9 Hardware and Software for Digital Signal Processors

Objectives:

This chapter introduces basics of digital signal processors such as processor architectures and hardware units, investigates fixed-point and floating-point formats, and illustrates the implementation of digital filters in real time.

9.1 Digital Signal Processor Architecture

Unlike microprocessors and microcontrollers, digital signal (DS) processors have special features that require operations such as fast Fourier transform (FFT), filtering, convolution and correlation, and real-time sample-based and block-based processing. Therefore, DS processors use a different dedicated hardware architecture.

We first compare the architecture of the general microprocessor with that of the DS processor. The design of general microprocessors and microcontrollers is based on the *Von Neumann architecture,* which was developed from a research paper written by John von Neumann and others in 1946. Von Neumann suggested that computer instructions, as we shall discuss, be numerical codes instead of special wiring. Figure 9.1 shows the Von Neumann architecture.

As shown in Figure 9.1, a Von Neumann processor contains a single, shared memory for programs and data, a single bus for memory access, an arithmetic unit, and a program control unit. The processor proceeds in a serial fashion in terms of fetching and execution cycles. This means that the central processing unit (CPU) fetches an instruction from memory and decodes it to figure out

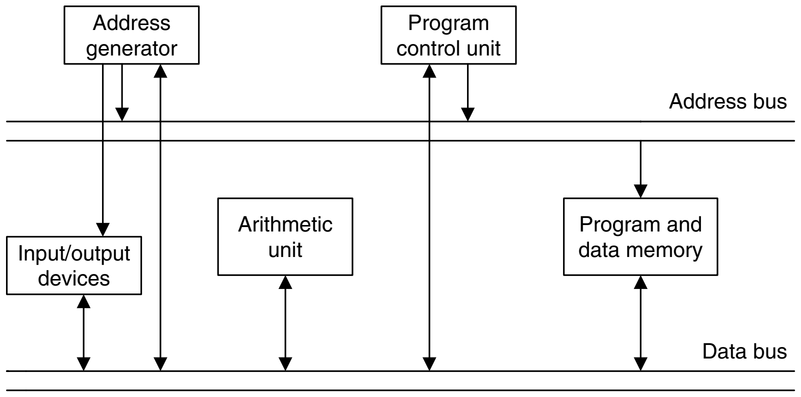


FIGURE 9.1 General microprocessor based on Von Neumann architecture.

what operation to do, then executes the instruction. The instruction (in machine code) has two parts: the *opcode* and the *operand.* The opcode specifies what the operation is, that is, tells the CPU what to do. The operand informs the CPU what data to operate on. These instructions will modify memory, or input and output (I/O). After an instruction is completed, the cycles will resume for the next instruction. One an instruction or piece of data can be retrieved at a time. Since the processor proceeds in a serial fashion, it causes most units to stay in a wait state.

As noted, the Von Neumann architecture operates the cycles of fetching and execution by fetching an instruction from memory, decoding it via the program control unit, and finally executing the instruction. When execution requires data movement—that is, data to be read from or written to memory—the next instruction will be fetched after the current instruction is completed. The Von Neumann–based processor has this bottleneck mainly due to the use of a single, shared memory for both program instructions and data. Increasing the speed of the bus, memory, and computational units can improve speed, but not significantly.

To accelerate the execution speed of digital signal processing, DS processors are designed based on the *Harvard architecture,* which originated from the Mark 1 relay-based computers built by IBM in 1944 at Harvard University. This computer stored its instructions on punched tape and data using relay latches. Figure 9.2 shows today's Harvard architecture. As depicted, the DS processor has two separate memory spaces. One is dedicated to the program code, while the other is employed for data. Hence, to accommodate two memory spaces, two corresponding address buses and two data buses are used. In this way, the program memory and data memory have their own

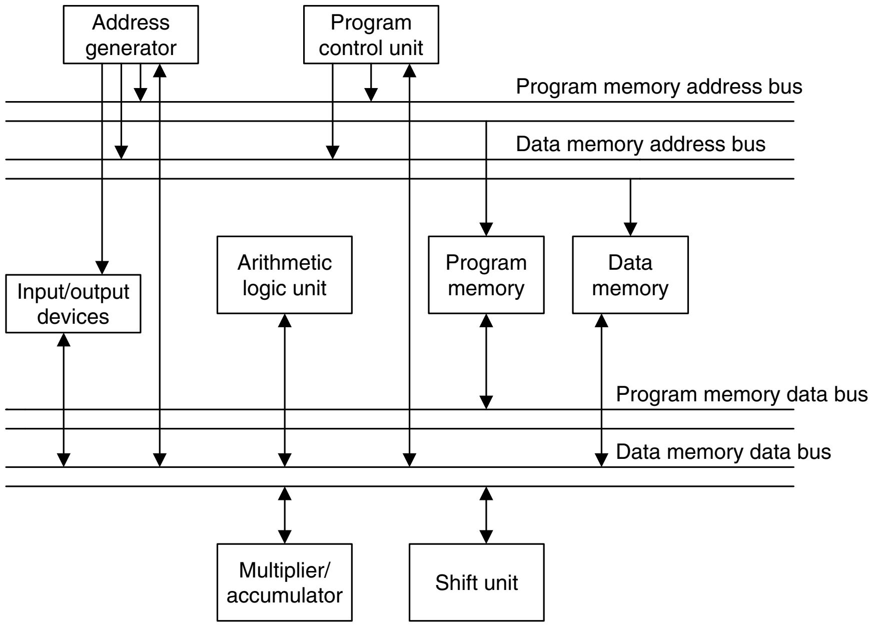


FIGURE 9.2 **Digital signal processors based on the Harvard architecture.**

connections to the program memory bus and data memory bus, respectively. This means that the Harvard processor can fetch the program instruction and data in parallel at the same time, the former via the program memory bus and the latter via the data memory bus. There is an additional unit called a *multiplier and accumulator* (MAC), which is the dedicated hardware used for the digital filtering operation. The last additional unit, the shift unit, is used for the scaling operation for fixed-point implementation when the processor performs digital filtering.

