CHAPTER

HARDWARE AND SOFTWARE FOR DIGITAL SIGNAL PROCESSORS 14

CHAPTER OUTLINE

14.1 DIGITAL SIGNAL PROCESSOR ARCHITECTURE

Unlike microprocessors and microcontrollers, digital signal processors (DSPs) 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, DSPs use different dedicated hardware architecture.

We first compare the architecture of the general microprocessor with that of the DSPs. The design of general microprocessors and microcontrollers is based on the *Von Neumann architecture*, which was developed based on a research paper written by John Von Neumann and others in 1946. Von Neumann suggested that computer instructions, as we discuss, should be numerical codes instead of special wiring. Fig. 14.1 shows the Von Neumann architecture.

As shown in Fig. 14.1, 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 what operation to do, and 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 instruction or piece of data can be retrieved at a time. Since the processor proceeds in a serial fashion, it causes most units staying 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 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, DSPs 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. Fig. 14.2 shows today's Harvard architecture. As depicted, the DSP has two separate memory spaces. One is dedicated for the program code, while the other is employed for data. Hence, to

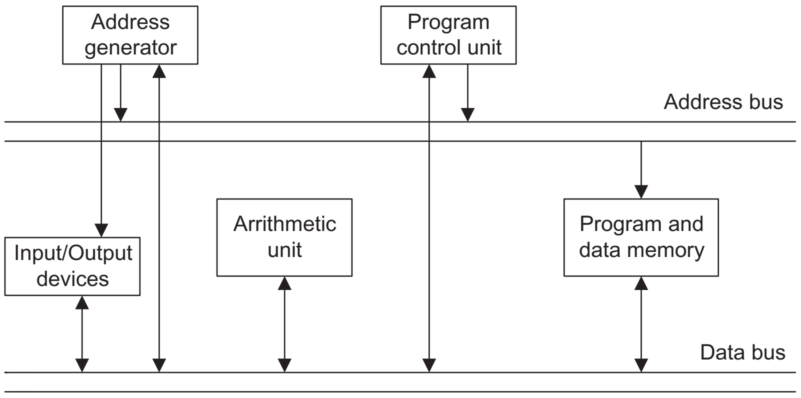


FIG. 14.1

General microprocessor based on the Von Neumann architecture.

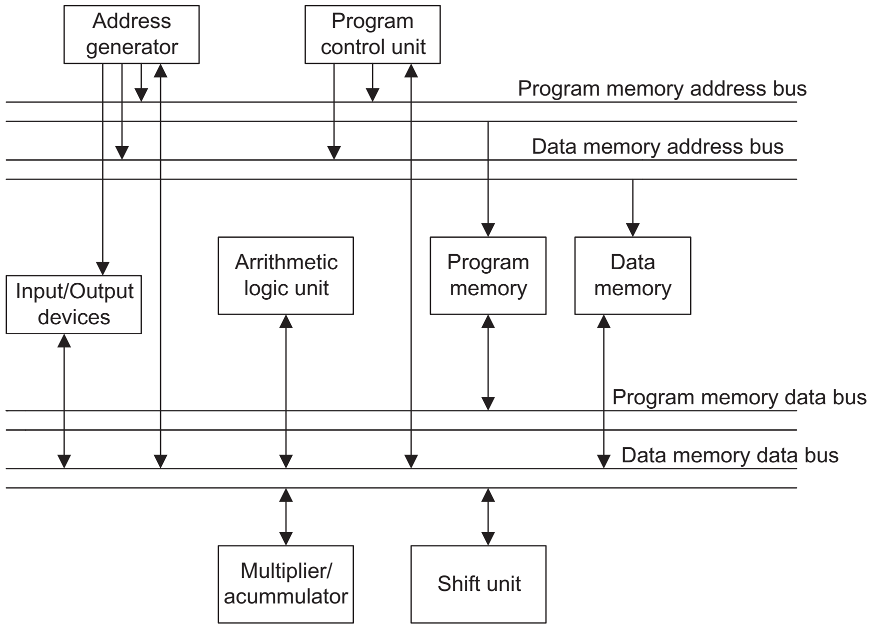


FIG. 14.2

DSPs based on the Harvard architecture.

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 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 the 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 Fig. 14.3. The fetch cycle obtains the opcode from the memory,

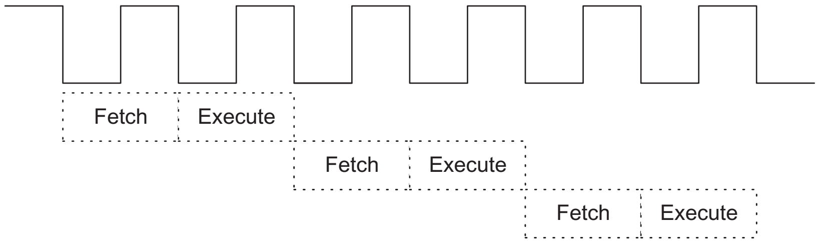


FIG. 14.3

Execution cycle based on the Von Neumann architecture.

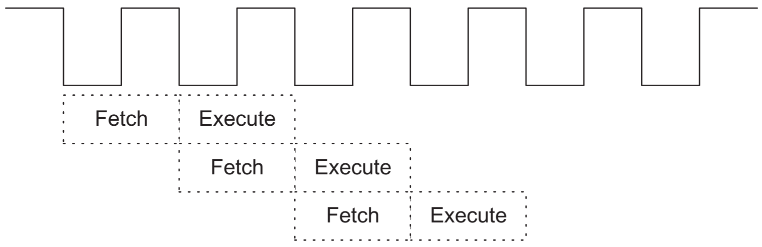


FIG. 14.4

Execution cycle based on Harvard architecture.

and the control unit will decode the instruction to determine the operation. Next is the execute cycle. Based 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 DSP, in which one register holds the filter coefficient while the other register holds the data to be processed, as depicted in Fig. 14.4.

As shown in Fig. 14.4, the execute and fetch cycles are overlapped. We call this the *pipelining* operation. The DSP 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 DSPs 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 the 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.

14.2 DSP HARDWARE UNITS

In this section, we briefly discuss special DSP hardware units.

14.2.1 MULTIPLIER AND ACCUMULATOR

As compared with the general microprocessors based on the Von Neumann architecture, the DSP uses the MAC, a special hardware unit for enhancing the speed of digital filtering. This is dedicated hardware, and the corresponding instruction is generally referred to as the MAC operation. The basic structure of the MAC is shown in Fig. 14.5.

As shown in Fig. 14.5, 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 the 32-bit accumulator unit. The result register holds the double-precision data from the accumulator.

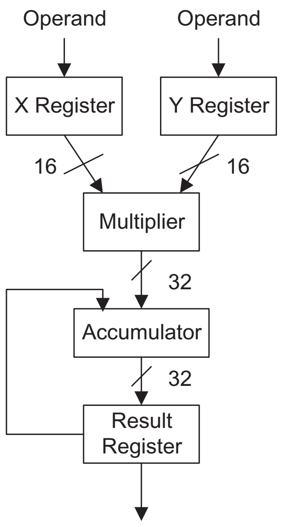


FIG. 14.5

The multiplier and accumulator (MAC) dedicated to DSP.

14.2.2 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 DSP 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 Fig. 14.2.

14.2.3 ADDRESS GENERATORS

The DSP 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 Fig. 14.2). Fig. 14.6 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. Fig. 14.7 gives a simple illustration of the 2-bit circular buffer. In the figure, there is

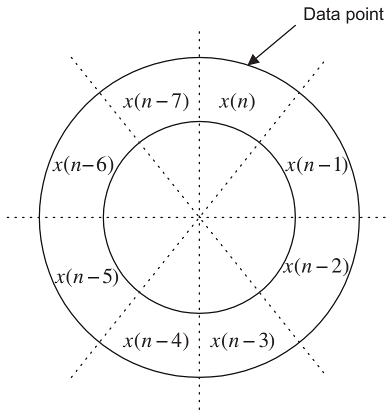


FIG. 14.6

Illustration of circular buffering.

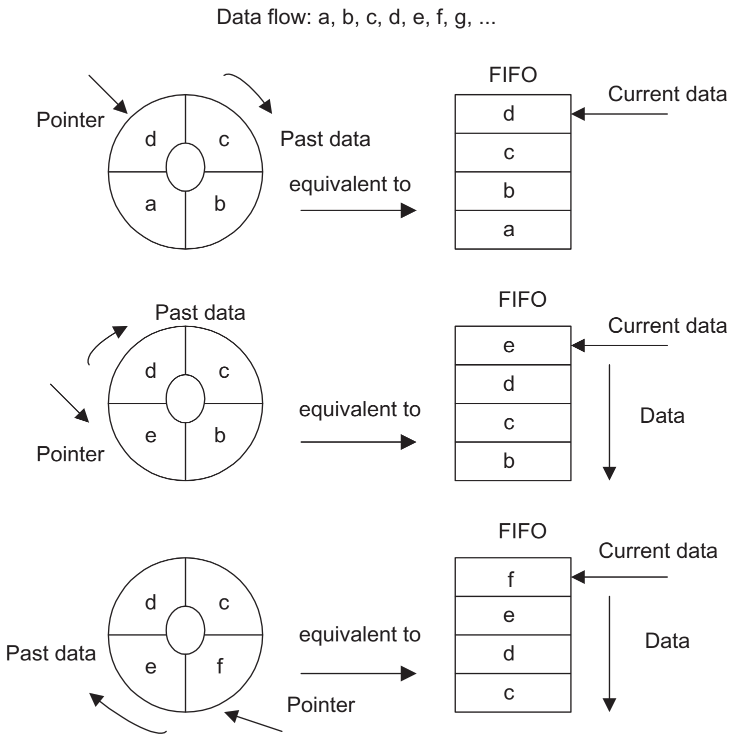


FIG. 14.7

Circular buffer and equivalent FIFO.

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 Fig. 14.7, 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 datum *e* at the top. For this FIFO, it takes four data movements. In the bottom drawing in Fig. 14.7, 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.

14.3 DSPs AND MANUFACTURES

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

The major manufactures 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 concentrate on TI families, review their architectures, and study real-time implementation using the fixed- and floating-point formats.

14.4 FIXED-POINT AND FLOATING-POINT FORMATS

In order to process real-world data, we need to select an appropriate DSP, as well as a DSP algorithm or algorithms for a certain application. Whether a DSP uses a fixed- or floating-point method depends on how the processor's CPU performs arithmetic. A fixed-point DSP represents data in *2's complement integer format* and manipulates data using integer arithmetic, while a floating-point processor represents number using a mantissa (fractional part) and an exponent in addition to the integer format and operates data using the floating-point arithmetic (discussed in Section 14.4.2).

Since the fixed-point DSP operates using the integer format, which represents only a very narrower dynamic range of the integer number, a problem such as the overflow of data manipulation may occur. Hence, we need to spend much more coding effort to deal with such a problem. As we see, we may use floating-point DSPs, which offer a wider dynamic range of data, so that coding becomes much easier. However, the floating-point DSP contains more hardware units to handle the integer arithmetic and the floating-point arithmetic, hence it 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 making 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 the DSP implementation can be classified as fixed or floating point.

14.4.1 FIXED-POINT FORMAT

We begin with two's complement representation. Considering a 3-bit two's complement (2's complement), we can represent all the decimal numbers shown in Table 14.1.

Let us review the 2's complement number system using Table 14.1. Converting a decimal number to its 2's complement form 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 representation as 011; however, for converting a decimal number of −3, we first get a 3-bit binary number for the magnitude in 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 14.1.

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 14.1.

EXAMPLE 14.1

Given

(a) $2 \times (-1)$

(b) $2 \times (-3)$,

operate each using its 2's complement.

Solution:

(a)

```
      0 1 0
    x 0 0 1
  ---------------
      0 1 0
    0 0 0
 +0 0 0
  ---------------
  0 0 0 1 0
```

The 2's complement of 00010 is 11,110. Removing two extended sign bits gives 110. The answer is 110 (−2), which is within the system.

(b)

```
         0 1 0
       x 0 1 1
    ---------------
         0 1 0
       0 1  0
    +0 0  0
   ---------------
     0 0 1 1 0
```

The 2’s complement of 00110 is 11,010. Removing two extended sign bits achieves 010. Since the binary number 010 is 2, which is not (−6) as what 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 14.2.

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.

Table 14.1 A 3-Bit 2’s Complement Number Representation

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

Table 14.2 A 3-Bit 2’s Complement System Using Fractional Representation

Decimal Number	Decimal Fraction	Two’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

Again, since

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

its positive-number 2's complement is 001. For the negative number, applying 2's complement to the binary number 001 leads to $110 + 1 = 111$, as we see in Table 14.2.

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 14.1. We first multiply two positive numbers:

```
        0 .1 0
      x 0 .1 1
    --------------
        0  1 0
      0  1  0
  + 0  0  0
  ---------------
    0 . 0  1  1  0
```

The 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 14.2, 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

$$1.10 = (-1) \times (0.01)_2 = -\left(0 \times (2)^{-1} + 1 \times (2)^{-2}\right) = -\frac{1}{2}.$$

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

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

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

```
        0 .1 1
      + 0 .0 1
    --------------
        1  0 0
  ---------------
        1. 0  0
```

where the result 1.00 is a negative number.

Adding two positive fractional numbers yields a negative number. Hence, overflow occurs.

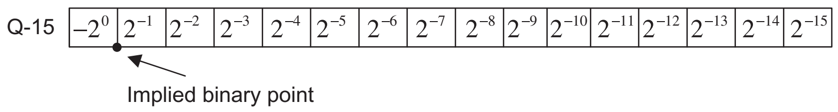


FIG. 14.8

Q-15 (fixed-point) format.

We see that this signed fractional number scheme partially solves the overflow in multiplications. This fractional number format is called the signed Q-2 format, where there are 2 magnitude bits plus one sign bit. The overflow in addition will be tackled using a scaling method discussed in the later section.

Q-format number representation is the most common one used in the fixed-point DSP implementation. It is defined in Fig. 14.8.

As indicated in Fig. 14.8, 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 Fig. 14.8 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.

EXAMPLE 14.2

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

Solution:

The conversion process is illustrated using Table 14.3. 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 <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 only use 16 bits to represent the number, we may lose accuracy after conversion. Like quantization, a truncation error is introduced. However, this error should be less than the interval size, in this case, $2^{-15}=0.0000305017$. We verify this in Example 14.5. An alternative way of conversion is to convert a fraction, let us convert $\frac{3}{4}$ to Q-2 format. We multiply $\frac{3}{4}$ by 2^2, and then convert the truncated integer to its binary, that is,

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

In this way, it follows that

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

Converting 18,354 to its binary representation will achieve the same answer.

