Elec4621:
Advanced Digital Signal Processing
Chapter 3: Filter Implementation Techniques

Dr. D. S. Taubman

March 11, 2011

1 Filter Structures

1.1 Canonical Form

In the previous chapter, we gave the following difference equation as a general form for LTI filters which can be implemented causally with a finite amount of computation and memory:

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

Here, N is the number of non-trivial poles and M is the number of non-trivial zeros in the filter's transfer function,

$$\begin{aligned} H(z) &= \frac{a_0 + a_1 z^{-1} + \cdots + a_M z^{-M}}{1 - b_1 z^{-1} - \cdots - b_N z^{-N}} \\ &= \frac{a_M}{-b_N} \cdot \frac{(z-z_1)(z-z_2)\cdots(z-z_M)}{z^M} \cdot \frac{z^N}{(z-p_1)(z-p_2)\cdots(z-p_N)} \end{aligned}$$

The most direct implementation of equation (1) is that shown in Figure 1. This structure is a direct transcription of the difference equations and is known as the "direct form." We have split the accumulation process into two adders primarily to distinguish between contributions from the first and second summations on the right hand side of equation (1).

An alternative implementation structure may be derived by noting that $H(z)$ may be regarded as the product of two transfer functions: an all-zero transfer function,

$$H_1(z) = a_0 + a_1 z^{-1} + \cdots + a_M z^{-M}$$

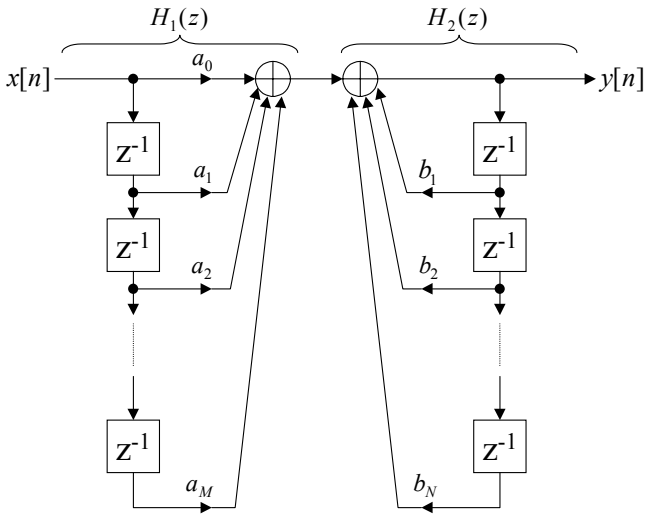

Figure 1: *Direct form implementation structure for a filter with M zeros and N poles.*

and an all-pole transfer function,

$$H_2(z) = \frac{1}{1 - b_1 z^{-1} - \cdots - b_N z^{-N}}$$

The left hand section of Figure 1 implements $H_1(z)$, feeding its result to the right hand section which implements $H_2(z)$. But $H(z) = H_1(z) H_2(z) = H_2(z) H_1(z)$ can be realized by implementing the all-zero and all-pole sections in the opposite order, as illustrated in Figure 2.

The structure shown in Figure 2 is known as the "canonical form." It has one obvious advantage over the direct form structure: **it requires less memory**. This is because the all-pole filter requires delayed copies of its output and the all-zero filter requires delayed copies of its input. By cascading them in the order of Figure 2, the delay operators may be shared. Each delay requires memory to store a single sample.

You may think of each string of delay elements, identified by $\boxed{Z^{-1}}$ in the figures, as a shift register. The canonical form requires a single shift register, with storage for $\max\{M, N\}$ samples, while the direct form structure requires two shift registers, having combined storage for $M+N$ samples. Filters are linear state machines and the canonical form exposes its fundamental states. The direct implementation involves additional redundant states. There are specific types of filters which we shall encounter, whose state information can be reduced further, without affecting the order of the filter, but these are not generic.

Apart from memory, the two implementation structures depicted in Figures 1 and 2 have very different implications for the propagation of numerical round-

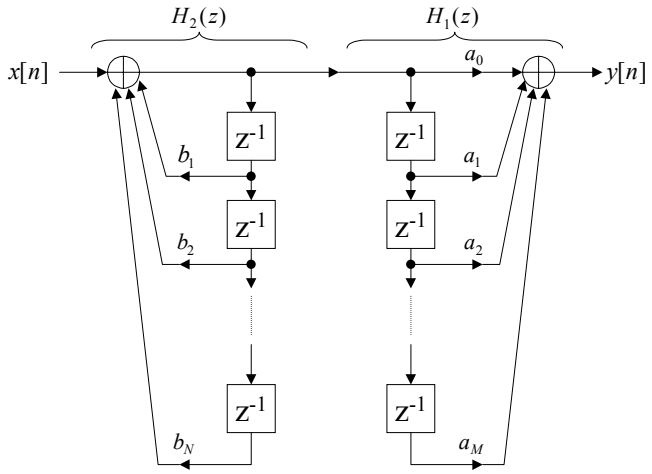

Figure 2: *Canonical form implementation structure for a filter with M zeros and N poles.*

off errors to the filter output. As we shall see in Section 2, round-off errors may be modeled as independent white noise sources injected into each of the accumulator blocks in the filter structure. In the direct form structure of Figure 1, there is only one accumulator block and its round-off errors are processed by the all-pole filter, $H_2(z)$. This tends to amplify the round-off noise power at frequencies (points on the unit circle) near the poles. The canonical form shown in Figure 2 has noise sources injected at the filter input and at the filter output. If the pole-zero structure is such that zeros nearly cancel the poles, the direct form will result in substantially higher peaks in the round-off noise power spectrum than the canonical structure.

The lesson to be learned from the above discussion is that a filter may be implemented in various quite different ways, each of which can have very different implications for memory, round-off noise and other important attributes which we shall examine here. Armed with an arsenal of different implementation structures, and tools for analyzing the implications of each structure, you will be able to try a variety of strategies and select the one which is most suited to your application. In the sections which follow, we investigate a number of other useful implementation structures.

1.2 Cascade Structures

Since we are interested in filters with real-valued coefficients, $H(z)$ can always be factored into a cascade of first and second order sections,

$$H(z) = H_1(z) \cdot H_2(z) \cdot \cdots$$

where first order sections have the form

$$H_i(z) = \frac{z - z_i}{z - p_i}$$

and second order sections have the form

$$H_i(z) = \frac{z^2 + a_{1,i}z + a_{2,i}}{z^2 + b_{1,i}z + b_{2,i}}$$

If $M = N$, all zeros and poles in these segments are non-trivial. If $M > N$, there will be some trivial poles, while if $M < N$ there will be some trivial zeros. Each filter section may, itself be implemented in direct form, or in canonical form.

One advantage of cascade structures over the direct or canonical forms shown in Figures 1 and 2 is that the pole and zero locations are much less sensitive to coefficient quantization effects. Most filter designs yield coefficients with irrational values which cannot be represented exactly using finite precision arithmetic. Since the coefficients of the implemented filter are not identical to the designed values, the poles and zeros of the implemented filter will also be different. If the numerator or denominator polynomials have high order, the pole and zero locations can be very sensitive to small changes in the coefficients, a_i and b_i. If care is not taken, a filter with poles close to the unit circle can even become unstable as a result of coefficient quantization effects. The situation is substantially better for cascade structures, since each section has its own poles and zeros, whose locations are affected by only one or two coefficients.

1.3 Parallel Structures

Rather than expanding $H(z)$ as a product of first and second order factors, a partial fraction expansion may be used to express $H(z)$ as a sum of first and second order systems. The following example illustrates the general principle.

