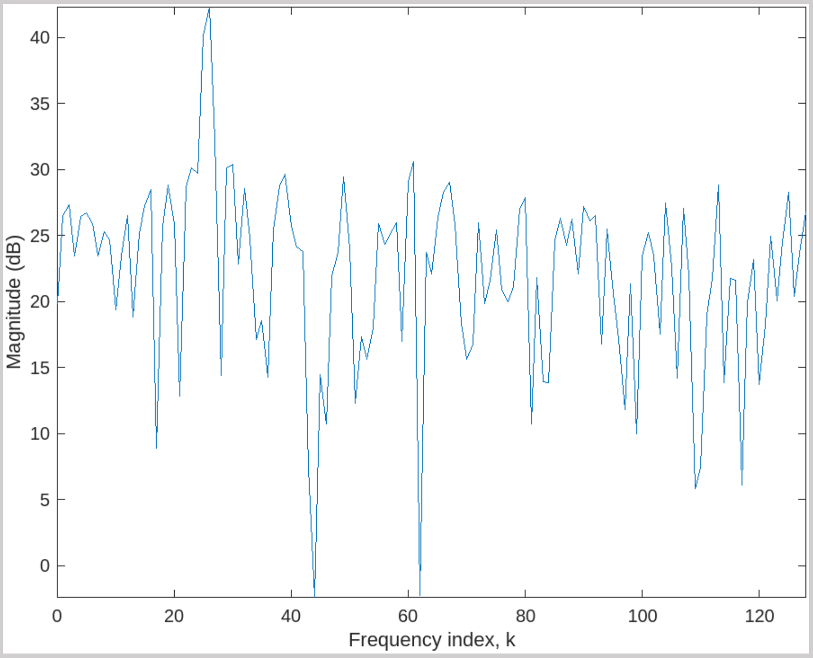
Figure 3.18 Spectrum of sine wave corrupted by white noise, SNR = 0 dB



3.5 Fixed-Point Representations and Quantization Effects

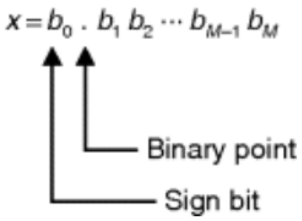
The basic element in digital hardware is the binary device that contains 1 bit of information. A register (or memory unit) containing B bits of information is called the B -bit word. There are several different methods for representing numbers and carrying out arithmetic operations. In this book, we focus on widely used fixed-point implementations [9–14].

3.5.1 Fixed-Point Formats

The most commonly used fixed-point representation of a fractional number x is illustrated in Figure 2.19. The word length is B $(= M + 1)$ bits, that is, M magnitude bits and one sign bit. The most significant bit (MSB) is the sign bit, which represents the sign of the number as follows:

$$(2.77)\ b_0 = \begin{cases} 0, & x \geq 0 \ \text{(positive number)} \\ 1, & x < 0 \ \text{(negative number)} \end{cases}$$

Figure 3.19 Fixed-point representation of binary fractional numbers



The remaining M bits represent the magnitude of the number. The rightmost bit, b_M, is called the least significant bit (LSB), which represents the precision of the number.

As shown in *Figure 3.19*, the decimal value of a positive ($b_0 = 0$) binary fractional number x can be expressed as

(2.78) $(x)_{10} = b_1 2^{-1} + b_2 2^{-2} + \cdots + b_M 2^{-M}$

$= \sum_{m=1}^{M} b_m 2^{-m}$

Example 2.22

The largest (positive) 16-bit fractional number in binary format is $x = 0111\ 1111\ 1111\ 1111b$ (the letter "b" denotes the binary representation of the number). The decimal value of this number can be computed as

$(x)_{10} = \sum_{m=1}^{15} 2^{-m} = 2^{-1} + 2^{-2} + \cdots + 2^{-15} = 1 - 2^{-15} \approx 0.999969$

The smallest non-zero positive number is $x = 0000\ 0000\ 0000\ 0001b$. The decimal value of this number is

$(x)_{10} = 2^{-15} = 0.000030518$

Negative numbers ($b_0 = 1$) can be represented using three different formats: the sign magnitude, 1's complement, and 2's complement. Fixed-point digital signal processors usually use the 2's complement format to represent negative numbers because it allows the processor to perform addition and subtraction using the same hardware. With the 2's complement format, a negative number is obtained by complementing all the bits of the positive binary number and then adding one to the LSB.