Continued

EXAMPLE 14.2—CONT'D

Table 14.3 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.

The next example illustrates the signed Q-15 representation for a negative number.

EXAMPLE 14.3

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

Solution:

Converting the Q-15 format for the corresponding positive number with the same magnitude using the procedure described in Example 14.2, 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 1.110101110000010.

EXAMPLE 14.4

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

Solution:

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

$$0.001010001111110.$$

Then the decimal number is

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

EXAMPLE 14.5

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

Solution:

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 14.2 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 truncation error that may accompany it.

Examples 14.6 and 14.7 are devoted to illustrating data manipulations in the Q-15 format.

EXAMPLE 14.6

Add the two numbers in Examples 14.4 and 14.5 in Q-15 format.

Solution:

Binary addition is carried out as follows:

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

Then the result is:

$$0.011001100110100.$$

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 14.7

This is a simple illustration of the fixed-point multiplication.

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

Solution:

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

```
               0.010
           ×   0.100
          -----------
                0 0 0 0
              0 0 0 0
            0 0 1 0
       +  0 0 0 0
  --------------------------
         0.  0 0 1 0 0 0
```

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 $0.25 \times 0.5 = 0.125$.

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. When 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 number 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 14.7, 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 DSP to hold the multiplication result using a double word size such as the MAC operation, as shown in Fig. 14.9 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.

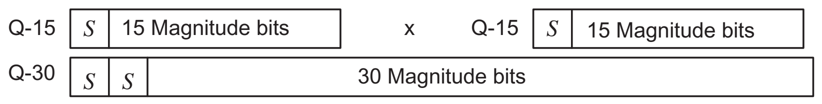


FIG. 14.9

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-30 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 DSP, it is costly 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 a Q-2 system shown in Table 14.2, 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.

14.4.2 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 the floating-point number representation is given by

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

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 Fig. 14.10.

Since the 12-bit mantissa has limits between -1 to +1, the dynamic range is controlled by the number of bits assigned to the exponent. The bigger the number of bits designated to the exponent, the larger the dynamic range. The number of bits for mantissa defines the interval in the normalized range; as shown in Fig. 14.10, 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 Fig. 14.10, we can determine the most negative and most positive numbers as:

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

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

The smallest positive number is given by

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

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.

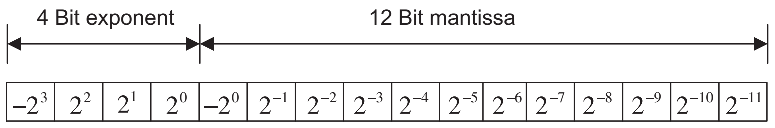


FIG. 14.10

Floating-point format.

EXAMPLE 14.8

Convert each of the following decimal numbers to the floating-point number using the format specified in Fig. 14.10.

(a) 0.1601230

(b) −20.430527

Solution:

(a) 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

$$1110\,010100011111.$$

(b) 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 $-20.430527=-0.1596135\times 2^7$ …. 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 −0.6438454 as

$$101011100101.$$

Cascading the exponent bits and mantissa bits, we achieve

$$0101\,101011100101$$

The floating-point 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:

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

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 $0.5 \le |M_1| < 1$ and $0.5 \le |M_2| < 1$. 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 14.9 and 14.10 serve to illustrate manipulations.

EXAMPLE 14.9

Add two floating-point numbers achieved in Example 14.8:

$$1110\,010100011111 = 0.640136718 \times 2^{-2}$$

$$0101\,101011100101 = -0.638183593 \times 2^{5}.$$

Solution:

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

$$0101\,000000001010 = 0.00500168 \times 2^{5}.$$

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:

$0101\ 101011101111 = -(2^{-1}+2^{-3}+2^{-7}+2^{-11}) \times 2^5 = -0.633300781 \times 2^5 = -20.265625.$

EXAMPLE 14.10

Multiply the two floating-point numbers achieved in Example 14.8:

$$1110\,010100011111 = 0.640136718 \times 2^{-2}$$

$$0101\,101011100101 = -0.638183593 \times 2^{5}.$$

Solution:

From the results in Example 14.8, we have the bit patterns for these two numbers as

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

Continued

EXAMPLE 14.10—CONT'D

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 the 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

$$010100011111 \times 010100011011 = 001101010100.$$

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

$$M = 110010101100.$$

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

$$0011110010101100.$$

Converting this number back to the decimal number, we can 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 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:

$$0111\,011000000000 + 0111\,01000000000$$

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:

$$0111\,011000000000 \times 0111\,01100000000$$

Adding two positive exponents gives

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

Multiplying two mantissa numbers gives

$$0.11000000000 \times 0.1100000000 = 0\,.10010000000 \text{ (OK!)}.$$

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:

$$1001\,001000000000 \div 0111\,01000000000.$$

First, subtracting two exponents leads to.

1001 (negative) – 0111 (positive) = 1001 + 1001 = 0010 (positive; the underflow occurs).

Then, dividing two mantissa numbers, it follows that

$$0.01000000000 \div 0.1000000000 = 0.10000000000 \text{ (OK!)}$$

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.

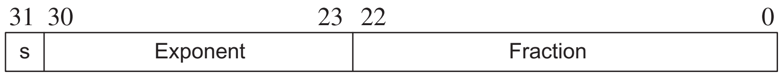
Now that we understand the basic principles of the floating-point formats, we can next examine two floating-point formats of the Institute of Electrical and Electronics Engineers (IEEE).

14.4.3 IEEE FLOATING-POINT FORMATS

Single-Precision Format:

IEEE floating-point formats are widely used in many modern DSPs. 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 Fig. 14.11.

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 Fig. 14.11 can be



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

FIG. 14.11

IEEE single-precision floating-point format.

applied to convert the IEEE 754 standard (single precision) to the decimal number. The following simple examples also illustrate this conversion.

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

Let us look at Example 14.11 for more explanation.

EXAMPLE 14.11

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

$$110000000.010\ldots0000.$$

Solution:

From the bit pattern in Fig. 14.11, we can identify the sign bit, exponent, and fractional as:

$$\begin{aligned}
&s = 1,\quad E = 2^7 = 128\\
&1.F = 1.01_2 = (2)^0 + (2)^{-2} = 1.25.
\end{aligned}$$

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:

$$\begin{aligned}
&0\ 00000000\ 00000000000000000000000 = 0\\
&1\ 00000000\ 00000000000000000000000 = -0
\end{aligned}$$

$$\begin{aligned}
&0\ 11111111\ 00000000000000000000000 = \text{Infinity}\\
&1\ 11111111\ 00000000000000000000000 = -\text{Infinity}
\end{aligned}$$

$$\begin{aligned}
&0\ 11111111\ 0000010000000000000000 = \text{NaN}\\
&1\ 11111111\ 0010001000100101010101 = \text{NaN}
\end{aligned}$$

$$
\begin{aligned}
0\,00000001\,00000000000000000000000 &= (-1)^0 \times (1.0_2) \times 2^{1-127} = 2^{-126} \\
0\,00000000\,10000000000000000000000 &= (-1)^0 \times (0.1_2) \times 2^{0-126} = 2^{-127} \\
0\,00000000\,00000000000000000000001 &= (-1)^0 \times (0.00000000000000000000001_2) \times 2^{0-126} \\
&= 2^{-149}\ \text{(smallest positive value)}
\end{aligned}
$$

Double-Precision Format:

The IEEE double-precision format is described in Fig. 14.12.

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 11 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 11 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 compare with the single-precision case of 23 bits. Applying the conversion formula shown in Fig. 14.12 is similar to the single-precision case.

EXAMPLE 14.12

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

$$001000\ldots0.110\ldots0000$$

Solution:

Using the bit pattern in Fig. 14.12, we have

$$s = 0,\quad E = 2^9 = 512 \text{ and}$$

$$1.F = 1.11_2 = (2)^0 + (2)^{-1} + (2)^{-2} = 1.75.$$

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 purpose 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 $x = -$Inifinity

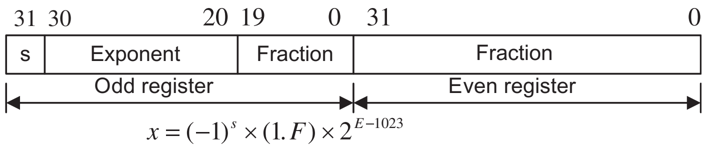


FIG. 14.12

IEEE double-precision floating-point format.

- If $E=2047$, F is zero, and S is 0, then $x=+\text{Inifinity}$
- If $0<E<2047$, then $x=(-1)^s \times (1.\,F) \times 2^{E-1023}$, 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 $x=0$.

14.4.4 FIXED-POINT DSPs

Analog Devices, TI, and Motorola all manufacture fixed-point DSPs. Analog Devices offers a fixed-point DSP family such as ADSP21xx. TI provides various generations of fixed-point DSPs based on historical development, architecture features, and computational performance. Some of the most common ones are TMS320C1x (first generation), TMS320C2x, TMS320C5x, and TMS320C62x. Motorola manufactures a variety of fixed-point processors, such as DSP5600x family. The new families of fixed-point DSPs are expected to continue to grow. Since they share some basic common features such as program memory and data memory with associated address buses, arithmetic logic units (ALUs), program control units, MACs, shift units, and address generators, here we focus on an overview of the TMS320C54x processor. The typical TMS320C54x fixed-point DSP architecture appears in Fig. 14.13.

The fixed-point TMS320C50 families supporting 16-bit data have on-chip program memory and data memory in various size and configuration. They include data RAM (random-access memory) and program ROM (read-only memory) used for program code, instruction, and data. Four data buses and four address buses are accommodated to work with the data memory and program memory. The program memory address bus and program memory data bus are responsible for fetching the program instruction. As shown in Fig. 14.13, the C and D data memory address buses and the C and D data memory data buses deal with fetching data from the data memory while the E data memory address bus and E data memory data bus are dedicated to move data into data memory. In addition, the E memory data bus can access the I/O devices.

Computational units consist of an ALU, an MAC, and a shift unit. For TMS320C54x families, the ALU can fetch data from the C, D, and program memory data buses and access the E memory data bus. It has two independent 40-bit accumulators, which are able to operate the 40-bit addition. The multiplier, which can fetch data from C and D memory data buses and write data via E data memory data bus, is capable of operating 17-bit $\times$ 17-bit multiplications. The 40-bit shifter has the same capability of bus access as the MAC, allowing all possible shifts for scaling and fractional arithmetic such as we have discussed for the Q-format.

The program control unit fetches instructions via the program memory data bus. Again, in order to speed up memory access, there are two address generators available: one responsible for program addresses and one for data addresses.

Advanced Harvard architecture is employed, where several instructions operate at the same time for given a given single instruction cycle. Processing performance offers 40 MIPS (million instruction sets per second). To further explore this subject, the reader is referred to Dahnoun (2000), Embree (1995), Ifeachor and Jervis (2002) and Van der Vegte (2002), as well as the web site (www.ti.com).

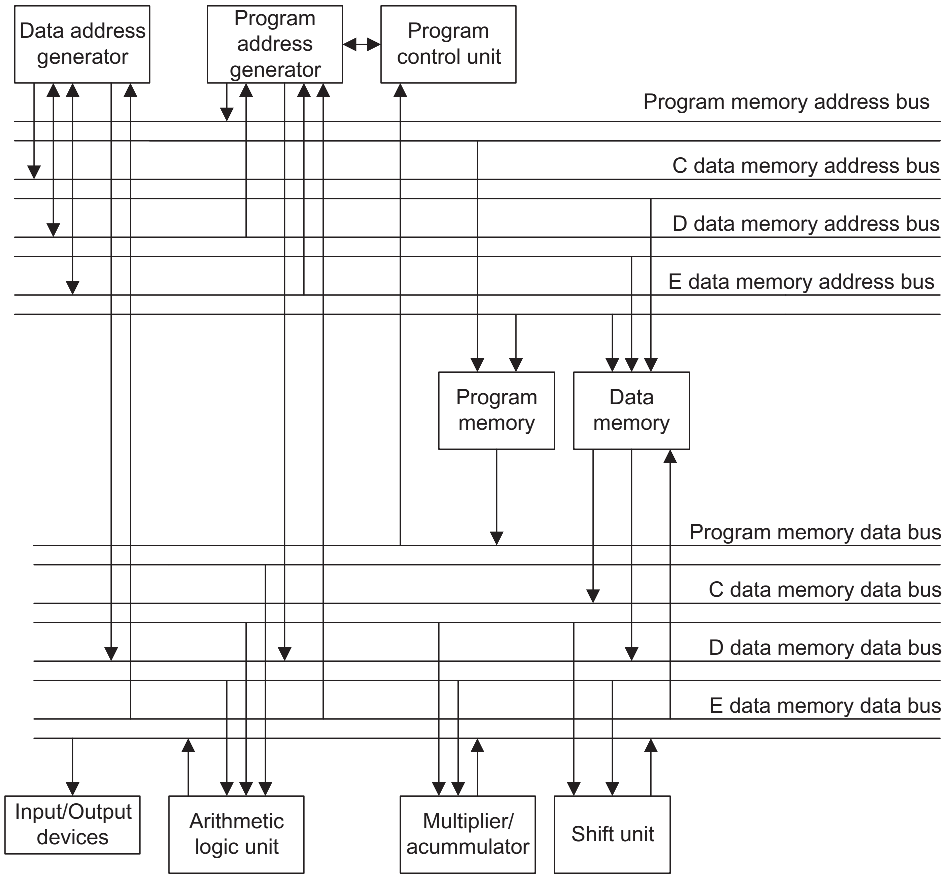


FIG. 14.13

Basic architecture of TMSC320C54x family.

14.4.5 FLOATING-POINT DSPs

Floating-point DSPs perform DSP operations using the floating-point arithmetic, as we discussed earlier. The advantages of using the floating-point processor include getting rid of finite word length effects such as overflows, round-off errors, truncation errors, and coefficient quantization error. Hence, in terms of coding, we do not need to scale input samples to avoid overflow, shift the accumulator result to fit the DAC word size, scale the filter coefficients, or apply Q-format arithmetic. The floating-point DSP with high-performance speed and calculation precision facilitates a friendly environment to develop and implement DSP algorithms.

Analog Devices provides the floating-point DSP families such as ADSP210xx and TigerSHARC. TI offers a wide range of the floating-point DSP families, in which the TMS320C3x is the first generation, followed by the TMSC320C4x and TMS320C67x families. Since the first generation of a floating-point DSP is less complicated than later generations but still has the common basic features, we overview the first-generation architecture first.