Example 1 *Consider a filter with the following transfer function*

$$H(z) = \frac{\left(z - \frac{1}{2}\right) z^2}{\left(z - \frac{1}{4}\right)\left(z^2 + z + \frac{1}{2}\right)}$$

The filter has a non-trivial zero at $z = \frac{1}{2}$, a simple pole at $z = \frac{1}{4}$ and a pair of complex conjugate poles at $z = re^{\pm j\theta}$ where $r = \frac{1}{\sqrt{2}}$ and $\theta = \frac{\pi}{4}$. A partial fraction expansion is obtained by expressing $H(z)$ as

$$H(z) = \frac{1}{2}\frac{z + \alpha}{z - \frac{1}{4}} + \frac{1}{2}\frac{z^2 + \beta z + \gamma}{z^2 + z + \frac{1}{2}}$$

and solving for the coefficients, α, β and γ. The coefficients are found by equating the numerator of $H(z)$ with

$$\frac{1}{2}(z + \alpha)\left(z^2 + z + \frac{1}{2}\right) + \frac{1}{2}\left(z^2 + \beta z + \gamma\right)\left(z - \frac{1}{4}\right)$$

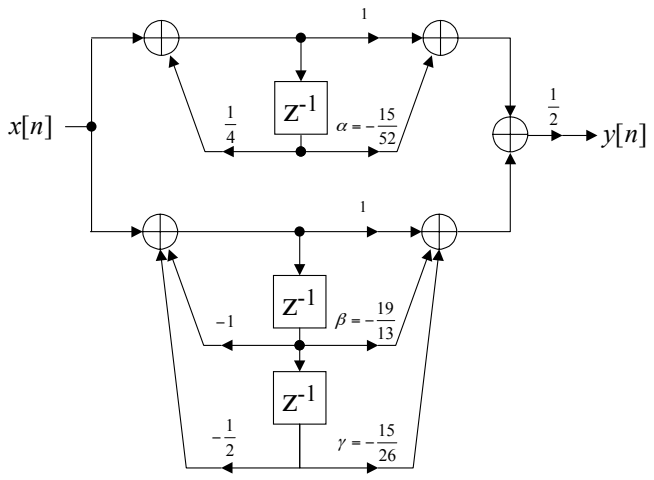

Figure 3: *Parallel filter structure, realizing* $H(z) = \frac{\left(z-\frac{1}{2}\right)z^2}{\left(z-\frac{1}{4}\right)\left(z^2+z+\frac{1}{2}\right)}$.

Multiplying both polynomials by 2 *yields*

$$2z^3 - z^2 = 2z^3 + z^2\left(1+\alpha+\beta-\frac{1}{4}\right) + z\left(\frac{1}{2}+\alpha-\frac{1}{4}\beta+\gamma\right) + 1\left(\frac{1}{2}\alpha-\frac{1}{4}\gamma\right)$$

So

$$\begin{aligned}\alpha &= -\beta - \frac{7}{4}\\ \gamma &= -\alpha + \frac{1}{4}\beta - \frac{1}{2}\\ \gamma &= 2\alpha\end{aligned}$$

Solving these equations yields $\left\{\beta = -\frac{19}{13}, \alpha = -\frac{15}{52}, \gamma = -\frac{15}{26}\right\}$. *The corresponding parallel filter structure is shown in Figure 3.*

Parallel structures are not so popular as cascade structures, in part due to the added complexity of expanding the filter in this form and in part because they often have higher implementation complexity. Pole positions have exactly the same sensitivity to coefficient quantization as they do in the cascade structure; however, zero locations are generally more sensitive to coefficient quantization than they are in the cascade structure.

One advantage of parallel implementations over cascade implementations is their sensitivity to numerical round-off errors. The round-off noise sources associated with a parallel implementation are injected at the input and output of each parallel filter section, while the round-off noise sources in a cascade structure are injected at the input and output of each successive filter stage. If a filter has multiple poles close together and near the unit circle, round-off

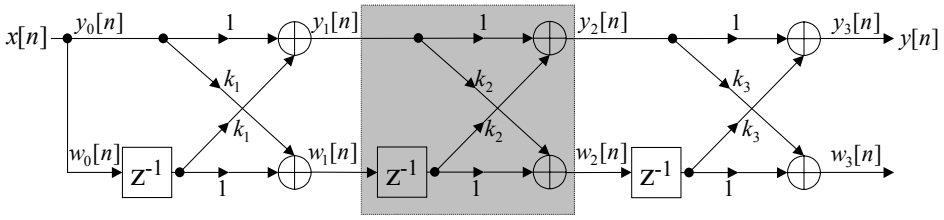

Figure 4: *Third order FIR lattice filter implementation. The shaded box identifies the regular structure which is repeated throughout the lattice.*

errors introduced at the beginning of a cascade will be shaped by the transfer function associated with the remaining elements in the cascade, with high gain at frequencies close to the poles. By contrast, round-off errors introduced into a parallel structure are only shaped by the transfer function of a single section, having at most two poles.

1.4 Lattice Structures

In this section, we consider an entirely different and much less obvious implementation structure to those described above. Figure 4 shows a lattice structure which can be used for implementing FIR filters. We shall consider a complementary all-pole lattice structure shortly.

The characteristic feature of lattices is that they are composed of regular repeating sections, each of which has two inputs and two outputs (unlike the single input, single output sections we have considered previously). The final filter is obtained by joining the two front-end inputs and selecting one of the two back-end outputs. For ease of discussion, we will use the symbols, $y_m[n]$ and $w_m[n]$, to describe the outputs from the m^{th} lattice section. The inputs to that lattice section are then $y_{m-1}[n]$ and $w_{m-1}[n]$. These are illustrated in Figure 4.

The relationship between these various quantities may be described by the following equations

$$\begin{aligned} y_m[n] &= y_{m-1}[n] + k_m w_{m-1}[n-1] \\ w_m[n] &= w_{m-1}[n-1] + k_m y_{m-1}[n] \end{aligned}$$

In the Z-transform domain, we may write these in the following convenient

matrix form:

$$\begin{aligned}
\begin{pmatrix} Y_m(z) \\ W_m(z) \end{pmatrix} &= \begin{pmatrix} 1 & k_m \\ k_m & 1 \end{pmatrix} \begin{pmatrix} Y_{m-1}(z) \\ z^{-1} W_{m-1}(z) \end{pmatrix} \\
&= \begin{pmatrix} 1 & k_m z^{-1} \\ k_m & z^{-1} \end{pmatrix} \begin{pmatrix} Y_{m-1}(z) \\ W_{m-1}(z) \end{pmatrix} \\
&= \begin{pmatrix} 1 & k_m z^{-1} \\ k_m & z^{-1} \end{pmatrix} \begin{pmatrix} 1 & k_{m-1} z^{-1} \\ k_{m-1} & z^{-1} \end{pmatrix} \begin{pmatrix} Y_{m-2}(z) \\ W_{m-2}(z) \end{pmatrix} \\
&\vdots \\
&= \left[\prod_{i=1}^{m} \begin{pmatrix} 1 & k_i z^{-1} \\ k_i & z^{-1} \end{pmatrix} \right] \begin{pmatrix} X(z) \\ X(z) \end{pmatrix}
\end{aligned}$$

This transfer matrix expansion provides us with one method to determine the filter transfer function, $Y_m(z)/X(z)$.

To understand the lattice much better, however, let $A_m(z)$ and $B_m(z)$ denote the transfer functions,

$$\begin{aligned}
A_m(z) &= \frac{Y_m(z)}{X(z)} \\
B_m(z) &= \frac{W_m(z)}{X(z)}
\end{aligned}$$

It turns out that these transfer functions are mirror images of each other for all m. Specifically, the corresponding impulse responses satisfy

$$b_m[n] = a_m[m-n]$$

Note that both impulse responses have support $0 \le n \le m$. In the Z-transform, time reversal is equivalent to replacing z with z^{-1}, so this relationship may be written

$$B_m(z) = z^{-m} A_m\left(z^{-1}\right)$$

and, of course, we also have

$$A_m(z) = z^{-m} B_m\left(z^{-1}\right)$$

The mirror image relationship between $A_m(z)$ and $B_m(z)$ may be shown by a simple recursion. When $m = 0$, we trivially have $A_0(z) = B_0(z) = 1$. Now supposing the symmetry to hold for some m, we may recursively verify that it holds at $m+1$. We get

$$\begin{aligned}
B_{m+1}(z) &= z^{-1} B_m(z) + k_{m+1} A_m(z) \\
&= z^{-(m+1)} A_m\left(z^{-1}\right) + k_{m+1} z^{-m} B_m\left(z^{-1}\right) \\
&= z^{-(m+1)} \left[A_m\left(z^{-1}\right) + k_{m+1} z B_m\left(z^{-1}\right) \right] \\
&= z^{-(m+1)} \left[A_m(u) + k_{m+1} u^{-1} B_m(u) \right]_{u=z^{-1}} \\
&= z^{-(m+1)} A_{m+1}\left(z^{-1}\right)
\end{aligned}$$

We may use the mirror image relationship between the transfer functions of the lower and upper lattice branches to work back from a desired all-zero transfer function, $H(z) = Y(z)/X(z)$, to find the corresponding lattice coefficients. The key is to observe that the forward relationship

$$\begin{pmatrix} A_m(z) \\ B_m(z) \end{pmatrix} = \begin{pmatrix} 1 & k_m z^{-1} \\ k_m & z^{-1} \end{pmatrix} \begin{pmatrix} A_{m-1}(z) \\ B_{m-1}(z) \end{pmatrix}$$

may be inverted as

$$\begin{aligned} \begin{pmatrix} A_{m-1}(z) \\ B_{m-1}(z) \end{pmatrix} &= \begin{pmatrix} 1 & k_m z^{-1} \\ k_m & z^{-1} \end{pmatrix}^{-1} \begin{pmatrix} A_m(z) \\ B_m(z) \end{pmatrix} \\ &= \frac{1}{z^{-1} - k_m^2 z^{-1}} \begin{pmatrix} z^{-1} & -k_m z^{-1} \\ -k_m & 1 \end{pmatrix} \begin{pmatrix} A_m(z) \\ z^{-m} A_m\left(z^{-1}\right) \end{pmatrix} \end{aligned}$$

from which we get

$$A_{m-1}(z) = \frac{A_m(z) - k_m z^{-m} A_m\left(z^{-1}\right)}{1 - k_m^2} \tag{2}$$

Working backward through the lattice, we may determine each coefficient, k_m, in turn by recognizing that $A_{m-1}(z)$ is an order $m-1$ filter, so that $a_m[m] - k_m a_m[0]$ must be 0. That is

$$k_m = \frac{a_m[m]}{a_m[0]} \tag{3}$$

The lattice coefficients are generally known as "reflection coefficients."

