Chapter 4

Architectures for Programmable Digital Signal-Processing Devices

4.1 Introduction

In this chapter, architectural features of programmable DSP devices are described based on the DSP operations these devices are generally required to perform. The features are examined from the points of view of functional needs, programmability, speed, and interfacing requirements of these devices. Commonly used hardware implementations are also described for various functional units. Following are the topics covered in this chapter:

Basic architectural features
DSP computational building blocks
Bus architecture and memory
Data addressing capabilities
Address generation unit
Programmability and program execution
Speed issues
Features for external interfacing

4.2 Basic Architectural Features

A programmable DSP device should provide instructions similar to a microprocessor. These instructions can then be used to design programs for implementing DSP algorithms. The basic computational capabilities provided by way of instructions should include the following [1–3, 11]:

- Arithmetic operations such as add, subtract, and multiply.
- Logic operations such as AND, OR, XOR, and NOT.

- Multiply and accumulate (MAC) operation.
- Signal scaling operations for scaling the signal before and/or after digital signal processing.

It is important that dedicated high-speed hardware be provided to carry out these operations. For instance, multiply operation can be done much faster on a hardware multiplier than on a microcoded multiplier realized using the shift and add technique, as is often done in microprocessors.

In addition to the computational units, support architecture should include the following hardware features [10]:

- On-chip registers for storage of intermediate results.
- On-chip memories for signal samples (RAM).
- On-chip program memory for programs and fixed data such as filter coefficients (ROM).

▷ **Example 4.1** Investigate the basic features that should be provided in the DSP architecture to be used to implement the following N^{th}-order FIR filter:

$$y(n) = \sum_{i=0}^{N-1} h(i)x(n-i); \quad n = 0, 1, 2, \ldots \tag{4.1}$$

where $x(n)$ denotes the input sample; $y(n)$, the output sample; and $h(i)$, the ith filter coefficient. $x(n-i)$ is the input sample i samples earlier than $x(n)$.

Solution The FIR filter requires the following basic features for implementing Eq. 4.1:

1. Memory for storage of signal samples $x(n)$, $x(n-1)$, ..., etc. (RAM).
2. Memory for storage of filter coefficients: $h(0)$, $h(1)$, ..., etc. (ROM).
3. A hardware multiplier and an adder to carry out the multiply and accumulate (MAC) operation.
4. A register to keep track of accumulation (accumulator).
5. A register to point to the current signal sample being used (signal pointer).
6. A register to point to the current filter coefficient being used (coefficient pointer).
7. A register to keep count of the MAC operations that remain to be done (counter).
8. Capability to scale the signal value $x(n)$ as it is read from the memory and the computed signal $y(n)$ as it is stored in the memory (shifters at input and output).

Computational units such as the multiplier, the arithmetic logic unit (ALU), shifters, etc. will be described in the next section. Subsequent sections will examine the other functional units such as the memory, the addressing unit and the program execution unit.

4.3 DSP Computational Building Blocks

In this section, we learn about the hardware building blocks that carry out the basic DSP computations. While choosing these computational building blocks, we keep in mind the requirements of speed and accuracy, which are the two key issues in the design of DSP systems. At the same time, we should ensure that such building blocks could be configured to implement many different applications. That is, while each building block should be optimized for functionality and speed, the design should be sufficiently general so that it can be easily integrated with other blocks to implement overall DSP systems.

Following are the basic building blocks that are essential to carry out DSP computations [5–9]:

- Multiplier
- Shifter
- Multiply and accumulate (MAC) unit
- Arithmetic logic unit

In the following subsections, we shall discuss each of these blocks in detail.

4.3.1 Multiplier

The advent of single-chip multipliers and their integration into the microprocessor architecture are the most important reasons for the availability of commercial VLSI chips capable of implementing DSP functions. These multipliers, called *parallel* or *array multipliers*, implement complete multiplication of two binary numbers to generate the product in a single processor cycle. Earlier multiplication schemes relied either on software such as the shift and add algorithm or on microcoded controllers, which implement the same algorithm in hardware. Both these options require several processor cycles to complete the multiplication. The advances made in VLSI technology, both in terms of speed and size, have made possible the hardware implementation of parallel multipliers.

From earlier chapters, it is apparent that multiplication is one of the key operations in implementing DSP functions. However, before we design an actual multiplier, we should be clear about its specifications such as speed, accuracy, and dynamic range. The number of bits used to represent the multiplication operands and whether they are represented in fixed-point or floating-point format decide the accuracy and dynamic range of the multiplier. The speed, on the other hand, is decided by the architecture employed. For a given technology, there are several architectures for parallel multipliers, which trade off speed for reductions in circuit complexity and power dissipation. The choice of the architecture depends on the application.

				A_3 B_3	A_2 B_2	A_1 B_1	A_0 B_0
				A_3B_0	A_2B_0	A_1B_0	A_0B_0
			A_3B_1	A_2B_1	A_1B_1	A_0B_1	
		A_3B_2	A_2B_2	A_1B_2	A_0B_2		
	A_3B_3	A_2B_3	A_1B_3	A_0B_3			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

(a)

Figure 4.1(a) The 4 × 4 binary multiplication

Parallel Multiplier

Let us consider the multiplication of two unsigned numbers A and B. Let the number A be represented using m bits ($A_{m-1}A_{m-2}\ldots A_0$) and the number B, using n bits ($B_{n-1}B_{n-2}\ldots B_0$). The multiplicand A, the multiplier B, and the product P are given by [4–6]

$$A = \sum_{i=0}^{m-1} A_i 2^i \tag{4.2}$$

$$B = \sum_{j=0}^{n-1} B_j 2^j \tag{4.3}$$

$$P = \sum_{j=0}^{n-1} \left[\sum_{i=0}^{m-1} A_i B_j 2^{i+j} \right] \tag{4.4}$$

and can have a maximum of $(m+n)$ bits. Each bit of the product P is obtained by a summation of bits A_iB_j using an array of single-bit adders. The bits A_iB_j, where the index i takes on values from 0 to $m-1$, and the index j from 0 to $n-1$, are formed using AND gates. Figure 4.1(a) shows the multiplication operation using 4 bits for both A and B ($A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$). Figure 4.1(b) shows the hardware structure of the multiplier for this example. The structure is regular and requires twelve 3 input, 2 output adders. It can be shown that for an $n \times n$ multiplier, the number of adders required is $n(n-1)$.

Multiplier for Signed Numbers

The multiplier shown in Figure 4.1(b) is known as *Braun multiplier* [7] and is the basis for most of today's commercial implementations. Several improve-

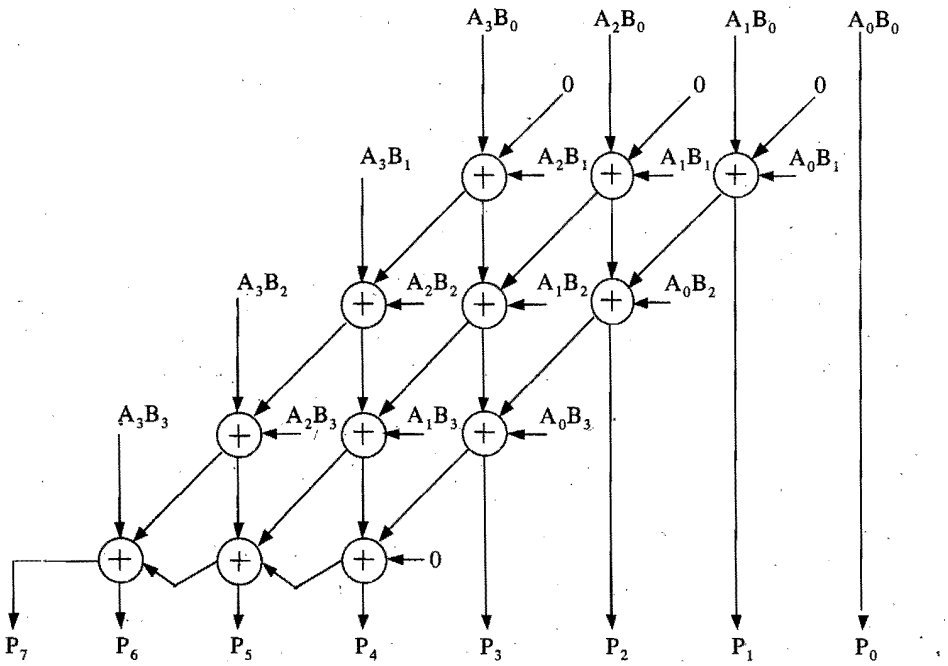


Figure 4.1(b) The structure of a 4 × 4 Braun multiplier

ments on this basic structure are possible and have been used to increase the speed and reduce the hardware complexity and power dissipation. We will not be dealing with these variations here. However, we will consider one modification of the Braun structure, which is essential to carry out multiplication of signed numbers.

Braun's multiplier does not take into account the signs of the numbers that are being multiplied. Additional hardware is required before and after the multiplication when signed numbers, represented in 2's complement form, are used. It would be desirable to have a structure that can directly operate on 2's complement numbers.

Consider two numbers A and B represented in 2's complement format. Let A have m bits and B, n bits. A and B can be written as follows:

$$A = -A_{m-1}2^{m-1} + \sum_{i=0}^{m-2} A_i 2^i \tag{4.5}$$

$$B = -B_{n-1}2^{n-1} + \sum_{j=0}^{n-2} B_j 2^j \tag{4.6}$$

The product $P = P_{m+n-1} \ldots P_1 P_0$ can be written as

$$P = A_{m-1}B_{n-1}2^{m+n-2} + \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} A_i B_j 2^{i+j} - \sum_{i=0}^{m-2} A_i B_{n-1} 2^{n-1+i}$$

$$- \sum_{j=0}^{n-2} A_{m-1} B_j 2^{m-1+j} \tag{4.7}$$

The two subtractions in Eq. 4.7 can be expressed as additions of 2's complement numbers. In doing so, Eq. 4.7 gets modified to an expression with only additions and no subtractions and can then be implemented through a structure similar to the Braun multiplier using only adders. The modified structure for handling signed numbers is called the *Baugh–Wooley multiplier* [8].

Speed

The shift and add technique of multiplication normally used in microprocessors requires n processor cycles to carry out an $n \times n$ multiplication. The cycle time is the time to access the operands, perform add and shift, and store the result in the product register. The parallel multiplier, on the other hand, is a fully combinational implementation, and once the operands are made available to the multiplier, the multiplication time is only the longest path delay time through the gates and adders.

Normally, one would want to achieve the highest possible speed of operation for a given DSP function. This would mean a multiplication time comparable to the processing times of other computational units as well as the access times of memories holding the program and data. As memory technology advances, lower and lower access times are achieved. In order to make the best use of such speeds in a DSP implementation, it would be highly desirable to design multipliers operating at the highest possible speeds. This is possible only with a fully parallel implementation.

Bus Widths

Consider a multiplier with inputs X and Y and the product Z. If X and Y are represented with n bits each, Z can have a maximum of $2n$ bits. Let us assume that both X and Y are in the memory and the product Z has also to be written back to the memory. A single-cycle execution of the multiplication will then require two buses of width n bits each (for X and Y) and a third bus of width $2n$ bits (for Z). This type of bus architecture is expensive to implement. A number of practical considerations, however, make it possible to realize the multiplication with a less extensive bus architecture. First, the program bus can be used to transfer one of the operands (say, Y) after the multiplication instruction has been fetched from the program memory. This does not cause

an additional overhead when repeated multiplications are carried out, as is generally the case with many DSP algorithms. This is because, the instruction, once fetched, usually resides in an on-chip cache. Second, a separate bus for the product Z can be dispensed with, since one of the buses (say, that of X) can be used to transfer the product to the memory as the operand X would have been latched long before the product Z is made available. To handle the $2n$ bits of Z, there are two available alternatives:

a. Use the X bus (n bits) and save Z at two successive memory locations using two memory accesses.
b. Discard the lower n bits of Z and save only the higher n bits. This is the option most often used since one of the two operands X and Y (usually Y) is normalized to one before multiplication so that the n bits discarded from Z are the less significant fractional bits. However, if the product Z is to be further processed (e.g., added to the previous result as is the case in a multiply and accumulate operation), all $2n$ bits of the product Z are retained and passed on to the next stage to retain the accuracy of the product. The decision on discarding lower-order bits or saving the entire word is made after the accumulation process is completed.

For applications in which speed is not the main issue, buffers and latches may be provided at inputs and the output, as shown in Figure 4.2. A single bus can then be used to preload the operands in the input latches before the multiplication and transfer the result from the output latches/buffers to the memory or the next stage, if necessary in two cycles after the multiplication.

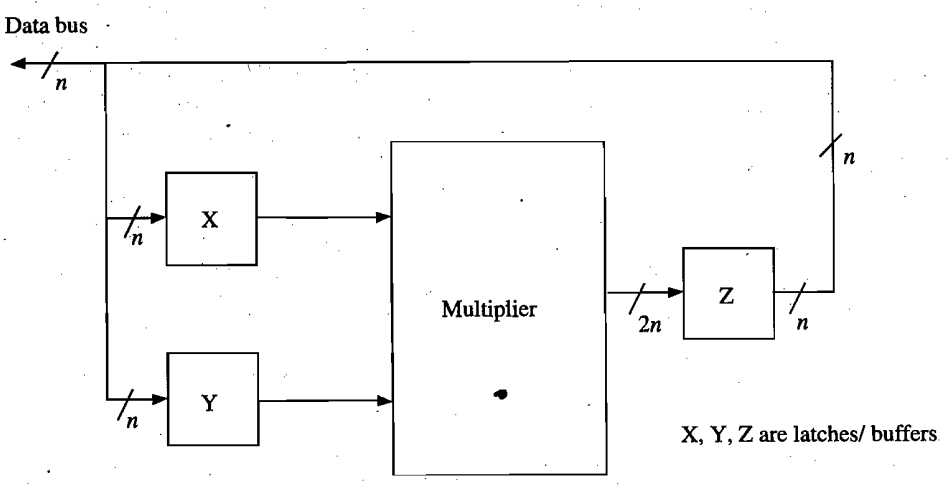


Figure 4.2 A multiplier with input and output latches/buffers

4.3.2 Shifter

Shifter is an essential component of any DSP architecture. Shifters are required to scale down or scale up operands and results to avoid errors resulting from overflows and underflows during computations. Let us consider the following cases:

a. It is required to compute the sum of N numbers, each represented by n bits. As the accumulated sum grows, the number of bits required representing it increases. The maximum number of bits to which the sum can grow is $(n + \log_2 N)$ bits. However, if each of the N numbers is scaled down by $\log_2 N$ bits prior to the addition, the loss of the result due to overflow can be avoided. The accumulator will then hold the sum scaled down by $\log_2 N$ bits. Although the accuracy of the sum is reduced because of the loss of $\log_2 N$ lower-order bits, the summation would be completed without the occurrence of the overflow error. The actual sum can be obtained by scaling up the result by $\log_2 N$ bits, when required.

b. When two numbers, each represented by n bits, are multiplied, the product can have a maximum of $2n$ bits. When this product is saved in memory, which is also n bits wide, the lower-order n bits are generally discarded, resulting in loss of accuracy. However, in the case of multiplication of two signed numbers, the accuracy can be slightly improved by shifting the product by one bit position to the left before saving the n higher-order bits. This is because the $2n$-bit product will have two sign bits, and even after discarding one of them (by a single-bit left shift), the sign of the product is still preserved. The accuracy improves because, instead of discarding all the n lower-order bits, we now discard only $(n - 1)$ bits.

c. When carrying out floating-point additions, the operands should be normalized to have the same exponent. This is accomplished by shifting one of the operands by the required number of bit positions so that it has the same exponent as the other operand.

The cases illustrated above are examples of situations that require shifting of data while implementing DSP operations.

▷ **Example 4.2** It is required to find the sum of 64 numbers each represented by 16 bits. How many bits should the accumulator have so that the sum can be computed without the occurrence of overflow error or loss of accuracy?

Solution When 64 numbers are added, the sum can grow by a maximum of $\log_2 64 = 6$ bits. To avoid overflow, the total number of bits the accumulator should have is $16 + 6 = 22$.

▷ **Example 4.3** If, for the problem of Example 4.2, it is decided to have an accumulator with only 16 bits but shift the numbers before the addition to prevent overflow, by how many bits should each number be shifted?

Solution Since the sum can grow by 6 bits, in order to prevent overflow, each number should be shifted by 6 bits to the right before the addition.

▷ **Example 4.4** If all the numbers in the problem of Example 4.3 are fixed-point integers, what is the actual sum of the numbers?

Solution Since each number has been shifted to the right by 6 bits, the sum should be shifted left by 6 positions to get the actual value.

$$\text{The actual sum} = (\text{content of the accumulator}) \times 2^6$$

▷ **Example 4.5** What is the error in the computation of the sum in the problem of Example 4.4?

Solution Since the six lowest significant bits have been lost in the process of summation, the sum could be off by as much as $2^6 - 1 = 63$.

Barrel Shifter

In conventional microprocessors shifting is normally implemented by an operation similar to the one performed in a shift register. The operation takes one clock cycle for every single bit shift. Such a scheme requires unduly large amounts of time to implement multibit shifts, which are generally required in DSP computations. In DSPs, on the other hand, in order to preserve the computational speed of single-cycle instruction execution, shifts by several bits should be accomplished in a single cycle. This is possible by a combinational circuit known as the *barrel shifter*. The barrel shifter connects the input lines representing a word to a group of output lines with the required shift determined by its control inputs, as shown in Figure 4.3(a). Control input also determines the direction of the shift (left or right). If the input word has n bits, and shifts from 0 to $n - 1$ bit positions to the right or left are to be implemented, the control input requires $\log_2 n$ lines to determine the number of bits to be shifted. Further, an additional line is also required for the control input to indicate the direction of the shift. In practice, however, the direction of shift is usually fixed, with the result that only $\log_2 n$ lines are required for the control input. Bits shifted out of the input word are discarded and the new bit positions are filled with zeros in the case of left shift. In the case of right shift, the new bit positions are replicated with the most significant bit to maintain the sign of the shifted result.

Figure 4.3(b) shows an implementation of a barrel shifter with four input bits, $(A_3A_2A_1A_0)$ and four output bits $(B_3B_2B_1B_0)$. Using this shifter, it is

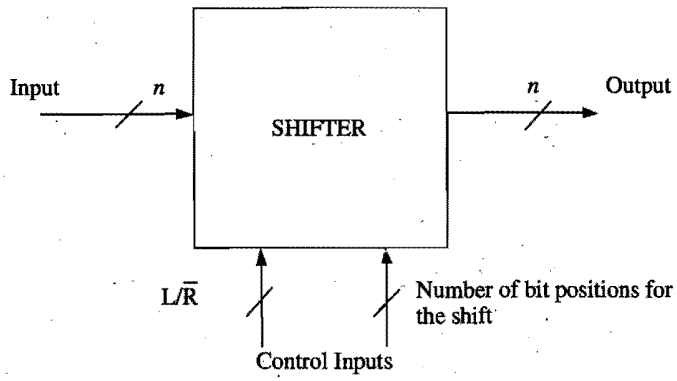
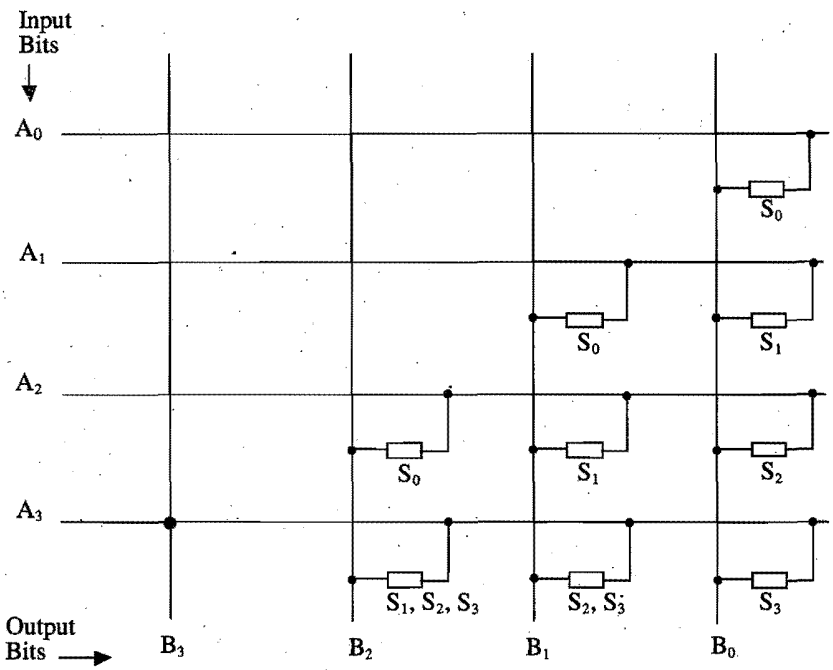


Figure 4.3(a) Block diagram of a barrel shifter



Input	Shift (Switch)	Output ($B_3B_2B_1B_0$)
$A_3A_2A_1A_0$	0 (S_0)	$A_3A_2A_1A_0$
$A_3A_2A_1A_0$	1 (S_1)	$A_3A_3A_2A_1$
$A_3A_2A_1A_0$	2 (S_2)	$A_3A_3A_3A_2$
$A_3A_2A_1A_0$	3 (S_3)	$A_3A_3A_3A_3$

Figure 4.3(b) Implementation of a 4-bit, shift-right barrel shifter

possible to realize right shift by 0, 1, 2, or 3 bit positions by setting the control inputs (S_0, S_1, S_2, or S_3) high, respectively. Only one of the control inputs can be high at any time and this input closes all the switches controlled by it and enables the appropriate paths between the inputs and the outputs.

Since the circuit for a barrel shifter is a combinational logic circuit, the time taken to implement the shift is the total combinational delay involved in decoding the control lines and setting up the path from the input lines to the output lines. This delay is only a fraction of a clock cycle. In fact, in practical DSPs, shifting is combined with data transfer. Both operations are executed in a single clock cycle.

▷ **Example 4.6** A barrel shifter is to be designed with 16 inputs for left shifts from 0 to 15 bits. How many control lines are required to implement the shifter?

Solution The number of control lines required is four, since 4 bits are needed to code any number between 0 and 15, the range over which the shift is required to be accomplished.

4.3.3 Multiply and Accumulate (MAC) Unit

Most DSP applications such as filters require the accumulation of the products of a series of successive multiplications. In order to implement this accumulation, we need an add/subtract unit and an additional register called the *accumulator* at the output of the multiplier. The configuration of such a multiply and accumulate unit, commonly known as the MAC unit, is shown in Figure 4.4.

The MAC unit consists of a multiplier that multiplies two n-bit numbers X and Y and gives a product $2n$ bits wide. This is added to or subtracted from the contents of the accumulator in the add/sub unit. The result is saved in the accumulator. The MAC unit can thus be used to implement functions of the type $A + BC$. If the accumulator is cleared at the start of a series of multiplications, it will contain the accumulated sum of the products on completion of all the multiplications.

Although multiplication and accumulation are two distinct operations, each normally requiring a separate instruction execution cycle, the two can work in parallel. At a time when the multiplier is computing a product, the accumulator accumulates the product of the previous multiplication. If N products are to be accumulated, $N - 1$ multiplies can overlap with accumulations. During the very first multiply, the accumulator is idle since there is nothing to accumulate. Likewise, during the very last accumulation, the multiplier is idle since all the N products have been computed. Thus it takes a total of $N + 1$ instruction execution cycles to compute the sum of products of N multiplications. If N is large, this works out to a speed of nearly one multiply and accumulate (MAC) operation per instruction execution cycle. This pipelined

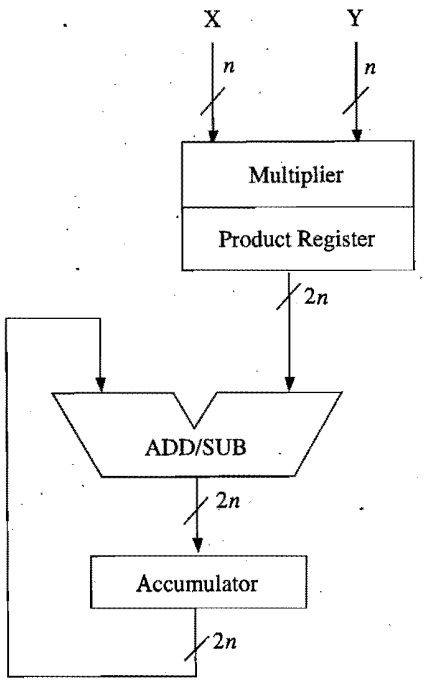


Figure 4.4 A MAC unit

operation of a multiplier and an accumulator working in parallel to effectively execute a MAC operation per cycle is a standard feature of many commercial DSP devices.

▷ **Example 4.7** If a sum of 256 products is to be computed using a pipelined MAC unit, and if the MAC execution time of the unit is 100 nsec, what will be the total time required to complete the operation?

Solution To carry out 256 MAC operations, 257 execution cycles are required.

$$\text{The total time required} = 257 \times 100 \times 10^{-9} \text{ sec} = 25.7 \text{ μsec.}$$

Overflow and Underflow

When designing a MAC unit, one has to pay attention to the word sizes encountered at the input of the multiplier and the sizes of the add/subtract unit and the accumulator, as overflow and underflow conditions may be encoun-

tered otherwise. Provision of barrel shifters at the inputs and the output of the MAC unit, provision of guard bits in the accumulator, and provision of saturation logic are the frequently used techniques to prevent overflow and underflow conditions from occurring in the MAC unit. Now let us consider each of these provisions in detail.

Shifters

Shifters are normally provided at the inputs and the output of the MAC unit. The input shifters help to normalize data samples and/or filter coefficients as they are fed into the multiplier, to avoid overflow of the accumulated result at the output. Likewise, the shifter at the output is used to denormalize the result after the sum of products computation, before being saved in the memory. In addition, the output shifter may also be used to discard the redundant sign bit in 2's complement product or to shift the output by the required number of positions before saving to preserve the maximum possible accuracy. This is done when the number to be saved is preceded by several leading 0s or 1s. As shifters provided in the MAC unit are typically barrel shifters, they do not require additional clock cycles to implement the shifts.

Guard Bits

Sometimes, in order to preserve accuracy, the inputs to the multiplier are not normalized. In such a case, when repetitive MAC operations are performed, the accumulated sum grows with each MAC operation. This increases the number of bits required to represent the result without loss of accuracy. One way to handle this growth is to provide extra bits in the accumulator. These extra bits, called *guard bits* or *extension bits*, allow for the growth of the accumulated sum as more and more product terms are added up. When the computation of the required sum of products is completed, the extension bits may be saved as a separate word, if required. Alternatively, the sum along with the guard bits may be shifted by the required amount and saved as a single word. When guard bits are provided in the accumulator, the size of the add/subtract unit also increases correspondingly.

▷ **Example 4.8** Consider a MAC units whose inputs are 16-bit numbers. If 256 products are to be summed up in this MAC, how many guard bits should be provided for the accumulator to prevent overflow condition from occurring?

Solution In general, the product of a 16 × 16 multiplication has 32 bits. Since 256 such products are to be summed, the sum can grow by a maximum of $\log_2 256 = 8$ bits. Therefore, the number of guard bits required to prevent the occurrence of overflow is 8.

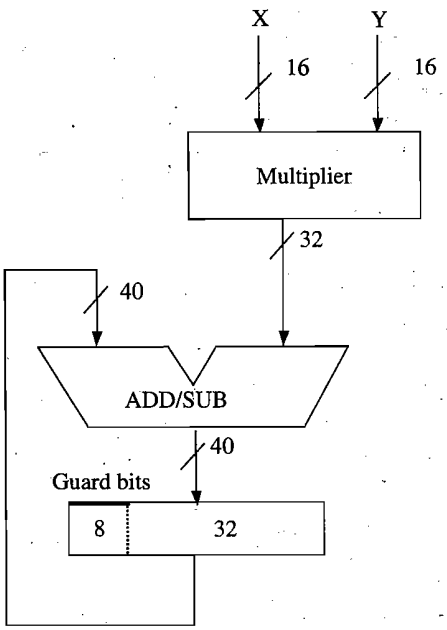


Figure 4.5 A MAC unit with accumulator guard bits

Figure 4.5 shows a block diagram of the MAC unit with guard bits for this example.

Saturation Logic

With or without guard bits, an overflow condition occurs when the accumulated result becomes larger than the largest number it can hold. Likewise, when handling a negative number, an underflow will occur if the contents of the accumulator become smaller than the smallest number it can hold. In such situations, it may be better to limit the accumulator contents to the most positive (or the most negative) value to avoid an error known as the wrap-around error.

Limiting the accumulator contents to its saturation limits is achieved with a simple logic circuit called the *saturation logic*. The circuit, shown in Figure 4.6, detects the overflow and underflow condition and accordingly loads the accumulator with the most positive or the most negative value, overriding the value computed by the MAC unit. The overflow/underflow condition is detected by monitoring the carry into the MSB and the carry out of the MSB. If carry-in is not equal to carry-out, the overflow/underflow condition occurs. The selection between the most negative and the most positive numbers is made based on the sign bit of the number.

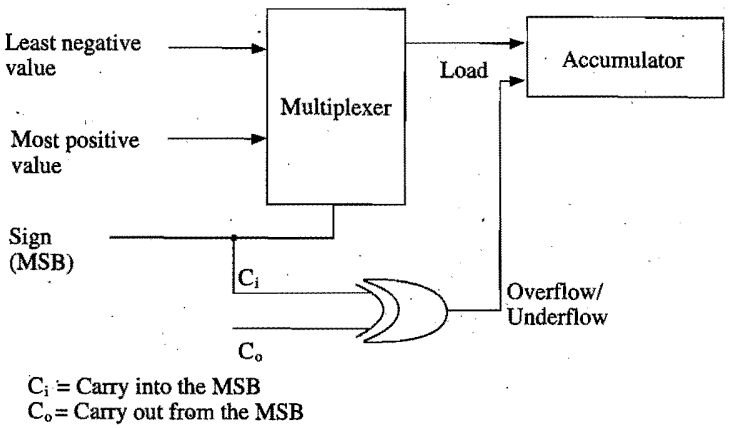


Figure 4.6 A schematic diagram of the saturation logic

4.3.4 Arithmetic and Logic Unit

In addition to shift, multiply, and multiply-and-accumulate (MAC) operations, a DSP is required to carry out several arithmetic and logic operations. These are the operations, such as add, subtract, increment, decrement, negate, AND, OR, NOT, EXOR, and compare, that are also implemented in a conventional microprocessor. This means that the ALU of a DSP is similar to the ALU of a microprocessor but with additional features such as shift and multiply discussed in the earlier sections. Figure 4.7 shows the block diagram of the ALU of a typical DSP device.

Apart from providing arithmetic and logic functions, the design of an ALU for a DSP incorporates several other features borrowed from a general-purpose microprocessor. Three of these features are discussed next

Status Flags

It is important to know the status of the accumulator after arithmetic or a logic operation. This information is used for program sequencing and scaling. The ALU includes circuitry to generate status flags after arithmetic and logic operations. These flags include sign, zero, carry, and overflow. For instance, if the execution of an instruction results in overflow, the overflow flag is set; otherwise it is reset.

Overflow Management

Features similar to those explained in the previous section on MAC are also required in the ALU for overflow management. These features are generally

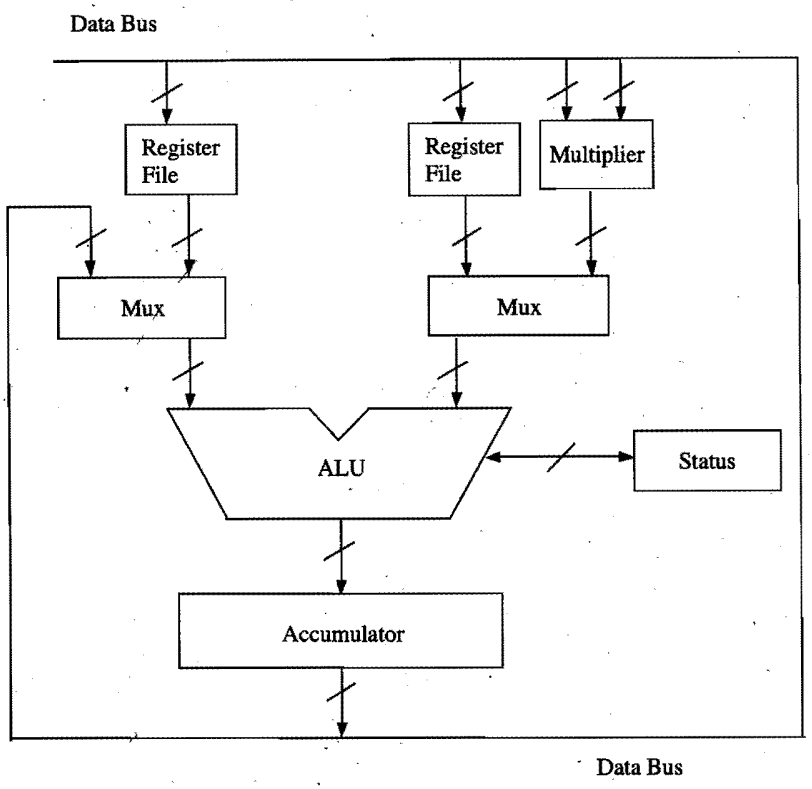


Figure 4.7 Block diagram of an arithmetic logic unit

combined with the status flags. For example, depending on the status of the overflow and the sign flags, the saturation logic can come into effect to limit the accumulator contents to its most positive or the most negative value.

Register File

A feature that improves the efficiency of an ALU is the implementation of a large general-purpose register file. Instead of moving data in and out of the ALU to memory during the course of an arithmetic computation, it may be faster to have intermediate results of arithmetic computations stored in the ALU until the computation is complete and the result is ready to be saved. This is possible by providing a file of general-purpose registers in addition to the accumulator as part of the ALU architecture.

In conventional microprocessors, the von Neumann architecture is used, wherein the program and the data reside in the same memory and a single bus (Address + Data) is used to access both, as is shown in Figure 4.8(a). This slows down the program execution considerably as the processor has to wait for the data even after the instruction is made available to it. In order to avoid this waiting and to speed up the program execution, it is desirable to have the program and data reside in two separate memories and have two buses for the processor to access the two memories. This modification, which is called

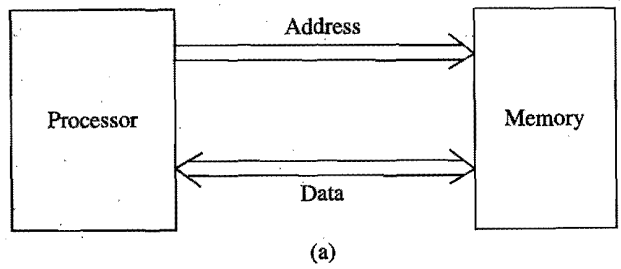


Figure 4.8(a) The bus structure of von Neumann architecture

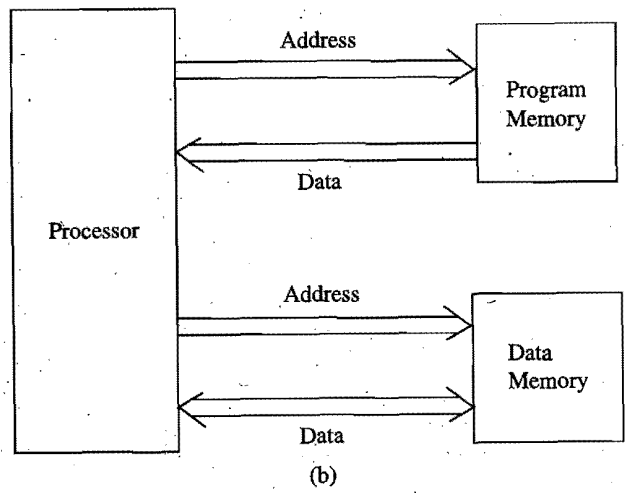


Figure 4.8(b) The bus structure of Harvard architecture

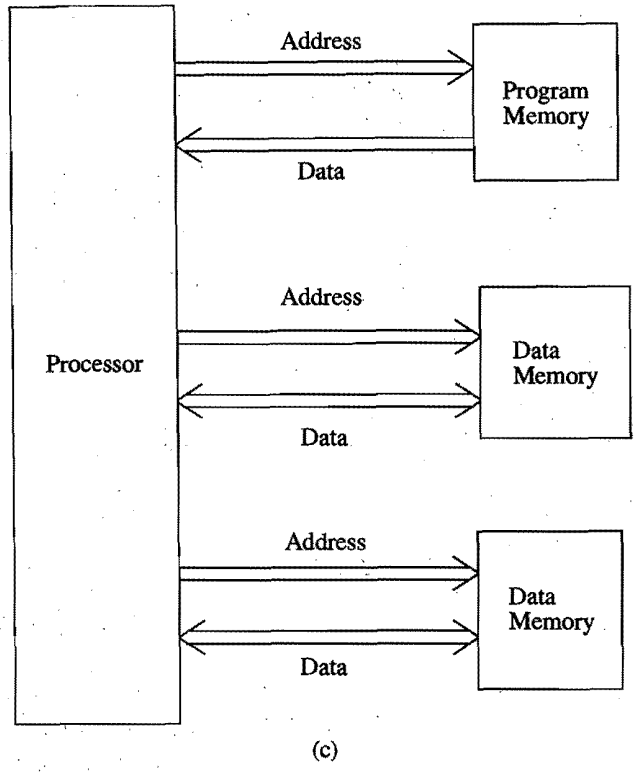


Figure 4.8(c) The bus structure for the architecture with one program memory and two data memories

the *Harvard architecture*, is shown in Figure 4.8(b). In fact, even this may not solve the problem completely. For example, the multiplication operation requires two operands to be fetched from the memory; one may be a data sample and the other, a coefficient. Even with separate memories for the program and data, it is not possible to fetch the two operands required for the multiplication along with the program instruction, and the processor has to wait for the second operand. It would therefore be required to provide dual data memories (for data and filter coefficients, for example) in addition to program memory and provide each with a separate bus for the processor to access them simultaneously. Figure 4.8(c) shows a possible bus structure of this type. As we can see, this will require a lot of hardware and interconnections to implement, thereby increasing the cost. Therefore, a compromise solution needs to be found to strike a balance between the hardware complexity and speed requirement of the multiplication operation, which is the most critical DSP operation in terms of the overall speed of algorithm implementation.

4.4.1 On-Chip Memory

A compromise between having multiple memories with individual buses for each and having fewer memories and buses is to provide some of the memories along with their buses on-chip. Even though the processor has to make simultaneous accesses to all the memories, only some of these are to the memories external to the DSP, thereby reducing the interconnection requirements to external devices.

On-chip memories help in running DSP algorithms faster than when the memories are located off-chip. This is because on-chip memories can have dedicated address and data buses unlike off-chip memories, whose buses are often multiplexed to reduce the pin count on the DSP. There are several issues related to the design of on-chip memories; two of these are considered next.

Speed

The on-chip memories should match the speeds of the ALU operations in order to maintain the single-cycle instruction execution requirement of the DSP. However, this is not a serious constraint because execution times of complex arithmetic operations such as multiplication are generally longer than memory access times. In fact, very often, more memory accesses than one are possible within a single instruction cycle, as will be explained later.

Size

Size is a major constraint for on-chip memories. In a given area of a DSP chip as many DSP functions as possible must be packed in order to get the best possible performance. On the other hand, the more area occupied by the on-chip memory, the less will be the area available for the other functions. The sizes of the on-chip memories are optimized taking into account the speed advantage, but without compromising any essential features required on the DSP.

4.4.2 Organization of the On-Chip Memory

Ideally, the entire memory required to implement a DSP algorithm should reside on-chip. This means, that the on-chip memory should be partitioned into program and data spaces. If necessary, the data memory should be further divided into separate areas for storing data samples, coefficients, and results. This way, an instruction with two operands can be fetched and executed and the result saved all in a single cycle. Writing the program and data into the on-chip memories is done before the program execution. Likewise, the results

are read off the on-chip memory after the program execution is completed. However, this scheme is not practical because the different memory blocks and their buses take an enormous amount of chip area, thereby limiting the scope of other functions that are to be provided on the chip. There are several other ways in which the on-chip memory can be organized efficiently and in a cost-effective manner.

