Richard Lyons
Understanding Digital Signal Processing
716

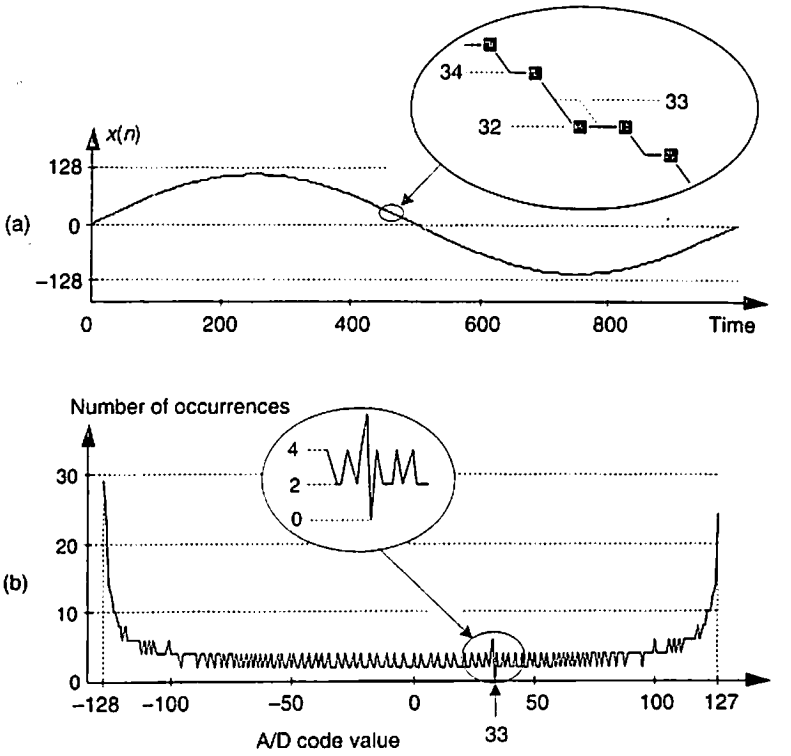

Figure 13-26 Eight-bit converter missing codes: (a) missing code of 00100001, decimal 33; (b) histogram plot.

In practice, the input analog sinewave must have an amplitude somewhat greater than the analog signal that we intend to digitize in tual application, and a frequency that is unrelated to (incoherent with sampling rate. In an effort to exercise (test) all of the converter's output we digitize as many cycles of the input sinewave as possible for ou togram test.

13.10 FAST FIR FILTERING USING THE FFT

In the late 1960s, while contemplating the notion of time-domain convo DSP pioneer Thomas Stockham (digital audio expert and inventor of th pact disc) realized that time-domain convolution could sometimes b formed much more efficiently using fast Fourier transform (FFT) algo rather than using the direct convolution implemented with tapped-del FIR filters. The principle behind this FFT-based convolution scheme,

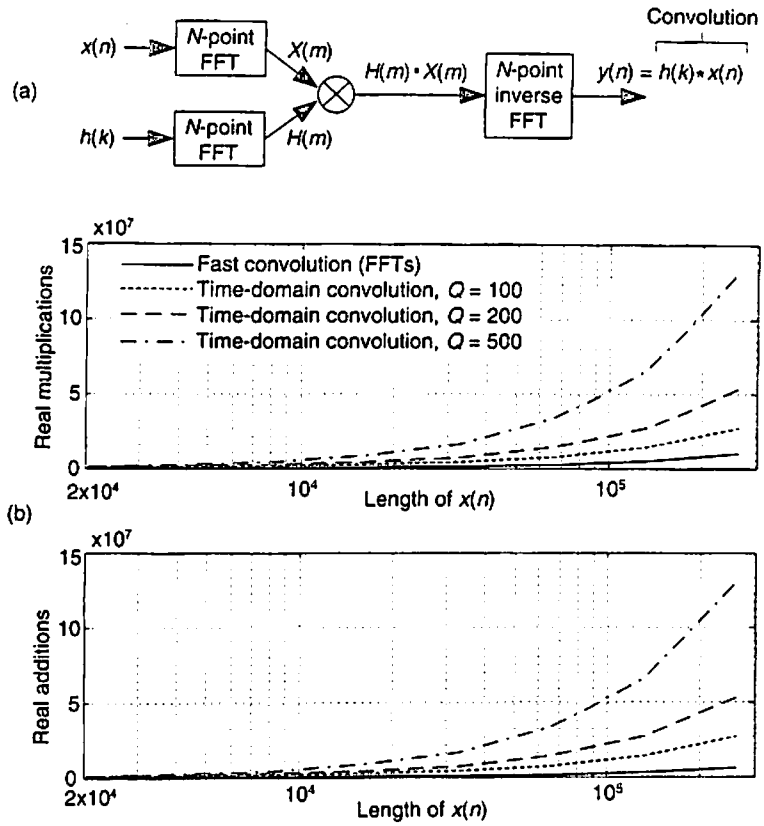

Figure 13-27 Fast convolution: (a) basic process; (b) computational workloads for various FIR filter tap lengths Q.