Fig. 14.14 shows the typical architecture of TI' TMS320C3x families. We discuss some key features briefly. Further detail can be found in the TMS320C3x User's Guide (Texas Instruments, 1991), the TMS320C6x CPU and Instruction Set Reference Guide (Texas Instruments, 1998), and other studies (Dahnoun, 2000; Embree, 1995; Ifeachor and Jervis, 2002; Kehtaranavaz and Simsek, 2000; Sorensen and Chen, 1997; Van der Vegte, 2002).

TMS320C3x family consists of 32-bit single-chip floating-point processors that support both integer and floating-point operations.

The processor has a large memory space and is equipped with dual-access on-chip memories. A program cache is employed to enhance the execution of commonly used codes. Similar to the fixed-point processor, it uses the Harvard architecture, where there are separate buses used for program and data so that instructions can be fetched at the same time that data are being accessed. There also exist memory buses and data buses for direct-memory access (DMA) for concurrent I/O and CPU operations, and peripheral access such as serial ports, I/O ports, memory expansion, and an external clock.

The C3x CPU contains the floating-point/integer multiplier; an ALU, which is capable of operating both integer and the floating-point arithmetic; a 32-bit barrel shifter; internal buses; a CPU register file; and dedicated auxiliary register arithmetic units (ARAUs). The multiplier operates single-cycle

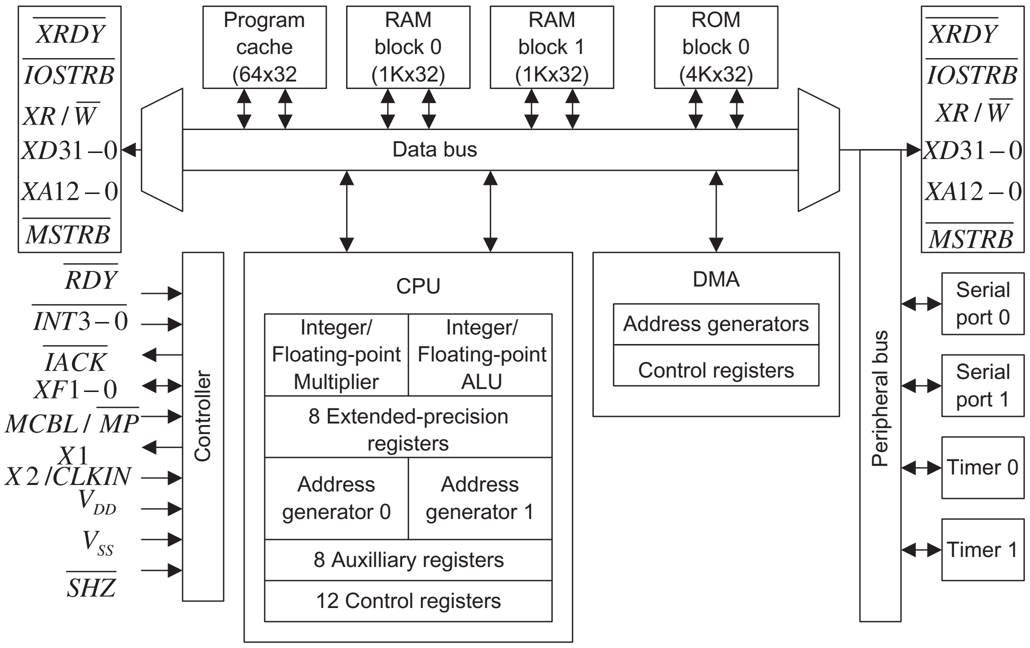


FIG. 14.14

The typical TMS320C3x floating-point DSP.

multiplications on 24-bit integers and on 32-bit floating-point values. Using parallel instructions to perform a multiplication, an ALU will cost a single cycle, which means that a multiplication and an addition are equally fast. The ARAUs support addressing modes, in which some of them are specific to DSP such as circular buffering and bit-reversal addressing (digital filtering and FFT operations). The CPU register file offers 28 registers, which can be operated on by the multiplier and ALU. The special functions of the registers include eight-extended 40-bit precision registers for maintaining the accuracy of the floating-point results. Eight auxiliary registers can be used for addressing and for integer arithmetic. These registers provide internal temporary storage of internal variables instead of external memory storage, to allow performance of arithmetic between registers. In this way, program efficiency is greatly increased.

The prominent feature of C3x is its floating-point capability, allowing operation of numbers with a very larger dynamic range. It offers implementing of the DSP algorithm without worrying about problems such as overflows and coefficient quantization. Three floating-point formats are supported. A short 16-bit floating-point format has 4 exponent bits, 1 sign bit, and 11 mantissa bits. A 32-bit single-precision format has 8 exponent bits, 1 sign bit, and 23 fraction bits. A 40-bit extended precision format contains 8 exponent bits, 1 sign bit, and 31 fraction bits. Although the formats are slightly different from the IEEE 754 standard, conversions are available between these formats.

The TMS320C30 offers high-speed performance with 60-ns single-cycle instruction execution time which is equivalent to 16.7 MIPS. For speech quality applications with an 8 kHz sampling rate, it can handle over 2000 single-cycle instructions between two samples (125 μs). With instruction enhancement such as pipeline executing each instruction in a single cycle (four cycles required from fetch to execution by the instruction itself) and a multiple interrupt structure, this high-speed processor validates implementation of real-time applications in the floating-point arithmetic.

14.5 FINITE IMPULSE RESPONSE AND INFINITE IMPULSE RESPONSE FILTER IMPLEMENTATIONS IN FIXED-POINT SYSTEMS

With knowledge of the IEEE format and of filter realization structures such as the direct-form I, direct-form II, and parallel and cascade forms (Chapter 6), we can study digital filter implementation in the fixed-point processor. In the fixed-point system, where only integer arithmetic is used, we prefer input data, filter coefficients, and processed output data to be in the Q-format. In this way, we avoid overflow due to multiplications and can prevent overflow due to addition by scaling input data. When the filter coefficients are out of the Q-format range, coefficient scaling must be taken into account to maintain the Q-format. We develop FIR filter implementation in Q-format first, and then infinite impulse response (IIR) filter implementation next. In addition, we assume that with a given input range in Q-format, the filter output is always in Q-format even if the filter passband gain is larger than 1.

First, to avoid the overflow for an adder, we can scale the input down by a scale factor S, which can be safely determined by the following equation

$$S = I_{\max} \cdot \sum_{k=0}^{\infty} |h(k)| = I_{\max} \cdot (|h(0)| + |h(1)| + |h(2)| + \cdots), \tag{14.1}$$

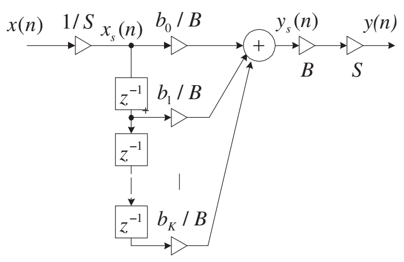


FIG. 14.15

Direct-form I implementation of the FIR filter.

where $h(k)$ is the impulse response of the adder output and $I_{\max}$ the maximum amplitude of the input in Q-format. Note that this is not an optimal factor in terms of reduced signal-to-noise ratio. However, it shall prevent the overflow. Eq. (14.1) means that the adder output can actually be expressed as a convolution output:

$$\text{adder output} = h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + \cdots. \tag{14.2}$$

Assuming the worst condition, that is, that all the inputs $x(n)$ reach a maximum value of $I_{\max}$ and all the impulse coefficients are positive, the sum of the adder gives the most conservative scale factor, as shown in Eq. (14.1). Hence, scaling down of the input by a factor of S will guarantee that the output of the adder is in Q-format.

When some of the FIR coefficients are larger than 1, which is beyond the range of Q-format representation, coefficient scaling is required. The idea is that scaling down the coefficients will make them <1, and later the filtered output will be scaled up by the same amount before it is sent to DAC. Fig. 14.15 describes the modified implementation.

In the figure, the scale factor B makes the coefficients b_k/B convertible to the Q format. The scale factors of S and B are usually chosen to be a power of 2, so the simple shift operation can be used in the coding process. Let us implement an FIR filter containing filter coefficients larger than 1 in the fixed-point implementation.

EXAMPLE 14.13

Given the FIR filter

$$y(n) = 0.9x(n) + 3x(n-1) + 0.9x(n-2),$$

with a passband gain of 4, and assuming that the input range only occupies 1/4 of the full range for a particular application, develop the DSP implementation equations in the Q-15 fixed-point system.

Solution:

The adder may cause overflow if the input data exist for $\frac{1}{4}$ of a full dynamic range. The scale factor is determined using the impulse response, which consists of the FIR filter coefficients, as discussed in Chapter 3.

$$S = \frac{1}{4}(|h(0)| + |h(1)| + |h(2)|) = \frac{1}{4}(0.9 + 3 + 0.9) = 1.2.$$

Overflow may occur. Hence, we select $S=2$ (a power of 2). We choose $B=4$ to scale all the coefficients to be <1, so the Q-15 format can be used. According to Fig. 14.15, the developed difference equations are given by

$$x_s(n)=\frac{x(n)}{2}$$

$$y_s(n)=0.225x_s(n)+0.75x_s(n-1)+0.225x_s(n-2)$$

$$y(n)=8y_s(n)$$

Next, the direct-form I implementation of the IIR filter is illustrated in Fig. 14.6.

As shown in Fig. 14.16, the purpose of a scale factor C is to scale down the original filter coefficients to the Q-format. The factor C is usually chosen to be a power of 2 for using a simple shift operation in DSP.

EXAMPLE 14.14

The following IIR filter,

$$y(n)=2x(n)+0.5y(n-1),$$

uses the direct-form I, and for a particular application, the maximum input is $I_{\max}=0.010....0_2=0.25$.

Develop the DSP implementation equations in the Q-15 fixed-point system.

Solution:

This is an IIR filter whose transfer function is

$$H(z)=\frac{2}{1-0.5z^{-1}}=\frac{2z}{z-0.5}.$$

Applying the inverse z-transform, we have the impulse response

$$h(n)=2\times(0.5)^n u(n).$$

To prevent overflow in the adder, we can compute the S factor with the help of the Maclaurin series or approximate Eq. (14.1) numerically. We get

$$S=0.25\times\left(2(0.5)^0+2(0.5)^1+2(0.5)^2+\cdots\right)=\frac{0.25\times2\times1}{1-0.5}=1.$$

MATLAB function **impz()** can also be applied to find the impulse response and the S factor:

```
>> h=impz(2,[1 -0.5]);  % Find the impulse response
>> sf=0.25*sum(abs(h))  %  Determine the sum of absolute values of h(k)
sf =1
```

Hence, we do not need to perform the input scaling. However, we need scale down all the coefficients to use the Q-15 format. A factor of $C=4$ is selected. From Fig. 14.16, we get the differences equations as

Continued

EXAMPLE 14.14—CONT'D

$$x_s(n) = x(n)$$

$$y_s(n) = 0.5_s x(n) + 0.125 y_f(n-1)$$

$$y_f(n) = 4 y_s(n)$$

$$y(n) = y_f(n).$$

We can develop these equations directly. First, we divide the original difference equation by a factor of 4 to scale down all the coefficients to be <1, that is,

$$\frac{1}{4} y_f(n) = \frac{1}{4} \times 2 x_s(n) + \frac{1}{4} \times 0.5 y_f(n-1)$$

and define a scaled output

$$y_s(n) = \frac{1}{4} y_f(n).$$

Finally, substituting $y_s(n)$ to the left side of the scaled equation and rescaling up the filter output as $y_f(n) = 4 y_s(n)$, we have the same results as we got before.

The fixed-point implementation for the direct-form II is more complicated. The developed direct-form II implementation of the IIR filter is illustrated in Fig. 14.17.

As shown in Fig. 14.17, two scale factors A and B are designated to scale denominator coefficients and numerator coefficients to their Q-format representations, respectively. Here S is a special factor to scale down the input sample so that the numerical overflow in the first sum in Fig. 14.17 can be prevented. The difference equations are given in Chapter 6 and listed here:

$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2) - \cdots - a_M w(n-M)$$

$$y(n) = b_0 w(n) + b_1 w(n-1) + \cdots + b_M w(n-M).$$

The first equation is scaled down by the factor A to ensure that all the denominator coefficients are <1, that is,

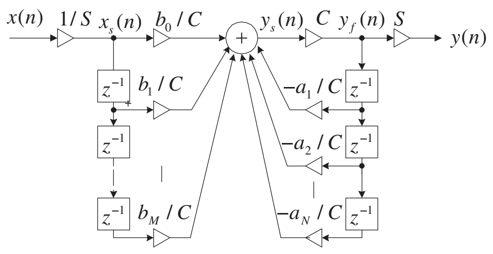


FIG. 14.16

Direct-form I implementation of the IIR filter.

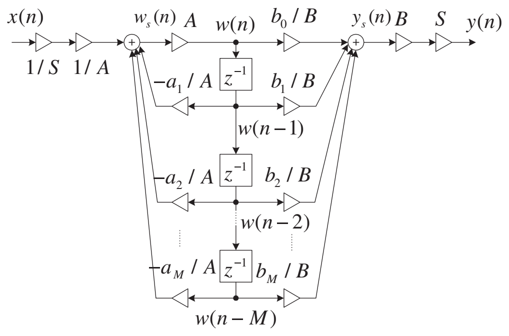


FIG. 14.17

Direct-form II implementation of the IIR filter.

$$w_s(n)=\frac{1}{A}w(n)=\frac{1}{A}x(n)-\frac{1}{A}a_1w(n-1)-\frac{1}{A}a_2w(n-2)-\cdots-\frac{1}{A}a_Mw(n-M)$$

$$w(n)=A\times w_s(n).$$

Similarly, scaling the second equation yields

$$y_s(n)=\frac{1}{B}y(n)=\frac{1}{B}b_0w(n)+\frac{1}{B}b_1w(n-1)+\cdots+\frac{1}{B}b_Mw(n-M)$$

and

$$y(n)=B\times y_s(n).$$

To avoid the first adder overflow (first equation), the scale factor S can safely be determined by Eq. (14.3):

$$S=I_{\max}\left(|h(0)|+|h(1)|+|h(2)|+\cdots\right), \tag{14.3}$$

where $h(k)$ is the impulse response of the filter whose transfer function is the reciprocal of the denominator polynomial, where the poles can cause a larger value to the first sum. Hence, $h(n)$ is given by

$$h(n)=Z^{-1}\left(\frac{1}{1+a_1z^{-1}+\cdots+az^{-M}}\right) \tag{14.4}$$

All the scale factors A, B, and S are usually chosen to be a power of 2, respectively, so that the shift operations can be used in the coding process. Example 14.15 serves as an illustration.