Example 2 *Consider the third order FIR filter,*

$$H(z) = 1 + \frac{1}{4}z^{-1} - \frac{1}{4}z^{-2} + \frac{1}{2}z^{-3}$$

We know that $A_3(z) = H(z)$, so equation (3) yields

$$k_3 = \frac{a_3[3]}{a_3[0]} = \frac{h[3]}{h[0]} = \frac{1}{2}$$

Next, we use equation (2) to find

$$\begin{aligned} A_2(z) &= \frac{\left(1 + \frac{1}{4}z^{-1} - \frac{1}{4}z^{-2} + \frac{1}{2}z^{-3}\right) - \frac{1}{2}\left(\frac{1}{2} - \frac{1}{4}z^{-1} + \frac{1}{4}z^{-2} + z^{-3}\right)}{1 - \frac{1}{4}} \\ &= \frac{4}{3}\left(\frac{3}{4} + \frac{3}{8}z^{-1} - \frac{3}{8}z^{-2}\right) \\ &= 1 + \frac{1}{2}z^{-1} - \frac{1}{2}z^{-2} \end{aligned}$$

We can now use equation (3) again to find

$$k_2 = \frac{a_2[2]}{a_2[0]} = -\frac{1}{2}$$

from which we get

$$\begin{aligned} A_1(z) &= \frac{\left(1+\frac{1}{2}z^{-1}-\frac{1}{2}z^{-2}\right)+\frac{1}{2}\left(-\frac{1}{2}+\frac{1}{2}z^{-1}+z^{-2}\right)}{1-\frac{1}{4}} \\ &= \frac{4}{3}\left(\frac{3}{4}+\frac{3}{4}z^{-1}\right) \\ &= 1+z^{-1} \end{aligned}$$

Then

$$k_1 = \frac{a_1[1]}{a_1[0]} = 1$$

and we have found all of the reflection coefficients.

1.4.1 Properties of Lattice Filters

Since the lattice structure involve more multiplications than the cascade, direct or canonical filter structures, one might reasonably ask why we would want to go to the trouble. Lattice filters actually arise naturally in the context of adaptive filtering systems, where they have excellent properties which we shall consider later in the subject. For now, however, we point out the following very useful property of lattice filters:

> An FIR filter has minimum phase (all zeros inside the unit circle) if and only if all of the reflection coefficients are less than 1, i.e., $|k_m| < 1$.

Recall that minimum phase filters are the only filters which we can invert causally. In fact, the inverse filter may be implemented using the all-pole lattice structure described next, which is guaranteed to remain stable so long as the reflection coefficients are all less than 1. The coefficients for such filters may be efficiently represented as fixed-point quantities of the form

$$k_m \approx k'_m \cdot 2^{-(W-1)}$$

where k'_m is a W-bit two's complement integer.

1.4.2 All-Pole Lattice Structure

Figure 5 shows an all-pole lattice structure. Note the close similarity between this structure and that in Figure 4. In fact, if k_1, k_2, ■, k_M are the reflection coefficients of an FIR lattice with filter transfer function $H(z)$, then they are also the reflection coefficients of the IIR lattice with filter transfer function, $1/H(z)$. Of course, we require $H(z)$ to have minimum phase, which is simply arranged by ensuring that $|k_m| < 1$ for all m.

The reciprocal relationship between the FIR and IIR lattice structures in Figures 4 and 5 is easily explained by considering the repeating structures,

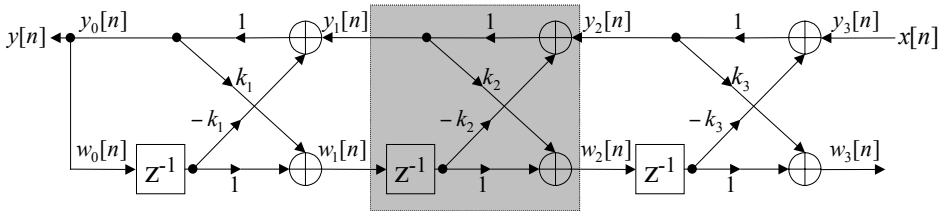

Figure 5: *Third order all-pole lattice filter whose transfer function is the reciprocal of that of the FIR lattice filter in Figure 4. We have deliberately drawn the filter backwards, with the input on the right and the output on the left to emphasize the connection between the FIR and IIR lattice structures and their internal state variables.*

identified by shaded boxes in the figures. These structures clearly enforce exactly the same relationship between the vectors,

$$\begin{pmatrix} y_{m-1}[n] \\ w_{m-1}[n] \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} y_m[n] \\ w_m[n] \end{pmatrix}$$

The signal flow direction is reversed in the upper branch, so addition of $k_m w_{m-1}[n-1]$ to $y_{m-1}[n]$ to get $y_m[n]$ is replaced by subtraction of $k_m w_{m-1}[n-1]$ from $y_m[n]$ to get $y_{m-1}[n]$. There is no other difference between the FIR and IIR lattices. It follows that the relationship between the signals $(y_0[n], w_0[n])$ and $(y_M[n], w_M[n])$ must also be preserved. In the IIR lattice, however, $y_M[n]$ is the input and $y_0[n]$ is the output, so the IIR lattice must invert the FIR lattice.

If we pass a signal, $x[n]$, into the input of an FIR lattice and then pass the output signal, $y[n]$ into the input (at $y_M[n]$) of the corresponding IIR lattice, all intermediate quantities and all states in the IIR lattice will assume identical values to their counterparts in the FIR lattice.

1.5 Other Structures

While the implementation structures described above are most widely used, they are far from exhaustive. As an example, Figure 6 provides an alternate implementation for a second order all-pole filter (or filter section). This example is modeled on the structure of coupled oscillators found in nature and in analog circuits. Two single pole sections have their inputs and outputs partially coupled, resulting in an all-pole filter with complex conjugate poles.

The transfer function for this filter is easily found by recognizing that each of the single-pole building blocks has transfer function $G(z) = \frac{z^{-1}}{1-az^{-1}} = \frac{1}{z-a}$,

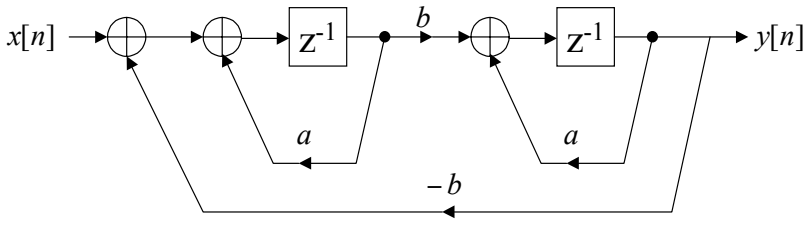

Figure 6: *Second order all-pole filter based on coupled single-pole sections.*

and hence the complete coupled structure has transfer function

$$\begin{aligned} H(z) &= \frac{G(z)\, bG(z)}{1 + bG(z)\, bG(z)} \\ &= \frac{\frac{b}{(z-a)^2}}{1 + \frac{b^2}{(z-a)^2}} = \frac{b}{(z-a)^2 + b^2} \end{aligned}$$

The poles are thus at locations $z = a \pm jb$.

The coupled implementation requires twice as many multiplications as a direct, canonical or cascade implementation. On the other hand, its pole positions have low sensitivity to quantization. Regardless of how much error we make in the approximation of a and b, the filter is guaranteed to have complex conjugate pole pairs.

