BY MICHAEL CHRISTENSEN • UNIVERSITY OF FLORIDA,
AND FRED J TAYLOR • UNIVERSITY OF FLORIDA AND THE ATHENA GROUP INC

Fixed-point-IIR-filter challenges

IIR FILTERS CAN MEET HIGHLY SELECTIVE MAGNITUDE-FREQUENCY-RESPONSE SPECIFICATIONS WITH A LOW-ORDER APPROACH. AS SUCH, AN IIR IS OFTEN THE TECHNOLOGY OF CHOICE IN REALIZING FREQUENCY-SELECTIVE TONE DETECTORS, NARROWBAND SPECTRAL FILTERS, NOISE REJECTERS, AND DIGITAL CONTROLLERS. THEIR DESIGN, HOWEVER, ENTAILS SOME UNIQUE CHALLENGES.

The historic role of IIR (infinite-impulse-response)-filter-design software is to translate a set of frequency-domain design specifications into a transfer function that is based on recognized IIR-filter models. You can use any of dozens of commercially available software packages to synthesize a transfer function from a set of user specifications. The SPT (signal-processing toolbox) and FDATool (filter-design and -analysis tool) in The Mathworks' (www.mathworks.com) Matlab, for example, contain many of the objects you need to synthesize an IIR-transfer function. These functions include Matlab's Butterworth, Chebyshev I, Chebyshev II, elliptic, Burg-AR (autoregressive), covariance-AR, and Yule-Walker-AR functions. You use the first four deterministic filter classes to synthesize a classic fixed-coefficient filter based on user-specified passband-critical frequencies and maximum attenuation and stopband-critical frequencies and minimum attenuation. The last three design methods synthesize AR, fixed-coefficient, feedback-only IIR filters in terms of measured or desired input/output spectral responses.

The choice of which filter model to use is generally not the issue. Often, the designer specifies or selects the IIR type from a restricted list. For performance and cost reasons, designers generally prefer fixed-point approaches over floating-point instantiations. Unfortunately, fixed-point designs are highly susceptible to a range of degrading finite-word-length effects,

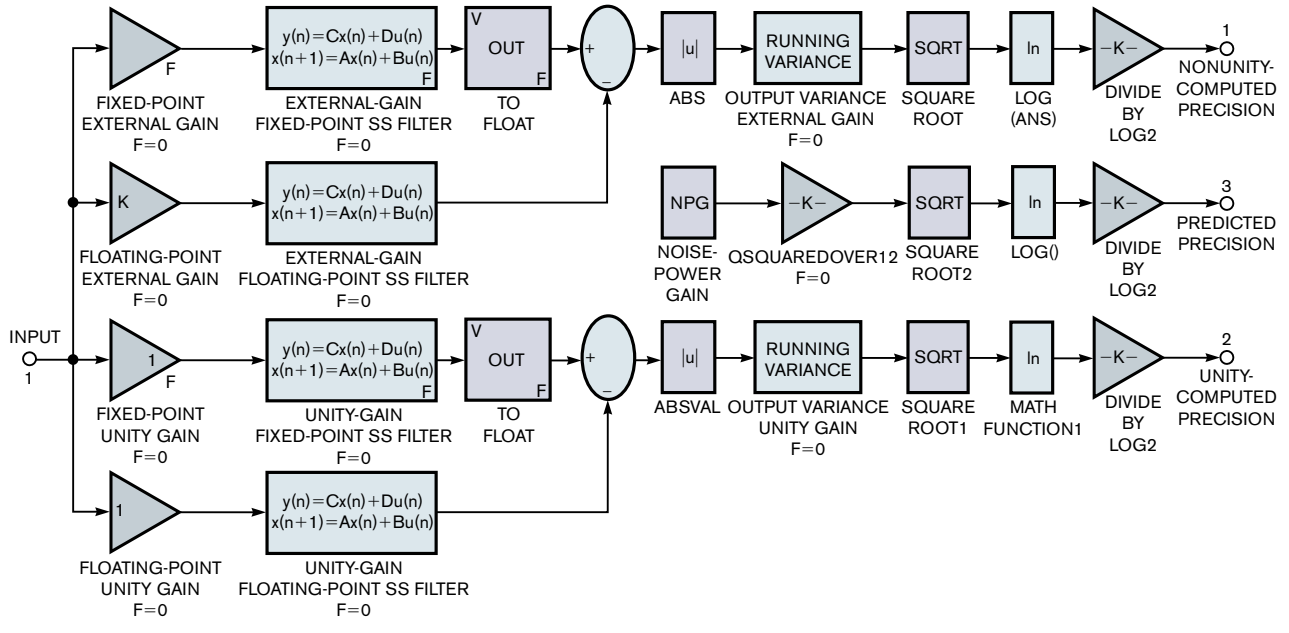

Figure 1 A Simulink simulator tests the behavior of a 16-bit filter for fractional precisions ranging from F∈[0,15].

in some cases rendering a design unusable. As a result, you must take extreme care to ensure that the final outcome meets the target design requirements.

FINITE-WORD-LENGTH EFFECTS

The goal of a fixed-point IIR-filter design is to maximize filter performance and minimize finite-word-length effects, which include register overflow and arithmetic-round-off errors. The more egregious error is register overflow, which occurs whenever a filter's dynamic-range requirements exceed the dynamic-range limitations of a fixed-point register. Arithmetic-round-off errors result from imprecise arithmetic, which in turn reduces precision. Register overflow can cause a system to behave in an unpredictable, nonlinear manner, producing potentially large runtime errors. In the absence of register overflow, a system behaves linearly but possibly with degraded precision due to the accumulation of various arithmetic errors accumulating within a filter. A filter-design engineer should be able to quantify and control the effects of arithmetic errors to ensure that the final outcome meets some minimum precision requirement. This process begins with making design choices. Although the filter type may be non-negotiable, the choice of architecture normally is negotiable. The choice of architecture is a critical factor in controlling finite-word-length effects.