In general, the decimal value of a B -bit binary fractional number can be calculated as

(2.79) $(x)_{10} = -b_0 + \sum_{m=1}^{15} b_m 2^{-m}$

For example, the smallest (negative) 16-bit fractional number in binary format is $x = 1000\ 0000\ 0000\ 0000b$. From (2.79), its decimal value is −1. Therefore, the range of fractional binary numbers is

(2.80) $-1 \le x \le (1 - 2^{-M})$

For a 16-bit fractional number x, the decimal value range is $-1 \le x \le (1 - 2^{-15})$ with a resolution of 2^{-15}.

Example 2.23

Table 3.2 lists 4-bit binary numbers representing both integers and fractional numbers (decimal values) using the 2's complement format.

Table 3.2 Four-bit binary numbers in the 2's complement format and their corresponding decimal values

Binary numbers	Integers (sxxx.)	Fractions (s.xxx)
0000	0	0.000
0001	1	0.125
0010	2	0.250
0011	3	0.375
0100	4	0.500
0101	5	0.675
0110	6	0.750
0111	7	0.875
1000	-8	-1
1001	-7	-0.875
1010	-6	-0.750
1011	-5	-0.675
1100	-4	-0.500
1101	-3	-0.375
1110	-2	-0.250
1111	-1	-0.125

Example 2.24

Sixteen-bit data x with the decimal value 0.625 can be initialized using the binary form $x = 0101\ 0000\ 0000\ 0000b$, the hexadecimal form $x = 0x5000$, or the decimal integer

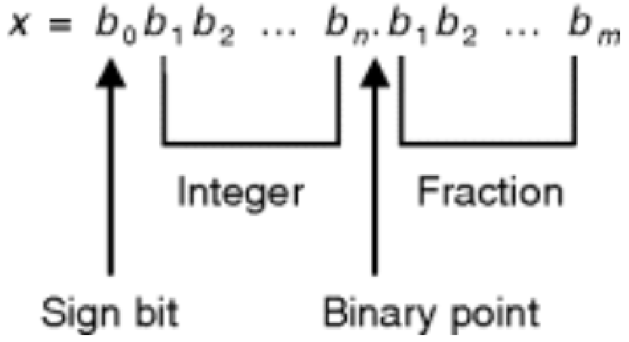
$$x = 2^{14} + 2^{12} = 20480$$

As shown in *Figure 3.19*, the easiest way to convert the normalized 16-bit fractional number into the integer that can be used by the C55xx assembler is to move the binary point to the right by 15 bits (at the right of b_M). Since shifting the binary point right by 1 bit is equivalent to multiplying the fractional number by 2, this can be done by multiplying the decimal value by $2^{15} = 32768$. For example, 0.625*32 768 = 20 480.

It is important to note that an implied binary point is used to represent the binary fractional number. The position of the binary point will affect the accuracy (dynamic range and precision) of the number. The binary point is a programmer's convention and the programmer has to keep track of the binary point when manipulating fractional numbers in assembly language programming.

Different notations can be used to represent different fractional formats. Similar to *Figure 3.19*, the more general fractional format $Qn.m$ is illustrated in *Figure 3.20*, where $n + m = M = B - 1$. There are n bits to the left of the binary point representing the integer portion, and m bits to the right representing fractional values. The most popular used fractional number representation shown in *Figure 3.19* is called the $Q0.15$ format ($n = 0$ and $m = 15$), which is also simply called the $Q15$ format since there are 15 fractional bits. Note that the $Qn.m$ format is represented in MATLAB® as $[B\ m]$. For example, the $Q15$ format is represented as $[16\ 15]$.