2 Finite Word Length Effects

2.1 Multiply-Accumulate Operations

Linear filters are invariably implemented through a sequence of multiplications and additions. In each case, a signal-dependent quantity, x, is multiplied by some coefficient, a, and the result is added to a previously computed signal-dependent quantity, s. We may summarize this "multiply-accumulate" operation as

$$s \leftarrow s + a \cdot x$$

Example 3 *Consider the canonical second order filter structure shown in Figure 7. The implementation involves two memories (or states), s_1 and s_2, corresponding to the outputs of the first and second delay element, respectively. As each new input sample $x[n]$ arrives, the following operations are performed to derive a new output sample, $y[n]$.*

- $s_0 \leftarrow x[n]$ *use s_0 to accumulate output of first adder*
- $s_0 \leftarrow s_0 + b_1 \cdot s_1$

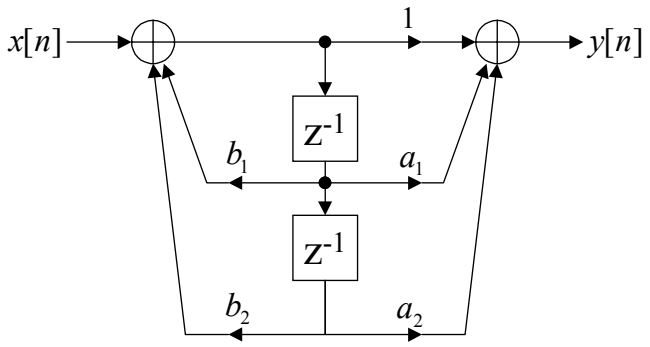

Figure 7: *Second order canonical filter section.*

- $s_0 \leftarrow s_0 + b_2 \cdot s_2$
- $y \leftarrow s_0$ *use y to accumulate output of second adder*
- $y \leftarrow y + a_1 \cdot s_1$
- $y \leftarrow y + a_2 \cdot s_2$
- $y\,[n] \leftarrow y$
- $s_2 \leftarrow s_1 \leftarrow s_0$ *operate the shift register to prepare for next sample*

The complete implementation thus involves four multiply-accumulate operations per sample. Two temporary accumulator registers, y and s_0 are also used, together with two state variables, s_1 and s_2. To see why state variables are fundamentally different to the temporary accumulator variables, consider the following.

If the complete filter involves multiple cascaded sections, the implementation of each section may share the same temporary accumulator variables, but each section must have its own independent set of state variables.

2.2 Fixed-Point Arithmetic

By and large, DSP chips are general purpose micro-processors which have been specially designed to process multiply-accumulate operations with a high throughput[1]. DSP chips are designed either for fixed-point or floating-point processing, but in this section we focus on fixed-point processing. Fixed-point arithmetic can be performed more efficiently, with lower latency and less circuitry, but it places a greater demand on the implementor to customize the dynamic range of the data which is being processed to the available word lengths.

[1]Other DSP-specific features typically include multiple data/address buses and special address generation logic which is well adapted to certain typical DSP operations.

For full custom hardware solutions, similar considerations apply and fixed-point implementations generally result in smaller, less expensive chips with higher throughput. We briefly consider floating point implementations in Section 2.7.

Fixed-point processors treat all quantities as integers. It is up to the implementor to establish a relationship between these integers and the real-valued quantities which would ideally be processed. For example, suppose that the input sequence satisfies $|x[n]| < A$ for some bound, A, and we wish to represent the samples as two's complement integers, each having W bits. This could be achieved by scaling the input samples by $2^{W-1}/A$ and rounding to the nearest integer. The implementor must then bear in mind that the integer values are scaled versions of the original input samples, scaling the output back accordingly.

To minimize the impact of integer rounding effects, it is desirable to scale the results produced by each processing step so that they utilize as many of the available bits as possible. This can lead to a large number of different scale factors, which may require additional multiplications[2]. For this reason, it is common to adopt a convention which restricts the scale factors to exact powers of 2. This leads to a description of fixed-point quantities in terms of two quantities: the number, I, of integer bits; and the number, F, of fraction bits.

We say that a real-valued quantity x has an $I.F$ fixed point representation as a $W = I + F$ bit integer x', if

$$x = 2^{-F} x' \tag{4}$$

We can also express this fixed point representation graphically by inserting a "binary point" after the F^{th} most significant bit.

Example 4 *The real-valued quantity,* $x = 3.125$*, may be represented exactly as an unsigned* 2.3 *fixed point quantity,*

$$x = 11.001$$

The 3 *fraction bits are separated from the* 2 *integer bits by the binary point. The integer* x'*, used to represent* x*, is given by*

$$x' = 11001$$

which has a value of $25 = 2^3 x$

We will usually be working with two's complement signed integer representations. This means that the maximum and minimum integers which can be represented using W bits are -2^{W-1} and $2^{W-1} - 1$, respectively. Consequently, the range of real-valued quantities, x, which can be represented without overflow in an $I.F$ fixed-point representation is given by

$$-2^{I+F-1} \leq 2^F x \leq 2^{I+F-1} - 1$$

[2] You should always implement scaling through multiplication, rather than division, since multiplication is much faster, especially when working with high precision representations.

or, equivalently,

$$-2^{I-1} \leq x \leq 2^{I-1} - 2^{-F}$$

It is often convenient to approximate this range by

$$|x| < 2^{I-1}$$

Example 5 *The real-valued quantity,* $x = -2.625$*, may be expressed in a signed* 4.4 *representation by the* 8*-bit integer*

$$\begin{aligned} x' &= -2^4 x = -42 \\ &= 11010110 \end{aligned}$$

The maximum value which can be represented as a 4.4 *quantity is*

$$0111.1111 = \frac{127}{16} = 3.9375$$

The smallest quantity is

$$1000.0000 = \frac{-128}{16} = -4$$

If two fixed-point quantities have the same number of fraction bits, they can be added directly as integers, without any round-off errors. If they have different numbers of fraction bits, one or both of the quantities must have its representation changed by multiplying or dividing the integer values by an appropriate power of 2. We refer to this as a normalization shift, since multiplication or division by a power of 2 amounts to shifting the two's complement integer quantities to the left or right, respectively.

Example 6 *Suppose we need to add a* 4.4 *quantity,* x_1*, to a* 6.2 *quantity,* x_2*. Both quantities are represented as* 8*-bit integers, but with different numbers of fraction bits. If* x_2 *really requires* 6 *integer bits, we can expect the result to also require at least* 6 *integer bits, meaning that we should drop* 2 *fraction bits from* x_1 *before adding the numbers. We can implement this as follows (C-language syntax used for convenience):*

- $x'_1 \leftarrow (x'_1 + 2) >> 2$ – *Obtains nearest (by rounding)* 6.2 *representation of* x_1*.*
- $y' \leftarrow x_1^{'} + x'_2$ – y' *has* 2 *fraction bits.*

Multiplying two W-bit signed integers, x'_1 and x'_2, yields a $(2W-1)$-bit signed integer, y', in the range $2^{2W-2} < y \leq 2^{2W-2}$. If the multiplicands correspond to $I_1.F_1$ and $I_2.F_2$ fixed-point quantities, respectively, y' is an $(I_1 + I_2 - 1).(F_1 + F_2)$ fixed-point representation of $y = x_1 \cdot x_2$. Note that multiplication itself does not introduce any round-off errors. Round-off errors arise when we need to reduce the precision of the result back to a W-bit quantity by discarding fraction bits.

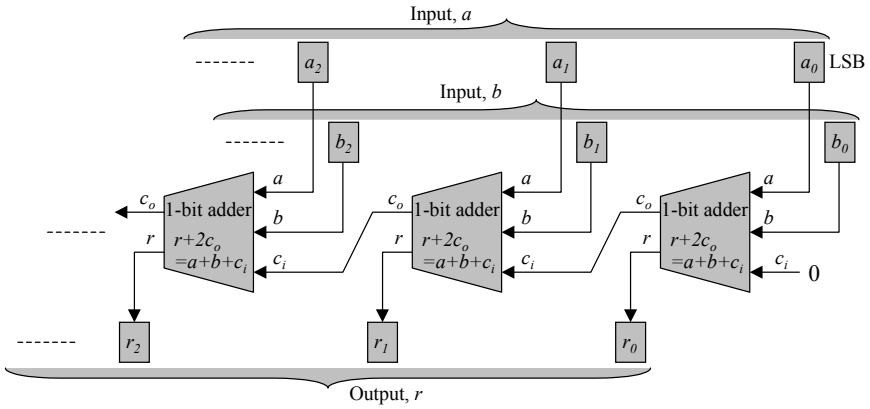

Figure 8: *Typical ripple-adder circuit constructed from* 1*-bit full adder cells, each of which sums three input bits,* a, b, c_i*, and outputs the sum as a* 2*-bit quantity,* r, c_o*.*

2.2.1 Relative Complexity of Operations

Since fixed-point arithmetic is actually integer arithmetic, with a superimposed convention for the interpretation of the integers, the complexity of the relevant operations depends only upon the word size, W. Addition of two W_+-bit operands is almost invariably implemented as a logic circuit consisting of a cascade of W_+ full adder cells, as shown in Figure 8. Each one-bit full adder takes three inputs: an input bit from each of the two words being added; and a carry bit form the next least significant bit position. The full adder cell generates two outputs: a result bit; and a carry bit, to be propagated to the next more significant bit position.