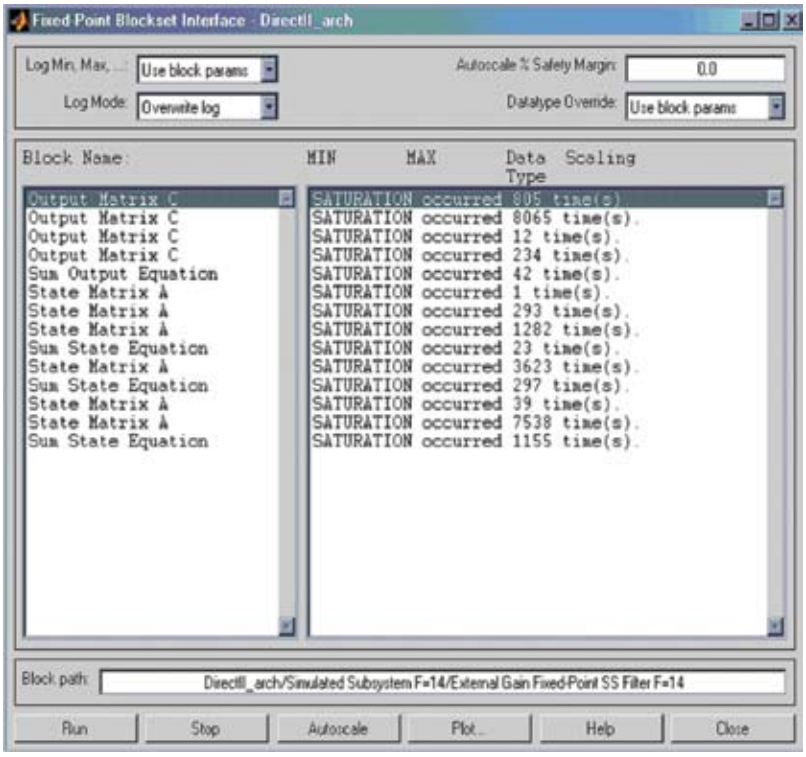

Figure 2 When the simulation is complete, you can use the fixed-point GUI to examine runtime-saturation effects.

REGISTER OVERFLOW

The most serious finite-word-length effect is register overflow. Register overflow introduces large nonlinear distortions into a system's output, often rendering a filter useless. A filter designer must eliminate or control the effects of runtime register overflow. Some standard techniques are available to mitigate this problem. One effective means of controlling fixed-point-overflow errors is to perform all arithmetic using a two's complement arithmetic unit. Two's complement possesses the important modulo(2^n) property that ensures that the sum of a string of two's complement numbers is a valid two's complement outcome, ensuring that the accumulator does not overflow. Alternatively, designers can use a saturating-arithmetic approach. A saturating-arithmetic unit "clamps" a register's contents at the register's extreme values if overflow occurs. Even with saturating arithmetic, the effect of register overflow creates serious errors.

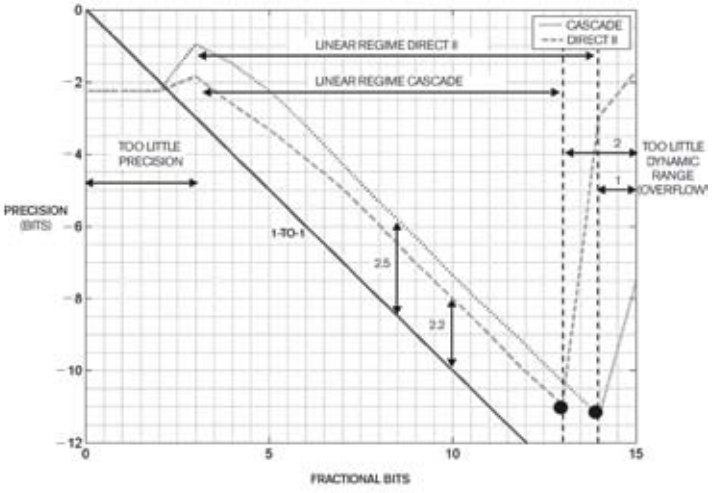

Figure 3 With the simulated output, you can partition the results.

Scaling the input to a lower level can eliminate register-overflow conditions. Experimentally determining the required scale factor can be a tenuous approach. Furthermore, the test inputs may not represent an input-worst-case event and therefore may underestimate scaling needs. Scaling reduces the precision of the input, which in turn reduces output precision. Another means of eliminating runtime overflow is to use extended-precision arithmetic and registers. Extended-

(continued on pg 120)

IIR-FILTER ARCHITECTURES

You define the physical implementation of a digital filter in terms of its architecture. "Architecture" refers to how a designer builds a filter using primitive building-block elements, such as shift registers, memory, multipliers, and adders. Many DSP-system engineers are generally aware of only one or two possible filter architectures. However, many architectural choices exist, each carrying relative advantages and disadvantages. Some architectural choices are application-specific, and others have general-purpose implications. Some provide better control of finite-word-length effects, and others emphasize reduced complexity and increased speed. The more common architectures are Direct I, Direct II, cascade, parallel, normal cascade, normal parallel, lattice/ladder, wave, and biquadratic.

