CHAPTER 1

Numerical Representation

To process a signal digitally, it must be represented in a digital format. This point may seem obvious, but it turns out that there are a number of different ways to represent numbers, and this representation can greatly affect both the result and the number of circuit resources required to perform a given operation. This chapter is focused more for people who are implementing digital signal processing (DSP) and is not really required to understand DSP fundamental concepts.

Digital electronics operate on bits, of course, which are used to form binary words. The bits can be represented as binary, decimal, octal, hexadecimal, or another form. These binary numbers can be used to represent "real" numbers. There are two basic types of arithmetic used in DSP: floating point and fixed point. Fixed-point numbers have a fixed decimal point as part of the number. Examples are 1234 (the same as 1234.0), 12.34, and 0.1234. This is the type of number we normally use every day. A floating-point number has an exponent. The most common example is scientific notation, used on many calculators. In floating point, 1,200,000 would be expressed as 1.2×10^6, and 0.0000234 would be expressed as 2.34×10^{-5}. Most of our discussion will focus on fixed-point numbers, as they are most commonly found in DSP applications. Once we understand DSP arithmetic issues with fixed-point numbers, then there is short discussion of floating-point numbers.

In DSP, we pretty much exclusively use signed numbers, meaning that there are both positive and negative numbers. This leads to the next point, which is how to represent the negative numbers.

In signed fixed-point arithmetic, the binary number representations include a sign, a radix or decimal point, and the magnitude. The sign indicates whether the number is positive or negative, and the radix (also called decimal) point separates the integer and fractional parts of the number.

The sign is normally determined by the leftmost, or most significant bit (MSB). The convention is that a zero is used for positive and one for negative. There are several formats to represent negative numbers, but the almost universal method is known as *2s complement*. This is the method discussed here.

Furthermore, fixed-point numbers are usually represented as either integer or fractional. In integer representation, the decimal point is to the right of the least significant bit (LSB), and there is no fractional part in the number. For an 8-bit number, the range that can be represented is from -128 to $+127$, with increments of 1.

Digital Signal Processing 101. DOI: 10.1016/B978-1-85617-921-8.00005-5

In fractional representation, the decimal point is often just to the right of the MSB, which is also the sign bit. For an 8-bit number, the range that can be represented is from −1 to +127/128 (almost +1), with increments of 1/128. This may seem a bit strange, but in practice, fractional representation has advantages, as will be explained.

This chapter presents several tables, with each row giving equivalent binary and hexadecimal numbers. The far right column gives the actual value in the chosen representation—for example, 16-bit integer representation. The actual value represented by the hex/binary numbers depends on which representation format is chosen.

1.1 Integer Fixed-Point Representation

The following table provides some examples showing the 2s complement integer fixed-point representation.

Table 1.1: 8-Bit integer representation

Binary	Hexadecimal	Actual Decimal Value
0111 1111	0x7F	127
0111 1110	0x7E	126
0000 0010	0x02	2
0000 0001	0x01	1
0000 0000	0x00	0
1111 1111	0xFF	−1
1111 1110	0xFE	−2
1000 0001	0x81	−127
1000 0000	0x80	−128

The 2s complement representation of the negative numbers may seem nonintuitive, but it has several very nice features. There is only one representation of 0 (all 0s), unlike other formats that have a “positive” and “negative” zero. Also, addition and multiplication of positive and negative 2s complement numbers work properly with traditional digital adder and multiplier structures. A 2s complement number range can be extended to the left by simply replicating the MSB (sign bit) as needed, without changing the value.

The way to interpret a 2s complement number is to use the mapping for each bit shown in the following table. A 0 bit in a given location of the binary word means no weight for that bit. A 1 in a given location means to use the weight indicated. Notice the weights double with each bit moving left, and the MSB is the only bit with a negative weight. You should satisfy yourself that all negative numbers will have an MSB of 1, and all positive numbers and zero have an MSB of 0.

Table 1.2: 2s complement bit weighting with 8 bit words

8-Bit Signed Integer	MSB							LSB
Bit weight	−128	64	32	16	8	4	2	1
Weight in powers of 2	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

This can be extended to numbers with larger number of bits. Following is an example with 16 bits. Notice how the numbers represented in a lesser number of bits (e.g., 8 bits) can be easily put into 16-bit representation by simply replicating the MSB of the 8-bit number eight times and tacking onto the left to form a 16-bit number. Similarly, as long as the number represented in the 16-bit representation is small enough to be represented in 8 bits, the leftmost bits can simply be shaved off to move to the 8-bit representation. In both cases, the decimal point stays to the right of the LSB and does not change location. This can be seen easily by comparing, for example, the representation of −2 in the 8-bit representation table and again in the 16-bit representation table.