EXAMPLE 14.15

Given the IIR filter:

$$y(n)=0.75x(n)+1.49x(n-1)+0.75x(n-2)-1.52y(n-1)-0.64y(n-2),$$

with a passband gain of 1 and a full range of input, use the direct-form II implementation to develop the DSP implementation equations in the Q-15 fixed-point system.

Continued

EXAMPLE 14.15—CONT'D

Solution:

The difference equations without scaling in the direct-form II implementation are given by

$$w(n) = x(n) - 1.52w(n-1) - 0.64w(n-2)$$

$$y(n) = 0.75w(n) + 1.49w(n-1) + 0.75w(n-2).$$

To prevent the overflow in the first adder, we have the reciprocal of the denominator polynomial as

$$A(z) = \frac{1}{1 + 1.52z^{-1} + 0.64z^{-2}}.$$

Using the MATLAB function leads to.

```
>> h=impz(1,[1 1.52 0.64]);
>> sf=sum(abs(h))
   sf = 10.4093
```

We choose the S factor as $S = 16$ and we choose $A = 2$ to scale down the denominator coefficients by half. Since the second adder output after scaling is

$$y_s(n) = \frac{0.75}{B}w(n) + \frac{1.49}{B}w(n-1) + \frac{0.75}{B}w(n-2),$$

we have to ensure that each coefficient is <1, as well as the sum of the absolute values

$$\frac{0.75}{B} + \frac{1.49}{B} + \frac{0.75}{B} < 1$$

to avoid the second adder overflow. Hence $B = 4$ is selected. We develop the DSP equations as

$$x_s(n) = x(n)/16$$

$$w_s(n) = 0.5x_s(n) - 0.76w(n-1) - 0.32w(n-2)$$

$$w(n) = 2w_s(n)$$

$$y_s(n) = 0.1875w(n) + 0.3725w(n-1) + 0.1875w(n-2)$$

$$y(n) = (B \times S)y_s(n) = 64y_s(n).$$

The implementation for cascading the second-order section filters can be found in Ifeachor and Jervis (2002).

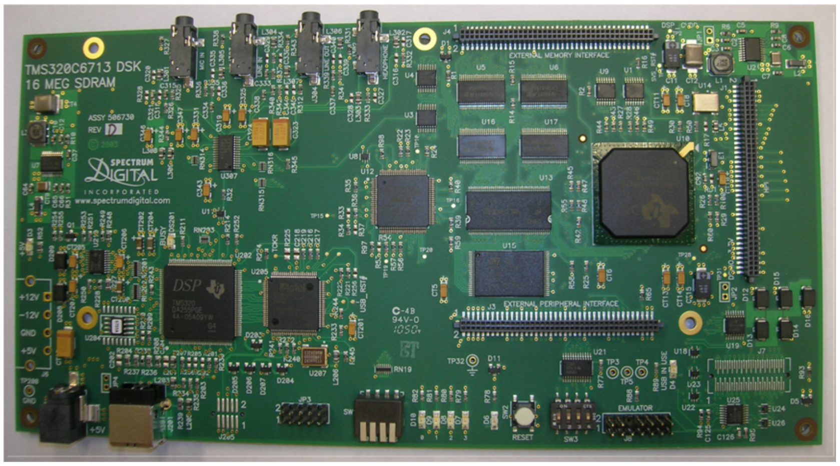
A practical example is presented in the next section. Note that if a floating-point DSP is used, all the scaling concerns should be ignored, since the floating-point format offers a large dynamic range, so that overflow hardly even happens.

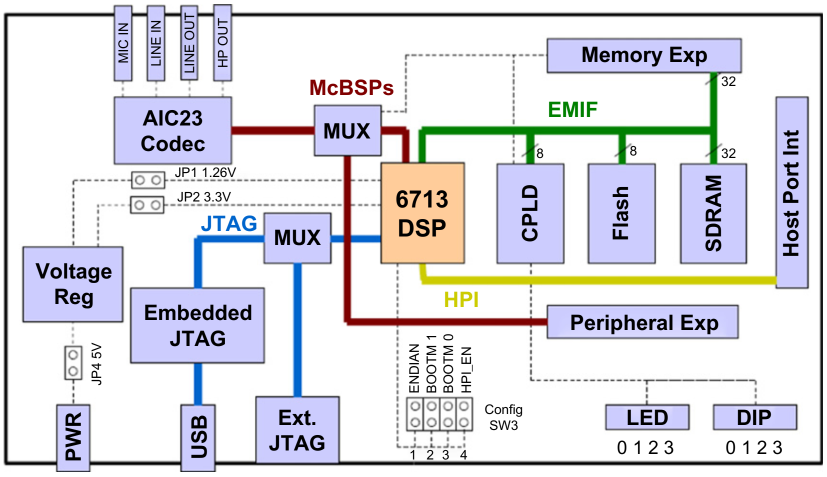
14.6 DIGITAL SIGNAL PROCESSING PROGRAMMING EXAMPLES

In this section, we first review the TMS320C67x DSK (DSP Starter Kit), which offers floating-point and fixed-point arithmetic. We will then investigate the real-time implementation of digital filters.

14.6.1 OVERVIEW OF TMS320C67x DSK

In this section, a TI TMS320C6713 DSK (DSP Starter Kit) shown in Fig. 14.18 is chosen for demonstration. This DSK board has an approximate size of 5 × 8 inches, a clock rate of 225 MHz and a 16-bit stereo codec TL V320AIC23 (AIC23) which deals with analog inputs and outputs. The onboard codec AIC23 applies sigma-delta technology for ADC and DAC functions. The codec runs using a 12 MHz system clock and the sampling rate can be selected from a range of 8–96 kHz for speech and audio processing. Other boards such as a TI TMS320C6711 DSK can also be found in the references (Kehtaranavaz and Simsek, 2000; TMS320C6x CPU and Instruction Set Reference Guide; 1999).


(A) TMS320C6713 DSK board



(B) TMS320C6713 DSK block diagram

FIG. 14.18

C6713 DSK board and block diagram. (A) TMS320C6713 DSK board. (B) TMS320C6713 DSK block diagram.

Courtesy of Texas Instruments.

The on-board daughter card connections facilitate the external units for advanced applications such as external peripheral and external memory interfaces (EMIFs). The TMS320C6713 DSK board consists of 16 MB (megabytes) of synchronous dynamic RAM (SDRAM) and 512 kB (kilobytes) of flash memory. There are four onboard connections: MIC IN for microphone input; LINE IN for line input; LINE OUT for line output; and HEADPHONE for a headphone output (multiplexed with LINE OUT). The four user DIP switches can be read from a program running within DSK board as well as they can provide with a user feedback control of interface. The four LEDs (light-emitting diodes) on the DSK board can be controlled by a running DSP program. The onboard voltage regulators provide 1.26 V for the DSP core while 3.3 V for the memory and peripherals. The USB port provides the connection between the DSK board and the host computer, where the user program is developed, compiled, and downloaded to the DSK for real-time applications using the user-friendly software called the Code Composer Studio (CCS), which we discuss this later.

In general, the TMS320C67x operates at a high clock rate of 300 MHz. Combining with high speed and multiple units operating at the same time has pushed its performance up to 2400 MIPS at 300 MHz. Using this number, the C67x can handle 0.3 MIPS between two speech samples at a sampling rate of 8 kHz and can handle over 54,000 instructions between the two audio samples with a sampling rate of 44.1 kHz. Hence, the C67x offers great flexibility for real-time applications with a high-level C language.

Fig. 14.19 shows a C67x architecture overview, while Fig. 14.20 displays a more detailed block diagram. C67x contains three main parts, which are the CPU, the memories, and the peripherals. As shown in Fig. 14.9, these three main parts are joined by an EMIF interconnected by internal buses to facilitate interface with common memory devices; DMA; a serial port; and a host-port interface (HPI).

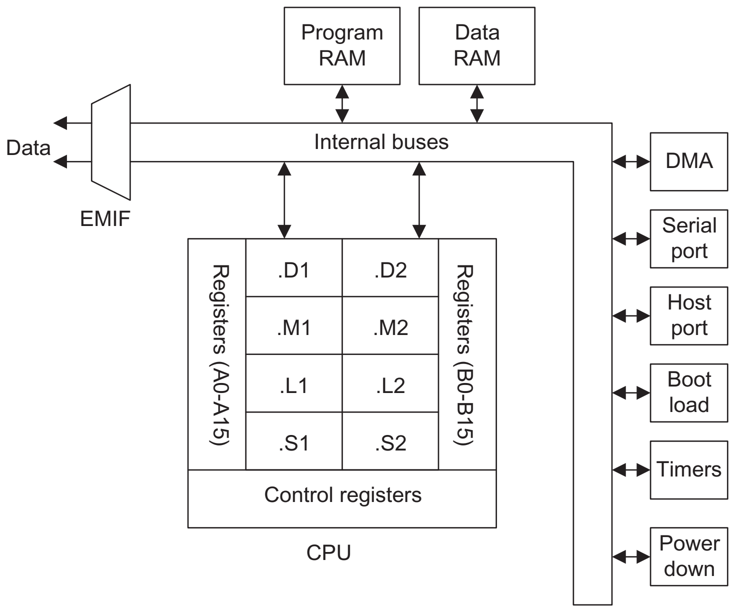


FIG. 14.19

Block diagram of TMS320C67x floating-point DSP.

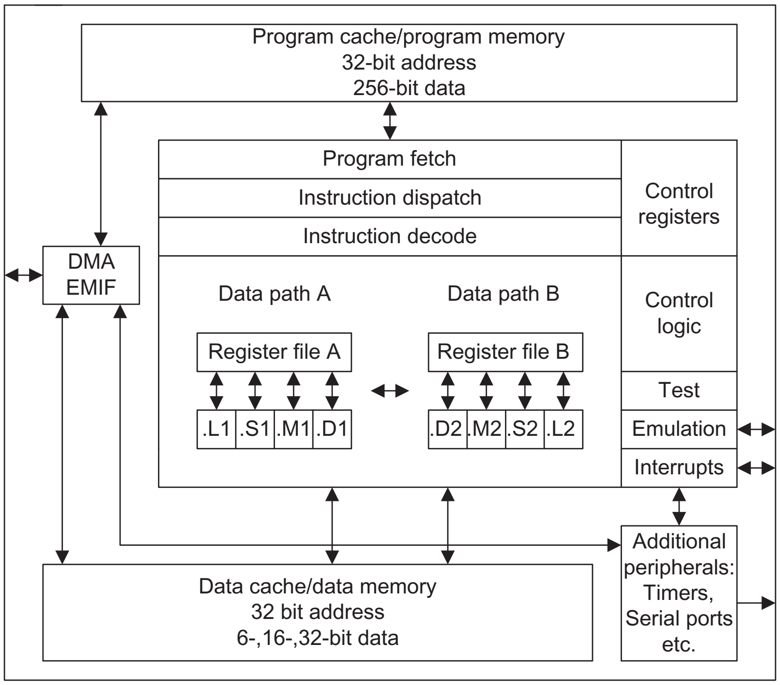


FIG. 14.20

Registers of TMS320C67x floating-point DSP.

Since this section is devoted to show DSP coding examples, C67x key features and references are briefly listed here:

(1) Architecture: The system uses Texas Instruments Veloci^{TM} architecture, which is an enhancement of the VLIW (very long instruction word architecture) (Dahnoun, 2000; Ifeachor and Jervis, 2002; Kehtaranavaz and Simsek, 2000).

(2) CPU: As shown in Fig. 14.20, the CPU has eight functional units divided into two sides A and B, each consisting of units .D, .M, .L, and .S. For each side, an .M unit is used for multiplication operation, an. L unit is used for logical and arithmetic operations, and a .D unit is used for loading/storing and arithmetic operations. Each side of the C67x CPU has sixteen 32-bit registers that the CPU must go through for interface. More detail can be found in Appendix D (Texas Instruments, 1991) as well as in Kehtaranavaz and Simsek (2000) and Texas Instruments (1998).

(3) Memory and internal buses: Memory space is divided into internal program memory, internal data memory, and internal peripheral and external memory space. The internal buses include a 32-bit program address bus, a 256-bit program data bus to carrying out eight 32-bit instructions (VLIW), two 32-bit data address buses, two 64-bit load data buses, two 64-bit store data buses, two 32-bit DMA buses, and two 32-bit DMA address buses responsible for reading and writing. There also exit a 22-bit address bus and a 32-bit data bus for accessing off-chip or external memory.

(4) Peripherals:

- **(a)** EMIF, which provides the required timing for accessing external memory
- **(b)** DMA, which moves data from one memory location to another without interfering with the CPU operations
- **(c)** Multichannel buffered serial port (McBSP) with a high-speed multi-channel serial communication link
- **(d)** HPI, which lets a host access internal memory
- **(e)** Boot loader for loading code from off-chip memory or the HPI to internal memory
- **(f)** Timers (two 32-bit counters)
- **(g)** Power-down units for saving power for periods when the CPU is inactive.

The software tool for the C67x is the CCS provided by TI. It allows the user to build and debug programs from a user-friendly graphical user interface (GUI) and extends the capabilities of code development tools to include real-time analysis. Installation, tutorial, coding, and debugging information can be found in the CCS Getting Started Guide (Texas Instruments, 2001) and in Kehtaranavaz and Simsek (2000).

Particularly for the TMS320C6713 DSK with a clock rate of 225 MHz, it has capability to fetch eight 32-bit instructions every 4.4 ns (1/225 MHz). The functional block diagram is shown in Fig. 14.21. The detailed description can found in Chassaing and Reay (2008).

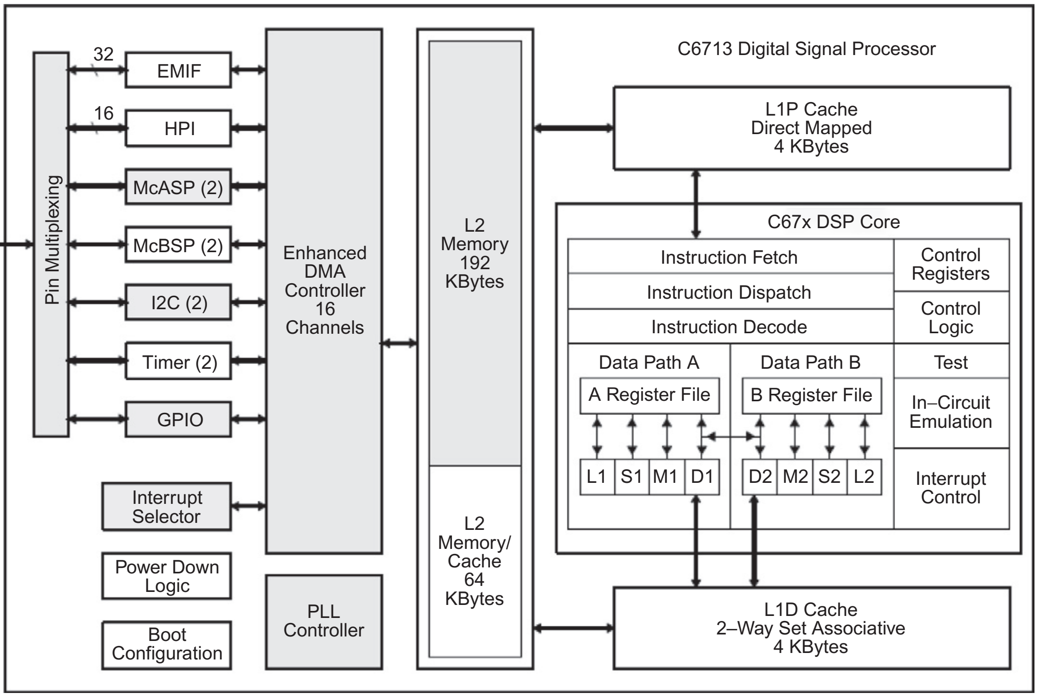


FIG. 14.21

Functional block diagram and registers of TMS320C6713.