1. Many DSP algorithms require repeated executions of a single instruction such as the multiply and accumulate or a loop consisting of a few instructions. The result is normally saved only after the repetitions are completed. It is, therefore, sufficient to provide only two blocks of on-chip memories to hold the operands required for the execution of the instructions. The instruction or instructions required to carry out the repetitive calculations can reside in the external memory and, once fetched, can be repetitively used by keeping them in an instruction cache. Since the result is to be saved less frequently, there is no need to provide a separate memory for this purpose.
2. On-chip memories can be designed such that they can be accessed more than once in an instruction cycle. This way, fewer memory blocks can serve to hold the program, the operands, and results. This means that their access times should be sufficiently small to match the timing requirements of single-cycle instruction execution. Considering the advances made in memory design technology, it is possible to integrate dual-access on-chip memories on today's commercial DSPs. For example, let us assume that there are two on-chip memories and two buses in a DSP device. If each of these memories is fast enough to be accessed twice in each instruction cycle, execution of a multiply instruction that includes an instruction fetch, two operand fetches, and a memory access to save the result can be completed in one clock cycle.
3. On-chip memories can be configured for different uses at different times depending on the requirements. For example, if a DSP has two blocks of on-chip memory, ordinarily one of them will be configured as program memory and the other as the data memory. However, for execution of instructions, which requires two operands to be fetched simultaneously, they can both be configured as data memories. The instruction itself can be fetched from an external memory or it can reside in an on-chip cache.

In addition to program memory and data memories, DSP architecture should provide for a separate stack that can be directly accessed by the program counter. This provision can considerably reduce the overheads during the subroutine and interrupt calls and returns. If the cost becomes an issue in the choice of access times required for memories in a multiple memory system, it is preferable to provide faster memories for those segments that are more frequently accessed than the others.

4.5 Data Addressing Capabilities

The data processed by a digital signal-processing scheme typically consist of signal samples and filter coefficients. An efficient way of accessing data while performing computations can go a long way in the overall performance of an implementation. The provision of flexibility in accessing data helps in writing efficient programs for various applications. The data addressing capability of a programmable DSP device is provided by means of its addressing modes. The addressing modes that can enhance DSP implementations consist of immediate, register, direct, and indirect addressing modes. We now discuss each of these modes. These modes are summarized in Table 4.1.

Table 4.1 Summary of DSP Addressing Modes

Addressing Mode	Operand	Sample Format	Operation
Immediate	Immediate value	*ADD #imm*	*#imm* + A → A
Register	Register contents	*ADD reg*	*reg* + A → A
Direct	Memory address contents	*ADD mem*	*mem* + A → A
Indirect	Memory contents with address in the register	*ADD *addrreg*	**addrreg* + A → A

Notations used in describing the operation in the table:
#imm = value represented by *imm*,
reg = contents of register *reg*,
mem = contents of memory location with address *mem*, and
**addrreg* = contents of memory location whose address is the contents of address register *addrreg*,
→ represents the transfer from left to right.

4.5.1 Immediate Addressing Mode

The capability to include data as part of the instruction is provided by the immediate addressing mode. For example, a DSP processor may allow the programmer to write the instruction

ADD #imm

to add the value represented by *imm* to the accumulator register, A. In other words, the operation

#imm + A → A

is implemented. In such an addressing mode data has to be a fixed number known at the time of writing instructions. Filter coefficients are examples of this kind of data.

4.5.2 Register Addressing Mode

In the register addressing mode a processor register provides the operand. Using this addressing mode the DSP processor may provide an instruction

$$ADD\ reg$$

to implement

$$reg + \mathrm{A} \rightarrow \mathrm{A}$$

4.5.3 Direct Addressing Mode

In the direct addressing mode a memory operand is specified by providing its memory address. For instance a DSP processor may allow an instruction

$$ADD\ mem$$

to implement

$$mem + \mathrm{A} \rightarrow \mathrm{A}$$

A signal sample stored in a memory location can be accessed using direct addressing mode. This mode, however, requires an explicit knowledge of the memory address, *mem*.

4.5.4 Indirect Addressing Mode

In the indirect addressing mode an operand is accessed using a pointer. A pointer is typically a register that holds the address of the location where the operand resides. For example, to add to the accumulator, A, the content of the memory location whose address is held in *addrreg*, the following instruction is implemented:

$$ADD\ *addrreg$$

which means

$$*addrreg + \mathrm{A} \rightarrow \mathrm{A}$$

In order to use this addressing mode, *addrreg* needs to be loaded before the use. Any memory location can be accessed by simply changing the register contents.

The indirect addressing mode can be enhanced by providing an automatic capability to manipulate the pointer register just before (pre) or just after (post) the use. The pointer register may be incremented or decremented. It

may also be possible to add or subtract the contents of another register (offset register) provided in the architecture. This leads to the following enhanced indirect addressing modes:

Post_increment addressing mode,
Post_decrement addressing mode,
Pre_increment addressing mode,
Pre_decrement addressing mode,
Post_offset_add addressing mode,
Post_offset_subtract addressing mode,
Pre_offset_add addressing mode, and
Pre_offset_subtract addressing mode.

These enhanced indirect addressing modes are summarized in Table 4.2.

Table 4.2 Enhancements to Indirect Addressing Mode

Addressing Mode	Sample Format	Operation
Post_increment	*ADD *addrreg+*	A ← A + **addrreg*, *addrreg* ← *addrreg* + 1
Post_decrement	*ADD *addrreg−*	A ← A + **addrreg*, *addrreg* ← *addrreg* − 1
Pre_increment	*ADD + *addrreg*	*addrreg* ← *addrreg* + 1, A ← A + **addrreg*
Pre_decrement	*ADD − *addrreg*	*addrreg* ← *addrreg* − 1, A ← A + **addrreg*
Post_add_offset	*ADD *addrreg, offsetreg+*	A ← A + **addrreg*, *addrreg* ← *addrreg* + *offsetreg*

(continued)

Table 4.2 Continued

Addressing Mode	Sample Format	Operation
Post_subtract_offset	*ADD *addrreg, offsetreg–*	A ← A + **addrreg*, *addrreg* ← *addrreg – offsetreg*
Pre_add_offset	*ADD offsetreg+, *addrreg*	*addrreg* ← *addrreg + offsetreg*, A ← A + **addrreg*
Pre_subtract_offset	*ADD offsetreg–, *addrreg*	*addrreg* ← *addrreg – offsetreg*, A ← A + **addrreg*

In order to realize the indirect addressing mode and its enhanced versions in a DSP architecture, additional hardware operating in conjunction with its addressing unit is required. For example to provide *pre_offset_add* addressing mode, an adder and another register to hold the offset are needed. It also means extra time for operand accessing or, alternatively, the need for computing the operand address using a dedicated address arithmetic unit working in parallel with the main arithmetic unit.

▷ **Example 4.9** What are the memory addresses of the operands in each of the following cases of indirect addressing modes? In each case, what will be the content of the *addrreg* after the memory access? Assume that the initial contents of the *addrreg* and the *offsetreg* are 0200h and 0010h, respectively.

a. ADD **addrreg–*,

b. ADD + **addrreg*

Table 4.3 Solution for Example 4.9

Instruction	Addressing Mode	Operand Address	Contents of *addrreg* after the Memory Access
a	*Post_decrement*	0200h	0200h – 1h = 01FFh
b	*Pre_increment*	0200h + 1h = 0201h	0201h
d	*Pre_add_offset*	0200h + 10h = 0210h	0210h
d	*Post_subtract_offset*	0200h	0200h – 10h = 01F0h

c. ADD *offsetreg+*, **addrreg*

d. ADD **addrreg*, *offsetreg–*

Solution The solution is given in Table 4.3.

4.5.5 Special Addressing Modes

In addition to the addressing modes mentioned earlier, special addressing modes are provided in the architecture of a DSP to implement real-time signal processing and to compute DFT using FFT algorithms. Real-time signal processing is enhanced by the provision of a circular buffer and the addressing mode that goes with it. The FFT implementation requires data to be accessed in a nonsequential, yet regular, manner. The data for FFT is accessed by what is called as *bit-reversed index*. A bit-reversed addressing mode is generally provided in the architecture to support FFT implementations. Similarly, to process two-dimensional data, it will be advantageous to provide a special addressing mode that can help access data organized in a matrix form. Now we consider two of these special addressing modes.

Circular Addressing Mode

The provision of a circular buffer allows one to handle a continuous stream of incoming data samples. In a circular buffer, successive data samples are stored in sequential buffer locations until the end of the buffer is reached. After reaching the end we start all over from the beginning of the buffer. This process can go on forever as long as the data samples get processed in a timely manner at a rate faster than the incoming data. To access a data sample from a circular buffer, a *circular addressing mode* is of great help. The implementation of such an addressing mode in hardware requires three registers: a pointer register (PNTR) to keep track of current address, a start address register (SAR) to hold the start address of the buffer, and an end address register (EAR) to hold the end address of the buffer. The pointer register should have the capability of getting incremented/decremented. Different forms of the indirect addressing mode for the pointer register are required in order to update the pointer for different applications. The pointer-updating algorithm is given in Figure 4.9.

The different cases that are encountered during the updating process of the pointer are shown in Figure 4.10. These cases are:

1. SAR < EAR, and updated PNTR > EAR
2. SAR < EAR, and updated PNTR < SAR
3. SAR > EAR, and updated PNTR > SAR
4. SAR > EAR, and updated PNTR < EAR

The buffer size in the first two cases = (EAR − SAR + 1) and in the last two it is = (SAR − EAR + 1).

```
; Pointer Updating Algorithm for the Circular Addressing Mode

Updated PNTR ← PNTR ± increment
If SAR < EAR
   and if Updated PNTR > EAR, then
          New PNTR ← Updated PNTR - Buffer size
   and if Updated PNTR < SAR, then
          New PNTR ← Updated PNTR + Buffer size
If SAR > EAR
   and if Updated PNTR > SAR, then
          New PNTR ← Updated PNTR - Buffer size
   and if Updated PNTR < EAR, then
          New PNTR ← Updated PNTR + Buffer size
Else
          New PNTR ← Updated PNTR
```

Figure 4.9 Register pointer updating algorithm for circular buffer addressing mode. SAR = start address register contents, EAR = end address register contents, PNTR = pointer

▷ **Example 4.10** A DSP has a circular buffer with the start and the end addresses as 0200h and 020Fh, respectively. What would be the new values of the address pointer of the buffer if, in the course of address computation, it gets updated to (a) 0212h, (b) 01FCh?

Solution

$$\text{The buffer length} = 020\text{Fh} - 0200\text{h} + 1 = 10\text{h}$$

a. The new value of the pointer is updated value − buffer length, i.e., 0212h − 0010h = 0202h.

b. The new value of the pointer is updated value + buffer length, i.e., 01FCh + 0010h = 020Ch.

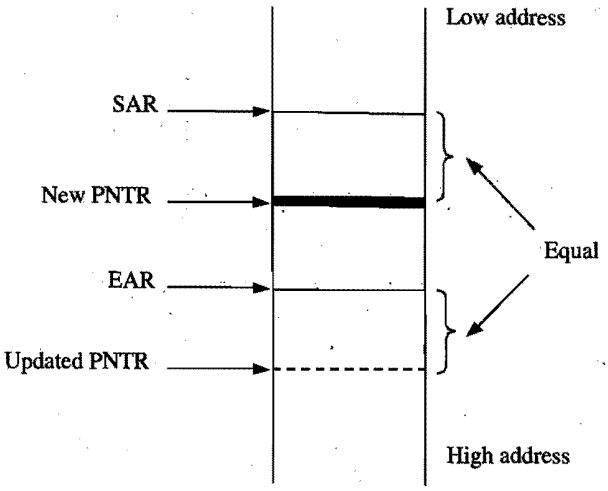
▷ **Example 4.11** Repeat the problem of Example 4.10 if the start and end addresses of the circular buffer are 0210h and 0201h, respectively.

Solution

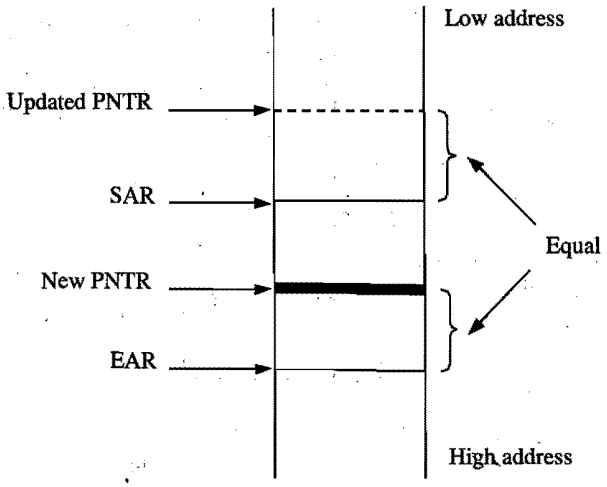
a. The new value of the pointer is the updated value − buffer length, i.e., 0212h − 0010h = 0202h.

b. The new value of the pointer is the updated value + buffer length, i.e., 01FCh + 0010h = 020Ch.

Note that these values are the same as those in the previous example. This shows that in a circular buffer, the address pointer wraps around to point to an address inside the buffer, irrespective of whether the buffer start address is higher or the end address is higher.



Case 1: SAR < EAR, and Updated PNTR > EAR

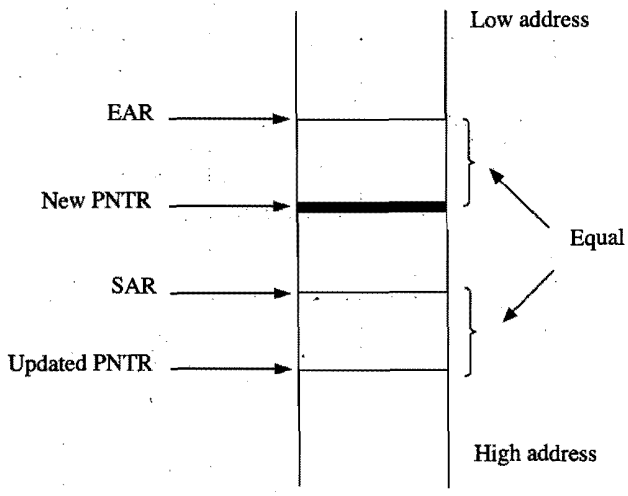


Case 2: SAR < EAR, and Updated PNTR < SAR

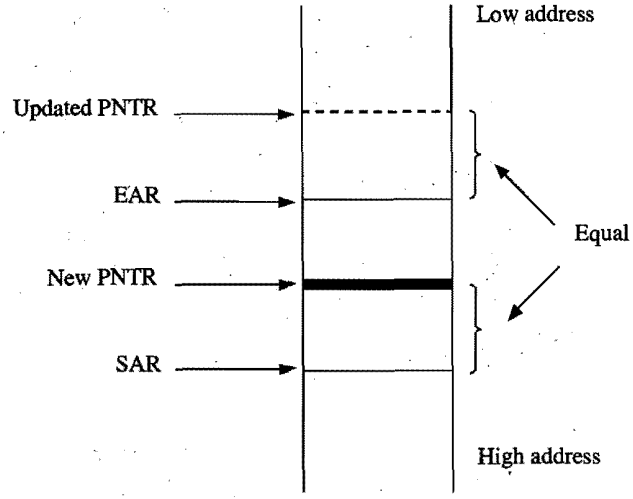
Figure 4.10 Different cases that arise in updating the pointer in circular buffer addressing mode *(continued)*

Bit-Reversed Addressing Mode

Special data access capability is needed in the FFT algorithm implementation. In the algorithm called *decimation in time* (DIT) FFT, the naturally ordered data needs to be accessed according to the indices, as shown in Table 4.4 for



Case 3: SAR > EAR, and Updated PNTR > SAR



Case 4: SAR > EAR, and Updated PNTR < EAR

Figure 4.10 Continued

an 8-point FFT. That is, in the case of an 8-point FFT, the input data $x(0)$, $x(1)$, $x(2)$, $x(3)$, $x(4)$, $x(5)$, $x(6)$, and $x(7)$ need to be accessed in the order $x(0)$, $x(4)$, $x(2)$, $x(6)$, $x(1)$, $x(5)$, $x(3)$, and $x(7)$. The interesting point is that the indices describing the order of data access can be obtained as follows: start

Table 4.4 Index Computation Using Bit-Reversed Addressing Mode for an 8-point FFT

Input Index (natural order)	Output Index (bit-reversed order)
000 = 0	000 = 0
001 = 1	100 = 4
010 = 2	010 = 2
011 = 3	110 = 6
100 = 4	001 = 1
101 = 5	101 = 5
110 = 6	011 = 3
111 = 7	111 = 7

with index 0, obtain each current index by adding (in a special way) half the size of the FFT to the corresponding previous index, i.e.,

$$\text{Current index} = \text{previous index} + \text{B}(1/2(\text{FFT size})) \tag{4.8}$$

The addition however, is different in the sense that during addition the carry must propagate from the most significant to the least significant bit.

The *reverse-carry-add* operation can be provided in the architecture to implement this special addressing mode. The architecture will require a register to keep track of the index at any time in addition to the capability to propagate the carry in the reverse direction during the add operation in order to generate the next index to be used to access data. To provide this capability in parallel with the instruction execution, a special address generation unit is employed.

▷ **Example 4.12** Compute the sequence in which the input data should be ordered for a 16-point DIT FFT.

Solution Assuming that the first sample is located at address 0, the next sample should be located at address 0 + B(length of FFT/2) = 0 + 8 = 8. This address can be arrived at by carrying out binary addition with reverse carry propagation as follows:

Initial address in binary = 0000

Half the length of the FFT in binary = 1000

Next address (add with reverse carry propagation) = 1000

To compute the address of the third sample, repeat the operation.

Initial address in binary = 1000

Half the length of the FFT in binary = 1000

Next address (add with reverse carry propagation) = 0100

The process is repeated until the addresses of all the 16 samples are computed. Table 4.5 gives the results.

Table 4.5 Solution for Example 4.12

Sample Number	Binary Address	Hexa-decimal Address
1	0000	0
2	1000	8
3	0100	4
4	1100	C
5	0010	2
6	1010	A
7	0110	6
8	1110	E
9	0001	1
10	1001	9
11	0101	5
12	1101	D
13	0011	3
14	1011	B
15	0111	7
16	1111	F

4.6 Address Generation Unit

The function of the address generation unit is to provide the addresses of the operands required to carry out the DSP operations. Since many instructions, such as the multiply instruction, require more than one operand for their execution, the address generation unit should work fast enough to provide the addresses within the time constraints imposed by the instruction execution requirements.

Further, in a DSP implementation, the address generation unit may be required to perform some computation of its own in order to arrive at the operand addresses. This is because of the need for the various enhancements

to the indirect addressing mode as well as some special addressing modes, such as the circular addressing mode and the bit-reversed addressing mode. These special features were discussed in Section 4.5. In order to carry out the computations required for the specialized addressing modes the address generation unit in a DSP implementation is provided with a separate arithmetic unit of its own. This way, address computation overhead is removed from the main ALU, thereby allowing it to perform more efficiently.

Address generation typically involves one of the following operations:

1. Getting a new value from an immediate operand, a register, or a memory location.
2. Incrementing or decrementing the current address.
3. Adding or subtracting an offset to the current address.
4. Adding or subtracting an offset to the current address, comparing the new address with the limits defined for a circular addressing mode, and generating a new address as per the circular addressing mode algorithm.
5. Generating a new address from the current address by applying the bit-reversed addressing mode algorithm.

The hardware necessary to carry out the various operations listed above may consist of the following: an ALU; registers to store the current value, the offset, and the new value; registers to store the limits of the circular buffer; logic to implement the circular addressing mode; and the logic to implement the bit-reversed addressing mode. The block diagram of a typical addressing unit is shown in Figure 4.11.

4.7 Programmability and Program Execution

A programmable DSP device needs to provide programming capability similar to that of a microprocessor. It should be possible to write programs involving branching, loops, and subroutines. The branching capability is needed in order to alter conditionally or unconditionally the normal execution sequence. The looping operation is desirable in order to repeat a section of the program the desired number of times. The subroutine handling instructions provide the capability to develop structured software.

The implementation of repeat capability should be hardware based so that it can be programmed with minimal or zero overhead. For instance, a counter is needed to keep track of the number of times the execution of a block of instructions remains to be repeated. A dedicated register for this purpose can enhance the performance. Repeat is an operation that is needed in the implementation of many DSP algorithms, and hence its hardware implementation has a direct bearing on the overall performance of a DSP scheme.

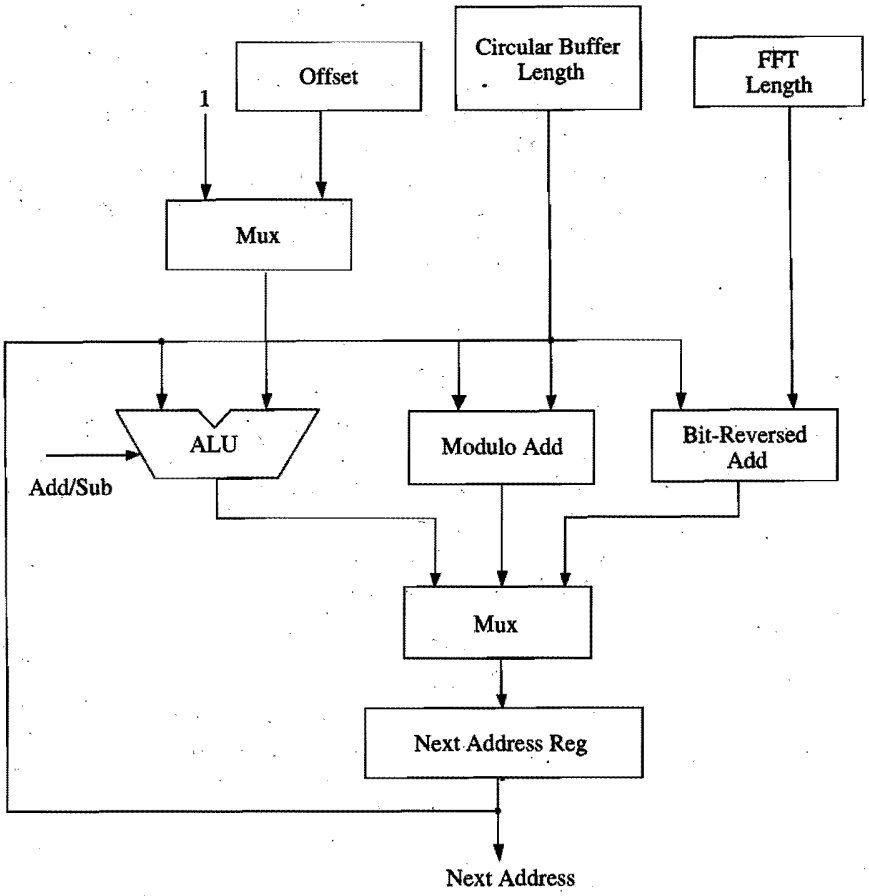


Figure 4.11 Block diagram of an address generation unit

The subroutine implementation requires saving the return address in the stack. In a general-purpose microprocessor, a part of the memory is used to implement the stack. This means that to save the return address as well as to restore it on return, the processor requires to carry out memory read and write operations using the system data bus. These operations add to the overhead and make the overall program execution slow, thereby lowering the performance. For a DSP device, it is desirable that a last-in–first-out (LIFO) buffer directly interfaces to the program counter (instruction pointer) to save the return address. This approach avoids the use of the system bus for accessing the stack and thus speeds up the subroutine branching as well as its return.

4.7.1 Program Control

Like microprocessors, a DSP requires a control unit, which provides the necessary control and timing signals for proper execution of instructions. In microprocessors, the control unit is generally implemented by means of a microcoded sequencer. Each instruction of the microprocessor is broken down into several microinstructions and stored in a microstore as a microcode. Whenever one of the instructions is to be executed, the corresponding microcode is called from the microstore and executed, in a manner very similar to the execution of subroutines in a program. This type of control unit is easy to design and implement and uses less hardware. However, it is not very fast since execution of each instruction requires several accesses to the microstore. For a DSP, on the other hand, the speed of execution of instructions is a critical issue. For this reason the design of various building blocks is optimized for speed. In a DSP, the microcoded control unit is replaced by a hardwired design. In a hardwired design, the control unit is designed as a single, comprehensive, hardware unit taking into account the complete instruction set of the DSP. Although the hardware complexity is high and the design is not easy to change to incorporate additional features, this works much faster compared to the microcoded design and reduces the overhead for the instruction execution time.

4.7.2 Program Sequencer

The program sequencer, which is a part of the control unit, generates instruction addresses in the sequence needed to access instructions. Normally, instructions are executed in the order in which they are stored in the memory. However, there are several exceptions to this normal flow. Examples are subroutines, loops, and branching. The program sequencer hardware computes the instruction address under various conditions.

After fetching each instruction from the program memory, the sequencer generates the address from which the next instruction is to be fetched. The next address is from one of the following sources:

1. The program counter, which is incremented after each instruction fetch.
2. The instruction register, which holds the address of the instruction in branching, looping, and subroutine calls.
3. The interrupt vector table, in the case of interrupt service routines.
4. The stack, which holds the return addresses in the case of return from subroutines, return from interrupt service routines, and end of loops.

Figure 4.12 shows the block diagram of a program sequencer. The program sequencer, in effect, acts as a multiplexer, which selects the address of the next

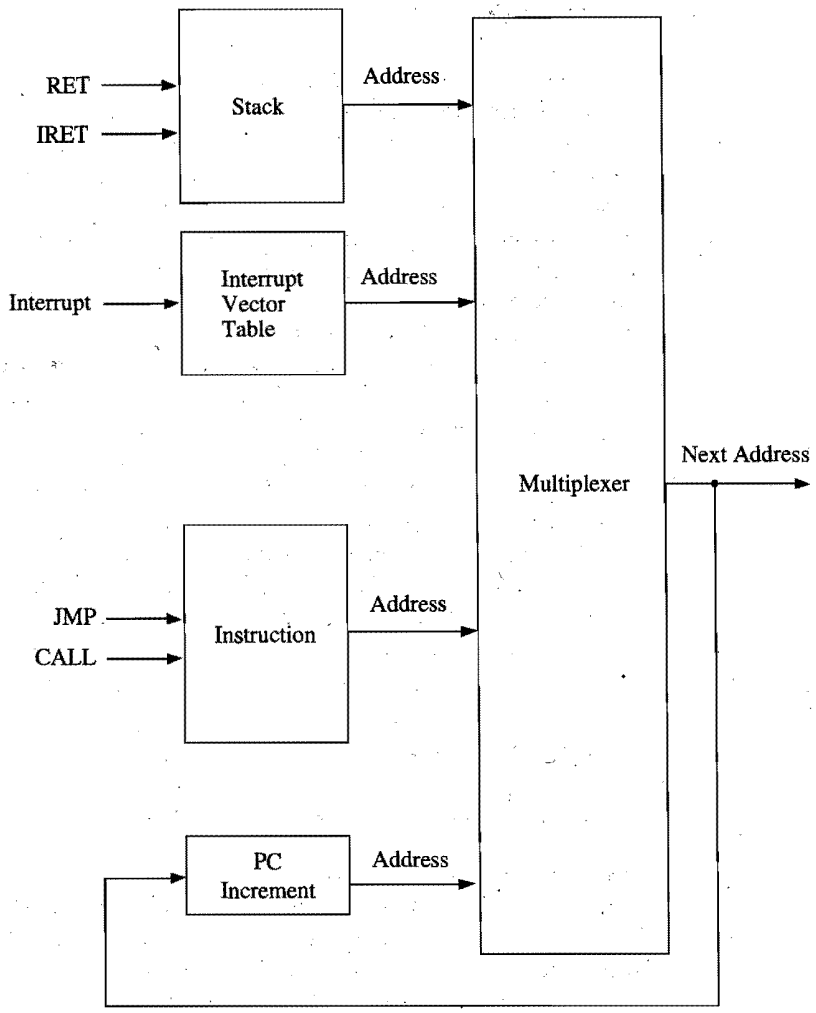


Figure 4.12 A conceptual diagram of a program sequencer

instruction to be obtained from one of the sources listed above. In order to carry out this task, several hardware features are incorporated in the program sequencer. The program counter has to be updated after every fetch. Circuitry is provided for this purpose. Counters are provided to hold the counts in the case of loop and repeat instructions. Stacks push the return addresses for subroutines and interrupt service routines and while executing loops and repeat instructions. The program sequencer also requires a logic block to test conditions under which jump and loop instructions are executed as well as to

determine when to terminate loop and repeat instructions. This logic, called the *condition logic*, tests various arithmetic conditions by means of status flags to decide if conditional jump and loop instructions are to be executed. This logic also monitors repeat and loop counters to determine when these have to be terminated to return to the normal program flow.

4.8 Speed Issues

Fast execution of algorithms is an essential requirement of a digital signal-processing architecture. In order to meet this requirement, DSP architecture must include features that facilitate high speed of operation and large throughputs. Many of these features are possible due to advances in VLSI technology and design innovations. In this section, we will discuss some of these features and see how they can increase the execution speed of the DSP architecture. We shall also discuss certain trade-offs between speed and performance in relation to some of these features.

4.8.1 Hardware Architecture

Functions such as multiplication, scaling, loops and repeats, and special addressing modes are essential for signal-processing algorithms. The architectures designed for the signal-processing applications should implement these functions in the quickest possible time. This is achieved by hardware units, which are specially designed to implement these functions. For example, conventional microprocessors implement the multiplication by means of a microprogram (microcode) using the well-known shift and add algorithm. This approach takes a large number of clock cycles to implement. In order to increase the speed of the operations considerably, parallel multipliers have been used to carry out the entire multiplication in a single clock cycle. Thanks to breakthroughs in VLSI technology, this is possible today. Similar hardware solutions have also been found to implement the other functions mentioned earlier to reduce overheads and to increase the speed. Such methods typically replace the slow microprogrammed solutions used in conventional microprocessors.

Harvard architecture, which separates the program and data memories with separate buses for each, increases the speed of execution of programs considerably. Dual data memories with individual buses for each help in accessing dual operands simultaneously.

Multiple external memories require multiple buses external to the DSP. In addition to being expensive, external buses are slow for program access and execution. By providing on-chip memories and an instruction cache, program

execution is speeded up considerably. Further, these on-chip memories can also be accessed twice in a clock cycle, thereby reducing the number of separate memories and buses required in a device.

In addition to the hardware issues mentioned earlier, there are many techniques used in DSP architectures to increase their speed of operation. We shall consider two of these techniques: parallelism and pipelining.

4.8.2 Parallelism

A very major requirement to achieve high speed of operation in DSP architecture is the provision of parallelism. Parallelism may mean several things. One is the provision of functional units, which may operate in parallel and increase the throughput. For example, instead of the same arithmetic unit being used to do computations on data and address, a separate address arithmetic unit can be provided to take care of address computations. This frees up the main arithmetic unit to concentrate on data computations alone and thereby increases the throughput. Another example, which was discussed earlier, is the provision of multiple memories and multiple buses to fetch an instruction and operands simultaneously. In short, there are many functional blocks operating simultaneously for each of the most commonly used DSP operations, such as add, multiply, shift, etc. This way, algorithms can perform more than one operation at the same time, such as adding while carrying out a multiply, shifting while reading data from memory, etc.

Availability of multiple functional units can increase the speed of the DSP architectures. They should be exploited to their full potential by structuring the instructions to carry out the required operations in parallel. This requires complex hardware to control these units, and the controller is hardwired rather than microprogrammed in order to ensure high speed. The architecture should be such that instructions and data required for a computation are fetched from the memory simultaneously.

An ideal parallelism in the DSP architecture with regard to the multiply and accumulate operation, which is the most used operation in DSP implementations, should be able to accomplish the following operations in a single clock cycle:

- Fetch instructions and multiple data required for the computation
- Shift data as they are fetched in order to accomplish scaling
- Carry out a multiplication operation on the fetched data
- Add the product to the previously computed result in the accumulator
- Save the accumulator contents in the memory storage, if required, and
- Compute new addresses for the instruction and data required for the next operation

4.8.3 Pipelining

An architectural feature to increase the speed of the DSP algorithm is pipelining. In a pipelined architecture, an instruction to be executed is broken into a number of steps. A separate unit of the architecture performs each of these steps. When the first of these units performs the first step on the current instruction, the second unit will be performing the second step on the previous instruction, the third unit will be performing the third step on the instruction prior to that, etc. If p steps were required to complete the execution of each instruction, it would take p units of time for the complete execution of each instruction. However, since all the units will work all the time, one output will flow out of the architecture at the end of each time unit, and the throughput can be maintained as one instruction per unit time. A problem with this approach is dividing each instruction into steps taking equal amounts of time to perform and designing the architectural units accordingly. In practice, however, this may not be entirely possible and the slowest unit decides the throughput. A second problem is the extra time required at the start of algorithm execution, as the pipeline has to be filled before the result of the first instruction can start to flow out. This initial delay in units of time, called the *pipeline latency*, is related to the number of units in the pipeline. Likewise, when there is a change in the instruction sequence, as in the case of a branch or a loop, the pipeline needs to be cleared before the steps of the new instruction can be loaded into the pipeline, thereby causing a delay. This condition can, however, be avoided, at the cost of additional hardware to anticipate the branch instruction ahead of time and not filling the pipeline beyond the branch instruction. As an example, let us assume that the execution of an instruction can be broken into five steps: instruction fetch, instruction decode, operand fetch, execute, and save the result. Figure 4.13 shows how a pipelined

Time Slot	Step 1	Step 2	Step 3	Step 4	Step 5	Result
t_0	Inst 1					
t_1	Inst 2	Inst 1				
t_2	Inst 3	Inst 2	Inst 1			
t_3	Inst 4	Inst 3	Inst 2	Inst 1		
t_4	Inst 5	Inst 4	Inst 3	Inst 2	Inst 1	Inst 1 complete
t_5	Inst 6	Inst 5	Inst 4	Inst 3	Inst 2	Inst 2 complete
•	•	•	•	•	•	•

Figure 4.13 Pipelining for speeding up the execution of an instruction

processor will handle this. For the sake of simplicity we will assume that all the steps take equal amounts of time.

As we can see from the figure, the output corresponding to the first instruction is available after 5 units of time. However, once the result starts to come out, we get an output after each unit of time. In other words, the steady-state throughput of the system is one instruction per unit time.

4.8.4 System Level Parallelism and Pipelining

The parallelism and pipelining concepts explained in the last two subsections can be extended to the implementation of DSP algorithms. Consider the example of an 8-tap (8 coefficients) FIR filter given by

$$y(n) = \sum_{i=0}^{7} h(i)x(n-i) \tag{4.9}$$

The filter can be implemented in many ways depending on the number of multipliers and accumulators available. Let us look at some of these implementations.

Implementation Using a Single MAC Unit

If only one multiplier and accumulator is available, it must be used 8 times to compute the eight product terms in Eq. 4.9 and find their sum. Figure 4.14(a) shows such an implementation. Each input sample is delayed from the previous sample by $8T$, where T is the time taken by the multiplier and accumulator to compute one product term and add it to the previously accumulated sum in the accumulator. Input samples and the filter coefficients are fed to the multiplier through multiplexers, which are controlled such that the correct combination of a sample and the corresponding filter coefficient are fed to the multiplier at a given time. As each product term is generated, it is added to the previously accumulated sum in the MAC unit. After all the eight product terms are accumulated, the MAC contents are available as the output. Output $y(n)$ is available $8T$ units of time after $x(n)$ is made available to the filter. At this time, a new sample $x(n+1)$ is applied to the filter. The filter then uses eight samples, namely, $x(n+1)$, $x(n)$, $x(n-1)$, ..., $x(n-6)$ to compute $y(n+1)$ after another $8T$ units of time. Thus, this implementation can take in a fresh input sample once every $8T$ units of time and generate an output sample at the same rate. In other words, the maximum sampling rate that this filter implementation can handle is $1/8T$.

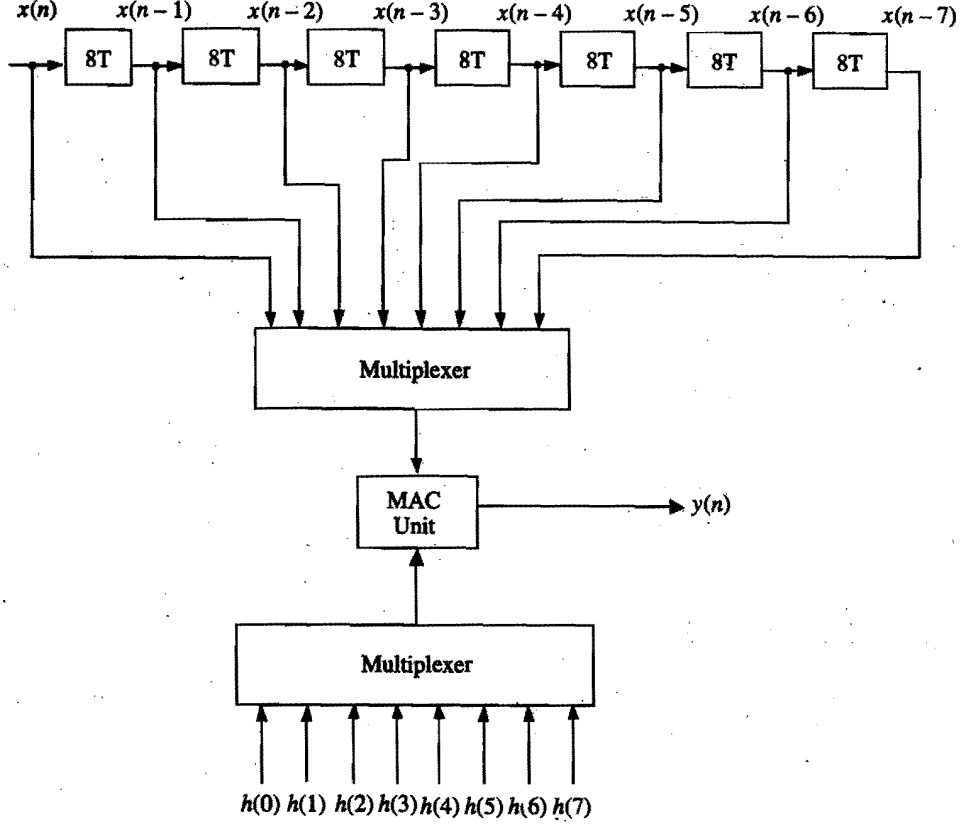


Figure 4.14(a) Single MAC implementation of an 8-tap FIR filter

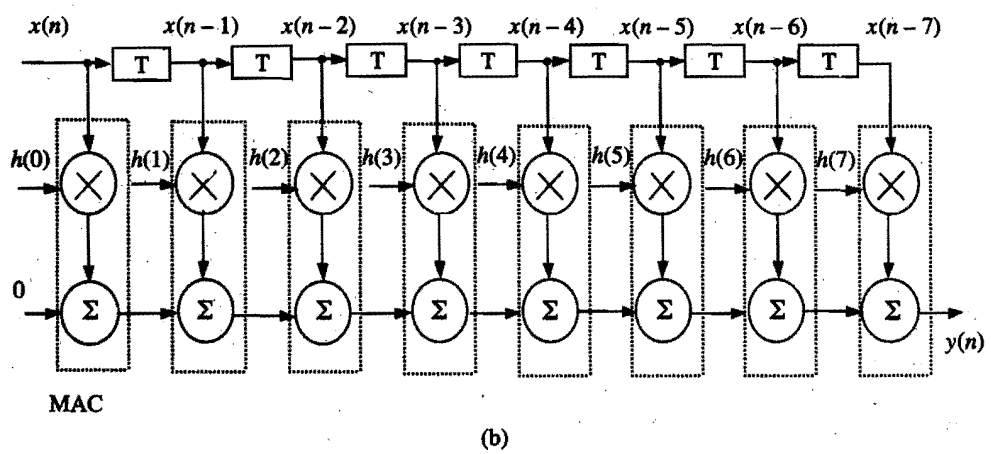


Figure 4.14(b) Pipelined implementation of an 8-tap FIR filter using eight MACs

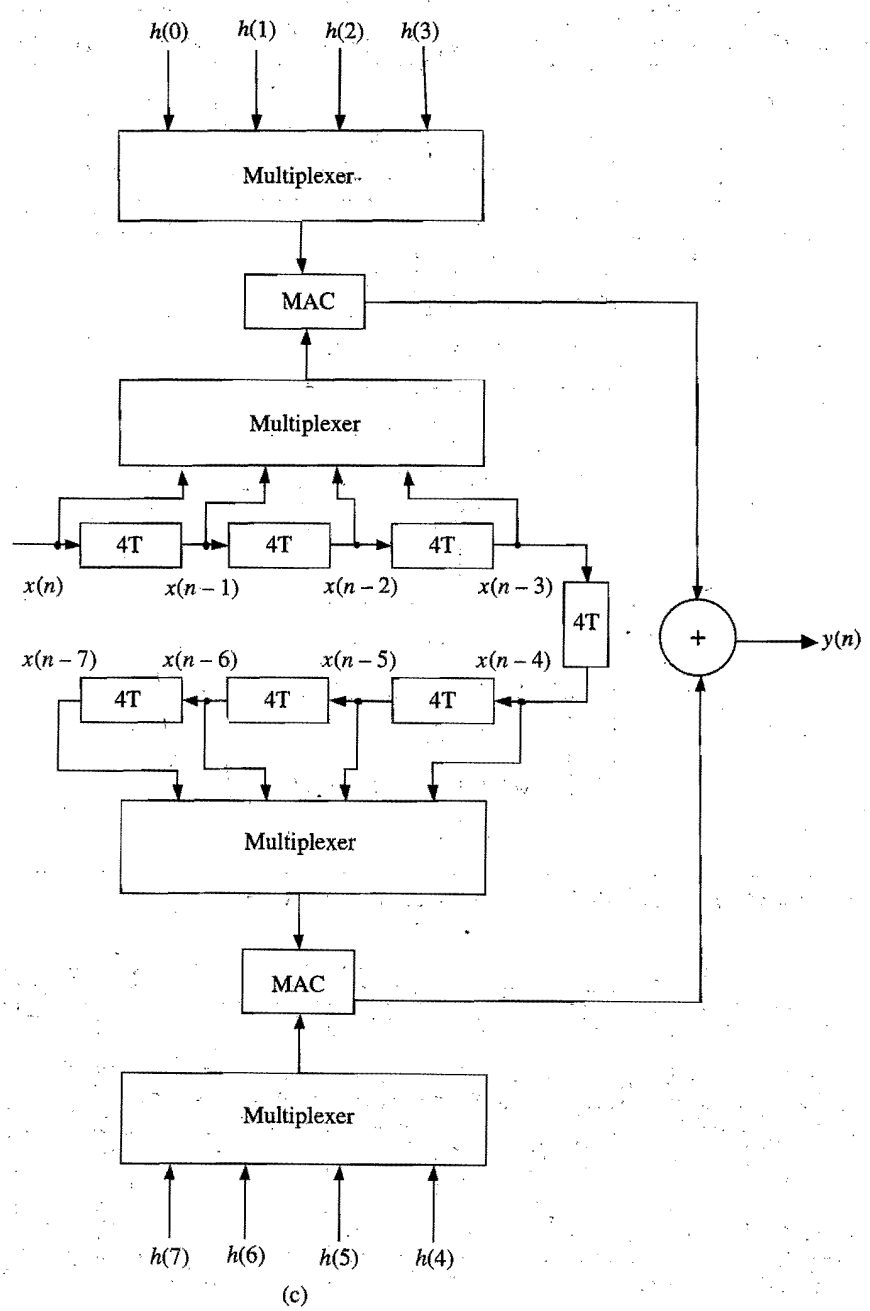


Figure 4.14(c) Parallel implementation of an 8-tap FIR filter using two MAC units

Pipelined Implementation Using Eight Multipliers and Eight Accumulators

The implementation of the FIR filter of Eq. 4.9 can be speeded up if more multipliers and accumulators are available. Let us assume that there are eight multipliers and eight accumulators connected in a pipelined structure, as shown in Figure 4.14(b). Each multiplier computes one product term and passes it on to the corresponding accumulator, which in turn adds it to the summation passed on from the previous accumulator. Since all the multipliers and accumulators work all the time, a new output sample is generated once every T units of time. This is the time required by the multiplier and accumulator to compute one product term and add it to the sum passed on from the previous stage of the pipeline. This implementation can take in a new input sample once every T units of time and generate an output sample at the same rate. In other words, this filter implementation works 8 times faster than the simple one MAC implementation.

