beta e_c(T)\, y(T - j + 1)$$

In this equation, y(T–j+1) represents the past value of the input signal "contained" in the jth tap of the transversal filter. For example, y(T), the present value of the input signal, corresponds to the first tap and y(T–42) corresponds to the forty-third filter tap. The step size β controls the "gain" of the adaptation and allows a tradeoff between the convergence rate of the algorithm and the amount of random fluctuation of the coefficients after convergence.

### 5.6.1 Single-Precision Stochastic Gradient

The transversal filter subroutine that performs the sum-of-products operation to calculate $e_c(T)$, the estimation error, is given in *FIR Filters*, earlier in this chapter. The subroutine that updates the filter coefficients is shown in Listing 5.16. This subroutine is based on the assumption that all *n* data values used to calculate the coefficient are real.

The first instruction multiplies $e_c(T)$ (the estimation error at time T, stored in MX0) by $\beta$ (the step size, stored in MY1) and loads the product into the MF register. In parallel with this multiplication, the data memory read which transfers y(T–n+1) (pointed to by I2) to MX0 is performed. The nth coefficient update value, $\beta e_c(T)$ y(T–n+1), is computed by the next instruction in parallel with the program memory read which transfers the nth coefficient (pointed to by I6) to the ALU input register AY0. The *adapt* loop is then executed *n* times in 2n+2 cycles to update all *n* coefficients. The first time through the loop, the nth coefficient is updated in parallel with a dual fetch of y(T–n+2) and the (n–1)th coefficient. The updated nth coefficient value is then written back to program memory in parallel with the computation of the (n–1)th coefficient update value, $e_c(T)$ y(T–n+2). This continues until all *n* coefficients have been updated and execution falls out of the loop. If desired, the saturation mode of the ALU may be enabled prior to entering the routine so as to automatically implement a saturation capability on the coefficient updates.

The maximum allowable filter order when using the stochastic gradient algorithm for an adaptive filtering application is determined primarily by the processor cycle time, the sampling rate, and the number of other computations required. The transversal filter subroutine requires a total of n+7 cycles for a filter of length *n*, while the gradient update subroutine requires 2n+9 cycles to update all *n* coefficients. At an 8-kHz sampling rate and an instruction cycle time of 125 nanoseconds, the ADSP-2100 can implement an SG transversal filter of approximately 300 taps. This implementation would also have 84 instruction cycles for miscellaneous operations.

```
.MODULE rsg_sub;

{
   Real SG Coefficient Adaptation Subroutine

   Calling Parameters
        MX0 = Error
        MY1 = Beta
        I2 –> Oldest input data value in delay line
        L2 = Filter length
        I6 –> Beginning of filter coefficient table
        L6 = Filter length
        M1,M5 = 1
        M6 = 2
        M3,M7 = –1
        CNTR = Filter length

   Return Values
        Coefficients updated
        I2 –> Oldest input data value in delay line
        I6 –> Beginning of filter coefficient table

   Altered Registers
        AY0,AR,MX0,MF,MR

   Computation Time
        (2 × Filter length) + 6 + 3 cycles

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

.ENTRY  rsg;

rsg:    MF=MX0*MY1(RND), MX0=DM(I2,M1);         {MF=Error × Beta}
        MR=MX0*MF(RND), AY0=PM(I6,M5);
        DO adapt UNTIL CE;
           AR=MR1+AY0, MX0=DM(I2,M1), AY0=PM(I6,M7);
adapt:     PM(I6,M6)=AR, MR=MX0*MF(RND);
        MODIFY (I2,M3);                         {Point to oldest data}
        MODIFY (I6,M7);                         {Point to start of table}
        RTS;
.ENDMOD;
```

**Listing 5.16 Single-Precision Stochastic Gradient**

### 5.6.2 Double-Precision Stochastic Gradient

In some adaptive filtering applications, such as the local echo cancellation required in high-speed data transmission systems, the precision afforded by 16-bit filter coefficients is not adequate. In such applications it is desirable to perform the coefficient adaptation (and generally the filtering operation as well) using a higher-precision representation for the coefficient values. The subroutine in Listing 5.17 implements a stochastic gradient adaptation algorithm that is again based on the equations in the previous section but performs the coefficient adaptation in double precision. Data values, of course, are still maintained in single precision.