Multiplication can be implemented in various ways, but the most common architecture found in DSP applications (and also most modern CPU's) is a "parallel" multiplication circuit consisting of a $W_\times \times W_\times$ array of one-bit adder cells, coupled by AND gates, as shown in Figure 9. The total number of gates required to implement a multiplier is proportional to the product of the word sizes of the two multiplicands. Thus, for example, a 16×16 multiplier unit requires 256 one-bit adder cells. By contrast, a 16-bit adder requires only 16 one-bit adder cells.

Although the complexity of a parallel multiplier grows with the square of input word size, its propagation delay does not. The critical signal propagation path in Figure 9 skirts the upper and left hand boundaries of the array of adder cells, involving 8 adder cells. In general, the propagation delay for a parallel adder circuit is proportional to the sum of the word sizes of its inputs. Thus, the delay associated with a 16×16 parallel multiplier circuit is 32 times the propagation delay associated with a single adder cell, only twice the delay associated with a 16-bit adder.

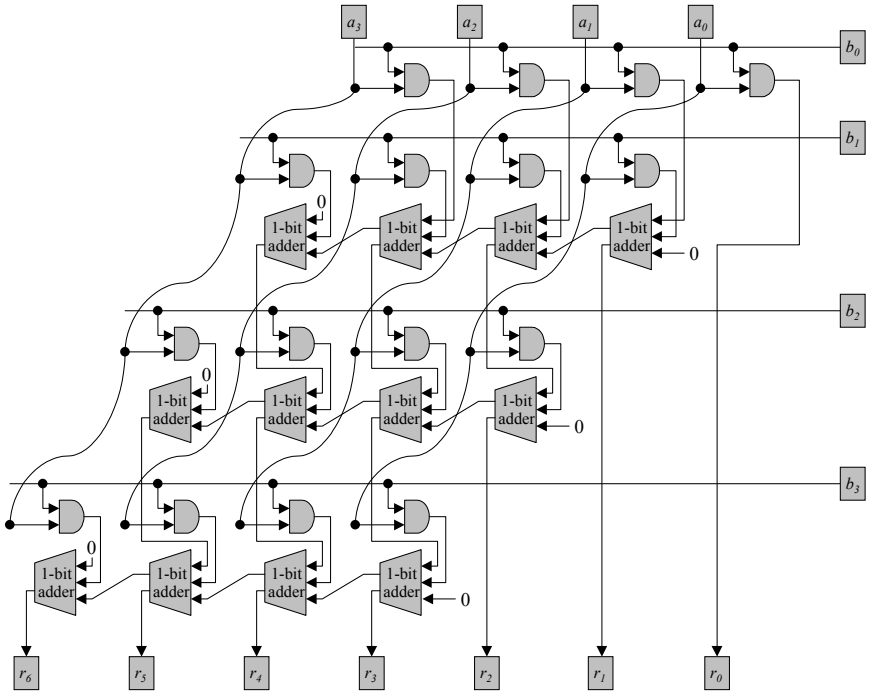

Figure 9: *4-bit by 4-bit parallel multplication circuit involving* 16 *one-bit adder cells and producing* 7 *output bits. For simplicity, the circuit shown here works only with an unsigned multplicand, b. a can be signed or unsigned.*

If the parallel multiplier unit in Figure 9 is coupled directly to the adder unit in Figure 8 to implement a multiply-accumulate unit, the total propagation delay is identical to that of the multiplier by itself, plus a single adder cell delay. In other words, both the complexity and the propagation delay of a multiply-accumulate unit are almost identical to those of the multiplier by itself.

The lesson to be learned from the above is that additions are comparatively cheap and multiplications dominate the cost of hardware DSP solutions. DSP chips generally provide accumulators with significantly larger bit-depths than the multiplier, since the cost of implementing the accumulator is negligible in comparison with that of the multiplier. The accumulator bit-depth, W_+ is typically more than twice as large as the multiplicand bit-depth, $W_\times$.

For those of you who are wondering about division, you can forget it immediately. Division cannot be effectively implemented using a parallel logic unit, so dividers are typically a great deal slower than multipliers. You should never use division in any algorithm unless you absolutely cannot avoid it. Division by a fixed coefficient, a, is much better implemented as a multiplication by $\frac{1}{a}$.

2.3 Dynamic Range Selection

In this section we consider the problem of determining appropriate fixed-point representations for each point in the implementation of a filter. To avoid frequent renormalizations (adjustments in the number of fraction bits by bit shifting, as in Example 6), it is convenient to select the same representation for all quantities entering each accumulator block.

Since adders are much cheaper to implement than multipliers, accumulators can normally work with higher bit-depth than the multiplicands entering multiplier units. For example, a typical multiplier unit might accept 16-bit inputs, producing a 32-bit result, while all additions might be performed using 32-bit arithmetic. Since the output of each accumulator is usually an input to a subsequent multiplier, the outputs must be renormalized by discarding extra fraction bits.

To summarize the above statements, a good implementation should determine a suitable fixed-point representation for the input to and the output from each accumulator block. An appropriate design procedure is as follows.

1. Pick a fixed-point representation for the input samples, $x[n]$. The input to a digital filter generally arrives as integers already, with some implied fixed-point representation. For example, audio samples often arise from sampling and quantizing an analog waveform in the range $-1V$ to $+1V$, so it is natural to think of the W-bit integers as having a $1.(W-1)$ fixed-piont representation. Thus, 16-bit audio may be understood as having a 1.15 representation and 24-bit audio might be understood as having a 1.23 representation, with 7 extra fraction bits. In any event, let I_0 and F_0 denote the number of integer and fraction bits associated with the input representation.

2. Identify the accumulator blocks in the filter structure.

3. Identify the transfer function, $H_i(z)$, from the filter input to the output of the i^{th} accumulator block and find the associated BIBO gain:

$$G_i = \sum_{n=0}^{\infty} |h_i[n]|$$

To avoid any possibility of overflow, the output representation must be able to hold samples which are G_i times larger than the filter input samples. Accordingly, the number of integer bits required for the accumulator output representation is

$$I_i^{\text{out}} = I_0 + \lceil \log_2 G_i \rceil$$

where $\lceil a \rceil$ means "round a up to the nearest integer." Assuming a consistent word size, $W_\times = I_0 + F_0$, for all multiplier inputs, we must have

$$F_i^{\text{out}} = F_0 - \lceil \log_2 G_i \rceil$$

fraction bits at the output of the i^{th} accumulator block.

4. Next, we need to select the representation at the input to each accumulator block. Overflow in the accumulator is of no concern so long as the final result fits into the selected representation. This is because overflow bits from the outputs of multipliers or at intermediate points in the accumulation of multiplier outputs are guaranteed to cancel each other out, so long as we use a two's complement representation. It follows that the number of integer bits used during accumulation need only satisfy

$$I_i^{\text{in}} \geq I_i^{\text{out}} \tag{5}$$

The number of fraction bits used during accumulation will generally be much larger than the number of fraction bits available for multiplier inputs, since W_+ is typically at least twice as large as $W_\times$.

At this step, we select the minimum value, $I_i^{\text{in}} = I_i^{\text{out}}$, which maximizes the number of accumulator fraction bits and hence maximizes the accuracy with which we can represent the filter coefficients. We may find later, however, that I_i^{in} needs to be increased to satisfy other constraints (see Step 6).

