INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad -500 043

ELECTRONICS & COMMUNICATION ENGINEERING

COURSE LECTURE NOTES

Course Name	**DIGITAL SIGNAL PROCESSORS AND ARCHITECTURE**
Course Code	AEC507
Programme	B.Tech
Semester	VIII
Course Coordinator	Ms. C.Devisupraja, Assistant Professor, ECE
Lecture Numbers	1-63
Topic Covered	All

COURSE OBJECTIVES

The course should enable the students to:

I	Impart the knowledge of basic DSP concepts and number systems to be used, different types of A/D, D/A conversion errors.
II	Learn the architectural differences between DSP and General purpose processor.
III	Learn about interfacing of serial & parallel communication devices to the processor.
IV	Implement the DSP & FFT algorithms.

COURSE LEARNING OUTCOMES (CLOs):

Students, who complete the course, will have demonstrated the ability to do the following:

AEC507.01	Understand how digital to analog (D/A) and analog to digital (A/D) converters operate on a signal and be able to model these operations mathematically.
AEC507.02	Understand the inter-relationship between DFT and various transforms.
AEC507.03	Understand the IEE-754 floating point and source of errors in DSP implementations.
AEC507.04	Understand the fast computation of DFT and appreciate the FFT Processing.
AEC507.05	Understand the concept of multiplier and multiplier Accumulator.
AEC507.06	Design SMID,VLIW architectures.
AEC507.07	Understand the modified bus structures and memory access in PDSPs.
AEC507.08	Understand the special addressing modes in PDSPs.

AEC507.09	Understand the architecture of TMS320C54XX DSPs.
AEC507.10	Understand the addressing modes and memory space of TMS320C54XX DSPs.
AEC507.11	Understand the various interrupts and pipeline operation of TMS320C54XX processors.
AEC507.12	Analyze the Program control, instruction set and programming.
AEC507.13	Understand the concept of on-chip Peripherals.
AEC507.14	Understand the significance of memory space organization.
AEC507.15	Analyze external bus interfacing signals.
AEC507.16	Explain about parallel I/O interface, programmed I/O.
AEC507.17	Understand the significance of Interrupts and Direct Memory Access.
AEC507.18	Understand the basic concepts of convolution and correlation.
AEC507.19	Compare the characteristics of IIR and FIR filters.
AEC507.20	Analyze the concepts of interpolation and decimation filters.

SYLLABUS

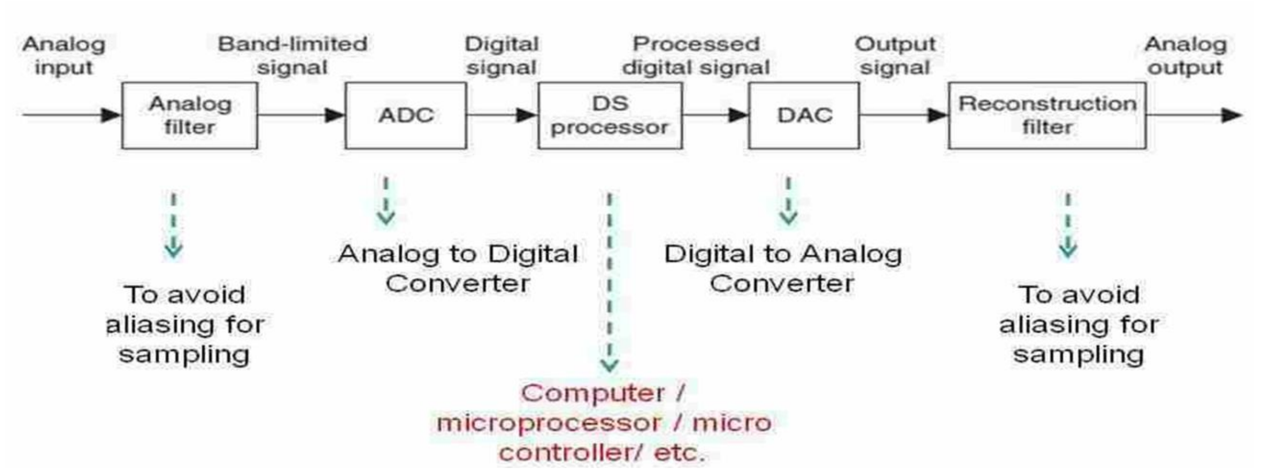
UNIT – I: INTRODUCTION TO DIGITAL SIGNAL PROCESSING
Introduction: Digital signal-processing system, discrete Fourier Transform (DFT) and fast Fourier transform (FFT), differences between DSP and other micro processor architectures; Number formats: Fixed point, floating point and block floating point formats, IEEE-754 floating point, dynamic range and precision, relation between data word size and instruction word size; Sources of error in DSP implementations: A/D conversion errors, DSP computational errors, D/A conversion errors, Q-notation.
UNIT – II: ARCHITECTURE OF PROGRAMMABLE DSPs
Multiplier and multiplier accumulator, modified bus structures and memory access in PDSPs, multiple access memory, multiport memory, SIMD, VLIW architectures, pipelining, special addressing modes in PDSPs, on-chip peripherals.
UNIT – III: OVERVIEW OF TMS320C54XX PROCESSOR
Architecture of TMS320C54XX DSPs, addressing modes, memory space of TMS320C54XX processors. Program control, instruction set and programming, on-chip peripherals, interrupts of TMS320C54XX processors, pipeline operation.
UNIT – IV: INTERFACING MEMORY AND I/O PERIPHERALS TO PDSPs
Memory space organization, external bus interfacing signals, memory interface, parallel I/O interface, programmed I/O, interrupts and I/O, direct memory access (DMA).
UNIT – V: IMPLEMENTATIONS OF BASIC DSP ALGORITHMS
The Q-notation, convolution, correlation, FIR filters, IIR filters, interpolation filters, decimation filters, an FFT algorithm for DFT filters computation of the signal spectrum.

UNIT-1

Introduction to Digital Signal Processing

DSP is a technique of performing the mathematical operations on the signals in digital domain. As real time signals are analog in nature we need first convert the analog signal to digital, then we have to process the signal in digital domain and again converting back to analog domain. Thus ADC is required at the input side whereas a DAC is required at the output end. A typical DSP system is as shown in figure 1.1.

- The main function of low pass ant aliasing filter is to band limit the input signal to the folding frequency without distortion.
- It should be noted that even if the signal is band limited, there is always wide-band additive noise which will be folded back to create aliasing.
- When an analog voltage is connected directly to an ADC, the conversion process can be adversely affected if the voltage is changing during the conversion time.
- The quality of conversion process can be improved by using sample and hold circuit



Analog input
Band-limited signal
Digital signal
Processed digital signal
Output signal
Analog output
Analog filter
ADC
DS processor
DAC
Reconstruction filter
To avoid aliasing for sampling
Analog to Digital Converter
Computer / microprocessor / micro controller/ etc.
Digital to Analog Converter
To avoid aliasing for sampling

Advantages of DSP

- Programmability: software digital signal processes can be quickly modified, in contrast to analog circuits, which must be physically rearranged.
- Versatility: Flexible and easy to upgrade.
- Stability: Less sensitive environmental changes such as electromagnetic interference.

Need for DSP

Analog signal Processing has the following drawbacks:

- They are sensitive to environmental changes
- Aging
- Uncertain performance in production units
- Variation in performance of units
- Cost of the system will be high
- Scalability

If Digital Signal Processing would have been used we can overcome the above shortcomings of ASP.

A Digital Signal Processing System

A computer or a processor is used for digital signal processing. Anti aliasing filter is a LPF which passes signal with frequency less than or equal to half the sampling frequency in order to avoid Aliasing effect. Similarly at the other end, reconstruction filter is used to reconstruct the samples from the staircase output of the DAC (Figure 1.2).

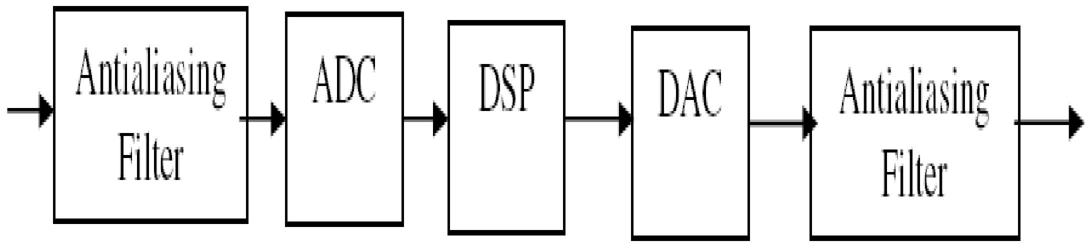


Fig 1.2 The Block Diagram of a DSP System

Signals that occur in a typical DSP are as shown in figure 1.3.

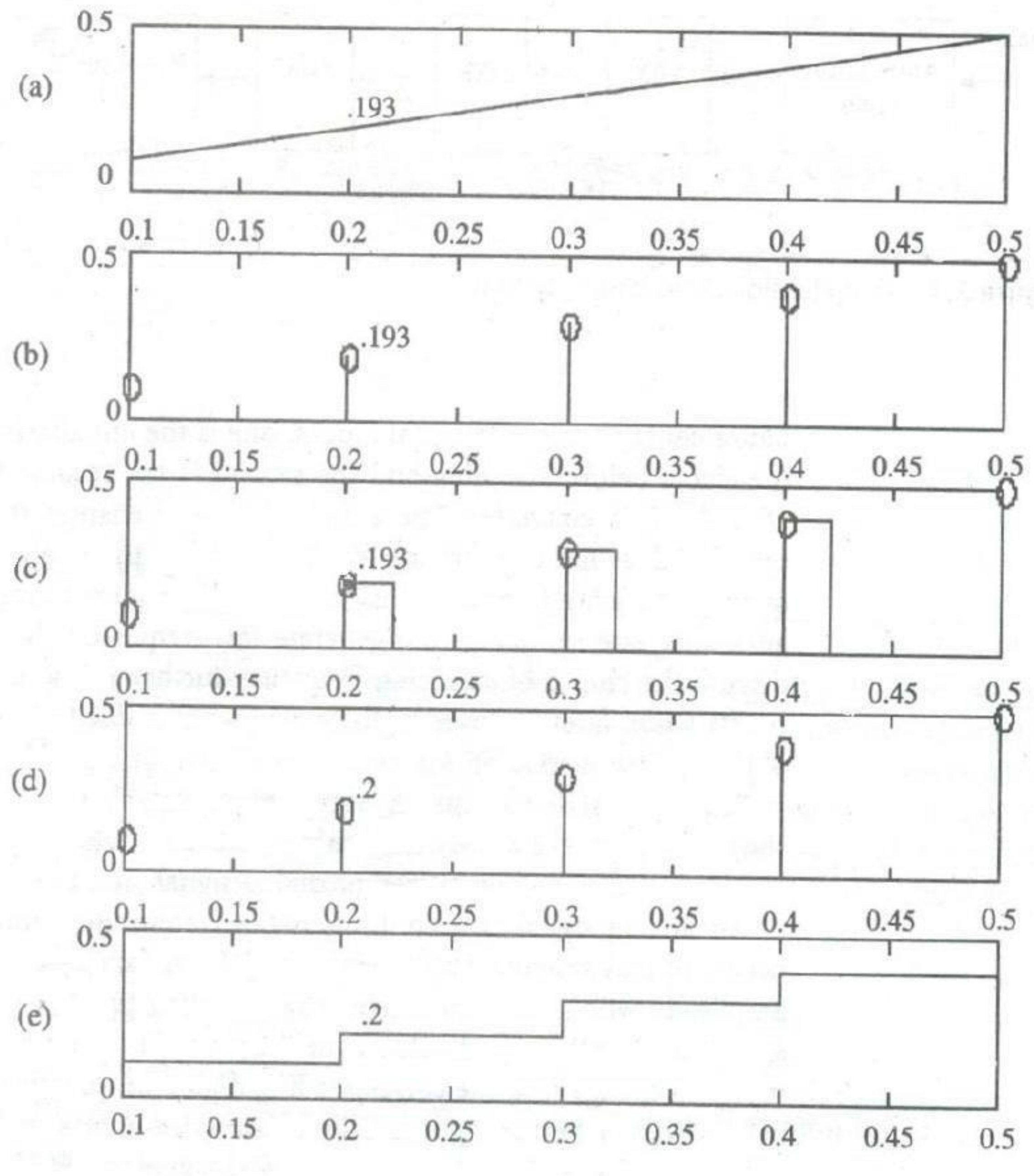


Fig 1.3: (a) Continuous time signal (b) Sampled Signal (c) Sampled Data Signal (d) Quantized Signal (e) DAC Output

The Sampling Process

ADC process involves sampling the signal and then quantizing the same to a digital value. In order to avoid Aliasing effect, the signal has to be sampled at a rate at least equal to the Nyquist rate. The condition for Nyquist Criterion is as given below, fs= 1/T □ □ 2 fm

Where, fs is the sampling frequency, fm is the maximum frequency component in the message signal. If the sampling of the signal is carried out with a rate less than the Nyquist rate, the higher frequency components of the signal cannot be reconstructed properly. The plots of the reconstructed outputs for various conditions are as shown in figure 1.4.

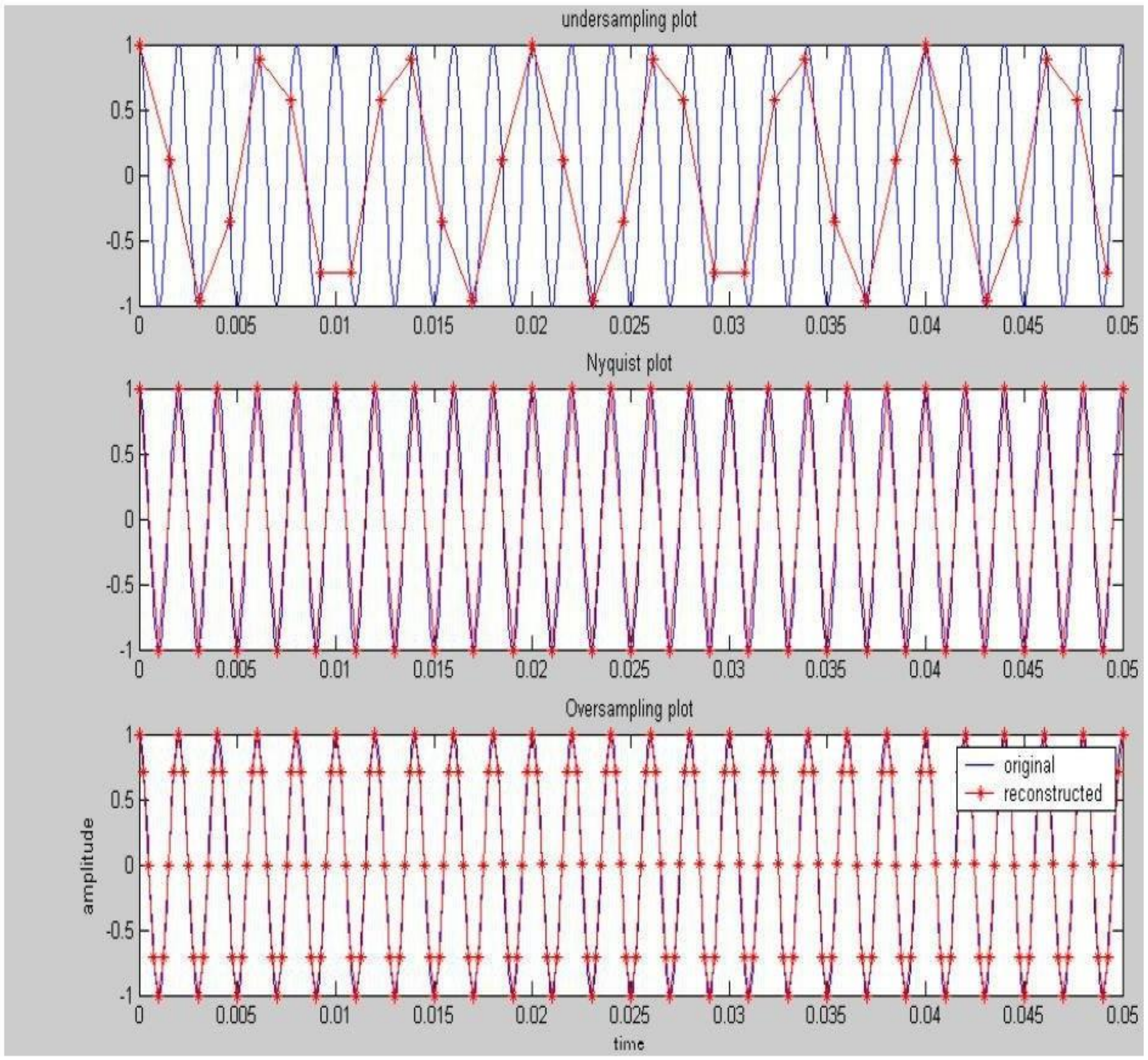


Fig 1.4 Verification of Sampling Theorem

Discrete Time Sequences

Consider an analog signal x(t) given by, x(t)= A cos (2 ft). If this signal is sampled at a Sampling Interval T, in the above equation replacing t by nT we get, x (nT) = A cos (2 fnT) where n= 0,1, 2,..etc

For simplicity denote x (nT) as x (n)

- x (n) = A cos (2πfnT) where n= 0,1, 2,..etc

We have fs=1/T also θ = 2ΠfnT

- x (n) = A cos (2πfnT)= A cos (2πfn/fs) = A cos πn

The quantity isθ called as digital frequency.

θ = 2πfT = 2πf/fs radians

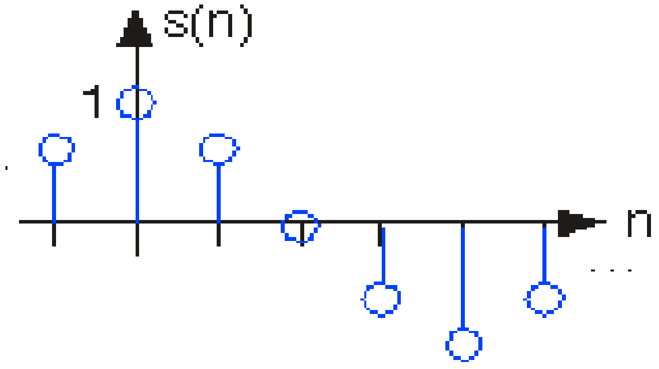


Fig 1.5 A Cosine Waveform

A sequence that repeats itself after every period N is called a periodic sequence.

Consider a periodic sequence x (n) with period N x (n)=x (n+N) n=……..,-1,0,1,2,……..

Frequency response gives the frequency domain equivalent of a discrete time sequence. It is denoted as **$X(e^{j\theta})=\sum x(n)\ e^{-jn\theta}$**

Frequency response of a discrete sequence involves both magnitude response and phase response.

Discrete Fourier Transform and Fast Fourier Transform

DFT Pair:

DFT is used to transform a time domain sequence x (n) to a frequency domain sequence X (K).The equations that relate the time domain sequence x (n) and the corresponding frequency domain sequence X (K) are called DFT Pair and is given by,

$DFT(FFT):$

$$X(k)=\sum_{n=0}^{N-1} x(n)\cdot e^{-j\left(\frac{2\pi}{N}\right)nk} \quad (k=0,1,\ldots,N-1)$$

$IDFT(IFFT):$

$$x(n)=\frac{1}{N}\sum_{k=0}^{N-1} X(k)\cdot e^{j\left(\frac{2\pi}{N}\right)nk} \quad (n=0,1,\ldots,N-1)$$

The Relationship between DFT and Frequency Response:

We have,

$X\ (e^{j\theta}) = \Sigma x(n)\ e^{-jn\theta}$

Also

$X\ (K) = \Sigma x(n)\ e^{-j2\pi nk/N}$

$\therefore X\ (K) = X\ (e^{j\theta})$ at $\theta = 2\pi k/N$

From the above expression it is clear that we can use DFT to find the Frequency response of a discrete signal. Spacing between the elements of X(k) is given as f=fs/N=1/NT=1/T0.Where T0 isthe signal record length.

It is clear from the expression of f that, in order to minimize the spacing between the samples N has to be a large value. Although DFT is an efficient technique of obtaining the frequency response of a sequence, it requires more number of complex operations like additions and multiplications.

Thus many improvements over DFT were proposed. One such technique is to use the periodicity property of the twiddle factor $e^{-j2\ /N}$. Those algorithms were called as Fast Fourier Transform Algorithms. The following table depicts the complexity involved in the computation using DFT algorithms.

Table 1.1 Complexity in DFT algorithm

Operations	Number of Computations
Complex Multiplications	N^2
Complex Additions	N (N-1)
Real Multiplications	$4N^2$
Real Additions	2N (2N-1)
Trigonometric Functions	$2N^2$

FFT algorithms are classified into two categories via

1. Decimation in Time FFT
2. Decimation in Frequency FFT

In decimation in time FFT the sequence is divided in time domain successively till we reach the sequences of length 2. Whereas in Decimation in Frequency FFT, the sequence X(K) is divided successively. The complexity of computation will get reduced considerably in case of FFT algorithms.

Linear Time Invariant Systems

A system which satisfies superposition theorem is called as a linear system and a system that has same input output relation at all times is called a Time Invariant System. Systems, which satisfy both the properties, are called LTI systems.

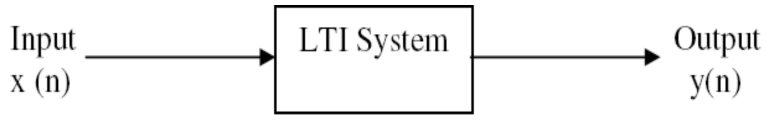


Fig 1.6 An LTI System

LTI systems are characterized by its impulse response or unit sample response in time domain whereas it is characterized by the system function in frequency domain.

Convolution

Convolution is the operation that related the input output of an LTI system, to its unit sample response. The output of the system y (n) for the input x (n) and the impulse response of the system being h (n) is given as y (n) = x(n) * h(n) = ∑ x(k) h(n-k), x(n) is the input of the system, h(n) is the impulse response of the system, y(n) is the output of the system.

Z Transformation

Z Transformations are used to find the frequency response of the system. The Z Transform for a discrete sequence x (n) is given by, **X(Z)= ∑x(n)** $\mathbf{z^{-n}}$

The System Function

An LTI system is characterized by its System function or the transfer function. The system function of a system is the ratio of the Z transformation of its output to that of its input. It is denoted as H (Z) and is given by H (Z) = Y (Z)/ X (Z).

The magnitude and phase of the transfer function H (Z) gives the frequency response of the system. From the transfer function we can also get the poles and zeros of the system by solving its numerator and denominator respectively.

Digital Filters

Filters are used to remove the unwanted components in the sequence. They are characterized by the impulse response h (n). The general difference equation for an Nth order filter is given by

y (n) = ∑a_ky(n-k)+ ∑ b_k x(n-k)

A typical digital filter structure is as shown in figure 1.7.

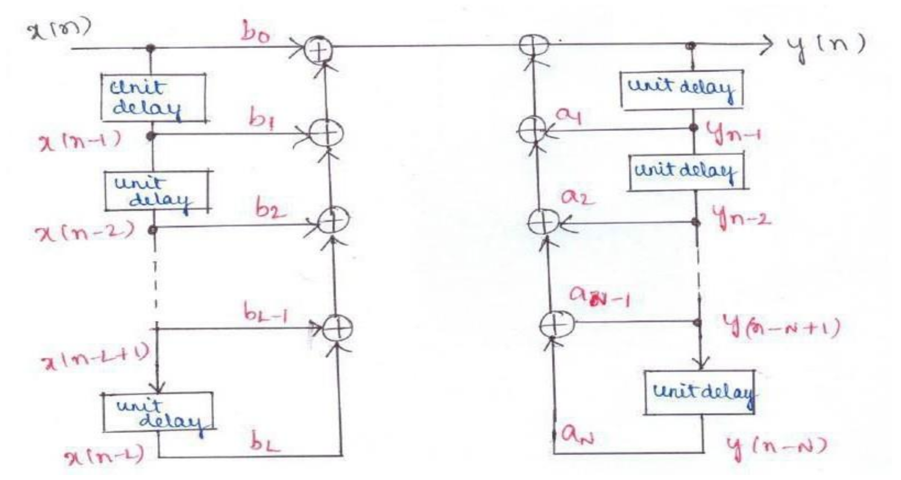


Fig 1.7 Structure of a Digital Filter

Values of the filter coefficients vary with respect to the type of the filter. Design of a digital filter involves determining the filter coefficients. Based on the length of the impulse response, digital filters are classified into two categories via Finite Impulse Response (FIR) Filters and Infinite Impulse Response (IIR) Filters.

FIR Filters

FIR filters have impulse responses of finite lengths. In FIR filters the present output depends only on the past and present values of the input sequence but not on the previous output sequences. Thus they are non recursive hence they are inherently stable.FIR filters possess linear phase response. Hence they are very much applicable for the applications requiring linear phase response.

The difference equation of an FIR filter is represented as

$$y(n) = \Sigma b_k x(n-k)$$

The frequency response of an FIR filter is given as

$$H(e^{j\theta}) = \Sigma b_k e^{-jk\theta}$$

$$H(Z) = \Sigma b_k Z^{-k}$$

The major drawback of FIR filters is, they require more number of filter coefficients to realize a desired response as compared to IIR filters. Thus the computational time required will also be more.

IIR Filters

Unlike FIR filters, IIR filters have infinite number of impulse response samples. They are recursive filters as the output depends not only on the past and present inputs but also on the past outputs. They generally do not have linear phase characteristics. Typical system function of such filters is given by,

$$H(Z) = (b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots\ldots\ldots\ldots b_L z^{-L}) / (1 - a_1 z^{-1} - a_2 z^{-2} - \ldots\ldots\ldots a_N z^{-N})$$

Stability of IIR filters depends on the number and the values of the filter coefficients. The major advantage of IIR filters over FIR is that, they require lesser coefficients compared to FIR filters for the same desired response, thus requiring less computation time.

FIR Filter Design

Frequency response of an FIR filter is given by the following expression,

$$H(e^{j\theta}) = \Sigma b_k e^{-jk\theta}$$

Design procedure of an FIR filter involves the determination of the filter coefficients bk.

$$b_k = (1/2\pi) \int H(e^{j\theta}) e^{-jk\theta} d\theta$$

IIR Filter Design

IIR filters can be designed using two methods viz using windows and direct method. In this approach, a digital filter can be designed based on its equivalent analog filter. An analog filter is designed first for the equivalent analog specifications for the given digital specifications. Then using appropriate frequency transformations, a digital filter can be obtained. The filter specifications consist of passband and stopband ripples in dB and Passband and Stopband frequencies in rad/sec.

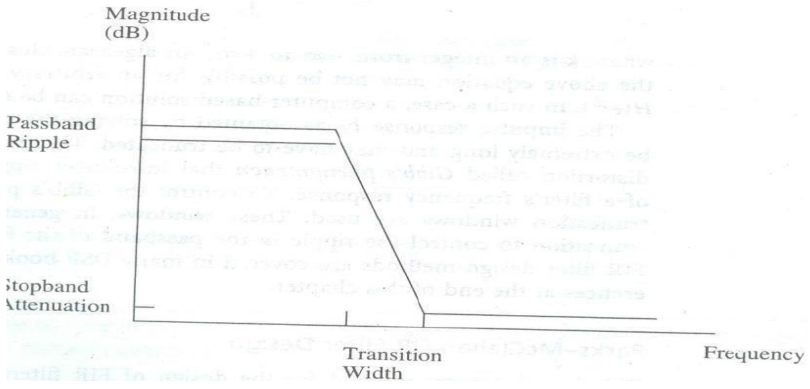


Fig 1.11 Lowpass Filter Specifications

Direct IIR filter design methods are based on least squares fit to a desired frequency response. These methods allow arbitrary frequency response specifications.

Decimation and Interpolation

Decimation and Interpolation are two techniques used to alter the sampling rate of a sequence. Decimation involves decreasing the sampling rate without violating the sampling theorem whereas interpolation increases the sampling rate of a sequence appropriately by considering its neighboring samples.