The two most popular architectural choices are Direct II and cascade. However, all can implement a given transfer function, H(z). If you build an IIR (infinite-impulse-response) filter using floating-point arithmetic, then all architectures would have identical input/output behavior. This behavior is not the case when you implement designs using fixed-point arithmetic. Fixed-point arithmetic gives rise to finite-word-length effects that can degrade a filter's performance. The choice of architecture, in turn, strongly influences the severity of these errors. It is therefore essential that fixed-point-IIR-filter developers know how to exercise control over the design environment and minimize the impact of these errors on the outcome.

An IIR filter's transfer function, H(z), which you produce using The Mathworks' Matlab or another commercial software tool, quantifies only an IIR filter's input/output behavior. A transfer function, unfortunately, does not quantify the filter's internal workings in which errors emerge and accumulate. The filter's internal structure, or architecture, includes state variables. You use the state-variable model to audit the information entering, exiting, and residing within a digital filter. Equation A illustrates a state-variable model of a single-input, single-output, Nth-order IIR (Figure A). The state-model is $x[k+1]=Ax[k+1]=x[k]+bu[k]$, and the output model is $[yk]=c^T x[k]+du[k]$, where x[k] is the N-dimension state vector of state variables, u[k] is the input, and y[k] is the output. The other elements of the state-variable model are an N×N feedback matrix A, a 1×N input vector b, an N×1 output vector c, and a scalar-direct input/output-path gain d. The filter's N registers store x[k] and x[k+1] on the next clock cycle. Because state-variable models model the internal behavior of digital filters and this information is critical in managing register overflow, understanding the relationship between the state model and a given architecture is important.

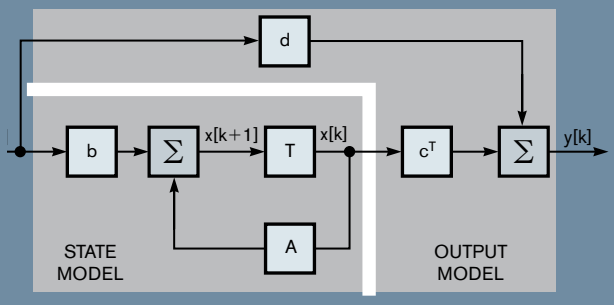

Figure A Understanding the relationship between the state model and a given architecture is important.

The Direct II architecture is a common IIR form. It is based on interpreting an Nth-order IIR transfer function, H(z), as the following equation shows:

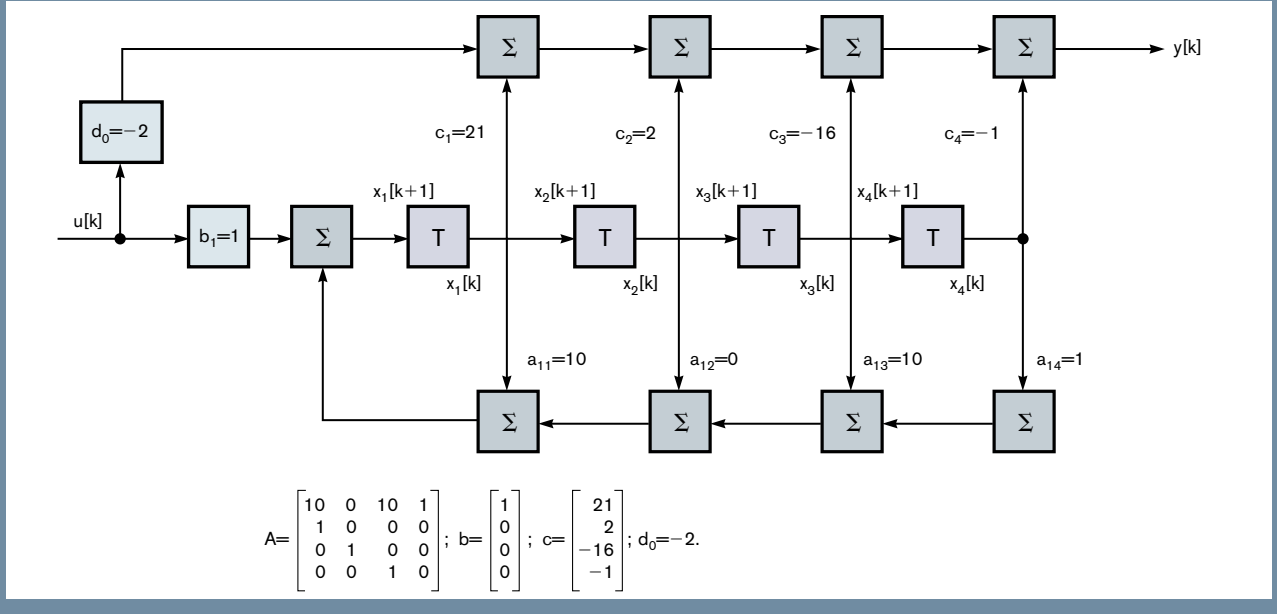

Figure B This Direct II state four-tuple S=[A,b,c,d] induces this architecture.