Table 1.3: 16-Bit signed integer representation

Binary	Hexadecimal	Actual Decimal Value
0111 1111 1111 1111	0×7FFF	32,767
0111 1111 1111 1110	0×7FFE	32,766
0000 0000 1000 0000	0×0080	128
0000 0000 0111 1111	0×007F	127
0000 0000 0111 1110	0×007E	126
0000 0000 0000 0010	0×0002	2
0000 0000 0000 0001	0×0001	1
0000 0000 0000 0000	0×0000	0
1111 1111 1111 1111	0×FFFF	−1
1111 1111 1111 1110	0×FFFE	−2
1111 1111 1000 0001	0×FF81	−127
1111 1111 1000 0000	0×FF80	−128
1111 1111 0111 1111	0×FF80	−129
1000 0000 0000 0001	0×FF80	−32,767
1000 0000 0000 0000	0×FF80	−32,768

Table 1.4: 2s complement bit weighting with 16 bit word

MSB															LSB
−32,768	16,384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
-2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Now, let's look at some examples of trying to adding combinations of positive and negative 8-bit numbers together using a traditional unsigned digital adder. We throw away the carry bit from the last (MSB) adder stage.

Case #1: Positive and negative number sum		
+15	0000 1111	0x0F
−1	1111 1111	0xFF
+14	0000 1110	0x0E
Case #2: Positive and negative number sum		
−31	1110 0001	0xE1
+16	0001 0000	0x80
−15	1111 0000	0xF0
Case #3: Two negative numbers being summed		
−31	1110 0001	0xE1
−64	1100 0000	0xC0
−95	1010 0001	0xF0
Case #4: Two positive numbers being summed; result exceeds range		
+64	0100 0000	0x40
+64	0100 0000	0x40
+128	1000 0000**	0x80**

**Notice all the results are correct, except the last case. The reason is that the result, +128, cannot be represented in the range of an 8-bit 2s complement number.

Integer representation is often used in many software applications because it is familiar and works well. However, in DSP, integer representation has a major drawback. In DSP, there is a lot of multiplication. When you multiply a bunch of integers together, the results start to grow rapidly. It quickly gets out of hand and exceeds the range of values that can be represented. As we saw previously, 2s complement arithmetic works well, as long as you do not exceed the numerical range. This has led to the use of fractional fixed-point representation.

1.2 Fractional Fixed-Point Representation

The basic idea behind fractional fixed-point representation is all values are in the range from +1 to −1, so if they are multiplied, the result will not exceed this range. Notice that, if you want to convert from integer to 8-bit signed fractional, the actual values are all divided by 128. This maps the integer range of +127 to −128 to almost +1 to −1.

Table 1.5: 8-Bit fractional representation

Binary	Hexadecimal	Actual Decimal Value
0111 1111	0x7F	127/128 = 0.99219
0111 1110	0x7E	126/128 = 0.98438
0000 0010	0x02	2/128 = 0.01563
0000 0001	0x01	1/128 = 0.00781
0000 0000	0x00	0
1111 1111	0xFF	−1/128 = −0.00781
1111 1110	0xFE	−2/128 = −0.01563
1000 0001	0x81	−127/128 = −0.99219
1000 0000	0x80	−1.00

Table 1.6: 2s complement weighting for 8 bit fractional word

8-Bit Signed Fractional	MSB							LSB
Weight	−1	1/2	1/4	1/8	1/16	1/32	1/64	1/128
Weight in powers of 2	-2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}

Table 1.7: 16-Bit signed fractional representation

Binary	Hexadecimal	Actual Decimal Value
0111 1111 1111 1111	0x7FFF	32,767/32,768
0111 1111 1111 1110	0x7FFE	32,766/32,768
0000 0000 1000 0000	0x0080	128/32,768
0000 0000 0111 1111	0x007F	127/32,768
0000 0000 0111 1110	0x007E	126/32,768
0000 0000 0000 0010	0x0002	2/32,768
0000 0000 0000 0001	0x0001	1/32,768
0000 0000 0000 0000	0x0000	0
1111 1111 1111 1111	0xFFFF	−1/32,768
1111 1111 1111 1110	0xFFFE	−2/32,768
1111 1111 1000 0001	0xFF81	−127/32,768
1111 1111 1000 0000	0xFF80	−128/32,768
1111 1111 0111 1111	0xFF7F	−129/32,768
1000 0000 0000 0001	0x8001	−32,767/32,768
1000 0000 0000 0000	0x8000	−1

Table 1.8: 2s complement weighting for 16 bit fractional word

MSB															LSB
−1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024	1/2048	1/4096	1/8192	1/16,384	1/32,768
-2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

Fractional fixed point is often expressed in *Q format*. The representation shown above is Q15, which means that there are 15 bits to the right of the radix or decimal point. It might also be called Q1.15, meaning that there are 15 bits to right of the decimal point and one bit to the left.