fast convolution (also called *block convolution* or *FFT convolution*), is diagrammed in Figure 13–27(a). In that figure $x(n)$ is an input signal sequence and $h(k)$ is the Q-length impulse response (coefficients) of a tapped-delay line FIR filter. Figure 13–27(a) is a graphical depiction of one form of the convolution theorem: Multiplication in the frequency domain is equivalent to convolution in the time domain.

The standard convolution equation, for a Q-tap FIR filter, given in Eq. (5–6) is repeated here for reference as

$$y(n) = \sum_{k=0}^{Q-1} h(k)x(n-k) = h(k) * x(n) \tag{13–67}$$

where the symbol "*" means convolution. When the filter's $h(k)$ impulse response has a length greater than 40 to 80 (depending on the hardware and

software being used), the process in Figure 13–27(a) requires fewer computations than directly implementing the convolution expression in Eq. (13–67). Consequently, this fast convolution technique is a computationally efficient signal processing tool, particularly when used for digital filtering. Fast convolution's gain in computational efficiency becomes quite significant when the lengths of $h(k)$ and $x(n)$ are large.

Figure 13–27(b) indicates the reduction in the fast convolution algorithm's computational workload relative to the standard (tapped-delay line) time-domain convolution method, Eq. (13–67), versus the length of the $x(n)$ sequence for various filter impulse response lengths Q. (Please do not view Figure 13–27(b) as any sort of *gospel truth*. That figure is merely an indicator of fast convolution's computational efficiency.)

The necessary forward and inverse FFT sizes, N, in Figure 13–27(a) must of course be equal and are dependent upon the length of the original $h(k)$ and $x(n)$ sequences. Recall from Eq. (5–29) that if $h(k)$ is of length Q and $x(n)$ is of length P, the length of the final $y(n)$ sequence will be L where

$$\text{Length of } y(n): \; L = Q + P - 1. \tag{13–67'}$$

For this fast convolution technique to yield valid results, the forward and inverse FFT sizes *must* be equal to or greater than L. So, to implement fast convolution we must choose an N-point FFT size such that $N \geq L$, and zero-pad $h(k)$ and $x(n)$ so they have new lengths equal to N. The desired $y(n)$ output is the real part of the first L samples of the inverse FFT. Note that the $H(m)$ sequence, the FFT of the FIR filter's $h(k)$ impulse response, need only be computed once and stored in memory.

Now if the $x(n)$ input sequence length P is so large that FFT processing becomes impractical, or your hardware memory buffer can only hold small segments of the $x(n)$ time samples, then $x(n)$ must be partitioned into multiple blocks of samples and each sample block processed individually. If the partitioned-$x(n)$ block lengths are N, a straightforward implementation of Figure 13–27(a) leads to time-domain aliasing errors in $y(n)$ due to the circular nature (spectral wraparound) of the discrete Fourier transform (and the FFT). Two techniques are used to avoid that time-domain aliasing problem, the *overlap-and-save* method and the *overlap-and-add* method. Of these two methods, let's first have a look at the overlap-and-save fast convolution filtering technique shown in Figure 13–28(a).

Given that the desired FIR filter's $h(k)$ impulse response length is Q and the $x(n)$ filter input sequence is of length P, the steps to perform overlap-and-save fast convolution filtering are as follows:

1. Choose an FFT size of N, where N is an integer power of two equal to roughly four times Q.

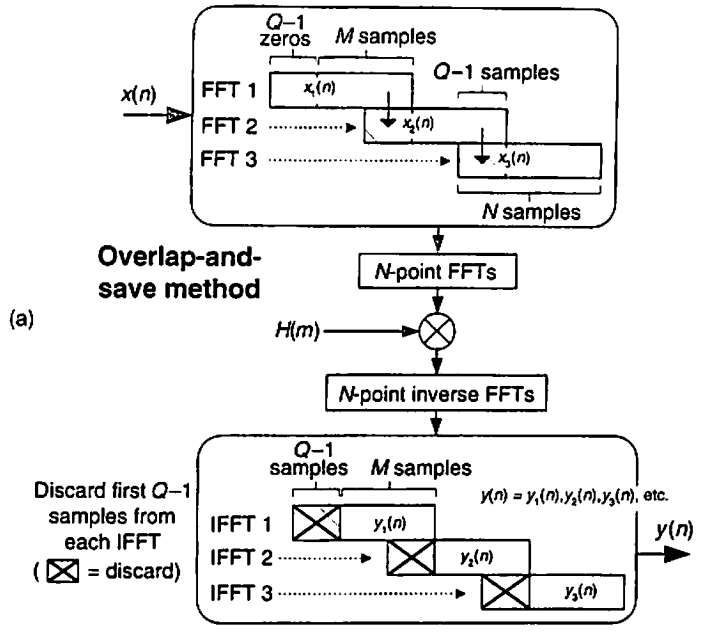

Figure 13-28 Fast convolution block processing (continues).