Parallel Implementation Using Two MAC Units

A third implementation of the FIR filter of Eq. 4.9 is shown in Figure 4.14(c). This implementation uses two MAC units and an adder at the output. Each MAC computes four of the eight product terms in Eq. 4.9. Input samples and the filter coefficients are fed to the MACs using multiplexers that are controlled such that correct combinations of samples and the corresponding filter coefficients are fed to the two MACs at any given time. If T time units are required to compute one pair of products and add them to the previously accumulated sum in the MAC units, it will require $4T$ units of time to generate the final output by adding the outputs of the two MACs. At this time, a new input sample can be applied to the filter for computation of the next output sample. The speed of this implementation is 2 times that of one MAC implementation of Figure 4.14(a) and one fourth of that of the pipelined eight-multiplier, eight-accumulator implementation of Figure 4.14(b). The maximum rate at which input samples can be applied to this filter implementation is 2 times that of the first implementation and one fourth that of the second.

Table 4.6 Performance Summary of Different Implementations of an 8 -tap FIR Filter

Type of Implementation	Maximum Sample Rate	Maximum Throughput
One MAC	$1/8T$	One sample in $8T$ units of time
Pipelined (8 Multipliers and 8 Adders)	$1/T$	One sample in T units of time
Two MAC	$1/4T$	One sample in $4T$ units of time

T = MAC time

Table 4.6 summarizes the performance of the three implementations described above. The example shows that it is possible to achieve higher-speed implementation by the use of parallelism and/or pipelining. This, however, increases the hardware complexity.

4.9 Features for External Interfacing

It is important for a DSP device to be able to communicate with the outside world. The outside world provides the signal to be processed and receives the processed signal. Therefore, most of the peripherals used with conventional microprocessors are also needed in a DSP system. These peripherals include interfaces for interrupts, direct memory access, serial I/O, and parallel I/O. In addition, since DSP is a digital device that is expected to process analog signals, conversions from analog-to-digital and digital-to-analog representations need to be carried out outside the device. From signal interfacing viewpoint, a DSP device should be capable of handling commonly available serial and parallel signal converters. All these features require the availability of appropriate address, data, and control signals to set up interfaces with the peripherals. The inclusion of a timer in the architecture is also very desirable to implement events at regular intervals, such as periodically initiating an A/D converter to start the conversion. A timer should be able to interrupt the processor to get its attention when needed so that the data acquisition can go on in the background simultaneously with the execution of the signal-processing program.

4.10 Summary

In this chapter, architectural features of programmable DSP devices have been examined based on the most frequently used DSP operations. Computational building blocks and other functional units have been described along with examples of implementations. Bus architecture and memory organization are explained to show how they help in realizing fast implementations of DSP algorithms. Trade-off between complexity and speed has also been discussed to show how the architectural features of programmable DSP devices can be optimized for efficient implementations.

In summary, the following is a list of architectural features of a programmable DSP device that should be evaluated before implementing an algorithm:

- Data representation format: fixed-point, floating-point formats and data word length for accuracy and dynamic range.
- Computational capability: an ALU with a hardware multiplier and shifters for scaling.

- Harvard architecture: provision of separate memories for program and data to fetch instructions and data simultaneously.
- On-chip memories: provision of on-chip program and data memories to avoid bus contention and to speed up program execution.
- Addressing modes: data addressing capabilities including indirect, indexed, circular buffer, and bit-reversed addressing modes.
- Programmability: programming capabilities including subroutines, branching, loops and repeats.
- Hardwired control: fast implementation of sequencing and control for single-cycle instruction execution.
- Parallelism: multiple functional units for parallel implementation of different functions such as simultaneous execution of an arithmetic operation and an address computation.
- Pipelining: simultaneous operation of different stages of an instruction execution by splitting it into steps handled by individually designed units.
- Interfacing: provision to interface serial devices such as A/D and D/A converters, parallel I/O, interrupt, and direct memory access.

References

1. Allen, J. "Computer Architecture for Digital Signal Processing," *IEEE Proceedings*, Vol. 73, pp. 852–873, May 1985.
2. Lee, E. A. "Programmable DSP Architectures: Part I," *IEEE ASSP Magazine*, pp. 4–19, October 1988.
3. Lee, E. A. "Programmable DSP Architectures: Part II," *IEEE ASSP Magazine*, pp. 4–14, October 1989.
4. Kung, S. Y. *VLSI Array Processors*, Englewood Cliffs, NJ, Prentice Hall, 1988.
5. Higgin, R. J. *Digital Signal Processing in VLSI*, Englewood Cliffs, NJ, Prentice Hall, 1990.
6. Kung, S. Y., Whitehouse, H. T., and Kailath, T. *VLSI and Modern Signal Processing*, Englewood Cliffs, NJ, Prentice Hall, 1985.
7. Braun, E. L. *Digital Computer Design*, New York, Academic Press, 1963.
8. Baugh, C. R., and Wooley, B. A. "A 2's Complement Parallel, Array Multiplication Algorithm," *IEEE Trans. Computers*, Vol. C-22, pp. 1045–1047, December 1973.
9. Lapsley, P., Bier, J., Shoham, A., and Lee, E. A. *DSP Processor Fundamentals: Architectures and Features*, Piscataway, NJ, IEEE Press, 1997.

10. Eyre, J., and Bier, J. "DSP Processors Hit the Mainstream," *Computer*, pp. 51–59, August 1998.

11. Bates, A., and Paterson-Stephens, I. *The DSP Handbook: Algorithms, Applications and Design Techniques*, Englewood Cliffs, NJ, Prentice Hall, 2002.

Assignments

4.1 What distinguishes a digital signal processor from a general-purpose microprocessor with regard to basic capabilities?

4.2 Specify the basic architecture required to implement the following operations so that they can be executed in the least possible time:

a. $(x_1 + jy_1)(x_2 + jy_2)$

b. $(0.5x_1 + 4x_2)/256$

4.3 Draw a structure similar to that of Figure 4.1(b) for an 8×8 unsigned binary multiplier.

4.4 How will you implement an 8×8 multiplier using 4×4 multipliers as the building blocks?

4.5 Suggest a scheme to implement a multiplier to multiply two complex numbers using the multiplier shown in Figure 4.1(b) as the building block.

4.6 Draw a structure based on Eq. 4.7 to multiply two 4-bit signed numbers, A and B.

4.7 a. Assuming the availability of a single 16-bit data bus, how many memory accesses will be required to access two 16-bit operands from the memory, multiply them, and save the 32-bit product back in the memory?

b. Suggest a suitable hardware scheme to implement the multiplication specified in part (a).

4.8 Figure 4.3(b) shows the structure of a 4-bit barrel shifter. The switches shown connect each input bit to one of the output lines, depending on the number of bits to be shifted. Suggest a suitable hardware scheme for the switches and redraw Figure 4.3(b) by replacing the switches with its hardware. Also show how the control inputs control the switches to achieve the desired shift.

4.9 What should be the minimum width of the accumulator in a DSP device that receives 10-bit A/D samples and is required to add 64 of them without causing an overflow?

4.10 a. What is meant by overflow in an arithmetic computation? How is an overflow condition detected in an ALU?

b. By means of numerical examples using 8-bit, 2's complement numbers, illustrate the conditions of (i) no overflow, (ii) overflow, (iii) no underflow, and (iv) underflow resulting from arithmetic operations in an ALU. In each case, verify if the circuit of Figure 4.6 can detect the condition.

4.11 Suggest the memory architecture required for a DSP device to implement each of the following algorithms:

a. N-tap FIR filter

b. 2^M-point FFT

c. autocorrelation of a segment of N samples

d. crosscorrelation of two sequences of N samples each

4.12 Figure 4.8(c) allows for an instruction and two operands to be fetched simultaneously from the memory to the DSP to execute a multiply instruction in a single cycle. However, to save the result in memory, one more memory access is required. Can you specify an architecture that allows the result to be written back to the memory in the same cycle?

4.13 Identify the addressing modes of the operands in each of the following instructions (AR stands for address register):

ADD #1234h

ADD 1234h

ADD *AR+

ADD offsetaddr–, *AR

4.14 What is the bit-reversed sequence of 32 samples $x_0, x_1, x_2, \ldots, x_{31}$ as obtained by sampling a signal?

4.15 Table 4.4 shows how bit reversing is done for 8 points. A similar algorithm can be used for any 2^n points. Specify using a block diagram how it can be implemented in hardware.

4.16 How will you organize samples and filter coefficients using a circular buffer addressing scheme to implement a 32-tap FIR filter given by

$$y(n) = \sum_{k=0}^{31} b_k x(n-k)$$

4.17 When a two-dimensional array of data such as a matrix is organized in a memory with linear (or one-dimensional) addressing, it is usually arranged in a row-ordered format. That is, all the elements of the first row are placed first in successive memory locations, starting with the very first location. This is followed by the elements of the second row, and so on, until all the elements of all the rows are arranged. Write a pseudocode to compute the address of any given element of this matrix, say, the element (i, j), assuming that there are N rows and M columns in the matrix.

4.18 Suggest a hardware architecture for the addressing unit that computes the two-dimensional address described in Problem 4.17 without the overhead required for computing it in software.

4.19 Given below is the pseudocode of a software loop normally used in a general-purpose microprocessor for repetitive execution of an arithmetic operation.

Modify the code for a DSP with zero-overhead looping hardware:

```
Load count register
    Back:  Get operands; Compute; Update pointers
           Decrement Count
           If Count is not zero then jump Back
           Proceed
```

4.20 Explain the difference between a single-instruction, zero-overhead hardware looping and multiple-instruction, zero-overhead hardware looping in terms of architectural requirements and the performance.

4.21 What is the difference between a microcoded program control and a hard-wired program control? Why is the latter preferred for DSP implementations?

4.22 List the major architectural features used in a digital signal processor to achieve high speed of program execution.

4.23 What architectural features are required in a DSP device to implement an FIR filter with N taps so that a steady-state throughput of one output sample per cycle is achieved?

4.24 List the essential peripherals required to implement the following DSP systems:

A speech processing system

A biomedical instrumentation system

An image processing system

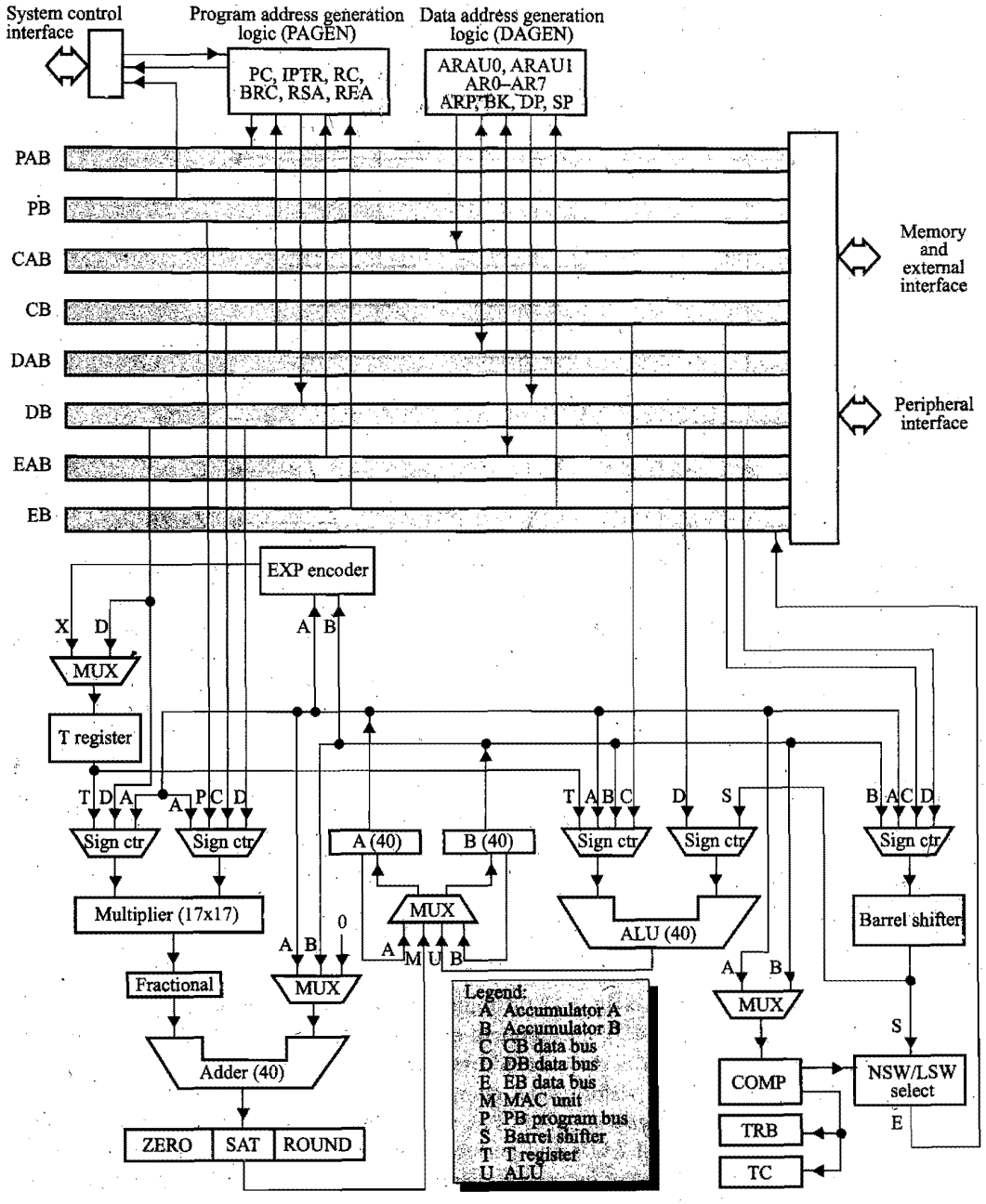


Figure 5.4 Functional architecture for TMS320C54xx processors

(Courtesy of Texas Instruments)

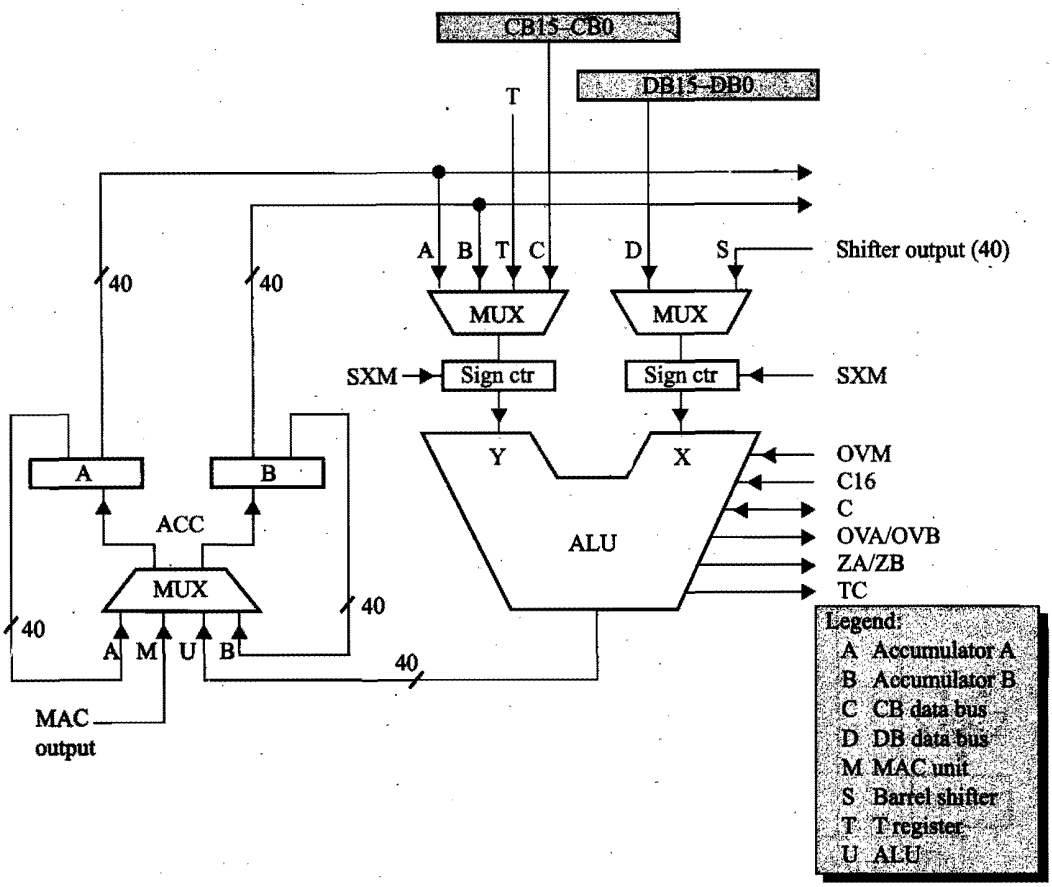


Figure 5.5 Functional diagram of the central processing unit of the TMS320C54xx processors (Courtesy of Texas Instruments Inc.)

barrel shifter; a 17 × 17-bit multiplier; a 40-bit adder; a compare, select and store unit (CSSU); an exponent encoder (EXP); a data address generation unit (DAGEN); and a program address generation unit (PAGEN).

The ALU performs 2's complement arithmetic operations and bit-level Boolean operations on 16-, 32-, and 40-bit words. It can also function as two separate 16-bit ALUs and perform two 16-bit operations simultaneously. Figure 5.5 shows the functional diagram of the ALU of the TMS320C54xx family of devices.

Accumulators A and B store the output from the ALU or the multiplier/adder block and provide a second input to the ALU. Each accumulator is divided into three parts: guard bits (bits 39–32), high-order word (bits 31–

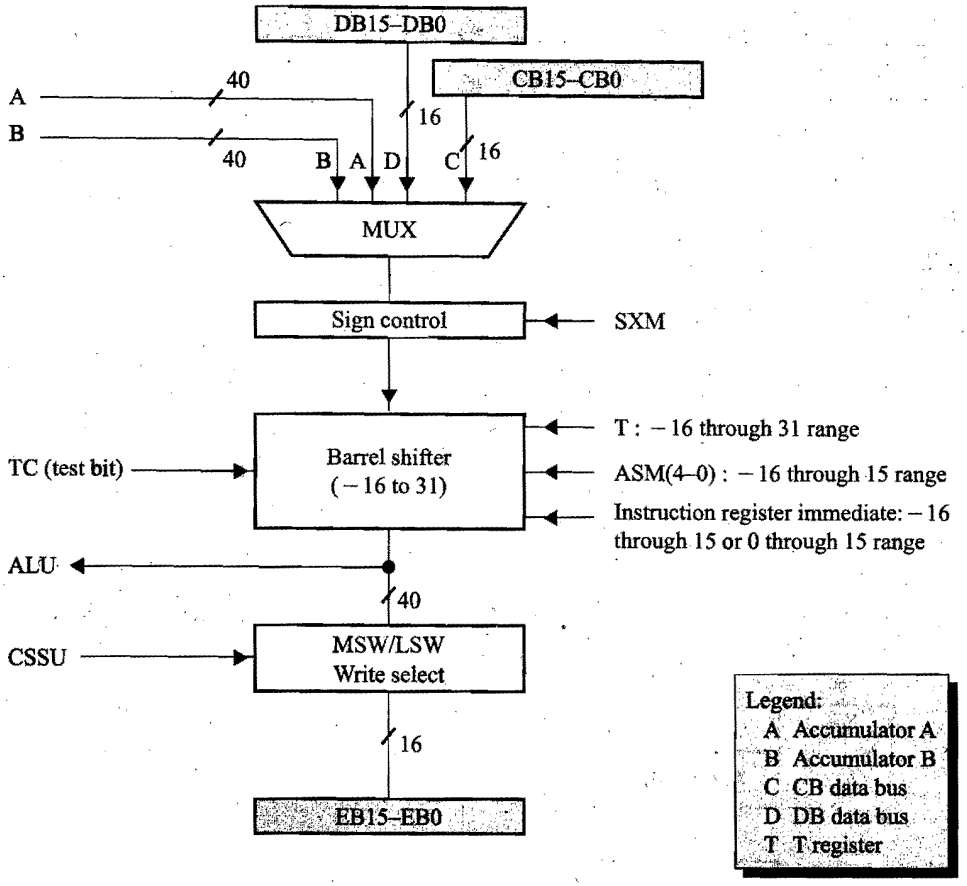


Figure 5.6 Functional diagram of the barrel shifter of the TMS320C54xx processors (Courtesy of Texas Instruments Inc.)

16), and low-order word (bits 15–0), which can be stored and retrieved individually.

The barrel shifter provides the capability to scale the data during an operand read or write. No overhead is required to implement the shift needed for the scaling operations. The '54xx barrel shifter can produce a left shift of 0 to 31 bits or a right shift of 0 to 16 bits on the input data. The shift requirements are defined in the shift count field of the instruction, the shift count field of status register ST1, or in the temporary register T. Figure 5.6 shows the functional diagram of the barrel shifter of TMS320C54xx processors.

The barrel shifter and the exponent encoder normalize the values in an accumulator in a single cycle. The LSBs of the output are filled with 0s, and

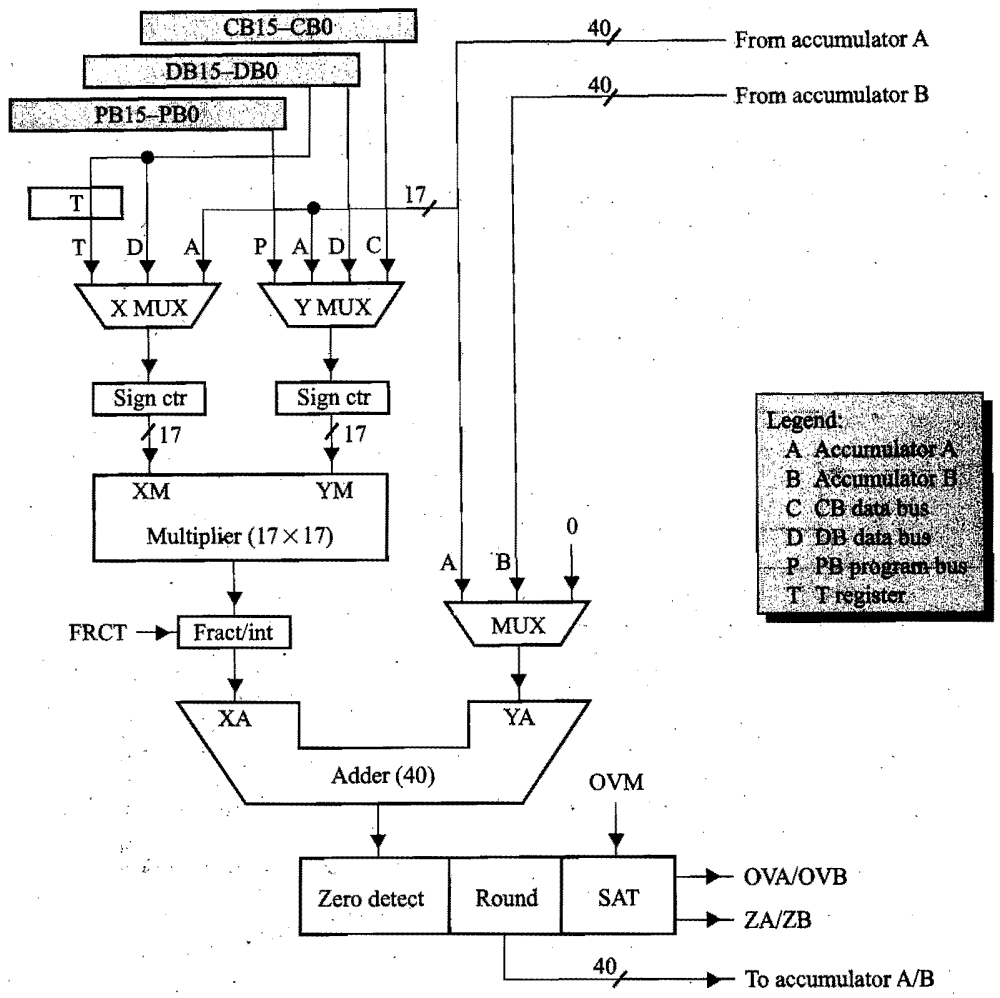


Figure 5.7 Functional diagram of the multiplier/adder unit of TMS320C54xx processors (Courtesy of Texas Instruments Inc.).

the MSBs can be either zero filled or sign extended, depending on the state of the sign-extension mode bit in the status register ST1. Additional shift capabilities enable the processor to perform numerical scaling, bit extraction, extended arithmetic, and overflow prevention operations.

The kernel of the DSP device architecture is its multiplier/adder unit. The multiplier/adder unit of TMS320C54xx devices performs 17×17 2's-complement multiplication with a 40-bit addition effectively in a single instruction cycle. In addition to the multiplier and adder, the unit consists of control

logic for integer and fractional computations and a 16-bit temporary storage register, T. Figure 5.7 shows the functional diagram of the multiplier/adder unit of TMS320C54xx processors.

The compare, select, and store unit (CSSU) is a hardware unit specifically incorporated to accelerate the add/compare/select operation. This operation is essential to implement the *Viterbi* algorithm used in many signal-processing applications.

The exponent encoder unit supports the EXP instruction, which stores in the T register the number of leading redundant bits of the accumulator content. This information is useful while shifting the accumulator content for the purpose of scaling.

5.3.3 Internal Memory and Memory-Mapped Registers

The amount and the types of memory of a processor have direct relevance to the efficiency and the performance obtainable in implementations with the processor. The '54xx memory is organized into three individually selectable spaces: program, data, and I/O spaces. All '54xx devices contain both RAM and ROM. RAM can be either dual-access type (DARAM) or single-access type (SARAM). The on-chip RAM for these processors is organized in pages having 128 word locations on each page.

The '54xx processors have a number of CPU registers to support operand addressing and computations. The CPU registers and peripheral registers are all located on page 0 of the data memory. Figures 5.8(a) and (b) show the internal CPU registers and peripheral registers with their addresses. Figure 5.8(c) shows the processor mode status (PMST) register that is used to configure the processor. It is a memory-mapped register located at address 1Dh on page 0 of the RAM. The peripheral registers are covered in subsequent chapters.

A part of on-chip ROM may contain a bootloader and look-up tables for functions such as sine, cosine, μ-law, and A-law. Details of the memory space of TMS320C54xx processors are discussed in Section 5.5.

5.4 Data Addressing Modes of TMS320C54xx Processors

Data addressing modes provide various ways to access operands to execute instructions and place results in the memory or the registers. The '54xx devices offer seven basic addressing modes: immediate addressing, absolute addressing, accumulator addressing, direct addressing, indirect addressing, memory-mapped register addressing, and stack addressing.

NAME	ADDRESS DEC	ADDRESS HEX	DESCRIPTION
IMR	0	0	Interrupt mask register
IFR	1	1	Interrupt flag register
—	2–5	2–5	Reserved for testing
ST0	6	6	Status register 0
ST1	7	7	Status register 1
AL	8	8	Accumulator A low word (15–0)
AH	9	9	Accumulator A high word (31–16)
AG	10	A	Accumulator A guard bits (39–32)
BL	11	B	Accumulator B low word (15–0)
BH	12	C	Accumulator B high word (31–16)
BG	13	D	Accumulator B guard bits (39–32)
TREG	14	E	Temporary register
TRN	15	F	Transition register
AR0	16	10	Auxiliary register 0
AR1	17	11	Auxiliary register 1
AR2	18	12	Auxiliary register 2
AR3	19	13	Auxiliary register 3
AR4	20	14	Auxiliary register 4
AR5	21	15	Auxiliary register 5
AR6	22	16	Auxiliary register 6
AR7	23	17	Auxiliary register 7
SP	24	18	Stack pointer register
BK	25	19	Circular buffer size register
BRC	26	1A	Block repeat counter
RSA	27	1B	Block repeat start address
REA	28	1C	Block repeat end address
PMST	29	1D	Processor mode status (PMST) register
XPC	30	1E	Extended program page register
—	31	1F	Reserved

(a)

Figure 5.8(a) Internal memory-mapped registers of TMS320C54xx signal processors (Courtesy of Texas Instruments Inc.)

5.4.1 Immediate Addressing

In this mode, the instruction contains the specific value of the operand. The operand can be short (3, 5, 8, or 9 bits in length) or long (16 bits in length). The instruction syntax for short operands occupies one memory location,

NAME	ADDRESS DEC	ADDRESS HEX	DESCRIPTION
DRR20	32	20	McBSP 0 Data Receive Register 2
DRR10	33	21	McBSP 0 Data Receive Register 1
DXR20	34	22	McBSP 0 Data Transmit Register 2
DXR10	35	23	McBSP 0 Data Transmit Register 1
TIM	36	24	Timer Register
PRD	37	25	Timer Period Register
TCR	38	26	Timer Control Register
—	39	27	Reserved
SWWSR	40	28	Software Watt-State Register
BSCR	41	29	Bank-Switching Control Register
—	42	2A	Reserved
SWCR	43	2B	Software Watt-State Control Register
HPIC	44	2C	HPI Control Register (HMODE = 0 only)
—	45–47	2D–2F	Reserved
DRR22	48	30	McBSP 2 Data Receive Register 2
DRR12	49	31	McBSP 2 Data Receive Register 1
DXR22	50	32	McBSP 2 Data Transmit Register 2
DXR12	51	33	McBSP 2 Data Transmit Register 1
SPSA2	52	34	McBSP 2 Subbank Address Register
SPSD2	53	35	McBSP 2 Subbank Data Register
—	54–55	36–37	Reserved
SPSA0	56	38	McBSP 0 Subbank Address Register
SPSD0	57	39	McBSP 0 Subbank Data Register
—	58–59	3A–3B	Reserved
GPIOCR	60	3C	General-Purpose I/O Control Register
GPIOSR	61	3D	General-Purpose I/O Status Register
CSIDR	62	3E	Device ID Register
—	63	3F	Reserved
DRR21	64	40	McBSP 1 Data Receive Register 2
DRR11	65	41	McBSP 1 Data Receive Register 1
DXR21	66	42	McBSP 1 Data Transmit Register 2
DXR11	67	43	McBSP 1 Data Transmit Register 1
—	68–71	44–47	Reserved
SPSA1	72	48	McBSP 1 Subbank Address Register
SPSD1	73	49	McBSP 1 Subbank Data Register
—	74–83	4A–53	Reserved
DMPREC	84	54	DMA Priority and Enable Control Register
DMSA	85	55	DMA Subbank Address Register

Figure 5.8(b) Peripheral registers for the TMS320C5416 processor

(Courtesy of Texas Instruments Inc.)

(continued)

DMSDI	86	56	DMA Subbank Data Register with Autoincrement‡
DMSDN	87	57	DMA Subbank Data Register
CLKMD	88	58	Clock Mode Register (CLKMD)
—	89–95	59–5F	Reserved

(b)

Figure 5.8(b) Continued

15–7	6	5	4	3	2	1	0
IPTR	MP/$\overline{\text{MC}}$	OVLY	AVIS	DROM	CLKOFF†	SMUL†	SST†

†These bits are only supported on C54x devices with revision A or later, or on C54x devices numbered C548 or greater.

(c)

Figure 5.8(c) Processor mode status (PMST) register of TMS320C54xx processors (Courtesy of Texas Instruments Inc.)

whereas that for long operands occupies two memory locations. This addressing mode can be used to initialize registers and memory locations. Examples of instructions using this addressing mode are

```
LD #20, DP   ; This accomplishes #20 → DP
RPT #0FFFFh  ; This accomplishes #FFFFh → RC
```

5.4.2 Absolute Addressing

In this mode, the instruction contains a specific address. The specified address may be for a data memory location (*dmad* addressing), a program memory location (*pmad* addressing), a port address (*PA* addressing), or a location in the data space specified directly (**(lk)* addressing). Examples of instructions using this mode of addressing are

```
MVKD 1000h, *AR5  ; 1000h → AR5 (dmad addressing)
MVPD 1000h, *AR7  ; 1000h → *AR7 (pmad addressing)
PORTR 05h, *AR3   ; 05h → *AR3 (PA addressing)
LD *(1000h), A    ; *(1000h) → A (*(lk) addressing)
```

5.4.3 Accumulator Addressing

This mode uses the accumulator contents as the address and is used to move data between a program memory location and a data memory location. Ex-

amples of instructions in this mode are READA and WRITA. READA transfers a word from a program-memory location specified by accumulator A to a data-memory location. WRITA transfers a word from a data-memory location to a program-memory location specified by accumulator A.

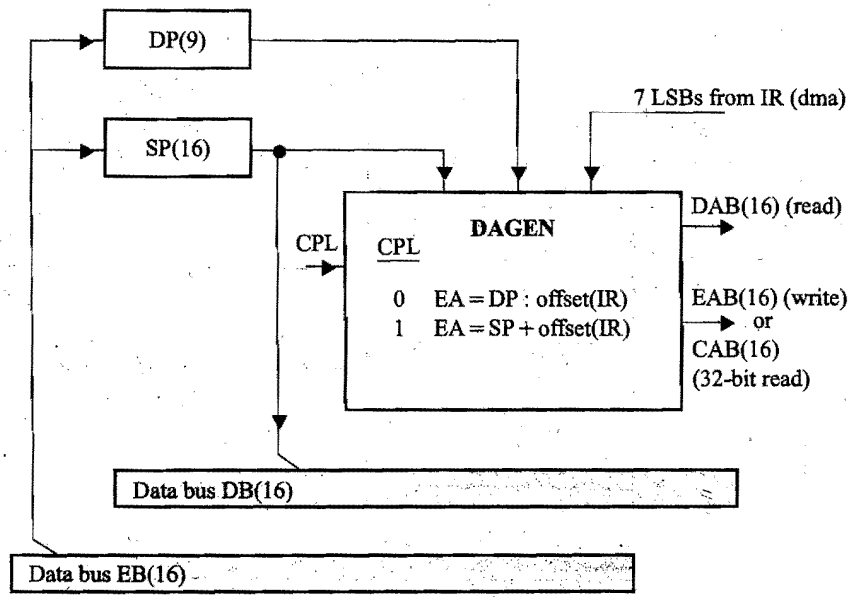
Here is an example:

```
READA *AR2  ; This accomplishes *A → *AR2
```

5.4.4 Direct Addressing

In the direct addressing mode, the 16-bit address of the data-memory location is formed by combining the lower 7 bits of the data-memory address contained in the instruction with a base address given by the data-page pointer (DP) or the stack pointer (SP). Figure 5.9 shows the operation of the direct addressing mode of TMS320C54xx processors.

Using this form of addressing, one can access a page of 128 contiguous locations without changing the DP or the SP. The compiler mode bit (CPL), located in the status register ST1, is used to select between the two pointers



Legend: EA Effective address
IR Instruction register

Figure 5.9 Block diagram of the direct addressing mode for TMS320C54xx processors (Courtesy of Texas Instruments Inc.)

used to generate the address. CPL = 0 selects DP and CPL = 1 selects SP. For example, when CPL = 0, to add the contents of the memory location 0 on page 4 in the data memory to accumulator B, we can use the instruction sequence:

```
LD #4, DP  ; DP = 4 = upper 9 bits of address
ADD=0, B   ; Lower 7 bits of the address
```

With this example the contents of the first locations on data page 4 (memory address 0200h) are added to accumulator B.

It should be remembered that when SP is used instead of DP, the effective address is computed by adding the 7-bit offset to SP.