Figure 3.20 A general binary fractional number



Example 2.25

The decimal value of the 16-bit binary number $x = 0100\ 1000\ 0001\ 1000b$ depends on which Q format is used by the programmer. Using a larger n increases the dynamic range of the number with the cost of decreasing the precision, and vice versa. Three examples representing the 16-bit binary number $x = 0100\ 1000\ 0001\ 1000b$ are given as follows:

$$Q0.15, x = 2^{-1} + 2^{-4} + 2^{-11} + 2^{-12} = 0.56323$$

$$Q2.13, x = 2^1 + 2^{-2} + 2^{-9} + 2^{-10} = 2.25293$$
$$Q5.10, x = 2^4 + 2^1 + 2^{-6} + 2^{-7} = 18.02344$$

Fixed-point arithmetic is often used by DSP hardware for real-time processing because it offers fast operation and relatively economical implementation. Its drawbacks include small dynamic range and low resolution. These problems will be discussed in the following sections.

3.5.2 Quantization Errors

As discussed in Section 2.4.1, numbers used by digital devices are represented by a finite number of bits. The errors caused by the difference between the desired and actual values are called the finite-wordlength (finite-precision, numerical, or quantization) effects. In general, finite-precision effects can be broadly categorized into the following classes:

1. Quantization errors:
   (a) Signal quantization.
   (b) Coefficient quantization.
2. Arithmetic errors:
   (a) Roundoff (or truncation).
   (b) Overflow.

3.5.3 Signal Quantization

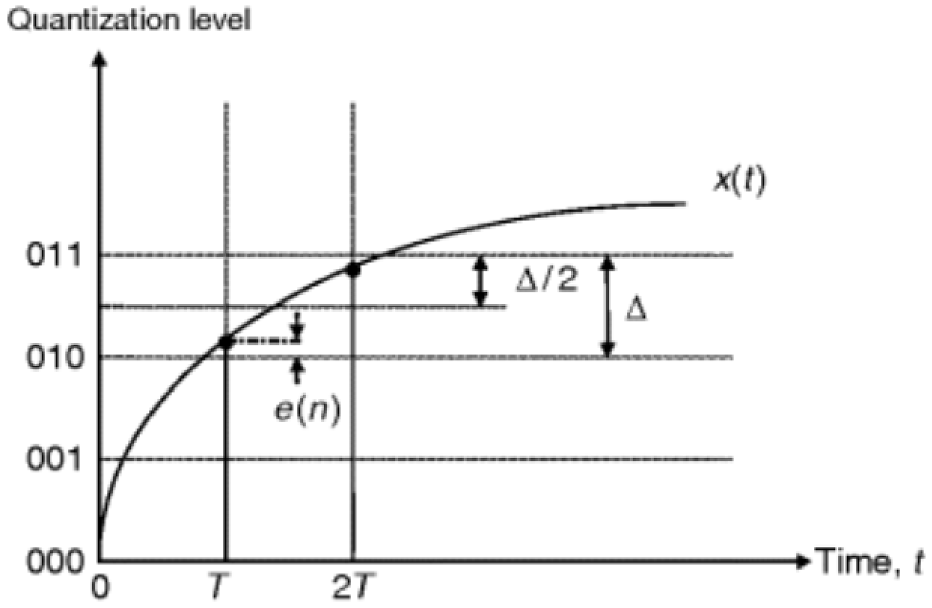
As discussed in Chapter 1, the ADC converts an analog signal $x(t)$ into a digital signal $x\ (n)$. The input signal is first sampled to obtain the discrete-time signal $x\ (nT)$ with infinite precision. Each $x\ (nT)$ value is then encoded (quantized) using B -bit wordlength to obtain the digital signal $x\ (n)$. We assume that the signal $x\ (n)$ is interpreted as the Q15 fractional number shown in *Figure 3.19* such that $-1 \leq x\ (n) < 1$. Thus, the dynamic range of fractional numbers is 2. Since the quantizer employs B bits, the number of quantization levels available is 2^B. The spacing between two successive quantization levels is