Courtesy of Texas Instruments.

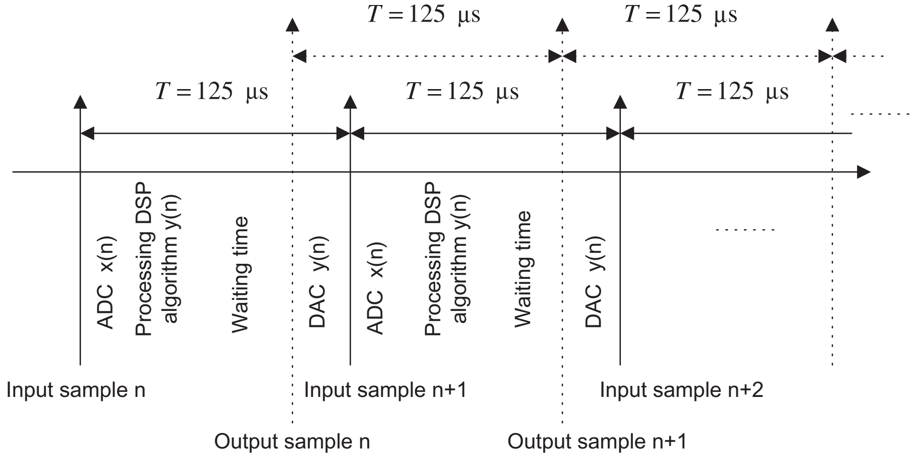


FIG. 14.22

Concept of real-time processing.

14.6.2 CONCEPT OF REAL-TIME PROCESSING

We illustrate the real-time implementation shown in Fig. 14.22, where the sampling rate is 8000 samples per second; that is, the sampling period $T = 1/f_s = 125$ μs, which is the time between two samples.

As shown in Fig. 14.22, the required timing includes an input sample clock and an output sample clock. The input sample clock maintains the accuracy of sampling time for each ADC operation, while the output sampling clock keeps the accuracy of time instant for each DAC operation. The time between the input sample clock n and output sample clock n consists of the ADC operation, algorithm processing, and the wait for the next ADC operation. The numbers of instructions for ADC and DSP algorithm must be estimated and verified to ensure that all instructions have been completed before the DAC begins. Similarly, the number of instructions for DAC must be verified so that DAC instructions will be finished between the output sample clock n and the next input sample clock $n+1$. Timing usually is set up using the DSP interrupts (we will not pursue the interrupt setup here).

Next, we focus on the implementation of the DSP algorithm in the floating-point system for simplicity. A DSK setup example (Tan and Jiang, 2010) is depicted in Fig. 14.23, while a skeleton code for the verification of the input and output is depicted in Fig. 14.24.

14.6.3 LINEAR BUFFERING

During DSP such as digital filtering, past inputs, and past outputs are required to be buffered and updated for processing the next input sample. Let us first study the FIR filter implementation.

FIR Filtering:

Consider implementation for the following 3-tap FIR filter:

$$y(n) = 0.5x(n) + 0.2x(n-1) + 0.5x(n-2).$$

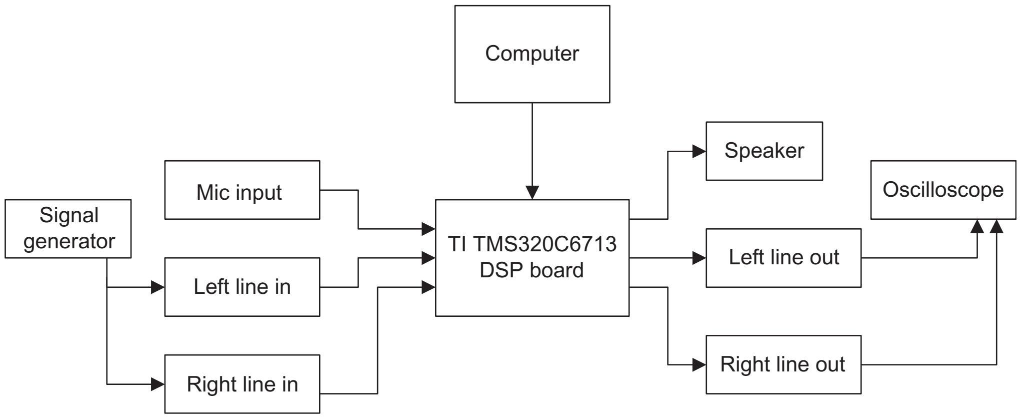


FIG. 14.23

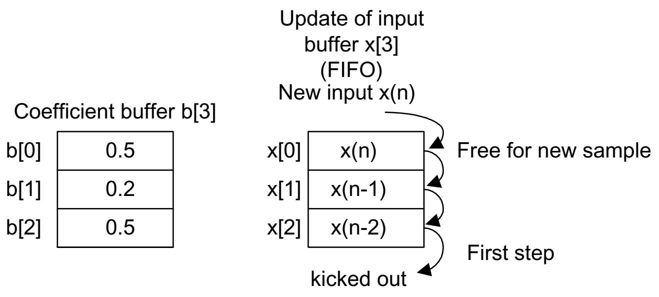
TMS320C6713 DSK setup example.

```
float x[1]={0.0};
float y[1]={0.0};
interrupt void c_int11()
{
     float lc; /*left channel input */
     float rc; /*right channel input */
     float lcnew; /*left channel output */
     float rcnew; /*right channel output */
//Left channel and right channel inputs
    AIC23_data.combo=input_sample();
    lc=(float) (AIC23_data.channel[LEFT]);
    rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
    x[0]=lc; /* Input from the left channel */
    y[0]=x[0];  /* simplest DSP equation */
// End of the DSP algorithm
     lcnew=y[0];
     rcnew=y[0];
     AIC23_data.channel[LEFT]=(short) lcnew;
     AIC23_data.channel[RIGHT]=(short) rcnew;
     output_sample(AIC23_data.combo);
}
```

FIG. 14.24

Program segment for verifying input and output.

The buffer requirements are shown in Fig. 14.25. The coefficient buffer b[3] contains three FIR coefficients, and the coefficient buffer is fixed during the process. The input buffer x[3], which holds the current and past inputs, is required to be updated. The FIFO update is adopted here with the segment of codes shown in Fig. 14.25. For each input sample, we update the input buffer using FIFO, which begins at the end of the data buffer; the oldest sampled is kicked out first from the buffer and updated with the value from the upper location. When the FIFO completes, the first memory



```
float x[3]={0.0, 0.0, 0.0};
float b[3]={0.5, 0.2, 0.5};
float y[1]={0.0};
interrupt void c_int11()
{
     float lc; /*left channel input */
     float rc; /*right channel input */
     float lcnew; /*left channel output */
     float rcnew; /*right channel output */
     int i;
//Left channel and right channel inputs
    AIC23_data.combo=input_sample();
    lc=(float) (AIC23_data.channel[LEFT]);
    rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
   for(i=2; i>0; i--)    /* Update the input buffer x[3] */
   {   x[i]=x[i-1]; }
    x[0]= (float) lc;  /*Input from the left channel */
    y[0]=0;
     for(i=0; i<3; i++)
   {  y[0]=y[0]+b[i]*x[i]; } /* FIR filtering */
// End of the DSP algorithm
     lcnew=y[0];
     rcnew=y[0];
     AIC23_data.channel[LEFT]=(short) lcnew;
     AIC23_data.channel[RIGHT]=(short) rcnew;
     output_sample(AIC23_data.combo);
}
```

FIG. 14.25

Example of FIR filtering with linear buffer update.

location x[0] will be free to be used to store the current input sample. The segment of code in Fig. 14.25 explains implementation.

Note that in the code segment, $x[0]$ holds the current input sample $x(n)$, while $b[0]$ is the corresponding coefficient; $x[1]$ and $x[2]$ hold the past input samples $x(n-1)$ and $x(n-2)$, respectively; similarly, $b[1]$ and $b[2]$ are the corresponding coefficients.

Again, note that using the array and loop structures in the code segment is for simplicity in notations and assume that the reader is not familiar with the C pointers in the C language. The concern for simplicity has to do mainly with the DSP algorithm. More coding efficiency can be achieved using the C pointers and a circular buffer. The DSP-oriented coding implementation can be found in Kehtaranavaz and Simsek (2000) and Chassaing and Reay (2008).

IIR Filtering:

Similarly, we can implement an IIR filter. It requires an input buffer, which holds the current and past inputs; an output buffer, which holds the past outputs; a numerator coefficient buffer; and a denominator coefficient buffer. Considering the following IIR filter for implementation,

$$y(n) = 0.5x(n) + 0.7x(n-1) - 0.5x(n-2) - 0.4y(n-1) + 0.6y(n-2),$$

we accommodate the numerator coefficient buffer b[3], the denominator coefficient buffer a[3], the input buffer x[3], and the output buffer y[3] shown in Fig. 14.26. The buffer updates for input x[3] and output y[3] are FIFO. The implementation is illustrated in the segment of code listed in Fig. 14.26.

Again, note that in the code segment, $x[0]$ holds the current input sample, while $y[0]$ holds the current processed output, which will be sent to the DAC unit for conversion. The coefficient $a[0]$ is never modified in the code. We keep that for a purpose of notation simplicity and consistency during the programming process.

Digital Oscillation with IIR Filtering:

The principle for generating digital oscillation is described in Chapter 8, where the input to the digital filter is the impulse sequence, and the transfer function is obtained by applying the z-transform of the digital sinusoid function. Applications can be found in dual-tone multifrequency (DTMF) tone generation, digital carrier generation for communications, and so on. Hence, we can modify the implementation of IIR filtering for tone generation with the input generated internally instead of using the ADC channel.

Let us generate an 800 Hz tone with the digital amplitude of 5000. According to the section in Chapter 8 ("Applications: Generation and Detection of DTMF Tones Using the Goertzel Algorithm"), the transfer function, difference equation, and the impulse input sequence are found to be, respectively,

$$H(z) = \frac{0.587785z^{-1}}{1 - 1.618034z^{-1} + z^{-2}}$$

$$y(n) = 0.587785x(n-1) + 1.618034y(n-1) - y(n-2)$$

$$x(n) = 5000\delta(n).$$

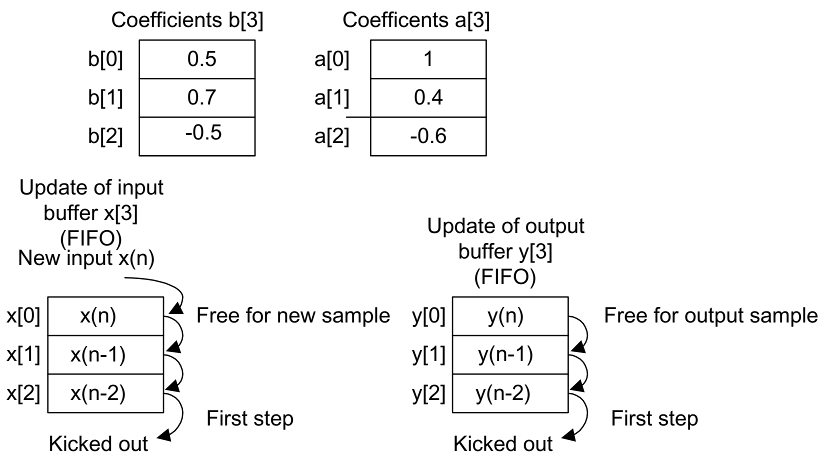
We define the numerator coefficient buffer b[2], the denominator coefficient buffer a[3], the input buffer x[2], and the output buffer y[3], shown in Fig. 14.27, which also shows the modified implementation for tone generation.

Initially, we set $x[0] = 5000$. Then it will be updated with $x[0] = 0$ for each current processed output sample $y[0]$.