5.4.5 Indirect Addressing

In indirect addressing, any location in the data space can be accessed by means of an address contained in an auxiliary register. The '54xx devices have eight 16-bit auxiliary registers (AR0–AR7). Indirect addressing is used when

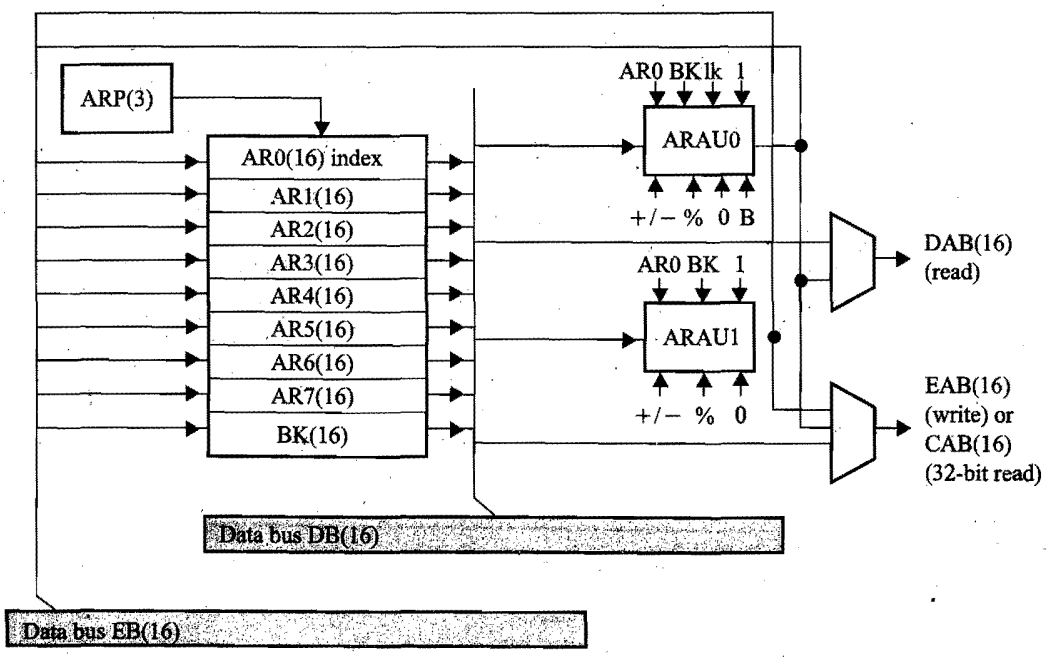


Figure 5.10 Block diagram for the indirect addressing mode of TMS320C54xx processors

(Courtesy of Texas Instruments Inc.)

there is a need to step through a sequence of locations in the memory in fixed-sized steps.

Two auxiliary register arithmetic units (ARAU0 and ARAU1) are used to modify the contents of the auxiliary registers for the indirect addressing mode. They perform unsigned, 16-bit arithmetic operations. The auxiliary registers can be loaded with an immediate value, loaded via the data bus, and modified by the indirect addressing field of any instruction that supports indirect addressing or by the modify auxiliary register (MAR) instruction and used as loop counters.

Figure 5.10 shows how ARAUs are used to generate an address in the indirect addressing mode using a single data-memory operand. An address can be modified before or after accessing the location or can be left unchanged. Modification can be by incrementing or decrementing the address by 1, adding a 16-bit offset, or indexing with the value in AR0. Each of these modifications may be carried out either before or after accessing the memory location. Table 5.2 gives the operand syntax and the corresponding ARAU operations for the single operand indirect addressing mode.

▷ **Example 5.1** Assuming the current contents of AR3 to be 200h, what will be its contents after each of the following TMS320C54xx addressing modes is used? Assume that the contents of AR0 are 20h.

a. *AR3 + 0

b. *AR3 − 0

c. *AR3+

d. *AR3−

e. *AR3

f. *+AR3(40h)

g. *+AR3(−40h)

Solution

a. AR3 ← AR3 + AR0;
AR3 = 200h + 20h = 220h.

b. AR3 ← AR3 − AR0;
AR3 = 200h − 20h = 1E0h.

c. AR3 ← AR3 + 1;
AR3 = 200h + 1 = 201h.

d. AR3 ← AR3 − 1;
AR3 = 200h − 1 = 1FFh.

e. AR3 is not modified.
AR3 = 200.

f. AR3 ← AR3 + 40h;
AR3 = 200h + 40h = 240h.

g. AR3 ← AR3 − 40h;
AR3 = 200h − 40h = 1C0h.

Table 5.2 Indirect Addressing Options with a Single Data-Memory Operand

Operand Syntax	Operation
*ARx	addr ← ARx
*ARx+	addr ← ARx
	ARx ← ARx + 1
*ARx−	addr ← ARx
	ARx ← ARx − 1
*+ARx	ARx ← ARx + 1
	addr ← ARx
*ARx + 0	addr ← ARx
	ARx ← ARx + AR0
*ARx − 0	addr ← ARx
	ARx ← ARx − AR0
*ARx + 0B	addr ← ARx
	ARx ← B(ARx + AR0)
*ARx − 0B	addr ← ARx
	ARx ← B(ARx − AR0)
*ARx + %	addr ← ARx
	ARx ← circ(ARx + 1)
*ARx − %	addr ← ARx
	ARx ← circ(ARx − 1)
*ARx + 0%	addr ← ARx
	ARx ← circ(ARx + AR0)
*AR0 − 0%	addr ← ARx
	ARx ← circ(ARx − AR0)
*(lk)	addr ← lk
*ARx(lk)	addr ← ARx + lk
*+ARx(lk)	ARx ← ARx + lk
	addr ← ARx
*+ARx(lk)%	ARx ← circ(ARx + lk)
	addr ← ARx

Circular Addressing

Many fast real-time algorithms, such as convolution, correlation, and FIR filters, require the implementation of a circular buffer in memory. A circular

buffer is a sliding window containing the most recent data. As new data come in, the buffer overwrites the oldest data. An indirect addressing mode with circular address modification allows implementation of circular buffers.

The circular-buffer size register (BK) specifies the size of the circular buffer. A circular buffer must start on an N-bit boundary; that is, the N LSBs of the base address of the circular buffer must be 0. For example, a 31-word circular buffer must start at an address whose five LSBs are 0 and the value 30 must be loaded into BK. Similarly, a 48-word circular buffer must start at an address whose six LSBs are 0 and the value 47 must be loaded into BK.

The algorithm for circular addressing works as follows:

```
If 0 ≤ index + step < BK:  index = index + step;
else if index + step ≥ BK:  index = index + step - BK;
else if index + step < 0:  index = index + step + BK.
```

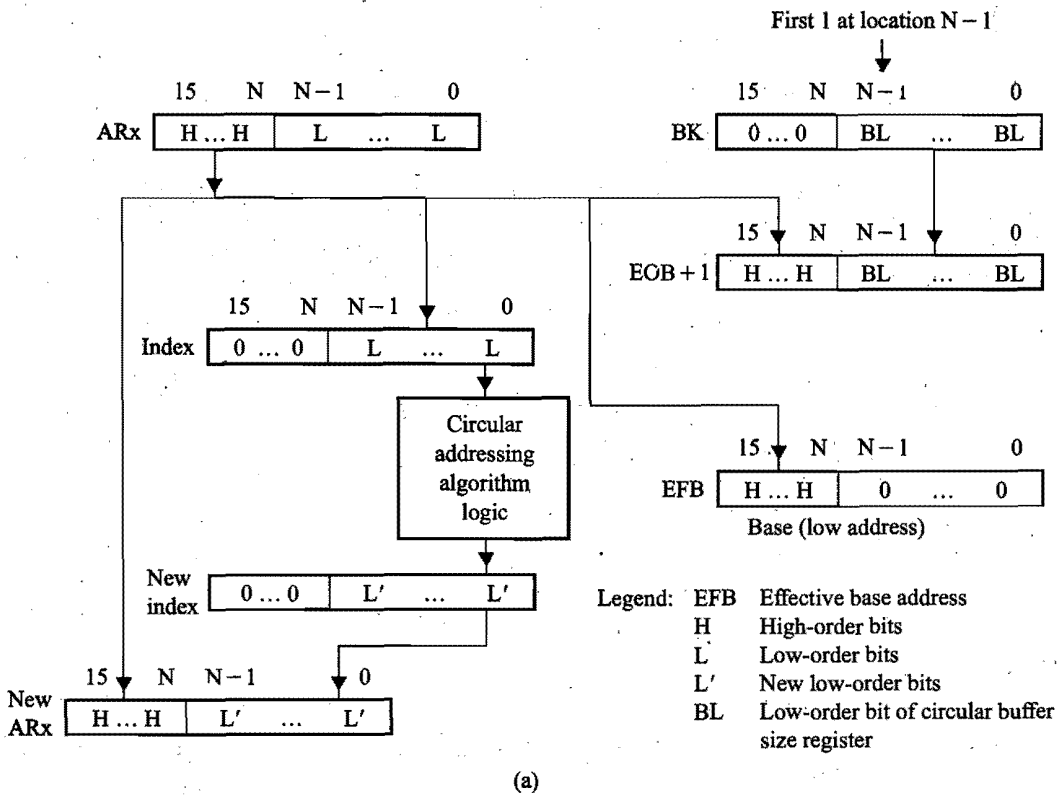


Figure 5.11(a) Block diagram of the circular addressing mode for TMS320C54xx processors

(Courtesy of Texas Instruments Inc.)

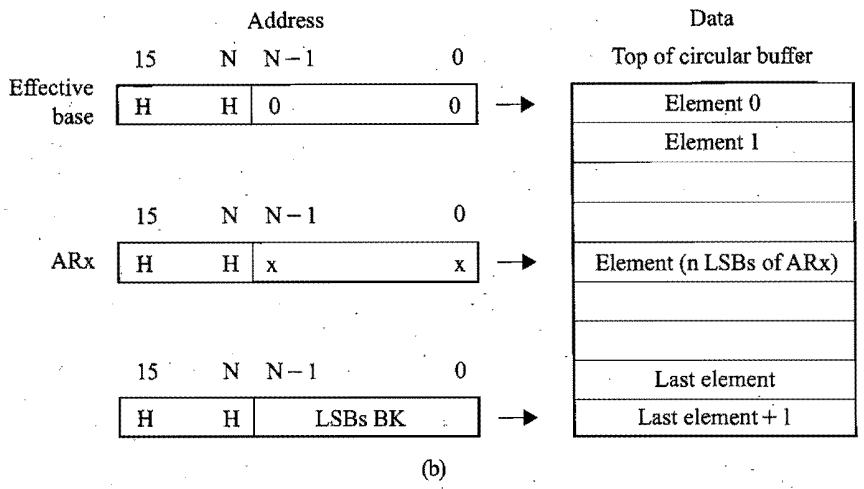


Figure 5.11(b) Circular addressing mode implementation in TMS320C54xx processors (Courtesy of Texas Instruments Inc.)

Figure 5.11(a) illustrates the relationships between BK, the auxiliary register ARx (the pointer), the bottom of the circular buffer, the top of the circular buffer, and the index into the circular buffer. Figure 5.11(b) shows how the circular buffer is implemented and illustrates the relationship between the generated values and the elements in the circular buffer.

▷ **Example 5.2** Assume that the register AR3 with contents 1020h is selected as the pointer for the circular buffer. Let BK = 40h to specify the circular buffer size as 41h. Determine the start and the end addresses for the buffer. What will be the contents of register AR3 after the execution of the instruction LD *AR3 + 0%, A, if the contents of register AR0 are 0025h?

Solution AR3 = 1020h means that currently it points to location 1020h. Making the lower 6 bits zeros gives the start address of the buffer as 1000h. Replacing the same bits with the BK gives the end address as 1040h.

The instruction

LD *AR3 + 0%, A

modifies AR3 by adding AR0 to it and applying the circular modification. It yields

AR3 = circ(1020h + 0025h) = circ(1045h) = 1045h − 40h = 1005h.

Thus the location 1005h is the one pointed to by AR3.

Bit-Reversed Addressing

Bit-reversed addressing is used in FFT algorithms. In this addressing mode, AR0 specifies one half of the size of the FFT. An auxiliary register points to the physical location of a data value. The address of the next location is generated by adding, in a bit-reversed manner, AR0 and the other specified auxiliary register. In the bit-reversed addition, the carry bit propagates from left to right, instead of right to left as in the regular add.

▷ **Example 5.3** Assuming the current contents of AR3 to be 200h, what will be its contents after each of the following TMS320C54xx addressing modes is used? Assume that the contents of AR0 are 20h.

a. *AR3 + 0B

b. *AR3 − 0B

Solution a. AR3 ← AR3 + AR0 with reverse carry propagation;
AR3 = 200h + 20h (with reverse carry propagation) = 220h.

b. AR3 ← AR3 − AR0 with reverse carry propagation;
AR3 = 200h − 20h (with reverse carry propagation) = 23Fh.

Dual-Operand Addressing

Dual data-memory operand addressing is used for instructions that simultaneously perform two reads (32-bit read) or a single read (16-bit read) and a parallel store (16-bit store) indicated by two vertical bars, ||. These instructions access operands using indirect addressing mode.

If in an instruction with a parallel store the source operand and the destination operand point to the same location, the source is read before writing to the destination. Only 2 bits are available in the instruction code for selecting each auxiliary register in this mode. Thus, just four of the auxiliary registers, AR2–AR5, can be used, The ARAUs, together with these registers, provide the capability to access two operands in a single cycle. Figure 5.12 shows how an address is generated using dual data-memory operand addressing.

5.4.6 Memory-Mapped Register Addressing

Memory-mapped register addressing is used to access the memory-mapped registers without affecting either the current data-page pointer (DP) value or the current stack-pointer (SP) value. This mode works for both direct and indirect addressing. Taking only the seven least significant bits of the 16-bit direct address or the value of the auxiliary register used for indirect addressing, the required address is generated.

For example, if AR1 is used indirectly to point to a memory-mapped register using the memory-mapped register addressing mode and its contents are

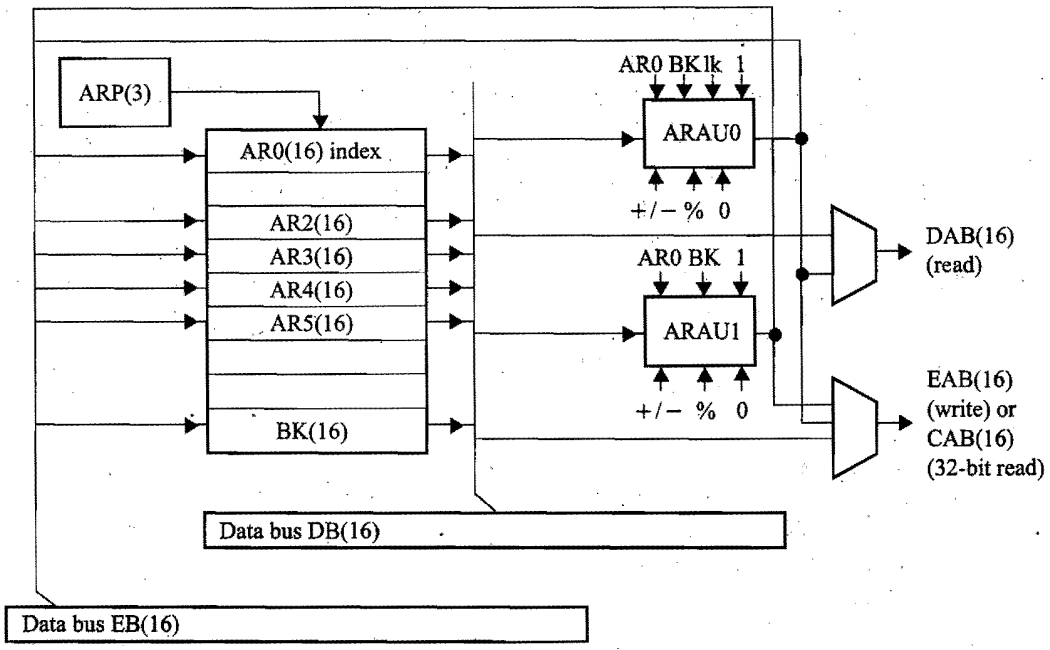


Figure 5.12 Block diagram of the indirect addressing mode of TMS320C54xx processors using dual memory operands

(Courtesy of Texas Instruments Inc.)

3825h, then AR1 points to the timer period register (PRD), since the seven LSBs of AR1 are 25h, which is the address of the PRD register. After execution, AR1 contains 0025h.

Consider the following instruction as another example:

```
LDM AR4, A
```

In this case the data stored at 0014h, which is the memory address of AR4, is loaded onto A.

5.4.7 Stack Addressing

The stack is used to store the return address during the servicing of interrupts and invoking of subroutines. It can also be used to pass parameters to subroutines during program execution. The stack is filled from the highest to the lowest memory address and emptied from the lowest to the highest address.

A 16-bit stack pointer (SP) is used to address the stack location at a given instance. SP points to the last element stored onto the stack. Instructions that access the stack for saving and recovering data on the stack consist of PUSHD, PUSHM, POPD, and POPM.

5.5 Memory Space of TMS320C54xx Processors

TMS320C54xx processors provide for a total of 128K words of memory extendable up to 8192K words. This includes both program memory and data memory. Within this space, RAM (both single access and dual access), ROM, EPROM, EEPROM, or memory-mapped peripherals may reside either on- or off-chip. The program memory space is used to store program instructions and the tables used in the execution of programs. The data-memory space is used to store data required to run programs and for external memory-mapped peripherals. Figures 5.13(a) and (b) show memory maps for the basic and extended memories of the TMS320C5416 processor.

The size of the data memory is 64K words, part of which is on-chip DARAM. The device automatically accesses the on-chip RAM when the address is within its range. Memory-mapped registers are also part of the data-memory space.

The program memory is organized into 128 pages, each of 64K word size. Page 0 is part of the basic 128K space, and pages 1 to 127 are extended pages. Out of the 64K words on page 0, 4K words are on-chip ROM. The remaining space on page 0 as well as the extended space consist of DARAM and SARAM, both on-chip and off-chip, as shown in Figures 5.13(a) and (b). The 4K on-chip ROM space contains a GSM EFR speech codec table, a bootloader, μ-law and A-law expansion tables, a sine look-up table, and an interrupt vector table.

The MP/$\overline{\text{MC}}$, OVLY, and DROM bits located in the processor mode status register (PMST) are used to enable and disable on-chip memories in the program and data spaces. The functions of these bits are described in Table 5.3.

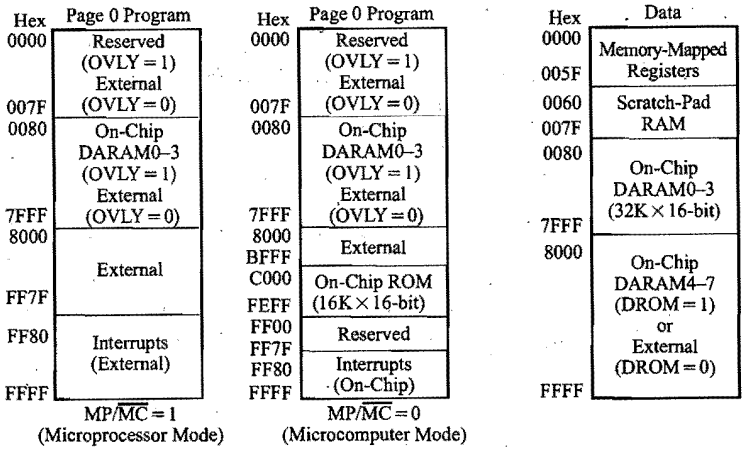
Example 5.4 What is the configuration of on-chip DARAM, on-chip SARAM, and ROM if MP/$\overline{\text{MC}}$ = 0, OVLY = 1, and DROM = 0 for TMS320C5416?

Solution

a. Since MP/$\overline{\text{MC}}$ = 0, 16K on-chip ROM is enabled as program memory at address c000h-feffh.

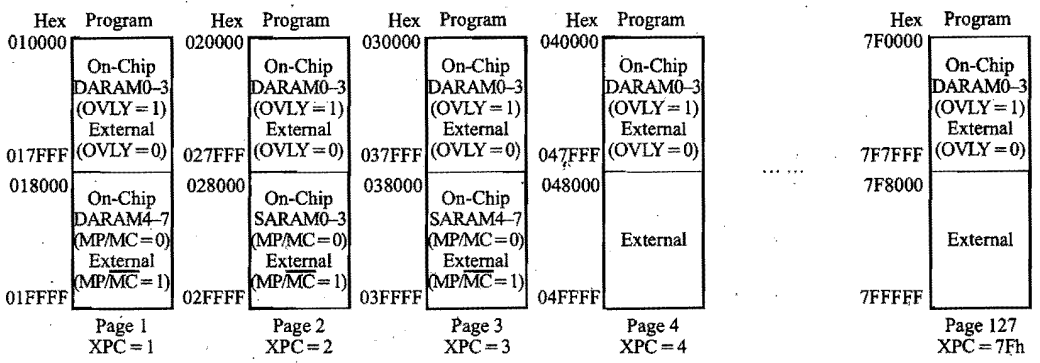
b. Since OVLY = 1, DARAM is mapped on to the program memory space at address 0080h-7fffh. Memory at addresses 000h-007fh is reserved for memory-mapped registers and the scratch pad purpose.

c. Since DROM = 0, ROM is not mapped on to the data memory.



Address ranges for on-chip DARAM in data memory are: DARAM0: 0080h–1FFFh; DARAM1: 2000h–3FFFh
DARAM2: 4000h–5FFFh; DARAM3: 6000h–7FFFh
DARAM4: 8000h–9FFFh; DARAM5: A000h–BFFFh
DARAM6: C000h–DFFFh; DARAM7: E000h–FFFFh

(a)



Address ranges for on-chip DARAM in program memory are: DARAM4: 018000h–019FFFh; DARAM5: 01A000h–01BFFFh
DARAM6: 01D0000h–01DFFFh; DARAM7: 01E000h–01FFFFh
Address ranges for on chip SARAM in program memory are: SARAM0: 028000h–029FFFh; SARAM1: 02A000h–02BFFFh
SARAM2: 02C000h–02DFFFh; SARAM3: 02E000h–02FFFFh
SARAM4: 038000h–039FFFh; SARAM5: 03A000h–03BFFFh
SARAM6: 03C000h–03DFFFh; SARAM7: 03E000h–03FFFFh

(b)

Figure 5.13 Memory map for the TMS320C5416 processor

(Courtesy of Texas Instruments Inc.)

Table 5.3 Processor Bits for Configuring the On-Chip Memories

PMST Bit	Logic	On-chip Memory Configuration
MP/$\overline{MC}$	0	ROM enabled
	1	ROM not available
OVLY	0	RAM in data space
	1	RAM in program space (except page 0)
DROM	0	ROM not in data space
	1	ROM in data space

▷ **Example 5.5** Repeat Example 5.4 if MP/$\overline{MC}$ = 1, OVLY = 1, and DROM = 1.

Solution

a. Since MP/$\overline{MC}$ = 1, TMS320C5416 is in microprocessor mode, the 16K ROM is off-chip in the program memory space.

b. Since OVLY = 1, DARAM is mapped on to the program memory space at address 0080h-7fffh. Memory at addresses 0000h-007fh is reserved for memory-mapped registers and the scratch pad purpose.

c. Since DROM = 1, 16K ROM is mapped on to the on-chip data memory at address c000h-feffh and memory from ff00h-ffffh is left for reserved purpose.

5.6 Program Control

The program control unit of TMS320C54xx processors contains the program counter (PC), the program counter-related hardware, hardware stack, repeat counters, and status registers. The PC addresses the program memory, either on-chip or off-chip, and is loaded in one of several ways, depending on the sequence of instructions being executed. These are

- Sequential: PC ← PC + 1.
- Branch: The PC is loaded with the immediate value following the branch instruction.
- Subroutine call: The PC is loaded with the immediate value following the call instruction.
- Interrupt: The PC is loaded with the address of the appropriate interrupt vector.
- Instructions such as BACC, CALA, etc.: The PC is loaded with the contents of the accumulator low word.

- End of a block repeat loop: The PC is loaded with the contents of the block repeat program address start register.
- Return: The PC is loaded from the top of the stack.

The program counter-related hardware PAGEN provides for the above options. The stack is used to save and restore the PC value during subroutine calls and interrupts. It can also be used to save and restore the accumulator low word or a data-memory value when required.

The TMS320C54xx processors provide hardware support for repetitive execution of either a single instruction or a block of instructions. Repeat counters are used for this purpose.

A single instruction can be repeated $N + 1$ times by loading the value N in the repeat counter register (RC). Likewise, a block of instructions can be repeated $N + 1$ times by loading the value N in the block repeat counter register (BRC).

5.7 TMS320C54xx Instructions and Programming

TMS320C54xx architecture supports an instruction set consisting of a large number of instructions [6]. Many of these are similar to the instructions for general-purpose microprocessors. However, the TMS320C54xx instruction set consists of a number of instructions that are specifically designed to carry out the numerically intensive signal-processing operations efficiently. In this section, we shall summarize the instruction set of the TMS320C54xx processors. In particular, we shall discuss those instructions that are frequently used to implement DSP algorithms and illustrate their use by means of sample programs.

5.7.1 Summary of the Instruction Set of TMS320C54xx Processors

TMS320C54xx assembly language instructions can be classified into the following categories based on their functions:

Load and Store Operations

- Load instructions; Examples: LD, LDM
- Store instructions; Examples: ST, STM
- Conditional store instructions; Examples: CMPS, STRCD
- Parallel load and store instructions; Example: ST||LD

- Parallel load and multiply instructions; Example: LD||MAC
- Parallel store and add/subtract instructions; Examples: ST||ADD, ST||SUB
- Parallel store and multiply instructions; Examples: ST||MPY, ST||MAC
- Miscellaneous load-type and store-type instructions; Examples: MVDD, MVPD

Arithmetic Operations

- Add instructions; Examples: ADD, ADDC
- Subtract instructions; Examples: SUB, SUBB
- Multiply instructions; Examples: MPY, MPYA
- Multiply–accumulate instructions; Examples: MAC, MACD
- Multiply–subtract instructions; Examples: MAS, MASA
- Double (32-bit operand) instructions; Examples: DADD, DSUB
- Application-specific instructions; Examples: EXP, LMS

Logical Operations

- AND instructions; Examples: AND, ANDM
- OR instructions; Examples: OR, ORM
- XOR instructions; Examples: XOR, XORM
- Shift instructions; Examples: ROL, SFTL
- Test instructions; Examples: BIT, CMPM

Program-Control Operations

- Branch instructions; Examples: B, BACC
- Call instructions; Examples: CALL, CALA
- Interrupt instructions; Examples: INTR, TRAP
- Return instructions; Examples: RET, FRET
- Repeat instructions; Examples: RPT, RPTB
- Stack-manipulating instructions; Examples: PUSHD, POPD
- Miscellaneous program-control instructions; Examples: IDLE, RESET

For detailed descriptions of these and other instructions, the reader is referred to the Texas Instruments' TMS320C54xx DSP Reference Set, Volume 2: *Mnemonic Instruction Set* [6]. We shall now discuss a few of these instructions in detail.

Multiply Instruction (MPY)

This instruction can take several forms. One such form is

MPY *Xmem*, *Ymem*, *dst*; where *Xmem* and *Ymem* are dual data-memory operands and *dst* is accumulator A or B.

The instruction multiplies a data-memory value by another data-memory value and stores the result in accumulator A or B. The register T is loaded with the Xmem value in the read-memory phase.

$$\text{dst} \leftarrow (\text{Xmem}) \times (\text{Ymem}); \quad \text{T} \leftarrow (\text{Xmem})$$

In the indirect addressing mode, the instruction can also modify the contents of the auxiliary registers used for indirect addressing.

▷ **Example 5.6** Describe the operation of the following MPY instructions:

a. MPY 13, B

b. MPY #01234, A

c. MPY *AR2−, *AR4 + 0, B

Solution Instruction (a) multiplies the current contents of the T register by the contents of the data-memory location 13 in the current data page. The result is placed in the accumulator B.

Instruction (b) multiplies the current contents of the T register by the constant 1234 and places the result in the accumulator A.

Instruction (c) multiplies the contents of memory pointed by AR2 by the contents of memory pointed by AR4. The result is placed in the accumulator B. During this instruction execution, register T is loaded with the contents of the same data-memory location pointed by AR2. AR2 is then decremented by 1 and AR4 is updated by adding to it the contents of AR0.

Multiply and Accumulate Instruction (MAC)

This instruction is an improvement over the MPY instruction. One of the several forms that this instruction can take is

MAC *Xmem*, *Ymem*, *src*, *dst*; where *Xmem* and *Ymem* are dual data-memory operands and *src* and *dst* are accumulators A and B.

The instruction multiplies a data-memory value by another data-memory value and adds the product to the contents of the source, which may be either of the two accumulators A and B. The result is stored in the other accumulator. The register T is loaded with the Xmem value.

$$\text{dst} \leftarrow (\text{Xmem}) \times (\text{Ymem}) + (\text{src}); \quad T \leftarrow (\text{Xmem})$$

Similar to the MPY instruction, this instruction can modify the contents of auxiliary registers used in indirect addressing.

▷ **Example 5.7** Describe the operation of the following MAC instructions:

a. MAC *AR5+, #1234h, A

b. MAC *AR3−, *AR4+, B, A

Solution Instruction (a) multiplies the contents of the data-memory location pointed by AR5 by the constant 1234h and adds the product to the contents of the accumulator A. During the execution, register T is loaded with the content of the data-memory location pointed by AR5. AR5 is then incremented by 1.

Instruction (b) multiplies the contents of the data memory pointed by AR3 by the contents of the data memory pointed by AR4. The contents of the accumulator B are added to the product and the result is placed in the accumulator A. The register T is loaded with the contents of the same data-memory location pointed by AR3. AR3 is then decremented by 1 and AR5 is incremented by 1.

The MAC instruction is used for computing the sum of a series of product terms.

Multiply and Subtract Instruction (MAS)

This instruction is similar to the MAC instruction. One form of this instruction is

MAS *Xmem*, *Ymem*, *src*, *dst*; where *Xmem* and *Ymem* are dual data-memory operands and *src* and *dst* are accumulators A and B.

The instruction multiplies a data-memory value by another data-memory value and subtracts the product from the contents of the source, which may be either of the two accumulators A and B. The result is stored in the other accumulator. The register T is loaded with the Xmem value in the read-memory phase.

$$\text{dst} \leftarrow (\text{src}) - (\text{Xmem}) \times (\text{Ymem}); \quad T \leftarrow (\text{Xmem})$$

In the indirect mode, in addition to the multiply operation, the instruction can modify the contents of the auxiliary registers used for indirect addressing.

▷ **Example 5.8** Describe the operation of the following MAS instruction:

```
MAS *AR3-, *AR4+, B, A
```

Solution This instruction multiplies the contents of the data memory pointed by AR3 by the contents of the data memory pointed by AR4. The product is subtracted from the contents of the accumulator B and the result is placed in the accumulator A. During this instruction, register T is loaded with the contents of the same data-memory location pointed by AR3. AR3 is then decremented by 1 and AR5 incremented by 1.

The MAS instruction is used for computing butterflies in FFT implementation.

Multiply, Accumulate, and Delay Instruction (MACD)

This instruction carries out all the functions of the MAC instruction and, in addition, copies the contents of the current data-memory address to the next higher data-memory address. However, the two operands of the multiplier are required to be a single data-memory value and a program-memory value. This feature is equivalent to implementing the z^{-1} delay encountered in digital signal-processing algorithms. For this reason, the MACD instruction is often used for implementing FIR filters. The format and all other features of the MACD instruction are same as those of the MAC instruction.

Repeat Instruction (RPT)

The format of this instruction is

```
        RPT Smem   ; Smem is a single data-memory operand
or      RPT #k     ; k is a short or a long constant.
```

The instruction loads the operand in the repeat counter, RC. The instruction following the RPT instruction is repeated $k+1$ times, where k is the initial value of the RC.

Due to the dedicated hardware support, the repeat instruction is used to repeat an instruction a given number of times without any penalty for looping. It may be used to compute the sum of products as required in the implementation of FIR filters.

▷ **Example 5.9** Explain what is accomplished by the following instruction sequence:

```
RPT  #2
MAC  *AR1+, *AR2-, A
```

Solution The first instruction loads the register RC with 2. This number is the repeat count for the next MAC instruction. The MAC instruction executes three times. It multiplies and accumulates in A the data locations contents pointed to by the registers AR1 and AR2. After each multiply and add the pointer AR1 is incremented and pointer AR2 is decremented.

Block Repeat Instruction (RPTB)

RPTB instruction has the format

RPTB *pmad*, where *pmad* is the program memory address denoting the end of the block of instructions to be repeated.

This instruction is similar to the RPT instruction, except that it repeats a block of code a given number of times without any penalty for looping. One more than the number of times the block of instructions is to be repeated is initially loaded into the memory-mapped block repeat counter register, BRC.

5.7.2 Programming Examples

We now look at a few sample programs written for the TMS320C54xx signal processors. These programs particularly illustrate the use of some of the signal-processing instructions and the addressing modes to access data operands.

▷ **Example 5.10** Write a program to find the sum of a series of signed numbers stored at successive locations in the data memory and place the result in the accumulator A, i.e.,

$$A = \sum_{i=410h}^{41fh} dmad(i) \tag{5.1}$$

Solution The TMS320C54xx program for this example is shown in Figure 5.14. AR1 is used as the pointer to the numbers and AR2 as the counter for the numbers. The program initializes the accumulator to 0, sets AR1 to 410h to point to the first number and AR2 to the initial count. This will be used to track the number of processed locations at each step of execution. Sign-extension mode is selected to handle signed numbers. The program adds each number in turn to the accumulator, increments the pointer and decrements the counter. The process is repeated until the count in AR2 reaches 0. At the end of the program, the accumulator A has the sum of the numbers in location s 410h to 41fh.

▷ **Example 5.11** Write a program to compute the sum of three product terms given by the equation

$$y(n) = h_0x(n) + h_1x(n-1) + h_2x(n-2) \tag{5.2}$$

where $x(n)$, $x(n-1)$ and $x(n-2)$ are data samples stored at three successive

```
*****************************************************************
*
* This program computes the signed sum of data memory locations
* from address 410h to 41fh. The result is placed in A.
*
*   A = dmad(410h) + dmad(411h) + ... dmad(41fh)
*
*****************************************************************

    .mmregs
    .global _c_int00

    .text

_c_int00:
    STM   #10H, AR2      ; Initialize counter AR2 = 10h
    STM   #410H, AR1     ; Initialize pointer AR1 = 410h
    LD    #0H, A         ; Initialize sum A = 0
    SSBX  SXM            ; Select sign extension mode

START:
    ADD   *AR1+, A       ; Add the next data value
    BANZ  START, *AR2-   ; Repeat if not done
    NOP                  ; No operation

    .end
```

Figure 5.14 TMS320C54xx program for Example 5.10

data-memory locations and h_0, h_1, and h_2 are constants stored at three other successive locations in the data memory. The result $y(n)$ is to be stored in the data memory. Use direct addressing mode to access the data memory.

Solution Let h_0, h_1, and h_2 be stored starting at address h, and $x(n)$, $x(n-1)$, and $x(n-2)$ starting at address 310h in the data memory. Product terms $h_0x(n)$, $h_1x(n-1)$, and $h_2x(n-2)$ are computed using the MPY instruction by moving one of the operands to register T and accessing the other operand directly from the data memory. Note that the data-page pointer, DP, needs to be initialized before using the direct addressing mode to access the operand. Product terms are computed in A or B and added. When all the three multiplications are done, the result accumulated in B is stored in the data memory $y(n)$. Since $y(n)$ is 32 bits long, it is saved at two successive locations labeled as y, with the lower 16 bits at memory location y and the higher 16 bits at the next memory location. The TMS320C54xx program for this example is shown in Figure 5.15.

```
*************************************************************************
*
* This program computes multiply and accumulate using direct addressing
* mode.
*
*   y(n) = h(0)x(n) + h(1)x(n-1) + h(2)x(n-2)
*
*   h(0), h(1), and h(2) are stored in data-memory locations starting at
*   location h and x(n), x(n-1), and x(n-2) are stored in data-memory
*   locations starting at location x. y(n) is saved in data-memory
*   location y (low 16 bits) and y + 1 (high 16 bits).
*
*************************************************************************

    .global _c_int00

x   .usect "Input Samples", 3
y   .usect "Output", 2
h   .usect "Coefficients", 3

    .text

_c_int00:
    SSBX  SXM      ; Select sign extension mode
    LD    #h, DP   ; Select the data page for coefficients
    LD    @h, T    ; Get the coefficient h(0)
    LD    #x, DP   ; Select the data page for input samples
    MPY   @x, A    ; A = x(n)*h(0)

    LD    #h, DP   ; Select the data page for coefficients
    LD    @h+1, T  ; Get the coefficient h(1)
    LD    #x, DP   ; Select the data page for input signals
    MPY   @x+1, B  ; B = x(n-1)*h(1)

    ADD   A, B     ; B = x(n)*h(0) + x(n-1)*h(1)

    LD    #h, DP   ; Select the data page for coefficients
    LD    @h+2, T  ; Get the coefficient h(2)
    LD    #x, DP   ; Select the data page for input signals
    MPY   @x+2, A  ; A = x(n-2)*h(3)

    ADD   A, B     ; B = x(n)*h(0) + x(n-1)*h(1) + x(n-2)*h(3)

    LD    #y, DP   ; Select the data page for output
    STL   B, @y    ; Save low part of output
    STH   B, @y+1  ; Save high part of output
    NOP            ; No operation

    .end
```

Figure 5.15 TMS320C54xx program for Example 5.11

▷ **Example 5.12** Repeat the problem of Example 5.11 using the indirect addressing mode to access data.

Solution In this example, let us use the auxiliary register AR2 to address the data using the indirect addressing mode. AR2 is initialized to 310h, the location where $x(n)$ is stored, and is advanced to the next address after each multiply opera-

```
*************************************************************************
*
* This program computes multiply and accumulate using indirect
* addressing mode.
*
*   y(n) = h(0)x(n) + h(1)x(n-1) + h(2)x(n-2)
*
*   h(0), h(1), and h(2) are stored in data-memory locations starting at
*   location h, x(n), x(n-1), and x(n-2) are stored in data-memory
*   locations 310h, 311h, & 312h resp. y(n) is saved in data-memory
*   location 313h (low 16 bits) and 314h (high 16 bits)
*
*************************************************************************
    .global _c_int00
h   .int 10, 20, 30
    .text
_c_int00:
    SSBX  SXM              ; Select sign extension mode
    STM   #310H, AR2       ; Initialize pointer AR2 for x(n) stored at
                             310H
    STM   #h, AR3          ; Initialize pointer AR3 for coefficients
    MPY   *AR2+, *AR3+, A  ; A = x(n)*h(0)
    MPY   *AR2+, *AR3+, B  ; B = x(n-1)*h(1)
    ADD   A, B             ; B = x(n)*h(0) + x(n-1)*h(1)
    MPY   *AR2+, *AR3+, A  ; A = x(n-2)*h(2)
    ADD   A, B             ; B = x(n)*h(0) + x(n-1)*h(1) + x(n-2)*h(2)
    STL   B, *AR2+         ; Save low part of result
    STH   B, *AR2+         ; Save high part of result
    NOP                    ; No operation
    .end
```

Figure 5.16 TMS320C54xx program for Example 5.12

tion. AR3 is used as the pointer to access coefficients starting at h. At the end of three multiply operations, AR2 points to 313h, the address at which the lower 16 bits of $y(n)$ are to be stored. The TMS320C54xx program for this example is shown in Figure 5.16.

▷ **Example 5.13** Repeat the problem of Example 5.11 by using the MAC instruction.

```
*************************************************************************
*
* This program computes multiply and accumulate using the MAC
* instruction
*
*   y(n) = h(0)x(n) + h(1)x(n-1) + h(2)x(n-2)
*
*   where, h(0), h(1), and h(2) are in the program-memory locations
*   starting at h, x(n), x(n-1), and x(n-2) are in data-memory locations
*   starting at x. y(n) is to be saved in location y (low 16 bits) and
*   y + 1 (high 16 bits).
*
*************************************************************************

    .global _c_int00

    .data
    .bss x, 3
    .bss y, 2

h   .int 10, 20, 30

    .text

_c_int00:
    SSBX  SXM              ; Select sign extension mode
    STM   #x, AR2          ; Initialize AR2 to point to x(n)
    STM   #h, AR3          ; Initialize AR3 to point to h(0)
    LD    #0H, A           ; Initialize result in A = 0

    RPT   #2               ; Repeat the next operation 3 times
    MAC   *AR2+, *AR3+, A  ; y(n) computed

    STM   #y, AR2          ; Select the page for y(n)
    STL   A, *AR2+         ; Save the low part of y(n)
    STH   A, *AR2+         ; Save the high part of y(n)
    NOP                    ; No operation

    .end
```

Figure 5.17 The TMS320C54xx program for Example 5.13

Solution The MAC instruction multiplies the contents of two data-memory locations and adds the result to the previous contents of the accumulator being used. (Note that only auxiliary registers AR2–AR5 can be used.) This instruction is repeated twice using RPT instruction. After each MAC instruction the auxiliary registers, which are being used, should be incremented by 1. Finally, the result is stored in the memory location pointed by "*y*" using STL instruction first for the lower 16 bits and then using STH instruction for the higher 16 bits. The TMS32054Cxx program for this example is shown in Figure 5.17.