Let us compare the executions of the two architectures. The Von Neumann architecture generally has the execution cycles described in Figure 9.3. The fetch cycle obtains the opcode from the memory, and the control unit will decode the instruction to determine the operation. Next is the execute cycle. Based on the decoded information, execution will modify the content of the register or the memory. Once this is completed, the process will fetch the next instruction and continue. The processor operates one instruction at a time in a serial fashion.

To improve the speed of the processor operation, the Harvard architecture takes advantage of a common DS processor, in which one register holds the filter coefficient while the other register holds the data to be processed, as depicted in Figure 9.4.

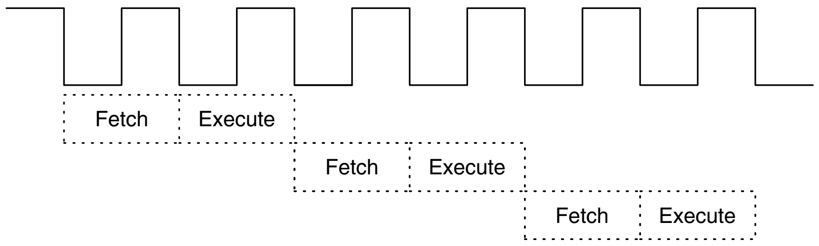


FIGURE 9.3 **Execution cycle based on the Von Neumann architecture.**

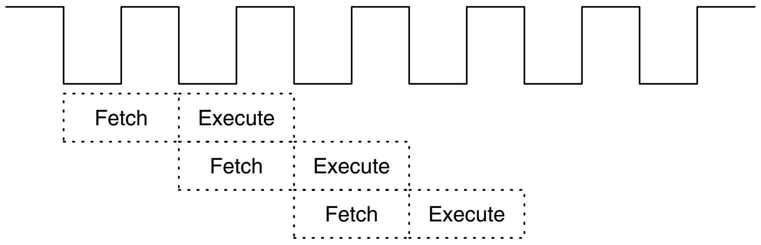


FIGURE 9.4 **Execution cycle based on the Harvard architecture.**

As shown in Figure 9.4, the execute and fetch cycles are overlapped. We call this the *pipelining* operation. The DS processor performs one execution cycle while also fetching the next instruction to be executed. Hence, the processing speed is dramatically increased.

The Harvard architecture is preferred for all DS processors due to the requirements of most DSP algorithms, such as filtering, convolution, and FFT, which need repetitive arithmetic operations, including multiplications, additions, memory access, and heavy data flow through the CPU.

For other applications, such as those dependent on simple microcontrollers with less of a timing requirement, the Von Neumann architecture may be a better choice, since it offers much less silica area and is thus less expansive.

9.2 Digital Signal Processor Hardware Units

In this section, we will briefly discuss special DS processor hardware units.

9.2.1 Multiplier and Accumulator

As compared with the general microprocessors based on the Von Neumann architecture, the DS processor uses the MAC, a special hardware unit for

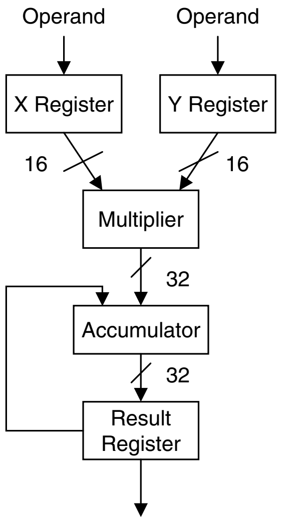


FIGURE 9.5 **The multiplier and accumulator (MAC) dedicated to DSP.**

enhancing the speed of digital filtering. This is dedicated hardware, and the corresponding instruction is generally referred to as MAC operation. The basic structure of the MAC is shown in Figure 9.5.

As shown in Figure 9.5, in a typical hardware MAC, the multiplier has a pair of input registers, each holding the 16-bit input to the multiplier. The result of the multiplication is accumulated in a 32-bit accumulator unit. The result register holds the double precision data from the accumulator.

9.2.2 Shifters

In digital filtering, to prevent overflow, a scaling operation is required. A simple scaling-down operation shifts data to the right, while a scaling-up operation shifts data to the left. Shifting data to the right is the same as dividing the data by 2 and truncating the fraction part; shifting data to the left is equivalent to multiplying the data by 2. As an example, for a 3-bit data word $011_2 = 3_{10}$, shifting 011 to the right gives $001_2 = 1$, that is, $3/2 = 1.5$, and truncating 1.5 results in 1. Shifting the same number to the left, we have $110_2 = 6_{10}$, that is, $3 \times 2 = 6$. The DS processor often shifts data by several bits for each data word. To speed up such operation, the special hardware shift unit is designed to accommodate the scaling operation, as depicted in Figure 9.2.

9.2.3 Address Generators

The DS processor generates the addresses for each datum on the data buffer to be processed. A special hardware unit for circular buffering is used (see the address generator in Figure 9.2). Figure 9.6 describes the basic mechanism of circular buffering for a buffer having eight data samples.

In circular buffering, a pointer is used and always points to the newest data sample, as shown in the figure. After the next sample is obtained from analog-to-digital conversion (ADC), the data will be placed at the location of $x(n-7)$, and the oldest sample is pushed out. Thus, the location for $x(n-7)$ becomes the location for the current sample. The original location for $x(n)$ becomes a location for the past sample of $x(n-1)$. The process continues according to the mechanism just described. For each new data sample, only one location on the circular buffer needs to be updated.