5. Next, we assign multiplier coefficients in such a way as to reconcile the differences between input and output representations. For example, an input to the i^{th} accumulator block may be formed by multiplying some coefficient, a, by the output from the $(i-1)^{\text{st}}$ accumulator block. It follows that the representation of a should have F_a fraction bits, where F_a is given by

$$F_a = F_i^{\text{in}} - F_{i-1}^{\text{out}}$$

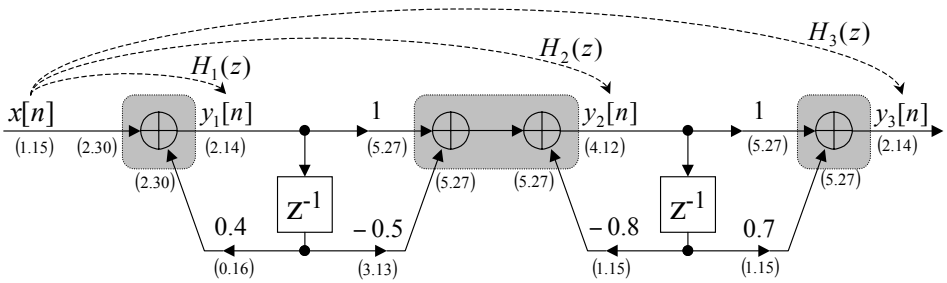

Figure 10: *Fixed-point representation selections for a cascade filter consisting of two first order sections. The filter has two real-valued zeros and two real-valued poles.*

6. It can happen that the above steps lead to some coefficients with insufficient integer bits in their representation. For example, assuming that all coefficients must be represented with the same word size, $W_\times$, the coefficient a has only $I_a = W_\times - F_a$ integer bits, requiring the coefficient to lie in the range
$$|a| < 2^{I_a - 1}$$
If this is too small, we must reduce the number of fraction bits, F_a. This, in turn, means reducing the number of fraction bits (increasing the number of integer bits) at the input to the relevant accumulator block, taking I_i^{in} above the minimum value given by equation (5). Reducing the number of fraction bits at the input to an accumulator is undesirable in that it reduces the accuracy of the representation. For this reason, this final adjustment step is best saved until last.

Example 7 *Consider the following transfer function.*
$$H(z) = \frac{z - 0.5}{z - 0.4} \cdot \frac{z + 0.7}{z + 0.8}$$
The filter may be implemented as a cascade of two first order sections, as shown in Figure 10.Evidently, there are three accumulator blocks, whose outputs we label $y_1[n]$, $y_2[n]$ and $y_3[n]$. The output of the entire filter is $y[n] = y_3[n]$. The transfer functions from $x[n]$ to each of these accumulator outputs are labeled

$H_1(z)$, $H_2(z)$ and $H_3(z)$. They are easily found to be

$$\begin{aligned} H_1(z) &= \frac{z}{z-0.4} \\ H_2(z) &= \frac{z-0.5}{z-0.4} \cdot \frac{z}{z+0.8} \\ &= \frac{1}{2}\frac{z-\frac{14}{15}}{z-0.4} + \frac{1}{2}\frac{z-\frac{7}{15}}{z+0.8} \\ H_3(z) &= H(z) = \frac{z-0.5}{z-0.4} \cdot \frac{z+0.7}{z+0.8} \\ &= \frac{1}{2}\frac{z-\frac{7}{12}}{z-0.4} + \frac{1}{2}\frac{z+\frac{7}{12}}{z+0.8} \end{aligned}$$

The corresponding impulse responses are

$$\begin{aligned} h_1[n] &= (0.4)^n u[n] \\ h_2[n] &= \frac{1}{2}(0.4)^n u[n] - \frac{14}{30}(0.4)^{n-1} u[n-1] \\ &+ \frac{1}{2}(-0.8)^n u[n] - \frac{7}{30}(-0.8)^{n-1} u[n-1] \\ h_3[n] &= \frac{1}{2}(0.4)^n u[n] - \frac{7}{24}(0.4)^{n-1} u[n-1] \\ &+ \frac{1}{2}(-0.8)^n u[n] + \frac{7}{24}(-0.8)^{n-1} u[n-1] \end{aligned}$$

Expanding these impulse responses, we evaluate the BIBO gains numerically. Note that we need only expand an initial prefix, since all impulse responses have exponential decay. We find BIBO gains of

$$\begin{aligned} G_1 &= \frac{1}{1-0.4} = 1\frac{2}{3} \\ G_2 &= 4.3571 \\ G_3 &= 1.6071 \end{aligned}$$

Assuming that the input has a 1.15 fixed-point representation of real-valued quantities in the range -1 to 1, we find that the outputs of the three accumulator blocks should have 2.14, 4.12 and 2.14 fixed-point representations, respectively.

If the accumulator has $W_+ = 32$ bits of precision, Step 4 in the design procedure selects optimal input representations of 2.30, 4.28 and 2.30 for the three accumulator blocks, noting that these may need to be adjusted in Step 6. Step 5 then yields the following filter coefficient representations.

- *Multiplication by 0.4 involves a 2.14 input and a 2.30 output. The coefficient must have a 0.16 representation, which is fine.*
- *Multiplication by -0.5 involves a 2.14 input and a 4.28 output. The coefficient must therefore have a 2.14 representation, which is also fine.*

- *Multiplication by* -0.8 *involves a* 4.12 *input and a* 4.28 *output. The coefficient must have a* 0.16 *representation, which is not possible. We will need to reduce the number of fraction bits at the input to the second accumulator block by at least* 1 *in the next step.*

- *Multiplication by* 0.7 *involves a* 4.12 *input and a* 2.30 *output, requiring a* -2.18 *representation for the coefficient. This is also not possible. We will need to reduce the number of fraction bits at the input to the third accumulator block by at least* 3 *in the next step.*

In step 6 we make the accumulator input representation adjustments mentioned above. The final fixed-point representations for the accumulator inputs and outputs and for the coefficients themselves are all shown in Figure 10.

Note that the two branches with coefficients of 1 *are implemented using simple upshifts to convert between their input and output representations. If we had selected scale factors for our fixed-point representation which were not powers of* 2 *we would have been able to extract a little bit more accuracy from the representations, but these branches would have required full multipliers, which is more costly than simply shifting bit positions. In hardware implementations, converting between different fixed-point representations in the power-of-2 framework adopted here is essentially free.*

We also note that the outputs from the three accumulators require normalization shifts (right shifts, with rounding) by 16, 15 *and* 13, *respectively. These normalization shifts are the sole source of round-off error, which we shall investigate further in Section 2.5.*

2.3.1 Alternate Implementation Environments

In the above, we have considered a conventional DSP implementation environment, in which the word sizes available for multiplication and accumulation operators are fixed and generally different. When working with general purpose CPU's, there is very little difference except that the accumulation bit-depth, W_+, is usually identical to the multiplicand bit-depth, $W_\times$. This affects the number of fraction bits available for accumulation, which requires the designer to carefully balance the number of fraction bits dedicated to representing intermediate sample values, with the number of fraction bits used to represent coefficients.

In hardware implementations, normalizating downshifts are almost free and up-shifts are completely free – adding extra fraction bits to a word is simply a matter of rewiring the bits. Consider Step 6 in the design process described above, where we had to make adjustments to ensure that filter coefficient representations have sufficient integer bits. In Example 7, we found that the ideal representation for the output of the multiplier with coefficient -0.8 had 0 integer bits and 16 fraction bits. Rather than adjusting the number of fraction bits used by the accumulator, and hence all multipliers feeding into that accumulator, one could instead change the coefficient to -0.4 and shift the output of the multiplier one bit position to the left (multiplication by 2). The shift is free in

hardware, and the coefficient of -0.4 can be represented as a 0.16 fixed-point number. In the same way, we could divide the coefficient 0.7 by 8, allowing it to be represented as a -2.18 fixed-point number[3], shifting the result 3 bit positions to the left. It follows that hardware implementations can and should always have the same number of integer bits at the accumulator input and its normalized (downshifted) output, i.e., $I_i^{\text{in}} = I_i^{\text{out}}$.

Another implementation environment which is becoming increasingly important is that created by the multi-word instruction sets supported by most modern general purpose CPU's and by some DSP chips. The intel MMX instruction set is typical of these multi-word operations. As an example, MMX instructions allow four 16-bit by 16-bit multiplications to be performed simultaneously, by leveraging the high performance parallel multiplier matrix which is required for double-precision floating point multiplication. The snag is that the output from each 16×16 multiplier has only 16 bits, rather than the 31 required to effect true multiplication. The most useful variant of the operation effectively forms

$$y = \left\lfloor \frac{x \times a}{2^{16}} \right\rfloor$$

That is, the two multiplicands, x and a, are effectively multiplied to form a 31-bit result, whose 16 least significant bits are immediately discarded. In this operation, there is no opportunity to implement accumulators with higher precision than the multiplicands, nor is the output y rounded to the integer nearest to $\frac{x \times a}{2^{16}}$. As a result, y is both a scaled and a biased version of the true product. All of these effects can be appropriately compensated by careful implementation, but we do not have space here to discuss such matters further.