2. Append $(N–Q)$ zero-valued samples to the end of the $h(k)$ impulse response and perform an N-point FFT on the extended sequence, producing the complex $H(m)$ sequence.
3. Compute integer M using $M = N–(Q–1)$.
4. Insert $(Q–1)$ zero-valued samples prior to the first M samples of $x(n)$, creating the first N-point FFT input sequence $x_1(n)$.
5. Perform an N-point FFT on $x_1(n)$, multiply that FFT result by the $H(m)$ sequence, and perform an N-point inverse FFT on the product. Discard the first $(Q–1)$ samples of the inverse FFT results to generate the first M-point output block of data $y_1(n)$.
6. Attach the last $(Q–1)$ samples of $x_1(n)$ to the beginning of the second M-length block of the original $x(n)$ sequence, creating the second N-point FFT input sequence $x_2(n)$ as shown in Figure 13–28(a).
7. Perform an N-point FFT on $x_2(n)$, multiply that FFT result by the $H(m)$ sequence, and perform an N-point inverse FFT on the product. Discard the first $(Q–1)$ samples of the inverse FFT results to generate the second M-point output block of data $y_2(n)$.
8. Repeat Steps 6 and 7 until we have gone through the entire original $x(n)$ filter input sequence. Depending on the length P of the original $x(n)$

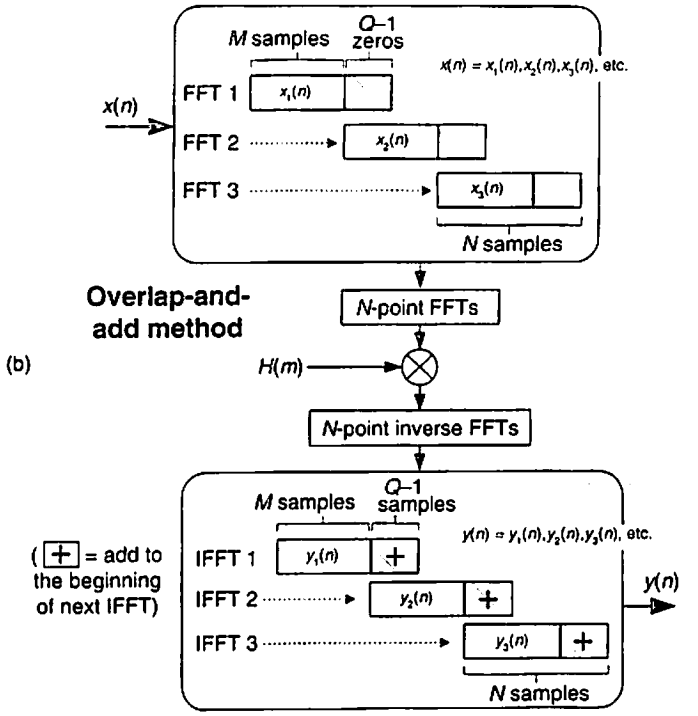

Figure 13-28 (continued)

input sequence and the chosen value for N, we must append anywhere from Q–1 to N–1 zero-valued samples to the end of the original $x(n)$ input samples in order to accommodate the final block of forward and inverse FFT processing.

9. Concatenate the $y_1(n)$, $y_2(n)$, $y_3(n)$, . . . sequences shown in Figure 13–28(a), discarding any unnecessary trailing zero-valued samples, to generate your final linear-convolution filter output $y(n)$ sequence.
10. Finally, experiment with different values of N to see if there exists an *optimum* N that minimizes the computational workload for your hardware and software implementation. In any case, N must not be less than (M+Q–1). (Smaller N means many small-sized FFTs are needed, and large N means fewer, but larger-sized, FFTs are necessary. Pick your poison.)