Decimation

Decimation is a process of dropping the samples without violating sampling theorem. The factor by which the signal is decimated is called as decimation factor and it is denoted by M. It is given by,

$$y(m)=w(mM)= \Sigma\, b_k\, x(mM-k) \quad \text{where } w(n)= \Sigma\, b_k\, x(n-k)$$

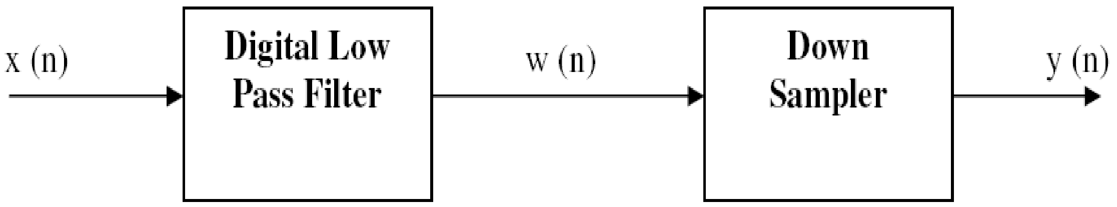


Fig 1.12 Decimation Process

Interpolation

Interpolation is a process of increasing the sampling rate by inserting new samples in between. The input output relation for the interpolation, where the sampling rate is increased by a factor L, is given as,

$$y(m) = \Sigma\, b_K\, w(m-k)$$

where $w(n) = x(m/L)$, $m = 0, \pm L, \pm 2L\ldots\ldots$

0 Otherwise

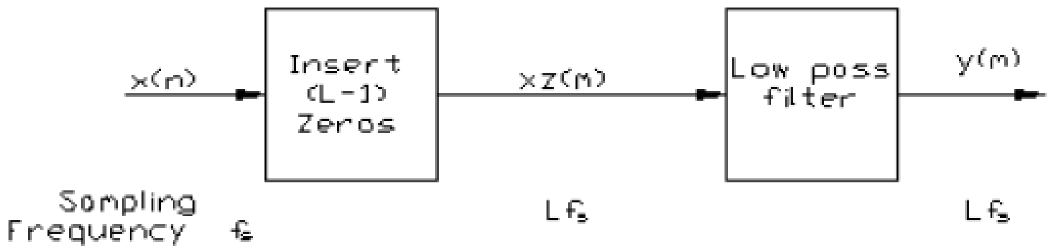
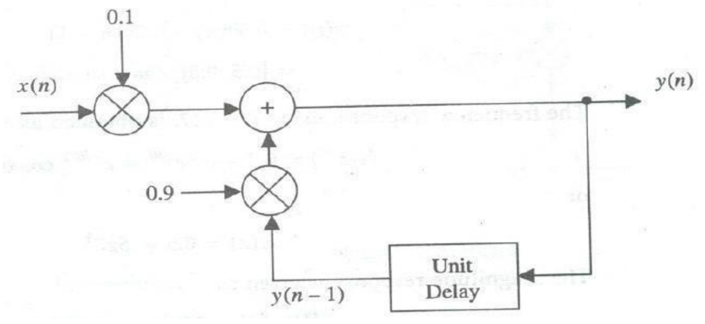


Fig 1.13 Interpolation Process

Problems:

1. **Obtain the transfer function of the IIR filter whose difference equation is given by y (n)= 0.9y (n-1)+0.1x (n)**

y (n)= 0.9y (n-1)+0.1x (n)
Taking Z transformation both sides
Y (Z) = 0.9 Z-1 Y (Z) + 0.1 X (Z)
Y (Z) [1- 0.9 Z-1] = 0.1 X (Z)
The transfer function of the system is given bythe expression,
H (Z)= Y(Z)/X(Z)
= 0.1/ [1- 0.9 Z^{-1}]
Realization of the IIR filter with the above difference equation is as shown in figure.



2. Let x(n)= [0 3 6 9 12] be interpolated with L=3. If the filter coefficients of the filters are bk=[1/3 2/3 1 2/3 1/3], obtain the interpolated sequence

After inserting zeros,
w (m) = [0 0 0 3 0 0 6 0 0 9 0 0 12]
bk=[1/3 2/3 1 2/3 1/3]
We have,
y(m)= bk w(m-k) = b-2 w(m+2)+ b-1 w(m+1)+ b0 w(m)+ b1 w(m-1)+ b2 w(m-2)
Substituting the values of m, we get
y(0)= b-2 w(2)+ b-1 w(1)+ b0 w(0)+ b1 w(-1)+ b2 w(-2)= 0
y(1)= b-2 w(3)+ b-1 w(2)+ b0 w(1)+ b1 w(0)+ b2 w(-1)=1
y(2)= b-2 w(4)+ b-1 w(3)+ b0 w(2)+ b1 w(1)+ b2 w(0)=2
Similarly we get the remaining samples as,
y (n) = [0 1 2 3 4 5 6 7 8 9 10 11 12]

UNIT-2
Architectures for Programmable Digital Signal Processing Devices

Basic Architectural Features

A programmable DSP device should provide instructions similar to a conventional microprocessor. The instruction set of a typical DSP device should include the following,

a. Arithmetic operations such as ADD, SUBTRACT, MULTIPLY etc
b. Logical operations such as AND, OR, NOT, XOR etc
c. Multiply and Accumulate (MAC) operation
d. Signal scaling operation

In addition to the above provisions, the architecture should also include,

a. On chip registers to store immediate results
b. On chip memories to store signal samples (RAM)
c. On chip memories to store filter coefficients (ROM)

DSP Computational Building Blocks

Each computational block of the DSP should be optimized for functionality and speed and in the meanwhile the design should be sufficiently general so that it can be easily integrated with other blocks to implement overall DSP systems.

Multipliers

The advent of single chip multipliers paved the way for implementing DSP functions on a VLSI chip. Parallel multipliers replaced the traditional shift and add multipliers now days. Parallel multipliers take a single processor cycle to fetch and execute the instruction and to store the result. They are also called as Array multipliers. The key features to be considered for a multiplier are:

a. Accuracy
b. Dynamic range
c. Speed

The number of bits used to represent the operands decides the accuracy and the dynamic range of the multiplier. Whereas speed is decided by the architecture employed. If the multipliers are implemented using hardware, the speed of execution will be very high but the circuit complexity will also increases considerably. Thus there should be a tradeoff between the speed of execution and the circuit complexity. Hence the choice of the architecture normally depends on the application.

Parallel Multipliers

Consider the multiplication of two unsigned numbers A and B. Let A be represented using m bits as (Am-1 Am-2 …….. A1 A0) and B be represented using n bits as (Bn-1 Bn-2 …….. B1 B0). Then the product of these two numbers is given by,

				A_3	A_2	A_1	A_0
				B_3	B_2	B_1	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			
P7	P6	P5	P4	P3	P2	P1	P0

This operation can be implemented paralleling using Braun multiplier whose hardware structure is as shown in the figure 2.1.

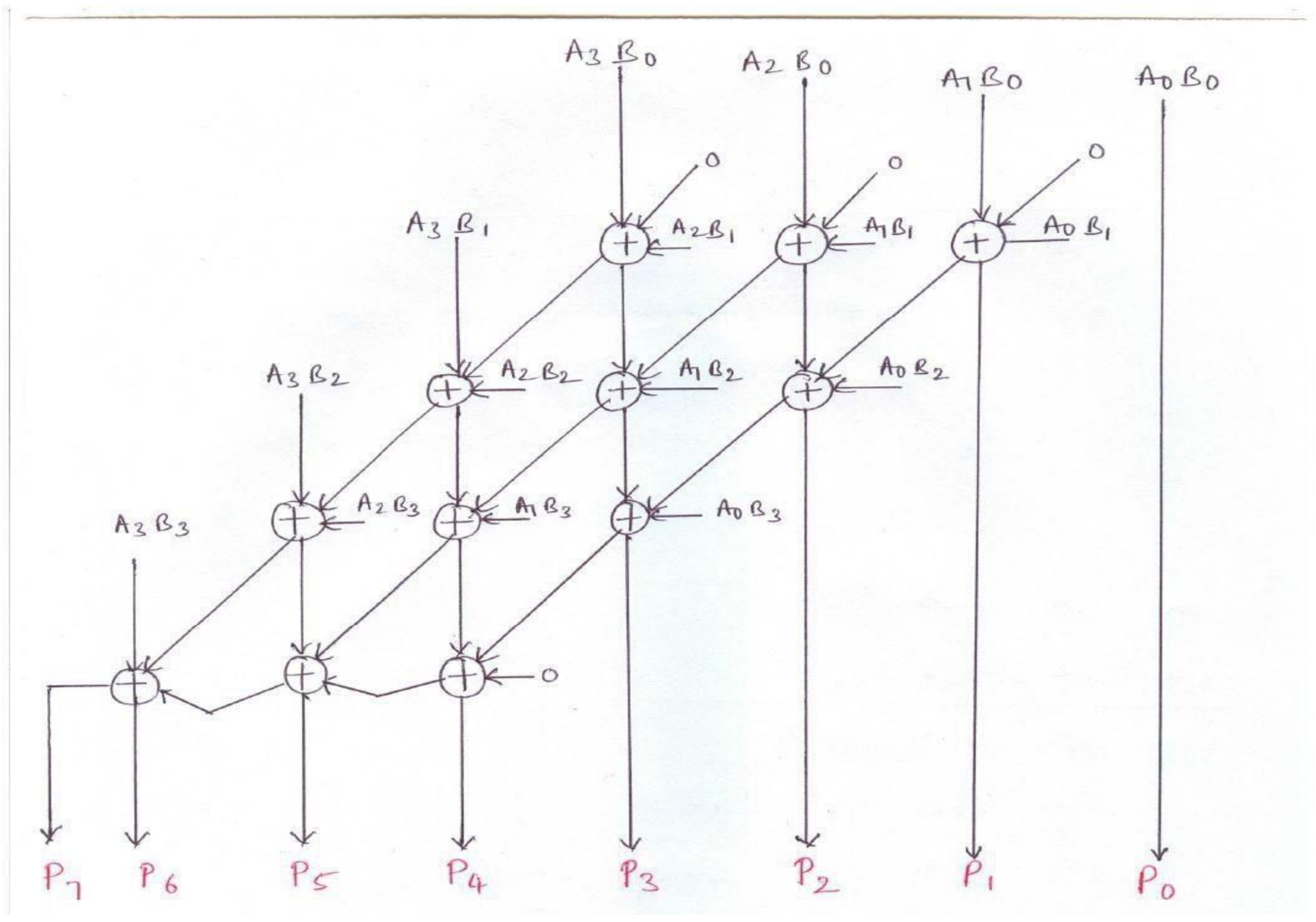


Fig 2.1 Braun Multiplier for a 4X4 Multiplication

Multipliers for Signed Numbers

In the Braun multiplier the sign of the numbers are not considered into account. In order to implement a multiplier for signed numbers, additional hardware is required to modify the Braun multiplier. The modified multiplier is called as Baugh-Wooley multiplier.

Consider two signed numbers A and B,

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

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

Product P= $P_{m+n-1} \ldots\ldots P_1\, P_0$

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

Speed

Conventional Shift and Add technique of multiplication requires n cycles to perform the multiplication of two n bit numbers. Whereas in parallel multipliers the time required will be the longest path delay in the combinational circuit used. As DSP applications generally require very high speed, it is desirable to have multipliers operating at the highest possible speed by having parallel implementation.

Bus Widths

Consider the multiplication of two n bit numbers X and Y. The product Z can be at most 2n bits long. In order to perform the whole operation in a single execution cycle, we require two buses of width n bits each to fetch the operands X and Y and a bus of width 2n bits to store the result Z to the memory. Although this performs the operation faster, it is not an efficient way of implementation as it is expensive. Many alternatives for the above method have been proposed. One such method is to use the program bus itself to fetch one of the operands after fetching the instruction, thus requiring only one bus to fetch the operands. And the result Z can be stored back to the memory using the same operand bus. But the problem with this is the result Z is 2n bits long whereas the operand bus is just n bits long. We have two alternatives to solve this problem, a. Use the n bits operand bus and save Z at two successive memory locations. Although it stores the exact value of Z in the memory, it takes two cycles to store the result.
b. Discard the lower n bits of the result Z and store only the higher order n bits into the memory. It is not applicable for the applications where accurate result is required. Another alternative can be used for the applications where speed is not a major concern. In which latches are used for inputs and outputs thus requiring a single bus to fetch the operands and to store the result (Fig 2.2).

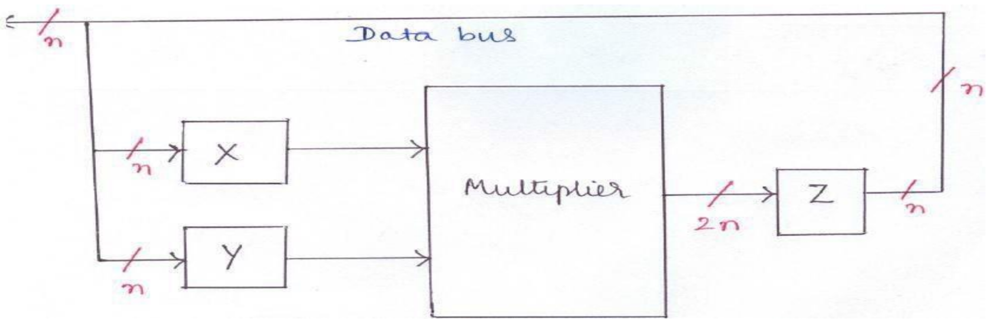


Fig 2.2: A Multiplier with Input and Output Latches

Shifters

Shifters are used to either scale down or scale up operands or the results. The following scenarios give the necessity of a shifter

a. While performing the addition of N numbers each of n bits long, the sum can grow up to n+log2 N bits long. If the accumulator is of n bits long, then an overflow error will occur. This can be overcome by using a shifter to scale down the operand by an amount of log2N.
b. Similarly while calculating the product of two n bit numbers, the product can grow up to 2n bits long. Generally the lower n bits get neglected and the sign bit is shifted to save the sign of the product.
c. Finally in case of addition of two floating-point numbers, one of the operands has to be shifted appropriately to make the exponents of two numbers equal.

From the above cases it is clear that, a shifter is required in the architecture of a DSP.

Barrel Shifters

In conventional microprocessors, normal shift registers are used for shift operation. As it requires one clock cycle for each shift, it is not desirable for DSP applications, which generally involves more shifts. In other words, for DSP applications as speed is the crucial issue, several shifts are to be accomplished in a single execution cycle. This can be accomplished using a barrel shifter, which connects the input lines representing a word to a group of output lines with the required shifts determined by its control inputs. For an input of length n, log2 n control lines are required. And an dditional control line is required to indicate the direction of the shift.
The block diagram of a typical barrel shifter is as shown in figure 2.3.

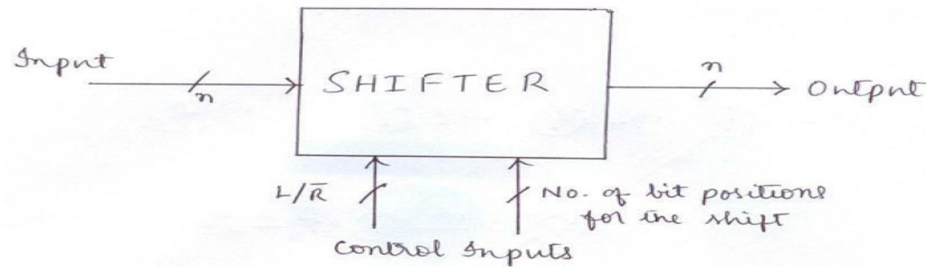
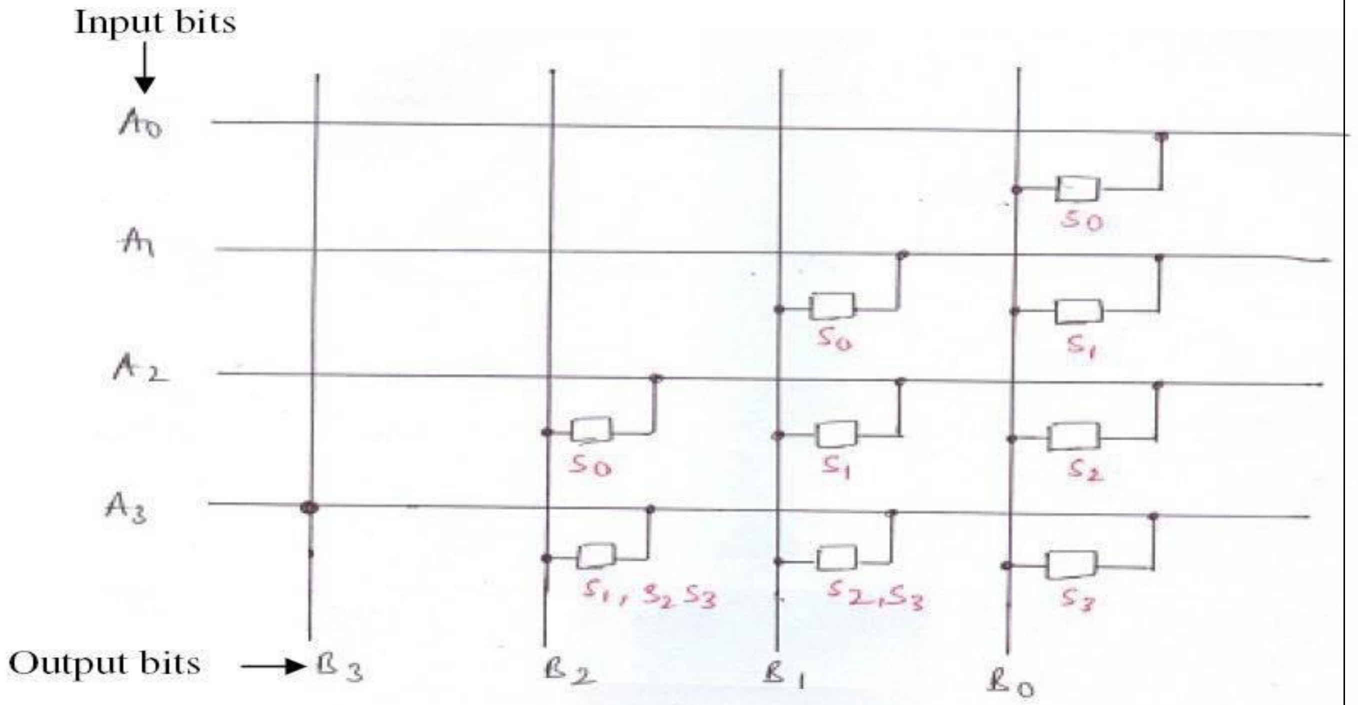
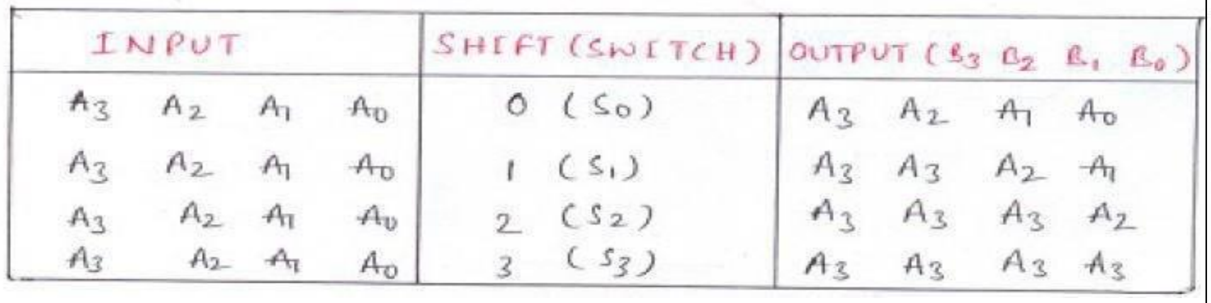


Fig 2.3 A Barrel Shifter





INPUT	SHIFT (SWITCH)	OUTPUT (B3 B2 B1 B0)
A3 A2 A1 A0	0 (S0)	A3 A2 A1 A0
A3 A2 A1 A0	1 (S1)	A3 A3 A2 A1
A3 A2 A1 A0	2 (S2)	A3 A3 A3 A2
A3 A2 A1 A0	3 (S3)	A3 A3 A3 A3

Fig 2.4 Implementation of a 4 bit Shift Right Barrel Shifter

Figure 2.4 depicts the implementation of a 4 bit shift right barrel shifter. Shift to right by 0, 1, 2 or 3 bit positions can be controlled by setting the control inputs appropriately.

Multiply and Accumulate Unit

Most of the DSP applications require the computation of the sum of the products of a series of successive multiplications. In order to implement such functions a special unit called a multiply and Accumulate (MAC) unit is required. A MAC consists of a multiplier and a special register called Accumulator. MACs are used to implement the functions of the type A+BC. A typical MAC unit is as shown in the figure 2.5.

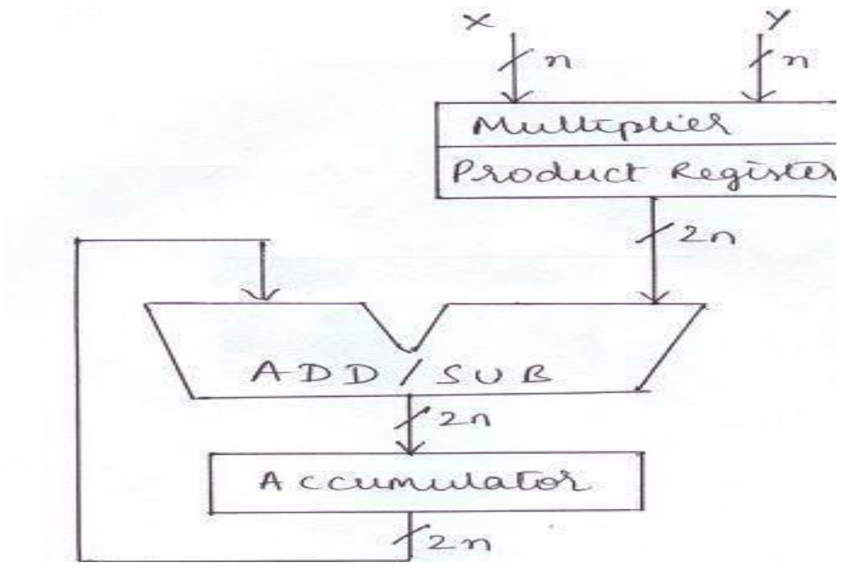


Fig 2.5 A MAC Unit

Although addition and multiplication are two different operations, they can be performed in parallel. By the time the multiplier is computing the product, accumulator can accumulate the product of the previous multiplications. Thus if N products are to be accumulated, N-1 multiplications can overlap with N-1 additions. During the very first multiplication, accumulator will be idle and during the last accumulation, multiplier will be idle. Thus N+1 clock cycles are required to compute the sum of N products.

Overflow and Underflow

While designing a MAC unit, attention has to be paid to the word sizes encountered at the input of the multiplier and the sizes of the add/subtract unit and the accumulator, as there is a possibility of overflow and underflows. Overflow/underflow can be avoided by using any of the following methods viz

a. Using shifters at the input and the output of the MAC
b. Providing guard bits in the accumulator
c. Using saturation logic

Shifters

Shifters can be provided at the input of the MAC to normalize the data and at the output to de normalize the same.

Guard bits

As the normalization process does not yield accurate result, it is not desirable for some applications. In such cases we have another alternative by providing additional bits called guard bits in the accumulator so that there will not be any overflow error. Here the add/subtract unit also has to be modified appropriately to manage the additional bits of the accumulator.

Saturation Logic

Overflow/ underflow will occur if the result goes beyond the most positive number or below the least negative number the accumulator can handle. Thus the overflow/underflow error can be resolved by loading the accumulator with the most positive number which it can handle at the time of overflow and the least negative number that it can handle at the time of underflow. This method is called as saturation logic. A schematic diagram of saturation logic is as shown in figure 2.7. In saturation logic, as soon as an overflow or underflow condition is satisfied the accumulator will be loaded with the most positive or least negative number overriding the result computed by the MAC unit.

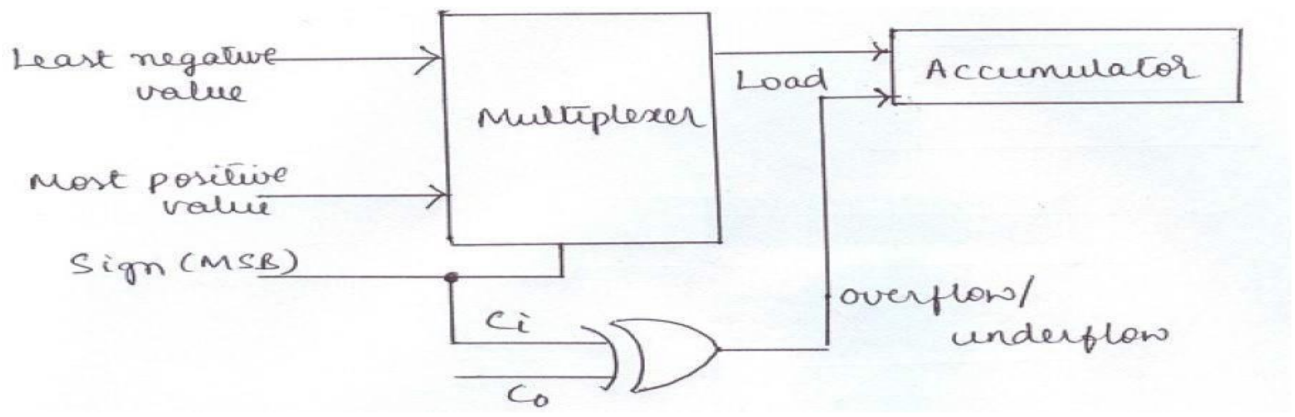


Fig 2.7: Schematic Diagram of the Saturation Logic

Arithmetic and Logic Unit

A typical DSP device should be capable of handling arithmetic instructions like ADD, SUB, INC, DEC etc and logical operations like AND, OR , NOT, XOR etc. The block diagram of a typical ALU for a DSP is as shown in the figure 2.8.
It consists of status flag register, register file and multiplexers.

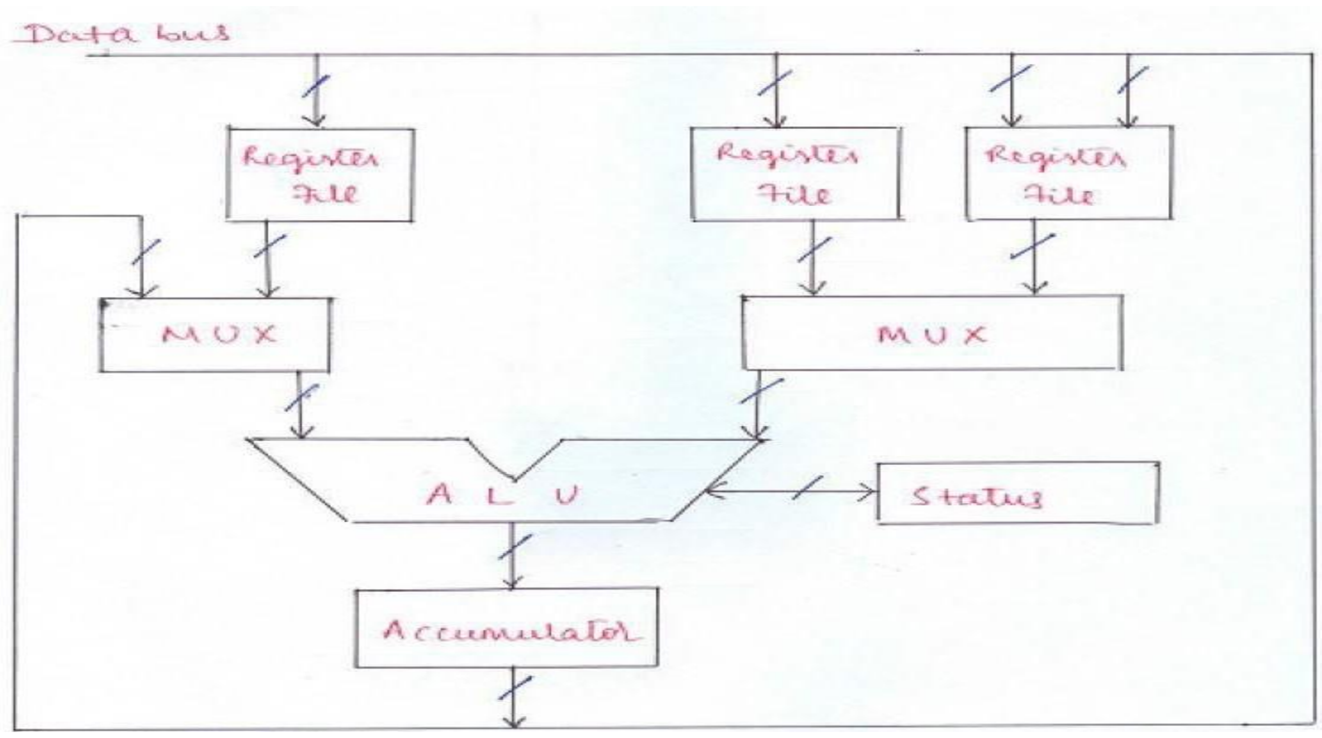


Fig 2.8 Arithmetic Logic Unit of a DSP

Status Flags

ALU includes circuitry to generate status flags after arithmetic and logic operations. These flags include sign, zero, carry and overflow.

Overflow Management

Depending on the status of overflow and sign flags, the saturation logic can be used to limit the accumulator content.

Register File

Instead of moving data in and out of the memory during the operation, for better speed, a large set of general purpose registers are provided to store the intermediate results.

Bus Architecture and Memory

Conventional microprocessors use Von Neumann architecture for memory management wherein the same memory is used to store both the program and data (Fig 2.9). Although this architecture is simple, it takes more number of processor cycles for the execution of a single instruction as the same bus is used for both data and program.