2.3.2 Rethinking the Gain Factors

In the above design procedure, we computed BIBO gains, G_i, between the filter input and each of the accumulator outputs. These gains represent worst-case expansions in the dynamic range of an arbitrary input seqence, $x[n]$. The worst case expansions are invariably obtained when $x[n]$ alternates between the two extremes of its own dynamic range. Specifically, $x[n] = A \operatorname{sign}(h[K-n])$ where A is the maximum possible value for $|x[n]|$, $\operatorname{sign}(h)$ is 1 if h is positive and -1 if h is negative, and K is an arbitrary integer.

These worst-case gains might not actually be observed in reality, especially if we know that not all input signals, $x[n]$, are actually possible. When implementing filters for narrow-band signals, $x[n]$, it is common to assume that the worst-case expansion is given by

$$G_i = \max_{\omega \in \Omega} \left| \hat{h}_i(\omega) \right| = \max_{\omega \in \Omega} |\, H_i(z)|_{z=e^{j\omega}}|$$

where Ω is the set of frequencies which are of interest. In this case, G_i, is the maximum gain in the amplitude of any pure sinusoid in the frequency range of

[3] -2 integer bits simply means that the first two bit positions after the binary point are never used.

interest. It can be sufficiently smaller than the BIBO gain, leading to more fraction bits in the fixed-point representations and hence less round-off error. There is, however, some risk that the representations may have insufficient dynamic range and hence overflow.

In simple two's complement arithmetic, overflow is a serious concern. Consider, for example, what happens if a 3.5 accumulator representation overflows with the value 4, representing it as 100.00000, which is equal to -4. Overflow in two's complement arithmetic causes the numerical representation to "wrap-around" leading to massive errors. For this reason, if the BIBO gains are not used, the implementation should employ saturating arithmetic. Saturating arithmetic checks for overflow and selects the representable value which is closest to the true out-of-range value.

Saturating arithmetic comes with some of its own drawbacks which are worth noting. Firstly, it is somewhat more expensive to implement a saturating accumulator than a simple two's complement accumulator. Secondly and more significantly, a saturating accumulator typically saturates the results of each incremental addition. Consider, for example, the central accumulator in Figure 10, which has 3 inputs. Depending on the order in which the inputs are added, the results produced by a saturating accumulator may be different.

Moreover, when designing the fixed-point representations for a saturating accumulator one must be careful to ensure that each incremental output from the accumulator (after adding each new quantity into the total) can be represented without overflow, under the conditions for which the filter is being designed. By contrast, with regular two's complement addition, it is sufficient to ensure that the accumulator has sufficient precision to accommodate the final result, in which case intermediate overflow bits will be guaranteed to cancel. This can detract from some of the savings achieved by selecting less gains, G_i, which are less conservative than the BIBO gains.

2.4 Coefficient Quantization

The design procedures outlined above are intended to assign as many fraction bits as possible to the representation of filter coefficients. Nevertheless, most filter designs yield irrational filter coefficients which cannot be represented exactly with any finite number of bits. We refer to the process of coercing filter coefficients to implementable values as "coefficient quantization." In general, the filter which is implemented will have slightly different poles and zeros to those which were originally intended. As we have already mentioned the impact of coefficient quantization can be very dependent on the implementation structure. The following examples are intended to clarify this point.

Example 8 *Consider a second order all-pole filter, implemented as in Figure 11. The transfer function is* $H(z) = \frac{z^2}{z^2+b_1 z+b_2}$. *The design calls for complex conjugate poles at* $z = re^{\pm j\theta}$ *where*

$$b_2 = r^2 \text{ and } b_1 = -2r\cos\theta$$

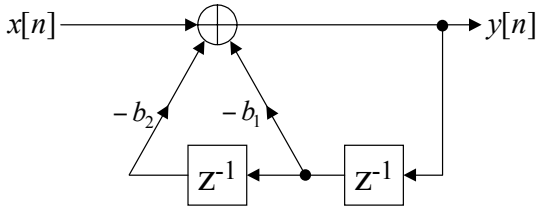

Figure 11: *Direct implementation of a second order all-pole filter.*

The two coefficients in the implementation are thus directly related to the radius, r, and the real-part, $r\cos\theta$, of the poles. Evidently, for stability we require $0 \leq b_2 < 1$ and $|b_1| < 2$. Now suppose both quantities are to be represented using signed 3-bit integers; b_1 as a 2.1 quantity, and b_2 as an unsigned 1.2 quantity. Since b_2 must be positive to get complex conjugate poles, the available pole positions, p, which can be described under these conditions have

$$|p| \in \left\{\sqrt{\frac{1}{4}}, \sqrt{\frac{2}{4}}, \sqrt{\frac{3}{4}}\right\}$$

and

$$\Re(p) \in \left\{0, \pm\frac{1}{4}, \pm\frac{2}{4}, \pm\frac{3}{4}\right\}$$

In fact, the only stable complex conjugate pole positions which can be achieved are those indicated in Figure 12. Note that only 15 different conjugate pole pairs can be realized and they are quite irregularly spaced.

Example 9 *Consider the coupled implementation of a second order all-pole filter, as shown in Figure 6. We showed previously that the pole positions associated with this realization are given by*

$$p = a \pm jb$$

so that quantizing the coefficients is exactly equivalent to quantizing the real and imaginary parts of the pole positions. If we again assume a 3-bit signed fixed-point representation for the filter coefficients, the complex conjugate pole pairs which can be achieved lie on the regular rectangular grid shown in Figure 13. In this case, there are 19 possible conjugate pole pairs.

Unfortunately, implementation structure affects many different properties of the filter, whose dependencies can be difficult to track down and optimize. These properties include memory consumption, coefficient quantization, round-off errors and limit cycle behaviour, some aspects of which are discussed further below. There is no general method to select the best structure for implementing a particular filter. Instead, several different structures can be examined, separately optimizing their parameters and representations, to determine the most appropriate structure for a given application.

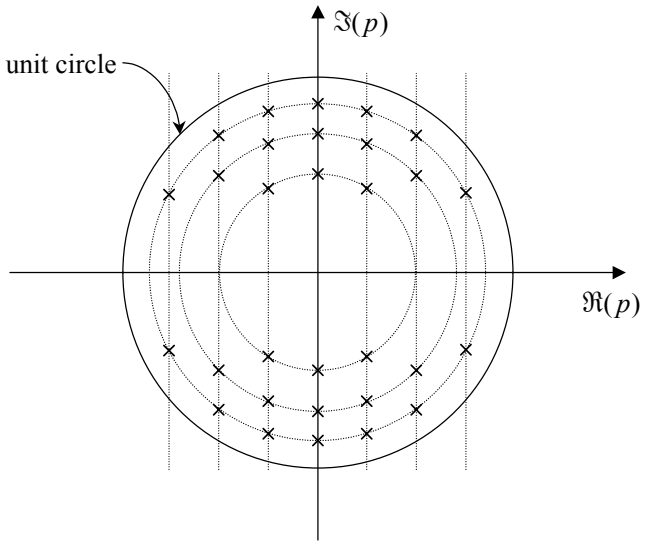

Figure 12:

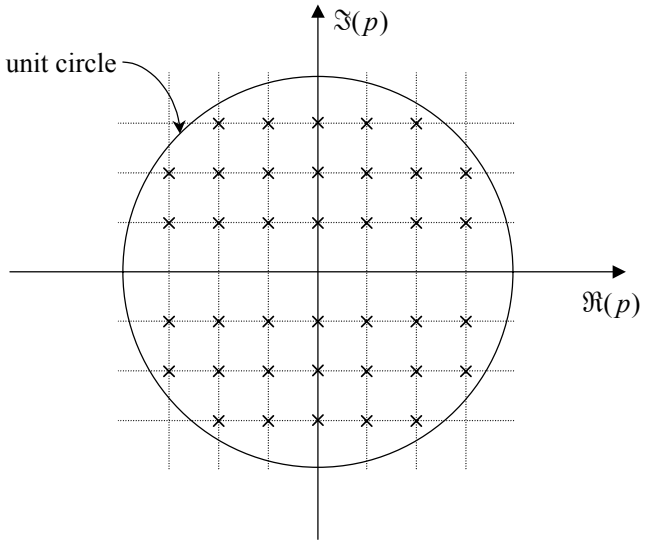

Figure 13:

2.5 Round-Off Effects

Following the implementation strategies outlined above, it should be clear that the only sources of numerical error are the normalizing downshifts which are applied at the output of each accumulator block to decrease the precision from W_+ (accumulator bit-depth) to $W_\times$ (multiplicand bit-depth). If the normalized representation at the output of the i^{th} accumulator block has F_i^{out} fraction bits, the round-off error, δ_i, is in the range

$$-2^{-\left(F_i^{\text{out}}+1\right)} \leq \delta_i \leq 2^{-\left(F_i^{\text{out}}+1\right)}$$

Moreover, under reasonable conditions this error is generally uncorrelated with the input, uncorrelated with other sources of round-off error and uncorrelated from sample to sample.