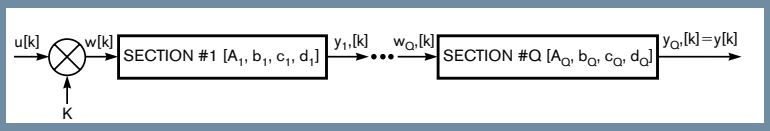

Figure C You use the basic cascade architecture to implement a transfer function.

$$H(z)=K\frac{N(z)}{D(z)}=\frac{b_0+b_1z^{-1}+\cdots b_Nz^{-N}}{a_0+a_1z^{-1}+\cdots a_Nz^{-N}}$$
$$=K\left(\frac{b_0}{a_0}+\frac{(b_1-b_0a_1/a_0)z^{-1}+\cdots+(b_N-b_0a_N/a_0)z^{-N}}{a_0+a_1z^{-1}+\cdots+a_Nz^{-N}}\right)$$
$$=K\left(\frac{b_0}{a_0}+\frac{c_1z^{-1}+\cdots+c_Nz^{-N}}{a_0+a_1z^{-1}+\cdots+a_Nz^{-N}}\right)$$
$$=K\left(d_0+C(z)\left(\frac{1}{D(z)}\right)\right). \quad \textbf{(A)}$$

The following equation yields the Direct II state-variable four-tuple (A, b, c, d):

$$A=\begin{bmatrix}-a_1 & \cdots & -a_{N-2} & -a_{N-1} & -a_N\\ 1 & \cdots & 0 & 0 & 0\\ \vdots & \ddots & \vdots & \vdots & \vdots\\ 0 & \cdots & 1 & 0 & 0\\ 0 & \cdots & 0 & 1 & 0\end{bmatrix};\ b=\begin{bmatrix}1\\0\\\cdots\\0\\0\end{bmatrix}. \quad \textbf{(B)}$$

The (i,j) element of A defines the path gain between state $x_j[k]$ and $x_i[k+1]$, b_i is the path gain between the input and $x_i[k+1]$, c_i is the path gain between $x_i[k]$ and the output, and d_0 is the gain of the direct input/output path. One caveat you should know about is that Matlab state-variable programs make state assignments in a reverse order to what you generally find in signal-processing literature and textbooks.

You can convert the transfer function H(z) into a Direct II form using the Matlab functions TF2SS or ZP2SS. The function TF2SS converts a transfer function, H(z)=KN(z)/D(z), into a Direct II state form using the syntax [A,B,C,D]=TF2SS(NUM,DEN). Similarly, the function ZP2SS converts transfer function H(z) having zeros (Z), poles (P), and input-scale factor (K), into a Direct II state form using the syntax [A,B,C,D]=ZP2SS(Z,P,K).

Consider the fourth-order transfer function H(z):

$$H(z)=\frac{-2+z^{-1}+2z^{-2}+4z^{-3}+z^{-4}}{1+10z^{-1}+0z^{-2}-10z^{-3}-z^{-4}}$$
$$=-2+\frac{21z^{-1}+2z^{-2}-16z^{-3}-z^{-4}}{1+10z^{-1}+0z^{-2}-10z^{-3}-z^{-4}}, \quad \textbf{(C)}$$

which you factor using Equation A. From this factorization, you can construct a database for a Direct II filter. Specifically, [A,B,C,D]=tf2ss([−2 1 2 4 1],[1 10 0 −10 −1]).

A=−10 0 10 1 1 0 0 0 0 1 0 0 0 0 10 (Direct II).

B=1 0 0 0.

C=21 2 −16 −1.

D=−2.

Figure B graphically interprets this Direct II state four-tuple S=[A,b,c,d] and its induced architecture.

Figure C shows another important IIR form, the cascade architecture. You use the basic cascade architecture to implement a transfer function of the form:

$$H(z)=K\prod_{i=1}^{Q}H_i(z). \quad \textbf{(D)}$$

Assume that $H_i(z)$ is a first- or second-order subfilter, which you define in terms of real coefficients. You define first-order subfilters in terms of real poles and zeros of H(z). You define second-order subfilters by combining

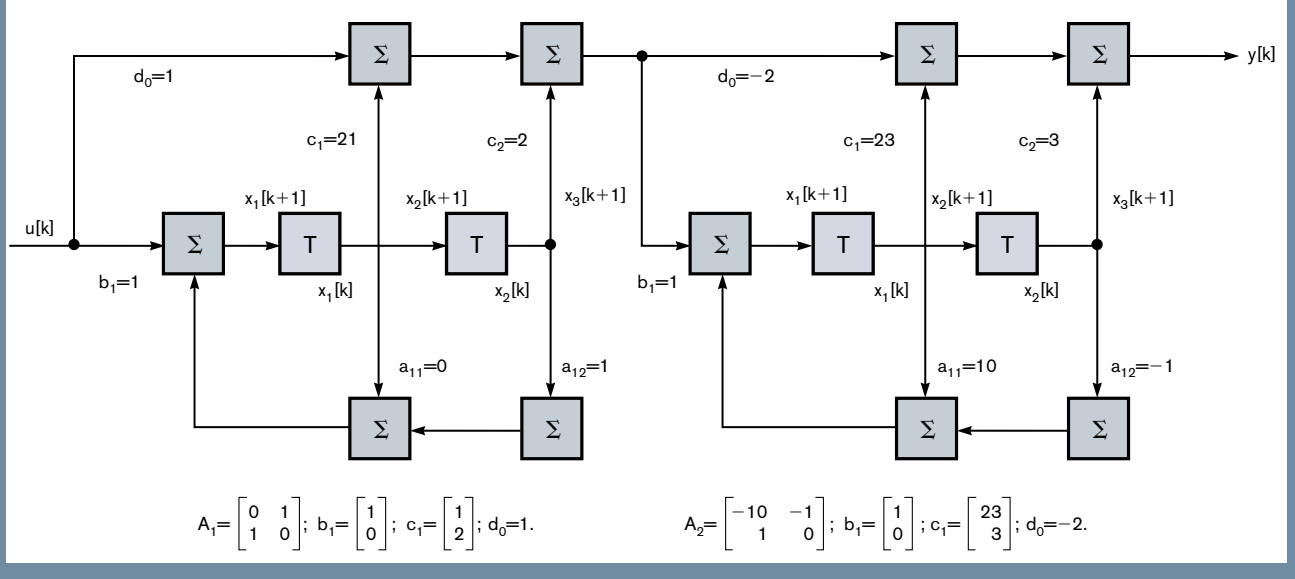

Figure D This cascade filter comprises two second-order Direct II sections.