The circular buffer acts like a first-in/first-out (FIFO) buffer, but each datum on the buffer does not have to be moved. Figure 9.7 gives a simple illustration of the 2-bit circular buffer. In the figure, there is data flow to the ADC (*a, b, c, d, e, f, g, ...*) and a circular buffer initially containing *a, b, c,* and *d*. The pointer specifies the current data of *d*, and the equivalent FIFO buffer is shown on the right side with a current data of *d* at the top of the memory. When *e* comes in, as shown in the middle drawing in Figure 9.7, the circular buffer will change the pointer to the next position and update old *a* with a new datum *e*. It costs the pointer only one movement to update one datum in one step. However, on the right side, the FIFO has to move each of the other data down to let in the new

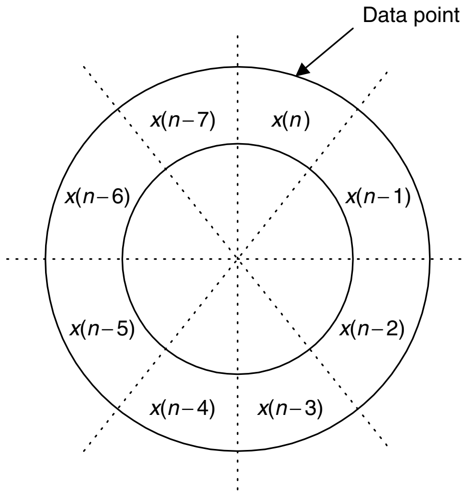


FIGURE 9.6 Illustration of circular buffering.

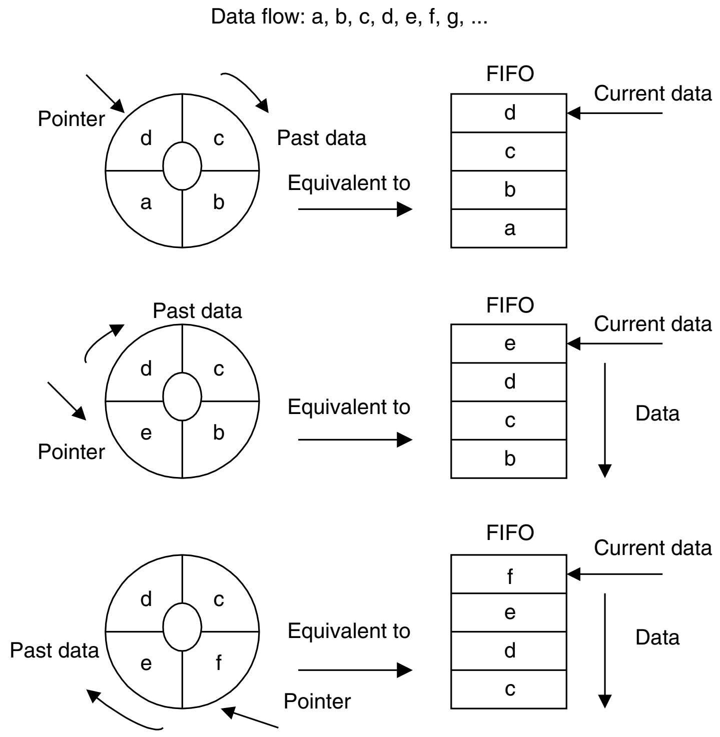


FIGURE 9.7 **Circular buffer and equivalent FIFO.**

datum *e* at the top. For this FIFO, it takes four data movements. In the bottom drawing in Figure 9.7, the incoming datum *f* for both the circular buffer and the FIFO buffer continues to confirm our observations.

Like finite impulse response (FIR) filtering, the data buffer size can reach several hundreds. Hence, using the circular buffer will significantly enhance the processing speed.

9.3 Digital Signal Processors and Manufacturers

DS processors are classified for general DSP and special DSP. The general-DSP processor is designed and optimized for applications such as digital filtering, correlation, convolution, and FFT. In addition to these applications, the special-DSP processor has features that are optimized for unique applications such as audio processing, compression, echo cancellation, and adaptive filtering. Here, we will focus on the general-DSP processor.

The major manufacturers in the DSP industry are Texas Instruments (TI), Analog Devices, and Motorola. TI and Analog Devices offer both fixed-point DSP families and floating-point DSP families, while Motorola offers fixed-point DSP families. We will concentrate on TI families, review their architectures, and study real-time implementation using the fixed- and floating-point formats.

9.4 Fixed-Point and Floating-Point Formats

In order to process real-world data, we need to select an appropriate DS processor, as well as a DSP algorithm or algorithms for a certain application. Whether a DS processor uses a fixed- or floating-point method depends on how the processor's CPU performs arithmetic. A fixed-point DS processor represents data in *2's complement integer format* and manipulates data using integer arithmetic, while a floating-point processor represents numbers using a mantissa (fractional part) and an exponent in addition to the integer format and operates data using floating-point arithmetic (discussed in a later section).

Since the fixed-point DS processor operates using the integer format, which represents only a very narrow dynamic range of the integer number, a problem such as overflow of data manipulation may occur. Hence, we need to spend much more coding effort to deal with such a problem. As we shall see, we may use floating-point DS processors, which offer a wider dynamic range of data, so that coding becomes much easier. However, the floating-point DS processor contains more hardware units to handle the integer arithmetic and the floating-point arithmetic, hence is more expensive and slower than fixed-point processors in terms of instruction cycles. It is usually a choice for prototyping or proof-of-concept development.

When it is time to make the DSP an application-specific integrated circuit (ASIC), a chip designed for a particular application, a dedicated hand-coded fixed-point implementation can be the best choice in terms of performance and small silica area.

The formats used by DSP implementation can be classified as fixed or floating point.

9.4.1 Fixed-Point Format

We begin with 2's complement representation. Considering a 3-bit 2's complement, we can represent all the decimal numbers shown in Table 9.1.

TABLE 9.1 A 3-bit 2's complement number representation.

