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 FDA Tool (filter-design and -analysis tool) in The Mathworks' 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, 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.

**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.

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-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:

\[
G_{\text{MAX, }i} = \sum_{n=0}^{\infty} |h_i[n]| \leq 2^i, \quad (1)
\]

where \(h_i[n]\) is the impulse response at the output of the ith shift register—that is, the ith state location. \(G_{\text{MAX, }i}\) 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 runtime-overflow error. You can generate the impulse response, \(h_i[n]\), 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 \(m \leq M\). Because M is finite, you can compute the impulse response at each shift register and attendant \(L_1\) norm \(G_{\text{MAX, }i}\) 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.

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' 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"). 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 $\text{NPG}_{i,e}[1,n]$ to be the noise-power gain associated with the ith state, the output-noise error variance then becomes:

$$\sigma^2 = \frac{Q^2}{12} \sum_{i=1}^{n} \text{NPG}_i = \frac{Q^2}{12} \text{NPG}.$$ (2)

You can determine the filter's noise gain in bits using $\text{NG}_2 = \log_2(\sqrt{\text{NPG}})$. $\text{NG}_2$ 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; a coefficient-data format of $c_k\epsilon[N:F]$ bits; multiplier datapaths of $16 \times 16 \rightarrow 32$ bits; Direct II accumulator datapaths of $32+(32+N_{\text{DII}})\rightarrow 32+N_{\text{DII}}$ bits ($N_{\text{DII}}=\log_2(2.25)\sim 1.17$ bits; (sidebar Figure E) and cascade-accumulator datapaths of $32+(32+N_{c})\rightarrow 32+N_{c}$ bits ($N_{c}=\log_2(1.8)-0.8$ bits) (sidebar Figure F). [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 $N_{\text{DII}}=2\geq \log_2(2.25)$ bits. For a cascade filter, the head-room requirement is $N_{c}=1\geq \log_2(1.8)$ 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.
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.

References
Analyze and use to study the IIR. The largest L impulse responses of the cascade IIR, which you can compute the worst-case gain of 2.25. Referring to Figure F, direct II shift registers need an additional C=23 3.

Referring to Figure D, the program sos2zp converts second-order-filter sections, which you define in terms of real coefficients. You define complex poles and zeros and their complex conjugate pairs to form filter sections. Other pairing strategies can result in a few subsystems having small gains. This disparity creates a disparity in overall system performance. If a filter section is first-factorized, you pair poles with the closest zeros of H(z). You define second-order subfilters by combining complex poles and zeros and their complex conjugate pairs to form filter sections. Direct II implementations have been increasingly gaining favor because of this feature.

As a general rule, filters pair zeros with the closest poles of H(z), as the following equation shows:

\[ H(z) = \frac{b}{a} \]

where the ith row of the array sos specifies the z-domain definition of the second-order section to the original sos format. You factor using

\[ \text{sos} = \left[ \begin{array}{c} b_1 \ 1 \\ a_1 \end{array} \right] \ 2 \ 
\left[ \begin{array}{c} b_2 \ 1 \\ a_2 \end{array} \right] \ 2 \ \text{etc.} \]

You factor using \[ \text{sos} = \left[ \begin{array}{c} 1 \ 1 \ 1 \\ 0 \ 1 \end{array} \right] \ 2 \ 
\left[ \begin{array}{c} 1 \ 1 \ 1 \\ 0 \ 1 \end{array} \right] \ 2 \ \text{etc.} \]

You factor using \[ \text{sos} = \left[ \begin{array}{c} 1 \ 1 \\ 0 \ 1 \end{array} \right] \ 2 \ 
\left[ \begin{array}{c} 1 \ 1 \\ 0 \ 1 \end{array} \right] \ 2 \ \text{etc.} \]

Importantly, you perform these errors on the outcome. Design environment and minimize the impact of these errors on the outcome. The impact of these errors on the outcome strongly influences the severity of these errors. It is extremely important to use a design environment and minimize the impact of these errors on the outcome.

The Direct II architecture is a common IIR form. It is under the understanding the relationship between the state information is critical in managing register overflow, the next clock cycle. Because state-variable models unfortunately, do not quantify the filter's internal information. The filter's internal structure, or architecture, unfortunately, does not quantify the filter's internal state assignments in a reverse order to what you find in textbooks.

The filter's input/output behavior. A transfer function, H(z), as the following equation shows:

\[ H(z) = \frac{b}{a} \]

The filter's input/output behavior. A transfer function, H(z), as the following equation shows:

\[ H(z) = \frac{b}{a} \]

Similarly, the function ZP2SS converts transfer function H(z) having zeros (Z), poles (P), and input-output path gain (K) into a Direct II state form using the syntax \[ \text{ZP2SS}(Z, P, K) \]

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 others have small gains. This disparity creates a disparity in overall system performance. If a filter section is first-factorized, you pair poles with the closest zeros of H(z). You define second-order subfilters by combining complex poles and zeros and their complex conjugate pairs to form filter sections. Direct II implementations have been increasingly gaining favor because of this feature.

As a general rule, filters pair zeros with the closest poles of H(z), as the following equation shows:

\[ H(z) = \frac{b}{a} \]

where the ith row of the array sos specifies the z-domain definition of the second-order section to the original sos format. You factor using

\[ \text{sos} = \left[ \begin{array}{c} b_1 \ 1 \\ a_1 \end{array} \right] \ 2 \ 
\left[ \begin{array}{c} b_2 \ 1 \\ a_2 \end{array} \right] \ 2 \ \text{etc.} \]

You factor using \[ \text{sos} = \left[ \begin{array}{c} 1 \ 1 \ 1 \\ 0 \ 1 \end{array} \right] \ 2 \ 
\left[ \begin{array}{c} 1 \ 1 \ 1 \\ 0 \ 1 \end{array} \right] \ 2 \ \text{etc.} \]

You factor using \[ \text{sos} = \left[ \begin{array}{c} 1 \ 1 \\ 0 \ 1 \end{array} \right] \ 2 \ 
\left[ \begin{array}{c} 1 \ 1 \\ 0 \ 1 \end{array} \right] \ 2 \ \text{etc.} \]