complex poles and zeros and their complex conjugate pairs to form filter sections having only real coefficients. You can define the basic first- and second-order sections in terms of biquadratic or Direct II structures. The principal difference is that the Direct II architecture possesses a state-variable description, whereas the biquad does not. Direct II implementations have been increasingly gaining favor because of this feature.

As a general rule, filters pair zeros with the closest poles. This proximity-pairing strategy generally results in filter design that more uniformly distributes the subfilter gains across all filter sections, which is a desirable trait. Other pairing strategies can result in a few subsystems having excessively large dynamic-range requirements and others having small gains. This disparity creates a precision-allocation problem that can compromise overall system performance.

Matlab contains a collection of programs that relate to cascade-filter implementation. The function tf2sos converts digital-filter-transfer-function data to a set of second-order sections having the form:

$$\text{SOS} = \begin{bmatrix} b_{01} & b_{11} & b_{21} & 1 & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & 1 & a_{12} & a_{22} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ b_{0L} & b_{1L} & b_{2L} & 1 & a_{1L} & a_{2L} \end{bmatrix}, \quad \textbf{(E)}$$

where the ith row of the array sos specifies the coefficients of the ith subfilter:

$$H_i(z) = K\frac{B_i(z)}{A_i(z)} = K\frac{b_{0i}+b_{1i}z^{-1}+b_{2i}z^{-2}}{a_{0i}+a_{1i}z^{-1}+a_{2i}z^{-2}}. \quad \textbf{(F)}$$

The Matlab program zp2sos converts a transfer function, H(z), in terms of its zeros, poles, and gain, into an Lx6 sos array. The program sos2ss maps second-order-filter sections into a Direct II state-space form, and the program sos2tf converts a collection of second-order-filter sections into an overall transfer function. If a filter section is first-order, the coefficients b2i and a2i are zero. Finally, the program sos2zp converts second-order-filter sections into a zero-pole-gain form.

Consider a transfer function $H(z)=H_1(z)H_2(z)$, where:

$$H(z) = \frac{-2+z^{-1}+2z^{-2}+4z^{-3}+z^{-4}}{1+10z^{-1}+0z^{-2}-10z^{-3}-z^{-4}} = H_1(z)H_2(z);$$
$$H_1(z) = \frac{1+z^{-1}+z^{-2}}{1-z^{-2}} = 1+\frac{z^{-1}+2z^{-2}}{1-z^{-2}}; \text{ and} \quad \textbf{(G)}$$
$$H_2(z) = \frac{-2+3z^{-1}+z^{-2}}{1+10z^{-1}+z^{-2}} = -2+\frac{23z^{-1}+3z^{-2}}{1+10z^{-1}+z^{-2}}.$$

You can reduce the subfilters $H_1(z)$ and $H_2(z)$ to two cascaded, second-order Direct II sections using Matlab. The Matlab representation of the filter sections is sos=[1 1 1 1 0 −1; −2 3 1 1 1 0 1]; {$H_1(z)$, $H_2(z)$}. From this database, sos2tf can map the definition of the second-order section to the original transfer function as follows:

```
sos=[1 1 1 1 0 −1; −2 3 1 1 1 0 1].
[b,a]=sos2tf(sos).
b=−2 1 2 4 1.
a=1 10 0 −10 −1.
```

The vectors b and a are the coefficients of transfer function H(z). Finally, using sos2ss, you can convert the individual second-order sections into a Direct II state-variable form:

```
sos1=[1 1 1 1 0 −1]; [A,B,C,D]=sos2ss(sos1).
```

(continued on pg 118)