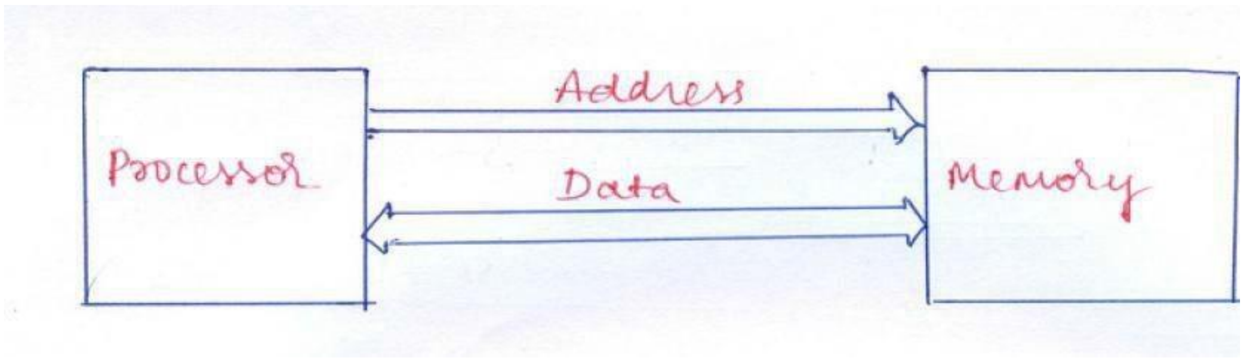


Fig 2.9 Von Neumann Architecture

In order to increase the speed of operation, separate memories were used to store program and data and a separate set of data and address buses have been given to both memories, the architecture called as Harvard Architecture. It is as shown in figure 2.10.

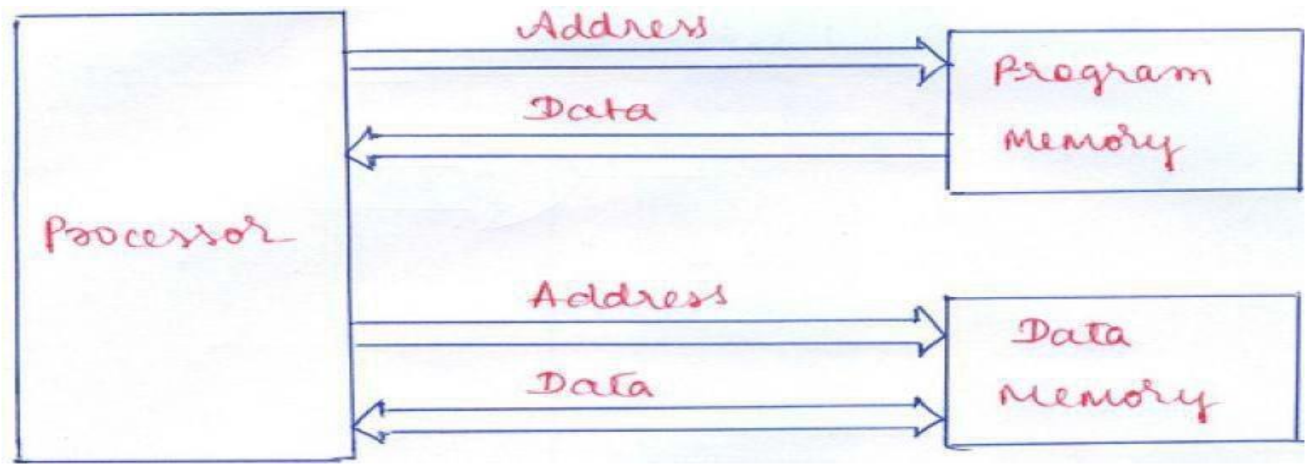


Fig 2.10 Harvard Architecture

Although the usage of separate memories for data and the instruction speeds up the processing, it will not completely solve the problem. As many of the DSP instructions require more than one operand, use of a single data memory leads to the fetch the operands one after the other, thus increasing the delay of processing. This problem can be overcome by using two separate data memories for storing operands separately, thus in a single clock cycle both the operands can be fetched together (Figure 2.11).

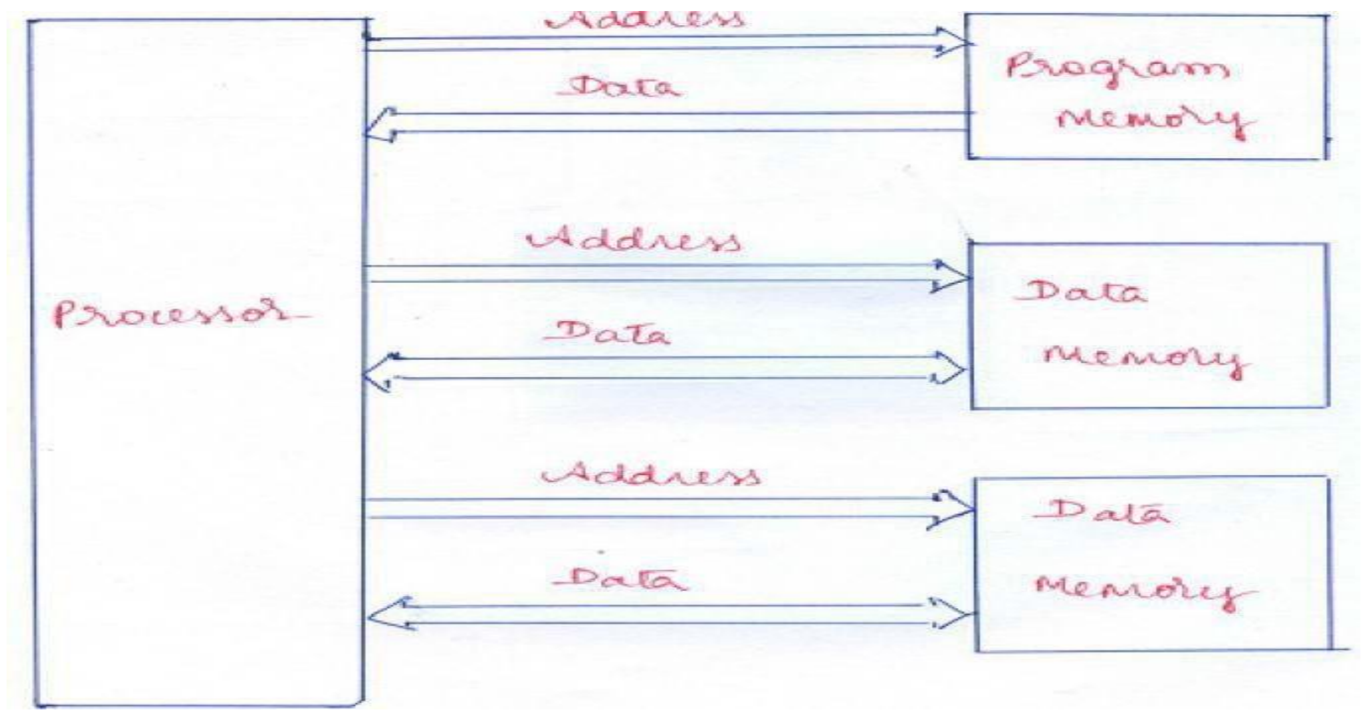


Fig 2.11 Harvard Architecture with Dual Data Memory

Although the above architecture improves the speed of operation, it requires more hardware and interconnections, thus increasing the cost and complexity of the system. Therefore there should be a trade off between the cost and speed while selecting memory architecture for a DSP.

On-chip Memories

In order to have a faster execution of the DSP functions, it is desirable to have some memory located on chip. As dedicated buses are used to access the memory, on chip memories are faster. Speed and size are the two key parameters to be considered with respect to the on-chip memories.

Speed

On-chip memories should match the speeds of the ALU operations in order to maintain the single cycle instruction execution of the DSP.

Size

In a given area of the DSP chip, it is desirable to implement as many DSP functions as possible. Thus the area occupied by the on-chip memory should be minimum so that there will be a scope for implementing more number of DSP functions on- chip.

Organization of On-chip Memories

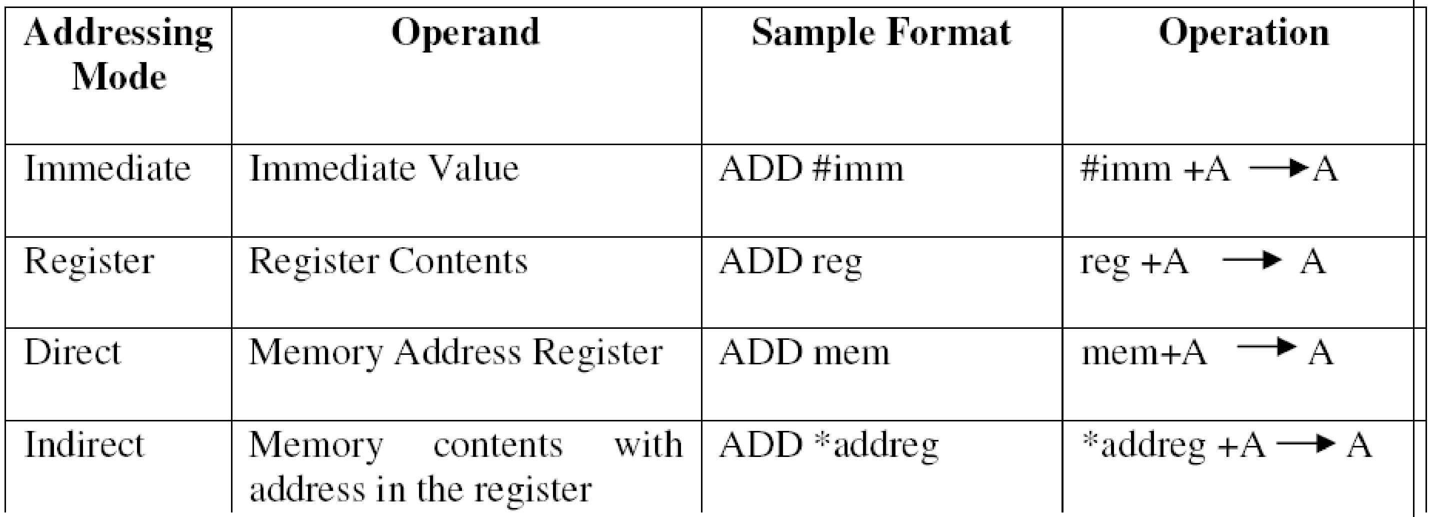
Ideally whole memory required for the implementation of any DSP algorithm has to reside on-chip so that the whole processing can be completed in a single execution cycle. Although it looks as a better solution, it consumes more space on chip, reducing the scope for implementing any functional block on-chip, which in turn reduces the speed of execution. Hence some other alternatives have to be thought of. The following are some other ways in which the on-chip memory can be organized.

a. As many DSP algorithms require instructions to be executed repeatedly, the instruction can be stored in the external memory, once it is fetched can reside in the instruction cache.
b. The access times for memories on-chip should be sufficiently small so that it can be accessed more than once in every execution cycle.
c. On-chip memories can be configured dynamically so that they can serve different purpose at different times.

Data Addressing Capabilities

Data accessing capability of a programmable DSP device is configured by means of its addressing modes. The summary of the addressing modes used in DSP is as shown in the table below.

Table 2.1 DSP Addressing Modes



Addressing Mode	Operand	Sample Format	Operation
Immediate	Immediate Value	ADD #imm	#imm +A $\longrightarrow$ A
Register	Register Contents	ADD reg	reg +A $\longrightarrow$ A
Direct	Memory Address Register	ADD mem	mem+A $\longrightarrow$ A
Indirect	Memory contents with address in the register	ADD *addreg	*addreg +A $\longrightarrow$ A

Immediate Addressing Mode
In this addressing mode, data is included in the instruction itself.

Register Addressing Mode
In this mode, one of the registers will be holding the data and the register has to be specified in the instruction.

Direct Addressing Mode
In this addressing mode, instruction holds the memory location of the operand.

Indirect Addressing Mode
In this addressing mode, the operand is accessed using a pointer. A pointer is generally a register, which holds the address of the location where the operands resides. Indirect addressing mode can be extended to inculcate automatic increment or decrement capabilities, which has lead to the following addressing modes.

Table 2.2 Indirect Addressing Modes

Addressing Mode	Sample Format	Operation
Post Increment	ADD *addreg+	A $\longrightarrow$ A + *addreg addreg $\longrightarrow$ addreg+1
Post Decrement	ADD *addreg-	A $\longrightarrow$ A + *addreg addreg $\longrightarrow$ addreg-1
Pre Increment	ADD +*addreg	addreg $\longrightarrow$ addreg+1 A $\longrightarrow$ A + *addreg
Pre Decrement	ADD -*addreg	addreg $\longrightarrow$ addreg-1 A $\longrightarrow$ A + *addreg
Post_Add_Offset	ADD *addreg, offsetreg+	A $\longrightarrow$ A + *addreg addreg $\longrightarrow$ addreg+offsetreg
Post_Sub_Offset	ADD *addreg, offsetreg-	A $\longrightarrow$ A + *addreg addreg $\longrightarrow$ addreg-offsetreg
Pre_Add_Offset	ADD offsetreg+,*addreg	addreg $\longrightarrow$ addreg+offsetreg A $\longrightarrow$ A + *addreg
Pre_Sub_Offset	ADD offsetreg-,*addreg	addreg $\longrightarrow$ addreg-offsetreg A $\longrightarrow$ A + *addreg

Special Addressing Modes

For the implementation of some real time applications in DSP, normal addressing modes will not completely serve the purpose. Thus some special addressing modes are required for such applications.

Circular Addressing Mode

While processing the data samples coming continuously in a sequential manner, circular buffers are used. In a circular buffer the data samples are stored sequentially from the initial location till the buffer gets filled up. Once the buffer gets filled up, the next data samples will get stored once again from the initial location. This process can go forever as long as the data samples are processed in a rate faster than the incoming data rate.

Circular Addressing mode requires three registers viz

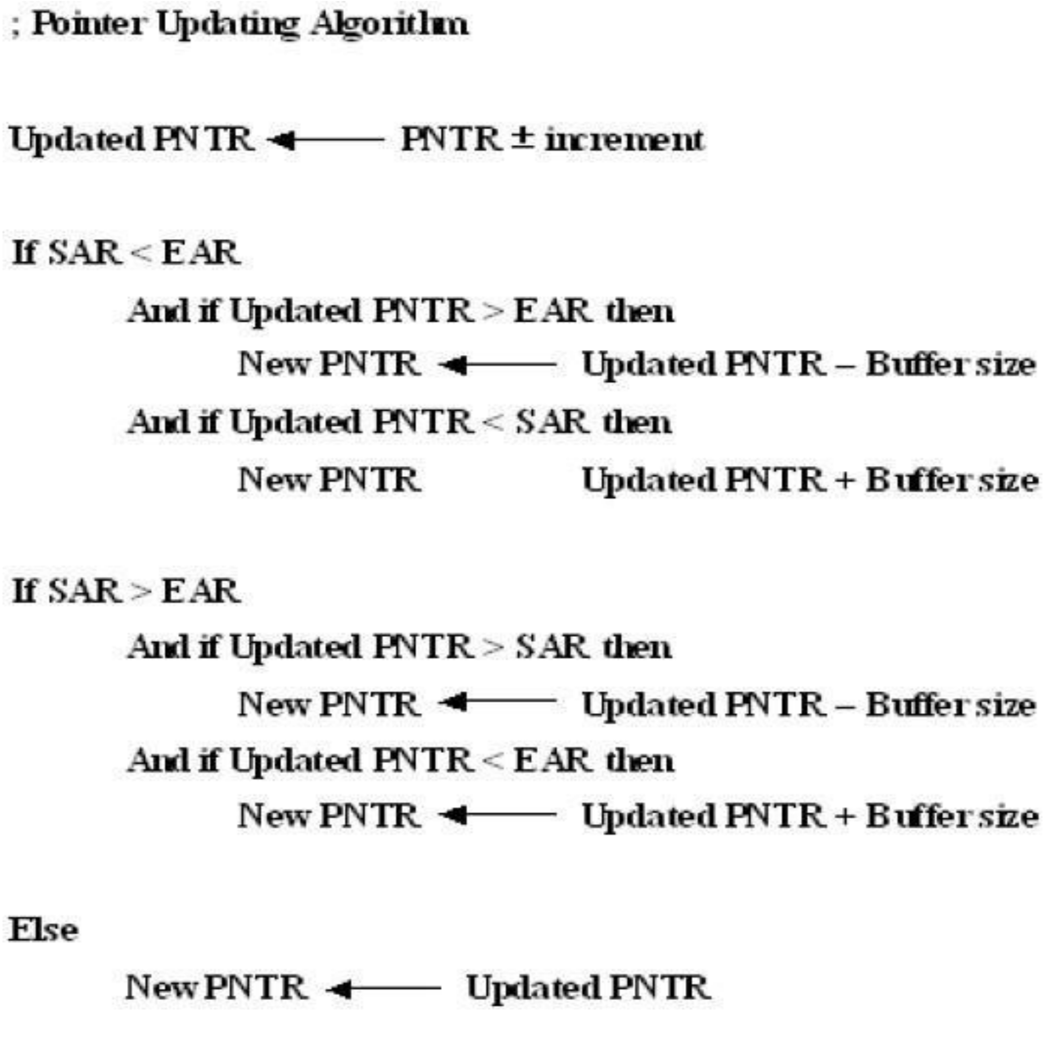
a. Pointer register to hold the current location (PNTR)

b. Start Address Register to hold the starting address of the buffer (SAR)

c. End Address Register to hold the ending address of the buffer (EAR)

There are four special cases in this addressing mode. They are

a. SAR < EAR & updated PNTR > EAR
b. SAR < EAR & updated PNTR < SAR
c. SAR >EAR & updated PNTR > SAR
d. SAR > EAR & updated PNTR < EAR
The buffer length in the first two case will be (EAR-SAR+1) whereas for the next tow cases (SAR-EAR+1)
The pointer updating algorithm for the circular addressing mode is as shown below.



```
; Pointer Updating Algorithm

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
```

Four cases explained earlier are as shown in the figure 2.12.

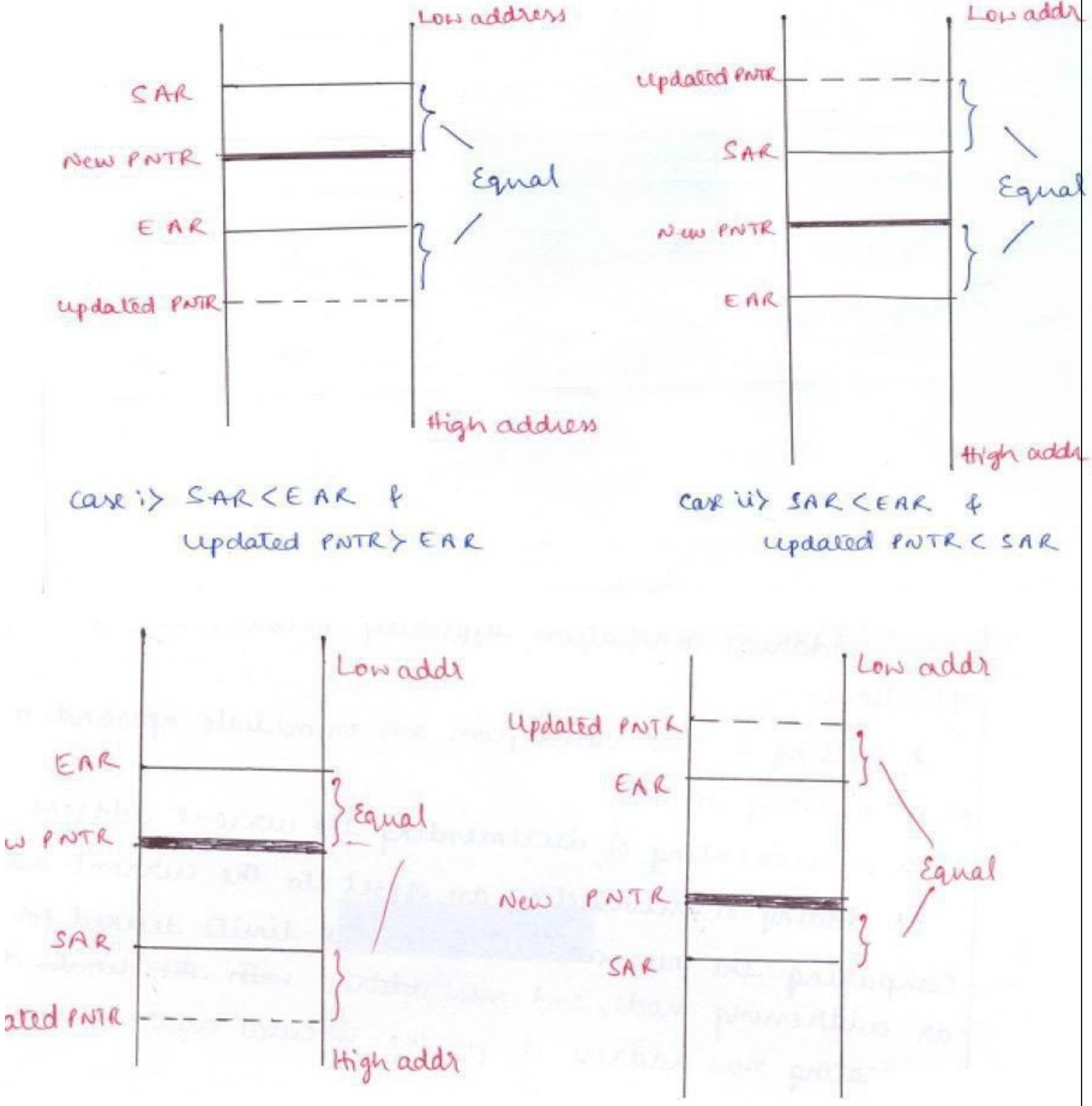


Fig 2.12 Special Cases in Circular Addressing Mode

Bit Reversed Addressing Mode

To implement FFT algorithms we need to access the data in a bit reversed manner. Hence a special addressing mode called bit reversed addressing mode is used to calculate the index of the next data to be fetched. It works as follows. Start with index 0. The present index can be calculated by adding half the FFT length to the previous index in a bit reversed manner, carry being propagated from MSB to LSB.

Current index= Previous index+ B (1/2(FFT Size))

Address Generation Unit

The main job of the Address Generation Unit is to generate the address of the operands required to carry out the operation. They have to work fast in order to satisfy the timing constraints. As the address generation unit has to perform some mathematical operations in order to calculate the operand address, it is provided with a separate ALU.

Address generation typically involves one of the following operations.

a. Getting value from immediate operand, register or a memory location

b. Incrementing/ decrementing the current address

c. Adding/subtracting the offset from the current address

d. Adding/subtracting the offset from the current address and generating new address according to circular addressing mode

e. Generating new address using bit reversed addressing mode

The block diagram of a typical address generation unit is as shown in figure 2.13.

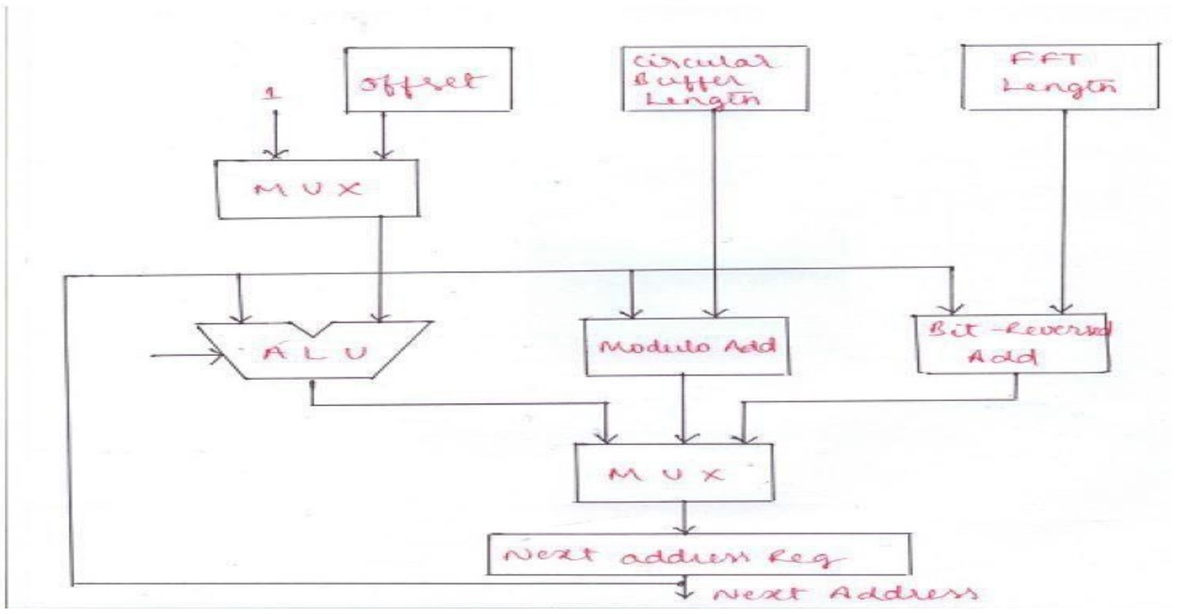


Fig 2.13 Address generation unit

Programmability and program Execution

A programmable DSP device should provide the programming capability involving branching, looping and subroutines. The implementation of repeat capability should be hardware based so that it can be programmed with minimal or zero overhead. A dedicated register can be used as a counter. In a normal subroutine call, return address has to be stored in a stack thus requiring memory access for storing and retrieving the return address, which in turn reduces the speed of operation. Hence a LIFO memory can be directly interfaced with the program counter.

Program Control

Like microprocessors, DSP also requires a control unit to provide necessary control and timing signals for the proper execution of the instructions. In microprocessors, the controlling is micro coded based where each instruction is divided into microinstructions stored in micro memory. As this mechanism is slower, it is not applicable for DSP applications. Hence in DSP the controlling is hardwired base where the Control unit is designed as a single, comprehensive, hardware unit. Although it is more complex it is faster.

Program Sequencer

It is a part of the control unit used to generate instruction addresses in sequence needed to access instructions. It calculates the address of the next instruction to be fetched. The next address can be from one of the following sources.

a. Program Counter
b. Instruction register in case of branching, looping and subroutine calls
c. Interrupt Vector table
d. Stack which holds the return address

The block diagram of a program sequencer is as shown in figure 2.14.

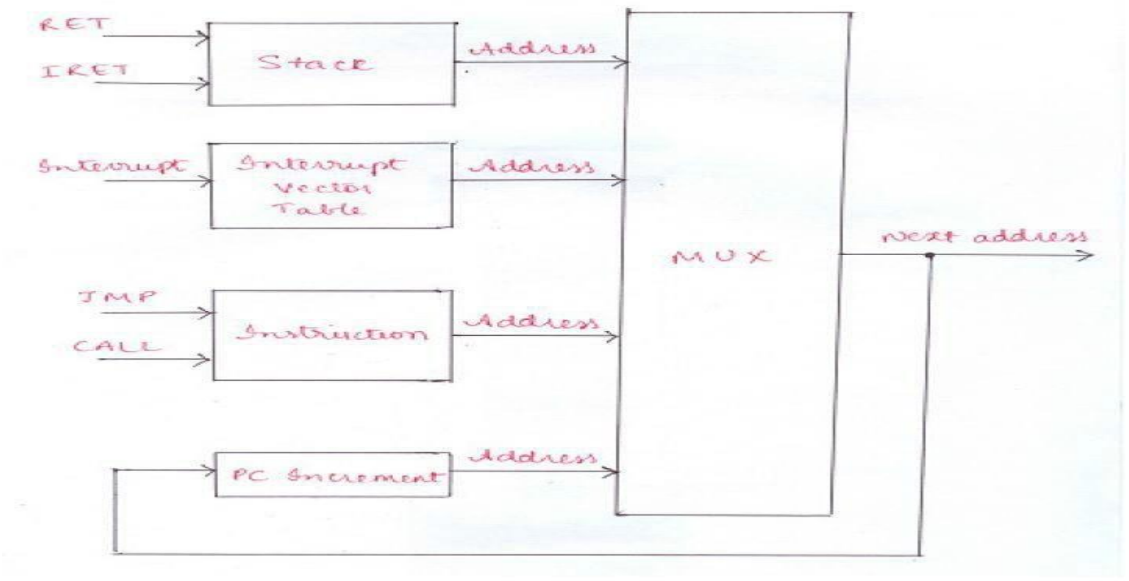


Fig 2.14 Program Sequencer

Program sequencer should have the following circuitry:

a. PC has to be updated after every fetch

b. Counter to hold count in case of looping

c. A logic block to check conditions for conditional jump instructions

d. Condition logic-status flag

Problems:

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.

<u>Solution:-</u>

y(n)= ∑h(i) x(n-i) n=0,1,2…

In order to implement the above operation in a DSP, the architecture requires the following features

i. A RAM to store the signal samples x (n)
ii. A ROM to store the filter coefficients h (n)
iii. An MAC unit to perform Multiply and Accumulate operation
iv. An accumulator to store the result immediately
v. A signal pointer to point the signal sample in the memory
vi. A coefficient pointer to point the filter coefficient in the memory
vii. A counter to keep track of the count
viii. A shifter to shift the input samples appropriately

2). It is required to find the sum of 64, 16 bit numbers. How many bits should the accumulator have so that the sum can be computed without the occurrence of overflow error or loss of accuracy?

The sum of 64, 16 bit numbers can grow up to (16+ log2 64)=22 bits long. Hence the accumulator should be 22 bits long in order to avoid overflow error from occurring.