14.6.4 SAMPLE C PROGRAMS

Floating-Point Implementation Example:

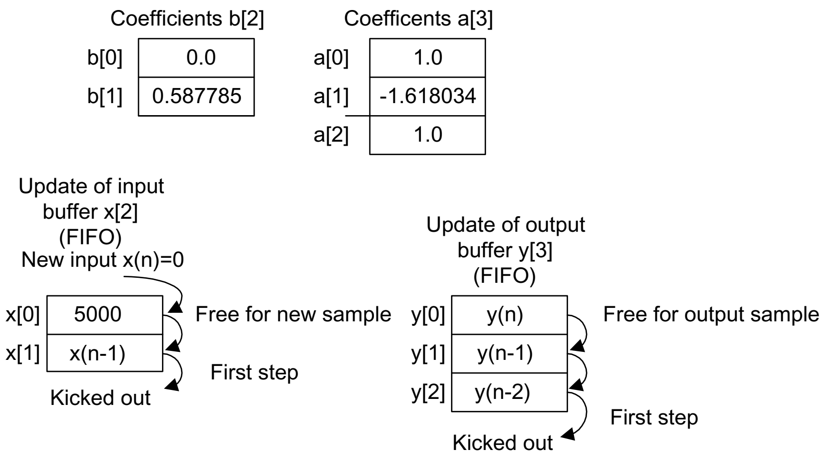
Real-time DSP implementation using a float-point processor is easy to program. The overflow problem hardly ever occurs. Therefore, we do not need to consider scaling factors, as described in the last section. The code segment shown in Fig. 14.28 demonstrates the simplicity of coding the floating-point IIR filter using the direct-form I structure.



```
float x[3]={0.0, 0.0, 0.0};
float b[3]={0.5, 0.7, -0.5};
float a[3]={1, 0.4, -0.6};
float y[3]={0.0, 0.0, 0.0};
interrupt void c_int11()
{
     float lc; /*left channel input */
     float rc; /*right channel input */
     float lcnew; /*left channel output */
     float rcnew; /*right channel output */
     int i;
//Left channel and right channel inputs
    AIC23_data.combo=input_sample();
    lc=(float) (AIC23_data.channel[LEFT]);
    rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
     for(i=2; i>0; i--)         /* Update the input buffer */
     {   x[i]=x[i-1];   }
     x[0]= lc; /* Input from the left channel */
      for (i=2;i>0;i--)      /* Update the output buffer */
      {    y[i]=y[i-1];  }
       y[0]=b[0]*x[0]+b[1]*x[1]+b[2]*x[2]-a[1]*y[1]-a[2]*y[2];
// End of the DSP algorithm
     lcnew=y[0];
     rcnew=y[0];
     AIC23_data.channel[LEFT]=(short) lcnew;
     AIC23_data.channel[RIGHT]=(short) rcnew;
     output_sample(AIC23_data.combo);
}
```

FIG. 14.26

Example of IIR filtering using linear buffer update.



```
float x[2]={5000, 0.0}; /*initialize the impulse input */
float b[2]={0.0, 0.587785};
float a[3]={1.0, -1.618034,  1.0};
float y[3]={0.0};
interrupt void c_int11()
{    float lc; /*left channel input */
     float rc; /*right channel input */
     float lcnew; /*left channel output */
     float rcnew; /*right channel output */
     int i;
//Left channel and right channel inputs
    AIC23_data.combo=input_sample();
    lc=(float) (AIC23_data.channel[LEFT]);
    rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
     y[0]=b[0]*x[0]+b[1]*x[1]+b[2]*x[2]-a[1]*y[1]-a[2]*y[2];
     for(i=1; i>0; i--)     /* Update the input buffer with zero input */
     {   x[i]=x[i-1];   }
      x[0]= 0;
      for (i=2;i>0;i--)     /* Update the output buffer */
      {    y[i]=y[i-1];  }
// End of the DSP algorithm
     lcnew=y[0];
     rcnew=y[0];
     AIC23_data.channel[LEFT]=(short) lcnew;
     AIC23_data.channel[RIGHT]=(short) rcnew;
     output_sample(AIC23_data.combo);
}
```

FIG. 14.27

Example of IIR filtering using linear buffer update and the impulse sequence input.

```
float a[5]={1.00, -2.1192, 2.6952, -1.6924, 0.6414};
float b[5]={0.0201, 0.00, -0.0402, 0.00, 0.0201};
float x[5]={0.0, 0.0, 0.0, 0.0, 0.0};
float y[5]={0.0, 0.0, 0.0, 0.0, 0.0};

interrupt void c_int11()
{
    float lc; /*left channel input */
    float rc; /*right channel input */
    float lcnew; /*left channel output */
    float rcnew; /*right channel output */
    int i;
//Left channel and right channel inputs
   AIC23_data.combo=input_sample();
   lc=(float) (AIC23_data.channel[LEFT]);
   rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
    for(i=4; i>0; i--)        /* Update the input buffer */
    {
          x[i]=x[i-1];
    }
    x[0]= lc; /* Input from the left channel */
     for (i=4;i>0;i--)      /* Update the output buffer */
     {
           y[i]=y[i-1];
      }
      y[0]=b[0]*x[0]+b[1]*x[1]+b[2]*x[2]+b[3]*x[3]+b[4]*x[4]-a[1]*y[1]-a[2]*y[2]-a[3]*y[3]-a[4]*y[4];
// End of the DSP algorithm
    lcnew=y[0];
    rcnew=y[0];
    AIC23_data.channel[LEFT]=(short) lcnew;
    AIC23_data.channel[RIGHT]=(short) rcnew;
    output_sample(AIC23_data.combo);
}
```

FIG. 14.28

Sample C code for IIR filtering (floating-point implementation).

Fixed-Point Implementation Example:

When the execution time is critical, the fixed-point implementation is preferred in a floating-point processor. We implement the following IIR filter with a unit passband gain in direct-form II:

$$H(z)=\frac{0.0201-0.0402z^{-2}+0.0201z^{-4}}{1-2.1192z^{-1}+2.6952z^{-2}-1.6924z^{-3}+0.6414z^{-4}}$$

$$w(n)=x(n)+2.1192w(n-1)-2.6952w(n-2)+1.6924w(n-3)-0.6414w(n-4)$$

$$y(n)=0.0201w(n)-0.0402w(n-2)+0.0201w(n-4).$$

Using MATLAB to calculate the scale factor S, it follows that.

```
» h=impz([1][1 -2.1192 2.6952 -1.6924 0.6414]);
» sf=sum(abs(h))
  sf =28.2196
```

Table 14.4 Filter Coefficients in Q-15 Format

IIR Filter	Filter Coefficients	Q-15 Format (Hex)
$-a_1$	0.5298	0x43D0
$-a_2$	−0.6738	0xA9C1
$-a_3$	0.4230	0x3628
$-a_4$	−0.16035	0xEB7A
b_0	0.0201	0x0293
b_1	0.0000	0x0000
b_2	−0.0402	0xFADB
b_3	0.0000	0x000
b_4	0.0201	0x0293

Hence we choose $S=32$. To scale the filter coefficients in the Q-15 format, we use the factors $A=4$ and $B=1$. Then the developed DSP equations are

$$x_s(n)=x(n)/32$$

$$w_s(n)=0.25x_s(n)+0.5298w_s(n-1)-0.6738w_s(n-2)+0.4231w_s(n-3)-0.16035w_s(n-4)$$

$$w(n)=4w_s(n)$$

$$y_s(n)=0.0201w(n)-0.0402w(n-2)+0.0201w(n-4)$$

$$y(n)=32y_s(n).$$

Using the method described in Section 14.5, we can convert filter coefficients into the Q-15 format; each coefficient is listed in Table 14.4.

The list of codes for the fixed-point implementation is displayed in Fig. 14.29, and some coding notations are given in Fig. 14.30.

Note that this chapter has provided only basic concepts and an introduction to real-time DSP implementation. The coding detail and real-time DSP applications will be treated in a separate DSP course, which deals with real-time implementations.

14.7 ADDITIONAL REAL-TIME DSP EXAMPLES

In this section, we examine more examples of real-time DSP implementations.

14.7.1 ADAPTIVE FILTERING USING THE TMS320C6713 DSK

The implementation for system modeling is discussed in Section 9.3 and is depicted in Fig. 14.31, where the input is fed from a function generator. The unknown system is a bandpass filter with the lower cutoff frequency of 1400 Hz and upper cutoff frequency of 1600 Hz. As shown in Fig. 14.31, the left input channel (Left Line In [LCI]) is used for the input while the left output channel (Left Line Out [LCO]) and the right output channel (Right Line Out [RCO]) are designated as the system output

```
/*float a[5]={1.00, -2.1192, 2.6952, -1.6924, 0.6414}; float b[5]={0.0201, 0.00, -0.0402, 0.00, 0.0201};*/
short a[5]={0x2000, 0x43D0, 0xA9C1, 0x3628, 0xEB7A}; /* coefficients in Q-15 format */
short b[5]={0x0293, 0x0000, 0xFADB, 0x0000, 0x0293};
int w[5]={0, 0, 0, 0, 0};
int sample;

interrupt void c_int11()
{
    float lc; /*left channel input */
    float rc; /*right channel input */
    float lcnew; /*left channel output */
    float rcnew; /*right channel output */
    int i, sum=0;
//Left channel and right channel inputs
   AIC23_data.combo=input_sample();
   lc= (float) (AIC23_data.channel[LEFT]);
   rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
  sample = (int) lc; /*input sample from the left channel*/
  sample = (sample << 16);  /* move to high 16 bits */
  sample = (sample>>5); /* scaled down by 32 to avoid overflow */
  for (i=4;i>0;i--)
  {
   w[i]=w[i-1];
  }
  sum= (sample >> 2); /* scaled down by 4 to use Q-15 */
  for (i=1;i<5;i++)
  {
   sum += (_mpyhl(w[i],a[i])) <<1;
  }
  sum = (sum <<2); /* scaled up by 4 */
  w[0]=sum;
  sum =0;
  for(i=0;i<5;i++)
  {
   sum += (_mpyhl(w[i],b[i]))<<1;
  }
  sum = (sum << 5);  /* scaled up by 32 to get y(n) */
  sample= (sum>>16); /* move to low 16 bits */
// End of the DSP algorithm
    lcnew=sample;
    rcnew=sample;
    AIC23_data.channel[LEFT]=(short) lcnew;
    AIC23_data.channel[RIGHT]=(short) rcnew;
    output_sample(AIC23_data.combo);
}
```

FIG. 14.29

Sample C code for IIR filtering (fixed-point implementation).

short coefficient; declaration of 16 bit signed integer
int sample, result; declaration of 32 bit signed integer
MPYHL assembly instruction (signed multiply high low 16 MSB x 16 LSB)

result = (_mpyhl(sample,coefficient)) <<1;

sample must be shifted left by 16 bits to be stored in the high 16 MSB.
coefficient is the 16 bit data to be stored in the low 16 LSB.
result is shifted left by one bit to get rid of the extended sign bit, and high 16 MSB's are designated for the processed data.
Final **result** will be shifted down to right by 16 bits before DAC conversion.

sample = (result>>16);

FIG. 14.30

Some coding notations for the Q-15 fixed-point implementation.

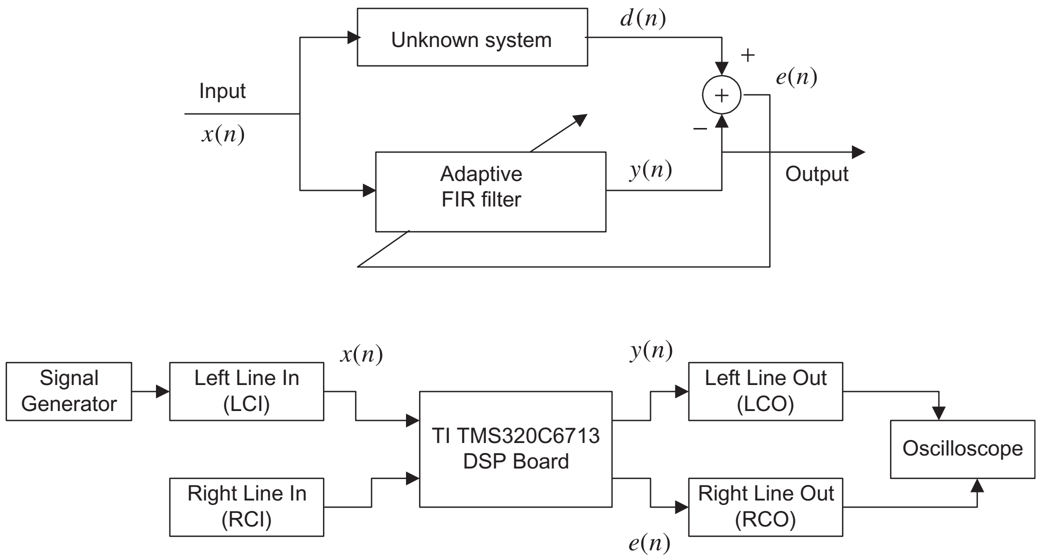


FIG. 14.31

Setup for system modeling using the LMS adaptive filter.

and error output, respectively. Note that the right input channel (Right Line In [RCI]) is not used. When the input frequency is swept from 200 to 3000 Hz, the output shows the maximum peak when the input frequency is dialed to around 1500 Hz. Hence, the adaptive filter acts like the unknown system. Fig. 14.32 lists the sample program segment.

With the advantage of the stereo input and output channels, we can conduct system modeling for an unknown analog system illustrated in Fig. 14.33, where RCI is used to feed the unknown analog system output to the DSK. The program segment is listed in Fig. 14.34.

Fig. 14.35A shows an example of an adaptive noise reduction system. Fig. 14.35B shows the details of the adaptive noise cancellation. The first DSP board is used to create the real-time corrupted signal which is obtained by mixing the mono audio source (Left Line In [LCI1]) from any audio device and the tonal noise (Right Line In [RCI1]) generated from a function generator. The output (Left line Out

```
/*Numerator coefficients */
/*for the bandpass filter (unknown system) with fL=1.4 kHz, fH=1.6 kHz*/
float b[5]={ 0.005542761540433,  0.000000000000002,  -0.011085523080870,
                0.000000000000003   0.005542761540431};
/*Denominator coefficients */
/*for the bandpass filter (unknown system) with fL=1.4 kHz, fH=1.6 kHz*/
float a[5]={ 1.000000000000000,  -1.450496619180500.  2.306093105231476,
               -1.297577189144526   0.800817049322883};
float x[40]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; /*Reference input buffer*/
float w[40]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; /*Adaptive filter coefficients*/
float d[5]={0.0, 0.0, 0.0, 0.0, 0.0}; /*Unknown system output */
float y[1]={0.0}; /* Adaptive filter output */
float e[1]={0.0}; /* Error signal */
float mu=0.000000000002; /*Adaptive filter convergence factor*/
interrupt void c_int11()
{
     float lc; /*left channel input */
     float rc; /*right channel input */
     float lcnew; /*left channel output */
     float rcnew; /*right channel output */
     int i;
//Left channel and right channel inputs
    AIC23_data.combo=input_sample();
    lc=(float) (AIC23_data.channel[LEFT]);
    rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
   for(i=39;i>0;i--) /*Update the input buffer*/
   { x[i]=x[i-1]; }
    x[0]=lc;
   for(i=4; i>0; i--)
   { d[i]=d[i-1]; }
   d[0]=b[0]*x[0]+ b[1]*x[1]+ b[2]*x[2]+ b[3]*x[3]+ b[4]*x[4]
           -a[1]*d[1]-a[2]*d[2]- a[3]*d[3]- a[4]*d[4]; /*Unknown system output*/
// Adaptive filter
   y[0]=0;
   for(i=0;i<40; i++)
   { y[0]=y[0]+w[i]*x[i];}
    e[0]=d[0]-y[0]; /* Error output */
    for(i=0;i<40; i++)
   { w[i]=w[i]+2*mu*e[0]*x[i];} /* LMS algorithm */
// End of the DSP algorithm
     lcnew=y[0]; /* Send the tracked output */
     rcnew=e[0]; /* Send the error signal*/
     AIC23_data.channel[LEFT]=(short) lcnew;
     AIC23_data.channel[RIGHT]=(short) rcnew;
     output_sample(AIC23_data.combo);
}
```

FIG. 14.32

Program segment for system modeling.