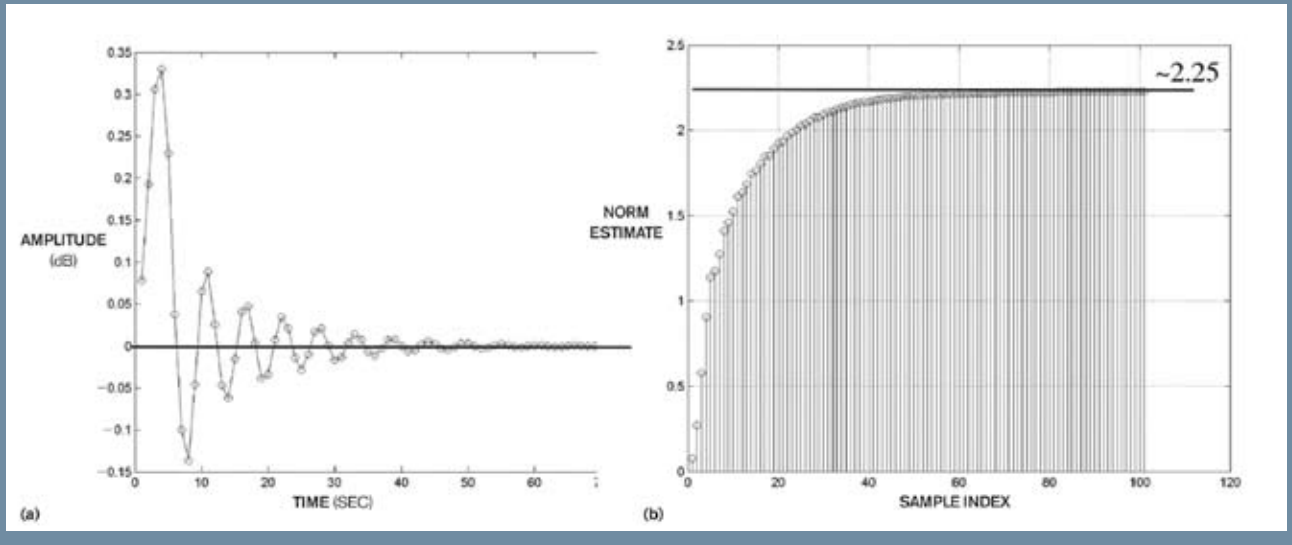

Figure E Measuring a Direct II filter's impulse response at a shift register (a) and its L_1-norm estimate (b) shows a worst-case gain of 2.25.

(continued from pg 116)

```
A=0 1 1 0.
B=1 0.
C=1 2.
D=1 1.
sos2=[-2 3 1 1 10 1].
[A,B,C,D]=sos2ss(sos2).
A=-10 -1 1 0.
B=1 0.
C=23 3.
D=-2.
```

Figure D shows the resultant cascade filter, comprising two second-order Direct II sections.

Using The MathWorks' FDATool (filter-design and-analysis tool), you can implement an eighth-order Chebyshev II IIR lowpass filter using a sampling frequency of 100 kHz, an attenuation frequency of 20 kHz, and a stopband attenuation of 30 dB. Using Matlab's architectural tools, you can implement the designed filter as a Direct II and cascade filter. First, compute the n-state-determined impulse-response vector and its L_1 norm, $\|h_i[k]\|_1$. Figure E shows the production of $\|h_1[k]\|_1$ for the Direct II case. You then use the state-determined impulse response to compute the worst-case gain of 2.25. Referring to Figure 2, note that all the shift registers are chained together. Therefore, the dynamic-range requirement of the first shift register is identical to that of all the other shift registers. This situation indicates that the Direct II shift registers need an additional $\log_2(2.25) \sim 1.17$ bits.

Figure F shows the L_1 norms of the state-determined impulse responses of the cascade IIR, which you can analyze and use to study the IIR. The largest L_1 norm is in the fourth subfilter and is approximately 1.8 ($\log_2(1.8) \sim 0.85$ bits), which is less than the maximal L_1 norm of the Direct II filter model.