1. In the previous problem, 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?

   As the length of the accumulator is fixed, the operands have to be shifted by an amount of log2 64 = 6 bits prior to addition operation, in order to avoid the condition of overflow.

2. If all the numbers in the previous problem are fixed point integers, what is the actual sum of the numbers?

   The actual sum can be obtained by shifting the result by 6 bits towards left side after the sum being computed. Therefore

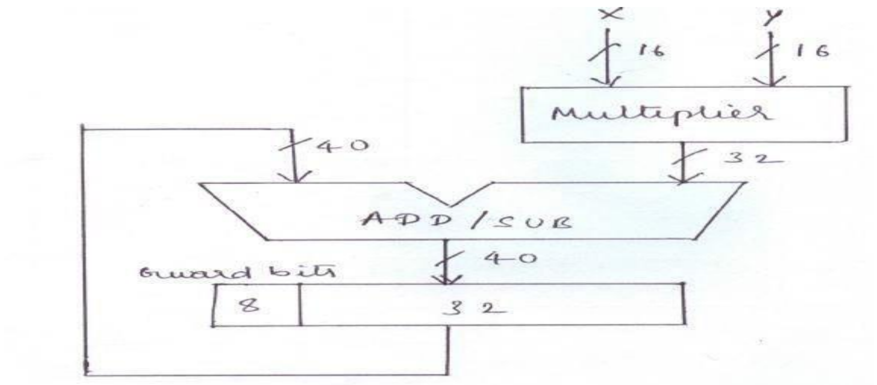
   Actual Sum= Accumulator content X 2^{6}

3. 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 100nsec, what will be the total time required to complete the operation?

As N=256 in this case, MAC unit requires N+1=257execution cycles. As the single MAC execution time is 100nsec, the total time required will be, (257*100nsec)=25.7usec

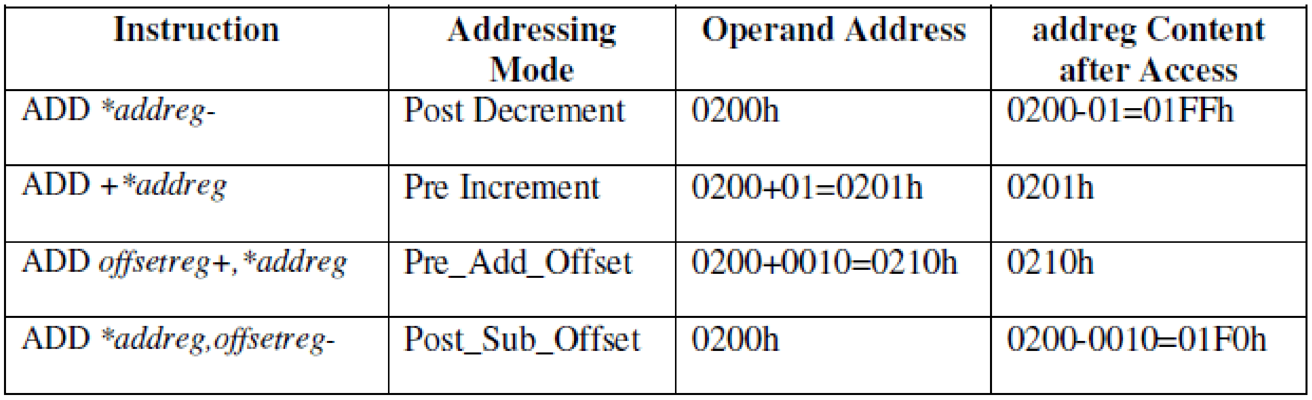
4. Consider a MAC unit whose inputs are 16 bit numbers. If 256 products are to be summed up in this MAC, how many guard bits should be provided for the accumulator to prevent overflow condition from occurring?

As it is required to calculate the sum of 256, 16 bit numbers, the sum can be as long as (16+ log2 256)=24 bits. Hence the accumulator should be capable of handling these 22 bits. Thus the guard bits required will be (24-16)= 8 bits.
The block diagram of the modified MAC after considering the guard or extention bits is as shown in the figure



5. 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 *addreg* after the memory access? Assume that the initial contents of the *addreg* and the *offsetreg* are 0200h and 0010h, respectively.

a. ADD **addreg*
b.ADD +addreg***
***c. ADD* offsetreg+,*addreg**
d. ADD **addreg,offsetreg-*



Instruction	Addressing Mode	Operand Address	addreg Content after Access
ADD **addreg-*	Post Decrement	0200h	0200-01=01FFh
ADD +**addreg*	Pre Increment	0200+01=0201h	0201h
ADD *offsetreg+,*addreg*	Pre_Add_Offset	0200+0010=0210h	0210h
ADD **addreg,offsetreg-*	Post_Sub_Offset	0200h	0200-0010=01F0h

6. 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
Buffer Length= (EAR-SAR+1) = 020F-0200+1=10h
a. New Address Pointer= Updated Pointer-buffer length = 0212-10=0202h
b. New Address Pointer= Updated Pointer+ buffer length = 01FC+10=020Ch

7. Repeat the previous problem for SAR= 0210h and EAR=0201h
Buffer Length= (SAR-EAR+1)= 0210-0201+1=10h
c. New Address Pointer= Updated Pointer- buffer length = 0212-10=0202h
d. New Address Pointer= Updated Pointer+ buffer length = 01FC+10=020Ch

9. Compute the indices for an 8-point FFT using Bit reversed Addressing Mode
Start with index 0. Therefore the first index would be (000)
Next index can be calculated by adding half the FFT length, in this case it is (100) to the previous index. i.e. Present Index= (000)+B (100)= (100)
Similarly the next index can be calculated as
Present Index= (100)+B (100)= (010)
The process continues till all the indices are calculated. The following table summarizes the calculation.

Index in Binary	BCD value	Bit reversed index	BCD value
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

UNIT-3

Programmable Digital Signal Processors

Introduction:

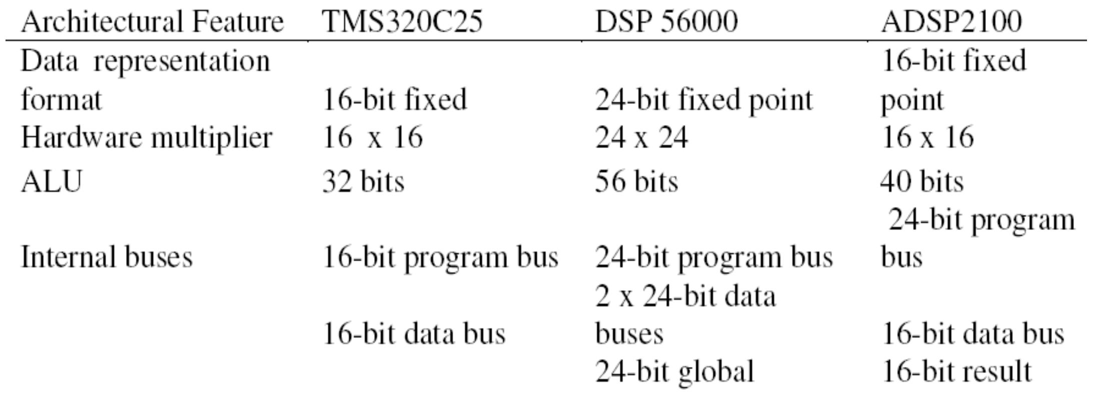
Leading manufacturers of integrated circuits such as Texas Instruments (TI), Analog devices & Motorola manufacture the digital signal processor (DSP) chips. These manufacturers have developed a range of DSP chips with varied complexity.
The TMS320 family consists of two types of single chips DSPs: 16-bit fixed point &32-bit floating-point. These DSPs possess the operational flexibility of high-speed controllers and the numerical capability of array processors

Commercial Digital Signal-Processing Devices:

There are several families of commercial DSP devices. Right from the early eighties, when these devices began to appear in the market, they have been used in numerous applications, such as communication, control, computers, Instrumentation, and consumer electronics. The architectural features and the processing power of these devices have been constantly upgraded based on the advances in technology and the application needs. However, their basic versions, most of them have Harvard architecture, a single-cycle hardware multiplier, an address generation unit with dedicated address registers, special addressing modes, on-chip peripherals interfaces. Of the various families of programmable DSP devices that are commercially available, the three most popular ones are those

from Texas Instruments, Motorola, and Analog Devices. Texas Instruments was one of the first to come out with a commercial programmable DSP with the introduction of its TMS32010 in 1982.

Summary of the Architectural Features of three fixed-Points DSPs



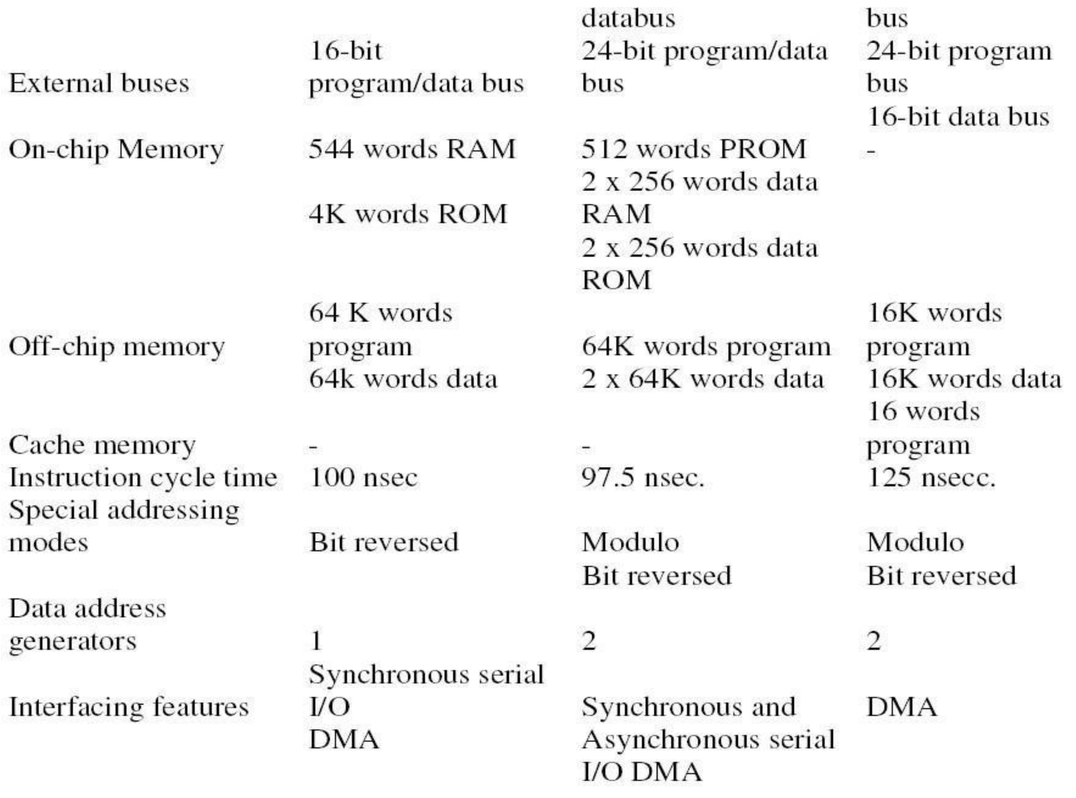
Architectural Feature	TMS320C25	DSP 56000	ADSP2100
Data representation format	16-bit fixed	24-bit fixed point	16-bit fixed point
Hardware multiplier	16 x 16	24 x 24	16 x 16
ALU	32 bits	56 bits	40 bits
Internal buses	16-bit program bus	24-bit program bus	24-bit program bus
	16-bit data bus	2 x 24-bit data buses	16-bit data bus
		24-bit global	16-bit result

The architecture of TMS320C54xx digital signal processors:

TMS320C54xx processors retain in the basic Harvard architecture of their predecessor, TMS320C25, but have several additional features, which improve their performance over it. Figure 3.1 shows a functional block diagram of TMS320C54xx processors. They have one program and three data memory spaces with separate buses, which provide simultaneous accesses to program instruction and two data operands and enables writing of result at the same time. Part of the memory is implemented on-chip and consists of combinations of ROM, dual-access RAM, and single-access RAM. Transfers between the memory spaces are also possible.

The central processing unit (CPU) of TMS320C54xx processors consists of a 40- bit arithmetic logic unit (ALU), two 40-bit accumulators, a barrel shifter, a 17x17 multiplier, a 40-bit adder, data address generation logic (DAGEN) with its own arithmetic unit, and program address generation logic (PAGEN). These major functional units are supported by a number of registers and logic in the architecture. A powerful instruction set with a hardware-supported, single-instruction repeat and block repeat operations, block memory move instructions, instructions that pack two or three simultaneous reads, and arithmetic instructions with parallel store and load make these devices very efficient for running high-speed DSP algorithms.

Several peripherals, such as a clock generator, a hardware timer, a wait state generator, parallel I/O ports, and serial I/O ports, are also provided on-chip. These peripherals make it convenient to interface the signal processors to the outside world. In these following sections, we examine in detail the various architectural features of the TMS320C54xx family of processors.



		databus	bus
External buses	16-bit program/data bus	24-bit program/data bus	24-bit program bus 16-bit data bus
On-chip Memory	544 words RAM 4K words ROM	512 words PROM 2 x 256 words data RAM 2 x 256 words data ROM	-
Off-chip memory	64 K words program 64k words data	64K words program 2 x 64K words data	16K words program 16K words data
Cache memory	-	-	16 words program
Instruction cycle time	100 nsec	97.5 nsec.	125 nsecc.
Special addressing modes	Bit reversed	Modulo Bit reversed	Modulo Bit reversed
Data address generators	1	2	2
Interfacing features	Synchronous serial I/O DMA	Synchronous and Asynchronous serial I/O DMA	DMA

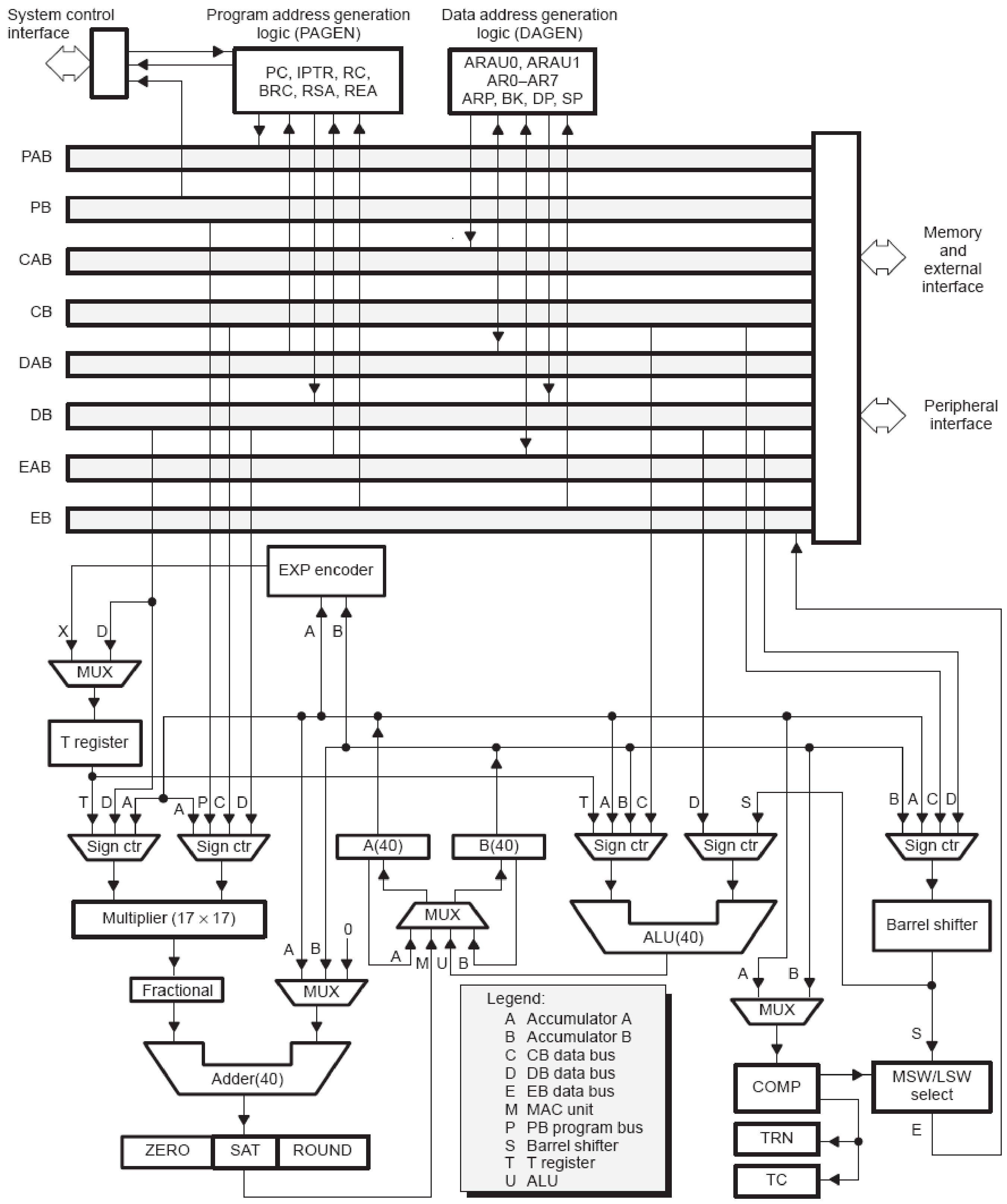


Figure 3.1.Functional architecture for TMS320C54xx processors.

<u>Bus Structure:</u>

The performance of a processor gets enhanced with the provision of multiple buses to provide simultaneous access to various parts of memory or peripherals. The 54xx architecture is built around four pairs of 16-bit buses with each pair consisting of an address bus and a data bus. As shown in Figure 3.1, these are The program bus pair (**PAB, PB**); which carries the instruction code from the program memory. Three data bus pairs (**CAB, CB; DAB, DB**; and **EAB, EB**); which interconnected the various units within the CPU. In Addition the pair CAB, CB and DAB, DB are used to read from the data memory, while The pair **EAB, EB**; carries the data to be written to the memory. The ‘54xx can generate up to two data-memory addresses per cycle using the two auxiliary register arithmetic unit (ARAU0 and ARAU1) in the DAGEN block. This enables accessing two operands simultaneously.

<u>Central Processing Unit (CPU):</u>

The ‘54xx CPU is common to all the ‘54xx devices. The ’54xx CPU contains a 40-bit arithmetic logic unit (**ALU**); two 40-bit accumulators (**A** and **B**); a barrel shifter; a
17 x 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 3.2 show 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 accumulators is divided into three parts: guards bits (bits 39-32), high-order word (bits-31-16), and low-order word (bits 15- 0), which can be stored and retrieved individually. Each accumulator is memory-mapped and partitioned. It can be configured as the destination registers. The guard bits are used as a head margin for computations.

AG(39-32)	AH(31-16)	AL(15-0)

BG(39-32)	BH(31-16)	BL(15-0)

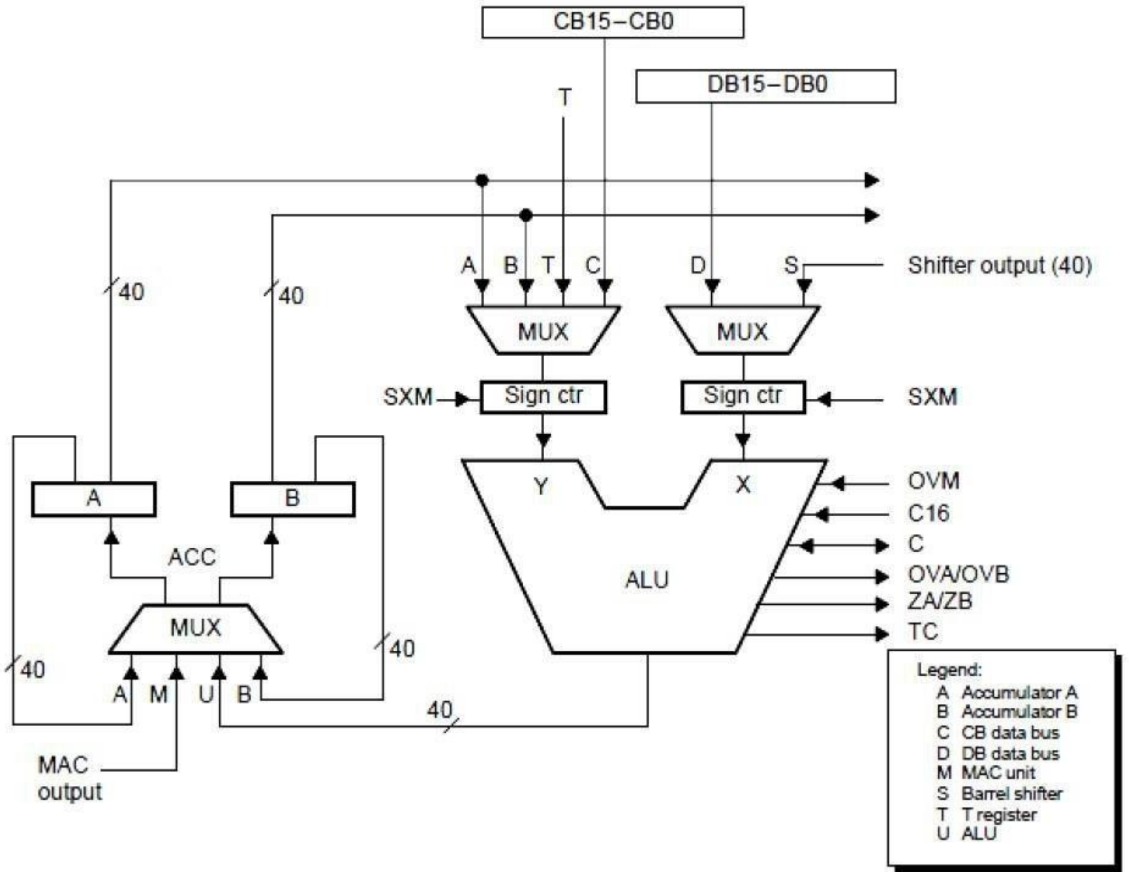


Figure 3.2.Functional diagram of the central processing unit of the TMS320C54xx processors.

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 count field of status registers ST1, or in the temporary
register T. Figure 3.3 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 with0s, and 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. An additional shift capability enables the processor to perform numerical scaling, bit extraction, extended arithmetic, and overflow prevention operations.

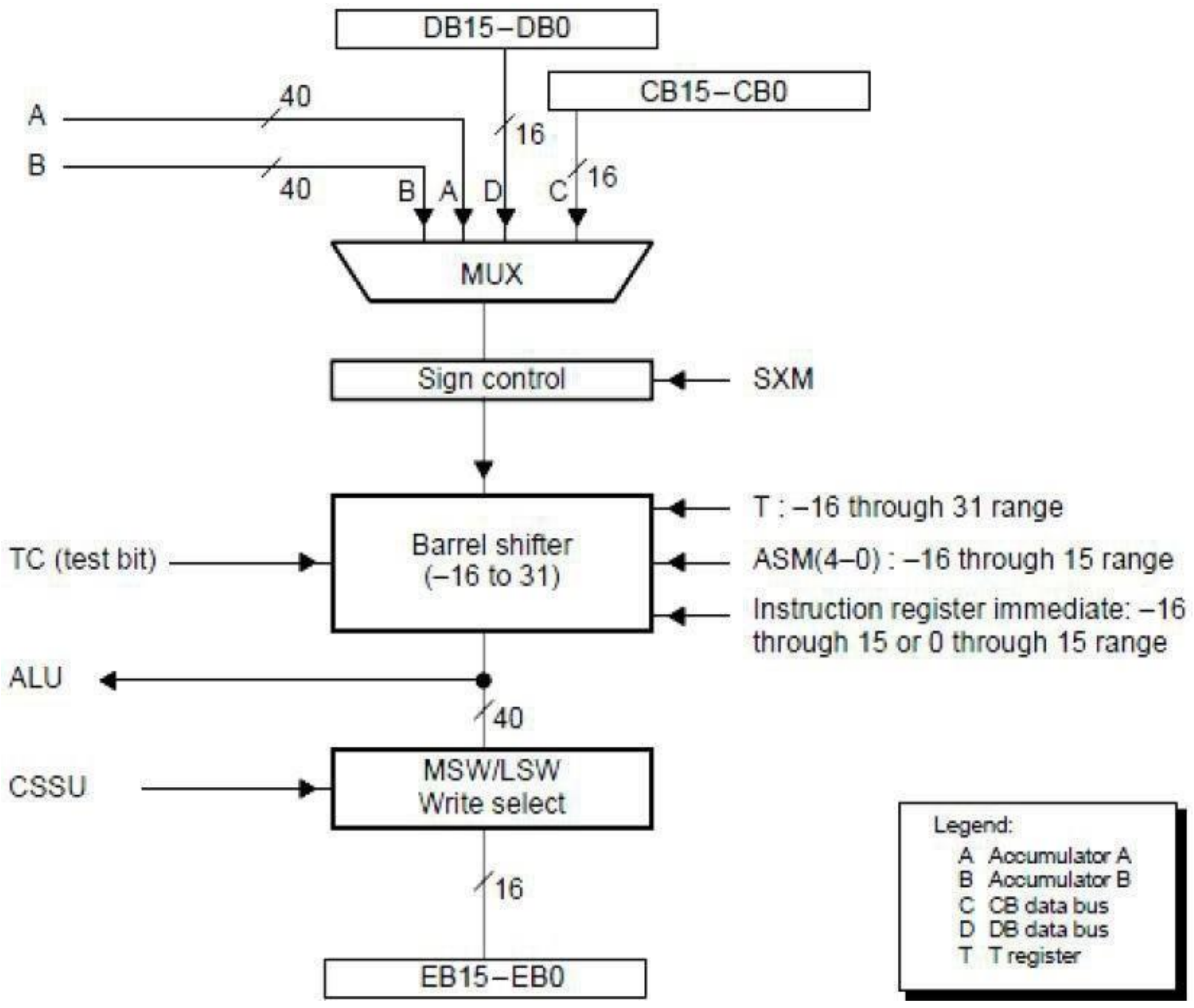


Figure 3.3.Functional diagram of the barrel shifter

Multiplier/adder unit: The kernel of the DSP device architecture is multiplier/adder unit. The multiplier/adder unit of TMS320C54xx devices performs 17 x 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 3.4 show 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 instructions, 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.