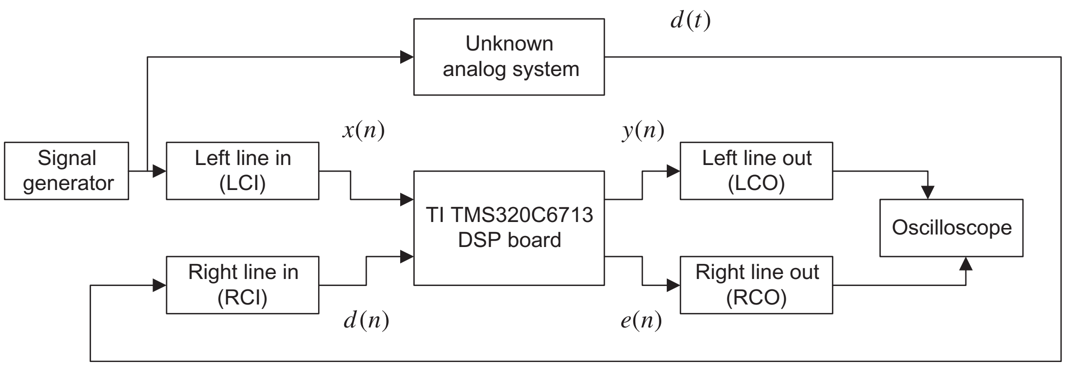


FIG. 14.33

System modeling using an LMS adaptive filter.

```
float x[40]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; /*Reference input buffer*/
float w[40]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; /*Adaptive filter coefficients*/
float d[1]={0.0}; /*Unknown system output */
float y[1]={0,0}; /* Adaptive filter output */
float e[1]={0.0}; /* Error signal */
float mu=0.000000000002; /*Adaptive filter convergence factor*/
interrupt void c_int11()
{
   float lc; /*left channel input */
   float rc; /*right channel input */
   float lcnew; /*left channel output */
   float rcnew; /*right channel output */
   int i;
//Left channel and right channel inputs
  AIC23_data.combo=input_sample();
  lc=(float) (AIC23_data.channel[LEFT]);
  rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
  for(i=39;i>0;i--) /*Update the input buffer*/
  { x[i]=x[i-1]; }
   x[0]=lc;
  d[0]=rc;  /*Unknown system output*/
// Adaptive filter
  y[0]=0;
  for(i=0;i<40; i++)
  { y[0]=y[0]+w[i]*x[i];}
   e[0]=d[0]-y[0]; /* Error output */
   for(i=0;i<40; i++)
  { w[i]=w[i]+2*mu*e[0]*x[i];} /* LMS algorithm */
// End of the DSP algorithm
   lcnew=y[0]; /* Send the tracked output */
   rcnew=e[0]; /* Send the error signal*/
   AIC23_data.channel[LEFT]=(short) lcnew;
   AIC23_data.channel[RIGHT]=(short) rcnew;
   output_sample(AIC23_data.combo);
}
```

FIG. 14.34

Program segment for modeling an analog system.

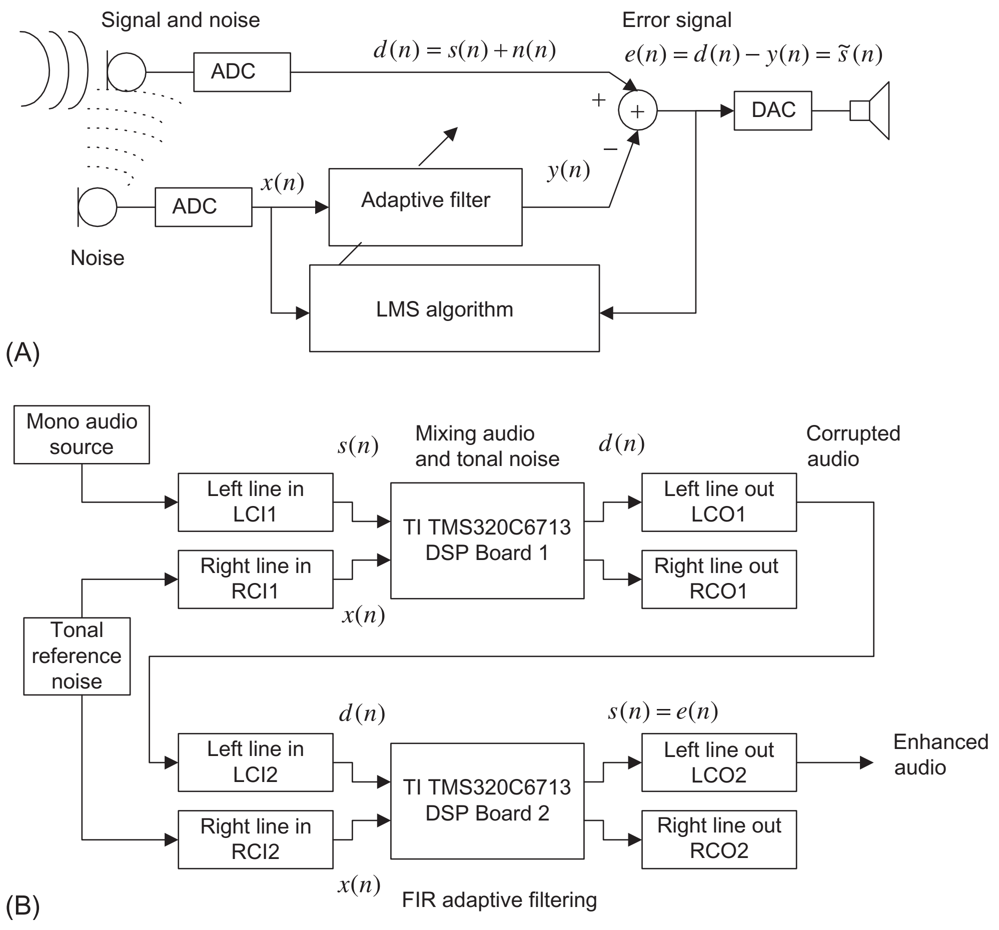


FIG. 14.35

(A) Block diagram for tonal noise cancellation. (B) Tonal noise cancellation with the adaptive filter.

[LCO1]) is the corrupted signal, which is fed to the second DSP board for noise cancellation application. The adaptive FIR filter in the second DSP board uses the reference input (Right Line In [RCI2]) to generate the output, which is used to cancel the tonal noise embedded in the corrupted signal (Left Line In [LCI2]). The output (Left Line Out [LCO2]) produces the clean mono audio signal (Jiang and Tan; 2012). Fig. 14.36 details the implementation.

Many other practical configurations can be implemented similarly.

14.7.2 SIGNAL QUANTIZATION USING THE TMS320C6713 DSK

Linear quantization (see Chapter 10) can be implemented as shown in Fig. 14.37. The program listed in Fig. 14.37 only demonstrates the left channel coding while the right channel coding can be easily extended. The program consists of both encoder and decoder. First, it converts the 16-bit 2's complement

```
float x[1]={0.0};  /* Tonal reference noise */
Float s[1]={0,0}; /* Audio signal */
float d[1]={0.0}; /* Corrupted signal*/
interrupt void c_int11()
{
     float lc; /*left channel input */
     float rc; /*right channel input */
      float lcnew; /*left channel output */
      float rcnew; /*right channel output */
      int i;
//Left channel and right channel inputs
    AIC23_data.combo=input_sample();
    lc=(float) (AIC23_data.channel[LEFT]);
    rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
    s[0]=lc;
    x[0]=rc;
    d[0]=s[0]+x[0];
// End of the DSP algorithm
     lcnew=d[0]; /* Send to DAC */
     rcnew=rc; /* keep the original data */
     AIC23_data.channel[LEFT]=(short) lcnew;
     AIC23_data.channel[RIGHT]=(short) rcnew;
     output_sample(AIC23_data.combo);
}
```

(A) Program segment for DSK1 (generation of the corrupted signal).

```
float x[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; /*Reference input buffer*/
float w[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; /*Adaptive filter coefficients*/
float d[1]={0.0}; /* Corrupted signal*/
float y[1]={0,0}; /* Adaptive filter output */
float e[1]={0.0}; /* Enhanced signal */
float mu=0.000000000004; /*Adaptive filter convergence factor*/
interrupt void c_int11()
{
     float lc; /*left channel input */
     float rc; /*right channel input */
      float lcnew; /*left channel output */
      float rcnew; /*right channel output */
      int i;
//Left channel and right channel inputs
    AIC23_data.combo=input_sample();
    lc=(float) (AIC23_data.channel[LEFT]);
    rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
   d[0]=lc; /*Corrupted signal*/
    for(i=19;i>0;i--) /*Update the reference noise buffer  input buffer*/
   { x[i]=x[i-1]; }
    x[0]=rc;
// Adaptive filter
   y[0]=0;
   for(i=0;i<20; i++)
   { y[0]=y[0]+w[i]*x[i];}
    e[0]=d[0]-y[0]; /* Enhanced output */
    for(i=0;i<20; i++)
   {  w[i]=w[i]+2*mu*e[0]*x[i]; }/* LMS algorithm */
// End of the DSP algorithm
     lcnew=e[0]; /* Send to DAC */
     rcnew=rc; /* keep the original data */
     AIC23_data.channel[LEFT]=(short) lcnew;
     AIC23_data.channel[RIGHT]=(short) rcnew;
     output_sample(AIC23_data.combo);
}
```

(B) Program segment for DSK 2 (LMS adaptive filter).

FIG. 14.36

Program segments for noise cancellation. (A) Program segment for DSK1 (generation of the corrupted signal). (B) Program segment for DSK 2 (LMS adaptive filter).

```
int PCMcode;
/* Sign-magnitude format: s magnitude bits, see Section 10.1,in Chapter 10 */
/* 16-bit PCM code   5-bit quantization   recovered 16-bit PCM */
// See the following example:
/*  16-bit sign-magnitude code      5-bit PCM code      16-bit recovered sign-magnitude code */
/*  sADCDEFGHIJKLMNO             sABCDE             sABCD10000000000    */
/* Note that, convert the two's complement form to the sign-magnitude form    */
/* before quantizing and at decoder site, convert                            */
/* the sign-magnitude form to the two's complement form for DAC            */
int nofbits=5; // specify the number of quantization bits
int sign[16]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,
          0x100,0x200,0x400,0x800,0x1000,0x2000,0x4000,0x8000}; // add sign bit (MSB)
int mask[16]={0x0,0x01,0x03,0x07,0x0f,0x1f,0x3f,0x7f,
          0xff,0x1ff,0x3ff,0x7ff,0xfff,0x1fff,0x3fff,0x7fff};//  mask for obtaining magnitude bits
int rec[16]={0x4000,0x2000,0x1000,0x0800,
              0x0400,0x0200,0x0100,0x0080,
              0x0040,0x0020,0x0010,0x0008,
              0x0004,0x0002,0x0001,0x0000};// the mid-point of the quantization interval
interrupt void c_int11()
{
     int lc; /*left channel input */
     int rc; /*right channel input */
     int lcnew; /*left channel output */
     int rcnew; /*right channel output */
     int temp, dec;
//Left channel and right channel inputs
    AIC23_data.combo=input_sample();
    lc=(int) (AIC23_data.channel[LEFT]);
    rc= (int) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
  /* Encoder :*/
  tmp =lc; // for the Left Line In channel
    if (tmp <0 )
    {
       tmp=-tmp;  // get magnitude bits to work with
     }
    tmp =(tmp>>(16-nofbits));
    PCMcode=tmp&mask[nofbits-1]; // get magnitude bits
   if (lc>=0)
    {
       PCMcode= PCMcode | sign[nofbits-1];  // add sign bit
    }
   /* PCM code (stored in the lower portion) */
   /* Decoder: */
      dec = PCMcode&mask[nofbits-1];  // obtain magnitude bits
      tmp= PCMcode&sign[nofbits-1]; // obtain the sign bit
      dec = (dec<<(16-nofbits)); // scale to 15 bit magnitude
      dec = dec | rec[nofbits-1]; // recovering the mid-point of the quantization interval
  lc =dec;
  if (tmp == 0x00)
   {
     lc =-dec;    // back to the two's complement form (change the sign)
   }
// End of the DSP algorithm
        lcnew=lc; /* Send to DAC */
        rcnew=lc;
        AIC23_data.channel[LEFT]=(short) lcnew;
        AIC23_data.channel[RIGHT]=(short) rcnew;
        output_sample(AIC23_data.combo);
}
```

FIG. 14.37

Encoding and decoding using the TMS320C6713 DSK.

EXAMPLE 14.16

Given a 16-bit datum: lc = −2050 (decimal), convert it to the 5-bit linear PCM code using information listed in Fig. 14.37.

Solution:

1. Encoding: We use pseudo code for illustration.

```
tmp = 2050 (decimal)=0x0802 (Hex)= 0000 1000 0000 0010 (binary)
After shifting 11 bits, tmp=0x0001 (Hex)
PCMcode =tmp&mask[5-1]=0x0001&0x000f=0x0001 (Hex) // Get magnitude bits
if lc >0 {
 PCMcode= PCMcode | sign[5-1] // add positive sign bit
} // This line is not executed since lc<0
 PCMcode=00001 (binary)
```

2. Decoding: We explain the decoding process via the pseudo code.

```
dec=PCMcode& mask[4]=0x0001& 0x000f=0x0001 (Hex)
tmp=PCMcode&sign[4]=0x0001&0x0010=0x0000 (the number is negative)
dec = dec<<11=0x0800
dec =dec | rec[4]=0x0800| 0x0400=0x0C00 (Hex)=3072 (decimal)
lc=-3072 (recovered decimal)
```

The same procedure can be followed for coding a positive decimal number.

data to the sign-magnitude format with truncated magnitude bits as required. Then the decoder converts the compressed PCM code back to the 16-bit data. The encoding and decoding can be explained in Example 14.16.

Fig. 14.38 demonstrates codes for digital μ-law encoding and decoding. It converts a 12-bit linear PCM code to an 8-bit compressed PCM code using the principles discussed in Section 10.2. Note that the program only performs left-channel coding.

14.7.3 SAMPLING RATE CONVERSION USING THE TMS320C6713 DSK

Downsampling by an integer factor of M using the TMS320C6713 is depicted in Fig. 14.39. The idea is that we set up the DSK running at the original sampling rate and update the DAC channel once for every M samples. The program (Tan and Jiang; 2008) is shown in Fig. 14.40.