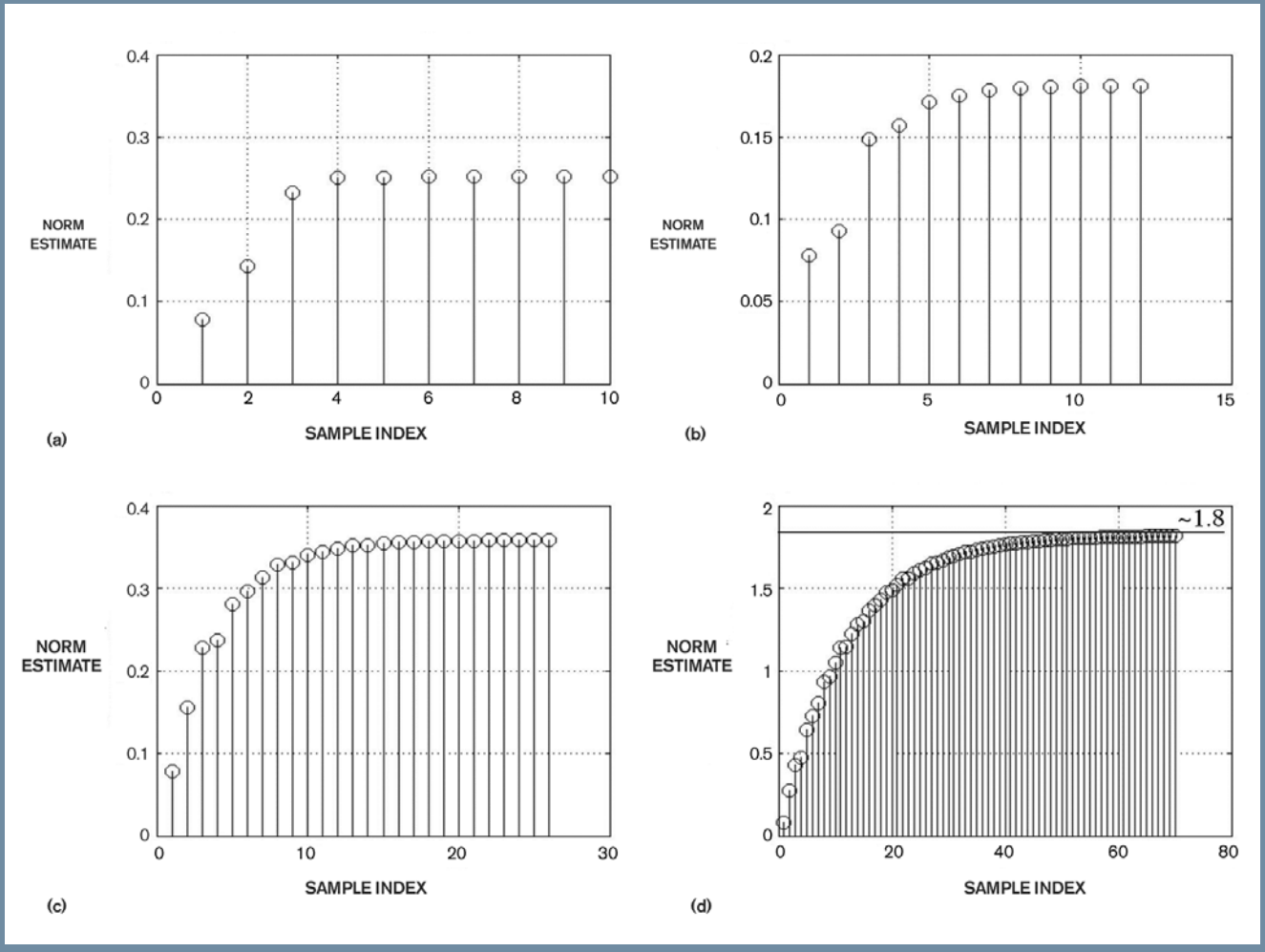

Figure F You can analyze the L_1 norms of the state-determined impulse responses of the cascade IIR to study the Direct II IIR in Stage 1 (a), Stage 2 (b), Stage 3 (c), and Stage 4 (d).

(*continued from pg 112*)

precision registers provide additional head room that allows the filter to store and preserve a system's states without introducing saturation errors.

The ideal method of overcoming the threat of runtime overflow is to determine the worst-case filter gain, which you measure at each register or state location. Mathematically, the worst-case gain measured at the ith shift-register location is:

$$G_{MAX:i} = \sum_{m=0}^{\infty} |h_i[m]| \le 2^I, \quad (1)$$

where $h_i[m]$ is the impulse response at the output of the ith shift register—that is, the ith state location. $G_{MAX:i}$ is the L_1 norm of the ith state register. The ith state norm that **Equation 1** defines states that I integer bits of precision are necessary to ensure that the ith state does not produce a run-time-overflow error. You can generate the impulse response, $h_i[m]$, using a state-variable model and general-purpose digital computer. You define **Equation 1** in terms of a vector-valued impulse response of the form $h[m+1]=Ah[m]+b\delta[m]$, where the ith element of the n-dimensional vector, h[m], is $h_i[m]$. A potential problem, however, arises when you note that **Equation 1** requires an infinite sum, which is unrealistic. Another approach, however, is available.

You can assume that the system under study is asymptotically stable. This assumption ensures that the impulse-response vector, h[m], essentially converges to zero by a finite-sample index $m \le M$. Because M is finite, you can compute the impulse response at each shift register and attendant L_1-norm $G_{MAX:i}$ in finite time. You can use Matlab's norm function to compute the L_1 norms, which you can then use to establish the dynamic-range requirements of the state registers. The following example demonstrates this concept.

SIMULINK HOSTS A PLETHORA OF FEATURES TO SUPPORT DESIGN ANALYSIS. FIXED-POINT FILTERS, FOR EXAMPLE, OFFER A MEASURE FOR THE DYNAMIC-RANGE NEEDS FOR INTERNAL CALCULATIONS.

The source of serious arithmetic error that an IIR produces involves fixed-point MAC (multiply-accumulate) or SAXPY (S=AX+Y) calls. You can round data at a number of locations within a MAC stream. It has become commonplace to employ extended-precision accumulators that can accept full-precision products from the multiplier and sequentially accumulate the products with sufficient head room to preclude runtime-accumulator overflow. Once you have summed the full-precision products, you can then round them to the filter's basic word length—16 bits, for example. Each rounding introduces an error having a mean value of zero and a variance of $\sigma^2=Q^2/12$, where Q is the quantization-step size and has a value of $Q=2^{-F}$ and where F denotes the fractional precision you assign to a data word. For example, in a Texas Instruments' (www.ti.com) Q.15 environment, F=15 bits.

Using this model, you can theoretically predict the effect of arithmetic-rounding errors by computing the NPG (noise-power gain) between a noise-injection point and the output. The noise-injection points are normally the state registers (see **sidebar** "IIR-filter architectures" on pg 114). In this paradigm, assume that the input to the ith state register contains m_i round-off-error sources, in which each source has a mean of zero and a variance of $\sigma^2=Q^2/12$. IIR filters are so treacherous because these errors recirculate through the filter, building up noise power over time and reducing the output SNR. You can conceptually compute the noise-power gain by tracing the signal-power path between a state-shift register and the output. By defining $NPG_i, i\epsilon[1,n]$ to be the noise-power gain associated with the ith state, the output-noise error variance then becomes:

$$\sigma^2 = \frac{Q^2}{12}\sum_{k=1}^{n} m_i NPG_i = \frac{Q^2}{12}(NPG). \quad (2)$$

You can determine the filter's noise gain in bits using $NG_2=\log_2(\sqrt{NPG})$. NG_2 is an estimate of the statistical degradation of the IIR filter's output in bits due to accumulated round-off errors in the filter. In practice, you can estimate the noise-power gain using fixed-point simulation.