The 16-bit coefficients are stored in program memory LSW first, so that the LSWs of all coefficients are stored at even addresses, and the MSWs at odd addresses. The coefficients thus require a circular buffer length (specified by L6) that is twice the length of the filter. As in the single-precision SG program in the previous section, the first instruction is used to compute the product of $e_c(T)$, the estimation error, and $\beta$, the step size, in parallel with the data memory read that transfers the oldest input data value in the delay line, y(T–n+1), to MX0. Upon entering the *adaptd* loop, y(T–n+1) is multiplied by $\beta e_c(T)$ to yield the nth coefficient update value. This is performed in parallel with the fetch of the LSW of the nth coefficient (to AY0). The next instruction computes the sum of the update value LSW (in MR0) and the LSW of the nth coefficient, while performing a dual fetch of the MSW of the nth coefficient (again to AY0) and the next data value in the delay line (to MX0). The LSW of the updated nth coefficient is then written back to program memory in parallel with the update of the MSW of the nth coefficient, and the final instruction of the loop writes this updated MSW to program memory. The *adaptd* loop continues execution in this manner until all *n* double-precision coefficients have been updated. If you want saturation capability on the coefficient update, you must enable and disable the saturation mode of the ALU within the update loop. The updates of the LSWs of the coefficients should be performed with the saturation mode disabled; the update of the MSWs of the coefficients should be performed with the saturation mode enabled.

To determine whether an application can benefit from double-precision adaptation, you should evaluate the performance of the associated filtering routine using both single-precision and double-precision coefficients. In some instances, maintaining the coefficients in double precision while performing the filtering operation on only the MSWs of the coefficients may result in the desired amount of cancellation. This adaptation can be achieved using the single-precision transversal filter routine with an M5 value of two. If more cancellation is needed, the routine in Listing 5.17 must be used.

```
.MODULE drsg_sub;

{
   Double-Precision SG Coefficient Adaptation Subroutine

   Calling Parameters
        MX0 = Error
        MY1 = Beta
        I2 —> Oldest input data value in delay line
        L2 = Filter length
        I6 —> Beginning of filter coefficient table
        L6 = 2 × Filter length
        M1,M5 = 1
        M3,M7 = –1
        CNTR = Filter length

   Return Values
        Coefficients updated
        I2 —> Oldest input data value in delay line
        I6 —> Beginning of filter coefficient table

   Altered Registers
        AY0,AR,MX0,MF,MR

   Computation Time
        (4 × Filter length) + 5 + 4 cycles

   All coefficients are assumed to be in 1.31 format.
   All data are assumed to be in 1.15 format.
}

.ENTRY  drsg;

drsg:   MF=MX0*MY1(RND), MX0=DM(I2,M1);         {MF=Error × Beta}
        MR=MX0*MF(SS);
        DO adaptd UNTIL CE;
               MX0=DM(I2,M1), AY0=PM(I6,M5);
               AR=MR0+AY0, AY0=PM(I6,M7);
               PM(I6,M5)=AR, AR=MR1+AY0+C;
adaptd:        MR=MX0*MF(SS), PM(I6,M5)=AR;
        MODIFY (I2,M3);                         {Point to oldest data}
        RTS;
.ENDMOD;
```

**Listing 5.17 Double-Precision Stochastic Gradient**

# 5 Digital Filters

## 5.7 REFERENCES


Bellanger, M. 1984. *Digital Processing of Signals: Theory and Practice*. New York: John Wiley and Sons.

Bloom, P.J. October 1985. *High Quality Digital Audio in the Entertainment Industry: An Overview of Achievements and Changes*. IEEE ASSP Magazine, Vol. 2, Num. 4, pp. 13-14.

Crochiere, Ronald E. and Lawrence R. Rabiner. 1983. *Multirate Digital Signal Processing.* Englewood Cliffs, N.J.: Prentice-Hall.

Hamming, R. W. 1977. *Digital Filters*. Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Honig, M. and Messerschmitt, D. 1984. *Adaptive Filters: Structures, Algorithms, and Applications*. Boston: Kluwer Academic Publishers.

Jackson, L. B. 1986. *Digital Filters and Signal Processing*. Boston: Kluwer Academic Publishers.

Liu, Bede and Abraham Peled. 1976. *Theory Design and Implementation of Digital Signal Processing*, pp. 77-88. John Wiley & Sons.

Liu, Bede and Fred Mintzer. December 1978. *Calculation of Narrow Band Spectra by Direct Decimation*. IEEE Trans. Acoust. Speech Signal Process., Vol. ASSP-26, No. 6, pp. 529-534.

Oppenheim, A. V., and Schafer, R. W. 1975. *Digital Signal Processing*. Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Oppenheim, A.V. ed. 1978. *Applications of Digital Signal Processing*. Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Otnes, R.K and L.E Enochson. 1978. *Applied Time Series Analysis*, pp. 202-212. Wiley-Interscience.

Rabiner, L. R. and Gold, B. 1975. *Theory and Applications of Digital Signal Processing*. Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Schafer, Ronald W. and Lawrence R. Rabiner. June 1973. *A Digital Signal Processing Approach to Interpolation*. Proc. IEEE, vol. 61, pp. 692-702.

Widrow, B., and Hoff, M., Jr. 1960. *Adaptive Switching Circuits*. IRE WESCON Convention Record, Pt. 4., pp. 96-104.