In summary, each accumulator may be modeled by a true error-free accumulator, plus an independent white noise process. Assuming that the number of normalizing downshift discards a significant number of fraction bits, the round-off error process for accumulator block i may be modeled as having zero mean[4] and a variance of

$$\sigma_i^2 = \frac{1}{2^{-F_i^{\text{out}}}} \int_{-2^{-\left(F_i^{\text{out}}-1\right)}}^{2^{-\left(F_i^{\text{out}}-1\right)}} x^2 dx = \frac{2^{-2F_i^{\text{out}}}}{12}$$

The total output noise power (variance) is given by

$$\sigma^2 = \sum_i \sigma_i^2 P_i$$

where P_i is a power gain factor, computed from the transfer function $T_i(z)$, between the input to the i^{th} accumulator block and the output of the entire filter. The power gain may be expressed in any of the following ways:

$$\begin{aligned} P_i &= \sum_{n=0}^{\infty} \left(t_i[n]\right)^2 \\ &= \frac{1}{2\pi} \int_{-\pi}^{\pi} \left|\hat{t}_i(\omega)\right|^2 d\omega \\ &= \frac{1}{2\pi} \int_{-\pi}^{\pi} T_i\left(e^{j\omega}\right) T_i\left(e^{-j\omega}\right) d\omega \end{aligned}$$

The power spectral density of the output noise process is given by

$$S(\omega) = \sum_i \sigma_i^2 \left|T_i\left(e^{j\omega}\right)\right|^2$$

[4] Actually, the round-off error process is very slightly symmetrical, so there is a very small mean offset. If, for example, the normalizing downshift discards 2 fraction bits to form a 1.3 result, rounding to the nearest value and upward where there is an ambiguity, the possible round-off errors are $+2\times 2^{-5}$, $+1\times 2^{-5}$, 0 and -1×2^{-5}. In most cases, a large number of fraction bits are discarded and the round-off error is distributed almost symmetrically about 0.

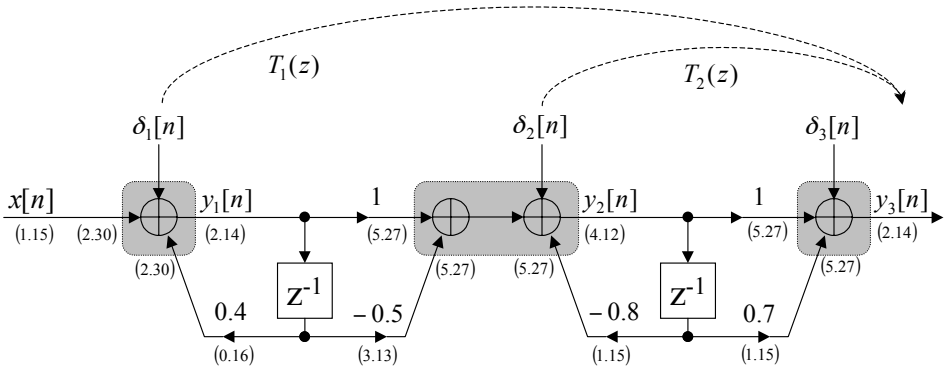

Figure 14: Round-off noise model for the filter implementation in Figure 10.

Example 10 *Consider the cascade system shown in Figure 10. There are three accumulator blocks, each with its own source of round-off noise, as shown in Figure 14. These noise processes have variances,*

$$\sigma_1^2 = \sigma_3^2 = \frac{2^{-28}}{12} = -95dB \quad and \quad \sigma_2^2 = \frac{2^{-24}}{12} = -83dB$$

The power gain factors and the noise power spectral density may be computed from the three noise transfer functions,

$$\begin{aligned} T_1(z) &= \frac{z-0.5}{z-0.4} \cdot \frac{z+0.7}{z+0.8} \\ T_2(z) &= \frac{z+0.7}{z+0.8} \\ T_3(z) &= 1 \end{aligned}$$

2.6 Limit Cycles

An interesting and potentially annoying consequence of round-off error is that the output from a recursive filter can exhibit small harmonic fluctuations even when the input signal is constant. The phenomenon is illustrated by the following simple example.

Example 11 *Consider a simple single-pole filter, with difference equation*

$$y[n] = x[n] + ay[n-1]$$

The filter's transfer function is

$$H(z) = \frac{z}{z-a}$$

and its DC gain is $\frac{1}{1-a}$*. To make the example as simple as possible, suppose that* $a = -\frac{1}{4}$*, so that the filter coefficient can be implemented exactly without any*

quantization at all. Multiplication by a is equivalent to a right-shift operation. Specifically, each new output sample is computed using the following expression

$$y'[n] = x'[n] - \left\lfloor \frac{y'[n]}{4} + \frac{1}{2} \right\rfloor$$

where the primed quantities are the integers used in the fixed-point representation of x and y and the operation enclosed by the floor delimiters, $\lfloor\rfloor$, simply rounds $y'[n]/4$ to the nearest integer. This operation may be implemented by adding 2 to $y'[n]$ and then shifting the result to the right by two bit positions.

Now suppose we supply a constant (DC) input of $x'[n] = 7$. The DC gain of the filter is $\frac{4}{5}$ so that the output should ideally converge to a constant value of $5\frac{3}{5}$, which cannot be represented exactly. The actual behaviour of the filter output is traced below:

$$\begin{aligned}
n &= 0 \quad \longrightarrow \quad y' = 7 \\
n &= 1 \quad \longrightarrow \quad y' = 7 - \left\lfloor \frac{7+2}{4} \right\rfloor = 5 \\
n &= 2 \quad \longrightarrow \quad y' = 7 - \left\lfloor \frac{5+2}{4} \right\rfloor = 6 \\
n &= 3 \quad \longrightarrow \quad y' = 7 - \left\lfloor \frac{6+2}{4} \right\rfloor = 5 \\
n &= 4 \quad \longrightarrow \quad y' = 7 - \left\lfloor \frac{5+2}{4} \right\rfloor = 6
\end{aligned}$$

Evidently, the filter output will continue to oscillate between 5 and 6. These oscillations are dependent on the DC level of the supplied input and they represent tones in the filter output which are unrelated to the frequency content of the input signal. If insufficient bits are used in the numerical implementation, these tones can become audible or otherwise perceptible in the filter output.

2.7 Floating-Point Arithmetic

Up until this point, we have considered only fixed-point arithmetic. Floating point implementations are conceptually much simpler, since the floating point processor automatically optimizes the numerical representation of every arithmetic result so as to minimize round-off error. Of course, the price paid for this is that floating point arithmetic is significantly more expensive to implement.

A floating point representation for the real-valued quantity, x, involves two integer valued quantities, e (a signed exponent), and m (an unsigned M-bit mantissa). The relationship is given by

$$|x| = 2^e \left(1 + 2^{-M} m\right)$$

Note that the sign of x is represented by an extra binary digit.

Most floating point processors employ either the IEEE single precision representation or the IEEE double precision representation. The single precision

representation uses 32 bits in total: one sign bit; $M = 23$ mantissa bits; and 8 exponent bits. This allows exponents in the range -128 to $+127$ and a relative accuracy of

$$\frac{\delta x}{|x|} \leq 2^{-(M+1)}$$

The double precision representation uses $M = 53$ mantissa bits and 10 exponent bits, with a 64-bit word size.

When floating point representations are employed, round-off errors are often much smaller than in the fixed-point case, since the number of fraction bits (determined by the exponent, e) is dynamically adjusted after each computation is performed, so as to maximize the accuracy of the representation. This adjustment process can add significant complexity, particularly to additions. Consider, for example, the addition of two quantities, 3.708 and -3.707. Each quantity is represented with an exponent of $e = 1$, but their sum requires an exponent of $e = -11$. In general, additions require extensive shifting of the mantissa bits both before and after the binary addition of their contents. This shifting is either performed sequentially (can require many clock cycles) or in parallel, using a "barrel shifting network" (can occupy a lot of silicon area on the chip).

In either case, the lesson to be learned is that floating point additions are much more complex than fixed-point additions and they can actually take significantly longer to execute than floating point multiplications. Moreover, round-off error may be introduced whenever two numbers are added, and whenever two numbers are multiplied, making it much more difficult to trace the effects of round-off error through a system for modeling purposes.