Decimal Number	2's Complement
3	011
2	010
1	001
0	000
−1	111
−2	110
−3	101
−4	100

Let us review the 2's complement number system using Table 9.1. Converting a decimal number to its 2's complement requires the following steps:

1. Convert the magnitude in the decimal to its binary number using the required number of bits.
2. If the decimal number is positive, its binary number is its 2's complement representation; if the decimal number is negative, perform the 2's complement operation, where we negate the binary number by changing the logic 1's to logic 0's and logic 0's to logic 1's and then add a logic 1 to the data. For example, a decimal number of 3 is converted to its 3-bit 2's complement as 011; however, for converting a decimal number of -3, we first get a 3-bit binary number for the magnitude in the decimal, that is, 011. Next, negating the binary number 011 yields the binary number 100. Finally, adding a binary logic 1 achieves the 3-bit 2's complement representation of -3, that is, $100 + 1 = 101$, as shown in Table 9.1.

As we see, a 3-bit 2's complement number system has a dynamic range from -4 to 3, which is very narrow. Since the basic DSP operations include multiplications and additions, results of operation can cause overflow problems. Let us examine the multiplications in Example 9.1.

Example 9.1.

Given

1. $2 \times (-1)$
2. $2 \times (-3)$,

a. Operate each using its 2's complement.

a. 1.
$$\begin{array}{r} 010 \\ \times\ 001 \\ \hline 010 \\ 000 \\ +\ 000 \\ \hline 00010 \end{array}$$

and 2’s complement of 00010 = 11110. Removing two extended sign bits gives 110.
The answer is 110 (−2), which is within the system.

2.
$$\begin{array}{r} 010 \\ \times\ 011 \\ \hline 010 \\ 010 \\ +\ 000 \\ \hline 00110 \end{array}$$

and 2’s complement of 00110 = 11010. Removing two extended sign bits achieves 010.
Since the binary number 010 is 2, which is not (−6) as we expect, overflow occurs; that is, the result of the multiplication (−6) is out of our dynamic range (−4 to 3).

Let us design a system treating all the decimal values as fractional numbers, so that we obtain the fractional binary 2’s complement system shown in Table 9.2.

To become familiar with the fractional binary 2’s complement system, let us convert a positive fraction number $\frac{3}{4}$ and a negative fraction number $-\frac{1}{4}$ in decimals to their 2’s complements. Since

$$\frac{3}{4} = 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2},$$

its 2’s complement is 011. Note that we did not mark the binary point for clarity. Again, since

$$\frac{1}{4} = 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2},$$

TABLE 9.2 A 3-bit 2's complement system using fractional representation.

Decimal Number	Decimal Fraction	2's Complement
3	3/4	0.11
2	2/4	0.10
1	1/4	0.01
0	0	0.00
−1	−1/4	1.11
−2	−2/4	1.10
−3	−3/4	1.01
−4	−4/4 = −1	1.00

its positive-number 2's complement is 001. For the negative number, applying the 2's complement to the binary number 001 leads to $110 + 1 = 111$, as we see in Table 9.2.

Now let us focus on the fractional binary 2's complement system. The data are normalized to the fractional range from -1 to $1 - 2^{-2} = \frac{3}{4}$. When we carry out multiplications with two fractions, the result should be a fraction, so that multiplication overflow can be prevented. Let us verify the multiplication $(010) \times (101)$, which is the overflow case in Example 9.1:

$$\begin{array}{r} 0.1\,0 \\ \times\ 0.1\,1 \\ \hline 0\,1\,0 \\ 0\,1\,0 \\ +\,0\,0\,0 \\ \hline 0.0\,1\,1\,0 \end{array}$$

2's complement of $0.0110 = 1.1010$.
The answer in decimal form should be

$$1.1010 = (-1) \times (0.0110)_2 = -\left(0 \times (2)^{-1} + 1 \times (2)^{-2} + 1 \times (2)^{-3} + 0 \times (2)^{-4}\right)$$
$$= -\frac{3}{8}.$$

This number is correct, as we can verify from Table 9.2, that is, $\left(\frac{2}{4} \times \left(-\frac{3}{4}\right)\right) = -\frac{3}{8}$.

If we truncate the last two least-significant bits to keep the 3-bit binary number, we have an approximated answer as

$$1.10 = (-1) \times (0.10)_2 = -\left(1 \times (2)^{-1} + 0 \times (2)^{-2}\right) = -\frac{1}{2}.$$

The truncation error occurs. The error should be bounded by $2^{-2} = \frac{1}{4}$. We can verify that

$$|-1/2 - (-3/8)| = 1/8 < 1/4.$$

To use such a scheme, we can avoid the overflow due to multiplications but cannot prevent the additional overflow. In the following addition example,

$$\begin{array}{r} 0.11 \\ +\ 0.01 \\ \hline 1.00 \end{array}$$

where the result 1.00 is a negative number.

Adding two positive fractional numbers yields a negative number. Hence, overflow occurs. We see that this signed fractional number scheme partially solves the overflow in multiplications. Such fractional number format is called the signed Q-2 format, where there are 2 magnitude bits plus one sign bit. The additional overflow will be tackled using a scaling method discussed in a later section.

Q-format number representation is the most common one used in fixed-point DSP implementation. It is defined in Figure 9.8.

As indicated in Figure 9.8, Q-15 means that the data are in a sign magnitude form in which there are 15 bits for magnitude and one bit for sign. Note that after the sign bit, the dot shown in Figure 9.8 implies the binary point. The number is normalized to the fractional range from -1 to 1. The range is divided into 2^{16} intervals, each with a size of 2^{-15}. The most negative number is -1, while the most positive number is $1 - 2^{-15}$. Any result from multiplication is within the fractional range of -1 to 1. Let us study the following examples to become familiar with Q-format number representation.

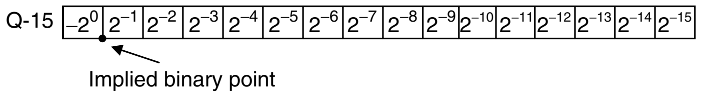


FIGURE 9.8 Q-15 (fixed-point) format.

Example 9.2.

a. Find the signed Q-15 representation for the decimal number 0.560123.