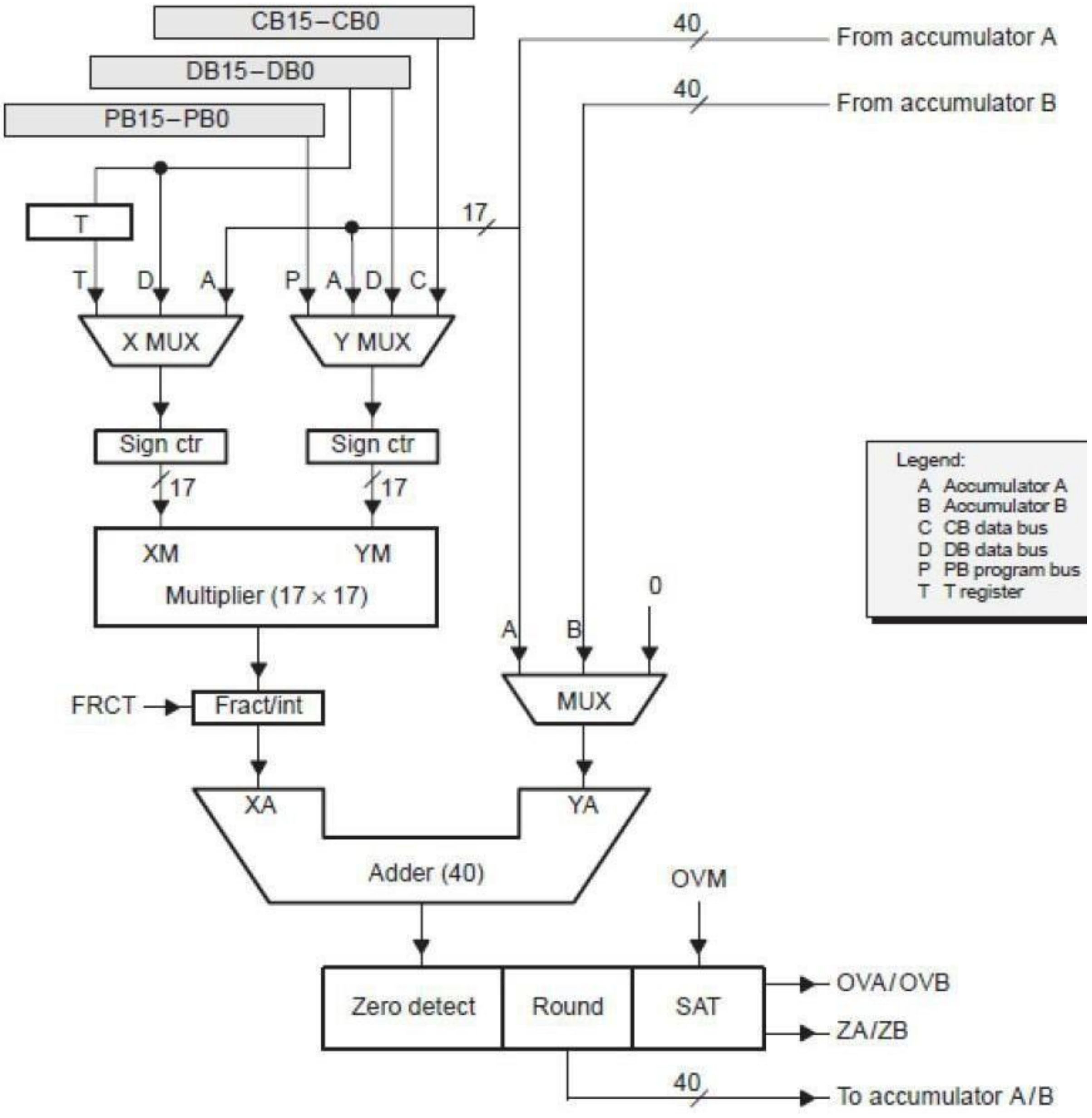
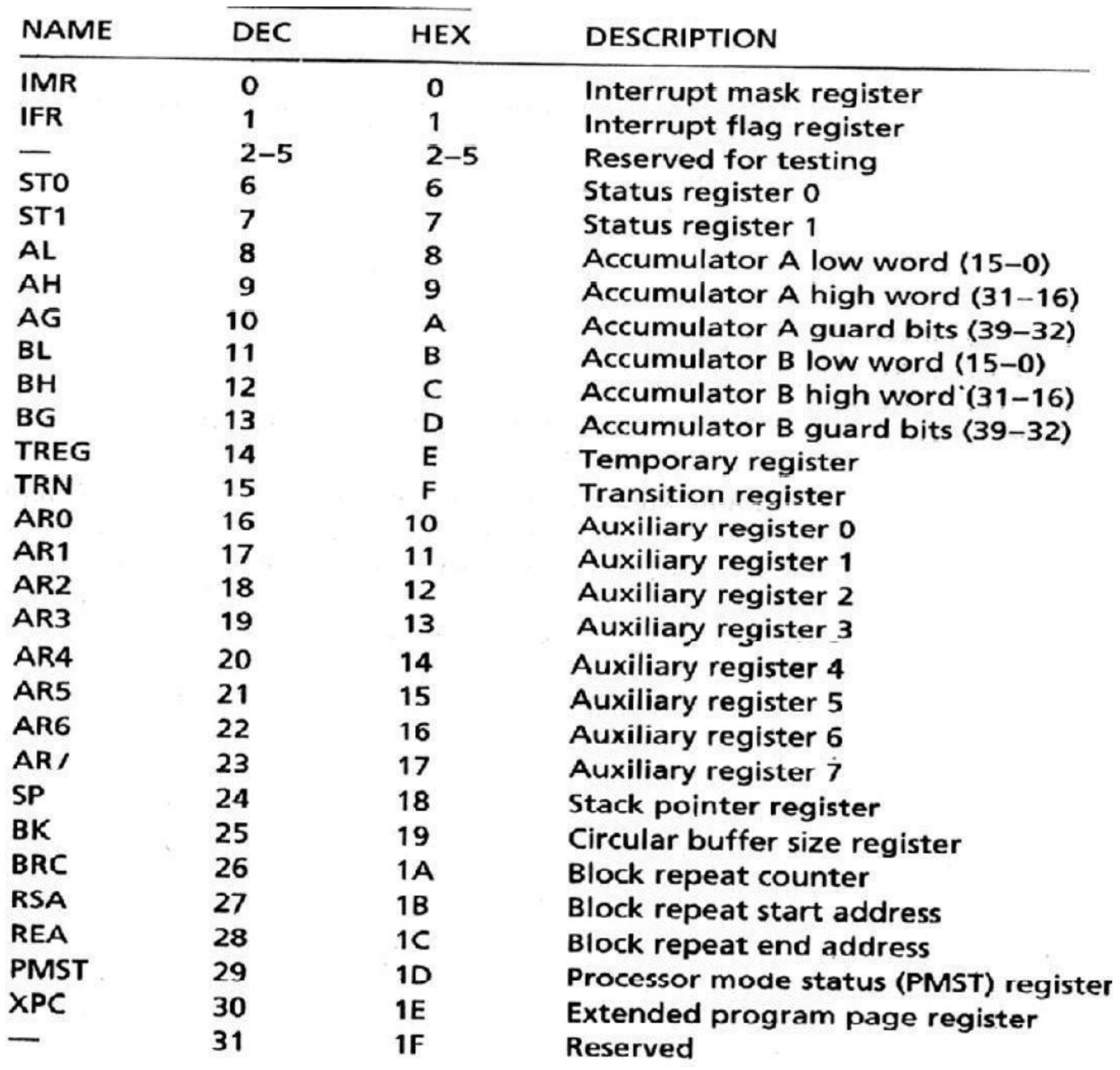


Figure 3.4. Functional diagram of the multiplier/adder unit of TMS320C54xx processors.

Internal Memory and Memory-Mapped Registers:

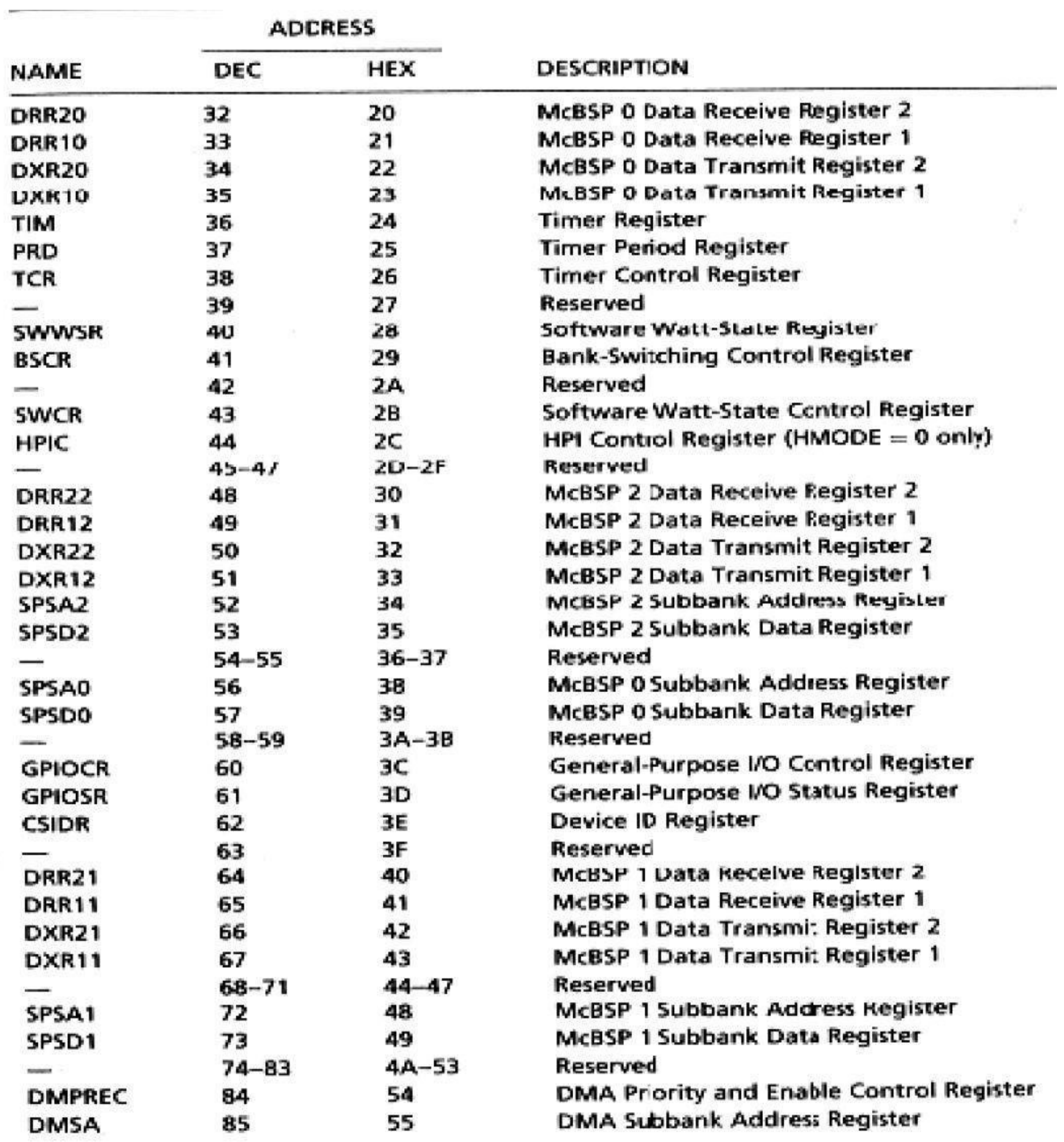
The amount and the types of memory of a processor have direct relevance to the efficiency and performance obtainable in implementations with the processors. The '54xx memory is organized into three individually selectable spaces: program, data, and I/O spaces. All '54xx devices contain both RAM and ROM. 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 peripherals registers are all located on page 0 of the data

memory. Figure 3.5(a) and (b) shows the internal CPU registers and peripheral registers with their addresses. The processors mode status (PMST) registers
that is used to configure the processor. It is a memory-mapped register located at address 1Dh on page 0 of the RAM. A part of on-chip ROM may contain a boot loader and look-up tables for function such as sine, cosine, μ- *law, and A*- law.



NAME	DEC	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

Figure 3.5(a) Internal memory-mapped registers of TMS320C54xx processors.



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 3.5(b).peripheral registers for the TMS320C54xx processors

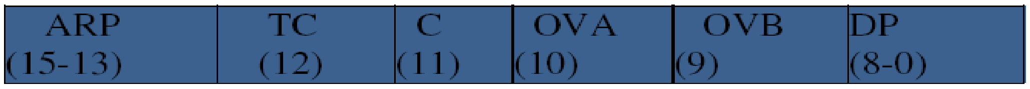
Status registers (ST0,ST1):

ST0: Contains the status of flags (OVA, OVB, C, TC) produced by arithmetic operations & bit manipulations.

ST1: Contain the status of various conditions & modes. Bits of ST0&ST1registers can be set or clear with the SSBX & RSBX instructions.

PMST: Contains memory-setup status & control information.

Status register0 diagram:

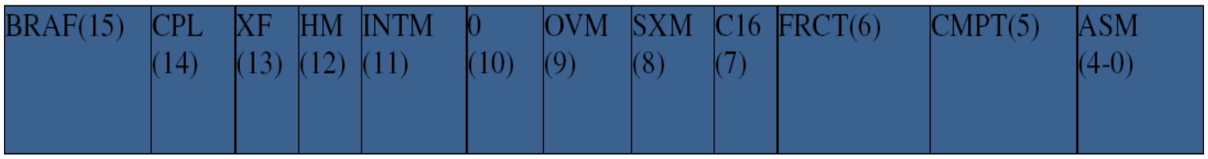


ARP (15-13)	TC (12)	C (11)	OVA (10)	OVB (9)	DP (8-0)

Figure 3.6(a). ST0 diagram

ARP: Auxiliary register pointer.
TC: Test/control flag.
C: Carry bit.
OVA: Overflow flag for accumulator A.
OVB: Overflow flag for accumulator B.
DP: Data-memory page pointer.

<u>Status register1 diagram:</u>



BRAF(15)	CPL (14)	XF (13)	HM (12)	INTM (11)	0 (10)	OVM (9)	SXM (8)	C16 (7)	FRCT(6)	CMPT(5)	ASM (4-0)

Figure 3.6(b). ST1 diagram

BRAF: Block repeat active flag

BRAF=0, the block repeat is deactivated.
BRAF=1, the block repeat is activated.

CPL: Compiler mode

CPL=0, the relative direct addressing mode using data page pointer is selected.
CPL=1, the relative direct addressing mode using stack pointer is selected.

HM: Hold mode, indicates whether the processor continues internal execution or acknowledge for external interface.

INTM: Interrupt mode, it globally masks or enables all interrupts.

INTM=0_all unmasked interrupts are enabled.
INTM=1_all masked interrupts are disabled.
0: Always read as 0

OVM: Overflow mode.

OVM=1_the destination accumulator is set either the most positive value or the most negative value.
OVM=0_the overflowed result is in destination accumulator.

SXM: Sign extension mode.

SXM=0 _Sign extension is suppressed.

SXM=1_Data is sign extended

C16: Dual 16 bit/double-Precision arithmetic mode.
C16=0_ALU operates in double-Precision arithmetic mode.
C16=1_ALU operates in dual 16-bit arithmetic mode.

FRCT: Fractional mode.

FRCT=1_the multiplier output is left-shifted by 1bit to compensate an extra sign bit.

CMPT: Compatibility mode.

CMPT=0_ ARP is not updated in the indirect addressing mode.
CMPT=1_ARP is updated in the indirect addressing mode.

ASM: Accumulator Shift Mode.

5 bit field, & specifies the Shift value within -16 to 15 range.

Processor Mode Status Register (PMST):

IPTR(15-7)	MP/MC(6)	OVLY(5)	AVIS(4)	DROM(3)	CLKOFF(2)	SMUL(1)	SST(0)

Figure 3.6(c).PMST register diagram

INTR: Interrupt vector pointer, point to the 128-word program page where the interrupt vectors reside.
MP/MC: Microprocessor/Microcomputer mode,
MP/MC=0, the on chip ROM is enabled.
MP/MC=1, the on chip ROM is enabled.

OVLY: RAM OVERLAY, OVLY enables on chip dual access data RAM blocks to be mapped into program space.

AVIS: It enables/disables the internal program address to be visible at the address pins.
DROM: Data ROM, DROM enables on-chip ROM to be mapped into data space.
CLKOFF: CLOCKOUT off.
SMUL: Saturation on multiplication.
SST: Saturation onstore.

Data Addressing Modes of TMS320C54X 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

1. Immediate addressing.
2. Absolute addressing.
3. Accumulator addressing.
4. Direct addressing.
5. Indirect addressing.
6. Memory mapped addressing
7. Stack addressing.

Immediate addressing:

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

Example: LD #20, DP.
RPT #0FFFFh.

Absolute Addressing:

The instruction contains a specified address in the operand.

1. Dmad addressing. MVDK Smem,dmad, MVDM dmad,MMR
2. Pmad addressing. MVDP Smem,pmad, MVPD pmem,Smad
3. PA addressing. PORTR PA, Smem,
4. *(lk) addressing .

Accumulator Addressing:

Accumulator content is used as address to transfer data between Program and Data memory.

Ex: READA *AR2

Direct Addressing:

Base address + 7 bits of value contained in instruction = 16 bit address. A page of 128 locations can be accessed without change in DP or SP.Compiler mode bit (CPL) in ST1 register is used.

If CPL =0 selects DP

CPL = 1 selects SP,

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.

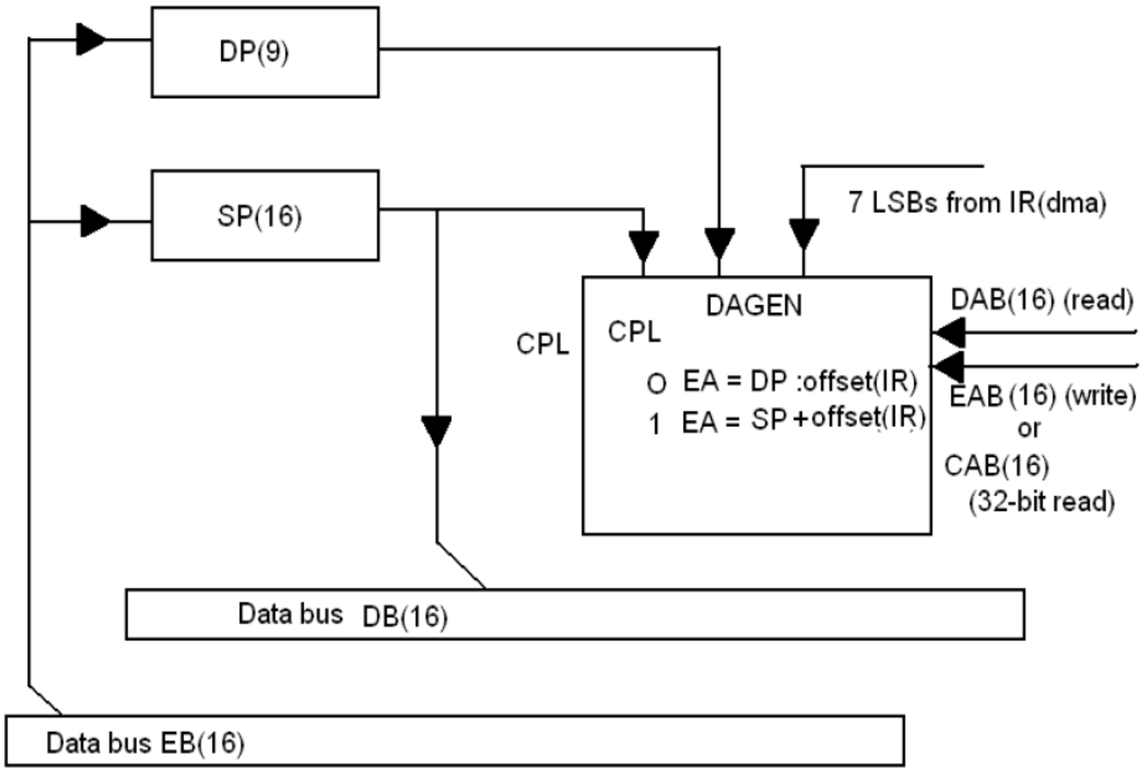


Figure 3.7 Block diagram of the direct addressing mode for TMS320C54xx Processors.

Indirect Addressing:

□ Data space isaccessed by address present in an auxiliary register.
TMS320C54xx have 8, 16 bit auxiliary register (AR0 – AR 7). Two auxiliary register arithmetic units (ARAU0 & ARAU1)
Used to access memory location in fixed step size. AR0 register is used for indexed and bit reverse addressing modes.
□ For single– operand addressing
MOD _ type of indirect addressing
ARF _ AR used for addressing
ARP depends on (CMPT) bit in ST1
CMPT = 0, Standard mode, ARP set to zero
CMPT = 1, Compatibility mode, Particularly AR selected by ARP

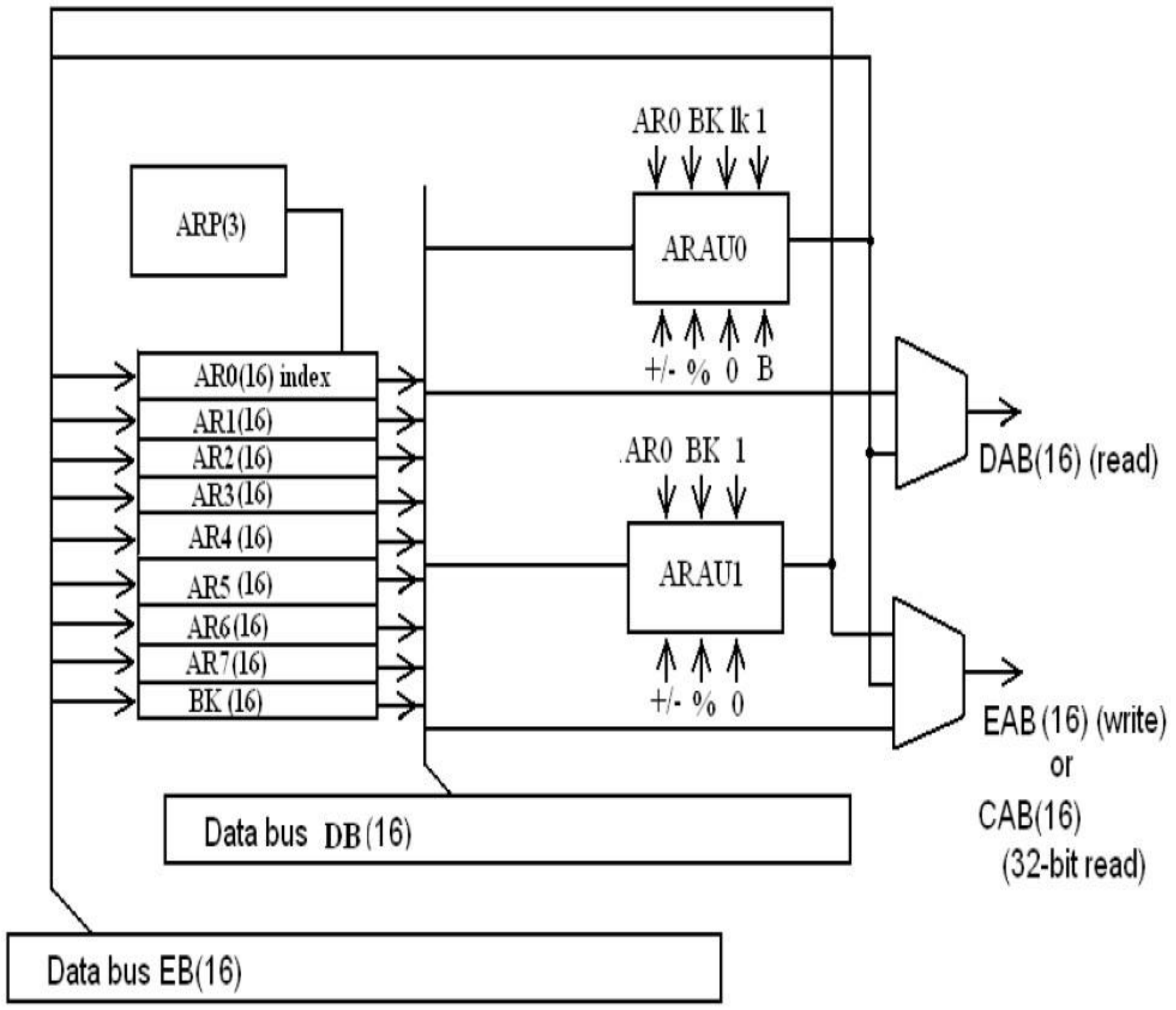


Figure 3.8 Block diagram of the indirect addressing mode for TMS320C54xx Processors.

Operand syntax	Function
*ARx	Addr = ARx;
*ARx -	Addr = ARx ; ARx = ARx -1
*ARx +	Addr = ARx; ARx = ARx +1
*+ARx	Addr = ARx+1; ARx = ARx +1
*ARx - 0B	Addr = ARx ; ARx = B(ARx – AR0)
*ARx - 0	Addr = Arx ; ARx = ARx – AR0
*ARx + 0	Addr = Arx ; ARx = ARx +AR0
*ARx + 0B	Addr = ARx ; ARx = B(ARx + AR0)
*ARx - %	Addr = ARx ; ARx = circ(ARx – 1)
*+AR – 0%	Addr = Arx; ARx = circ(ARx - AR0)
*ARx + %	Addr = ARx ; ARx = circ (ARx + 1)

Table 3.2 Indirect addressing options with a single data –memory operand.
Circular Addressing;

- Used in convolution, correlation and FIR filters.
- A circular buffer is a sliding window contains most recent data. Circular buffer of size R must start on a N-bit boundary, where 2N > R .
- The circular buffer size register (BK): specifies the size of circular buffer.
- Effective base address (EFB): By zeroing the N LSBs of a user selected AR (ARx).
- End of buffer address (EOB) : By repalcing the NLSBs of ARx with the N LSBs of BK. If 0 _ index + step < BK ; index = index +step;

else if index + step _ BK ; index = index + step - BK; else if
index + step < 0; index + step + BK

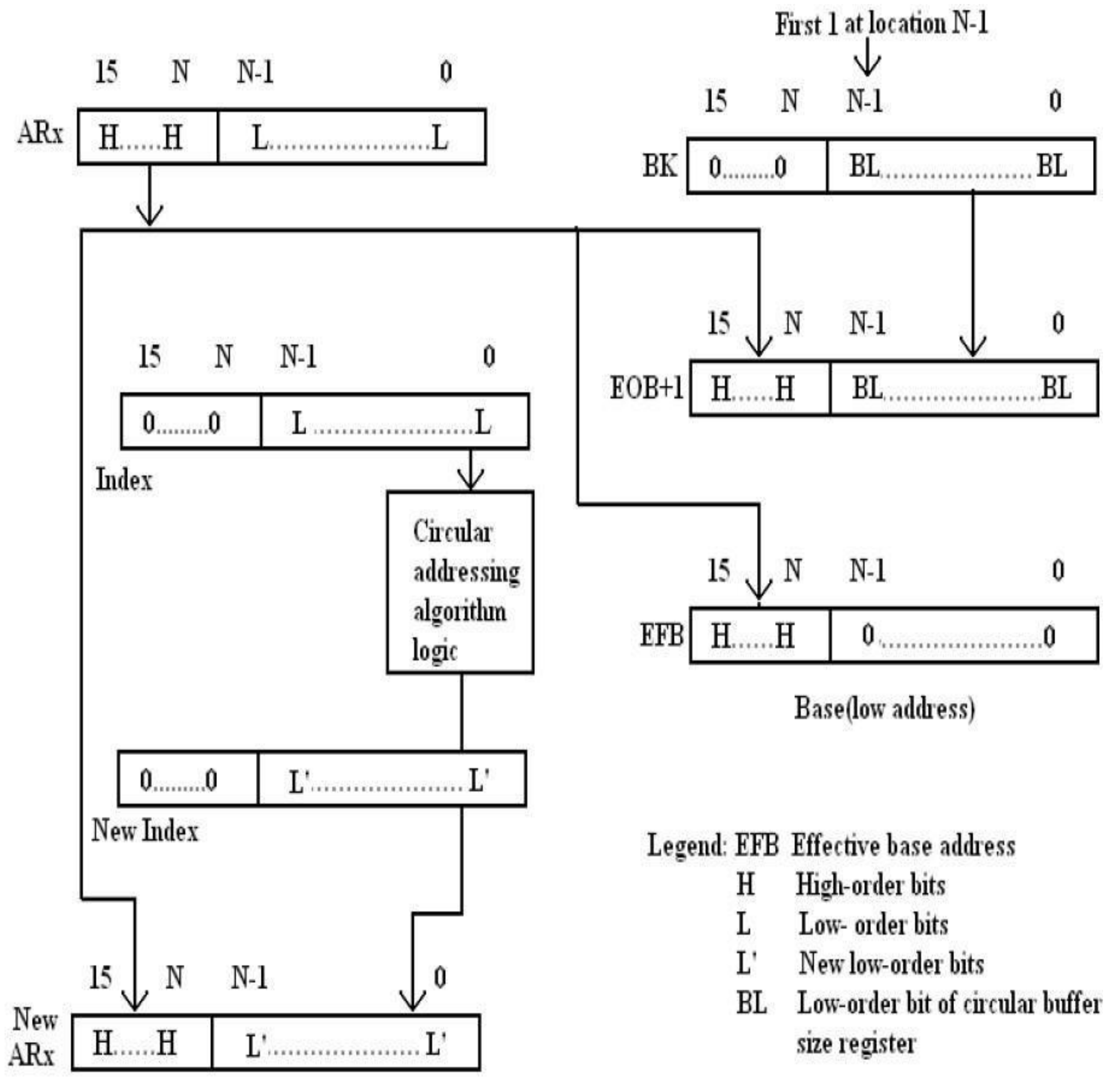


Figure 3.9 Block diagram of the circular addressing mode for TMS320C54xx Processors.

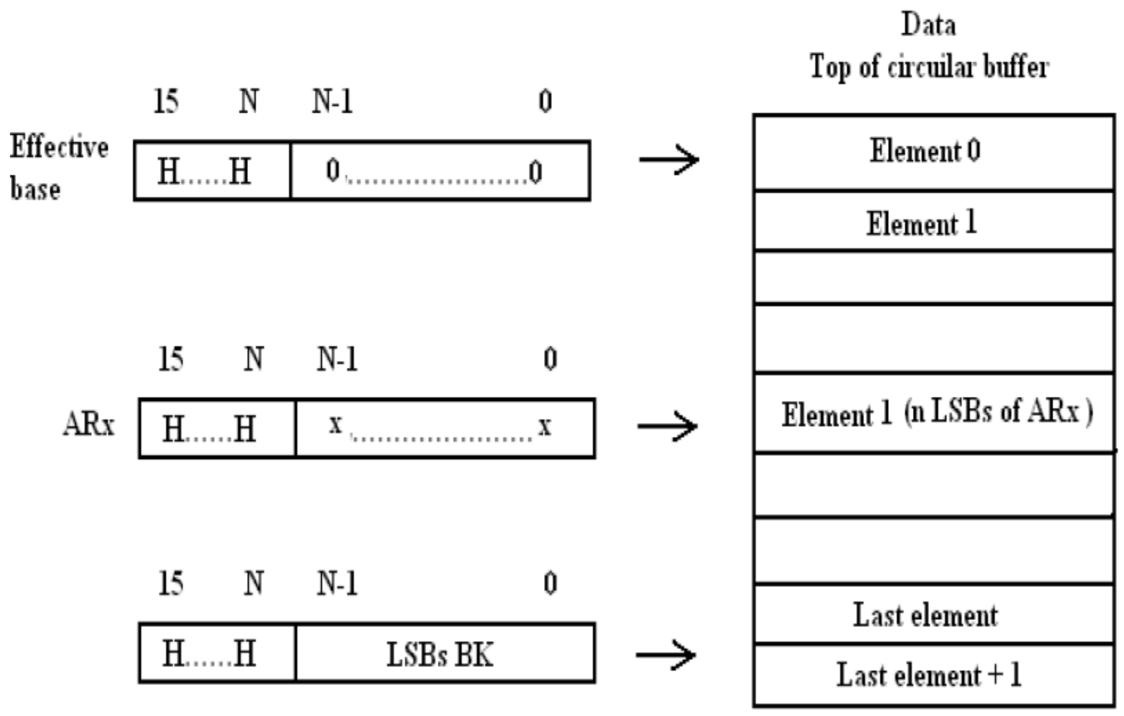


Figure 3.10 circular addressing mode implementation for TMS320C54xx Processors.