(2.81) $\Delta = \frac{2}{2^B} = 2^{-B+1} = 2^{-M}$

which is called the quantization step (interval, width, or resolution). For example, the output of a 4-bit converter with quantization interval $\Delta = 2^{-3} = 0.125$ is summarized in *Table 3.2*.

This book uses rounding (instead of truncation) for quantization. The input value $x\ (nT)$ is rounded to the nearest level as illustrated in *Figure 3.21* for a 3-bit ADC. Assuming there is a line equally between two adjacent quantization levels, the signal value above this line will be assigned to the higher quantization level, while the signal value below this line is assigned to the lower level. For example, the discrete-time signal $x\ (T)$ in *Figure 3.21* is rounded to 010b, since the real value is below the middle line between 010b and 011b, while $x\ (2T)$ is rounded to 011b since the value is above the middle line.

Figure 3.21 Quantization process related to a 3-bit ADC



The quantization error (or noise), $e\ (n)$, is the difference between the discrete-time signal $x(nT)$ and the quantized digital signal $x(n)$, expressed as

(2.82) $e\ (n) = x(n) - x(nT)$

Figure 3.21 clearly shows that

(2.83) $|e\ (n)| \leq \frac{\Delta}{2} = 2^{-B}$

Thus, the quantization noise generated by an ADC depends on the quantization step determined by the wordlength B. The use of more bits results in a smaller quantization step (or finer resolution), thus yielding lower quantization noise.

From (2.82), we can express the ADC output as the sum of the quantizer input $x\ (nT)$ and the error $e\ (n)$. That is,

(2.84) $x(n) = Q[x(nT)] = x(nT) + e(n)$

where $Q[.]$ denotes the quantization operation. Therefore, the nonlinear operation of the quantizer is modeled as the linear process that introduces an additive noise $e\ (n)$ into the digital signal $x\ (n)$.

For an arbitrary signal with fine quantization (B is large), the quantization error $e(n)$ is assumed to be uncorrelated with $x\ (n)$, and is a random noise that is uniformly distributed in the interval $[-\frac{\Delta}{2},\frac{\Delta}{2}]$. From (2.71), we have

(2.85) $E[x(n)] = \frac{-\frac{\Delta}{2}+\frac{\Delta}{2}}{2} = 0$

Thus, the quantization noise $e(n)$ has zero mean. From (2.73), the variance is

(2.86) $\sigma_e^2 = \frac{\Delta^2}{12} = {2^{-2B}}/{3}$

Therefore, larger wordlength results in smaller input quantization error.

The signal-to-quantization-noise ratio (SQNR) can be expressed as

(2.87) $SQNR = {\sigma_x^2}/{\sigma_e^2} = 3.2^{2B}.\sigma_x^2$

where σ_x^2 denotes the variance of the signal $x(n)$. Usually, the SQNR is expressed in dB as

(2.88) $SQNR\ (dB) = 10\log_{10} {\sigma_x^2}/{\sigma_e^2} = 4.77 + 6.02B + 10\log_{10}\sigma_x^2$

This equation indicates that for each additional bit used in the ADC, the converter provides about 6 dB gain. When using a 16-bit ADC ($B = 16$), the maximum SQNR is about 98.1 dB if the input signal is a sine wave. This is because the sine wave has maximum amplitude 1.0, so $10\log_{10}\sigma_x^2 = 10\log_{10} 1/2 = -3$ and (2.88) becomes 4.77 + 6.02 × 16 − 3.0 = 98.09. Another important feature of (2.88) is that the SQNR is proportional to the variance of the signal σ_x^2. Therefore, we want to keep the power of the signal as large as possible. This is an important consideration when we discuss scaling issues in Section 2.5.