Solution:

a. The conversion process is illustrated using Table 9.3. For a positive fractional number, we multiply the number by 2 if the product is larger than 1, carry bit 1 as a most-significant bit (MSB), and copy the fractional part to the next line for the next multiplication by 2; if the product is less than 1, we carry bit 0 to MSB. The procedure continues to collect all 15 magnitude bits.

We yield the Q-15 format representation as

$$0.100011110110010.$$

Since we use only 16 bits to represent the number, we may lose accuracy after conversion. Like quantization, the truncation error is introduced. However, this error should be less than the interval size, in this case, $2^{-15} = 0.000030517$. We shall verify this in Example 9.5. An alternative way of conversion is to convert a fraction, let's say $\frac{3}{4}$, to Q-2 format, multiply it by 2^2, and then convert the truncated integer to its binary, that is,

$$(3/4) \times 2^2 = 3 = 011_2.$$

TABLE 9.3 Conversion process of Q-15 representation.

Number	Product	Carry
0.560123 × 2	1.120246	1 (MSB)
0.120246 × 2	0.240492	0
0.240492 × 2	0.480984	0
0.480984 × 2	0.961968	0
0.961968 × 2	1.923936	1
0.923936 × 2	1.847872	1
0.847872 × 2	1.695744	1
0.695744 × 2	1.391488	1
0.391488 × 2	0.782976	0
0.782976 × 2	1.565952	1
0.565952 × 2	1.131904	1
0.131904 × 2	0.263808	0
0.263808 × 2	0.527616	0
0.527616 × 2	1.055232	1
0.055232 × 2	0.110464	0 (LSB)

MSB, most-significant bit; LSB, least-significant bit.

In this way, it follows that

$$(0.560123) \times 2^{15} = 18354.$$

Converting 18354 to its binary representation will achieve the same answer. The next example illustrates the signed Q-15 representation for a negative number.

Example 9.3.

a. Find the signed Q-15 representation for the decimal number −0.160123.

Solution:

a. Converting the Q-15 format for the corresponding positive number with the same magnitude using the procedure described in Example 9.2, we have

$$0.160123 = 0.001010001111110.$$

Then, after applying 2's complement, the Q-15 format becomes

$$-0.160123 = 1.110101110000010.$$

Alternative way: Since $(-0.160123) \times 2^{15} = -5246.9$, converting the truncated number −5246 to its 16-bit 2's complement yields 1110101110000010.

Example 9.4.

a. Convert the Q-15 signed number 1.110101110000010 to the decimal number.

Solution:

a. Since the number is negative, applying the 2's complement yields

$$0.001010001111110.$$

Then the decimal number is

$$-(2^{-3} + 2^{-5} + 2^{-9} + 2^{-10} + 2^{-11} + 2^{-12} + 2^{-13} + 2^{-14}) = -0.160095.$$

Example 9.5.

a. Convert the Q-15 signed number 0.100011110110010 to the decimal number.

Solution:

a. The decimal number is

$$2^{-1} + 2^{-5} + 2^{-6} + 2^{-7} + 2^{-8} + 2^{-10} + 2^{-11} + 2^{-14} = 0.560120.$$

As we know, the truncation error in Example 9.2 is less than $2^{-15} = 0.000030517$. We verify that the truncation error is bounded by

$$|0.560120 - 0.560123| = 0.000003 < 0.000030517.$$

Note that the larger the number of bits used, the smaller the round-off error that may accompany it.

Examples 9.6 and 9.7 are devoted to illustrating data manipulations in the Q-15 format.

Example 9.6.

a. Add the two numbers in Examples 9.4 and 9.5 in Q-15 format.

Solution:

a. Binary addition is carried out as follows:

$$\begin{array}{r} 1.\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \\ +\ 0.\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0 \\ \hline 1\ 0.\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0 \end{array}.$$

Then the result is

$$0.\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0.$$

This number in the decimal form can be found to be

$$2^{-2} + 2^{-3} + 2^{-6} + 2^{-7} + 2^{-10} + 2^{-11} + 2^{-13} = 0.400024.$$

Example 9.7.

This is a simple illustration of fixed-point multiplication.

a. Determine the fixed-point multiplication of 0.25 and 0.5 in Q-3 fixed-point 2's complement format.

Solution:

a. Since $0.25 = 0.010$ and $0.5 = 0.100$, we carry out binary multiplication as follows:

$$\begin{array}{r} 0.010 \\ \times\ 0.100 \\ \hline 0\ 000 \\ 00\ 00 \\ 001\ 0 \\ +\ 0\ 000 \\ \hline 0.001\ 000 \end{array}$$

Truncating the least-significant bits to convert the result to Q-3 format, we have

$$0.010 \times 0.100 = 0.001.$$

Note that $0.001 = 2^{-3} = 0.125$. We can also verify that $0.25 \times 0.5 = 0.125$.

As a result, the Q-format number representation is a better choice than the 2's complement integer representation. But we need to be concerned with the following problems.

1. Converting a decimal number to its Q-N format, where N denotes the number of magnitude bits, we may lose accuracy due to the truncation error, which is bounded by the size of the interval, that is, 2^{-N}.
2. Addition and subtraction may cause overflow, where adding two positive numbers leads to a negative number, or adding two negative numbers yields a positive number; similarly, subtracting a positive number from a negative number gives a positive number, while subtracting a negative number from a positive number results in a negative number.
3. Multiplying two numbers in Q-15 format will lead to a Q-30 format, which has 31 bits in total. As in Example 9.7, the multiplication of Q-3 yields a Q-6 format, that is, 6 magnitude bits and a sign bit. In practice, it is common for a DS processor to hold the multiplication result using a double word size such as MAC operation, as shown in Figure 9.9 for multiplying two numbers in Q-15 format. In Q-30 format, there is one sign-extended bit. We may get rid of it by shifting left by one bit to obtain Q-31 format and maintaining the Q-31 format for each MAC operation.