You can use The MathWorks' Simulink to perform an end-to-end fixed simulation of an eighth-order Chebyshev II IIR lowpass filter. The basic filter data's word length is 16 bits, and the filter comes with a full-precision multiplier and an extended-precision accumulator. A Simulink simulator tests the behavior of a 16-bit filter for fractional precisions ranging from $F\epsilon[0,15]$ (**Figure 1**). The key architectural choices defining the simulation are a data-word length, N, of 16 bits; a fractional precision of $F\epsilon[0:15]$ bits; an input-data format of $x[k]\epsilon[N:F]$ bits (N:F denotes an N-bit word with F fractional bits of precision); an output-data format of $y[k]\epsilon[N:F]$ bits; a coefficient-data format of $c_k\epsilon[N:F]$ bits; multiplier datapaths of $16\times16\rightarrow32$ bits; Direct II accumulator datapaths of $32+(32+N_{DII})\rightarrow32+N_{DII}$ bits ($N_{DII}\ge\log_2(2.25)\sim1.17$ bits; (**sidebar Figure E**, pg 116) and cascade-accumulator datapaths of $32+(32+N_C)\rightarrow32+N_C$ bits ($N_C\ge\log_2(1.8)\sim0.8$ bits) (**sidebar Figure F**, pg 118). [N:F] denotes an N-bit word with F fractional bits of precision. Assume that an extended-precision accumulator has additional head room. For the Direct II filter, the head-room requirement is $N_{DII}=2\ge\log_2(2.25)$ bits. For a cascade filter, the head-room requirement is $N_C=1\ge\log_2(1.8)$ bits. Upon accumulation, assume that data rounds to a signed 16-bit word having F fractional bits of precision, where $F\epsilon[0,15]$ bits. Simulink also hosts a plethora of features to support design analysis. Fixed-point filters, for example, offer a measure for the dynamic-range needs for internal calculations. The command sfrac(N,I) creates an N-bit structure having a signed fractional form with I integer bits. To illustrate, you can model

a 16-bit system with no fractional precision using the structure sfrac(16,15). When the simulation is complete, you can use the fixed-point GUI to examine runtime-saturation effects (**Figure 2**). The input-forcing function in the example is a 2048-sample, unit-bound, uniformly distributed random signal that emulates a worst-case input.

Figure 3 shows the simulated output, from which you can partition the results as those having too little precision due to having too few fractional bits of accuracy; those in the linear regime, meaning that they have sufficient dynamic range to inhibit runtime overflow and sufficient fractional precision to eliminate traumatic round-off errors; and those having too little dynamic range due to having too few integer bits, resulting in too small a dynamic range, and ultimately resulting in a plethora of runtime-overflow errors.

The Direct II architecture exhibits overflow contamination beginning at two integer bits, as the L_1-norm analysis predicts. Similarly, as the analysis predicts, the cascade filter began exhibiting register overflow at one integer bit. Moreover, the Direct II has slightly better statistical precision than the linear input/output operating range, which the analytical study predicts. In the linear region, the analysis predicts that the cascade architecture is about 0.3 of a bit inferior to a Direct II. The simulation suggests that the optimal cascade filter should carry a [16:14] format, resulting in a solution having statistically about 11.5 fractional bits of precision. The simulation also suggests that the Direct II filter should carry a [16:13] format, resulting in a solution having statistically about 11 fractional bits of precision.

The system always used a worst-case or nearly worst-case input to mimic the most severe conditions. Analyzing the system using an impulse or sinusoidal test signal produces different, erroneous, and ultimately inconclusive results. You can predict the optimal operational point of a fixed-point IIR using simulation to produce results that are consistent with classic analytical techniques. You can exploit the existence of predefined blocks by invoking the Simulink library browser from the Launch Pad in Matlab. EDN

REFERENCES

1. Cavicchi, Thomas J, *Digital Signal Processing*, ISBN: 0-471-12472-9, Wiley, 1999.
2. Chassaing, Rulph, *Digital Signal Processing and Applications with the C6713 and C6416 DSK*, ISBN: 0-471-70406-7, Wiley, 2005.
3. Mitra, Sanjit K, *Digital Signal Processing, Third Edition*, ISBN: 0073048372, McGraw Hill, 2006.
4. Ifeachor, Emmanuel C and Barrie W Jervis, *Digital Signal Processing: A Practical Approach, Second Edition*, Addison Wesley, 2001.
5. Oppenheim, Alan V, and Ronald W Schafer, *Digital Signal Processing*, ISBN: 0-132-14635-5, Prentice Hall, 1974.
6. Oppenheim, Alan V, and Ronald W Schafer, *Digital Signal Processing, Second Edition*, ISBN: 0-824-71357-5, Prentice Hall, 1999.
7. Taylor, Fred J, *Digital Filter Design Handbook*, Marcel Dekker, New York, 1983.
8. Taylor, Fred J, and Thanos Stouraitis, *Digital Filter Design Using the IBP PC*, ISBN: 0-824-77733-6, Marcel Dekker, 1987.
9. Taylor, Fred J, and Jon Mellott, *Hands-On Digital Signal Processing*, ISBN: 0-078-52930-1, McGraw Hill, 1998.

AUTHORS' BIOGRAPHIES

Michael Christensen is a research assistant at the University of Florida (Gainesville). As a doctoral student, he focuses his research on digital filters and machine learning. He holds bachelor's and master's degrees in engineering from the University of Florida.

Fred J Taylor is a professor at the University of Florida (Gainesville), where he researches and teaches DSP theory and practice. He holds a bachelor's degree in electrical engineering from the Milwaukee School of Engineering (WI) and master's and doctorate degrees from the University of Colorado—Boulder.