Example 2.26

Signal quantization effects may be subjectively evaluated by observing and listening to quantized speech. The sampling rate of speech file `timit1.asc` is $f_s = 8$ kHz with $B =$

16. This speech file can be viewed and played using the MATLAB® script (`example2_26.m`):

```
load timit1.asc;                % Speech sampled at 8 kHz, 16 bits
plot(timit1);
title('16-bit speech');
xlabel('Time index, n');
ylabel('Amplitude');
soundsc(timit1, 8000, 16);        % Play 16-bit speech
```

where the MATLAB® function `soundsc` automatically scales and plays the vector as sound.

We can simulate the quantization of data with 8-bit wordlength by

```
qx = round(timit1/256);          % Resampled to 8 bits
```

where the function `round` rounds the real number to the nearest integer. We then evaluate the quantization effects by

```
plot(qx); title('8-bit speech');
xlabel('Time index, n');
ylabel('Amplitude');
soundsc(qx, 8000, 16);            % Play 8-bit speech
```

By comparing the graph and sound of `timit1` and `qx`, the signal quantization effects may be understood.

3.5.4 Coefficient Quantization

The digital filter coefficients, b_l and a_m, computed by a filter design package such as MATLAB®, are usually represented using the floating-point format. When implementing a digital filter, these filter coefficients must be quantized for the given fixed-point processor. Therefore, the performance of the fixed-point digital filter will be different from its original design specifications.

The coefficient quantization effects become more significant when tighter specifications are used, especially for IIR filters. Coefficient quantization can cause serious problems if the poles of the IIR filters are too close to the unit circle. This is because these poles may move outside the unit circle due to coefficient quantization, resulting in an unstable filter. Such undesirable effects are far more pronounced in high-order systems.

Coefficient quantization is also affected by the digital filter structure. For example, the direct-form implementation of IIR filters is more sensitive to coefficient quantization than the cascade structure (to be introduced in Chapter 4) which consists of multiple sections of second- (or one first-)order IIR filters.

3.5.5 Roundoff Noise

Consider the example of computing the product $y(n) = \alpha x(n)$ in DSP systems. Assume α and $x(n)$ are B-bit numbers, and multiplication yields $2B$-bit product $y(n)$. In most applications, this product may have to be stored in memory or output as a B-bit word. The $2B$-bit product can be either truncated or rounded to B bits. Since truncation causes an undesired bias effect, we should restrict our attention to the rounding.

Example 2.27

In C programming, rounding a real number to an integer number can be implemented by adding 0.5 to the real number and then truncating the fractional part. The following C statement

```
y=(short) (x+0.5);
```

rounds the real number `x` to the nearest integer `y`. As shown in Example 2.26, MATLAB® provides the function round for rounding a real number.

The process of rounding a $2B$-bit product to B bits is similar to that of quantizing a discrete-time signal using a B-bit quantizer. Similar to (2.84), the nonlinear roundoff operation can be modeled as a linear process expressed as

(2.89) $y(n) = Q[\alpha x(n)] = \alpha x(n) + e(n)$

where $\alpha x(n)$ is the $2B$-bit product and $e(n)$ is the roundoff noise due to rounding the $2B$-bit product to B bits. The roundoff noise is a uniformly distributed random variable as defined in (2.83); thus, it has zero mean, and its power is defined in (2.86).

It is important to note that most commercially available fixed-point digital signal processors such as the TMS320C55xx have double-precision accumulator(s). As long as the program is carefully written, it is possible to ensure that rounding occurs only at the final stage of the calculation. For example, consider the computation of FIR filtering given in (2.14). We can keep the sum of all temporary products, $b_l x(n-l)$, in the double-precision accumulator. Rounding is only performed when the final sum is saved to memory with B-bit wordlength.