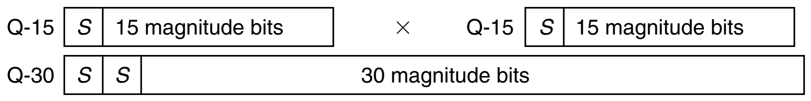


FIGURE 9.9 **Sign bit extended Q-30 format.**

Sometimes, the number in Q-31 format needs to be converted to Q-15; for example, the 32-bit data in the accumulator needs to be sent for 16-bit digital-to-analog conversion (DAC), where the upper most-significant 16 bits in the Q-31 format must be used to maintain accuracy. We can shift the number in Q-30 to the right by 15 bits or shift the Q-31 number to the right by 16 bits. The useful result is stored in the lower 16-bit memory location. Note that after truncation, the maximum error is bounded by the interval size of 2^{-15}, which satisfies most applications. In using the Q-format in the fixed-point DS processor, it is costive to maintain the accuracy of data manipulation.

4. Underflow can happen when the result of multiplication is too small to be represented in the Q-format. As an example, in the Q-2 system shown in Table 9.2, multiplying 0.01×0.01 leads to 0.0001. To keep the result in Q-2, we truncate the last two bits of 0.0001 to achieve 0.00, which is zero. Hence, underflow occurs.

9.4.2 Floating-Point Format

To increase the dynamic range of number representation, a floating-point format, which is similar to scientific notation, is used. The general format for floating-point number representation is given by

$$x = M \cdot 2^E, \tag{9.1}$$

where M is the mantissa, or fractional part, in Q format, and E is the exponent. The mantissa and exponent are signed numbers. If we assign 12 bits for the mantissa and 4 bits for the exponent, the format looks like Figure 9.10.

Since the 12-bit mantissa has limits between -1 and $+1$, the dynamic range is controlled by the number of bits assigned to the exponent. The bigger the number of bits assigned to the exponent, the larger the dynamic range. The number of bits for the mantissa defines the interval in the normalized range; as shown in Figure 9.10, the interval size is 2^{-11} in the normalized range, which is smaller than the Q-15. However, when more mantissa bits are used, the smaller interval size will be achieved. Using the format in Figure 9.10, we can determine the most negative and most positive numbers as:

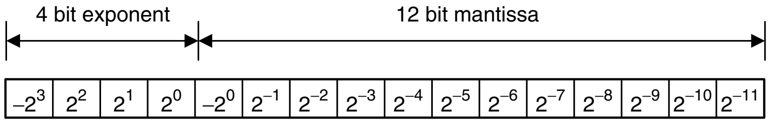


FIGURE 9.10 Floating-point format.

Most negative number $= (1.00000000000)_2 \cdot 2^{0111_2} = (-1) \times 2^7 = -128.0$

Most positive number $= (0.11111111111)_2 \cdot 2^{0111_2} = \left(1 - 2^{-11}\right) \times 2^7 = 127.9375.$

The smallest positive number is given by

Smallest positive number $= (0.00000000001)_2 \cdot 2^{1000_2} = \left(2^{-11}\right) \times 2^{-8} = 2^{-19}.$

As we can see, the exponent acts like a scale factor to increase the dynamic range of the number representation. We study the floating-point format in the following example.

Example 9.8.

a. Convert each of the following decimal numbers to the floating-point number using the format specified in Figure 9.10.

1. 0.1601230
2. −20.430527

Solution:

a. 1. We first scale the number 0.1601230 to $0.160123/2^{-2} = 0.640492$ with an exponent of -2 (other choices could be 0 or -1) to get $0.160123 = 0.640492 \times 2^{-2}$. Using 2's complement, we have $-2 = 1110$. Now we convert the value 0.640492 using Q-11 format to get 010100011111. Cascading the exponent bits and the mantissa bits yields

$$1110010100011111.$$

2. Since $-20.430527/2^5 = -0.638454$, we can convert it into the fractional part and exponent part as $-20.430527 = -0.638454 \times 2^5$.

Note that this conversion is not particularly unique; the forms $-20.430527 = -0.319227 \times 2^6$ and $-20.430527 = -0.1596135 \times 2^7 \ldots$ are still valid choices. Let us keep what we have now. Therefore, the exponent bits should be 0101. Converting the number 0.638454 using Q-11 format gives:

$$010100011011.$$

Using 2's complement, we obtain the representation for the decimal number -0.638454 as

$$101011100101.$$

Cascading the exponent bits and mantissa bits, we achieve

$$0101101011100101.$$

The floating arithmetic is more complicated. We must obey the rules for manipulating two floating-point numbers. Rules for arithmetic addition are given as:

$$x_1 = M_1 2^{E_1}$$
$$x_2 = M_2 2^{E_2}.$$

The floating-point sum is performed as follows:

$$x_1 + x_2 = \begin{cases} \left(M_1 + M_2 \times 2^{-(E_1-E_2)}\right) \times 2^{E_1}, & \text{if } E_1 \geq E_2 \\ \left(M_1 \times 2^{-(E_2-E_1)} + M_2\right) \times 2^{E_2} & \text{if } E_1 < E_2 \end{cases}$$

As a multiplication rule, given two properly normalized floating-point numbers:

$$x_1 = M_1 2^{E_1}$$
$$x_2 = M_2 2^{E_2},$$

where $0.5 \leq |M_1| < 1$ and $0.5 \leq |M_2| < 1$. Then multiplication can be performed as follows:

$$x_1 \times x_2 = (M_1 \times M_2) \times 2^{E_1+E_2} = M \times 2^E.$$

That is, the mantissas are multiplied while the exponents are added:

$$M = M_1 \times M_2$$
$$E = E_1 + E_2.$$

Examples 9.9 and 9.10 serve to illustrate manipulators.