5.8 On-Chip Peripherals

On-chip peripherals facilitate interfacing with external devices such as modems and analog-to-digital converters. They also provide certain features that are required for implementing real time systems using the processors. All the '54xx devices have the same CPU, but different on-chip peripherals are available in different devices. These peripherals include general-purpose I/O pins, a software-programmable wait-state generator, hardware timer, host port interface (HPI), clock generator, and serial ports. Of these, the general-purpose I/O and the software-programmable wait-state generator are described in Chapter 9 on parallel peripheral devices. The timer, the host port interface, clock generator, and serial ports are briefly described below. The tables in Appendix A give details of the information required for programming these on-chip peripherals.

5.8.1 Hardware Timer

The timer is an on-chip down counter that can be used to generate a signal to initiate an interrupt or to initiate any other process. The timer consists of three memory-mapped registers—TIM, PRD, and TCR. A logical block diagram of the timer circuit is shown in Figure 5.18. The timer register (TIM) is a 16-bit memory-mapped register that decrements at every pulse from the prescaler block (PSC). The timer period register (PRD) is a 16-bit memory-mapped register whose contents are loaded onto the TIM whenever the TIM decrements to zero or the device is reset ($\overline{\text{SRESET}}$). The timer can also be independently reset using the TRB signal. The timer control register (TCR) is a 16-bit memory-mapped register that contains status and control bits. Table 5.4 shows the functions of the various bits in the TCR. The prescaler block is also an on-chip counter. Whenever the prescaler bits count down to 0, a clock

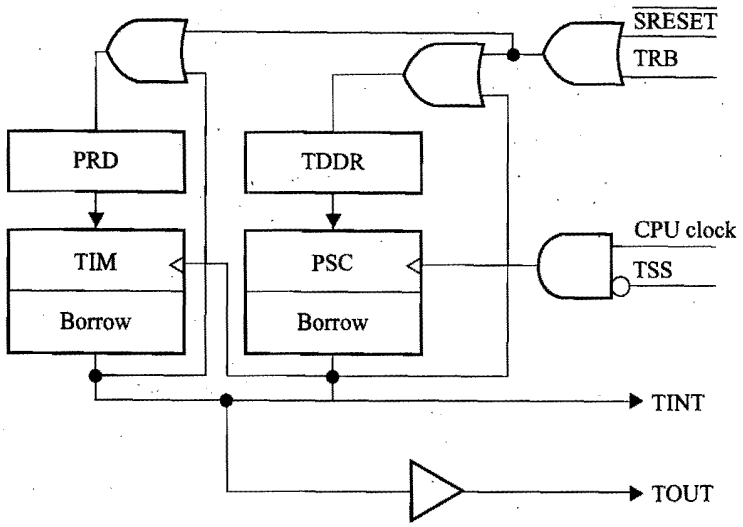


Figure 5.18 Logical block diagram of timer circuit
(Courtesy of Texas Instruments Inc.)

pulse is given to the TIM register that decrements the TIM register by 1. The TDDR bits contain the divide-down ratio, which is loaded onto the prescaler block after each time the prescaler bits count down to 0. That is to say that the 4-bit value of TDDR determines the divide-by ratio of the timer clock with respect to the system clock. In other words, the TIM decrements either at the rate of the system clock or at a rate slower than that as decided by the value of the TDDR bits. TOUT and TINT are the output signals generated as the TIM register decrements to 0. TOUT can trigger the start of the conversion signal in an ADC interfaced to the DSP. The sampling frequency of the ADC determines how frequently it receives the TOUT signal. TINT is used to generate interrupts, which are required to service a peripheral such as a DRAM controller periodically. The timer can also be stopped, restarted, reset, or disabled by specific status bits.

5.8.2 Host Port Interface (HPI)

The host port interface (HPI) is a unit that allows the DSP to interface to an 8-bit or a 16-bit host device or a host processor. The HPI communicates with the host independently of the DSP. The HPI features allow the host to interrupt the DSP, or vice versa, when required. The interface contains minimal

Table 5.4 Function of Various Bits in the TCR Registers

Bit	Name	Reset Value	Function
15–12	Reserved	—	Reserved; always read as 0.
11	Soft	0	Used in conjunction with the Free bit to determine the state of the timer when a breakpoint is encountered in the HLL debugger. When the Free bit is cleared, the Soft bit selects the timer mode. Soft = 0 The timer stops immediately. Soft = 1 The timer stops when the counter decrements to 0.
10	Free	0	Used in conjunction with the Soft bit to determine the state of the timer when a breakpoint is encountered in the HLL debugger. When the Free bit is cleared, the Soft bit selects the timer mode. Free = 0 The Soft bit selects the timer mode. Free = 1 The timer runs free regardless of the Soft bit.
9–6	PSC	—	Timer prescaler counter. Specifies the count for the on-chip timer. When PSC is decremented past 0 or the timer is reset, PSC is loaded with the contents of TDDR and the TIM is decremented.
5	TRB	—	Timer reload. Resets the on-chip timer. When TRB is set, the TIM is loaded with the value in the PRD and the PSC is loaded with the value in TDDR. TRB is always read as a 0.
4	TSS	0	Timer stop status. Stops or starts the on-chip timer. At reset, TSS is cleared and the timer immediately starts timing. TSS = 0 The timer is started. TSS = 1 The timer is stopped.
3–0	TDDR	0000	Timer divide-down ratio. Specifies the timer divide-down ratio (period) for the on-chip timer. When PSC is decremented past 0, PSC is loaded with the contents of TDDR.

(Courtesy of Texas Instruments Inc.)

external logic, so that a system with a host and a DSP can be designed without increasing the hardware on the board. The HPI interfaces to the PC parallel ports directly. A generic block diagram of the HPI is shown in Figure 5.19.

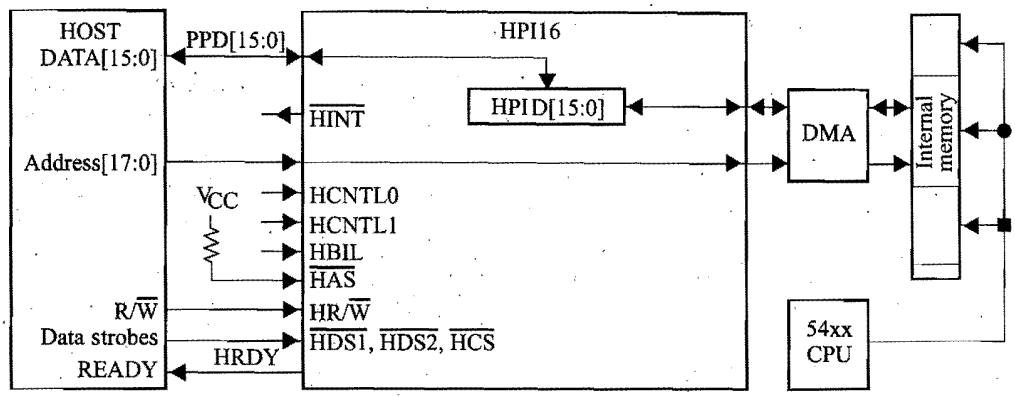


Figure 5.19 A generic diagram of the host port interface (HPI)
(Courtesy of Texas Instruments Inc.)

Important signals in the HPI are as follows:

- The 16-bit data bus and the 18-bit address bus.
- The host interrupt, HINT, for the DSP to signal the host when its attention is required.
- HRDY, a DSP output indicating that the DSP is ready for transfer.
- HCNTL0 and HCNTL1, control signals that indicate the type of transfer to carry out. The transfer types are data, address, etc.
- HBIL. If this is low it indicates that the current byte is the first byte; if it is high, it indicates that it is the second byte.
- HR/$\overline{W}$, indicates if the host is carrying out a read operation or a write operation.

By appropriately using these signals, the DSP device can be interfaced on a host such as a PC.

5.8.3 Clock Generator

The clock generator on TMS320C54xx devices has two options—an external clock and the internal clock. In the case of the external clock option, a clock source is directly connected to the device. The internal clock source option, on the other hand, uses an internal clock generator and a phase locked loop (PLL) circuit. The PLL, in turn, can be hardware configured or software programmed. Not all devices of the TMS320C54xx family have all these clock options; they vary from device to device.

5.8.4 Serial I/O Ports

Three types of serial ports are available on the '54xx devices, depending on the type of the device. These are synchronous, buffered, and time-division multiplexed ports.

The synchronous serial ports are high-speed, full-duplex ports that provide direct communication with serial devices, such as codec, and analog-to-digital (A/D) converters. A buffered serial port (BSP) is a synchronous serial port that is provided with an autobuffering unit and is clocked at the full clock rate. The autobuffering unit supports high-speed data transfers and reduces the overhead of servicing interrupts. A time-division multiplexed (TDM) serial port is a synchronous serial port that is provided to allow time-division multiplexing of the data. We will cover serial I/O in Chapter 10.

The functioning of each of these on-chip peripherals is controlled by memory-mapped registers assigned to the respective peripheral. Figure 5.8(b) gives the list of peripheral memory-mapped registers along with their addresses for the TMS320C54xx devices.

5.9 Interrupts of TMS320C54xx Processors

Many times, when the CPU is in the midst of executing a program, a peripheral device may require a service from the CPU. In such a situation, the main program may be interrupted by a signal generated by the peripheral device. This results in the processor suspending the main program in order to execute another program, called interrupt service routine, to service the peripheral device. On completion of the interrupt service routine, the processor returns to the main program to continue from where it left.

Interrupt may be generated either by an internal or an external device. It may also be generated by software. Not all interrupts are serviced when they occur. Only those interrupts that are called *nonmaskable* are serviced whenever they occur. Other interrupts, which are called *maskable* interrupts, are serviced only if they are enabled. There is also a priority to determine which interrupt gets serviced first if more than one interrupts occur simultaneously.

Almost all the devices of the TMS320C54xx family have 32 interrupts, However, the types and the number under each type vary from device to device. Some of these interrupts are reserved for use by the CPU. Figure 5.20 gives the types of interrupts, their locations, and priorities for TMS320C54xx processors.

A more detailed description of interrupts and how an interrupt is handled when it occurs is given in Chapter 9.

NAME	LOCATION		PRIORITY	FUNCTION
	DECIMAL	HEX		
$\overline{RS}$, SINTR	0	00	1	Reset (hardware and software reset)
$\overline{NMI}$, SINT16	4	04	2	Nonmaskable interrupt
SINT17	8	08	—	Software interrupt #17
SINT18	12	0C	—	Software interrupt #18
SINT19	16	10	—	Software interrupt #19
SINT20	20	14	—	Software interrupt #20
SINT21	24	18	—	Software interrupt #21
SINT22	28	1C	—	Software interrupt #22
SINT23	32	20	—	Software interrupt #23
SINT24	36	24	—	Software interrupt #24
SINT25	40	28	—	Software interrupt #25
SINT26	44	2C	—	Software interrupt #26
SINT27	48	30	—	Software interrupt #27
SINT28	52	34	—	Software interrupt #28
SINT29	56	38	—	Software interrupt #29
SINT30	60	3C	—	Software interrupt #30
$\overline{INT0}$, SINT0	64	40	3	External user interrupt #0
$\overline{INT1}$, SINT1	68	44	4	External user interrupt #1
$\overline{INT2}$, SINT2	72	48	5	External user interrupt #2
TINT, SINT3	76	4C	6	Timer interrupt
RINT0, SINT4	80	50	7	McBSP #0 receive interrupt (default)
XINT0, SINT5	84	54	8	McBSP #0 transmit interrupt (default)
RINT2, SINT6	88	58	9	McBSP #2 receive interrupt (default)
XINT2, SINT7	92	5C	10	McBSP #2 transmit interrupt (default)
$\overline{INT3}$, SINT8	96	60	11	External user interrupt #3
$\overline{HINT}$, SINT9	100	64	12	HPI interrupt
RINT1, SINT10	104	68	13	McBSP #1 receive interrupt (default)
XINT1, SINT11	106	6C	14	McBSP #1 transmit interrupt (default)
DMAC4, SINT12	112	70	15	DMA channel 4 (default)
DMAC5, SINT13	116	74	16	DMA channel 5 (default)
Reserved	120–127	78–7F	—	Reserved

Figure 5.20 Table for interrupt locations and priorities for TMS320C54xx processors (Courtesy of Texas Instruments Inc.)

5.10 Pipeline Operation of TMS320C54xx Processors

The CPU of '54xx devices has a six-level-deep instruction pipeline. The six stages of the pipeline are independent of each other. This allows overlapping execution of instructions. During any given cycle, up to six different instructions can be active, each at a different stage of processing. The six levels of the pipeline structure are program prefetch, program fetch, decode, access, read, and execute.

1. During program prefetch, the program address bus, PAB, is loaded with the address of the next instruction to be fetched.
2. In the fetch phase, an instruction word is fetched from the program bus, PB, and loaded into the instruction register, IR. These two phases form the instruction fetch sequence.
3. During the decode stage, the contents of the instruction register, IR, are decoded to determine the type of memory access operation and the control signals required for the data-address generation unit and the CPU.

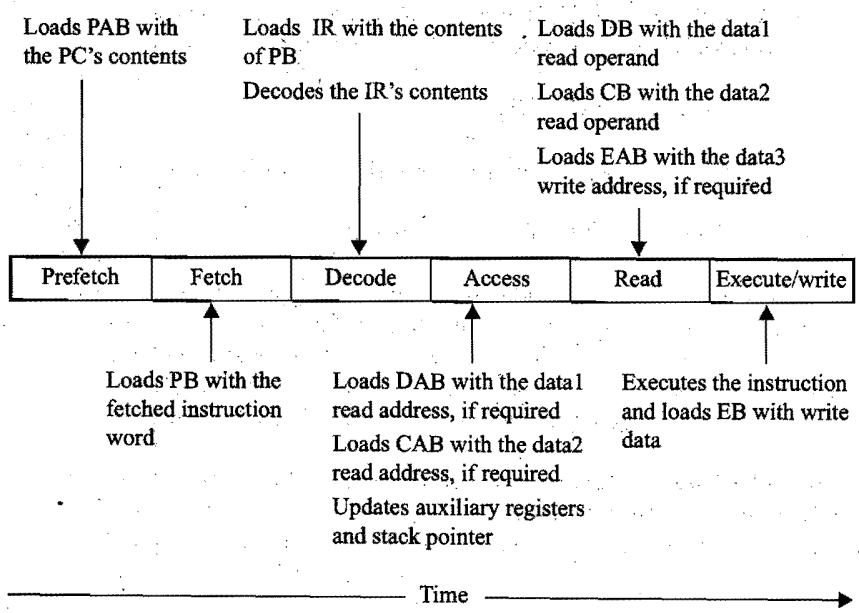


Figure 5.21 Six-stage pipeline of TMS320C54xx execution
(Courtesy of Texas Instruments Inc.)

4. The access phase outputs the read operand's address on the data address bus, DAB. If a second operand is required, the other data address bus, CAB, is also loaded with an appropriate address. Auxiliary registers in indirect addressing mode and the stack pointer (SP) are also updated.
5. In the read phase the data operand(s), if any, are read from the data buses, DB and CB. This phase completes the two-phase read process and starts the two-phase write process. The data address of the write operand, if any, is loaded into the data write address bus, EAB.
6. The execute phase writes the data using the data write bus, EB, and completes the operand write sequence. The instruction is also executed in this phase.

Figure 5.21 shows the six stages of the pipeline and the events that occur in each stage. The following examples demonstrate how the TMS320C54xx pipeline works while executing instructions.

▷ **Example 5.14** Show the pipeline operation of the following sequence of instructions if the initial value of AR3 is 80 and the values stored in memory location 80, 81, 82 are 1, 2, and 3.

```
LD *AR3+, A
ADD #1000h, A
STL A, *AR3+
⋮
⋮
```

Solution Figure 5.22 is the solution to this example problem.

Cycle	Prefetch	Fetch	Decode	Access	Read	Exec & Write	AR3	A
1	LD						80	X
2	ADD	LD					80	X
3	STL	ADD	LD				80	X
4		STL	ADD	LD			81	X
5			STL	ADD	LD		81	1
6				STL	[redacted]	LD	82	0001h
7					STL	ADD	82	1001h
8						STL	82	1001h

Figure 5.22 Pipeline operation of the instruction sequence of Example 5.14

Cycle	Prefetch	Fetch	Decode	Access	Read	Exec & Write	AR3	AR1	A	T
1	ADD						81	84	1	X
2	LD	ADD					81	84	1	X
3	MPY	LD	ADD				81	84	1	X
4	ADD	MPY	LD	ADD			82	84	1	X
5		ADD	MPY	LD	ADD		82	85	1	X
6			ADD	MPY	LD	ADD	83	85	03	06
7				ADD	MPY	LD	83	85	03	06
8					ADD	MPY	83	85	03	06
9						ADD	83	85	15h	06

Figure 5.23 Pipeline operation of the instruction sequence of Example 5.15

▷ **Example 5.15** Show the pipeline operation of the following sequence of instructions if the initial values of AR1, AR3, A are 84, 81, 1 and the values stored in memory location 81, 82, 83, 84 are 2, 3, 4, 6. Also provide the values of registers AR3, AR1, T and accumulator, A, after completion of each cycle.

```
ADD *AR3+, A
LD *AR1+, T
MPY *AR3+, B
ADD B, A
⋮
```

Solution Figure 5.23 is the solution to this example problem.

5.11 Summary

In this chapter, we have looked at the architectural features of the commercially available programmable digital signal processors. In particular, we have studied in detail the following features of the Texas Instruments TMS320C54xx DSPs:

- Architecture of the processors, consisting of the bus structure, central processing unit (CPU), and internal memory organization

- Addressing modes, consisting of immediate addressing, absolute addressing, accumulator addressing, direct addressing, indirect addressing, memory-mapped addressing, and stack addressing
- Address-generation unit, including single-operand address modifications, circular address modifications, bit-reversed address modifications, and dual-operand address modifications
- Assembly language instructions, including signal processing-specific instructions and programming examples
- Memory organization
- On-chip peripherals
- Interrupts
- Pipeline operation

References

1. TMS320C2x *User's Guide*, Texas Instruments, 1993.
2. DSP 56000/56001 Digital Signal Processor *User's Manual*, Motorola, 1993.
3. ADSP2101/2102 *User's Manual*, Analog Devices, 1993.
4. TMS320C54xx *DSP Reference Set*, Vols. 1 and 2, Texas Instruments, 1997.
5. TMS320VC5416 *DSP Data Manual*, Texas Instruments, 2002.
6. TMS320C54x *DSP Reference Set*, Vol. 2, Texas Instruments, 1997.

Assignments

5.1 How will you configure a TMS320C5416 processor to have the following on-chip memories? Specify the address range in each case.

On-chip DARAM: for program

On-chip ROM: for program

How much RAM for data will be available in the specified configuration?

5.2 Explain the difference between the internal and external modes of clocking TMS320C54xx processors. How do you vary the clock frequency in each case?

5.3 Identify the addressing mode of the source operand in each of the following instructions:

a. ADD *AR2, A

b. ADD *AR2+, A

c. ADD *AR2 + %, A
d. ADD #0ffh, A
e. ADD 1234h, A
f. ADD *AR2 + 0B, A
g. ADD *+AR2, A

5.4 What will be the contents of accumulator A after the execution of the instruction

LD *AR4, 4, A

if the current AR4 points to a memory location whose contents are 8b0eh and the SXM bit of the status register ST1 is set?

5.5 Write a sequence of TMS320C54xx instructions to configure a circular buffer with a start address at 0200h and an end address at 021fh with current buffer pointer (AR6) pointing to address 0205h.

5.6 Write a TMS320C54xx program to compute the equation

$$y = mx + c$$

Assume that x and c are stored in the data memory and m in the program memory. The result should be stored in the data memory.

5.7 Write a TMS320C54xx program to implement second-order IIR filter equations

$$d(n) = x(n) + d(n-1)a_1 + d(n-2)a_2$$
$$y(n) = d(n)b_0 + d(n-1)b_1 + d(n-2)b_2$$

where a_1, a_2, b_0, b_1, b_2 are filter coefficients (integers), $x(n)$ is the latest input sample, $y(n)$ is the filtered output sample, and $d(n)$ is an intermediate result. You may assume that, during calculations, all signals remain within values represented by 16 bits.

5.8 Write a TMS320C54xx program to read the cosine value of a variable from a table stored in the program memory and store it in the data memory. The variable is located at address VALUE in the data memory, and the cosine value should be stored at the same location. The cosine table is stored at address TABLE in the program memory.

5.9 Write a TMS320C54xx program to read 100h words from the input port at address INPORT and store them in the data memory starting at address BUFFER.

5.10 Write a TMS320C54xx program to mask the lower 6 bits of a word stored in the data memory and write the modified word back at the same location.

5.11 What is the role of the interrupt pins in a DSP device? Are these the only means of interrupting a DSP program? How do you prevent a signal on an interrupt pin from interrupting a time-critical program being executed by the DSP?

5.12 By means of a figure, explain the pipeline operation of the following sequence of TMS320C54xx instructions if the initial value of AR3 is 80 and the values stored in memory location 80, 81, 82 are 1, 2, and 3.

```
LD  *AR3+, A
ADD *AR3+, A
STL A, *AR3+
    .
    .
```

UNIT III TMS320C6X PROGRAMMABLE DSP PROCESSOR

Commercial TI DSP processors, Architecture of TMS320C6x DSP Processor, Linear and Circular addressing modes, TMS320C6x Instruction Set, Assembler directives, Linear Assembly, Interrupts, Multichannel buffered serial ports, Block diagram of TMS320C67xx DSP Starter Kit and Support Tools

COMMERCIAL TI DSP PROCESSORS :

- Digital signal processors, such as the TMS320 family of processors, are used in a wide range of applications, such as in communications, controls, speech processing,and so on. They are used in cellular phones, digital cameras, high-definitiontelevision (HDTV), radio, fax transmission, modems, and other devices.
- These devices have also found their way into the university classroom, where they provide an economical way to introduce real-time digital signal processing (DSP) ,
- **Texas Instruments** introduced the TM320C6x processor, based on the very-longinstruction- word (VLIW) architecture.This new architecture supports features that facilitate the development of efficient high-level language compilers.
- Throughout to the C/C++ language simply as C.Although TMS320C6x/assembly language can produce fast code, problems with documentation and maintenancemay exist.With the available C compiler, the programmer.
- Digital signal processors such as the TMS320C6x (C6x) family of processors are like fast special-purpose microprocessors with a specialized type of architecture and an instruction set appropriate for signal processing.
- The C6x notation is used to designate a member of **Texas Instruments' (TI) TMS320C6000 family of digital signal processors.** The architecture of the C6x digital signal processor is very well suited for numerically intensive calculations.
- Based on a very-long-instruction-word (VLIW) architecture, the C6x is considered to be TI's most powerful processor.
- Digital signal processors are used for a wide range of applications, from communications **and controls to speech and image processing.** The general-purpose digital signal processor is dominated by applications in communications (cellular).

- Applications embedded digital signal processors are dominated by consumer products.They are found in **cellular phones, fax/modems, disk drives, radio, printers,hearing aids, MP3 players, high-definition television (HDTV), digital cameras, and so on.**
- These processors have become the products of choice for a number of consumer applications, since they have become very cost-effective.They can handle different tasks, since they can be reprogrammed readily for a different application.
- DSP techniques have been very successful because of the development of low-cost software and hardware support. For example, modems and speech recognition can be less expensive using DSP techniques.
- DSP processors are concerned primarily with real-time signal processing. Realtime processing requires the processing to keep pace with some external event, whereas non-real-time processing has no such timing constraint.The external event to keep pace with is usually the analog input.Whereas analog-based systems with discrete electronic components such as resistors can be more sensitive to temperature changes, DSP-based systems are less affected by environmental conditions.
- DSP processors enjoy the advantages of microprocessors. They are easy to use, flexible, and economical. A number of books and articles address the importance of digital signal processors for a number of applications [1–22]. Various technologies have been used for real-time processing, from fiberoptics for very high frequency to DSPs very suitable for the audio-frequency range. Common applications using these processors have been for frequencies from 0 to 96kHz. Speech can be sampled at 8 kHz (the rate at which samples are acquired), which implies that each value sampled is acquired at a rate of 1/(8 kHz) or 0.125ms. A commonly used sample rate of a compact disk is 44.1kHz.
- Analog/digital (A/D)-based boards in the megahertz sampling rate rang are currently available.
- The basic system consists of an analog-to-digital converter (ADC) to capture an input signal. The resulting digital representation of the captured signal is then processed by a digital signal processor such as the C6x and then output through a digital-to-analog converter (DAC).
- Also included within the basic system are a special input filter for anti-aliasing to eliminate erroneous signals and an output filter to smooth or reconstruct the processed output signal.

TMS320C6713 Digital Signal Processor :

The TMS320C6713 (C6713) is based on the VLIW architecture, which is very well suited for numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. For example, with a clock rate of 225MHz, the C6713 is capable of fetching eight 32-bit instructions every 1/(225 MHz) or 4.44 ns.

Features of the C6713 include 264 kB of internal memory (8kB as L1P and L1D Cache and 256kB as L2 memory shared between program and data space), eight functional or execution units composed of six arithmetic-logic units (ALUs) and two multiplier units, a 32-bit address bus to address 4 GB (gigabytes), and two sets of 32-bit general-purpose registers.

The C67xx (such as the C6701, C6711, and C6713) belong to the family of the C6x floating-point processors, whereas the C62xx and C64xx belong to the family of the C6x fixed-point processors. The C6713 is capable of both fixed- and floatingpoint processing.

ARCHITECTURE OF TMS320C6X DSP PROCESSOR :

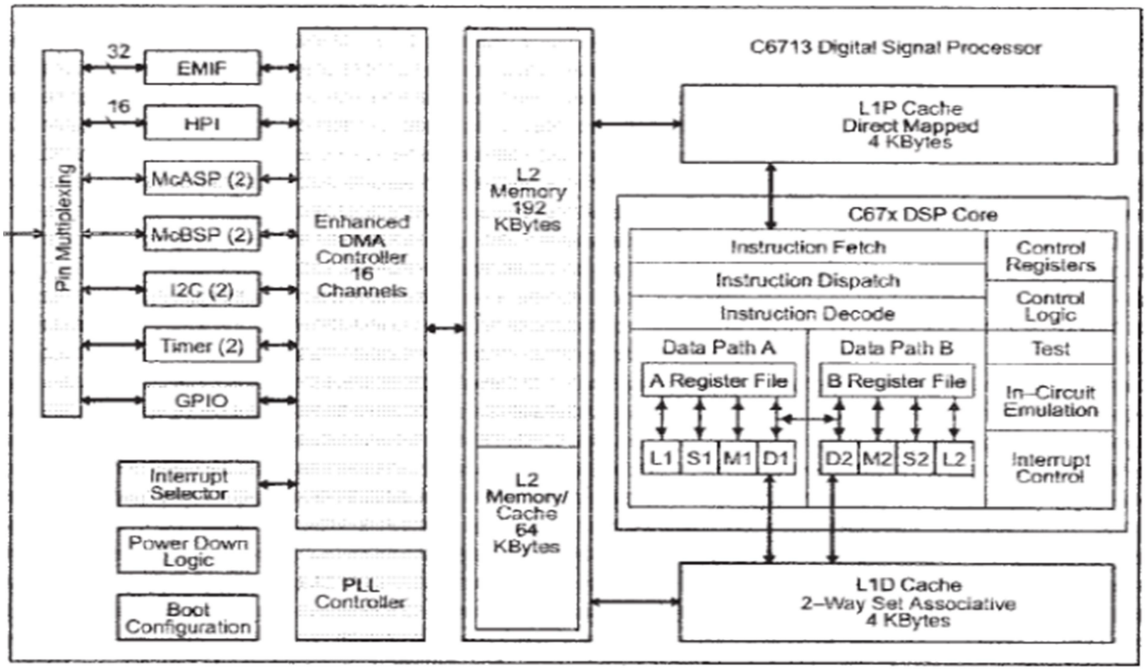
The TMS320C6713 onboard the DSK is a floating-point processor based on the VLIW architecture [6–10]. Internal memory includes a two-level cache architecture with 4 kB of level 1 program cache (L1P), 4 kB of level 1 data cache (L1D), and 256 kB of level 2 memory shared between program and data space. It has a glueless (direct) interface to both synchronous memories (SDRAM and SBSRAM) and asynchronous memories (SRAM and EPROM). Synchronous memory requires clocking but provides a compromise between static SRAM and dynamic DRAM, with SRAM being faster but more expensive than DRAM.

On-chip peripherals include two McBSPs, two timers, a host port interface (HPI), and a 32-bit EMIF. It requires 3.3 V for I/O and 1.26 V for the core (internal). Internal buses include a 32-bit program address bus, a 256-bit program data bus to accommodate eight 32-bit instructions, two 32-bit data address buses, two 64-bit data buses, and two 64-bit store data buses.With a 32-bit address bus, the total memory space is 232 = 4GB, including four external memory spaces: CE0, CE1, CE2, and CE3.

Independent memory banks on the C6x allow for two memory accesses within one instruction cycle. Two independent memory banks can be accessed using two independent buses. Since internal memory is organized into memory banks, two loads or two stores of instructions can be performed in parallel. No conflict results,

if the data accessed are in different memory banks. Separate buses for program, data, and direct memory access (DMA) allow the C6x to perform concurrent program fetches, data read and write, and DMA operations.With data and instructions residing in separate memory spaces, concurrent memory accesses are possible.

The C6x has a byte-addressable memory space. Internal memory is organized as separate program and data memory spaces, with two 32-bit internal ports (two 64- bit ports with the C64x) to access internal memory. The C6713 on the DSK includes 264kB of internal memory, which starts at 0x00000000, and 16MB of external SDRAM, mapped through CE0 starting at 0x80000000.The DSK also includes 512 kB of Flash memory (256 kB readily available to the user), mapped through CE1 starting at 0x90000000.



GURE 3.1. Functional block diagram of TMS320C6713 (Courtesy of Texas Instruments).

The L2 internal memory configuration, included with CCS [7]. Table 3.1 shows the memory map, also included with CCS [7]. A schematic diagram of the DSK is included with CCS (6713dsk_schem.pdf).

With the DSK operating at 225MHz, one can ideally achieve two multiplies and accumulates per cycle, for a total of 450 million multiplies and accumulates (MACs) per second. With six of the eight functional units in Figure 3.1 (not the .D units described below) capable of handling floating-point operations, it is possible to perform 1350 million floating-point operations per second (MFLOPS).

Operating at 225MHz, this translates into 1800 million instructions per second (MIPS) with a 4.44-ns instruction cycle time.

FUNCTIONAL UNITS :

The CPU consists of eight independent functional units divided into two data paths, A and B, as shown in Figure 3.1. Each path has a unit for multiply operations (.M), for logical and arithmetic operations (.L), for branch, bit manipulation, and arithmetic operations (.S), and for loading/storing and arithmetic operations (.D). The .S and .L units are for arithmetic, logical, and branch instructions. All data transfers make use of the .D units.

The arithmetic operations, such as subtract or add (SUB or ADD), can be performed by all the units, except the .M units (one from each data path). The eight unctional units consist of four floating/fixed-point ALUs (two .L and two .S), two fixed-point ALUs (.D units), and two floating/fixed-point multipliers (.M units). Each functional unit can read directly from or write directly to the register file within its own path.

Each path includes a set of sixteen 32-bit registers, A0 through A15 and B0 through B15. Units ending in 1 write to register file A, and units ending in 2 write to register file B. Two cross-paths (1x and 2x) allow functional units from one data path to access a 32-bit operand from the register file on the opposite side.There can be a maximum of two cross-path source reads per cycle.

Each functional unit side can access data from the registers on the opposite side using a cross-path (i.e., the functional units on one side can access the register set from the other side). There are 32 generalpurpose registers, but some of them are reserved for specific addressing or are used for conditional instructions.

LINEAR AND CIRCULAR ADDRESSING MODES :

Addressing modes determine how one accesses memory.They specify how data are accessed, such as retrieving an operand indirectly from a memory location. Both linear and circular modes of addressing are supported. The most commonly used mode is the indirect addressing of memory.

Indirect Addressing :

Indirect addressing can be used with or without displacement. Register R represents one of the 32 registers A0 through A15 and B0 through B15 that can

Specify or point to memory addresses.As such, these registers are pointers. Indirect addressing mode uses a "*" in conjunction with one of the 32 registers. To illustrate, consider R as an address register.

1. **R.* Register R contains the address of a memory location where a data value is stored.

2. **R++(d).* Register R contains the memory address (location). After the memory address is used, R is postincremented (modified) such that the new address is the current address offset by the displacement value d. If d = 1 (by default), the new address is R + 1, or R is incremented to the next higher address in memory. A double minus (--) instead of a double plus would update or postdecrement the address to R - d.

3. **++R(d).* The address is preincremented or offset by d, such that the current address is R + d. A double minus would predecrement the memory address so that the current address is R - d.

4. **+R(d).* The address is preincremented by d, such that the current address is R + d (as with the preceding case). However, in this case, R preincrements without modification. Unlike the previous case, R is not updated or modified.

Circular Addressing :

Circular addressing is used to create a circular buffer.This buffer is created in hardware and is very useful in several DSP algorithms, such as in digital filtering or correlation algorithms where data need to be updated. the implementation of a digital filter in assembly code using a circular buffer to update the "delay" samples. Implementing a circular buffer using C code is less efficient.

The C6x has dedicated hardware to allow a circular type of addressing. This addressing mode can be used in conjunction with a circular buffer to update samples by shifting data without the overhead created by shifting data directly. as a pointer reaches the end or "bottom" location of a circular buffer that contains the last element in the buffer, and is then incremented, the pointer is automatically wrapped around or points to the beginning or "top" location of the buffer that contains the first element.

Two independent circular buffers are available using BK0 and BK1 within the AMR. The eight registers A4 through A7 and B4 through B7, in conjunction with the two .D units, can be used as pointers (all registers can be used for linear addressing).The following code segment illustrates the use of a circular buffer using register B2 (only side B can be used) to set the appropriate values within AMR:

```
MVKL .S2 0x0004,B2 ;lower 16 bits to B2. Select A5 as pointer
MVKH .S2 0x0005,B2 ;upper 16 bits to B2. Select BK0, set N = 5
MVC .S2 B2,AMR ;move 32 bits of B2 to AMR
```

The two move instructions MVKL and MVKH (using the .S unit) move 0x0004 into the 16 LSBs of register B2 and 0x0005 into the 16 most significant bits (MSBs) of B2. The MVC (move constant) instruction is the only instruction that can access the AMR and the other control registers (shown in Appendix B) and executes only on the B side in conjunction with the functional units and registers on side B.

A 32- bit value is created in B2, which is then transferred to AMR with the instruction MVC to access AMR [6]. The value 0x0004 = (0100)b into the 16 LSBs of AMR sets bit 2 (the third bit) to 1 and all other bits to 0. This sets the mode to 01 and selects register A5 as the pointer to a circular buffer using block BK the modes associated with registers A4 through A7 and B4 through B7.The value 0x0005 = (0101)b into the 16MSBs of AMR sets bits 16 and 18 to 1 (other bits to 0). This corresponds to the value of *N* used to select the size of the buffer as $2N+1 = 64$ bytes using BK0. For example, if a buffer size of 128 is desired using BK0, the upper 16 bits of AMR are set to (0110)b = 0x0006.

If assembly code is used for the circular buffer, as execution returns to a calling C function, AMR needs to be reinitialized to the default linear mode. Hence the pointer's address must be saved.

TMS320C6x INSTRUCTION SET :

Assembly Code Format

An assembly code format is represented by the field

Label || [ ] Instruction Unit Operands ;comments

A label, if present, represents a specific address or memory location that contains an instruction or data. The label must be in the first column. The parallel bars (||) are there if the instruction is being executed in parallel with the previous instruction. The subsequent field is optional to make the associated instruction conditional. Five of the registers—A1, A2, B0, B1, and B2—are available to use as conditional registers. For example, [A2] specifies that the associated instruction executes if A2 is not zero. On the other hand, with [!A2], the associated instruction executes if A2 is zero. All C6x instructions can be made conditional with the registers A1, A2, B0, B1, and B2 by determining when the conditional register is zero.

The instruction field can be either an assembler directive or a mnemonic. An assembler directive is a command for the assembler. For example,
.word value
reserves 32 bits in memory and fill with the specified *value*. A mnemonic is an actual instruction that executes at run time. The instruction (mnemonic or assembler directive) cannot start in column 1. The Unit field, which can be one of the eight CPU units, is optional. Comments starting in column 1 can begin with either an asterisk or a semicolon, whereas comments starting in any other columns must begin with a semicolon. Code for the floating-point processors C3x/C4x is not compatible with code for the fixed-point processors C1x, C2x, and C5x/C54x. However, the code for the fixed-point processors C62x is compatible with the code for the floating-point C67x.

C62x code is actually a subset of C67x code. Additional instructions to handle double-precision and floating-point operations are available only on the C67x processor. Also, some additional instructions are available only on the fixed point C64x processor. Several code segments are presented to illustrate the C6x instruction set. Assembly code for the C6x processors is very similar to C3x/C4x code. Single-task types of instructions available for the C6x make it easier to program than either the previous generation of fixed- or floating-point processors.

This contributes to an efficient
compiler. Additional instructions available on the C64x (but not on the C62x) resemble the multitask types of instructions for C3x/C4x processors, AppendixA contains a list of the instructions for the C62x/C67x processors.

Types of Instructions :

The following illustrates some of the syntax of assembly code. It is optional to specify the eight functional units, although this can be useful during debugging and for code efficiency and optimization.

1. ***Add/Subtract/Multiply***
(a) The instruction
ADD .L1 A3,A7,A7 ; add A3 + A7 A7 (accum in A7)
adds the values in registers A3 and A7 and places the result in register A7. The unit .L1 is optional. If the destination or result is in B7, the unit would be .L2.
(b) The instruction
SUB .S1 A1,1,A1 ;subtract 1 from A1
subtracts 1 from A1 to decrement it using the .S unit.

(c) The parallel instructions
MPY .M2 A7,B7,B6 ; multiply 16 LSBs of A7, B7 Æ B6
|| **MPYH .M1 A7,B7,A6 ;**multiply 16MSBs of A7, B7 ÆA6

multiplies the lower or least significant 16 bits (LSBs) of both A7 and B7 and places the product in B6, in parallel (concurrently within the same
execution packet) with a second instruction that multiplies the higher or most significant 16 bits (MSBs) of A7 and B7 and places the result in A6.
In this fashion, two MAC operations can be executed within a single instruction cycle. This can be used to decompose a sum of products into two sets of sum of products: one set using the lower 16 bits to operate on the first, third, fifth, . . . number and another set using the higher 16 bits to operate on the second, fourth, sixth, . . . number. Note that the parallel
symbol is not in column 1.

2. ***Load/Store***
(a) The instruction
LDH .D2 *B2++,B7 ;load (B2) Æ B7, increment B2
|| **LDH .D1 *A2++,A7 ;**load (A2) Æ A7, increment A2
loads into B7 the half-word (16 bits) whose address in memory is specified/ pointed to by B2.Then register B2 is incremented (postincremented) to point at the next higher memory address. In parallel is another indirect addressing mode instruction to load into A7 the content in memory whose address is specified by A2. Then A2 is incremented to point at the next higher memory address.
The instruction LDW loads a 32-bit word. Two paths using .D1 and .D2 allow for the loading of data from memory to registers A and B using the instruction LDW.The double-word load floating-point instruction LDDW on the C6713 can simultaneously load two 32-bit registers into side A and two 32-bit registers into side B.
(b) The instruction
STW .D2 A1,*+A4[20] ;store A1Æ(A4) offset by 20 stores the 32-bit word A1 in memory whose address is specified by A4 offset by 20 words (32 bits) or 80 bytes. The address register A4 is preincremented with offset, but it is not modified (two plus signs are used if A4 is to be modified).
3. ***Branch/Move*****.**
The following code segment illustrates branching and data
transfer:
Loop MVKL .S1 x,A4 ;move 16 LSBs of x address ÆA4
MVKH .S1 x,A4 ;move 16 MSBs of x address ÆA4
.
SUB .S1 A1,1,A1 ;decrement A1
[A1] B .S2 Loop ;branch to Loop if A1 # 0
NOP 5 ;five no-operation instructions

STW .D1 A3,*A7 ;store A3 into (A7)
The first instruction moves the lower 16 bits (LSBs) of address *x* into register A4.The second instruction moves the higher 16 bits (MSBs) of address *x* into A4, which now contains the full 32-bit address of *x*. One must use the instructions
MVKL/MVKH in order to get a 32-bit constant into a register. Register A1 is used as a loop counter. After it is decremented with the SUB instruction, it is tested for a conditional branch. Execution branches to the label or address Loop if A1 is not zero. If A1 = 0, execution continues and data in register A3 are stored in memory whose address is specified (pointed) by A7.