Bit-Reversed Addressing:

- Used for FFT algorithms.
- AR0 specifies one half of the size of the FFT.
- The value of AR0 = 2N-1: N = integer FFT size = 2N
- AR0 + AR (selected register) = bit reverse addressing.
- The carry bit propagating from left to right.

Dual-Operand Addressing:

Dual data-memory operand addressing is used for instruction 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, II. These instructions access operands using indirect addressing mode.

If in an instruction with a parallel store the source operand 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 capability to access two operands in a single cycle. Figure 3.11 shows how an address is generated using dual data-memory operand addressing.

15 - 8	7 - 6	5 - 4	3 - 2	1 - 0
Opcode	Xmod	Xar	Ymod	Yar

Name	Function
Opcode	This field contains the operation code for the instruction
Xmod	Defined the type of indirect addressing mode used for accessing the Xmem operand
XAR	Xmem AR selection field defines the AR that contains the address of Xmem
Ymod	Defies the type of inderect addressing mode used for accessing the Ymem operand
Yar	Ymem AR selection field defines the AR that contains the address of Ymem

Table 3.3.Function of the different field in dual data memory operand addressing

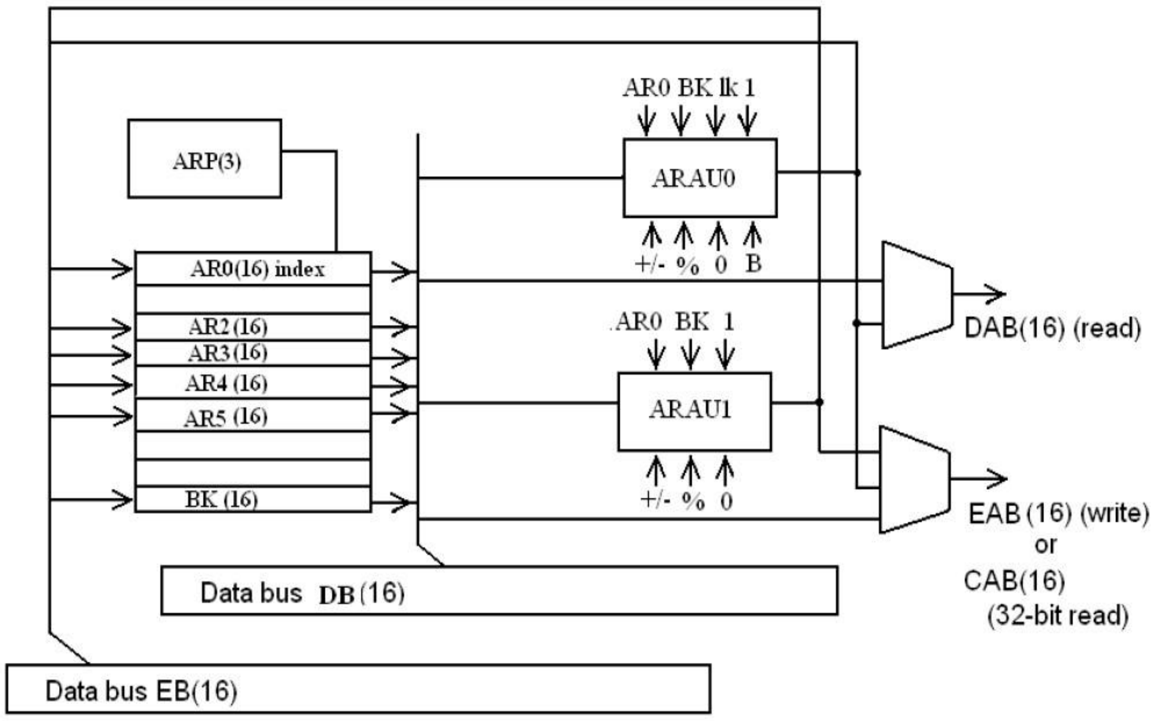


Figure 3.11 Block diagram of the Indirect addressing options with a dual data –memory operand.

Memory-Mapped Register Addressing:

- Used to modify the memory-mapped registers without affecting the current data page
- pointer (DP) or stack-pointer (SP)
    - Overhead for writing to a register is minimal
    - Works for direct and indirect addressing
    - Scratch –pad RAM located on data PAGE0 can be modified
- STM #x, DIRECT
- STM #tbl, AR1

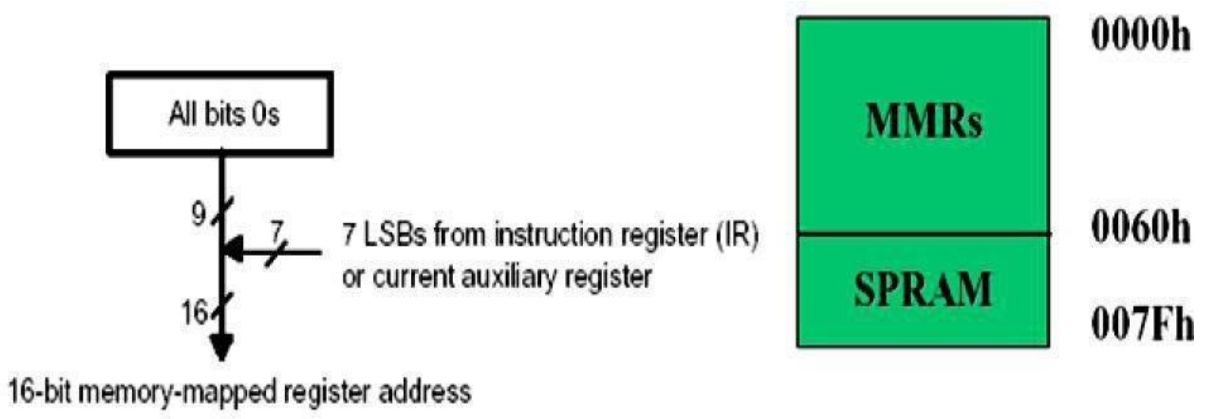

All bits 0s
9
7
7 LSBs from instruction register (IR) or current auxiliary register
16
16-bit memory-mapped register address
MMRs
SPRAM
0000h
0060h
007Fh

Figure 3.12.16 bit memory mapped register address generation.

3.4.7 Stack Addressing:

• Used to automatically store the program counter during interrupts and subroutines.
• Can be used to store additional items of context or to pass data values.
• Uses a 16-bit memory-mapped register, the stack pointer (SP).
• PSHD X2

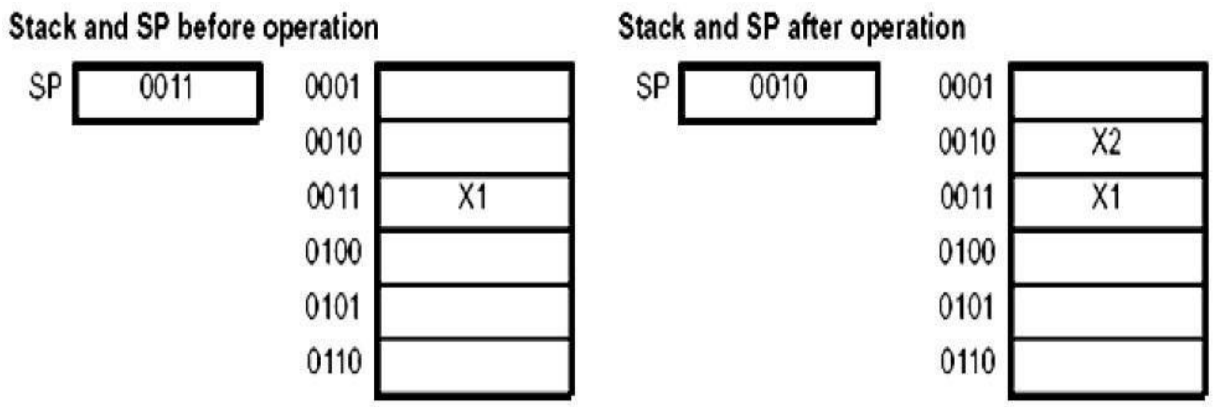
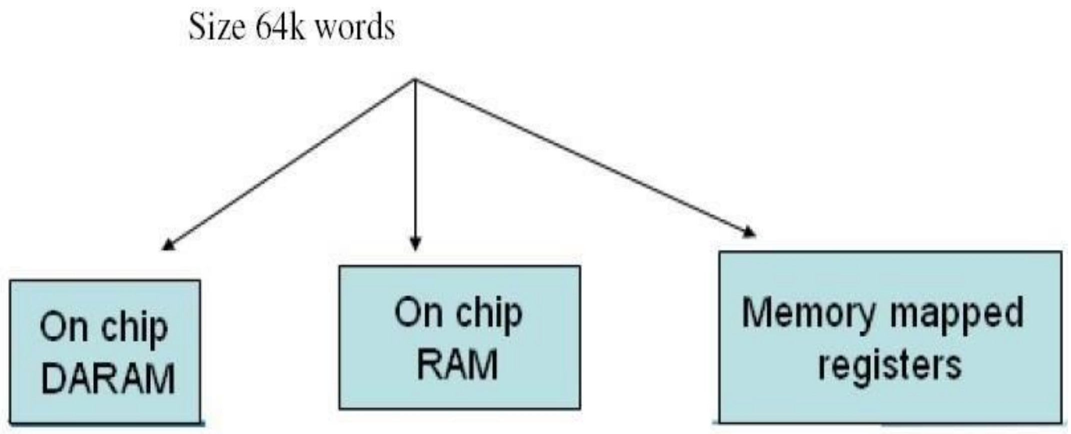


Figure 3.13. Values of stack &SP before and after operation.

Memory Space of TMS320C54xx Processors

- A total of 128k words extendable up to 8192k words.
- Total memory includes RAM, ROM, EPROM, EEPROM or Memory mapped peripherals.
- Data memory: To store data required to run programs & for external memorymapped registers.



Program memory: To store program instructions &tables used in the execution of programs.

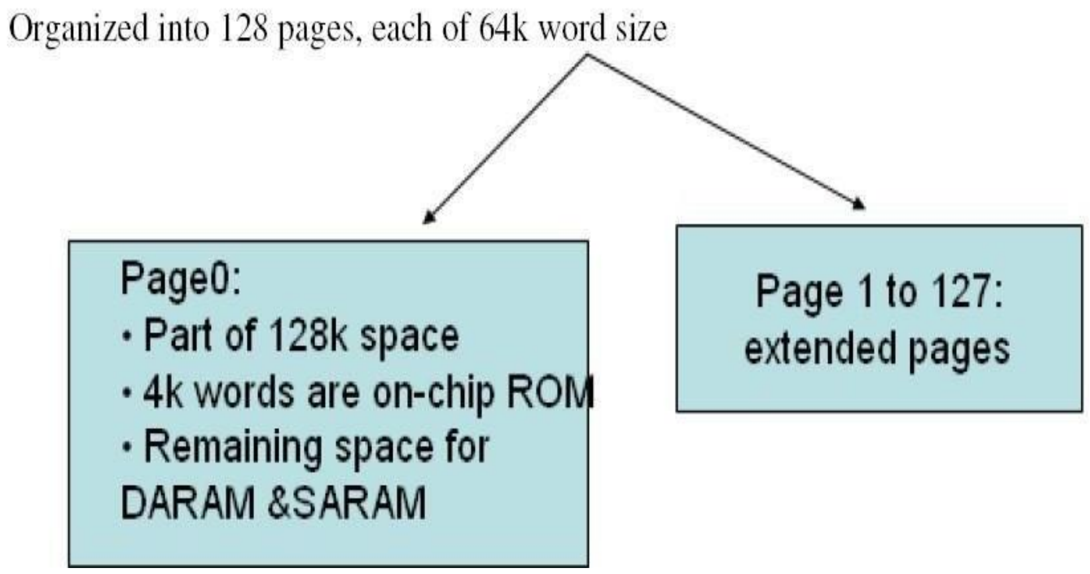
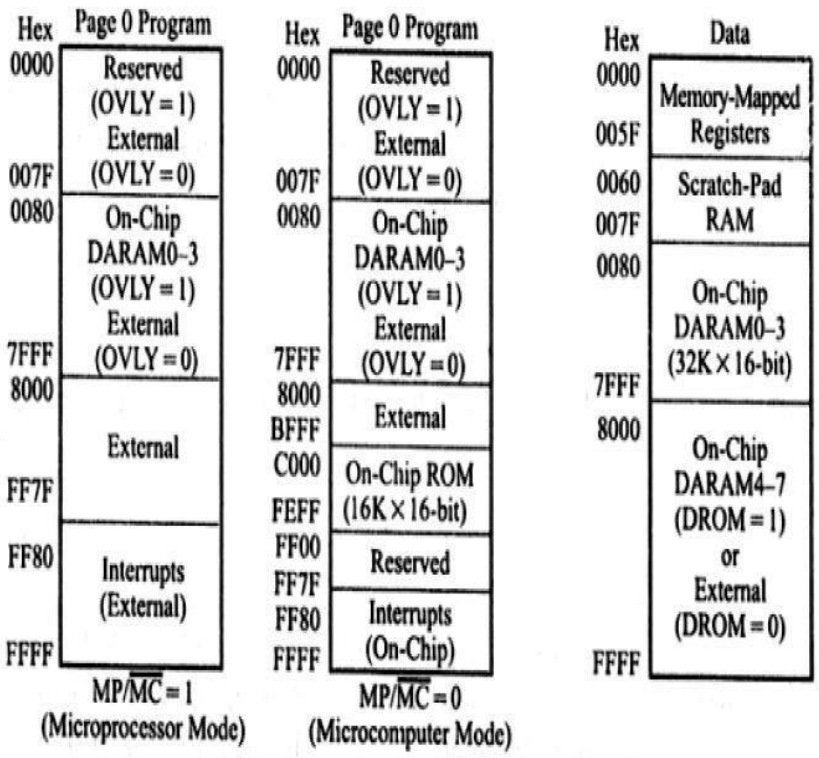

Organized into 128 pages, each of 64k word size
Page0:
• Part of 128k space
• 4k words are on-chip ROM
• Remaining space for DARAM &SARAM
Page 1 to 127: extended pages

Table 3.4.Function of different pin PMST register

PMST bit	Logic	On-chip memory configuration
MP/MC	0 1	ROM enabled ROM not available
OVLY	0 1	RAM in data space RAM in program space
DROM	0 1	ROM not in data space ROM in data space


Hex
Page 0 Program
0000
Reserved (OVLY = 1) External (OVLY = 0)
007F
0080
On-Chip DARAM0–3 (OVLY = 1) External (OVLY = 0)
7FFF
8000
External
FF7F
FF80
Interrupts (External)
FFFF
MP/$\overline{MC}$ = 1 (Microprocessor Mode)
Hex
Page 0 Program
0000
Reserved (OVLY = 1) External (OVLY = 0)
007F
0080
On-Chip DARAM0–3 (OVLY = 1) External (OVLY = 0)
7FFF
8000
External
BFFF
C000
On-Chip ROM (16K × 16-bit)
FEFF
FF00
Reserved
FF7F
FF80
Interrupts (On-Chip)
FFFF
MP/$\overline{MC}$ = 0 (Microcomputer Mode)
Hex
Data
0000
Memory-Mapped Registers
005F
0060
Scratch-Pad RAM
007F
0080
On-Chip DARAM0–3 (32K × 16-bit)
7FFF
8000
On-Chip DARAM4–7 (DROM = 1) or External (DROM = 0)
FFFF

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

Figure 3.14 Memory map for the TMS320C5416 Processor.

Program Control

- It contains program counter (PC), the program counter related H/W, hard stack, repeat counters &status registers.
- PC addresses memory in several ways namely:
- 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.

Problems:

1. Assuming the current content 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)

Solutio n:

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 = 200h
f. AR3 ← AR3 + 40h;
AR3 = 200 + 40h = 240h
g. AR3 ← AR3 - 40h;
AR3 = 200 - 40h = 1C0h

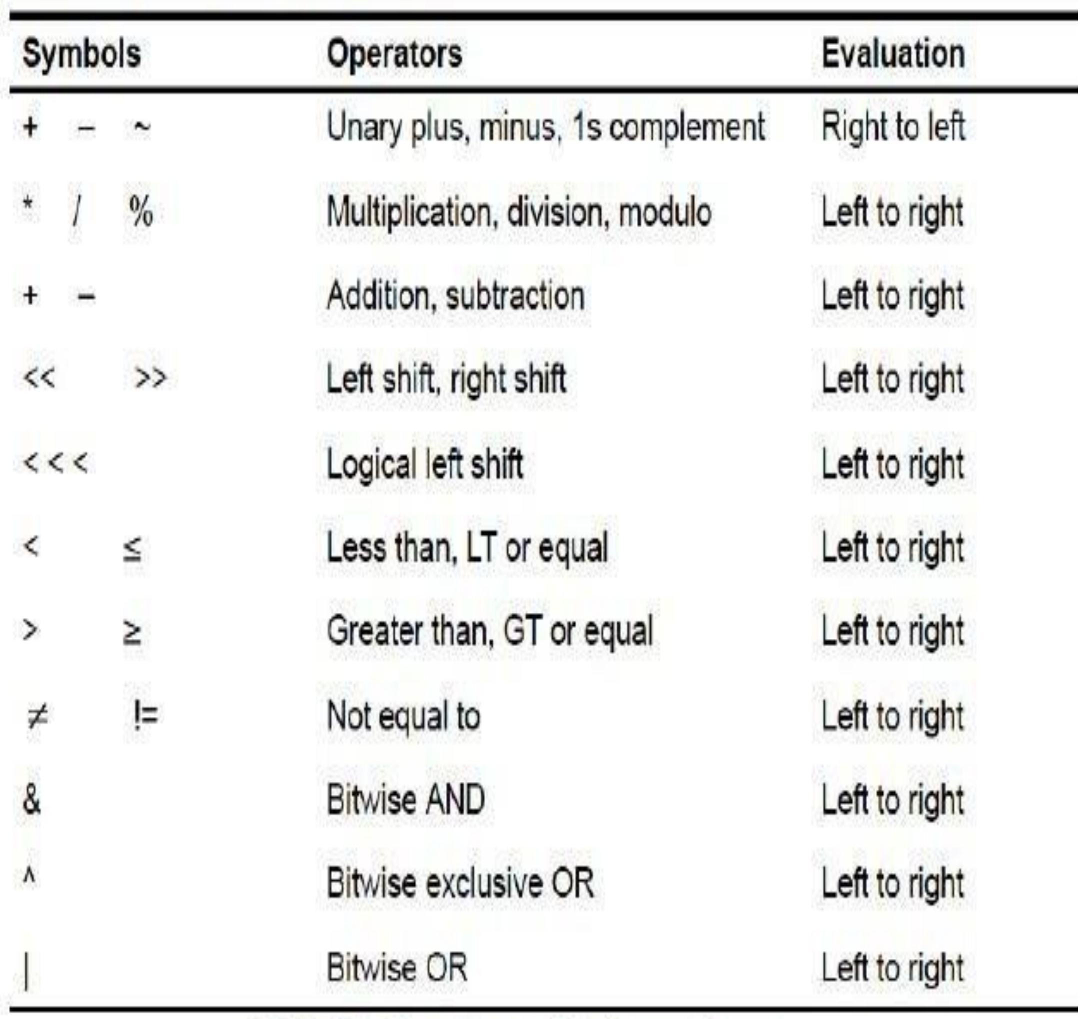
2. Assuming the current contents of AR3 to be 200h, what will be its contents after
each of the following TMS320C54xx addressing modes is used? Assume that the contents of AR0 are 20h
a. *AR3 + 0B
b. *AR3 – 0B

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

Instruction and programming

Operators Used in Instruction Set:



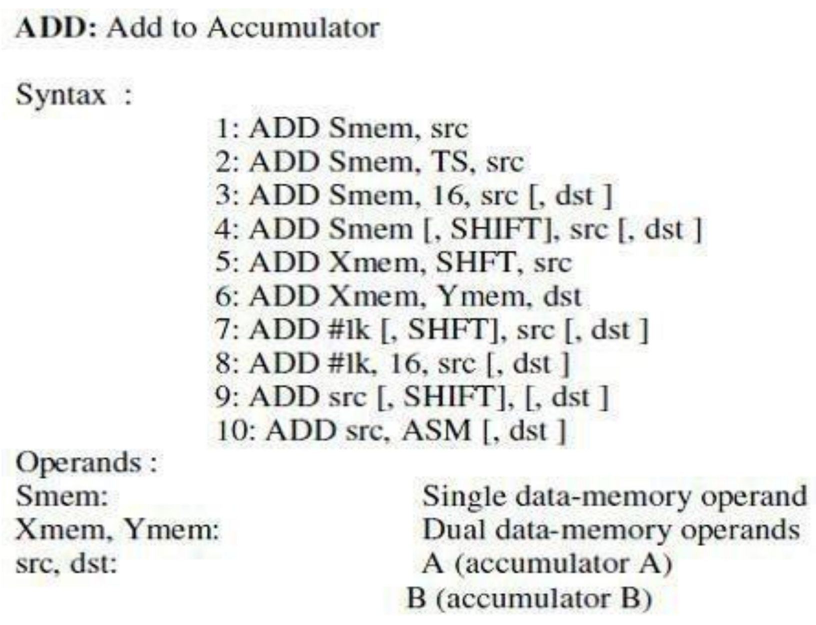
Symbols	Operators	Evaluation
+ − ~	Unary plus, minus, 1s complement	Right to left
* / %	Multiplication, division, modulo	Left to right
+ −	Addition, subtraction	Left to right
<< >>	Left shift, right shift	Left to right
<<<	Logical left shift	Left to right
< ≤	Less than, LT or equal	Left to right
> ≥	Greater than, GT or equal	Left to right
≠ !=	Not equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right

Table 4.1. Operator used in instruction set

4.1.1 Arithmetic Instructions:

Add Instructions:

Syntax	Expression
ADD Smem, src	src src = src + Smem
ADD Smem, TS, src	src = src + Smem << TS
ADD Smem, 16, src [, dst]	dst = src + Smem << 16
ADD Smem [, SHIFT], src [, dst]	dst = src + Smem << SHIFT
ADD Xmem, SHFT, src	src = src + Xmem <<□SHFT
ADD Xmem, Ymem, dst	dst = Xmem << 16 + Ymem << 16
ADD #lk [, SHFT], src [, dst]	dst = src + #lk << SHFT
ADD #lk, 16, src [, dst]	dst = src + #lk << 16
ADD src [, SHIFT] [, dst]	dst = dst + src << SHIFT
ADD src, ASM [, dst]	dst = dst + src << ASM
ADDC Smem, src	src = src + Smem + C
ADDM #lk, Smem	Smem = Smem + #lk



ADD: Add to Accumulator

Syntax :

1: ADD Smem, src
2: ADD Smem, TS, src
3: ADD Smem, 16, src [, dst]
4: ADD Smem [, SHIFT], src [, dst]
5: ADD Xmem, SHFT, src
6: ADD Xmem, Ymem, dst
7: ADD #lk [, SHFT], src [, dst]
8: ADD #lk, 16, src [, dst]
9: ADD src [, SHIFT], [, dst]
10: ADD src, ASM [, dst]

Operands :

Smem:	Single data-memory operand
Xmem, Ymem:	Dual data-memory operands
src, dst:	A (accumulator A)
	B (accumulator B)

$-32\,768 \leq lk \leq 32\,767$
$-16 \leq SHIFT \leq 15$
$0 \leq SHFT \leq 15$

Execution :

1: (Smem) + (src) → src
2: (Smem) << (TS) + (src)→ src
3: (Smem) << 16 + (src) → dst
4: (Smem) [<< SHIFT] + (src) → dst

SUB: Subtract From Accumulator

Syntax	1: **SUB** *Smem, src*
	2: **SUB** *Smem*, **TS**, *src*
	3: **SUB** *Smem*, **16**, *src* [, *dst*]
	4: **SUB** *Smem* [, *SHIFT*], *src* [, *dst*]
	5: **SUB** *Xmem, SHFT, src*
	6: **SUB** *Xmem, Ymem, dst*
	7: **SUB** #*lk* [, *SHFT*], *src* [, *dst*]
	8: **SUB** #*lk*, **16**, *src* [, *dst*]
	9: **SUB** *src* [, *SHIFT*], [, *dst*]
	10: **SUB** *src*, **ASM** [, *dst*]
Operands	src, dst: A (accumulator A)
	B (accumulator B)
	Smem: Single data-memory operand
	Xmem, Ymem: Dual data-memory operan
	$-32\,768 \leq lk \leq 32\,767$
	$0 \leq SHFT \leq 15$
	$-16 \leq SHIFT \leq 15$
Execution	1: (src) – (Smem) → src
	2: (src) – (Smem) << TS → src
	3: (src) – (Smem) << 16 → dst
	4: (src) – (Smem) << SHIFT → dst
	5: (src) – (Xmem) << SHFT → src
	6: (Xmem) << 16 – (Ymem) << 16 → dst
	7: (src) – lk << SHFT → dst
	8: (src) – lk << 16 → dst
	9: (dst) – (src) << SHIFT → dst
	10: (dst) – (src) << ASM → dst
Status Bits	Affected by SXM and OVM
	Affects C and OVdst (or OVsrc, if dst = src)

SUBB: Subtract From Accumulator with Borrow

Syntax	**SUBB** *Smem, src*	
Operands	src:	A (accumulator A) B (accumulator B)
	Smem:	Single data-memory operand
Execution	(src) – (Smem) – (logical inversion of C) $\rightarrow$ src	
Status Bits	Affected by OVM and C Affects C and OVsrc	

SUBC: Subtract Conditionally

Syntax	**SUBC** *Smem, src*	
Operands	Smem:	Single data-memory operand
	src:	A (accumulator A) B (accumulator B)
Execution	(src) – ((Smem) << 15) $\rightarrow$ ALU output If ALU output $\geq$ 0 Then ((ALU output) << 1) + 1 $\rightarrow$ src Else (src) << 1 $\rightarrow$ src	
Status Bits	Affected by SXM Affects C and OVsrc	

SUBS: Subtract with accumulator with sign extension suppressed

Syntax	**SUBS** *Smem, src*	
Operands	Smem:	Single data-memory operand
	src:	A (accumulator A) B (accumulator B)
Execution	(src) – unsigned (Smem) $\rightarrow$ src	
Status Bits	Affected by OVM Affects C and OVsrc	

MPY: Multiply With/Without Rounding

Syntax

1: **MPY[R]** *Smem, dst*
2: **MPY** *Xmem, Ymem, dst*
3: **MPY** *Smem, #lk, dst*
4: **MPY** *#lk, dst*

Operands

Smem:	Single data-memory operand
Xmem, Ymem:	Dual data-memory operands
dst:	A (accumulator A)
	B (accumulator B)

$-32\,768 \le \text{lk} \le 32\,767$

Execution

1: $(T) \times (\text{Smem}) \rightarrow \text{dst}$
2: $(\text{Xmem}) \times (\text{Ymem}) \rightarrow \text{dst}$
$(\text{Xmem}) \rightarrow T$
3: $(\text{Smem}) \times \text{lk} \rightarrow \text{dst}$
$(\text{Smem}) \rightarrow T$
4: $(T) \times \text{lk} \rightarrow \text{dst}$

Status Bits

Affected by FRCT and OVM
Affects OVdst

MPYA: Multiply by Accumulator A

Syntax

1: **MPYA** *Smem*
2: **MPYA** *dst*

Operands

Smem:	Single data-memory operand
dst:	A (accumulator A)
	B (accumulator B)

Execution

1: $(\text{Smem}) \times (A(32\text{–}16)) \rightarrow B$
$(\text{Smem}) \rightarrow T$
2: $(T) \times (A(32\text{–}16)) \rightarrow \text{dst}$

Status Bits

Affected by FRCT and OVM
Affects OVdst (OVB in syntax 1)

MPYU: Multiply Unsigned

SQUR: Square

Syntax	1: **SQUR** *Smem, dst* 2: **SQUR A**, *dst*
Operands	Smem: Single data-memory operand dst: A (accumulator A) B (accumulator B)
Execution	1: (Smem) → T (Smem) × (Smem) → dst 2: (A(32–16)) × (A(32–16)) → dst
Status Bits	Affected by OVM and FRCT Affects OVsrc