Example 9.9.

a. Add two floating-point numbers achieved in Example 9.8:

$$1110\ 010100011111 = 0.640136718 \times 2^{-2}$$
$$0101\ 101011100101 = -0.638183593 \times 2^5.$$

Solution:

a. Before addition, we change the first number to have the same exponent as the second number, that is,

$$0101\ 000000001010 = 0.005001068 \times 2^5.$$

Then we add two mantissa numbers:

$$\begin{array}{rl} & 0.\,0\,0\,0\,0\,0\,0\,0\,1\,0\,1\,0 \\ + & 1.\,0\,1\,0\,1\,1\,1\,0\,0\,1\,0\,1 \\ \hline & 1.\,0\,1\,0\,1\,1\,1\,0\,1\,1\,1\,1 \end{array}$$

and we get the floating number as

$$0101\ 101011101111.$$

We can verify the result by the following:

$$\begin{aligned} 0101\ 101011101111 &= -\left(2^{-1} + 2^{-3} + 2^{-7} + 2^{-11}\right) \times 2^5 \\ &= -0.633300781 \times 2^5 = -20.265625. \end{aligned}$$

Example 9.10.

a. Multiply two floating-point numbers achieved in Example 9.8:

$$1110\ 010100011111 = 0.640136718 \times 2^{-2}$$
$$0101\ 101011100101 = -0.638183593 \times 2^5.$$

Solution:

a. From the results in Example 9.8, we have the bit patterns for these two numbers as

$$E_1 = 1110, E_2 = 0101, M_1 = 010100011111, M_2 = 101011100101.$$

Adding two exponents in 2's complement form leads to

$$E = E_1 + E_2 = 1110 + 0101 = 0011,$$

which is $+3$, as we expected, since in decimal domain $(-2) + 5 = 3$.
As previously shown in the multiplication rule, when multiplying two mantissas, we need to apply their corresponding positive values. If the sign for the final value is negative, then we convert it to its 2's complement form. In our example, $M_1 = 010100011111$ is a positive mantissa. However, $M_2 = 101011100101$ is a negative mantissa, since the MSB is 1. To perform multiplication, we use 2's complement to convert M_2 to its positive value, 010100011011, and note that the multiplication result is negative. We multiply two positive mantissas and truncate the result to 12 bits to give

$$010100011111 \times 010100011011 = 001101000100.$$

Now we need to add a negative sign to the multiplication result with 2's complement operation. Taking 2's complement, we have

$$M = 110010111100.$$

Hence, the product is achieved by cascading the 4-bit exponent and 12-bit mantissa as:

$$0011\ 110010111100.$$

converting this number back to the decimal number, we verify the result to be $0.408203125 \times 2^3 = -3.265625$.

Next, we examine overflow and underflow in the floating-point number system.

Overflow

During operation, overflow will occur when a number is too large to be represented in the floating-point number system. Adding two mantissa numbers may lead to a number larger than 1 or less than -1; and multiplying two numbers causes the addition of their two exponents, so that the sum of the two exponents could overflow. Consider the following overflow cases.

Case 1. Add the following two floating-point numbers:

$$0111\ 011000000000 + 0111\ 010000000000.$$

Note that two exponents are the same and they are the biggest positive number in 4-bit 2's complement representation. We add two positive mantissa numbers as

$$\begin{array}{rl} & 0.\,1\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ + & 0.\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ \hline & 1.\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0. \end{array}$$

The result for adding mantissa numbers is negative. Hence, the overflow occurs.

Case 2. Multiply the following two numbers:

$$0111\ 011000000000 \times 0111\ 011000000000.$$

Adding two positive exponents gives

$$0111 + 0111 = 1000 \text{ (negative; the overflow occurs).}$$

Multiplying two mantissa numbers gives:

$$0.11000000000 \times 0.11000000000 = 0.10010000000 \text{ (OK!).}$$

Underflow

As we discussed before, underflow will occur when a number is too small to be represented in the number system. Let us divide the following two floating-point numbers:

$$1001\ 001000000000 \div 0111\ 010000000000.$$

First, subtracting two exponents leads to

$$\begin{aligned} 1001\,(\text{negative}) - 0111\,(\text{positive}) &= 1001 + 1001 \\ &= 0010\,(\text{positive; the underflow occurs}). \end{aligned}$$

Then, dividing two mantissa numbers, it follows that

$$0.01000000000 \div 0.10000000000 = 0.10000000000\ (\text{OK!}).$$

However, in this case, the expected resulting exponent is -14 in decimal, which is too small to be presented in the 4-bit 2's complement system. Hence the underflow occurs.

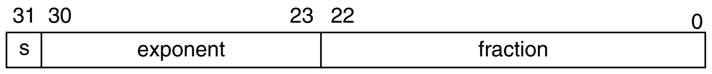
Understanding basic principles of the floating-point formats, we can next examine two floating-point formats of the Institute of Electrical and Electronics Engineers (IEEE).

9.4.3 IEEE Floating-Point Formats

Single Precision Format

IEEE floating-point formats are widely used in many modern DS processors. There are two types of IEEE floating-point formats (IEEE 754 standard). One is the IEEE single precision format, and the other is the IEEE double precision format. The single precision format is described in Figure 9.11.

The format of IEEE single precision floating-point standard representation requires 23 fraction bits F, 8 exponent bits E, and 1 sign bit S, with a total of 32 bits for each word. F is the mantissa in 2's complement positive binary fraction represented from bit 0 to bit 22. The mantissa is within the normalized range limits between $+1$ and $+2$. The sign bit S is employed to indicate the sign of the number, where when $S = 1$ the number is negative, and when $S = 0$ the number is positive. The exponent E is in excess 127 form. The value of 127 is the offset from the 8-bit exponent range from 0 to 255, so that E-127 will have a range from -127 to $+128$. The formula shown in Figure 9.11 can be applied to convert the IEEE 754 standard (single precision) to the decimal number. The following simple examples also illustrate this conversion:



$$x = (-1)^s \times (1.F) \times 2^{E-127}$$

FIGURE 9.11 IEEE single precision floating-point format.

$$0\ 10000000\ 00000000000000000000000 = (-1)^0 \times (1.0_2) \times 2^{128-127} = 2.0$$
$$0\ 10000001\ 10100000000000000000000 = (-1)^0 \times (1.101_2) \times 2^{129-127} = 6.51$$
$$1\ 10000001\ 10100000000000000000000 = (-1)^1 \times (1.101_2) \times 2^{129-127} = -6.5.$$

Let us look at Example 9.11 for more explanation.