The second fast convolution method, the *overlap-and-add* technique, is shown in Figure 13–28(b). In this method, the $x(n)$ input sequence is partitioned (segmented) into data blocks of length M, and our data overlapping takes place in the inverse FFT time-domain sequences. Given that the desired FIR filter's $h(k)$ impulse response length is Q and the $x(n)$ filter input se-

quence is of length P, the steps to perform overlap-and-add fast convolution filtering are as follows:

1. Choose an FFT size of N, where N is an integer power of two equal to roughly two times Q.
2. Append $(N–Q)$ zero-valued samples to the end of the $h(k)$ impulse response and perform an N-point FFT on the extended sequence, producing the complex $H(m)$ sequence.
3. Compute integer M using $M = N–(Q–1)$.
4. Append $(Q–1)$ zero-valued samples to the end of the first M samples, $x_1(n)$, of the original $x(n)$ sequence, creating the first N-point FFT input sequence.
5. Perform an N-point FFT on the first N-point FFT input sequence, multiply that FFT result by the $H(m)$ sequence, and perform an N-point inverse FFT on the product. Retain the first M samples of the inverse FFT sequence, generating the first M-point output block of data $y_1(n)$.
6. Append $(Q–1)$ zero-valued samples to the end of the second M samples, $x_2(n)$, of the original $x(n)$ sequence, creating the second N-point FFT input sequence.
7. Perform an N-point FFT on the second N-point FFT input sequence, multiply that FFT result by the $H(m)$ sequence, and perform an N-point inverse FFT on the product. Add the last $(Q–1)$ samples from the previous inverse FFT to the first $(Q–1)$ samples of the current inverse FFT sequence. Retain the first M samples of the sequence resulting from the $(Q–1)$-element addition process, generating the second M-point output block of data $y_2(n)$.
8. Repeat Steps 6 and 7 until we have gone through the entire original $x(n)$ filter input sequence. Depending on the length P of the original $x(n)$ input sequence and the chosen value for N, we must append anywhere from $Q–1$ to $N–1$ zero-valued samples to the end of the original $x(n)$ input samples in order to accommodate the final block of forward and inverse FFT processing.
9. Concatenate the $y_1(n)$, $y_2(n)$, $y_3(n)$, . . . sequences shown in Figure 13–28(b), discarding any unnecessary trailing zero-valued samples, to generate your final linear-convolution filter output $y(n)$ sequence.
10. Finally, experiment with different values of N to see if there exists an *optimum* N that minimizes the computational workload for your hardware and software implementation. N must not be less than $(M+Q–1)$. (Again, smaller N means many small-sized FFTs are needed, and large N means fewer, but larger-sized, FFTs are necessary.)

It's useful to realize that the computational workload of these fast convolution filtering schemes does not change as Q increases in length up to a value of N. Another interesting aspect of fast convolution, from a hardware standpoint, is that the FFT indexing bit-reversal problem discussed in Sections 4.5 and 4.6 is not an issue here. If the FFTs result in $X(m)$ and $H(m)$ having bit-reversed output sample indices, the multiplication can still be performed directly on the scrambled $H(m)$ and $X(m)$ sequences. Then an appropriate inverse FFT structure can be used that expects bit-reversed input data. That inverse FFT then provides an output sequence whose time-domain indexing is in the correct order. Neat!

By the way, it's worth knowing that there are no restrictions on the filter's finite-length $h(k)$ impulse response—$h(k)$ is not limited to being real-valued and symmetrical as is traditional with tapped-delay line FIR filters. Sequence $h(k)$ can be complex-valued, asymmetrical (to achieve nonlinear-phase filtering), or whatever you choose.

One last issue to bear in mind: the complex amplitudes of the standard radix-2 FFT's output samples are proportional to the FFT sizes, N, so the product of two FFT outputs will have a gain proportional to N^2. The inverse FFT has a normalizing gain reduction of only $1/N$. As such, our fast convolution filtering methods will have an overall gain that is not unity. We suggest that practitioners give this gain normalization topic some thought during the design of their fast convolution system.

To summarize this frequency-domain filtering discussion, the two fast convolution filtering schemes can be computationally efficient, compared to standard tapped-delay line FIR convolution filtering, particularly when the $x(n)$ input sequence is large and high-performance filtering is needed (requiring many filter taps; i.e., Q = 40 to 80). As for which method, overlap-and-save or overlap-and-add, should be used in any given situation, there is no simple answer. Choosing a fast convolution method depends on many factors: the fixed/floating-point arithmetic used, memory size and access latency, computational hardware architecture, and specialized built-in filtering instructions, etc.

13.11 GENERATING NORMALLY DISTRIBUTED RANDOM DATA

Section D.7 in Appendix D discusses the normal distribution curve as it relates to random data. A problem we may encounter is how actually to generate random data samples whose distribution follows that normal (Gaussian) curve. There's a straightforward way to solve this problem using any software package that can generate uniformly distributed random data, as most of them do[27]. Figure 13–29 shows our situation pictorially where we require