The key property of fractional representation is that the numbers grow smaller, rather than larger, during multiplication. And in DSP, we commonly sum the results of many multiplication operations. In integer math, the results of multiplication can grow large quickly (see the following example). And when we sum many such results, the final sum can be very large, easily exceeding the ability to represent the number in a fixed-point integer format.

As an analogy, think about trying to display an analog signal on an oscilloscope. You need to select a voltage range (volts/division) in which the signal amplitude does not exceed the upper and lower range of the display. At the same time, you want the signal to occupy a reasonable part of the screen, so the detail of the signal is visible. If the signal amplitude occupies only 1/10 of a division, for example, it is difficult to see the signal.

To illustrate this situation, imagine using a 16-bit fixed point, and the signal has a value of 0.75 decimal or 24,676 in integer (which is 75% of full scale), and is multiplied by a coefficient of value 0.50 decimal or 16,384 integer (which is 50% of full scale).

0.75	24,576
× 0.50	× 16,384
0.375	402,653,184

Now the larger integer number can be shifted right after every such operation to scale the signal within a range that it can be represented, but most DSP designers prefer to represent numbers as fractional because it is a lot easier to keep track of the decimal point.

Now consider multiplication again. If two 16-bit numbers are multiplied, the result is a 32-bit number. As it turns out, if the two numbers being multiplied are Q15, you might expect the result in the 32-bit register to be a Q31 number (MSB to the left of the decimal point, all other bits to the right). Actually, the result is in Q30 format; the decimal point has shifted down to the right. Most DSP processors will automatically left shift the multiplier output to compensate for this when operating in fractional arithmetic mode. In Field Programmable Gate Array (FPGA) or hardware design, the designer may have to take this into account when connecting data buses between different blocks. Appendix A explains the need for this extra left shift in detail, as it will be important for those implementing fractional arithmetic on FPGAs or DSPs.

0×4000	value = ½ in Q15
× 0×2000	value = ¼ in Q15
0×0800 0000	value = 1/16 in Q31

After left shifting by one, we get

0×1000 0000 value = 1/8 in Q31—the correct result!

If we use only the top 16-bit word from multiplier output, after the left shift, we get

0×1000 value = 1/8 in Q15—again, the correct result!

1.3 Floating-Point Representation

Many of the complications encountered using the preceding methods can be avoided by using floating-point format. Floating-point format is basically like scientific notation on your calculator. Because of this, a floating-point number can have a much greater dynamic range than a fixed-point number with an equivalent number of bits. Dynamic range means the ability to represent very small numbers to very large numbers.

The floating-point number has both a mantissa (which includes sign) and an exponent. The mantissa is the value in the main field of your calculator, and the exponent is off to the side, usually as a superscript. Each is expressed as a fixed-point number (meaning the decimal point is fixed). The mantissa is usually in signed fractional format, and the exponent in signed integer format. The size of the mantissa and exponent in number of bits will vary depending on which floating-point format is used.

The following table shows two common 32-bit formats: "two word" and IEEE 754.

Table 1.9: Floating point format summary

Floating-Point Formats	No. of Mantissa Bits	No. of Exponent Bits
"two word"	16, in signed Q15	16, signed integer
IEEE_STD-754	23, in unsigned Q15, plus 1 bit to determine sign (not 2s complement!)	8, unsigned integer, biased by +127

To convert a fixed-point number in floating-point representation, we shift the fixed number left until there are no redundant sign bits. This process is called *normalization*. The number of these shifts determines the exponent value.

The drawback of floating-point calculations is the resources required. When adding or subtracting two floating-point numbers, we must first adjust the number with smaller absolute value so that its exponent is equal to the number with larger absolute value; then we can add

the two mantissas. If the mantissa result requires a left or right shift to represent, the exponent is adjusted to account for this shift. When multiplying two floating-point numbers, we multiply the mantissas and then sum the exponents. Again, if the mantissa result requires a left or right shift to represent, the new exponent must be adjusted to account for this shift. All of this requires considerably more logic than fixed-point calculations and often must run at much lower speed (although recent advances in FPGA floating-point implementation may significantly narrow this gap). For this reason, most DSP algorithms use fixed-point arithmetic, despite the onus on the designer to keep track of where the decimal point is and ensure that the dynamic range of the signal never exceeds the fixed-point representation or else becomes so small that quantization error becomes insignificant. We will see more on quantization error in a later chapter.

If you are interested in more information on floating-point arithmetic, there are many texts that go into this topic in detail, or you can consult the IEEE_STD-754 document.