3.5.6 Fixed-Point Toolbox

The MATLAB®*Fixed-Point Toolbox* [15] provides fixed-point data types and arithmetic for developing fixed-point DSP algorithms. The toolbox provides the quantizer function for constructing a quantizer object. For example, we can use the syntax

```
q=quantizer
```

to create the quantizer object `q` with properties set to the following default values:

```
mode='fixed';

roundmode='floor';

overflowmode='saturate';

format=[16 15];
```

Note that `[16 15]` is equivalent to the Q15 format.

After we have constructed the quantizer object, we can apply it to data using the quantize function with the following syntax:

```
y =quantize(q, x)
```

The command `y =quantize(q, x)` uses the quantizer object `q` to quantize `x`. When `x` is a numeric array, each element of the `x` will be quantized.

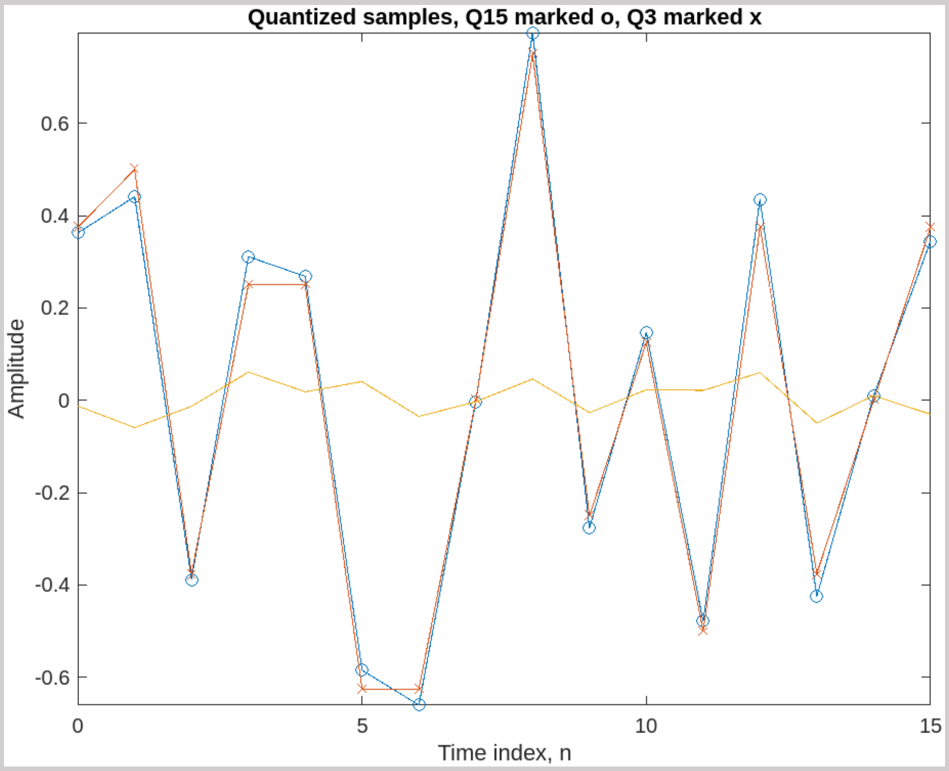
Example 2.28

Similar to Example 2.19, we generate a zero-mean white noise using MATLAB® function rand, which uses the double-precision, floating-point format. We then construct two quantizer objects and quantize the white noise to Q15 (16-bit) and Q3 (4-bit) formats. We plot the quantized noise in the Q15 and Q3 formats and the difference between these two formats in

Figure 3.22 using the following MATLAB® script (`example2_28.m`):

```
N=16;
n=[0:N-1];
xn = sqrt(3)*(rand(1,N)-0.5); % Generate zero-mean white noise
q15 = quantizer('fixed', 'convergent', 'wrap', [16 15]); %Q15
q3 = quantizer('fixed', 'convergent', 'wrap', [4 3]);    %Q3
```

Figure 3.22 Quantization using Q15 and Q3 formats and the difference $e(n)$



The MATLAB®*Fixed-Point Toolbox* also provides several radix conversion functions, which are summarized in Table 2.3. For example,

Table 3.3 List of radix conversion functions using quantizer object.