Upsampling by an integer factor of L using the TMS320C6713 is depicted in Fig. 14.41. Again, we set up the DSK running at the upsampling rate f_{sL} and acquire a sample from the ADC channel once for every L samples. The program (Tan and Jiang, 2008) is listed in Fig. 14.42.

```
int ulawcode;
/* Digital mu-law definition*/
// Sign-magnitude format: s segment quantization
// s=1 for the positive value, s =0 for the negative value
// Segment defines compression
// quantization with 16 levels
// See Section 11.2, Chapter 11
/* Segment     12-bit PCM      4-bit quantization interval    */
/*    0        s0000000ABCD    s000ABCD                       */
/*    1        s0000001ABCD    s001ABCD                       */
/*    2        s000001ABCDX     s010ABCD                      */
/*    3        s00001ABCDXX      s011ABCD                     */
/*    4        s0001ABCDXXX       s100ABCD                    */
/*    5        s001ABCDXXXX        s101ABCD                   */
/*    6        s01ABCDXXXXX         s110ABCD                  */
/*    7        s1ABCDXXXXXX          s111ABCD                 */
//
/* segment     recovered 12-bit PCM       */
/*    0        s0000000ABCD               */
/*    1        s0000001ABCD               */
/*    2        s000001ABCD1               */
/*    3        s00001ABCD10               */
/*    4        s0001ABCD100               */
/*    5        s001ABCD1000               */
/*    6        s01ABCDq0000               */
/*    7        s1ABCD100000               */
/* Note that, convert the two's complement form to the sign-magnitude form    */
/* before quantizing and at decoder site, convert                   */
/* the sign-magnitude form to the two's complement form for DAC           */
interrupt void c_int11()
{
     int lc; /*left channel input */
     int rc; /*right channel input */
     int lcnew; /*left channel output */
     int rcnew; /*right channel output */
     int tmp,ulawcode, dec;
//Left channel and right channel inputs
    AIC23_data.combo=input_sample();
    lc=(int) (AIC23_data.channel[LEFT]);
    rc= (int) (AIC23_data.channel[RIGHT]);
```

FIG. 14.38

Digital μ-law encoding and decoding.

(continued)

```
// Insert DSP algorithm below
 /* Encoder :*/
 tmp =lc;
 if (tmp <0 )
 { tmp=-tmp;  // Get magnitude bits to work with
 }
 tmp =(tmp>>4); // Linear scale down to 12 bits to use the u-255 law table
  if( (tmp&0x07f0)==0x0)   // Segment 0
  {  ulawcode= (tmp&0x000f); }
  if( (tmp&0x07f0)==0x0010)  // Segment 1
  {   ulawcode= (tmp&0x00f);
      ulawcode= ulawcode | 0x10; }
  if( (tmp&0x07E0)==0x0020)   // Segment 2
  {   ulawcode= (tmp&0x001f)>>1;
      ulawcode = ulawcode |0x20; }
  if( (tmp&0x07c0)==0x0040)   // Segment 3
  {  ulawcode= (tmp&0x003f)>>2;
      ulawcode= ulawcode | 0x30; }
  if( (tmp&0x0780)==0x0080) // Segment 4
  {  ulawcode= (tmp&0x007f)>>3;
      ulawcode =ulawcode | 0x40;}
  if( (tmp&0x0700)==0x0100) // Segment 5
  {  ulawcode= (tmp&0x00ff)>>4;
      ulawcode = ulawcode | 0x50;}
  if( (tmp&0x0600)==0x0200)  // Segment 6
  {  ulawcode= (tmp&0x01ff)>>5;
      ulawcode = ulawcode | 0x60; }
  if( (tmp&0x0400)==0x0400) // Segment 7
  {  ulawcode= (tmp&0x03ff)>>6;
      ulawcode=ulawcode | 0x70; }
   if (lc>=0)
   {
      ulawcode= ulawcode|0x80;
   }
  /* u-law code (8 bit compressed PCM code) for transmission and storage */
  /* Decoder: */
     tmp = ulawcode&0x7f;
     tmp = (tmp>>4);
  if ( tmp == 0x0) // Segment 0
  { dec = ulawcode&0xf; }
  if ( tmp == 0x1)  // Segment 1
  { dec = ulawcode&0xf | 0x10; }
  if ( tmp == 0x2)   // Segment 2
  {  dec = ((ulawcode&0xf)<<1) | 0x20;
      dec= dec |0x01; }
  if ( tmp == 0x3)   // Segment 3
  {   dec = ((ulawcode&0xf)<<2) | 0x40;
      dec = dec|0x02; }
  if ( tmp == 0x4)   // Segment 4
  {   dec = ((ulawcode&0xf)<<3) | 0x80;
      dec = dec|0x04; }
  if ( tmp == 0x5)   // Segment 5
  {  dec = ((ulawcode&0xf)<<4) | 0x0100;
      dec = dec|0x08; }
  if ( tmp == 0x6)   // Segment 6
```

FIG. 14.38, CONT'D

```
  {  dec = ((ulawcode&0xf)<<5) | 0x0200;
       dec =dec|0x10; }
  if ( tmp == 0x7)   // Segment 7
  {  dec = ((ulawcode&0xf)<<6) | 0x0400;
       dec = dec |0x20; }
    tmp =ulawcode & 0x80;
  lc =dec;
  if (tmp == 0x00)
  { lc=-dec;   // Back to 2's complement form
  }
   lc= (lc<<4);  // Linear scale up to 16 bits
// End of the DSP algorithm
    lcnew=lc; /* Send to DAC */
    rcnew=lc;
    AIC23_data.channel[LEFT]=(short) lcnew;
    AIC23_data.channel[RIGHT]=(short) rcnew;
    output_sample(AIC23_data.combo);
}
```

FIG. 14.38, CONT'D

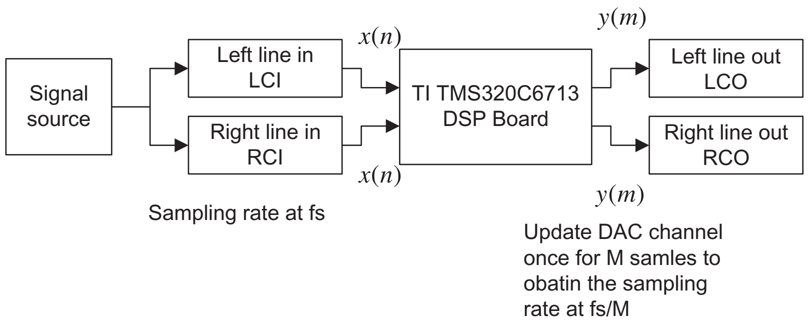


FIG. 14.39

Downsampling using the TMS320C6713.

```
int M=2;
int Mcount=0;
float x[67];
float y[1]={0.0};
interrupt void c_int11()
{
     float lc; /*Left channel input */
     float rc; /*Right channel input */
     float lcnew; /*Left channel output */
     float rcnew; /*Right channel output */
     int i,j;
     float sum;
//Left channel and right channel inputs
    AIC23_data.combo=input_sample();
    lc=(float) (AIC23_data.channel[LEFT]);
    rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
    Mcount++;
    for(i=66; i>0; i--)   // Update input buffer
    { x[i]=x[i-1]; }
     x[0]=lc;  // Load new sample
     sum=0.0;
    for(i=0;i<67;i++)     // FIR filtering
    { sum=sum+x[i]*b[i]; }
    if (Mcount== M)
    {  Mcount=0;
       y[0]=sum; }  // Update DAC with processed sample (decimation)
// End of the DSP algorithm
     lcnew=y[0]; /* Send to DAC, need to apply the anti-image filter to remove the S&H effect*/
     rcnew=y[0];
     AIC23_data.channel[LEFT]=(short) lcnew;
     AIC23_data.channel[RIGHT]=(short) rcnew;
     output_sample(AIC23_data.combo);
}
```

FIG. 14.40

Downsampling implementation. (Anti-aliasing filter has 67 coefficients which stored in an array b[67].)

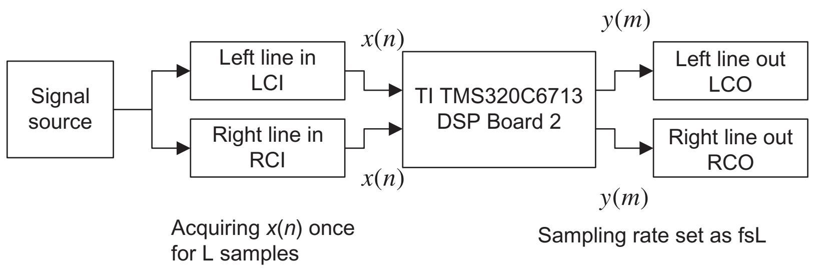


FIG. 14.41

Upsampling using the TMS320C6713.

```
int L=2;
int Lcount=0;
float x[67];
float y[1]={0.0};
interrupt void c_int11()
{
     float lc; /*Left channel input */
     float rc; /*Right channel input */
     float lcnew; /*Left channel output */
     float rcnew; /*Right channel output */
     int i,j;
//Left channel and right channel inputs
    AIC23_data.combo=input_sample();
    lc=(float) (AIC23_data.channel[LEFT]);
    rc= (float) (AIC23_data.channel[RIGHT]);
// Insert DSP algorithm below
    Lcount++;
    for(i=66; i>0; i--)   // Update input buffer with zeros
    { x[i]=x[i-1];  }
     x[0]=0;
    if (Lcount==L)
    {
      x[0]=lc;  // Load new sample for every L samples
      Lcount =0;
    }
    y[0]=0.0;
    for(i=0;i<67;i++)     // FIR filtering
    {  y[0]=y[0]+x[i]*b[i]; }
    y[0]=(float) L*y[0]
// End of the DSP algorithm
     lcnew=y[0]; /* Send to DAC */
     rcnew=y[0];
     AIC23_data.channel[LEFT]=(short) lcnew;
     AIC23_data.channel[RIGHT]=(short) rcnew;
     output_sample(AIC23_data.combo);
}
```

FIG. 14.42

Upsampling implementation. (Anti-image filter has 67 coefficients which stored in an array b[67].)

14.8 SUMMARY

1. The Von Neumann architecture consists of a single, shared memory for programs and data, a single bus for memory access, an arithmetic unit, and a program control unit. The Von Neumann processor operates fetching and execution cycles seriously.
2. The Harvard architecture has two separate memory spaces dedicated to program code and to data, respectively, two corresponding address buses, and two data buses for accessing two memory spaces. The Harvard processor offers fetching and executions in parallel.
3. The DSP special hardware units include an MAC dedicated to DSP filtering operations, a shifter unit for scaling and address generators for circular buffering.
4. The fixed-point DSP uses integer arithmetic. The data format Q-15 for the fixed-point system is preferred to avoid the overflows.

5. The floating-point processor uses the floating-point arithmetic. The standard floating-point formats include the IEEE single-precision and double-precision formats.
6. The architectures and features of fixed-point processors and floating-point processors were briefly reviewed.
7. Implementing digital filters in the fixed-point DSP system requires scaling filter coefficients so that the filters are in Q-15 format, and input scaling for adder so that overflow during the MAC operations can be avoided.
8. The floating-point processor is easy to code using the floating-point arithmetic and develop the prototype quickly. However, it is not efficient in terms of the number of instructions it has to complete compared with the fixed-point processor.
9. The fixed-point processor using fixed-point arithmetic takes much effort to code. But it offers the least number of the instructions for the CPU to execute.
10. Additional real-time DSP examples are provided, including adaptive filtering, signal quantization and coding, and sample rate conversion.

14.9 PROBLEMS

14.1 Find the signed Q-15 representation for a decimal number 0.2560123.

14.2 Find the signed Q-15 representation for a decimal number −0.2160123.

14.3 Find the signed Q-15 representation for a decimal number −0.3567921.

14.4 Find the signed Q-15 representation for a decimal number 0.4798762.

14.5 Convert the Q-15 signed number = 1.010101110100010 to a decimal number.

14.6 Convert the Q-15 signed number = 0.001000111101110 to a decimal number.

14.7 Convert the Q-15 signed number = 0.110101000100010 to a decimal number.

14.8 Convert the Q-15 signed number = 1.101000100101111 to a decimal number.

14.9 Add the following two Q-15 numbers:

$$1.101010111000001 + 0.010001111011010$$

14.10 Add the following two Q-15 numbers:

$$0.001010101000001 + 0.010101111010010$$

14.11 Add the following two Q-15 numbers:

$$1.001010101000001 + 1.010101111010010$$

14.12 Add the following two Q-15 numbers:

$$0.001010101000001 + 1.010101111010010$$

14.13 Convert each of the following decimal numbers to a floating-point number using the format specified in Fig. 14.10.

(a) 0.1101235
(b) −10.430527

14.14 Convert each of the following decimal numbers to a floating-point number using the format specified in Fig. 14.10.

(a) 2.5568921
(b) −0.678903
(c) 0.0000000
(d) −1.0000000

14.15 Add the following floating-point numbers whose formats are defined in Fig. 14.10, and determine the sum in the decimal format.

$$1101\,011100011011 + 0100\,101111100101.$$

14.16 Add the following floating-point numbers whose formats are defined in Fig. 14.10, and determine the sum in the decimal format.

$$0111\,110100011011 + 0101\,001000100101.$$

14.17 Add the following floating-point numbers whose formats are defined in Fig. 14.10, and determine the sum in the decimal format.

$$0001\,000000010011 + 0100\,001000000101.$$

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

$$110100000.010\ldots0000.$$

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

$$010100100.101\ldots0000.$$

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

$$011000\ldots0.1010\ldots0000$$

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

$$011000\ldots0.0110\ldots0000$$

14.22 Given the following FIR filter:

$$y(n) = -0.2x(n) + 0.6x(n-1) + 0.2x(n-2),$$

with a passband gain of 1 and the input being a full range, develop the DSP implementation equations in the Q-15 fixed-point system.

14.23 Given the following IIR filter,

$$y(n) = 0.6x(n) + 0.3y(n-1),$$

with a passband gain of 1 and the input being a full range, use the direct-form I method to develop the DSP implementation equations in the Q-15 fixed-point system.

14.24 Repeat Problem 14.23 using the direct-form II.

14.25 Given the following FIR filter:

$$y(n) = -0.36x(n) + 1.6x(n-1) + 0.36x(n-2),$$

with a passband gain of 2 and the input being half of range, develop the DSP implementation equations in the Q-15 fixed-point system.

14.26 Given the following IIR filter:

$$y(n) = 1.35x(n) + 0.3y(n-1),$$

with a passband gain of 2, and the input being half of range, use the direct-form I method to develop the DSP implementation equations in the Q-15 fixed-point system.

14.27 Repeat Problem 14.26 using the direct-form II.

14.28 Given the following IIR filter:

$$y(n) = 0.72x(n) + 1.42x(n-2) + 0.72x(n-2) - 1.35y(n-1) - 0.5y(n-2),$$

with a passband gain of 1 and a full range of input, use the direct-form I to develop the DSP implementation equations in the Q-15 fixed-point system.

14.29 Repeat Problem 14.28 using the direct-form II.