SQURA: Square and Accumulate

Syntax	**SQURA** *Smem, src*
Operands	Smem: Single data-memory operand src: A (accumulator A) B (accumulator B)
Execution	(Smem) → T (Smem) × (Smem) + (src) → src
Status Bits	Affected by OVM and FRCT Affects OVsrc

SQURS: Square and Subtract

Syntax	**SQURS** *Smem, src*
Operands	Smem: Single data-memory operand src: A (accumulator A) B (accumulator B)
Execution	(Smem) → T (src) – (Smem) × (Smem) → src
Status Bits	Affected by OVM and FRCT Affects OVsrc

MAC[R]: Multiply Accumulate With/Without Rounding

Syntax	1: **MAC[R]** *Smem, src*
	2: **MAC[R]** *Xmem, Ymem, src* [, *dst*]
	3: **MAC** *#lk, src* [, *dst*]
	4: **MAC** *Smem, #lk, src* [, *dst*]

Operands

Smem:	Single data-memory operands
Xmem, Ymem:	Dual data-memory operands
src, dst:	A (accumulator A)
	B (accumulator B)

$-32\,768 \leq \text{lk} \leq 32\,767$

Execution

1: $(\text{Smem}) \times (T) + (\text{src}) \rightarrow \text{src}$

2: $(\text{Xmem}) \times (\text{Ymem}) + (\text{src}) \rightarrow \text{dst}$
$(\text{Xmem}) \rightarrow T$

3: $(T) \times \text{lk} + (\text{src}) \rightarrow \text{dst}$

4: $(\text{Smem}) \times \text{lk} + (\text{src}) \rightarrow \text{dst}$
$(\text{Smem}) \rightarrow T$

Status Bits

Affected by FRCT and OVM
Affects OVdst (or OVsrc, if dst is not specified)

MACA[R]: Multiply by Accumulator A and Accumulate With/Without Rounding

Syntax

1: **MACA[R]** *Smem* [, *B*]
2: **MACA[R]** **T**, *src* [, *dst*]

Operands

Smem:	Single data-memory operand
src, dst:	A (accumulator A)
	B (accumulator B)

Execution

1: $(\text{Smem}) \times (A(32\text{–}16)) + (B) \rightarrow B$
$(\text{Smem}) \rightarrow T$

2: $(T) \times (A(32\text{–}16)) + (\text{src}) \rightarrow \text{dst}$

Status Bits

Affected by FRCT and OVM
Affects OVdst (or OVsrc, if dst is not specified) and OVB in syntax 1

MACD: Multiply by Program Memory and Accumulate With Delay

Syntax	**MACD** *Smem, pmad, src*
Operands	Smem: Single data-memory operand src: A (accumulator A) B (accumulator B) $0 \leq \text{pmad} \leq 65\,535$

MACP: Multiply by Program Memory and Accumulate

Syntax **MACP** *Smem, pmad, src*

Operands

Smem:	Single data-memory operand
src:	A (accumulator A)
	B (accumulator B)

$0 \leq \text{pmad} \leq 65\,535$

Execution

```
(pmad) → PAR
If (RC) ≠ 0
Then
    (Smem) × (Pmem addressed by PAR) + (src) → src
    (Smem) → T
    (PAR) + 1 → PAR
Else
    (Smem) × (Pmem addressed by PAR) + (src) → src
    (Smem) → T
```

Status Bits Affected by FRCT and OVM
Affects OVsrc

MACSU: Multiply Signed by Unsigned and Accumulate

Syntax **MACSU** *Xmem, Ymem, src*

Operands

Xmem, Ymem:	Dual data-memory operands
src:	A (accumulator A)
	B (accumulator B)

Execution unsigned(Xmem) × signed(Ymem) + (src) → src
(Xmem) → T

Status Bits Affected by FRCT and OVM
Affects OVsrc

MACSU: Multiply Signed by Unsigned and Accumulate

Syntax	**MACSU** *Xmem*, *Ymem*, *src*
Operands	Xmem, Ymem: Dual data-memory operands src: A (accumulator A) B (accumulator B)
Execution	unsigned(Xmem) × signed(Ymem) + (src) → src (Xmem) → T
Status Bits	Affected by FRCT and OVM Affects OVsrc

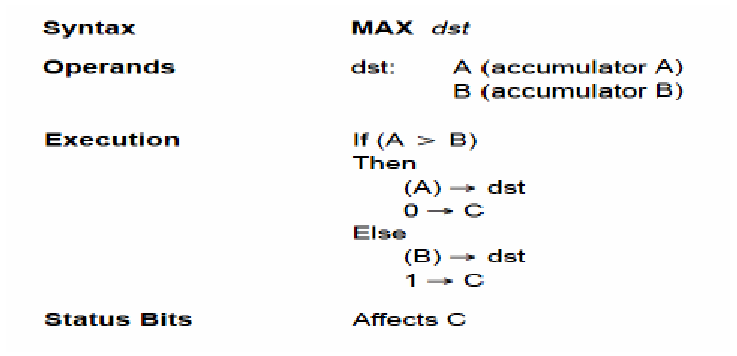
MAS[R] :Multiply and Subtract With/Without Rounding

Syntax	1: **MAS[R]** *Smem*, *src* 2: **MAS[R]** *Xmem*, *Ymem*, *src* [, *dst*]
Operands	Smem: Single data-memory operand Xmem, Ymem: Dual data-memory operands src, dst: A (accumulator A) B (accumulator B)

MASA[R] :Multiply by Accumulator A and Subtract With/Without Rounding

Syntax	1: **MASA** *Smem* [, *B*] 2: **MASA[R]** T, *src* [, *dst*]
Operands	Smem: Single data-memory operand src, dst: A (accumulator A) B (accumulator B)
Execution	1: (B) − (Smem) × (A(32–16)) → B (Smem) → T 2: (src) − (T) × (A(32–16)) → dst
Status Bits	Affected by FRCT and OVM Affects OVdst (or OVsrc, if dst is not specified) and OVB in syntax 1

MAX :Accumulator Maximum



Syntax **MAX** *dst*

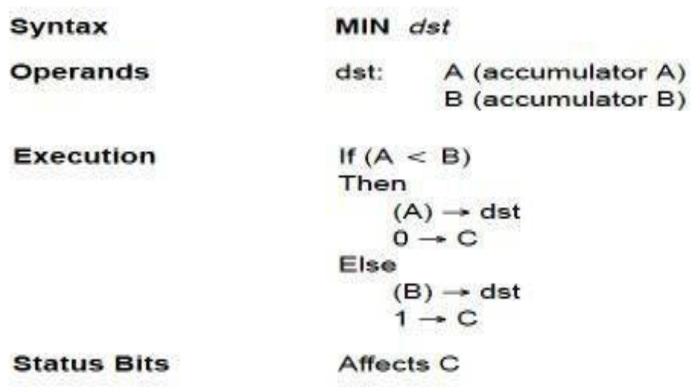
Operands dst: A (accumulator A)
B (accumulator B)

Execution

```
If (A > B)
Then
     (A) → dst
     0 → C
Else
     (B) → dst
     1 → C
```

Status Bits Affects C

MIN : Accumulator Minimum



Syntax **MIN** *dst*

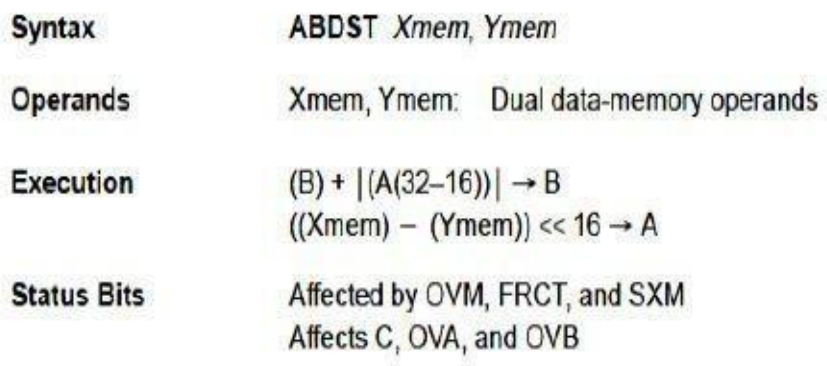
Operands dst: A (accumulator A)
B (accumulator B)

Execution

```
If (A < B)
Then
     (A) → dst
     0 → C
Else
     (B) → dst
     1 → C
```

Status Bits Affects C

ABDST: Absolute Distance



Syntax **ABDST** *Xmem, Ymem*

Operands Xmem, Ymem: Dual data-memory operands

Execution $(B) + |(A(32–16))| \rightarrow B$
$((Xmem) - (Ymem)) << 16 \rightarrow A$

Status Bits Affected by OVM, FRCT, and SXM
Affects C, OVA, and OVB

ABS: Absolute Value of Accumulator

```
ABS A
```

	Before Instruction		After Instruction
A	03 1234 5678	A	00 7FFF FFFF
OVM	1	OVM	1

CMPL :Complement Accumulator

Syntax	**CMPL** *src* **[**, *dst* **]**
Operands	src, dst: A (accumulator A) B (accumulator B)
Execution	$(\overline{src}) \rightarrow dst$
Status Bits	None

CMPM :Compare Memory With Long Immediate

Syntax **CMPM** *Smem, #lk*

Operands Smem: Single data-memory operan
$-32\,768 \leq lk \leq 32\,767$

Execution

```
If (Smem) = lk
Then
    1 → TC
Else
    0 → TC
```

Status Bits Affects TC

CMPS :Compare, Select and Store Maximum

Syntax **CMPS** *src, Smem*

Operands src: A (accumulator A)
B (accumulator B)
Smem: Single data-memory operand

Execution

```
If ((src(31–16)) > (src(15–0)))
Then
    (src(31–16)) → Smem
    (TRN) << 1 → TRN
    0 → TRN(0)
    0 → TC
Else
    (src(15–0)) → Smem
    (TRN) << 1 → TRN
    1 → TRN(0)
    1 → TC
```

Status Bits Affects TC

EXP: Accumulator Exponent

Syntax	**EXP** *src*
Operands	src: A (accumulator A) B (accumulator B)
Execution	If (src) = 0 Then 0 → T Else (Number of leading bits of src) − 8 → T
Status Bits	None

SAT : Saturate Accumulator

Operands	src: A (accumulator A) B (accumulator B)
Execution	Saturate (src) → src
Status Bits	Affects OVsrc

NORM: Normalization

Syntax	**NORM** *src* [, *dst*]
Operands	src, dst : A (accumulator A) B (accumulator B)
Execution	(src) << TS → dst
Status Bits	Affected by SXM and OVM Affects OVdst (or OVsrc, when dst = src)

4.1.2 Logical Operations:

AND: AND With Accumulator

Syntax	1: **AND** *Smem*, *src* 2: **AND** *#lk* [, *SHFT*], *src* [, *dst*] 3: **AND** *#lk*, **16**, *src* [, *dst*] 4: **AND** *src* [, *SHIFT*], [, *dst*]
Operands	Smem: Single data-memory operand src: A (accumulator A) B (accumulator B) $-16 \leq$ SHIFT ≤ 15 $0 \leq$ SHFT ≤ 15 $0 \leq$ lk $\leq 65\,535$
Execution	1: (Smem) AND (src) $\rightarrow$ src 2: lk << SHFT AND (src)$\rightarrow$ dst 3: lk << 16 AND (src)$\rightarrow$ dst 4: (dst) AND (src) << SHIFT $\rightarrow$ dst
Status Bits	None

ANDM:AND Memory With Long Immediate

Syntax	**ANDM** *#lk*, *Smem*
Operands	Smem: Single data-memory operand $0 \leq$ lk $\leq 65\,535$
Execution	lk AND (Smem) $\rightarrow$ Smem
Status Bits	None

OR: OR with Accumulator

Syntax

1: **OR** *Smem, src*
2: **OR** *#lk* [, *SHFT*], *src* [, *dst*]
3: **OR** *#lk*, **16**, *src* [, *dst*]
4: **OR** *src* [, *SHIFT*], [, *dst*]

Operands

src, dst : A (accumulator A)
B (accumulator B)
Smem : Single data-memory operand
$0 \le SHFT \le 15$
$-16 \le SHIFT \le 15$
$0 \le lk \le 65\,535$

Execution

1: (Smem) OR (src(15–0)) → src
src(39–16) unchanged
2: lk << SHFT OR (src) → dst
3: lk << 16 OR (src) → dst
4: (src or [dst]) OR (src) << SHIFT → dst

Status Bits None

ORM: OR Memory With Constant

Syntax **ORM** *#lk, Smem*

Operands Smem: Single data-memory operand
$0 \le lk \le 65\,535$

Execution lk OR (Smem) → Smem

Status Bits None

XOR: Exclusive OR With Accumulator

Syntax

1: **XOR** *Smem, src*
2: **XOR** *#lk* [, *SHFT*], *src* [, *dst*]
3: **XOR** *#lk*, **16**, *src* [, *dst*]
4: **XOR** *src* [, *SHIFT*] [, *dst*]

Operands

src, dst: A (accumulator A)
B (accumulator B)
Smem: Single data-memory operand
$0 \le SHFT \le 15$
$-16 \le SHIFT \le 15$
$0 \le lk \le 65\,535$

Execution	1: (Smem) XOR (src) → src 2: lk << SHFT XOR (src) → dst 3: lk << 16 XOR (src) → dst 4: (src) << SHIFT XOR (dst) → dst
Status Bits	None

XORM: Exclusive OR Memory with Constant

Syntax	**XORM** *#lk, Smem*
Operands	Smem: Single data-memory operand $0 \le lk \le 65\,535$
Execution	lk XOR (Smem) → Smem
Status Bits	None

ROL: Rotate Accumulator Left

Syntax	**ROL** *src*
Operands	src : A (accumulator A) B (accumulator B)
Execution	(C) → src(0) (src(30–0)) → src(31–1) (src(31)) → C 0 → src(39–32)
Status Bits	Affected by C Affects C

ROLTC: Rotate Accumulator Left Using TC

Syntax	**ROLTC** *src*
Operands	src: A (accumulator A) B (accumulator B)
Execution	(TC) → src(0) (src(30–0)) → src(31–1) (src(31)) → C 0 → src(39–32)
Status Bits	Affects C Affected by TC

,

ROR: Rotate Accumulator Right

Syntax	**ROR** *src*
Operands	src: A (accumulator A) B (accumulator B)
Execution	(C) → src(31) (src(31–1)) → src(30–0) (src(0)) → C 0 → src(39–32)
Status Bits	Affects C Affected by C

SFTA: Shift Accumulator Arithmetically

Syntax	**SFTA** *src*, *SHIFT* [, *dst*]
Operands	src, dst A (accumulator A) B (accumulator B) $-16 \le$ SHIFT ≤ 15

Execution

```
If SHIFT < 0
Then
    (src((–SHIFT) – 1)) → C
    (src(39–0)) << SHIFT → dst
    If SXM = 1
    Then
        (src(39)) → dst(39–(39 + (SHIFT + 1))) [or src(39–(39 + (SHIFT + 1))),
        if dst is not specified]
    Else
        0 → dst(39–(39 + (SHIFT + 1))) [or src(39–(39 + (SHIFT + 1))),
        if dst is not specified]
Else
    (src(39 – SHIFT)) → C
    (src) << SHIFT → dst
    0 → dst((SHIFT – 1)–0) [or src((SHIFT – 1)–0), if dst is not specified]
```

Status Bits	Affected by SXM and OVM Affects C and OVdst (or OVsrc, if dst = src)

SFTC: Shift Accumulator Conditionally

Syntax **SFTC** *src*

Operands src: A (accumulator A)
B (accumulator B)

Execution

```
If (src) = 0
Then
    1 → TC
Else
    If (src(31)) XOR (src(30)) = 0
    Then (two significant sign bits)
        0 → TC
        (src) << 1 → src
    Else (only one sign bit)
        1 → TC
```

Status Bits Affects TC

SFTL: Shift Accumulator Logically

Syntax **SFTL** *src*, *SHIFT* [, *dst*]

Operands src, dst: A (accumulator A)
B (accumulator B)
$-16 \le SHIFT \le 15$

Execution

```
If SHIFT < 0
Then
    src((–SHIFT) – 1) → C
    src(31–0) << SHIFT → dst
    0 → dst(39–(31 + (SHIFT + 1)))
If SHIFT = 0
Then
    0 → C
Else
    src(31 – (SHIFT – 1)) → C
    src((31 – SHIFT)–0) << SHIFT → dst
    0 → dst((SHIFT – 1)–0) [or src((SHIFT – 1)–0), if dst is not specified]
    0 → dst(39–32) [or src(39–32), if dst is not specified]
```

Status Bits Affects C

BIT : Test Bit

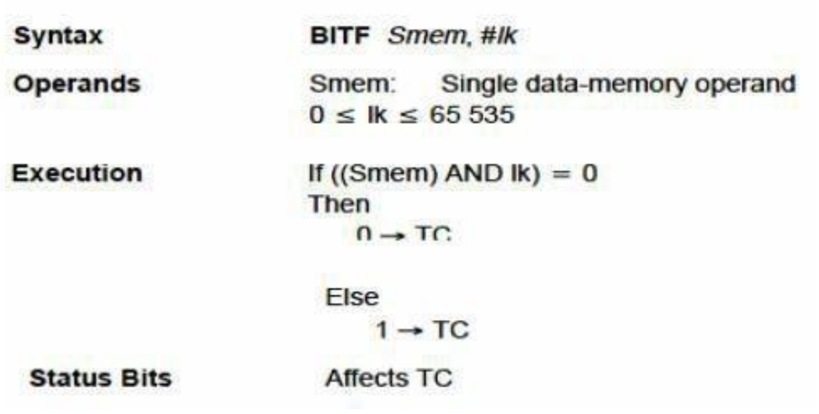
Syntax **BIT** *Xmem*, *BITC*

Operands Xmem: Dual data-memory operand
$0 \le BITC \le 15$

Execution (Xmem(15 – BITC)) → TC

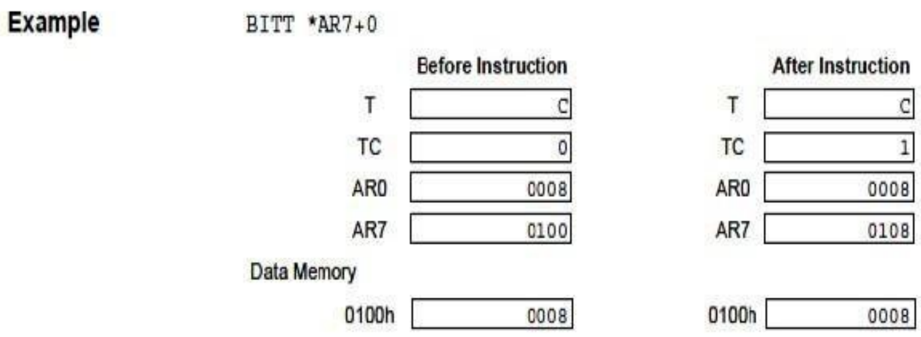
Status Bits Affects TC

BITF: Test Bit Field Specified by Immediate Value



Syntax	**BITF** *Smem, #lk*
Operands	Smem: Single data-memory operand 0 ≤ lk ≤ 65 535
Execution	If ((Smem) AND lk) = 0 Then 0 → TC Else 1 → TC
Status Bits	Affects TC

BITT :Test Bit Specified by T



Example `BITT *AR7+0`

	Before Instruction		After Instruction
T	C	T	C
TC	0	TC	1
AR0	0008	AR0	0008
AR7	0100	AR7	0108
Data Memory			
0100h	0008	0100h	0008

4.1.3.Load and Store operations:

LD: Load Accumulator with Shift

Syntax	1: **LD** *Smem, dst* 2: **LD** *Smem,* **TS**, *dst* 3: **LD** *Smem,* **16**, *dst* 4: **LD** *Smem* [, *SHIFT*], *dst* 5: **LD** *Xmem, SHFT, dst* 6: **LD** *#K, dst* 7: **LD** *#lk* [, *SHFT*], *dst* 8: **LD** *#lk,* **16**, *dst* 9: **LD** *src,* **ASM** [, *dst*] 10: **LD** *src* [, *SHIFT*], *dst* For additional load instructions, see *Load T/DP/ASM/ARP* on page 4-70.

Operands	Smem:	Single data-memory operand
	Xmem:	Dual data-memory operand
	src, dst:	A (accumulator A)
		B (accumulator B)

$0 \le K \le 255$
$-32\,768 \le lk \le 32\,767$
$-16 \le SHIFT \le 15$
$0 \le SHFT \le 15$

Execution

1: (Smem) → dst
2: (Smem) << TS → dst
3: (Smem) << 16 → dst
4: (Smem) << SHIFT → dst
5: (Xmem) << SHFT → dst
6: K → dst
7: lk << SHFT → dst
8: lk << 16 → dst
9: (src) << ASM → dst
10: (src) << SHIFT → dst

Status Bits

Affected by SXM in all accumulator loads
Affected by OVM in loads with SHIFT or ASM shift
Affects OVdst (or OVsrc, when dst = src) in loads with SHIFT or ASM shift

LD :Load T/DP/ASM/ARP

Syntax	1: **LD** *Smem*, **T**
	2: **LD** *Smem*, **DP**
	3: **LD** *#k9*, **DP**
	4: **LD** *#k5*, **ASM**
	5: **LD** *#k3*, **ARP**
	6: **LD** *Smem*, **ASM**

Operands	Smem:	Single data-memory operand
	$0 \leq k9 \leq 511$	
	$-16 \leq k5 \leq 15$	
	$0 \leq k3 \leq 7$	

Execution	1: (Smem) → T
	2: (Smem(8–0)) → DP
	3: k9 → DP
	4: k5 → ASM
	5: k3 → ARP
	6: (Smem(4–0)) → ASM
Status Bits	None

LDM: Load Memory-Mapped Register

Syntax	**LDM** *MMR, dst*	
Operands	MMR:	Memory-mapped register
	dst:	A (accumulator)
		B (accumulator)
Execution	(MMR) → dst(15–0)	
	00 0000h → dst(39–16)	
Status Bits	None	

LD||MAC[R] :Load Accumulator With Parallel Multiply Accumulate With/Without Rounding

Syntax	**LD** *Xmem, dst* || **MAC[R]** *Ymem* [, *dst_*]	
Operands	dst:	A (accumulator A) B (accumulator B)
	dst_:	If *dst* = A, then *dst_* = B; if *dst* = B, then *dst_* = A
	Xmem, Ymem:	Dual data-memory operands

Execution

```
(Xmem) << 16 → dst (31–16)
If (Rounding)
    Round (((Ymem) × (T)) + (dst_)) → dst_
Else
    ((Ymem) × (T)) + (dst_) → dst_
```

Status Bits

Affected by SXM, FRCT, and OVM
Affects OVdst_

LD||MAS[R]: Load Accumulator With Parallel Multiply Subtract With/Without Rounding

Syntax	**LD** *Xmem, dst* || **MAS[R]** *Ymem* [, *dst_*]	
Operands	Xmem, Ymem:	Dual data-memory operands
	dst:	A (accumulator A) B (accumulator B)
	dst_:	If *dst* = A, then *dst_* = B; if *dst* = B, then *dst_* = A

Execution

```
(Xmem) << 16 → dst (31–16)
If (Rounding)
    Round ((dst_) – ((T) × (Ymem))) → dst_
Else
    (dst_) – ((T) × (Ymem)) → dst_
```

Status Bits

Affected by SXM, FRCT, and OVM
Affects OVdst_

LDR: Load Memory Value in Accumulator High With Rounding

Syntax	**LDR** *Smem, dst*
Operands	Smem: Single data-memory operand dst: A (accumulator A) B (accumulator B)
Execution	(Smem) << 16 + 1 << 15 → dst(31–16)
Status Bits	Affected by SXM

LDU : Load Unsigned Memory Value

Syntax	**LDU** *Smem, dst*
Operands	Smem: Single data-memory operand dst: A (accumulator A) B (accumulator B)
Execution	(Smem) → dst(15–0) 00 0000h → dst(39–16)

LMS: Least Mean Square

Syntax	**LMS** *Xmem, Ymem*
Operands	Xmem, Ymem: Dual data-memory operands
Execution	$(A) + (Xmem) << 16 + 2^{15} \rightarrow A$ $(B) + (Xmem) \times (Ymem) \rightarrow B$
Status Bits	Affected by SXM, FRCT, and OVM Affects C, OVA, and OVB

LTD :Load T and Insert Delay

Syntax	**LTD** *Smem*
Operands	Smem: Single data-memory operand
Execution	(Smem) → T (Smem) → Smem + 1
Status Bits	None

ST : Store T, TRN, or Immediate Value Into Memory

Syntax	1: **ST T**, *Smem* 2: **ST TRN**, *Smem* 3: **ST** *#lk*, *Smem*
Operands	Smem: Single data-memory operand –32 768 ≤ lk ≤ 32 767
Execution	1: (T) → Smem 2: (TRN) → Smem 3: lk → Smem
Status Bits	None

STH : Store Accumulator High Into Memory

Syntax	1: **STH** *src*, *Smem* 2: **STH** *src*, **ASM**, *Smem* 3: **STH** *src*, *SHFT*, *Xmem* 4: **STH** *src* [, *SHIFT*], *Smem*
Operands	src: A (accumulator A) B (accumulator B) Smem: Single data-memory operand Xmem: Dual data-memory operand $0 \leq \text{SHFT} \leq 15$ $-16 \leq \text{SHIFT} \leq 15$
Execution	1: $(src) << (-16) \rightarrow Smem$ 2: $(src) << (ASM - 16) \rightarrow Smem$ 3: $(src) << (SHFT - 16) \rightarrow Xmem$ 4: $(src) << (SHIFT - 16) \rightarrow Smem$
Status Bits	Affected by SXM

STL: Store Accumulator Low Into Memory

Syntax	1: **STL** *src*, *Smem* 2: **STL** *src*, **ASM**, *Smem* 3: **STL** *src*, *SHFT*, *Xmem* 4: **STL** *src* [, *SHIFT*], *Smem*
Operands	src: A (accumulator A) B (accumulator B) Smem: Single data-memory operand Xmem: Dual data-memory operand $0 \leq \text{SHFT} \leq 15$ $-16 \leq \text{SHIFT} \leq 15$
Execution	1: $(src) \rightarrow Smem$ 2: $(src) << ASM \rightarrow Smem$ 3: $(src) << SHFT \rightarrow Xmem$ 4: $(src) << SHIFT \rightarrow Smem$
Status Bits	Affected by SXM

ST||ADD : Store Accumulator With Parallel Add

Syntax	**ST** *src, Ymem*	
	|| **ADD** *Xmem, dst*	
Operands	src, dst:	A (accumulator A)
		B (accumulator B)
	Xmem, Ymem:	Dual data-memory operands
	dst_:	If *dst* = A, then *dst_* = B; if *dst* = B, then *dst_* = A
Execution	(src) << (ASM − 16) → Ymem	
	(dst_) + (Xmem) << 16 → dst	
Status Bits	Affected by OVM, SXM, and ASM	
	Affects C and OVdst	

ST||LD: Store Accumulator with Parallel Load

Syntax	1:	**ST** *src, Ymem*
		|| **LD** *Xmem, dst*
	2:	**ST** *src, Ymem*
		|| **LD** *Xmem*, **T**
Operands	src, dst:	A (accumulator A)
		B (accumulator B)
	Xmem, Ymem:	Dual data-memory operands
Execution	1.	(src) << (ASM − 16) → Ymem
		(Xmem) << 16 → dst
	2.	(src) << (ASM − 16) → Ymem
		(Xmem) → T
Status Bits	Affected by OVM and ASM	
	Affects C	

ST||MAC[R]: Store Accumulator With Parallel Multiply Accumulate With/Without Rounding

Syntax	**ST** *src, Ymem* **|| MAC[R]** *Xmem, dst*
Operands	src, dst: A (accumulator A) B (accumulator B) Xmem, Ymem: Dual data-memory operands
Execution	See below
Status Bits	Affected by OVM, SXM, ASM, and FRCT Affects C and OVdst

```
(src << (ASM – 16)) → Ymem
If (Rounding)
    Then
    Round ((Xmem) × (T) + (dst)) → dst
Else
    (Xmem) × (T) + (dst) → dst
```

ST||MAS[R]: Store Accumulator With Parallel Multiply Subtract With/Without Rounding

Syntax	**ST** *src, Ymem* **|| MAS[R]** *Xmem, dst*
Operands	src, dst: A (accumulator A) B (accumulator B) Xmem, Ymem: Dual data-memory operands
Execution	See below
Status Bits	Affected by OVM, SXM, ASM, and FRCT Affects C and OVdst

```
(src << (ASM – 16)) → Ymem
If (Rounding)
    Then
    Round ((dst) – (Xmem) × (T))→ dst
Else
    (dst) – (Xmem) × (T) → dst
```

ST||MPY: Store Accumulator With Parallel Multiply

Syntax	**ST** *src, Ymem* || **MPY** *Xmem, dst*	
Operands	src, dst:	A (accumulator A) B (accumulator B)
	Xmem, Ymem:	Dual data-memory operands
Execution	(src << (ASM − 16)) → Ymem (T) × (Xmem) → dst	
Status Bits	Affected by OVM, SXM, ASM, and FRCT Affects C and OVdst	

ST||SUB: Store Accumulator With Parallel Subtract

Syntax	**ST** *src, Ymem* || **SUB** *Xmem, dst*	
Operands	src, dst:	A (accumulator A) B (accumulator B)
	Xmem, Ymem:	Dual data-memory operands
	dst_:	If *dst* = A, then *dst_* = B; if *dst* = B, then *dst_* = A.
Execution	(src << (ASM − 16)) → Ymem (Xmem) << 16 – (dst_) → dst	
Status Bits	Affected by OVM, SXM, and ASM Affects C and OVdst	

STRCD: Store T Conditionally

Syntax **STRCD** *Xmem, cond*

Operands Xmem: Dual data-memory operand

The following table lists the conditions (*cond* operand) for this instruction.