Function	Description
`bin2num`	Convert 2's complement binary string to a number
`hex2num`	Convert hexadecimal string to a number
`num2bin`	Convert a number to binary string
`num2hex`	Convert a number to its hexadecimal equivalent
`num2int`	Convert a number to a signed integer

```
y=num2int(q,x)
```

uses `q.format` to convert a number `x` to an integer `y`.

Example 2.29

In order to test some DSP algorithms using fixed-point C programs, we may need to generate specific data files for simulations. As shown in Example 2.28, we can use MATLAB® to generate a testing signal and construct a quantizer object. In order to save the Q15 data in integer format, we use the function `num2int` in the following MATLAB® script (`example2_29.m`):

```
N=16; n=[0:N-1];
xn = sqrt(3)*(rand(1,N)-0.5); % Generate zero-mean white noise
q15 = quantizer('fixed', 'convergent', 'wrap', [16 15]); %Q15
Q15int = num2int(q15,xn)
```

3.6 Overflow and Solutions

Assuming that the signals and filter coefficients have been properly normalized in the range of -1 and 1 for fixed-point arithmetic, the sum of two B -bit numbers may fall outside the range of -1 and 1. The term overflow means that the result of the arithmetic operation exceeds the capacity of the register used to hold the result. When using a fixed-point processor, the range of numbers must be carefully examined and adjusted in order to avoid overflow. This may be achieved by using different $Qn.m$ formats with desired dynamic ranges. For example, a larger dynamic range can be obtained by using a larger n, with the cost of reducing m (decreasing resolution).

Example 2.30

Assume 4-bit fixed-point hardware uses the fractional 2's complement format (see Table 2.2). If $x_1 = 0.875$ $(0111b)$ and $x_2 = 0.125$ $(0001b)$, the binary sum of $x_1 + x_2$ is $1000b$. The decimal value of this signed binary number is -1, not the correct answer $+1$. That is, when the addition result exceeds the dynamic range of the register, overflow occurs and an unacceptable error is produced.

Similarly, if img $(1100b)$ and img$(0101b)$, imgb, which is $+0.875$, not the correct answer -1.125. Therefore, subtraction may also result in underflow.

For the FIR filtering defined in (2.14), overflow will result in severe distortion of the output $y(n)$. For the IIR filter defined in (2.22), the overflow effect is much more serious because the errors will be fed back. The problem of overflow may be eliminated using saturation arithmetic and proper scaling (or constraining) signals at each node within the filter to maintain the magnitude of the signal. The MATLAB®*DSP System Toolbox* provides the function `scale(hd)` to scale the second-order IIR filter `hd` to reduce possible overflows when the filter is operated using fixed-point arithmetic.

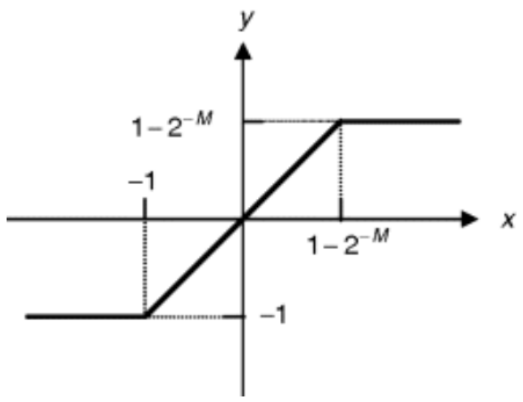
3.6.1 Saturation Arithmetic

Most digital signal processors have mechanisms to protect against overflow and automatically indicate the overflow if it occurs. For example, saturation arithmetic prevents overflow by clipping the results to a maximum value. Saturation logic is illustrated in Figure 2.23 and can be expressed as