ASSEMBLER DIRECTIVES :

An assembler directive is a message for the assembler (not the compiler) and is not an instruction. It is resolved during the assembling process and does not occupy memory space, as an instruction does. It does not produce executable code. Addresses of different sections can be specified with assembler directives. For example, the assembler directive *.sect "my_buffer"* defines a section of code or data named *my_buffer*.
The directives *.text* and *.data* indicate a section for text and data, respectively. Other assembler directives, such as *.ref* and *.def*, are used for undefined and defined symbols, respectively.The assembler creates several sections indicated by directives such as *.text* for code and *.bss* for global and static variables.

Other commonly used assembler directives are :

1. .**short:** to initialize a 16-bit integer.
2. **.int**: to initialize a 32-bit integer (also .word or .long).The compiler treats a long data value as 40 bits, whereas the C6x assembler treats it as 32 bits.
3. **.float:** to initialize a 32-bit IEEE single-precision constant.
4. **.double:** to initialize a 64-bit IEEE double-precision constant.

Initialized values are specified by using the assembler directives .byte, .short, or .int. Uninitialized variables are specified using the directive .usect, which creates an uninitialized section (like the .bss section), whereas the directive .sect creates an initialized section. For example, .usect "variable", 128 designates an uninitialized section named variable with a section size of 128 in bytes.

LINEAR ASSEMBLY :

An alternative to C, or assembly code, is linear assembly. An assembler optimizer (in lieu of a C compiler) is used in conjunction with a linear assembly-coded source program (with extension *.sa*) to create an assembly source program (with extension *.asm*) in much the same way that a C compiler optimizer is used in conjunction with a C-coded source program. The resulting assembly-coded program produced by the assembler optimizer is typically more efficient than one resulting from the C compiler optimizer. The assembly-coded program resulting from either a C-coded source program or a linear-assembly source program must be assembled to produce an object code.

Linear assembly code programming provides a compromise between coding effort and coding efficiency.The assembler optimizer assigns the functional unit and register to use (optional to be specified by the user), finds instructions that can execute in parallel, and performs software pipelining for optimization
Two programming examples at the end of this chapter illustrate a C program calling a linear assembly function. Parallel instructions are not valid in a linear assembly program. Specifying the functional unit is optional in a linear assembly program as well as in an assembly program.

In recent years, the C compiler optimizer has become more and more efficient. Although C code is less efficient (speed performance) than assembly code, it typically involves less coding effort than assembly code, which can be hand optimized to achieve 100 percent efficiency but with much greater coding effort. It is interesting to note that the C6x assembly code syntax is not as complex as that of the C2x/C5x or the C3x family of processors. It is actually simpler to “program” the C6x in assembly.

For example, the C3x instruction
DBNZD AR4,LOOP
decrements (due to the first D) a loop counter AR4 and branches (B) conditionally (if AR4 is nonzero) to the address specified by LOOP, with delay (due to the second D). The branch instruction with delay effectively allows the branch instruction to execute in a single cycle (due to pipelining). Such multitask instructions are not available on the C62x and C67x processors, although they were recently introduced on the C64x processor. In fact, C6x types of instructions are simpler. For example, separate instructions are available for decrementing a counter (with a SUB instruction) and branching.The simpler types of instructions are more amenable for a more efficient C compiler. However, although it is simpler to program in assembly code to perform a desired task, this does not imply or translate into an efficient assembly-coded program.

It can be relatively difficult to hand-optimize a program to yield a totally efficient (and meaningful) assembly-coded program. Linear assembly code is a cross between assembly and C. It uses the syntax of assembly code instructions such as ADD, SUB, and MPY, but with operands/registers as used in C. In some cases this provides a good compromise between C and assembly.

Linear assembler directives include
.cproc
.endproc
to specify a C-callable procedure or section of code to be optimized by the assembler optimizer.Another directive, *.reg*, is used to declare variables and use descriptive names for values that will be stored in registers.

INTERRUPTS :

An interrupt can be issued internally or externally. An interrupt stops the current CPU process so that it can perform a required task initiated by the interrupt. The program flow is redirected to an ISR. The source of the interrupt can be an ADC, a timer, and so on. On an interrupt, the conditions of the current process must be saved so that they can be restored after the interrupt task is performed. On interrupt, registers are saved and processing continues to an ISR. Then the registers are restored.

There are 16 interrupt sources. They include two timer interrupts, four external interrupts, four McBSP interrupts, and four DMA interrupts.Twelve CPU interrupts (INT4–INT11) are available. An interrupt selector is used to choose among the 12 interrupts.

Interrupt Control Registers :

The interrupt control registers are as follows:
1. CSR (control status register): contains the global interrupt enable (GIE) bit and other control/status bits
2. IER (interrupt enable register): enables/disables individual interrupts
3. IFR (interrupt flag register): displays the status of interrupts
4. ISR (interrupt set register): sets pending interrupts
5. ICR (interrupt clear register): clears pending interrupts
6. ISTP (interrupt service table pointer): locates an ISR
7. IRP (interrupt return pointer)
8. NRP (nonmaskable interrupt return pointer)
Interrupts are prioritized, with Reset having the highest priority.

The reset interrupt and nonmaskable interrupt (NMI) are external pins that have the first and second highest priority, respectively. The interrupt enable register (IER) is used to set a specific interrupt and can check if and which interrupt has occurred from the interrupt flag register (IFR).
NMI is nonmaskable, along with Reset. NMI can be masked (disabled) by clearing the nonmaskable interrupt enable (NMIE) bit within CSR. It is set to zero only upon reset or upon a nonmaskable interrupt. If NMIE is set to zero, all interrupts INT4 through INT15 are disabled.

The reset signal is an active-low signal used to halt the CPU, and the NMI signal alerts the CPU to a potential hardware problem.Twelve CPU interrupts with lower priorities are available, corresponding to the maskable signals INT4 through INT15. The priorities of these interrupts are: INT4, INT5, . . . , INT15, with INT4 having the highest priority and INT15 the lowest priority. For an NMI to occur, the NMIE bit must be 1 (active high).

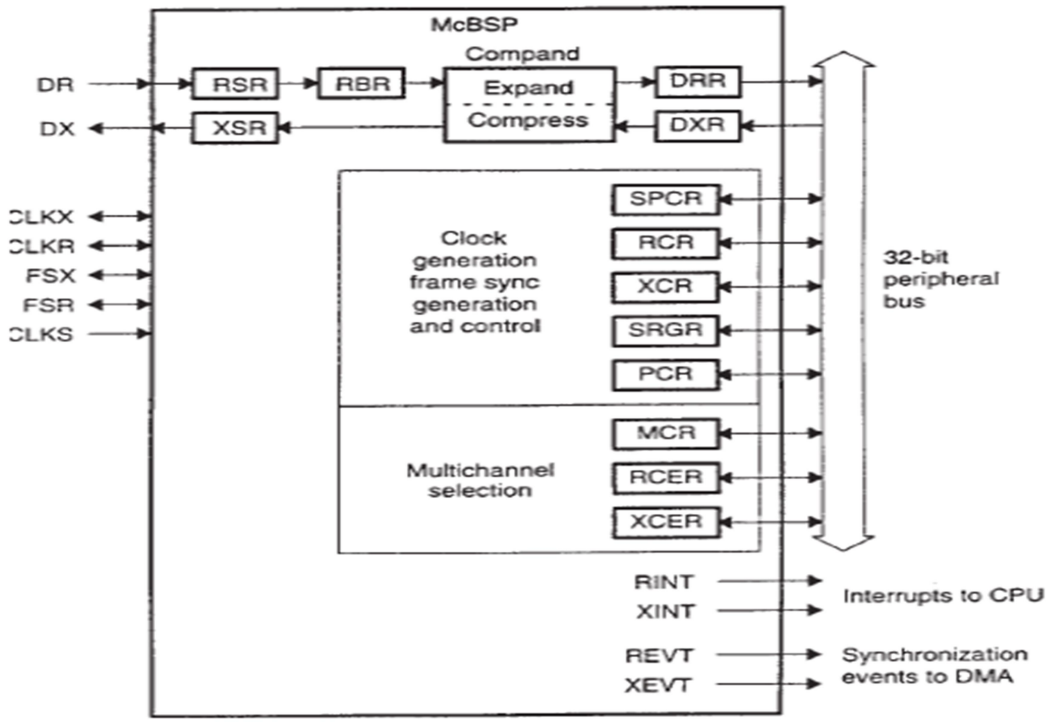
On reset (or after a previously set NMI), the NMIE bit is cleared to zero so that a reset interrupt may occur. To process a maskable interrupt, the GIE bit within the control status register (CSR) and the NMIE bit within the IER are set to 1. GIE is set to 1 with bit 0 of CSR set to 1, and NMIE is set to 1 with bit 1 of IER set to 1. Note that CSR can be ANDed with -2 (using 2's complement, the LSB is 0, while all other bits are 1's) to set the GIE bit to 0 and disable maskable interrupts globally. The interrupt enable (IE) bit corresponding to the desirable maskable interrupt is also set to 1.When the interrupt occurs, the corresponding IFR bit is set to 1 to show the interrupt status.

Interrupt Service Table :

Interrupt	Offset
RESET	000h
NMI	020h
Reserved	040h
Reserved	060h
INT4	080h
INT5	0A0h
INT6	0C0h
INT7	0E0h
INT8	100h
INT9	120h
INT10	140h
INT11	160h
INT12	180h
INT13	1A0h
INT14	1C0h
INT15	1E0h

MULTICHANNEL BUFFERED SERIAL PORTS :

Two McBSPs are available.They provide an interface to inexpensive (industry standard) external peripherals. McBSPs have features such as full-duplex communication, independent clocking and framing for receiving and transmitting, and direct interface to AC97 and IIS compliant devices.They allow several data sizes between 8 and 32 bits.



URE 3.4. Internal block diagram of McBSP (Courtesy of Texas Instruments).

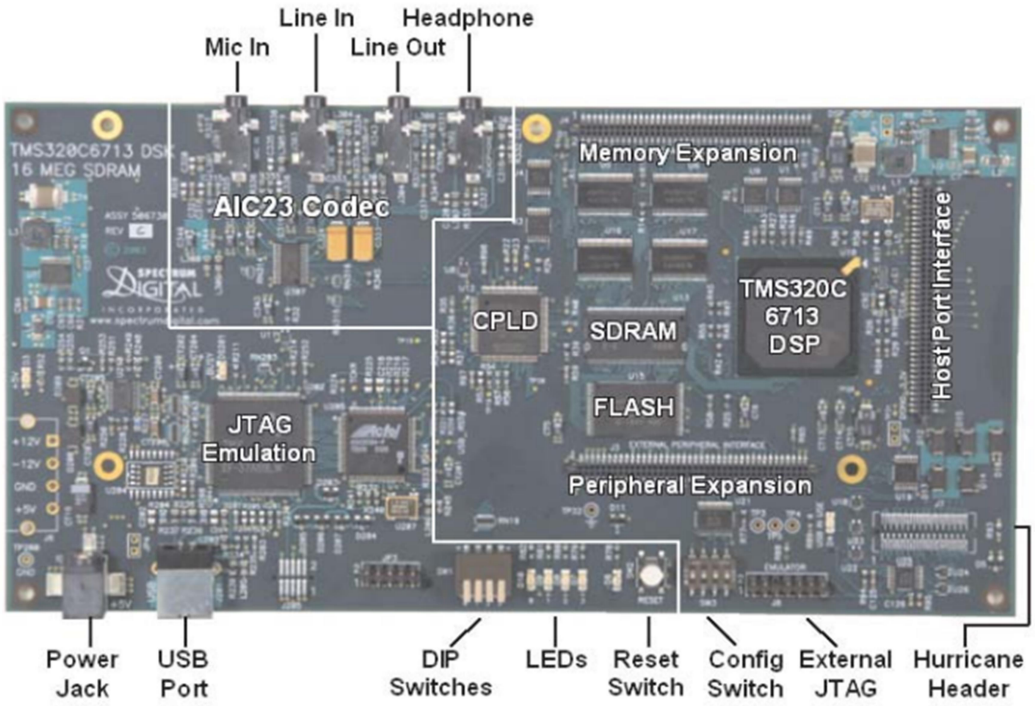
Clocking and framing associated with the McBSPs for input and output
are External data communication can occur while data are being moved internally.

Figure 3,4shows an internal block diagram of a McBSP. The data transmit (DX) and data receive (DR) pins are used for data communication. Control information (clocking and frame synchronization) is through CLKX, CLKR, FSX, and FSR.The CPU or DMA controller reads data from the data receive register (DRR) and writes data to be transmitted to the data transmit register (DXR). The transmit shift register (XSR) shifts these data to DX.The receive shift register (RSR) copies the data received on DR to the receive buffer register (RBR). The data in RBR are then copied to DRR to be read by the CPU or the DMA controller.

Other registers—the serial port control register (SPCR), receive/transmit control register (RCR/XCR), receive/transmit channel enable register (RCER/XCER), pin control register (PCR), and sample rate generator register (SRGR)—support further data communication [7].

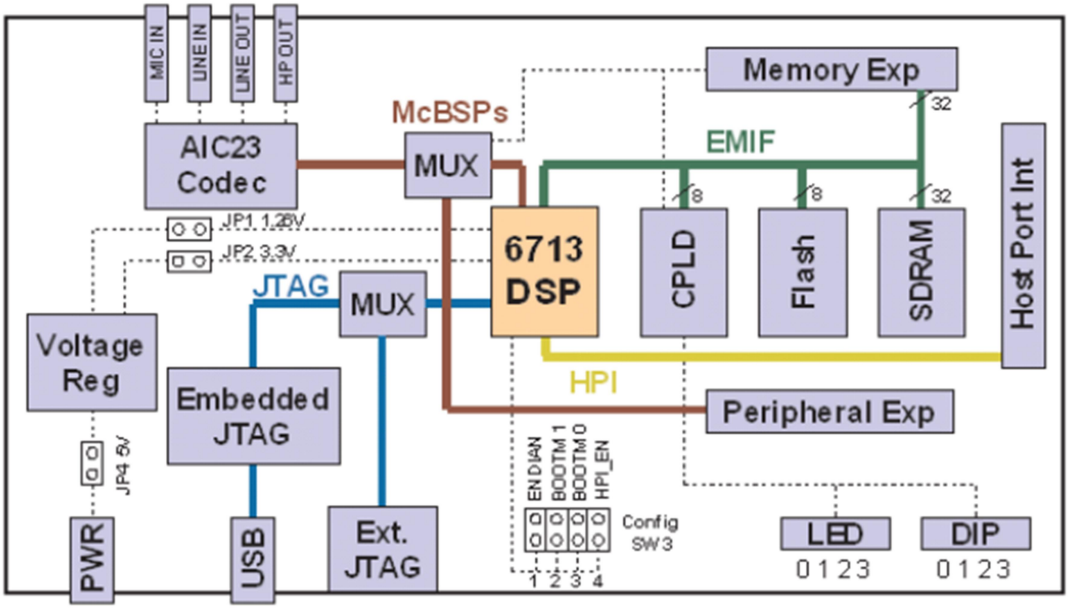
The two McBSPs are used for input and output through the onboard codec. McBSP0 is used for control and McBSP1 for transmitting and receiving data.

BLOCK DIGRAM OF TMS320C67xx DSP STARTER KIT AND SUPPORT TOOLS :

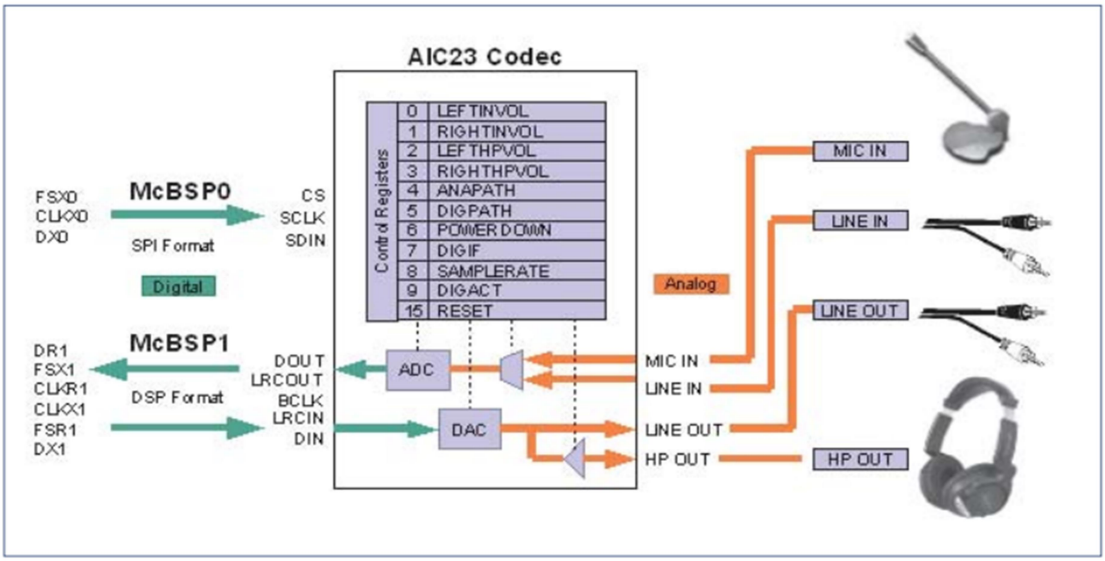


DSP TMS320C67XX STARTER KIT

BLOCK DIAGRAM OF DSP TMS320C67XX STARTER KIT :



Block diagram of TMS320C67XX DSP Starter Kit



AIC23 CODEC

TMS320C67XX Digital Signal Processor :

- The TMS320C6713 (C6713) is based on the VLIW architecture, which is very well suited for numerically intensive algorithms.
- The internal program memory is structured so that a total of eight instructions can be fetched every cycle.
- For example, with a clock rate of 225MHz, the C6713 is capable of fetching eight 32-bit instructions every 1/(225 MHz) or 4.44 ns.
- Features of the C6713 include 264 kB of internal memory (8kB as L1P and L1D Cache and 256kB as L2 memory shared between program and data space), eight functional or execution units composed of six arithmetic-logic units (ALUs) and two multiplier units, a 32-bit address bus to address 4 GB (gigabytes), and two sets of 32-bit general-purpose registers.
- The C67xx (such as the C6701, C6711, and C6713) belong to the family of the C6x floating-point processors, whereas the C62xx and C64xx belong to the family of the C6x fixed-point processors. The C6713 is capable of both fixed- and floatingpoint processing.

CODE COMPOSER STUDIO :

- CCS provides an IDE to incorporate the software tools. CCS includes tools for code generation, such as a C compiler, an assembler, and a linker.
- It has graphical capabilities and supports real-time debugging. It provides an easy-to-use software tool to build and debug programs.
- The C compiler compiles a C source program with extension .c to produce an assembly source file with extension.*asm*.The assembler assembles an.*asm* source file to produce a machine language object file with extension.*obj*.
- The linker combines object files and object libraries as input to produce an executable file with extension.*out*.

CCS Installation and Support :

Use the USB cable to connect the DSK board to the USB port on the PC. Use the 5-V power supply included with the DSK package to connect to the +5-V power connector on the DSK to turn it on. Install CCS with the CD-ROM included withthe DSK, preferably using the *c:\C6713* structure (in lieu of *c:\ti* as the default). The CCS icon should be on the desktop as “C6713DSK CCS” and is used to launch CCS.The code generation tools (C compiler, assembler, linker) are used with CCS version 2.x.

CCS provides useful documentations included with the DSK package on the following (see the Help icon):

1. Code generation tools (compiler, assembler, linker, etc.)
2. Tutorials on CCS, compiler, RTDX
3. DSP instructions and registers
4. Tools on RTDX, DSP/basic input/output system (DSP/BIOS), and so on.

CCS installation

1. myprojects: a folder supplied only for your projects. All the folders in the
accompanying book CD should be placed within this subdirectory.
2. bin: contains many utilities.
3. docs: contains documentation and manuals.
4. c6000\cgtools: contains code generation tools.
5. c6000\RTDX: contains support files for real-time data transfer.
6. c6000\bios: contains support files for DSP/BIOS.
7. examples: contains examples included with CCS.
8. tutorial: contains additional examples supplied with CCS.

Useful Types of Files :

You will be working with a number of files with different extensions. They include:

1. file.pjt: to create and build a project named file
2. file.c: C source program
3. file.asm: assembly source program created by the user, by the C compiler, or by the linear optimizer
4. file.sa: linear assembly source program.The linear optimizer uses *file.sa* as input to produce an assembly program *file.asm*
5. file.h: header support file
6. file.lib: library file, such as the run-time support library file rts6700.lib
7. file.cmd: linker command file that maps sections to memory
8. file.obj: object file created by the assembler
9. file.out: executable file created by the linker to be loaded and run on the C6713 processor
10. file.cdb: configuration file when using DSP/BIOS

DSK Board :

The DSK board includes 16MB (megabytes) of synchronous dynamic random access memory (SDRAM) and 256kB (kilobytes) of flash memory. Four connectorson the board provide input and output: MIC IN for microphone input, LINE IN for line input, LINE OUT for line output, and HEADPHONE for a headphone output (multiplexed with line output). The status of the four user dip switches on the DSK board can be read from a program and provides the user with a feedback control interface.

DSK SUPPORT TOOLS :

To perform the experiments, the following tools are used:

1. *TI's DSP starter kit (DSK)*. The DSK package includes:
(a) *Code Composer Studio* (CCS), which provides the necessary software support tools. CCS provides an integrated development environment (IDE), bringing together the C compiler, assembler, linker, debugger, and so on.

2 DSP Development System

DSK Support Tools 3
(b) A board, shown in Figure 1.1, that contains the TMS320C6713 (C6713) floating-point digital signal processor as well as a 32-bit stereo codec for input and output (I/O) support.

(c) A universal synchronous bus (USB) cable that connects the DSK board to a PC.

(d) A 5V power supply for the DSK board.
2. *An IBM-compatible PC*.The DSK board connects to the USB port of the PC through the USB cable included with the DSK package.

3. *An oscilloscope, signal generator, and speakers*. A signal/spectrum analyzer is optional. Shareware utilities are available that utilize the PC and a sound card to create a virtual instrument such as an oscilloscope, a function generator, or a spectrum analyzer.

CEC337 - DSP ARCHITECTURE AND PROGRAMMING

UNIT IV IMPLEMENTATION OF DSP ALGORITHMS

DSP Development system, On-chip, and On-board peripherals of C54xx and C67xx DSP development boards, Code Composer Studio (CCS) and support files, Implementation of Conventional FIR, IIR, and Adaptive filters in TMS320C54xx/TMS320C67xx DSP processors for real-time DSP applications, Implementation of FFT algorithm for frequency analysis in real-time.

DSP DEVELOPMENT SYSTEM :

INTRODUCTION

Digital signal processors such as the TMS320C6x (C6x) family of processors are like fast special-purpose microprocessors with a specialized type of architecture and an instruction set appropriate for signal processing. The C6x notation is used to designate a member of **Texas Instruments' (TI)** TMS320C6000 family of digital signal processors.

The architecture of the C6x digital signal processor is very well suited for numerically intensive calculations. Based on a **very-long-instruction-word (VLIW) architecture**, the C6x is considered to be TI's most powerful processor. Digital signal processors are used for a wide range of applications, from communications and **controls to speech and image processing**. The general-purpose digital signal processor is dominated by applications in communications (cellular).

Applications embedded digital signal processors are dominated by consumer products.
They are found in **cellular phones, fax/modems, disk drives, radio, printers, hearing aids, MP3 players, high-definition television (HDTV), digital cameras, and so on.** These processors have become the products of choice for a number of consumer applications, since they have become very cost-effective.They can handle different tasks, since they can be reprogrammed readily for a different application. DSP techniques have been very successful because of the development of low-cost software and hardware support. **For example, modems and speech recognition can be less expensive using DSP techniques.**

DSP processors are concerned primarily with real-time signal processing. Realtime processing requires the processing to keep pace with some external event, whereas non-real-time processing has no such timing constraint.The external event to keep pace with is usually the analog input.Whereas analog-based systems with discrete electronic components such as resistors can be more sensitive to temperature changes, DSP-based systems are less affected by environmental conditions. DSP processors enjoy the advantages of microprocessors.

They are easy to use, flexible, and economical.A number of books and articles address the importance of digital signal processors for a number of applications [1–22]. Various technologies have been used for real-time processing, from fiberoptics for very high frequency to DSPs very suitable for the audio-frequency range. Common applications using these processors have been for frequencies from 0 to 96kHz. Speech can be sampled at 8 kHz (the rate at which samples are acquired), which implies that each value sampled is acquired at a rate of 1/(8 kHz) or 0.125ms. A commonly used sample rate of a compact disk is 44.1kHz.

Analog/digital (A/D)-based boards in the megahertz sampling rate range are currently available. The basic system consists of an analog-to-digital converter (ADC) to capture an input signal. The resulting digital representation of the captured signal is then processed by a digital signal processor such as the C6x and then output through a digital-to-analog converter (DAC). Also included within the basic system are a special input filter for anti-aliasing to eliminate erroneous signals and an output filter to smooth or reconstruct the processed output signal.

DSK SUPPORT TOOLS

Most of the work presented in this book involves the design of a program to implement a DSP application. To perform the experiments, the following tools are used:

1. ***TI's DSP starter kit (DSK)*****.** The DSK package includes:

(a) ***Code Composer Studio*** **(CCS),** which provides the necessary software support tools. CCS provides an integrated development environment (IDE), bringing together the C compiler, assembler, linker, debugger, and so on.

(b) A board, that contains the TMS320C6713 (C6713) floating-point digital signal processor as well as a 32-bit stereo codec for input and output (I/O) support.

(c) A universal synchronous bus (USB) cable that connects the DSK board to a PC.

(d) A **5V power supply** for the DSK board.

2. ***An IBM-compatible PC***.The DSK board connects to the USB port of the PC through the USB cable included with the DSK package.

3. ***An oscilloscope, signal generator, and speakers***. A signal/spectrum analyzer is optional. Shareware utilities are available that utilize the PC and a sound card to create a virtual instrument such as an oscilloscope, a function generator, or a spectrum analyzer.

DSK Board

The DSK package is powerful, yet relatively inexpensive **($395),** with the necessary hardware and software support tools for real-time signal processing [23–43]. It is a complete DSP system.The DSK board, with an **approximate size of 5X8 inches**., includes the C6713 floating-point digital signal processor and a 32-bit stereo codec TLV320AIC23 (AIC23) for input and output.

The onboard codec AIC23 [37] uses a **sigma–delta technology** that provides ADC and DAC. It **connects to a 12-MHz system clock.**Variable sampling rates from 8 to 96 kHz can be set readily. A daughter card expansion is also provided on the DSK board. Two 80-pin connectors provide for external peripheral and external memory interfaces.Two project examples illustrate the use of the **external memory interface (EMIF)** with **light-emitting diodes (LEDs) and liquid-crystal displays (LCDs) for spectrum display.**

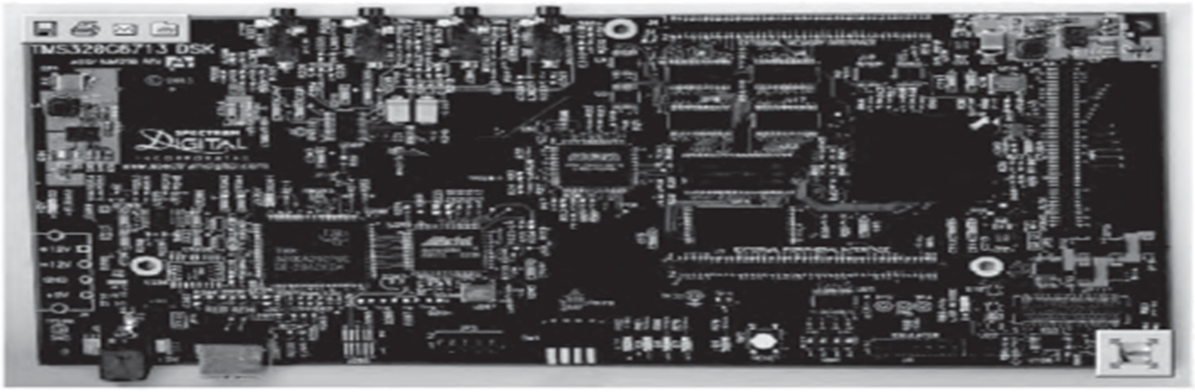
The DSK board includes **16MB (megabytes) of synchronous dynamic random access memory (SDRAM) and 256kB (kilobytes) of flash memory**. **Four connectors on the board provide input and output: MIC IN for microphone input, LINE IN for line input, LINE OUT for line output, and HEADPHONE for a headphone output (multiplexed with line output).** The status of the four user dip switches on the DSK board can be read from a program and provides the user with a feedback control interface.The DSK operates at 225MHz.Also onboard the DSK are voltage regulators that provide 1.26 V for the C6713 core and 3.3 V for its memory and peripherals.

TMS320C6713 Digital Signal Processor

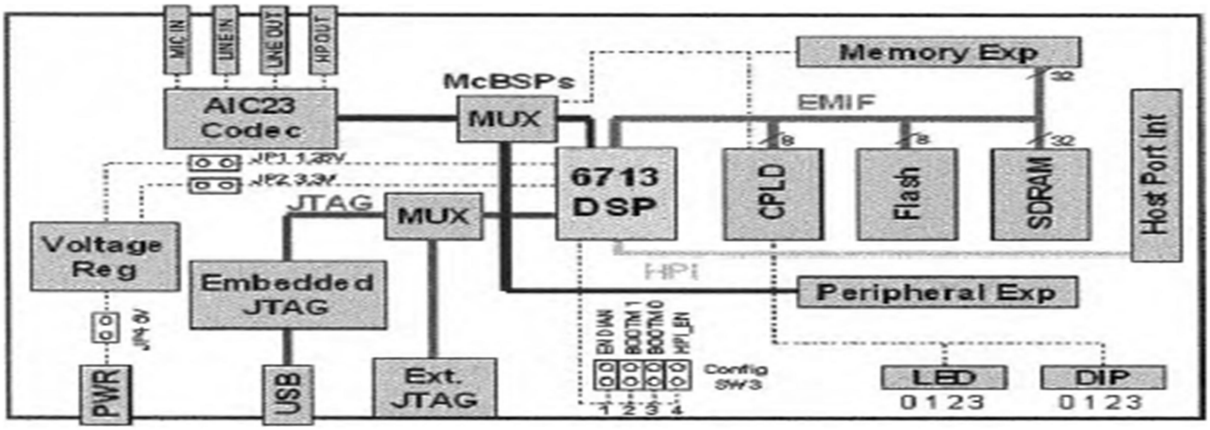
The TMS320C6713 (C6713) is based on the VLIW architecture, which is very well suited for numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. For example, with a clock rate of 225MHz, the C6713 is capable of fetching eight 32-bit instructions every 1/(225 MHz) or 4.44 ns.

Features of the C6713 include 264 kB of internal memory (8kB as L1P and L1D Cache and 256kB as L2 memory shared between program and data space), eight functional or execution units composed of six arithmetic-logic units (ALUs) and two multiplier units, a 32-bit address bus to address 4 GB (gigabytes), and two sets of 32-bit general-purpose registers.

The C67xx (such as the C6701, C6711, and C6713) belong to the family of the C6x floating-point processors, whereas the C62xx and C64xx belong to the family of the C6x fixed-point processors. The C6713 is capable of both fixed- and floatingpoint processing.



(a)



(b)

FIGURE 1.1. TMS320C6713-based DSK board: (*a*) board; (*b*) diagram. (Courtesy of Texas Instruments)

ON-CHIP PERIPHERALS OF C54xx AND C67xx DSP DEVELOPMENT BOARDS :

1. ***Timers and Counters***.
2. ***Serial communication interface***
   - i. **UART**
   - ii. **SPI**
   - iii. **I^2C**
3. ***ADC / DAC***.
4. ***DMA Controller***.
5. ***PWM***.
6. ***Watch dog Timers***.
7. ***Memory Interfaces***.
   - i. **SRAM**
   - ii. **DRAM**
   - iii. **EEPROM**
   - iv. **Flash memory**

1.Timers and Counters.

Digital signal processors typically have one or more on-chip timers that generate the hardware interrupt at periodic intervals. **They can be used to time or count events, generate pulses or interrupt the CPU.** The timers have two signaling modes, and can be clocked by an external clock or the CPU clock. By default they are clocked internally. The timer output can be configured as a timer output or a general-purpose output. When an internal clock drives the timer, the frequency on the timer input clock varies across the processor generations. the timer input frequency for different TI chips as a ratio of the CPU clocking rate.

2.Serial communication interface.

The SCI is constituted by three pins: **Receive data (RXD), transmit data (TXD) and the SCI serial clock (SCLK).** It provides a versatile connection to other units. Communication between the SCI and the DSP core is performed with memory mapped control and data registers.

i. **UART :** UART stands for **universal asynchronous receiver / transmitter** and defines a protocol, or set of rules, for exchanging serial data between two devices. UART is very simple and only uses two wires between transmitter and receiver to transmit and receive in both directions.

ii. **SPI : Serial Peripheral Interface (SPI)** is for synchronous serial communication, used primarily in embedded systems for short-distance wired communication between integrated circuits.

iii. **I^2C : Inter-Integrated Circuit**. It is a bus interface connection protocol incorporated into devices for serial communication. The I2C bus is a very popular and powerful bus used for communication between a master (or multiple masters) and a single or multiple slave devices.

3.ADC / DAC.

An **analog-to-digital converter (ADC)** is used to convert an analog signal such as voltage to a digital form so that it can be read and processed by a microcontroller. Most microcontrollers nowadays have built-in ADC converters. It is also possible to connect an external ADC converter to any type of microcontroller.
An **digital-to-analog (DAC**) is used correctly that converts digital signals into analog signals, ensuring accurate sound reproduction without affecting its properties. DSP optimizes the digital signal for the DAC's conversion process, resulting in the best possible output.

4.*DMA Controller*.

Direct memory access (DMA) is the process of transferring data without the involvement of the processor itself. It is often used for transferring data to/from input/output devices. A separate DMA controller is required to handle the transfer. The controller notifies the DSP processor that it is ready for a transfer.

***5.PWM*.**

The **Pulse-width modulator (PWM)** feature is very common in embedded systems. It provides a way to generate a pulse periodic waveform for motor control or can act as a digital-to-analog converter with some external components.

PWM method is used to reduce the harmonic content in the output voltage applied to induction motor. The pulses were generated using Texas Instruments TMS 320F28335 DSP controller and that triggers the inverter. Results of input and output of the inverter were captured using Digital Storage Oscilloscope.

6.Watch dog Timers.

A **watchdog timer (WDT)** is a timer that monitors microcontroller (MCU) programs to see if they are out of control or have stopped operating. It acts as a "watchdog" watching over MCU operation. A microcontroller (MCU) is a compact processor for controlling electronic devices.

***7.Memory Interfaces*.**

i.SRAM : Static random-access memory (static RAM or SRAM) is a type of random-access memory (RAM) that uses latching circuitry (flip-flop) to store each bit. SRAM is volatile memory; data is lost when power is removed.

ii.DRAM : DRAM (dynamic random access memory) is a type of semiconductor memory that is typically used for the data or program code needed by a computer processor to function. DRAM is a common type of random access memory (RAM) that is used in PCs, workstations and servers.

iii.EEPROM : EEPROM is an acronym that stands for Electrically Erasable Programmable Read-Only Memory. It denotes a type of rewritable storage chip or memory package that can continue to hold its stored information even without power. This is known as **non-volatile memory.** By using UV ultraviolet rays can erase the data**.**

iv.Flash memory : Flash memory is a long-life and non-volatile storage chip that is widely used in embedded systems. It can keep stored data and information even when the power is off. It can be electrically erased and reprogrammed.

ON-BOARD PERIPHERALS OF C54xx AND C67xx DSP DEVELOPMENT BOARDS :

1. ***JTAG Interface***.
2. ***Power supply Unit***.
3. ***Clock generator***.
4. ***Memory unit***.
5. ***Expansion headers*** .
6. ***GPIO***.
7. ***LED & Push buttons***.

1.JTAG Interface :

JTAG (named after the **Joint Test Action Group** which codified it) is an industry standard for verifying designs and testing printed circuit boards after manufacture. JTAG implements standards for on-chip instrumentation in electronic design automation (EDA) as a complementary tool to digital simulation.

JTAG/boundary-scan (IEEE Std 1149.1) is an electronic four port serial JTAG interface that allows access to the special embedded logic on a great many of today's ICs (chips) .

2.Power supply Unit :

A **5V power supply** for the DSK board.

3.Clock generator :

A clock signal generator is a circuit that produces a timing signal for use in synchronizing a system's operation. At its most basic level, a clock generator consists of a resonant circuit and an amplifier. A clock signal is produced by an electronic oscillator called a clock generator. The most common clock signal is in the form of a square wave with a 50% duty cycle. A simple technique for on-chip generation of a primary clock signal would be to use a ring oscillator Such a clock circuit has been used in low-end microprocessor chips. Simple on-chip clock generation circuit using a ring oscillator.

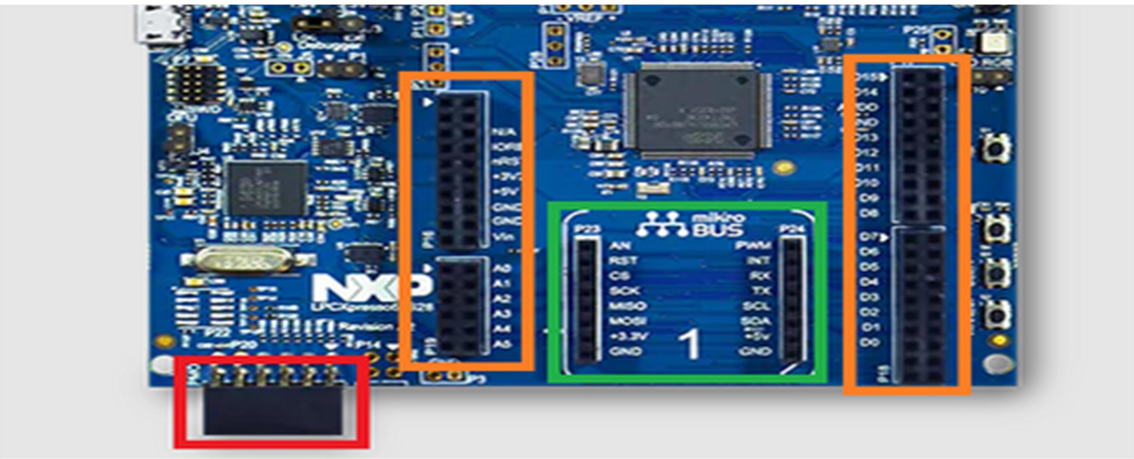
A free-running ring oscillator is used as internal clock and the output clock is generated using two counters. The clock generator is described in synthesisable VHDL-code and can therefore easily be made from standard cells found in any commercial standard CMOS cell library.

4.Memory unit :

They have various RAM and ROM configurations, a 16 bit I/O bus, and serial ports. The mid-range processor operates between 27-50 Mhz, with 16-32 bit floating point operations and 16-24 bit fixed point operations. A mid-range processor typically has around 32-40 bit registers.

5.Expansion headers:

An expansion header is a collection of expansion connectors placed on the development board. The pins from the connectors lead out the processor pins outside and can be identified by a label on the board. There can be multiple expansion headers on a single board.



6.GPIO :

GPIO stands for General Purpose Input/Output. It's a standard interface used to connect microcontrollers to other electronic devices. For example, it can be used with sensors, diodes, displays, and System-on-Chip modules.

7.LED & Push buttons :

LED stands for light emitting diode. LED lighting products produce light up to 90% more efficiently than incandescent light bulbs. How do they work? An electrical current passes through a microchip, which illuminates the tiny light sources we call LEDs and the result is visible light.