Example 9.11.

a. Convert the following number in the IEEE single precision format to the decimal format:

$$110000000.010\ldots0000.$$

Solution:

a. From the bit pattern in Figure 9.11, we can identify the sign bit, exponent, and fractional as:

$$s = 1, E = 2^7 = 128$$
$$1.F = 1.01_2 = (2)^0 + (2)^{-2} = 1.25.$$

Then, applying the conversion formula leads to

$$x = (-1)^1(1.25) \times 2^{128-127} = -1.25 \times 2^1 = -2.5.$$

In conclusion, the value x represented by the word can be determined based on the following rules, including all the exceptional cases:

- If $E = 255$ and F is nonzero, then $x = NaN$ ("Not a number").
- If $E = 255$, F is zero, and S is 1, then $x = -$Infinity.
- If $E = 255$, F is zero, and S is 0, then $x = +$Infinity.
- If $0 < E < 255$, then $x = (-1)^s \times (1.F) \times 2^{E-127}$, where $1.F$ represents the binary number created by prefixing F with an implicit leading 1 and a binary point.

- If $E = 0$ and F is nonzero, then $x = (-1)^s \times (0.F) \times 2^{-126}$. This is an "unnormalized" value.
- If $E = 0$, F is zero, and S is 1, then $x = -0$.
- If $E = 0$, F is zero, and S is 0, then $x = 0$.

Typical and exceptional examples are shown as follows:

$$0\ 00000000\ 00000000000000000000000 = 0$$
$$1\ 00000000\ 00000000000000000000000 = -0$$
$$0\ 11111111\ 00000000000000000000000 = \text{Infinity}$$
$$1\ 11111111\ 00000000000000000000000 = -\text{Infinity}$$
$$0\ 11111111\ 00000100000000000000000 = \text{NaN}$$
$$1\ 11111111\ 00100010001001010101010 = \text{NaN}$$
$$0\ 00000001\ 00000000000000000000000 = (-1)^0 \times (1.0_2) \times 2^{1-127} = 2^{-126}$$
$$0\ 00000000\ 10000000000000000000000 = (-1)^0 \times (0.1_2) \times 2^{0-126} = 2^{-127}$$
$$0\ 00000000\ 00000000000000000000001 =$$
$$(-1)^0 \times (0.00000000000000000000001_2) \times 2^{0-126} = 2^{-149}(\text{smallest positive value})$$

Double Precision Format

The IEEE double precision format is described in Figure 9.12.

The IEEE double precision floating-point standard representation requires a 64-bit word, which may be numbered from 0 to 63, left to right. The first bit is the sign bit S, the next eleven bits are the exponent bits E, and the final 52 bits are the fraction bits F. The IEEE floating-point format in double precision significantly increases the dynamic range of number representation, since there are eleven exponent bits; the double precision format also reduces the interval size in the mantissa normalized range of +1 to +2, since there are 52 mantissa bits as compared with the single precision case of 23 bits. Applying the conversion formula shown in Figure 9.12 is similar to the single precision case.

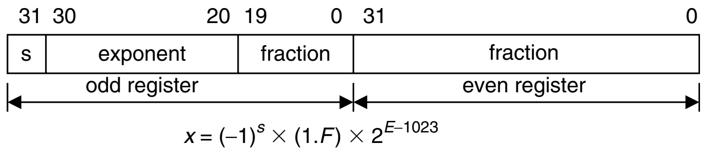


FIGURE 9.12 IEEE double precision floating-point format.

Example 9.12.

a. Convert the following number in IEEE double precision format to the decimal format:

$$001000\ldots0.110\ldots0000$$

Solution:

b. Using the bit pattern in Figure 9.12, we have

$$s = 0, E = 2^9 = 512 \text{ and}$$
$$1.F = 1.11_2 = (2)^0 + (2)^{-1} + (2)^{-2} = 1.75^{\cdot}$$

Then, applying the double precision formula yields

$$x = (-1)^0(1.75) \times 2^{512-1023} = 1.75 \times 2^{-511} = 2.6104 \times 10^{-154}.$$

For purposes of completeness, rules for determining the value x represented by the double precision word are listed as follows:

- If $E = 2047$ and F is nonzero, then $x = NaN$ ("Not a number")
- If $E = 2047$, F is zero, and S is 1, then $x = -\text{Infinity}$
- If $E = 2047$, F is zero, and S is 0, then $x = +\text{Infinity}$
- If $0 < E < 2047$, then $x = (-1)^s \times (1.\text{F}) \times 2^{E-1023}$, where $1.F$ is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point
- If $E = 0$ and F is nonzero, then $x = (-1)^s \times (0.F) \times 2^{-1022}$. This is an "unnormalized" value
- If $E = 0$, F is zero, and S is 1, then $x = -0$
- If $E = 0$, F is zero, and S is 0, then $x = 0$

9.4.5 Fixed-Point Digital Signal Processors

Analog Device, Texas Instruments, and Motorola all manufacture fixed-point DS processors. Analog Devices offers a fixed-point DSP family such as ADSP21xx. Texas Instruments provides various generations of fixed-point DSP processors based on historical development, architectural features, and computational performance. Some of the most common ones are TMS320C1x (first generation), TMS320C2x, TMS320C5x, and TMS320C62x. Motorola manufactures varieties of fixed-point processors, such as the DSP5600x family. The new families of fixed-point DS processors are expected to continue to grow.