$$(2.90)\ y = \begin{cases} 1-2^{-M}, & \text{if } x \geq 1-2^{-M} \\ x, & \text{if } -1 \leq x < 1 \\ -1, & \text{if } x < -1 \end{cases}$$

where x is the original addition result and y is the saturated adder output. If the adder is in saturation mode, the undesired overflow can be avoided since the 32-bit accumulator fills to its maximum (or minimum) value, but does not roll over. Similar to Example 2.28, when 4-bit hardware with saturation arithmetic is used, the addition result of $x_1 + x_2$ is $0111b$, or 0.875 in decimal value. Compared to the correct answer 1, there is an error of 0.125. This result is much better than the hardware without saturation arithmetic.

Figure 3.23 Characteristics of saturation arithmetic



Saturation arithmetic has a similar effect of "clipping" the desired waveform. This is the nonlinear operation that will introduce undesired nonlinear components into the saturated signal. Therefore, saturation arithmetic can be used to guarantee that overflow will not occur. Nevertheless, it should not be the only solution for solving overflow problems.

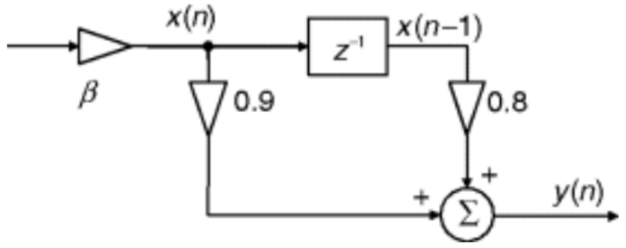
3.6.2 Overflow Handling

As mentioned earlier, the C55xx supports saturation logic to prevent overflow. The logic is enabled when the overflow mode bit (SATD) in status register ST1 is set (SATD = 1). The C55xx also provides overflow flags that indicate whether or not the arithmetic operation has overflowed. A flag will remain set until a reset is performed or when the status bit clear instruction is executed. If a conditional instruction (such as a branch, return, call, or conditional execution) that tests overflow status is executed, the overflow flag will be cleared.

3.6.3 Scaling of Signals

The most effective technique in preventing overflow is by scaling down the signal. For example, consider the simple FIR filter illustrated in *Figure 3.24* without the scaling factor β (or $\beta = 1$). Let $x(n) = 0.8$ and $x(n-1) = 0.6$; then the filter output $y(n) = 1.2$. When this filter is implemented on a fixed-point processor using the Q15 format without saturation arithmetic, undesired overflow occurs. As illustrated in *Figure 3.24*, the scaling factor, $\beta < 1$, can be used to scale down the input signal. For example, when $\beta = 0.5$, we have $x(n) = 0.4$ and $x(n-1) = 0.3$, and the result $y(n) = 0.6$ without overflow.

Figure 3.24 Block diagram of simple FIR filter with scaling factor β



If the signal $x(n)$ is scaled by β, the resulting signal variance becomes $\beta^2\sigma_x^2$. Thus, the SQNR in dB given in (2.88) changes to

$$(2.91)\ \text{SQNR} = 10\log_{10}\left(\beta^2\sigma_x^2 / \sigma_e^2\right) = 4.77 + 6.02B + 10\log_{10}\sigma_x^2 + 20\log_{10}\beta$$

Since we perform fractional arithmetic, $\beta < 1$ is used to scale down the input signal. The last term $20\log_{10}\beta$ has a negative value. Thus, scaling down the signal reduces the SQNR. For example, when $\beta = 0.5$, $20\log_{10}\beta = -6.02$ dB, thus reducing the SQNR of the input signal by about 6 dB. This is equivalent to losing 1 bit in representing the signal.