A push-button (also spelled pushbutton) or simply button is a simple switch mechanism to control some aspect of a machine or a process. Buttons are typically made out of hard material, usually plastic or metal.

CODE COMPOSER STUDIO (CCS) AND SUPPORT FILES :

CODE COMPOSER STUDIO (CCS) :

CCS provides an IDE to incorporate the software tools. **CCS includes tools for code generation, such as a C compiler, an assembler, and a linker**. It has graphical capabilities and supports real-time debugging. It provides an easy-to-use software tool to build and debug programs.

The C compiler compiles a C source program with extension .c to produce an assembly source file with extension.*asm*.The assembler assembles an.*asm* source file to produce a machine language object file with extension.*obj*.The linker combines object files and object libraries as input to produce an executable file with extension.*out*.

This executable file represents a linked common object file format (COFF), popular in Unix-based systems and adopted by several makers of digital signal processors [25]. This executable file can be loaded and run directly on the C6713 processor. the linear assembly source file with extension .sa, which is a cross between C and assembly code. A linear optimizer optimizes this source file to create an assembly file with extension .asm (similar to the task of the C compiler).

To create an application project, one can "add" the appropriate files to theproject. Compiler/linker options can readily be specified. A number of debuggingfeatures are available, including setting breakpoints and watching variables; viewing memory, registers, and mixed C and assembly code; graphing results; and monitoring execution time. One can step through a program in different ways (step into, over, or out).

Real-time analysis can be performed using real-time data exchange (RTDX).RTDX allows for data exchange between the host PC and the target DSK, as well as analysis in real time without stopping the target. Key statistics and performance can be monitored in real time. Through the **joint team action group (JTAG**), communication with on-chip emulation support occurs to control and monitor program execution. The C6713 DSK board includes a JTAG interface through the USB port.

CCS Installation and Support

Use the USB cable to connect the DSK board to the USB port on the PC. Use the 5-V power supply included with the DSK package to connect to the +5-V power connector on the DSK to turn it on. Install CCS with the CD-ROM included with the DSK, preferably using the *c:\C6713* structure (in lieu of *c:\ti* as the default).

The CCS icon should be on the desktop as "C6713DSK CCS" and is used to launch CCS.The code generation tools (C compiler, assembler, linker) are used with **CCS version 2.x.**

CCS provides useful documentations included with the DSK package on the following (see the Help icon):

1. Code generation tools (compiler, assembler, linker, etc.)
2. Tutorials on CCS, compiler, RTDX
3. DSP instructions and registers
4. Tools on RTDX, DSP/basic input/output system (DSP/BIOS), and so on.

An extensive amount of support material (*pdf* files) is included with CCS.There are also examples included with CCS within the folder *c:\C6713\examples*. They illustrate the board and chip support library files, DSP/BIOS, and so on. CCS Version 2.x was used to build and test the examples included in this book.A number of files included in the following subfolders/directories within *c:\C6713* (suggested structure during CCS installation) can be very useful:

1. *myprojects*: a folder supplied only for your projects. All the folders in the accompanying book CD should be placed within this subdirectory.
2. *bin*: contains many utilities.
3. *docs*: contains documentation and manuals.
4. *c6000\cgtools*: contains code generation tools.
5. *c6000\RTDX*: contains support files for real-time data transfer.
6. *c6000\bios*: contains support files for DSP/BIOS.
7. *examples*: contains examples included with CCS.
8. *tutorial*: contains additional examples supplied with CCS.

Useful Types of Files :

You will be working with a number of files with different extensions. They include:

1. file.pjt: to create and build a project named file
2. file.c: C source program
3. file.asm: assembly source program created by the user, by the C compiler,or by the linear optimizer
4. file.sa: linear assembly source program.The linear optimizer uses *file.sa* as input to produce an assembly program *file.asm*
5. file.h: header support file
6. file.lib: library file, such as the run-time support library file rts6700.lib
7. file.cmd: linker command file that maps sections to memory
8. file.obj: object file created by the assembler
9. file.out: executable file created by the linker to be loaded and run on the C6713 processor
10. file.cdb: configuration file when using DSP/BIOS

SUPPORT FILES :

The following support files located in the folder support (except the library files) are used for most of the examples and projects:

1. C6713dskinit.c: contains functions to initialize the DSK, the codec, the serial ports, and for I/O. It is not included with CCS.

2. C6713dskinit.h: header file with function prototypes. Features such as those used to select the mic input in lieu of line input (by default), input gain, and so on are obtained from this header file (modified from a similar file included with CCS).

3. C6713dsk.cmd: sample linker command file. This generic file can be changed when using external memory in lieu of internal memory.

4. Vectors_intr.asm: a modified version of a vector file included with CCS to handle interrupts. Twelve interrupts, INT4 through INT15, are available, and INT11 is selected within this vector file.They are used for interrupt-driven programs.

5. Vectors_poll.asm: vector file for programs using polling.

6. rts6700.lib,dsk6713bsl.lib,csl6713.lib: run-time, board, and chip support library files, respectively. These files are included with CCS and are located in *C6000\cgtools\lib*, *C6000\dsk6713\lib*, and *c6000\bios\lib*, respectively.

On-Chip Peripherals in C54x DSP Processor :

The following on-chip peripherals are available on C54x devices:

1. *General-purpose I/O pins: XF and BIO*
2. *Timer*
3. *Clock generator*
4. *Host port interface (HPI)*
   i. *8-bit standard*
   ii. *8-bit enhanced*
   iii. *16-bit enhanced*
5. *Synchronous serial port*
6. *Buffered serial port (BSP)*
7. *Multichannel buffered serial port (McBSP)*
8. *Time-division multiplexed (TDM) serial port*
9. *Software-programmable wait-state generator*
10. *Programmable bank-switch*

1. **General-Purpose I/O** :

The C54x DSP offers general-purpose I/O through two dedicated pins that are software controlled. The two dedicated pins are the branch control input pin (BIO) and the external flag output pin (XF).

i. **Branch Control Input Pin (BIO)** : BIO can be used to monitor the status of peripheral devices. It is especially useful as an alternative to using an interrupt when time-critical loops must not be disturbed. A branch can be conditionally executed dependent upon the state of the BIO input. Of the instructions that use BIO, the execute conditionally (XC) instruction samples the condition of BIO during the decode phase of the pipeline; all other conditional instructions (branch, call, and return) sample BIO during the read phase of the pipeline.

ii. **External Flag Output Pin (XF)** : XF can be used to signal external devices. The XF pin is controlled using software. It is driven high by setting the XF bit (in ST1) and is driven low by clearing the XF bit. The set status register bit (SSBX) and reset status register bit (RSBX) instructions can be used to set and clear XF, respectively. XF is also set high at device reset. Figure 8–1 shows the relationship between the time the SSBX or RSBX instruction is fetched and the time the XF pin is set or reset (refer to the TMS320C54x DSP data sheet for timing specifications). The XF timing shown is for a sequence of single-cycle instructions. Actual timing can vary with different instruction sequences.

2. **Timer :**

The on-chip timer is a software-programmable timer that consists of three registers and can be used to periodically generate interrupts. The timer resolution is the CPU clock rate of the processor.The high dynamic range of the timer is achieved with a 16-bit counter with a 4-bit prescaler. The C5402 and the C5420 have two on-chip timers.

3. Clock Generator :

The clock generator allows system designers to select the clock source. The sources that drive the clock generator are:

- A crystal resonator with the internal oscillator circuit. The crystal resonator circuit is connected across the X1 and X2/CLKIN pins of the C54x DSP. The CLKMD pins must be configured to enable the internal oscillator.
- An external clock. The external clock source is directly connected to the X2/CLKIN pin, and X1 is left unconnected.

The clock generator on the C54x devices consists of an internal oscillator and a phase-locked loop (PLL) circuit. Currently, there are two different types of PLL circuits on C54x devices. Some devices have hardware-configurable PLL circuits while others have software-programmable PLL circuit.

4.Host Port Interface :

The standard host port interface (HPI) is available on the C542, C545, C548, and C549 devices. The HPI is an 8-bit parallel port that interfaces a host device or host processor to the C54x DSP.

Information is exchanged between the C54x DSP and the host device through on-chip C54x DSP memory that is accessible by both the host and the C54x DSP. Enhanced host port interfaces are available on the C5402, C5410 (HPI-8), and C5420 (HPI-16) devices. This chapter does not describe these enhanced HPIs

The HPI interfaces to the host device as a peripheral, with the host device as master of the interface, facilitating ease of access by the host. The host device communicates with the HPI through dedicated address and data registers, to which the C54x DSP does not have direct access, and the HPI control register, using the external data and interface control signals Both the host device and the C54x DSP have access to the HPI control register.

5.Synchronous serial port :

In all C54x DSP serial ports, both receive and transmit operations are doublebuffered, thus allowing a continuous communications stream with either 8-bit or 16-bit data packets. The continuous mode provides operation that, once initiated, requires no further frame synchronization pulses (FSR and FSX) when transmitting at maximum packet frequency.

The serial ports are fully static and thus will function at arbitrarily low clocking frequencies. The maximum operating frequency for the standard serial port of one-fourth of CLKOUT (10 Mbit/s at 25 ns, 12.5 Mbit/s at 20 ns) is achieved when using internal serial port clocks. The maximum operating frequency for the BSP is CLKOUT. When the serial ports are in reset, the device may be configured to turn off the internal serial port clocks, allowing the device to run in a lower power mode of operation.

6.Buffered Serial Port (BSP) Interface :

The buffered serial port (BSP) is made up of a full-duplex, double-buffered serial port interface, which functions in a similar manner to the C54x DSP standard serial port, and an autobuffering unit (ABU)

The serial port section of the BSP is an enhanced version of the C54x DSP standard serial port. The ABU is an additional section of logic which allows the serial port section to read/write directly to C54x DSP internal memory independent of the CPU. This results in a minimum overhead for serial port transfers and faster data rates.

7.Time-Division Multiplexed (TDM) Serial Port Interface :

The time-division multiplexed (TDM) serial port allows the C54x DSP to communicate serially with up to seven other devices. The TDM port, therefore, provides a simple and efficient interface for multiprocessing applications. By means of the TDM bit in the TDM serial port control register (TSPC), the port can be configured in multiprocessing mode (TDM = 1) or stand-alone mode (TDM = 0).

When in stand-alone mode, the port operates as described in section 9.2. When in multiprocessing mode, the port operates as described in this section. The port can be shut down for low power consumption via the XRST and RRST bits,

8.Wait-State Generator :

The software-programmable wait-state generator can extend external bus cycles by up to seven machine cycles (14 machine cycles on C5402, C5409, C5410, and C5420 devices), providing a convenient means to interface the C54x DSP to slower external devices. Devices that require more than seven wait states can be interfaced using the hardware READY line.

When all external accesses are configured for zero wait states, the internal clocks to the wait-state generator are shut off. Shutting off these paths from the internal clocks allows the device to run with lower power consumption. The software-programmable wait-state generator is controlled by the 16-bit software wait-state register (SWWSR), which is memory-mapped to address 0028h in data space.

9.Bank-Switching Logic :

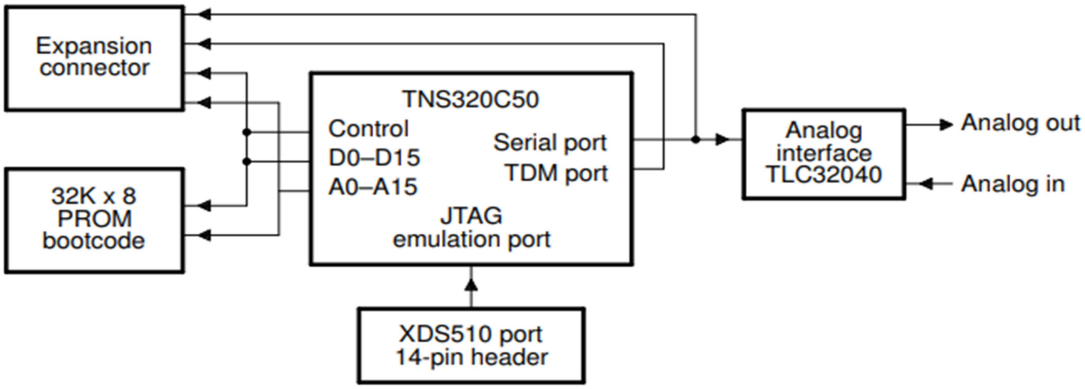
Programmable bank-switching logic allows the C54x DSP to switch between external memory banks without requiring external wait states for memories that need several cycles to turn off. The bank-switching logic automatically inserts one cycle when accesses cross memory-bank boundaries inside program or data space.

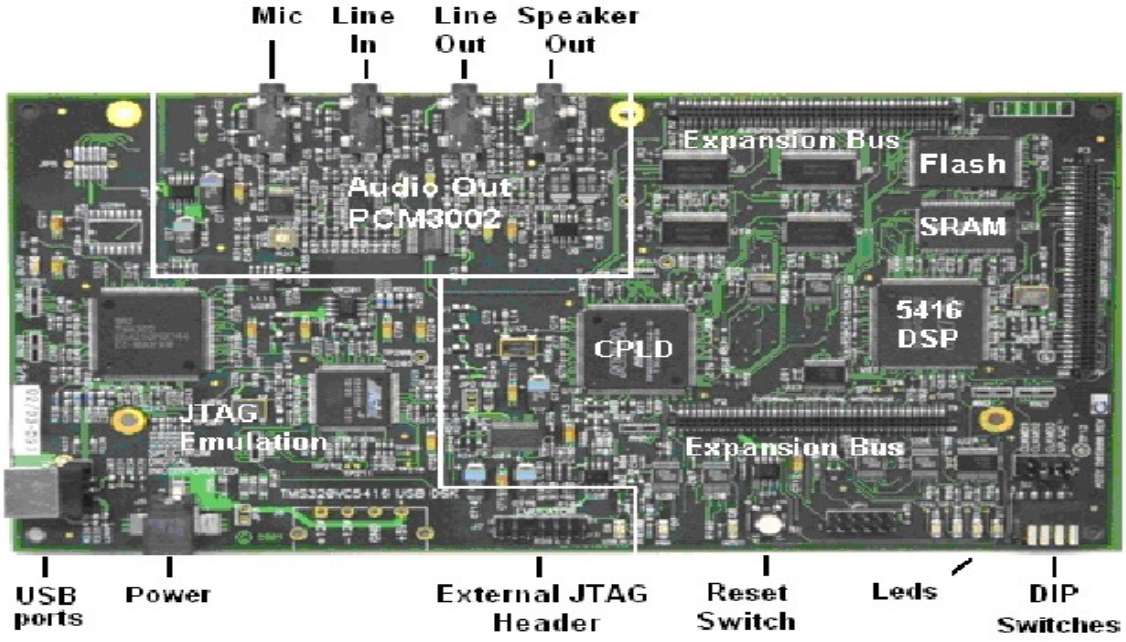
UNIT – II

BLOCK DIAGRAM OF TMS320C54XX DSP STARTER KIT :

Depicts the basic block diagram of the ’C50. It shows the interconnections, which include the host interface, analog interface, and emulation interface. PC communications are via the RS-232 port on the DSK board. The 32K bytes of PROM contain the kernel program for boot loading. All pins of the ’C50 are connected to the external I/O interfaces. The external I/O interfaces include four 24-pin headers, a 4-pin header, and a 14-pin XDS510 header. The TLC32040 AIC interfaces to the ’C50 serial port. Two RCA connectors provide analog input and output on the board.

Figure 1–1. ’C5x DSK Block Diagram





Example 4.10: FIR Implementation Using Two Different Methods (`fir2ways`)

Figure 4.34 shows a listing of the program `fir2ways.c`, which implements an FIR filter using two alternative methods for convolving and updating the delay samples. This example extends Example 4.3, in which the first method (method A) is used. In this first method, using two for loops, the delay samples are stored in the *N*-element array `dly` with the newest sample at the beginning of the buffer `dly[0]` and the oldest sample at the end of the buffer `dly[N-1]`. The convolution starts with the newest sample and the first filter coefficient using

$$y(n) = h(0)x(n) + h(1)x(n-1) + \cdots + h(N-1)x(n-(N-1))$$

In a second for loop, each sample value in array `dly` is shuffled such that, for example, the sample value `dly[i]` is shifted to become `dly[i+1]`.

The second method (method B) uses pointers to implement a circular buffer in array `dly`. In this case, the samples stored in the array are not shuffled or moved. Method B performs the convolution using one for loop.

Build and run this project as **`fir2ways`**. Verify that an FIR bandpass filter centered at 1 kHz is implemented. Change the method used, by editing the line in program `fir2ways.c` that reads

```
#define method 'A'
```

and verify that the resulting filter characteristic is the same as before.

Example 4.11: Voice Scrambling Using Filtering and Modulation (`scrambler`)

This example illustrates a voice scrambling/descrambling scheme. The approach makes use of basic algorithms for filtering and modulation. Modulation was introduced in the AM example in Chapter 2.

With voice as input, the resulting output is scrambled voice. The original descrambled voice is recovered when the output of the DSK is used as the input to a second DSK running the same program.

The scrambling method used is commonly referred to as frequency inversion. It takes an audio range, in this case 300 Hz to 3 kHz, and "folds" it about a 3.3-kHz carrier signal. The frequency inversion is achieved by multiplying (modulating) the audio input by a carrier signal, causing a shift in the frequency spectrum with upper and lower sidebands. In the lower sideband that represents the audible speech range, the low tones are high tones, and vice versa.

Figure 4.35 is a block diagram of the scrambling scheme. At point A we have an input signal, bandlimited to 3 kHz. At point B we have a double-sideband signal

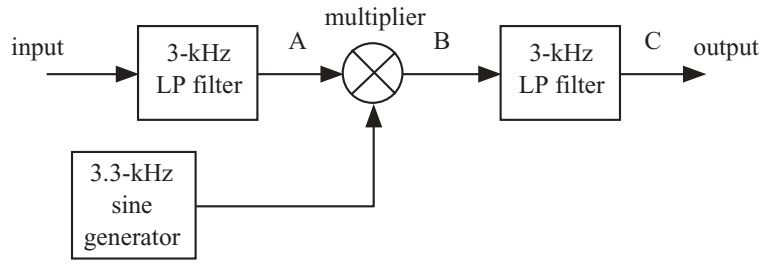


FIGURE 4.35. Block diagram of scrambler system.

with suppressed carrier. At point C the upper sideband and the section of the lower sideband between 3 and 3.3 kHz are filtered out. The scheme is attractive because of its simplicity. Only simple DSP algorithms—namely, filtering, sine wave generation, and amplitude modulation—are required for its implementation.

Figure 4.36 shows a listing of program `scrambler.c`, which operates at a sampling rate, `fs`, of 16 kHz. The input signal is first lowpass filtered using an FIR filter with 65 coefficients, `h`, defined in the file `lp3k64.cof`. The filtering algorithm used is identical to that used in, for example, program `fir.c`. The filter delay line is implemented using array `x1` and the output is assigned to variable `yn1`. The filter output (at point A in Figure 4.36) is multiplied (modulated) by a 3.3-kHz sinusoid stored as 160 samples (exactly 33 cycles) in array `sine160` (file `sine160.h`) . Finally, the modulated signal (at point B) is lowpass filtered again, using the same set of filter coefficients `h` (`lp3k64.cof`) but a different filter delay line implemented using array `x2` and the output variable `yn2`. The output is a scrambled signal (at point C). Using this scrambled signal as the input to a second DSK running the same algorithm, the original descrambled input is recovered as the output of the second DSK.

Build and run this project as **`scrambler`**. First, test the program using a 2-kHz sine wave as input. The resulting output is a lower sideband signal at 1.3 kHz. The upper sideband signal at 5.3 kHz is filtered out by the second lowpass filter. By varying the frequency of the sinusoidal input, you should be able to verify that input frequencies in the range 300–3000 Hz appear as output frequencies in the inverted range 3000 to 300 Hz.

A second DSK running the same program can be used to recover the original signal (simulating the receiving end). Use the output of the first DSK as the input to the second DSK.

Change the input source used by the program from LINE IN to MIC IN and test the scrambler and descrambler using speech from a microphone as the input. Run exactly the same program on each DSK, that is, including the line

```
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC
```

and connect LINE OUT on the first DSK (scrambler) to MIC IN on the second DSK (descrambler).

```
//scrambler.c

#include "DSK6713_AIC23.h"           // codec support
Uint32 fs=DSK6713_AIC23_FREQ_16KHZ;  //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_LINE; //select line in
#include "sine160.h"
#include "lp3k64.cof"                //filter coefficient file
float yn1, yn2;                      //filter outputs
float x1[N],x2[N];                   //filter delay lines
int index = 0;

interrupt void c_int11()
{
 short i;
                                     // first filter input
 x1[0]=(float)(input_left_sample()); //get input into delay line
 yn1 = 0.0;                          //initialise filter output
 for (i=0 ; i<N ; i++) yn1 += h[i]*x1[i];
 for (i=(N-1) ; i>0 ; i--) x1[i] = x1[i-1];
                                     // next mix with 3300Hz
 yn1 *= sine160[index++];
 if (index >= NSINE) index = 0;
                                     // now filter again
 x2[0] = yn1;                        //get input into delay line
 yn2 = 0.0;                          //initialise filter output
 for (i=0 ; i<N ; i++) yn2 += h[i]*x2[i];
 for (i=(N-1) ; i>0 ; i--) x2[i] = x2[i-1];
 output_left_sample((short)(yn2));   //output to codec
 return;
}

void main()
{
 comm_intr();                        //initialise McBSP, AD535
 while(1);                           //infinite loop
}
```

FIGURE 4.36. Scrambler program `scrambler.c`.

Interception of the speech signal could be made more difficult by changing the modulation frequency dynamically and by including (or omitting) the carrier frequency according to a predefined sequence: for example, a code for no modulation, another for modulating at frequency `fc1`, and a third code for modulating at frequency `fc2`.

This project was first implemented using the TMS320C25 [50] and also the TMS320C31 DSK.

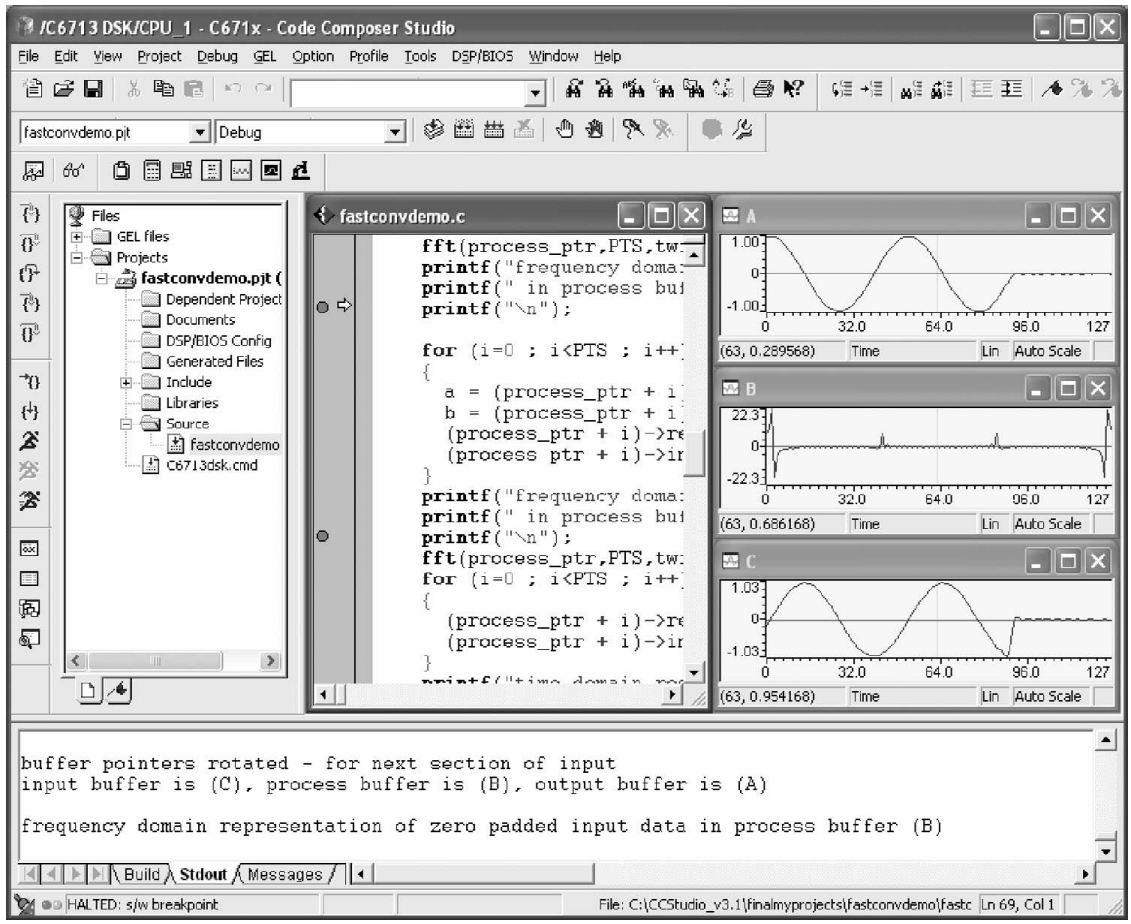


FIGURE 6.43. Code Composer window during execution of program `fastconvdemo.c`.

Example 6.13: Graphic Equalizer (`graphicEQ`)

Figure 6.45 shows a listing of the program `graphicEQ.c`, which implements a three-band graphic equalizer. TI's floating-point complex radix-2 FFT and IFFT support functions are used again in this project (see also Examples 6.5 and 6.6). The coefficient file `graphicEQcoeff.h` contains three sets of coefficients: lowpass at 1.3 kHz, bandpass between 1.3 and 2.6 kHz, and highpass at 2.6 kHz, designed with MATLAB's function `fir1`. Both the input samples and the three sets of coefficients are transformed into the frequency domain. The filtering is performed in the frequency domain based on the overlap-add scheme used in Examples 6.9–6.12 [15, 16]. Note that an alternative arrangement to the triple buffering used in those examples has been employed.

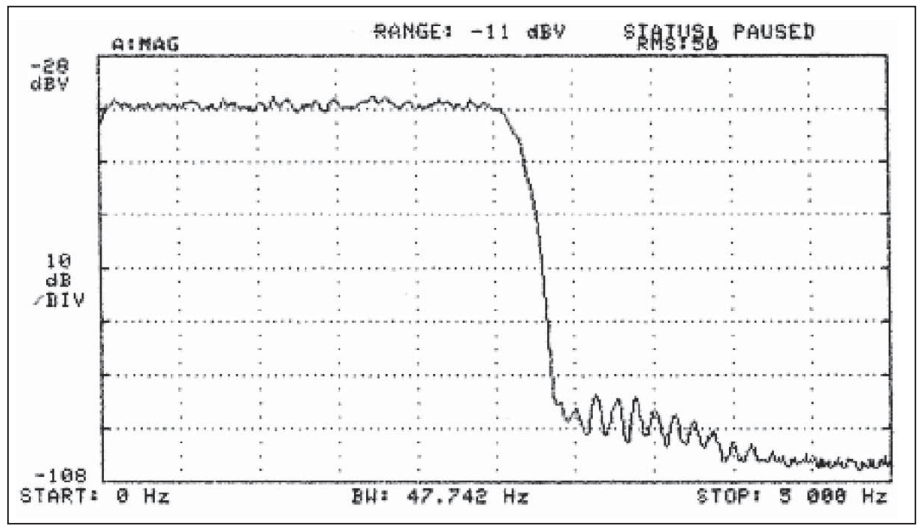
An array of *PTS/2* floating-point values, `iobuffer`, is used for both input and output. New input samples replace previously computed output samples as they are written to the DAC. Once `iobuffer` has been filled with *PTS/2* new input samples, these are copied to an intermediate buffer (array `samples`) and replaced by *PTS/2* output samples. Build this project as **`graphicEQ`** (use the optimization level -o1). Test the project using music or wideband noise as an input.

```
//graphicEQ.c Graphic Equalizer using TI floating-point FFT functions

#include "DSK6713_AIC23.h"          //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;        //set sampling rate
#include <math.h>
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_LINE; // select mic in
#include "GraphicEQcoeff.h"       //time-domain FIR coefficients
#define PI 3.14159265358979
#define PTS 256                   //number of points for FFT
//#define SQRT_PTS 16
#define RADIX 2
#define DELTA (2*PI)/PTS
typedef struct Complex_tag {float real,imag;} COMPLEX;
#pragma DATA_ALIGN(W,sizeof(COMPLEX))
#pragma DATA_ALIGN(samples,sizeof(COMPLEX))
#pragma DATA_ALIGN(h,sizeof(COMPLEX))
COMPLEX W[PTS/RADIX] ;            //twiddle array
COMPLEX samples[PTS];
COMPLEX h[PTS];
COMPLEX bass[PTS], mid[PTS], treble[PTS];
short buffercount = 0;            //buffer count for iobuffer samples
float iobuffer[PTS/2];            //primary input/output buffer
float overlap[PTS/2];             //intermediate result buffer
short i;                          //index variable
short flag = 0;                   //set to indicate iobuffer full
float a, b;                       //variables for complex multiply
short NUMCOEFFS = sizeof(lpcoeff)/sizeof(float);
short iTwid[PTS/2] ;
float bass_gain = 1.0;            //initial gain values
float mid_gain = 0.0;             //change with GraphicEQ.gel
float treble_gain = 1.0;

interrupt void c_int11(void)      //ISR
{
 output_left_sample((short)(iobuffer[buffercount]));
 iobuffer[buffercount++] = (float)((short)input_left_sample());
 if (buffercount >= PTS/2)        //for overlap-add method iobuffer
  {                               //is half size of FFT used
   buffercount = 0;
   flag = 1;
  }
}

main()
{
 digitrev_index(iTwid, PTS/RADIX, RADIX);
 for( i = 0; i < PTS/RADIX; i++ )
  {
   W[i].real = cos(DELTA*i);
   W[i].imag = sin(DELTA*i);
  }
 bitrev(W, iTwid, PTS/RADIX);     //bit reverse W

 for (i=0 ; i<PTS ; i++)
  {
   bass[i].real = 0.0;
```

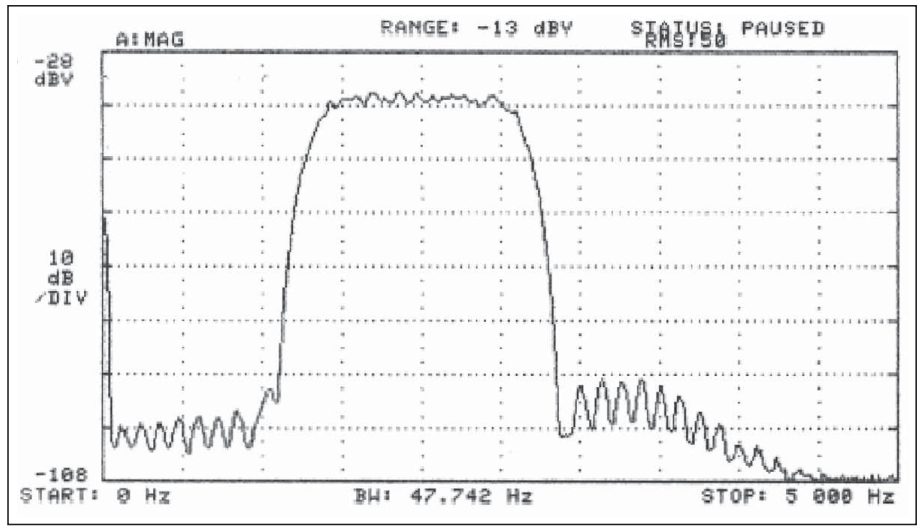
FIGURE 6.45. Equalizer program using TI's floating-point FFT support functions (graphicEQ.c).

```
   bass[i].imag = 0.0;
   mid[i].real = 0.0;
   mid[i].imag = 0.0;
   treble[i].real = 0.0;
   treble[i].imag = 0.0;
  }
 for (i=0; i<NUMCOEFFS; i++)      //same # of coeff for each filter
  {
   bass[i].real = lpcoeff[i];     //lowpass coeff
   mid[i].real =  bpcoeff[i];     //bandpass coeff
   treble[i].real = hpcoeff[i];   //highpass coef
  }

 cfftr2_dit(bass,W,PTS);          //transform each band
 cfftr2_dit(mid,W,PTS);               //into frequency domain
 cfftr2_dit(treble,W,PTS);

 comm_intr();                     //initialise DSK, codec, McBSP
 while(1)                         //frame processing infinite loop
  {
   while (flag == 0);             //wait for iobuffer full
          flag = 0;
   for (i=0 ; i<PTS/2 ; i++)      //iobuffer into samples buffer
    {
     samples[i].real = iobuffer[i];
     iobuffer[i] = overlap[i];    //previously processed output
    }                             //to iobuffer
   for (i=0 ; i<PTS/2 ; i++)
    {                             //upper-half samples to overlap
     overlap[i] = samples[i+PTS/2].real;
     samples[i+PTS/2].real = 0.0; //zero-pad input from iobuffer
    }
   for (i=0 ; i<PTS ; i++)
     samples[i].imag = 0.0;       //init samples buffer

   cfftr2_dit(samples,W,PTS);

   for (i=0 ; i<PTS ; i++)        //construct freq domain filter
    {                             //sum of bass,mid,treble coeffs
    h[i].real = bass[i].real*bass_gain + mid[i].real*mid_gain
              + treble[i].real*treble_gain;
    h[i].imag = bass[i].imag*bass_gain + mid[i].imag*mid_gain
              + treble[i].imag*treble_gain;
    }
   for (i=0; i<PTS; i++)          //frequency-domain representation
    {                             //complex multiply samples by h
     a = samples[i].real;
     b = samples[i].imag;
     samples[i].real = h[i].real*a - h[i].imag*b;
     samples[i].imag = h[i].real*b + h[i].imag*a;
    }

   icfftr2_dif(samples,W,PTS);

   for (i=0 ; i<PTS ; i++)
      samples[i].real /= PTS;
   for (i=0 ; i<PTS/2 ; i++)      //add 1st half to overlap
      overlap[i] += samples[i].real;
  }                               //end of infinite loop
}                                 //end of main()
```

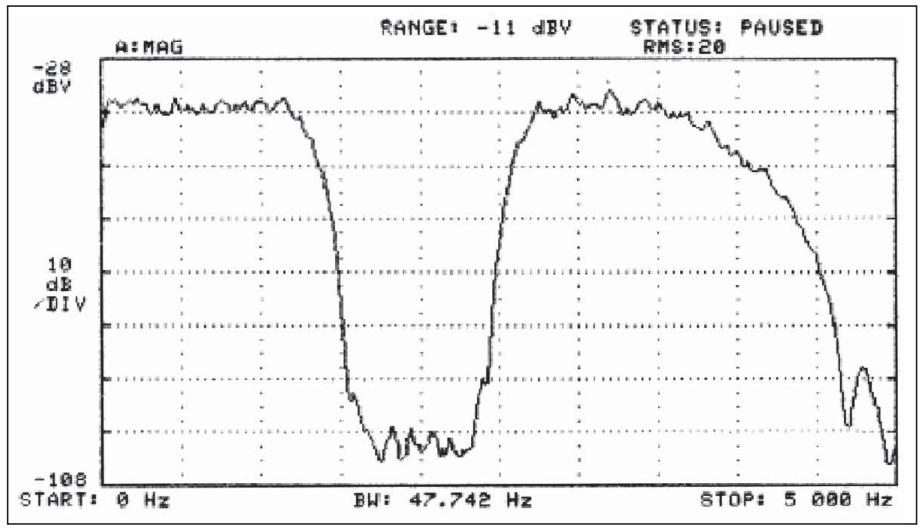
FIGURE 6.45. (*Continued*)



(a)



(b)



(c)

FIGURE 6.46. Output spectrum of a graphic equalizer obtained with a signal analyzer: (*a*) `bass_gain = treble_gain = 1`, `mid_gain = 0`; (*b*) `bass_gain = treble_gain = 0`, `mid_gain = 1`; (*c*) `bass_gain = mid_gain = 1`, `treble_gain = 0`.

which is the difference between the desired signal $d(n)$ and the adaptive filter's output $y(n)$. The weights or coefficients $w_k(n)$ are adjusted such that a mean squared error function is minimized. This mean squared error function is $E[e^2(n)]$, where E represents the expected value. Since there are k weights or coefficients, a gradient of the mean squared error function is required. An estimate can be found instead using the gradient of $e^2(n)$, yielding

$$w_k(n+1) = w_k(n) + 2\beta e(n)x(n-k) \qquad k = 0, 1, \ldots, N-1 \tag{7.3}$$

which represents the LMS algorithm [1–3]. Equation (7.3) provides a simple but powerful and efficient means of updating the weights, or coefficients, without the need for averaging or differentiating, and will be used for implementing adaptive filters. The input to the adaptive filter is $x(n)$, and the rate of convergence and accuracy of the adaptation process (adaptive step size) is β.

For each specific time n, each coefficient, or weight, $w_k(n)$ is updated or replaced by a new coefficient, based on (7.3), unless the error signal $e(n)$ is zero. After the filter's output $y(n)$, the error signal $e(n)$ and each of the coefficients $w_k(n)$ are updated for a specific time n, a new sample is acquired (from an ADC) and the adaptation process is repeated for a different time. Note that from (7.3), the weights are not updated when $e(n)$ becomes zero.

The linear adaptive combiner is one of the most useful adaptive filter structures and is an adjustable FIR filter. Whereas the coefficients of the frequency-selective FIR filter discussed in Chapter 4 are fixed, the coefficients, or weights, of the adaptive FIR filter can be adjusted based on a changing environment such as an input signal. Adaptive IIR filters (not discussed here) can also be used. A major problem with an adaptive IIR filter is that its poles may be updated during the adaptation process to values outside the unit circle, making the filter unstable.

The programming examples developed later will make use of equations (7.1)–(7.3). In (7.3) we simply use the variable β in lieu of 2β.

7.2 ADAPTIVE STRUCTURES

A number of adaptive structures have been used for different applications in adaptive filtering.

1. *For noise cancellation.* Figure 7.2 shows the adaptive structure in Figure 7.1 modified for a noise cancellation application. The desired signal d is corrupted by uncorrelated additive noise n. The input to the adaptive filter is a noise n' that is correlated with the noise n. The noise n' could come from the same source as n but modified by the environment. The adaptive filter's output y is adapted to the noise n. When this happens, the error signal approaches the desired signal d. The overall output is this error signal and not the adaptive filter's output y. If d is uncorrelated with n, the strategy is to minimize $E(e^2)$,

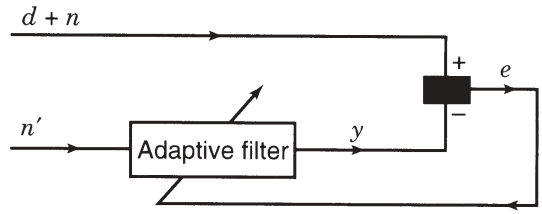


FIGURE 7.2. Adaptive filter structure for noise cancellation.

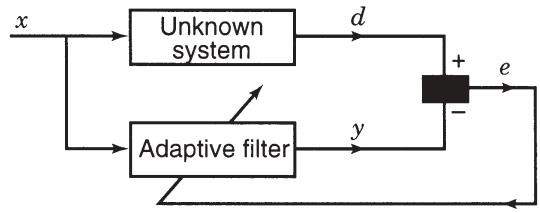


FIGURE 7.3. Adaptive filter structure for system identification.

where $E(\ )$ is the expected value. The expected value is generally unknown; therefore, it is usually approximated with a running average or with the instantaneous function itself. Its signal component, $E(d^2)$, will be unaffected and only its noise component $E[(n-y)^2]$ will be minimized. A more complete discussion is found in Widrow and Stearns [1]. This structure will be further illustrated with programming examples using C code.

2. *For system identification*. Figure 7.3 shows an adaptive filter structure that can be used for system identification or modeling. The same input is to an unknown system in parallel with an adaptive filter. The error signal e is the difference between the response of the unknown system d and the response of the adaptive filter y. This error signal is fed back to the adaptive filter and is used to update the adaptive filter's coefficients until the overall output $y = d$. When this happens, the adaptation process is finished, and e approaches zero. If the unknown system is linear and not time varying, then after the adaptation is complete, the filter's characteristics no longer change. In this scheme, the adaptive filter models the unknown system. This structure is illustrated later with three programming examples.
3. *Adaptive predictor*. Figure 7.4 shows an adaptive predictor structure that can provide an estimate of an input. This structure is illustrated later with a programming example.
4. Additional structures have been implemented, such as:
   (a) *Notch with two weights*, which can be used to notch or cancel/reduce a sinusoidal noise signal. This structure has only two weights or coefficients. It is shown in Figure 7.5 and is illustrated in Refs. 1, 3, and 4.

10

DSP Applications and Student Projects

This chapter can be used as a source of experiments, projects, and applications, demonstrating how the examples in earlier chapters can be combined and extended. It describes a number of applications and projects carried out by students (at Roger Williams University, the University of Massachusetts–Dartmouth, and at Worcester Polytechnic Institute). The descriptions are accompanied by program listings, not all of which are complete, but which are intended to serve as a starting point for development of further student projects.

Additional ideas for projects can be found in Refs. 1–6. A wide range of projects has been implemented on the floating-point C30 and C31 processors [7–21] as well as on the fixed-point TMS320C25 [22–28]. They range in topic from communications and controls to neural networks and also can be used as a source of ideas to implement other projects.

10.1 DTMF SIGNAL DETECTION USING CORRELATION, FFT, AND GOERTZEL ALGORITHM

This project implements the detection of a dual tone multifrequency (DTMF) tone and is decomposed into four smaller projects. The first miniproject uses a correlation scheme and displays the detected DTMF signals with the onboard LEDs. The second miniproject expands on the first one and uses RTDX that provides a PC–DSK interface to display on the PC monitor the detected DTMF signals by the C6x on the DSK. The third miniproject uses the FFT to estimate the DTMF signals. The

Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK, Second Edition By Rulph Chassaing and Donald Reay

TABLE 10.1 DTMF Encoding

Frequencies	1209 Hz	1336 Hz	1477 Hz
697 Hz	1	2	3
770 Hz	4	5	6
852 Hz	7	8	9
941 Hz	*	0	#

fourth miniproject uses Goertzel's algorithm and implements the DTMF detection on the C6416 DSK (can be transported readily to the C6713 DSK). The complete executable files for all four subprojects are included on the CD.

A DTMF signal consists of two sinusoidal signals: one from a group (row) of four low frequencies and the other from a group (column) of three high frequencies. This is illustrated in Table 10.1. When a key is pressed from a telephone, a DTMF signal is generated. For example, pressing button 6 generates a tone consisting of the summation of the two tones with frequencies of 770 and 1477 Hz, as shown in Table 10.1. For easier detection, these frequencies are chosen so that the sum or difference of any two frequencies does not equal that of any of the other frequencies.

Various schemes can be used to decode DTMF signals:

1. A correlation scheme, as described in this first miniproject. An RTDX option in the second miniproject provides a PC–DSK interface displaying the dialed (received) numbers on the PC screen.
2. The FFT (or the DFT) to detect the signals corresponding to the DTMF tones. The FFT is used in the third miniproject to estimate the weights associated with the seven frequencies.
3. Use of a bank of FIR filters so that each filter passes only one of the frequencies. The average power at the output of two of these filters should be larger than that at the other outputs, yielding the corresponding DTMF tone (not used in this project).
4. Use of Goertzel's algorithm [2, 22, 28, 29] in lieu of the FFT or DFT since only two frequencies need be detected/selected. This method (see Appendix F) can be more efficient than the FFT when a "small" number of spectrum points are required rather than the entire spectrum.

Each DTMF signal can be represented as

$$u(t) = A(\sin(\omega_1 t + \varphi_1) + \sin(\omega_2 t + \varphi_2))$$

where ω_1 and ω_2 are the two frequencies that need to be determined, and φ_1 and φ_2 are unknown phases. Frequency f_1 is one of the following frequencies: 697, 770, 852, or 941 Hz; and frequency f_2 is one of the following frequencies: 1209, 1336, or 1477 Hz [30, 31].

10.1.1 Using a Correlation Scheme and Onboard LEDs for Verifying Detection

The correlation scheme is as follows. Let the input signal be $u(t) = A(\sin(2\pi 697t + \varphi_1) + \sin(2\pi 1209t + \varphi_2))$. Since the input signal includes $\sin(2\pi 697t + \varphi_1)$, the correlation of the input signal with $\sin(2\pi 697t + \varphi_1)$ must be higher than the correlations with $\sin(2\pi 770t + \varphi_1)$, $\sin(2\pi 852t + \varphi_1)$, and $\sin(2\pi 941t + \varphi_1)$. The Fourier transform $\int u(t)e^{-j\omega t}dt$ has a peak at 697 Hz. Using Euler's formula for the exponential function, it becomes a correlation of $u(t)$ with sine and cosine functions. As a result, the input frequency can be determined by correlating the input signal with the sine and cosine for each possible frequency. The algorithm is as follows:

1. For each frequency, find the following correlations:

$$W_{\sin 697} = \sum_{n=1}^{N} u(t_n)\sin(2\pi 697 t_n), \quad W_{\cos 697} = \sum_{n=1}^{N} u(t_n)\cos(2\pi 697 t_n)$$
$$\cdots$$
$$W_{\sin 1477} = \sum_{n=1}^{N} u(t_n)\sin(2\pi 1477 t_n), \quad W_{\cos 1477} = \sum_{n=1}^{N} u(t_n)\cos(2\pi 1477 t_n)$$

2. For each frequency, find the maximum between sine weight and cosine weight:

$$W_{697} = \max(|W_{\sin 697}|, |W_{\cos 697}|)$$
$$\cdots$$
$$W_{1477} = \max(|W_{\sin 1477}|, |W_{\cos 1477}|)$$

3. Among the first four weights, choose the largest one; and among the last three weights, choose the largest one:

$$W_1 = \max(|W_{697}|, |W_{770}|, |W_{852}|, |W_{941}|)$$
$$W_2 = \max(|W_{1209}|, |W_{1336}|, |W_{1477}|)$$

4. The frequencies present in the input signal can then be obtained. If both W_1 and W_2, are larger than a threshold, turn on the appropriate LEDs corresponding to each character, as shown in Table 10.2.

Figure 10.1 shows the C source program `partial_dtmf.c` that can be completed readily. Build this project as DTMF. You can test this project first since the complete executable file `DTMF.out` is included on the CD in the folder DTMF. It can be tested using one of the following:

1. A phone to create the DTMF signals and a microphone to capture these signals as input to the DSK's mic input. Alternatively, a microphone with the

TABLE 10.2 Characters and Corresponding LEDs

1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
*	1010
0	1011
#	1100

```
//DTMF.c Core program to decode DTMF signals and turn on LEDs
#define N   100
#define thresh 40000
short i;short buffer[N]; short sin697[N],cos697[N],sin770[N],cos770[N];
...
long weight697,weight697_sin,weight697_cos; long ...weight1477_cos;
long weight1,weight2,choice1,choice2;
interrupt void c_int11()
{
 for (i = N-1; i > 0; i--)
      buffer[i]=buffer[i-1];                    // initialize buffer
 buffer[0] = input_sample();                    //input into buffer
 output_sample(buffer[0]*10);                   //output from buffer
 weight697_sin=0;  weight697_cos=0;             //weight @ each freq
 ...
 weight1477_sin = 0;  weight1477_cos =  0;
 for (i = 0; i < N; i++)
 {
  weight697_sin = weight697_sin + buffer[i]*sin697[i];
  weight697_cos = weight697_cos + buffer[i]*cos697[i];
 ...
  weight1477_cos= weight1477_cos + buffer[i]*cos1477[i];
 }
 //for each freq compare sine and cosine weights and choose largest
 if(abs(weight697_sin)>abs(weight697_cos))   weight697=abs(weight697_sin);
 else weight697 = abs(weight697_cos);
 ...
 if(abs(weight1477_sin)>abs(weight1477_cos)) weight1477 = abs(weight1477_sin);
 else weight1477 = abs(weight1477_cos);
 weight1=weight697; choice1=1;//among weight697,..weight941->largest
 if(weight770 > weight1) {weight1 = weight770; choice1=2;} //...
 if(weight941 > weight1) {weight1 = weight941; choice1=4;}
 weight2=weight1209; choice2=1;//among weight1209,..weight1477->largest
 if(weight1336> weight2) {weight2 = weight1336; choice2=2;}
```

FIGURE 10.1. Core C program using correlation to detect DTMF tones (`partial_dtmf.c`).

```
 if(weight1477> weight2) {weight2 = weight1477; choice2=3;}
 if((weight1>thresh)&&(weight2>thresh)) //set threshhold
 {  // depending on choices1 and 2 turn on corresponding LEDs
 if((choice1 == 1)&&(choice2 == 1)) { //button "1" -> 0001
   DSK6713_LED_off(0);DSK6713_LED_off(1);DSK6713_LED_off(2);DSK6713_LED_on(3);}
 ... //for button "2","3",..,"*","0"
 if((choice1 == 4)&&(choice2 == 3))  //button "#" -> 1100
   {DSK6713_LED_on(0);DSK6713_LED_on(1);DSK6713_LED_off(2);DSK6713_LED_off(3);}
 }          //end of if > threshold
 else { //weights below threshold, turn LEDs off
  DSK6713_LED_off(0);DSK6713_LED_off(1);DSK6713_LED_off(2);DSK6713_LED_off(3);}
 return;
}
void main()
{
DSK6713_LED_init();
DSK6713_LED_off(0);DSK6713_LED_off(1);DSK6713_LED_off(2);DSK6713_LED_off(3);
for (i = 0; i < N; i++)   //define sine/cosine for all 7 frequencies
 {
  buffer[i]=0;
  sin697[i]=1000*sin(2*3.14159*i/8000.*697);
  cos697[i]=1000*cos(2*3.14159*i/8000.*697);
  ...
  cos1477[i]=1000*cos(2*3.14159*i/8000.*1477);
 }
 comm_intr();  while(1); //init, infinite loop
}
```

FIGURE 10.1. (*Continued*)

necessary pre-amp can be used and connected directly to the line input on the DSK. For the threshold value set in the program, use 1,000,000 with the microphone input option. Dial a few numbers and verify the corresponding LEDs turning on based on the number detected.

2. Figure 10.2 shows the core of the MATLAB program *partial_dtmf.m* that generates/plays DTMF signals as input to the DSK. This program can be completed readily. Verify that all 12 DTMF signals 0, 1, . . . , # are consecutively generated by the MATLAB program, each lasting approximately 1.5 s. Also verify that the corresponding LEDs on the DSK are turned on for each detected DTMF signal. For the line input, use a threshold value of 40,000 in the program.
3. A tone generator using DialpadChameleon (can be downloaded from the web). This provides a pad with keys to generate short DTMF signals that can be used as input to the DSK.

The length of the signal affects the reliability of detection. If the buffer size is too small, the probability of turning on the wrong LEDs increases because of the uncertainty in frequency associated with short signals. If the buffer is too long, it complicates the detection near the transmission points. The Dialpad signals have the shortest duration.

the (`.wav`) chirp signal and verify that the results are identical to those achieved with the spectrogram in Figure 10.14b, being continuously updated within MATLAB. The file `vc_spectrogramdlg.cpp` contains the MATLAB commands for plotting the spectrogram. However, MATLAB is not used in this version to provide the RTDX link.

As in Section 10.7.2, you can obtain a fast and accurate plot by deleting the commands for including the title and the labels within the spectrogram plot. These commands are in the file `vc_spectrogramdlg.cpp`.

You can extend this project version using the radix-2 FFT (in lieu of the radix-4). Chapter 6 includes several examples based on the radix-2 FFT.

10.8 AUDIO EFFECTS (ECHO AND REVERB, HARMONICS, AND DISTORTION)

This project illustrates various audio effects such as distortion, echo and reverb, and harmonics [35]. Figure 10.16 shows the core program `soundboard.c` (virtually complete) that implements this project. The overall program flow consists of preamplification, distortion, echo/reverb, harmonics, and postamplification. Preamp and postamp are included to avoid overdriving the output. A sampling rate of 16 kHz is chosen, and a total of 10 sliders are used for the overall control. The slider gel file is on the CD in the folder `soundboard`.

The distortion effect is the simplest to implement. It requires overamplifying each sample and clipping it at maximum and minimum values. The acquired input sample is amplified based on whether it is positive or negative. The amplification polynomial used for the distortion component is used to amplify the signal in a nonlinear fashion. The result is scaled by a distortion magnitude controlled by a slider, then clipped so as not to overdrive the output.

The resulting output is processed for an echo/reverb effect (see Examples 2.4 and 2.5 on echo effects). The length of the echo is controlled by changing the buffer size where the samples are stored. A dynamic change of the echo length leads to a reverb effect. A fading effect with a decaying echo is obtained with a slider.

The third effect is harmonics boost. A harmonics buffer is used for this effect. Two main loop sections are created to produce two separate sets of harmonics. The larger (outer) loop combines the input with samples from the harmonics buffer at twice the input frequency. The smaller (inner) loop produces the next harmonics at four times the input frequency. The magnitudes of the harmonics are controlled with a slider.

These effects were tested successfully using the input from a keyboard with the keyboard output to a speaker. The audio output is sent to both channels of the codec (see Example 2.9), using the stereo capability of the onboard codec. The executable and gel files are included in the folder **`soundboard`**.

A drum effect section is included in the program for expanding the project. The use of external memory must be considered when applying many effects.

```
//Soundboard.c  Core C program for sound effects
union {Uint32 uint; short channel[2];} AIC23_data;
union {Uint32 uint; short channel[2];} AIC23_input;
short EchoLengthB = 8000;           //echo delay
short EchoBuffer[8000];             //create buffer
short echo_type = 1;                //to select echo or delay
short Direction = 1;                //1->longer echo,-1->shorter
short EchoMin=0,EchoMax=0;          //shortest/longest echo time
short DistMag=0,DistortionVar=0,VolSlider=100,PreAmp=100,DistAmp=10;
short HarmBuffer[3001];             //buffer
short HarmLength=3000;              //delay of harmonics
float output2;
short DrumOn=0,iDrum=0,sDrum=0;     //turn drum sound when = 1
int   DrumDelay=0,tempo=40000;      //delay counter/drum tempo
short ampDrum=40;                   //volume of drum sound
..                                  //addtl casting
interrupt void c_int11()            //ISR
{
AIC23_input.uint = input_sample();  //newest input data
input=(short)(AIC23_input.channel[RIGHT]+AIC23_input.channel[LEFT])/2;
input = input*.0001*PreAmp*PreAmp;
output=input;
output2=input;                      //distortion section
if (output2>0)
output2=0.0035*DistMag*DistMag*DistMag*((12.35975*(float)input)
        - (0.359375*(float)input*(float)input));
else  output2 =0.0035*DistMag*DistMag*DistMag*(12.35975*(float)input
        + 0.359375*(float)input*(float)input);
output2/=(DistMag+1)*(DistMag+1)*(DistMag+1);
if (output2 > 32000.0)  output2 = 32000.0 ;
else if (output2 < -32000.0 )  output2 = -32000.0;
output= (output*(1/(DistMag+1))+output2); //overall volume slider
input = output;                           //echo/reverb section
iEcho++;                                  //increment buffer count
if (iEcho >= EchoLengthB) iEcho = 0;      //if end of buffer reinit
output=input + 0.025*EchoAmplitude*EchoBuffer[iEcho];//newest+oldest
if(echo_type==1) EchoBuffer[iEcho] = output; //for decaying echo
else EchoBuffer[iEcho]=input;             //for single echo (delay)
EchoLengthB += Direction;                 //alter the echo length
if(EchoLengthB<EchoMin+100){Direction=1;} //echo delay is shortest->
if(EchoLengthB>EchoMax){Direction=-1;}    //longer,if longest->shorter
input=output;                             //output echo->harmonics gen
if(HarmBool==1) {                         //everyother sample...
 HarmBool=0;                              //switch the count
 HarmBuffer[iHarm]=input;                 //store sample in buffer
 if(HarmBool2==1){                        //everyother sample...
  HarmBool2=0;                            //switch the count
  HarmBuffer[uHarm] += SecHarmAmp*.025*input;//store sample in buffer
 }
 else{HarmBool2=1; uHarm++;               //or just switch the count,
      if(uHarm>HarmLength) uHarm=0;       //and increment the pointer
 }
}
```

FIGURE 10.16. Core C program to obtain various audio effects (`soundboard.c`).

```
else{HarmBool=1; iHarm++;                    //or just switch the count
if(iHarm>HarmLength) iHarm=0;}               //and increment the pointer
output=input+HarmAmp*0.0125*HarmBuffer[jHarm];//add harmonics to output
jHarm++;                                     //and increment the pointer
if(jHarm>HarmLength) jHarm=0;                //reinit when maxed out
DrumDelay--;                                 //decrement delay counter
if(DrumDelay<1) {                            //drum section
      DrumDelay=50000-Tempo;                 //if time for drumbeat
      DrumOn=1;                              //turn it on
}
if(0){                                       //if drum is on
 output=output+(kick[iDrum])*.05*(ampDrum);//play next sample
 if((sDrum%2)==1) {iDrum++;}                 //but play at Fs/2
 sDrum++;                                    //incr sample number
 if(iDrum>2500){iDrum=0; DrumOn=0;}          //drum off if last sample
}
output = output*.0001*VolSlider*VolSlider;
AIC23_data.channel[LEFT]=output;
AIC23_data.channel[RIGHT]=AIC23_data.channel[LEFT];
output_sample(AIC23_data.uint);              //output to both channels
}
main()          //init DSK,codec,McBSP and while(1) infinite loop
```

FIGURE 10.16. (*Continued*)

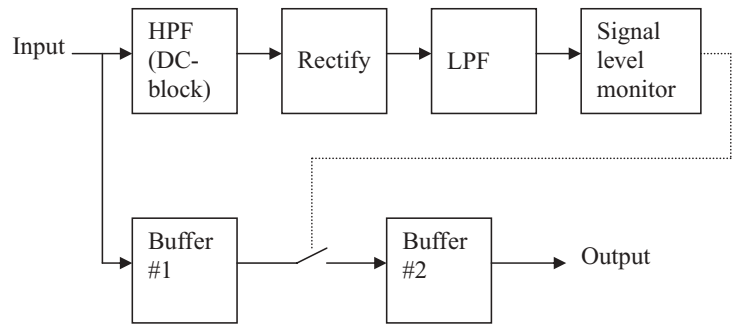


FIGURE 10.17. Block diagram for the detection of a voice signal from a microphone and playback of that signal in the reverse direction.

10.9 VOICE DETECTION AND REVERSE PLAYBACK

This project detects a voice signal from a microphone, then plays it back in the reverse direction. Figure 10.17 shows the block diagram that implements this project. All the necessary files are in the folder `detect_play`. Two circular buffers are used: an input buffer to hold 80,000 samples (10 seconds of data) continuously being updated and an output buffer to play back the input voice signal in the reverse direction. The signal level is monitored, and its envelope is tracked to determine whether or not a voice signal is present.

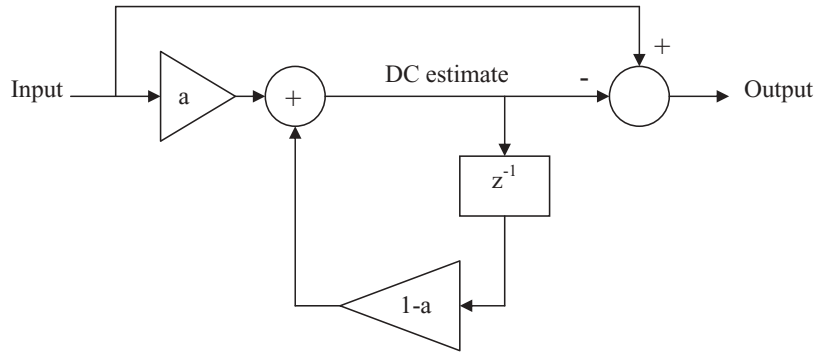


FIGURE 10.18. DC blocking first order IIR highpass filter for voice signal detection and reverse playback.

When a voice signal appears and subsequently dies out, the signal-level monitor sends a command to start the playback of the accumulated voice signal, specifying the duration of the signal in samples. The stored data are transferred from the input buffer to the output buffer for playback. Playback stops when one reaches the end of the entire signal detected.

The signal-level monitoring scheme includes rectification and filtering (using a simple first order IIR filter). An indicator specifies when the signal reaches an upper threshold. When the signal drops below a low threshold, the time difference between the start and end is calculated. If this time difference is less than a specified duration, the program continues into a no-signal state (if noise only). Otherwise, if it is more than a specified duration, a signal-detected mode is activated.

Figure 10.18 shows the DC blocking filter as a first-order IIR highpass filter. The coefficient `a` is much smaller than 1 (for a long time constant). The estimate of the DC filter is stored as a 32-bit integer.

The lowpass filter for the envelope detection is also implemented as a first order IIR filter, similar to the DC blocking filter except that the output is returned directly rather than being subtracted from the input. The filter coefficient `a` is larger for this filter to achieve a short time contant.

Build and test this project as **`detect_play`**.

10.10 PHASE SHIFT KEYING—BPSK ENCODING AND DECODING WITH PLL

See also the two projects on binary phase shift keying (BPSK) and modulation schemes in Sections 10.11 and 10.12. This project is decomposed into smaller mini-projects as background for the final project. The final project is the transmission of an encoded BPSK signal with voice as input and the reception (demodulation) of this signal with phase-locked loop (PLL) support on a second DSK. All the files associated with these projects are located in separate subfolders within the folder **PSK**.

10.15 SPEECH SYNTHESIS USING LINEAR PREDICTION OF SPEECH SIGNALS

Speech synthesis is based on the reproduction of human intelligible speech through artificial means [42–45]. Examples of speech synthesis technology include *text-to-speech* systems. The creation of synthetic speech covers a range of processes; and even though they are often lumped under the general term *text-to-speech*, a lot of work has been done to generate speech from sequences of the speech sounds. This would be a speech-sound (phoneme) to audio waveform synthesis, rather than going from text to phonemes (speech sounds) and then to sound. One of the first practical applications of speech synthesis was a speaking clock. It used optical storage for phrases and words (noun, verb, etc.), concatenated to form complete sentences. This led to a series of innovative products such as vocoders, speech toys, and so on. Advances in the understanding of the speech production mechanism in humans, coupled with similar advances in DSP, have had an impact on speech synthesis techniques. Perhaps the most singular factors that started a new era in this field were the computer processing and storage technologies. While speech and language were already important parts of daily life before the invention of the computer, the equipment and technology that developed over the last several years have made it possible to produce machines that speak, read, and even carry out dialogs. A number of vendors provide both recognition and speech technology. Some of the latest applications of speech synthesis are in cellular phones, security networks, and robotics.

There are different methods of speech synthesis based on the source. In a text-to-speech system, the source is a text string of characters read by the program to generate voice. Another approach is to associate intelligence in the program so that it can generate voice without external excitation. One of the earliest techniques was *Formant synthesis*. This method was limited in its ability to represent voice with high fidelity due to its inherent drawback of representing phonemes by three frequencies. This method and several analog technologies that followed were replaced by digital methods. Some early digital technologies were RELP (residue excited) and VELP (voice excited). These were replaced by new technologies, such as LPC (linear predictive coding), CELP (code excited), and PSOLA (pitch synchronous overlap-add). These technologies have been used extensively to generate artificial voice.

Linear Predictive Coding

Most methods that are used for analyzing speech start by transforming acoustic data into spectral form by performing short time Fourier analysis of the speech wave. Although this type of spectral analysis is a well-known technique for studying signals, its application to speech signal suffers from limitations due to the nonstationary and quasiperiodic properties of the speech wave. As a result, methods based on spectral analysis often do not provide a sufficiently accurate description of speech articulation. Linear predictive coding (LPC) represents the speech waveform directly in terms of time-varying parameters related to the transfer function

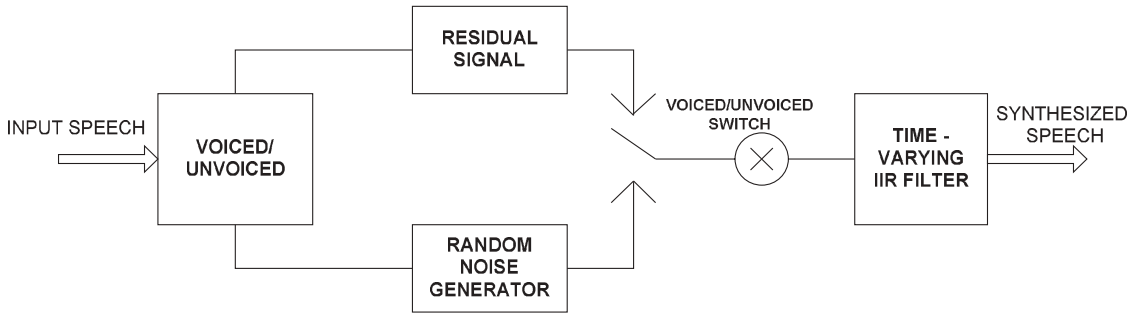


FIGURE 10.50. Diagram of the speech synthesis process.

of the vocal tract and the characteristics of the source function. It uses the knowledge that any speech can be represented by certain types of parametric information, including the filter coefficients (that model the vocal tract) and the excitation signal (that maps the source signals). The implementation of LPC reduces to the calculation of the filter coefficients and excitation signals, making it suitable for digital implementation.

Speech sounds are produced as a result of acoustical excitation of the human vocal tract. During production of the voiced sounds, the vocal chord is excited by a series of nearly periodic pulses generated by the vocal cords. In unvoiced sounds, excitation is provided by the air passing turbulently through constrictions in the tract. A simple model of the vocal tract is a discrete time-varying linear filter. Figure 10.50 is a diagram of the LPC speech synthesis. To reproduce the voice signal, the following are required:

1. An excitation signal
2. The LPC filter coefficients

The excitation mechanism can be approximated using a residual signal generator (for voiced signals) or a white Gaussian noise generator (for unvoiced signals) with adjustable amplitudes and periods. The linear predictor P, a transversal filter with p delays of one sample interval each, forms a weighted sum of past samples as the input of the predictor. The output of the predictor at the nth sampling instant is given by

$$s_n = \sum_{k=1}^{p} a_k \cdot (s_m) + \delta_n$$

where $m = n - k$ and δ_n represents the nth excitation sample.

Implementation

The input to the program is a sampled array of input speech using an 8-kHz sampling rate. The samples are stored in a header file. The length of the input speech

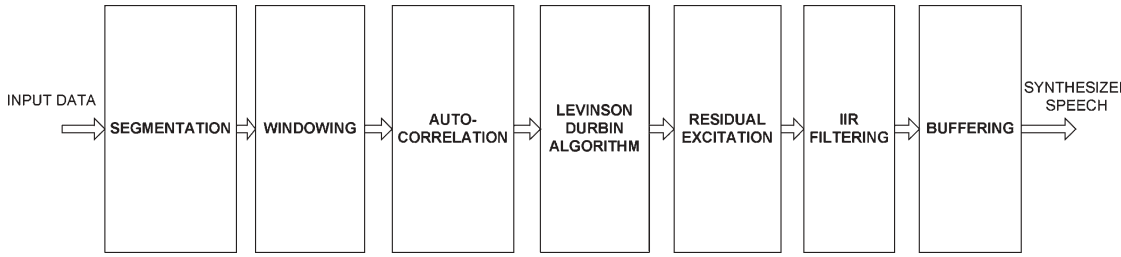


FIGURE 10.51. Speech synthesis algorithm with various modules.

array is 10,000 samples, translating into approximately 1.25 seconds of speech. The input array is segmented into a large number of frames, each 80 B long with an overlap of 40 B for each frame. Each frame is then passed to the following modules: windowing, autocorrelation, LPC, residual, IIR, and accumulate. External memory is utilized. A block diagram of the LPC speech synthesis algorithm with the various modules is shown in Figure 10.51.

1. *Segmentation*. This module separates the input voice into overlapping segments. The length of the segment is such that the speech segment appears stationary as well as quasiperiodic. The overlap provides a smooth transition between consecutive speech frames.
2. *Windowing*. The speech waveform is decomposed into smaller frames using the Hamming window. This suppresses the sidelobes in the frequency domain.
3. *Levinson–Durbin algorithm*. To calculate the LPC coefficients, the autocorrelation matrix of the speech frame is required. From this matrix, the LPC coefficients can be obtained using

$$r(i) = \sum_{k=1}^{p} a_k \cdot r(|i-k|)$$

   where $r(i)$ and ak represent the autocorrelation array and the coefficients, respectively.
4. *Residual signal*. For synthesis of the artificial voice, the excitation is given by the residual signal, which is obtained by passing the input speech frame through an FIR filter. It serves as an excitation signal for both voiced and unvoiced signals. This limits the algorithm due to the energy and frequency calculations required for making decisions about voiced/unvoiced excitation since, even for an unvoiced excitation that has a random signal as its source, the same principle of residue signal can still be used. This is because, in the case of unvoiced excitation, even the residue signal obtained will be random.

5. *Speech synthesis*. With the representation of the speech frame in the form of the LPC filter coefficients and the excitation signal, speech can be synthesized. This is done by passing the excitation signal (the residual signal) through an IIR filter. The residual signal generation and the speech synthesis modules imitate the vocal chord and the vocal tract of the speech production system in humans.
6. *Accumulation and buffering*. Since speech is segmented at the beginning, the synthesized voice needs to be concatenated. This is performed by the accumulation and buffering module.
7. *Output*. When the entire synthesized speech segment is obtained, it is played. During playback, the data are downsampled to 4 kHz to restore the intelligibility of the speech.

Implementation

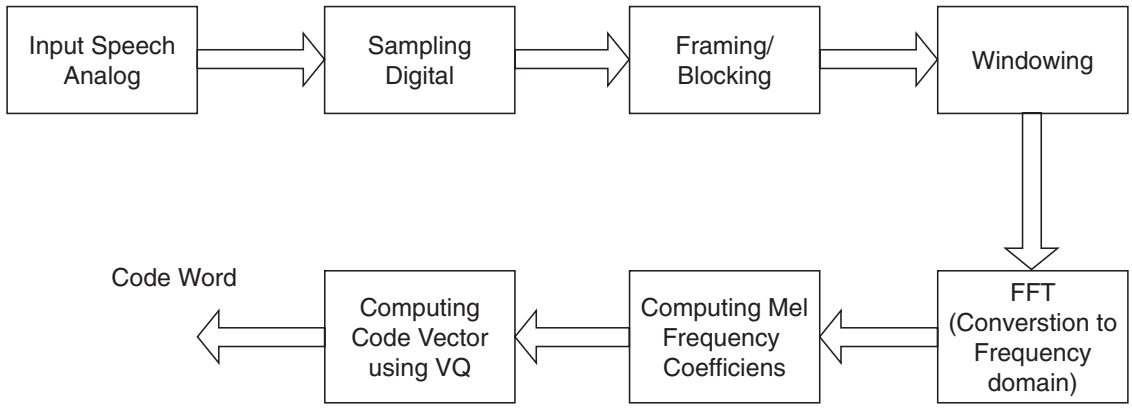
The complete support files are on the CD in the folder `speech_syn`. Generate a `.wav` file of the speech sample to be synthesized. For example, include *goaway.wav* in the MATLAB file `input_read.m`. The MATLAB file samples it for 8 kHz and stores the input samples array in the header file `input.h`. Include this generated header file in the main C source program `speech.c`. Build this project as **`speech_syn`**. Run the MATLAB program `input_read.m` to generate the two header files `input.h` (containing the input samples) and `hamming.h` (for the Hamming coefficients). Load/run `speech_syn.out` and verify the synthesized speech "go away" from a speaker connected to the DSK output. Three other `.wav` files are included in the folder and can be tested readily.

Results

Speech is synthesized for the following: "Go away," "Hello, professor," "Good evening," and "Vacation." The synthesized output voice is found to have considerable fidelity to the original speech. The voice/unvoiced speech phonemes are reproduced with considerable accuracy. This project can be improved with a larger buffer size for the samples and noise suppression filters. There is noise after each time the sentence is played. A speech recognition algorithm can be implemented in conjunction with the speech synthesis to facilitate a dialog.

10.16 AUTOMATIC SPEAKER RECOGNITION

This project implements an automatic speaker recognition system [46–50]. *Speaker recognition* refers to the concept of recognizing a speaker by his/her voice or speech samples. This is different from speech recognition. In automatic speaker recognition, an algorithm generates a hypothesis concerning the speaker's identity or authenticity. The speaker's voice can be used for ID and to gain access to services such as banking, voice mail, and so on.



(Spelling error-Computing-not computung)

FIGURE 10.52. Steps for speaker recognition implementation.

Speaker recognition systems contain two main modules: *feature extraction* and *classification*.

1. Feature extraction is a process that extracts a small amount of data from the voice signal that can be used to represent each speaker. This module converts a speech waveform to some type of parametric representation for further analysis and processing. Short-time spectral analysis is the most common way to characterize a speech signal. The Mel-frequency cepstrum coefficients (MFCCs) are used to parametrically represent the speech signal for the speaker recognition task. The steps in this process are shown in Figure 10.52:
   - **(a)** Block the speech signal into frames, each consisting of a fixed number of samples.
   - **(b)** Window each frame to minimize the signal discontinuities at the beginning and end of the frame.
   - **(c)** Use FFT to convert each frame from time to frequency domain.
   - **(d)** Convert the resulting spectrum into a Mel-frequency scale.
   - **(e)** Convert the Mel spectrum back to the time domain.
2. Classification consists of models for each speaker and a decision logic necessary to render a decision. This module classifies extracted features according to the individual speakers whose voices have been stored. The recorded voice patterns of the speakers are used to derive a classification algorithm. Vector quantization (VQ) is used. This is a process of mapping vectors from a large vector space to a finite number of regions in that space. Each region is called a *cluster* and can be represented by its center, called a *codeword*. The collection of all clusters is a *codebook*. In the training phase, a speaker-specific VQ codebook is generated for each known speaker by clustering his/her training acoustic vectors. The distance from a vector to the closest codeword of a codebook is called a *VQ distortion*. In the recognition phase, an input utterance of an

unknown voice is vector-quantized using each trained codebook, and the total VQ distortion is computed. The speaker corresponding to the VQ codebook with the smallest total distortion is identified.

Speaker recognition can be classified with identification and verification. *Speaker identification* is the process of determining which registered speaker provides a given utterance. *Speaker verification* is the process of accepting or rejecting the identity claim of a speaker. This project implements only the speaker identification (ID) process. The speaker ID process can be further subdivided into *closed set* and *open set*. The *closed set* speaker ID problem refers to a case where the speaker is known *a priori* to belong to a set of M speakers. In the *open set* case, the speaker may be out of the set and, hence, a "none of the above" category is necessary. In this project, only the simpler closed set speaker ID is used.

Speaker ID systems can be either *text-independent* or *text-dependent*. In the *text-independent* case, there is no restriction on the sentence or phrase to be spoken, whereas in the *text-dependent* case, the input sentence or phrase is indexed for each speaker. The text-dependent system, implemented in this project, is commonly found in speaker verification systems in which a person's password is critical for verifying his/her identity.

In the *training phase*, the feature vectors are used to create a model for each speaker. During the *testing phase*, when the test feature vector is used, a number will be associated with each speaker model indicating the degree of match with that speaker's model. This is done for a set of feature vectors, and the derived numbers can be used to find a likelihood score for each speaker's model. For the speaker ID problem, the feature vectors of the test utterance are passed through all the speakers' models and the scores are calculated. The model having the best score gives the speaker's identity (which is the decision component).

This project uses MFCC for feature extraction, VQ for classification/training, and the Euclidean distance between MFCC and the trained vectors (from VQ) for speaker ID. Much of this project was implemented with MATLAB [47].

Mel-Frequency Cepstrum Coefficients

MFCCs are based on the known variation of the human ear's critical bandwidths. A Mel-frequency scale is used with a linear frequency spacing below 1000 Hz and a logarithmic spacing above that level. The steps used to obtain the MFCCs follow.

1. *Level detection*. The start of an input speech signal is identified based on a prestored threshold value. It is captured after it starts and is passed on to the framing stage.
2. *Frame blocking*. The continuous speech signal is blocked into frames of N samples, with adjacent frames being separated by M ($M < N$). The first frame consists of the first N samples. The second frame begins M samples after the

first frame and overlaps it by $N - M$ samples. Each frame consists of 256 samples of speech signal, and the subsequent frame starts from the 100th sample of the previous frame. Thus, each frame overlaps with two other subsequent frames. This technique is called *framing*. The speech sample in one frame is considered to be stationary.

3. *Windowing*. After framing, windowing is applied to prevent spectral leakage. A Hamming window with 256 coefficients is used.
4. *Fast Fourier transform*. The FFT converts the time-domain speech signal into a frequency domain to yield a complex signal. Speech is a real signal, but its FFT has both real and imaginary components.
5. *Power spectrum calculation*. The power of the frequency domain is calculated by summing the square of the real and imaginary components of the signal to yield a real signal. The second half of the samples in the frame are ignored since they are symmetric to the first half (the speech signal being real).
6. *Mel-frequency wrapping*. Triangular filters are designed using the Mel-frequency scale with a bank of filters to approximate the human ear. The power signal is then applied to this bank of filters to determine the frequency content across each filter. Twenty filters are chosen, uniformly spaced in the Mel-frequency scale between 0 and 4 kHz. The Mel-frequency spectrum is computed by multiplying the signal spectrum with a set of triangular filters designed using the Mel scale. For a given frequency f, the mel of the frequency is given by

$$B(f) = [1125 \ln(1 + f/700)] \text{ mels}$$

If m is the mel, then the corresponding frequency is

$$B^{-1}(m) = [700 \exp(m/1125) - 700] \text{ Hz}$$

The frequency edge of each filter is computed by substituting the corresponding mel. Once the edge frequencies and the center frequencies of the filter are found, boundary points are computed to determine the transfer function of the filter.

7. *Mel-frequency cepstrum coefficients*. The log mel spectrum is converted back to time. The discrete cosine transform (DCT) of the log of the signal yields the MFCCs.

Speaker Training—VQ

VQ is a process of mapping vectors from a large vector space to a finite number of regions in that space. Each region is called a *cluster* and can be represented by its center, the codeword. As noted earlier, a codebook is the collection of all the clusters. An example of a one-dimensional VQ has every number less than –2 approximated by –3; every number between –2 and 0 approximated by –1; every number

between 0 and 2 approximated by +1; and every number greater than 2 approximated by +3. These approximate values are uniquely represented by 2 bits, yielding a one-dimensional, 2-bit VQ. An example of a two-dimensional VQ consists of 16 regions and 16 stars, each of which can be uniquely represented by 4 bits (a two-dimensional 4-bit VQ). Each pair of numbers that fall into a region are approximated by a star associated with that region. The stars are called *codevectors*, and the regions are called *encoding regions*. The set of all the codevectors is called the *codebook*, and the set of all encoding regions is called the *partition* of the space.

Speaker Identification (Using Euclidean Distances)

After computing the MFCCs, the speaker is identified using a set of trained vectors (samples of registered speakers) in an array. To identify the speaker, the Euclidean distance between the trained vectors and the MFCCs is computed for each trained vector. The trained vector that produces the smallest Euclidean distance is identified as the speaker.

Implementation

The design is first tested with MATLAB. A total of eight speech samples from eight different people (eight speakers, labeled S1 to S8) are used to test this project. Each speaker utters the same single digit, *zero*, once in a training session (then also in a testing session). A digit is often used for testing in speaker recognition systems because of its applicability to many security applications. This project was implemented on the C6711 DSK and can be transported to the C6713 DSK. Of the eight speakers, the system identified six correctly (a 75% identification rate). The identification rate can be improved by adding more vectors to the training codewords. The performance of the system may be improved by using two-dimensional or four-dimensional VQ (training header file would be $8 \times 20 \times 4$) or by changing the quantization method to dynamic time wrapping or hidden Markov modeling. A `readme` file to test this project is on the CD in the folder **`speaker_recognition`**, along with all the appropriate support files. These support files include several modules for framing and windowing, power spectrum, threshold detection, VQ, and the Mel-frequency spectrum.

10.17 μ-LAW FOR SPEECH COMPANDING

An analog input such as speech is converted into digital form and compressed into 8-bit data. μ-Law *encoding* is a nonuniform quantizing logarithmic compression scheme for audio signals. It is used in the United States to compress a signal into a logarithmic scale when coding for transmission. It is widely used in the telecommunications field because it improves the SNR without increasing the amount of data.

The dynamic range increases, while the number of bits for quantization remains the same. Typically, μ-law compressed speech is carried in 8-bit samples. It carries