Cond	Description	Condition Code	Cond	Description	Condition Code
AEQ	(A) = 0	0101	BEQ	(B) = 0	1101
ANEQ	(A) ≠ 0	0100	BNEQ	(B) ≠ 0	1100
AGT	(A) > 0	0110	BGT	(B) > 0	1110
AGEQ	(A) ≥ 0	0010	BGEQ	(B) ≥ 0	1010
ALT	(A) < 0	0011	BLT	(B) < 0	1011
ALEQ	(A) ≤ 0	0111	BLEQ	(B) ≤ 0	1111

Execution

```
If (cond)
    (T) → Xmem
Else
    (Xmem) → Xmem
```

Status Bits None

4.1.4. Miscellaneous Load-Type and Store-Type Instructions

MVDD: Move Data From Data Memory to Data Memory With X, Y addressing

Syntax **MVDD** *Xmem, Ymem*

Operands Xmem, Ymem: Dual data-memory operands

Execution (Xmem) → Ymem

Status Bits None

MVDK: Move Data From Data Memory to Data Memory With Destination Addressing

Syntax **MVDK** *Smem, dmad*

Operands Smem: Single data-memory operand
$0 \le \text{dmad} \le 65\,535$

Execution

```
(dmad) → EAR
If (RC) ≠ 0
Then
    (Smem) → Dmem addressed by EAR
    (EAR) + 1 → EAR
Else
    (Smem) → Dmem addressed by EAR
```

Status Bits None

MVDM: Move Data From Data Memory to Memory-Mapped Register

Syntax	**MVDM** *dmad, MMR*
Operands	MMR: Memory-mapped register $0 \leq$ dmad $\leq$ 65 535
Execution	see below
Status Bits	None

```
dmad → DAR
If (RC) ≠ 0
Then
    (Dmem addressed by DAR) → MMR
    (DAR) + 1 → DAR
Else
    (Dmem addressed by DAR) → MMR
```

MVDP: Move Data from Data Memory to Program Memory

Syntax	**MVDP** *Smem, pmad*
Operands	Smem: Single data-memory operand $0 \leq$ pmad $\leq$ 65 535
Execution	see below
Status Bits	None

```
pmad → PAR
If (RC) ≠ 0
Then
    (Smem) → Pmem addressed by PAR
    (PAR) + 1 → PAR
Else
    (Smem) → Pmem addressed by PAR
```

MVKD: Move Data From Data Memory to Data Memory With Source Addressing

Syntax **MVKD** *dmad, Smem*

Operands Smem: Single data-memory operand
$0 \leq \text{dmad} \leq 65\,535$

Execution

```
dmad → DAR
If (RC) ≠ 0
Then
    (Dmem addressed by DAR) → Smem
    (DAR) + 1 → DAR
Else
    (Dmem addressed by DAR) → Smem
```

Status Bits None

Example 1 `MVKD 300h, 0`

	Before Instruction		After Instruction
DP	004	DP	004
Data Memory			
0200h	ABCD	0200h	1234
0300h	1234	0300h	1234

MVMD: Move Data From Memory-Mapped Register to Data Memory

Syntax **MVMD** *MMR, dmad*

Operands MMR: Memory-mapped register
$0 \leq \text{dmad} \leq 65\,535$

Execution

```
dmad → EAR
If (RC) ≠ 0
Then
    (MMR) → Dmem addressed by EAR
    (EAR) + 1 → EAR
Else
    (MMR) → Dmem addressed by EAR
```

Status Bits None

MVMM: Move Data From Memory-Mapped Register to Memory-Mapped Register

Syntax	**MVMM** *MMRx, MMRy*
Operands	MMRx: AR0–AR7, SP MMRy: AR0–AR7, SP
Execution	(MMRx) → MMRy
Status Bits	None

Example `MVMM SP, AR1`

	Before Instruction		After Instruction
AR1	3EFF	AR1	0200
SP	0200	SP	0200

MVPD: Move Data From Program Memory to Data Memory

Syntax **MVPD** *pmad, Smem*

Operands Smem: Single data-memory operand
$0 \leq \text{pmad} \leq 65\,535$

Execution

```
pmad → PAR
If (RC) ≠ 0
Then
    (Pmem addressed by PAR) → Smem
    (PAR) + 1 → PAR
Else
    (Pmem addressed by PAR) → Smem
```

Status Bits None

PORTR: Read Data from Port

PORTW: Write Data to Port

Syntax	**PORTW** *Smem, PA*
Operands	Smem: Single data-memory operand $0 \leq \text{PA} \leq 65\,535$
Execution	(Smem) → PA
Status Bits	None

READA: Read Program Memory addressed by Accumulator A and Store in Data Memory

Syntax **READA** *Smem*

Operands Smem: Single data-memory operand

Execution

```
A → PAR
If ((RC) ≠ 0)
     (Pmem (addressed by PAR)) → Smem
     (PAR) + 1 → PAR
     (RC) – 1 → RC
Else
     (Pmem (addressed by PAR)) → Smem
```

Status Bits None

WRITA: Write Data to Program Memory Addressed by Accumulator A

Syntax **WRITA** *Smem*

Operands Smem: Single data-memory operand

Execution

```
A → PAR
If (RC) ≠ 0
Then
     (Smem) → (Pmem addressed by PAR)
     (PAR) + 1 → PAR
     (RC) – 1 → RC
Else
     (Smem) → (Pmem addressed by PAR)
```

Status Bits None

Branch Instructions

B[D]: Branch Unconditionally

Syntax **B[D]** *pmad*

Operands $0 \leq \text{pmad} \leq 65\,535$

Execution pmad → PC

Status Bits None

BACC[D]: Branch to Location Specified by Accumulator

Syntax	**BACC[D]** *src*
Operands	src: A (accumulator A) B (accumulator B)
Execution	(src(15–0)) → PC
Status Bits	None

BANZ[D]: Branch on Auxiliary Register Not Zero

Syntax	**BANZ[D]** *pmad*, *Sind*
Operands	Sind: Single indirect addressing operand 0 ≤ pmad ≤ 65 535
Execution	If ((ARx) ≠ 0) Then pmad → PC Else (PC) + 2 → PC
Status Bits	None

BC [D]: Branch Conditionally

Syntax	**BC[D]** *pmad*, *cond* [, *cond* [, *cond*]]
Execution	If (cond(s)) Then pmad → PC Else (PC) + 2 → PC
Status Bits	Affects OVA or OVB if OV or NOV is chosen

FB [D]: Far Branch Unconditionally

Syntax	**FB[D]** *extpmad*
Operands	0 ≤ extpmad ≤ 7F FFFF
Execution	(pmad(15–0)) → PC (pmad(22–16)) → XPC
Status Bits	None

FBACC [D]: Far Branch to Location Specified by Accumulator

Syntax	**FBACC[D]** *src*
Operands	src: A (accumulator A) B (accumulator B)
Execution	(src(15–0)) → PC (src(22–16)) → XPC
Status Bits	None

CALA [D]: Call Subroutine at Location Specified by Accumulator

Syntax	**CALA[D]** *src*
Operands	src: A (accumulator A) B (accumulator B)
Execution	**Nondelayed** (SP) – 1 → SP (PC) + 1 → TOS (src(15–0)) → PC **Delayed** (SP) – 1 → SP (PC) + 3 → TOS (src(15–0)) → PC
Status Bits	None

CALL[D]: Call Unconditionally

Syntax	**CALL[D]** *pmad*
Operands	$0 \le$ pmad $\le$ 65 535
Execution	**Nondelayed** (SP) – 1 → SP (PC) + 2 → TOS pmad → PC **Delayed** (SP) – 1 → SP (PC) + 4 → TOS pmad → PC
Status Bits	None

CC [D]: Call Conditionally

Syntax **CC[D]** *pmad*, *cond* [, *cond* [, *cond*]]

Operands 0 ≤ pmad ≤ 65 535

The following table lists the conditions (*cond* operand) for this instruction.

Cond	Description	Condition Code	Cond	Description	Condition Code
BIO	$\overline{\text{BIO}}$ low	0000 0011	NBIO	$\overline{\text{BIO}}$ high	0000 0010
C	C = 1	0000 1100	NC	C = 0	0000 1000
TC	TC = 1	0011 0000	NTC	TC = 0	0010 0000
AEQ	(A) = 0	0100 0101	BEQ	(B) = 0	0100 1101
ANEQ	(A) ≠ 0	0100 0100	BNEQ	(B) ≠ 0	0100 1100
AGT	(A) > 0	0100 0110	BGT	(B) > 0	0100 1110
AGEQ	(A) ≥ 0	0100 0010	BGEQ	(B) ≥ 0	0100 1010
ALT	(A) < 0	0100 0011	BLT	(B) < 0	0100 1011
ALEQ	(A) ≤ 0	0100 0111	BLEQ	(B) ≤ 0	0100 1111
AOV	A overflow	0111 0000	BOV	B overflow	0111 1000
ANOV	A no overflow	0110 0000	BNOV	B no overflow	0110 1000
UNC	Unconditional	0000 0000			

Execution

Nondelayed

```
If (cond(s))
Then
    (SP) − 1 → SP
    (PC) + 2 → TOS
    pmad → PC
Else
    (PC) + 2 → PC
```

Delayed

```
If (cond(s))
Then
    (SP) − 1 → SP
    (PC) + 4 → TOS
    pmad → PC
Else
    (PC) + 2 → PC
```

Status Bits Affects OVA or OVB (if OV or NOV is chosen)

FCALA [D]: Far Call Subroutine at Location Specified by Accumulator

Syntax **FCALA[D]** *src*

Operands src: A (accumulator A)
B (accumulator B)

Execution

Nondelayed

```
(SP) – 1 → SP
(PC) + 1 → TOS
(SP) – 1 → SP
(XPC) → TOS
(src(15–0)) → PC
(src(22–16)) → XPC
```

Delayed

```
(SP) – 1 → SP
(PC) + 3 → TOS
(SP) – 1 → SP
(XPC) → TOS
(src(15–0)) → PC
(src(22–16)) → XPC
```

Status Bits None

FCALL[D]: Far Call Unconditionally

Syntax	**FCALL[D]** *extpmad*
Operands	$0 \le$ extpmad $\le$ 7F FFFF
Execution	**Nondelayed** (SP) − 1 → SP (PC) + 2 → TOS (SP) − 1 → SP (XPC) → TOS (pmad(15–0)) → PC (pmad(22–16)) → XPC
	Delayed (SP) − 1 → SP (PC) + 4 → TOS (SP) − 1 → SP (XPC) → TOS (pmad(15–0)) → PC (pmad(22–16)) → XPC
Status Bits	None

Interrupt Instructions:

INTR: Software Interrupt

Syntax	**INTR** *K*
Operands	$0 \le K \le 31$
Execution	(SP) − 1 → SP (PC) + 1 → TOS interrupt vector specified by K → PC 1 → INTM
Status Bits	Affects INTM and IFR

TRAP: Software Interrupt

Syntax	**TRAP** *K*
Operands	0 ≤ K ≤ 31
Execution	(SP) − 1 → SP (PC) + 1 → TOS Interrupt vector specified by K → PC
Status Bits	None

Return Instructions

FRET [D]: Far Return

Syntax	**FRET[D]**
Operands	None
Execution	(TOS) → XPC (SP) + 1 → SP (TOS) → PC (SP) + 1 → SP
Status Bits	None

FRETE [D]: Enable Interrupts and Far Return From Interrupt

Syntax	**FRETE[D]**
Operands	None
Execution	(TOS) → XPC (SP) + 1 → SP (TOS) → PC (SP) + 1 → SP 0 → INTM
Status Bits	Affects INTM

RC [D]: Return Conditionally

Syntax **RC[D]** *cond* [, *cond* [, *cond*]]

Operands The following table lists the conditions (*cond* operand) for this instruction.

Cond	Description	Condition Code	Cond	Description	Condition Code
BIO	$\overline{\text{BIO}}$ low	0000 0011	NBIO	$\overline{\text{BIO}}$ high	0000 0010
C	C = 1	0000 1100	NC	C = 0	0000 1000
TC	TC = 1	0011 0000	NTC	TC = 0	0010 0000
AEQ	(A) = 0	0100 0101	BEQ	(B) = 0	0100 1101
ANEQ	(A) ≠ 0	0100 0100	BNEQ	(B) ≠ 0	0100 1100
AGT	(A) > 0	0100 0110	BGT	(B) > 0	0100 1110
AGEQ	(A) ≥ 0	0100 0010	BGEQ	(B) ≥ 0	0100 1010
ALT	(A) < 0	0100 0011	BLT	(B) < 0	0100 1011
ALEQ	(A) ≤ 0	0100 0111	BLEQ	(B) ≤ 0	0100 1111
AOV	A overflow	0111 0000	BOV	B overflow	0111 1000
ANOV	A no overflow	0110 0000	BNOV	B no overflow	0110 1000
UNC	Unconditional	0000 0000			

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	Z	0	C	C	C	C	C	C	C	C

Execution

```
If (cond(s))
Then
   (TOS) → PC
   (SP) + 1 → SP
Else
   (PC) + 1 → PC
```

Status Bits None

RET [D]: Return

Syntax	**RET[D]**
Operands	None
Execution	(TOS) → PC (SP) + 1 → SP
Status Bits	None

RETF [D]: Enable Interrupts and Fast Return From Interrupt

Syntax	**RETF[D]**
Operands	None
Execution	(RTN) → PC (SP) + 1 → SP 0 → INTM
Status Bits	Affects INTM

Repeat Instructions

RPT: Repeat Next Instruction

Syntax	1: **RPT** *Smem* 2: **RPT** #*K* 3: **RPT** #*lk*
Operands	Smem: Single data-memory operand $0 \leq K \leq 255$ $0 \leq lk \leq 65\,535$
Execution	1: (Smem) → RC 2: K → RC 3: lk → RC
Status Bits	None

RPTB [D]: Block Repeat

Syntax	**RPTB[D]** *pmad*
Operands	0 ≤ pmad ≤ 65 535
Execution	1 → BRAF If (delayed) then (PC) + 4 → RSA Else (PC) + 2 → RSA pmad → REA
Status Bits	Affects BRAF

RPTZ: Repeat Next Instruction and Clear Accumulator

Syntax	**RPTZ** *dst, #lk*
Operands	dst: A (accumulator A) B (accumulator B) 0 ≤ lk ≤ 65 535
Execution	0 → dst lk → RC
Status Bits	None

Stack-Manipulating Instructions FRAME:

Stack Pointer Immediate Offset

Operands	−128 ≤ K ≤ 127
Execution	(SP) + K → SP
Status Bits	None
Example	`FRAME 10h`

	Before Instruction		After Instruction
SP	1000	SP	1010

POPD: Pop Top of Stack to Data Memory

Syntax	**POPD** *Smem*
Operands	Smem: Single data-memory operand
Execution	(TOS) → Smem (SP) + 1 → SP
Status Bits	None

POPM: Pop Top of Stack to Memory-Mapped Register

Syntax	**POPM** *MMR*
Operands	MMR: Memory-mapped register
Execution	$(TOS) \rightarrow MMR$ $(SP) + 1 \rightarrow SP$
Status Bits	None

PSHD: Push Data-Memory Value onto Stack

Syntax	**PSHD** *Smem*
Operands	Smem: Single data-memory operand
Execution	$(SP) - 1 \rightarrow SP$ $(Smem) \rightarrow TOS$
Status Bits	None

PSHM: Push Memory-Mapped Register onto Stack

Syntax	**PSHM** *MMR*
Operands	MMR: Memory-mapped register
Execution	$(SP) - 1 \rightarrow SP$ $(MMR) \rightarrow TOS$
Status Bits	None

Miscellaneous Program-Control Instructions

SSBX: Set Status Register Bit

Syntax	**SSBX** *N*, *SBIT*
Operands	$0 \leq SBIT \leq 15$ N = 0 or 1
Execution	$1 \rightarrow STN(SBIT)$
Status Bits	None

RSBX: Reset Status Register Bit

Syntax	**RSBX** *N, SBIT*
Operands	0 ≤ SBIT ≤ 15 N = 0 or 1
Execution	$0 \rightarrow$ STN(SBIT)
Status Bits	None

Example 1 `RSBX SXM ; SXM means: n=1 and SBIT=8`

	Before Instruction		After Instruction
ST1	35CD	ST1	34CD

NOP: No Operation

Syntax	**NOP**
Operands	None
Execution	None
Status Bits	None

RESET: Software Reset

Syntax	**RESET**
Operands	None
Execution	These fields of PMST, ST0, and ST1 are loaded with the values shown:

(IPTR) << 7 → PC	0 → OVA	0 → OVB
1 → C	1 → TC	0 → ARP
0 → DP	1 → SXM	0 → ASM
0 → BRAF	0 → HM	1 → XF
0 → C16	0 → FRCT	0 → CMPT
0 → CPL	1 → INTM	0 → IFR
0 → OVM		

Status Bits	The status bits affected are listed in the execution section.

On chip peripherals:

It facilitates interfacing with external devices. The peripherals are:

- General purpose I/O pins
- A software programmable wait state generator.
- Hardware timer
- Host port interface (HPI)
- Clock generator
- Serial port

It has two general purpose I/O pins:

- ➢ BIO-input pin used to monitor the status of external devices.
- ➢ XF- output pin, software controlled used to signal external devices

Software programmable wait state generator:

- ➢ Extends external bus cycles up to seven machine cycles.

Hardware Timer

- ➢ An on chip down counter
- ➢ Used to generate signal to initiate any interrupt or any other process

□ Consists of 3 memory mapped registers:

- ➢ The timer register (TIM)
- ➢ Timer period register (PRD)
- ➢ Timer controls register (TCR)
  - Pre scaler block (PSC).
  - TDDR (Time Divide Down ratio)
  - TIN &TOUT

The timer register (TIM) is a 16-bit memory-mapped register that decrements at every pulse from the prescaler block (PSC).

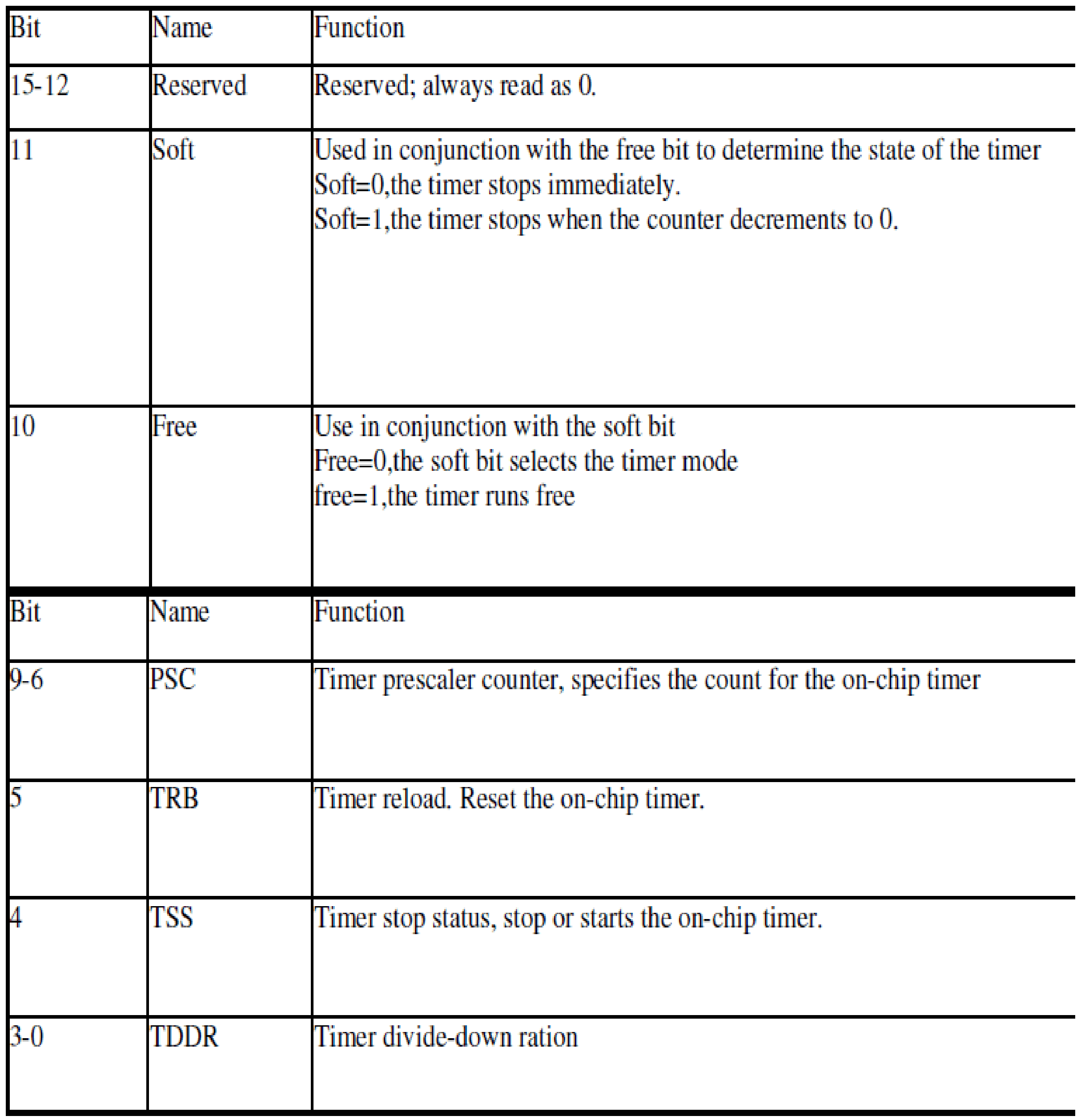
The timer period register (PRD) is a 16-bit memory-mapped register whose contents are loaded onto the TIM whenever the TIM decrements to zero or the device is reset (SRESET).

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 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 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 signal 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.



Bit	Name	Function
15-12	Reserved	Reserved; always read as 0.
11	Soft	Used in conjunction with the free bit to determine the state of the timer Soft=0,the timer stops immediately. Soft=1,the timer stops when the counter decrements to 0.
10	Free	Use in conjunction with the soft bit Free=0,the soft bit selects the timer mode free=1,the timer runs free
9-6	PSC	Timer prescaler counter, specifies the count for the on-chip timer
5	TRB	Timer reload. Reset the on-chip timer.
4	TSS	Timer stop status, stop or starts the on-chip timer.
3-0	TDDR	Timer divide-down ration

Table 4.6. Pin details of software wait state generator

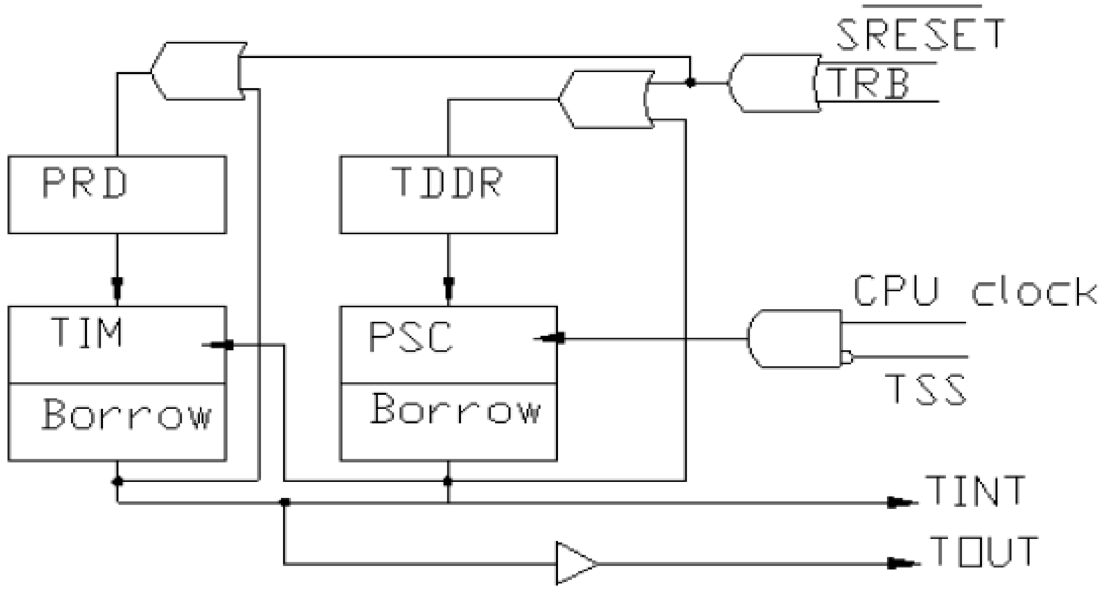
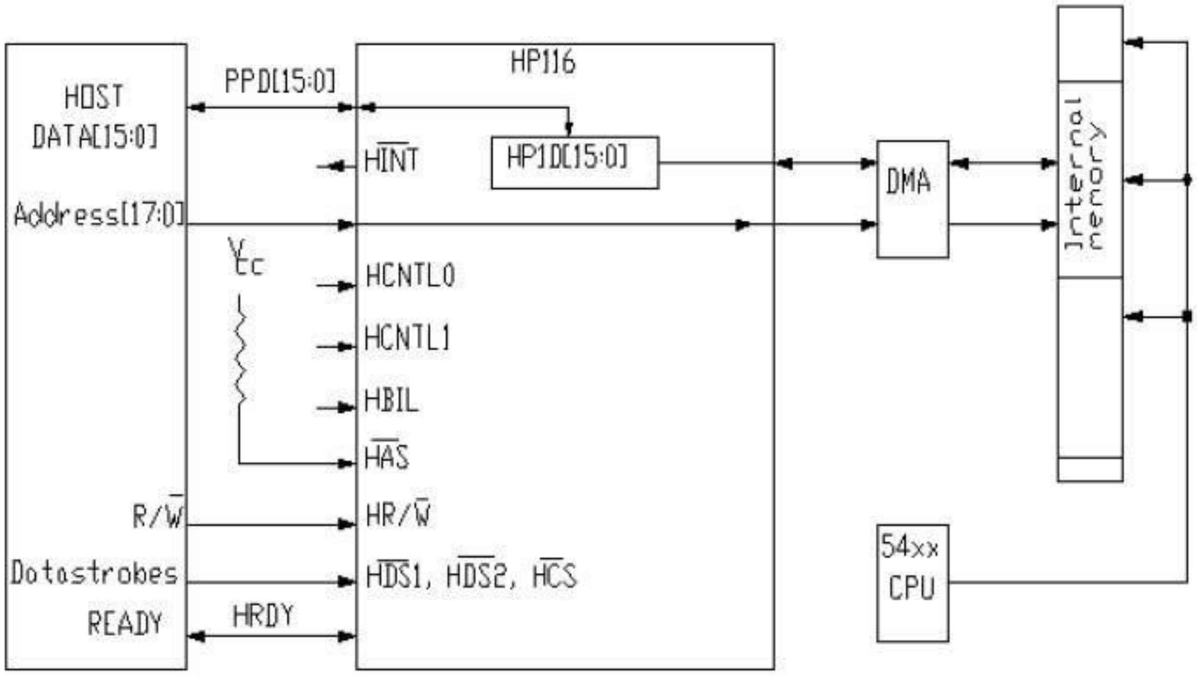


Figure 4.2.Logical block diagram of timer circuit.

Host port interface (HPI):

- Allows to interface to an 8bit or 16bit host devices or a host processor
- Signals in HPI are:
- Host interrupt (HINT)
- HRDY
- HCNTL0 &HCNTL1
- HBIL
- HR/w



4.3. A generic diagram of the host port interface (HPI)

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 it attention is required.
- HRDY, a DSP output indicating that the DSP is ready for transfer.
- HCNTL0 and HCNTL1, control signal 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 second byte.
- HR/W indicates if the host is carrying out a read operation or a write operation

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.

Serial I/O Ports:

Three types of serial ports are available:

- Synchronous ports.
- Buffered ports.
- Time-division multiplexed ports.

The synchronous serial ports are high-speed, full-duplex ports and that provide direct communications with serial devices, such as codec, and analog-to-digital (A/D) converters. A buffered serial port (BSP) is synchronous serial port that is provided with
an auto buffering unit and is clocked at the full clock rate. The head 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. The functioning of each of these on-chip peripherals is controlled by memory-mapped registers assigned to the respective peripheral.

Interrupts of TMS320C54xx Processors:

Many times, when 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 devices. 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 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.

Pipeline operation of TMS320C54xx Processors:

The CPU of ‘54xx devices have 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 from 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.
4 The access phase outputs the read operand’s on the data address bus, DAB. If a second operand is required, the other data address bus, CAB, 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 processes. 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 executed in this phase.

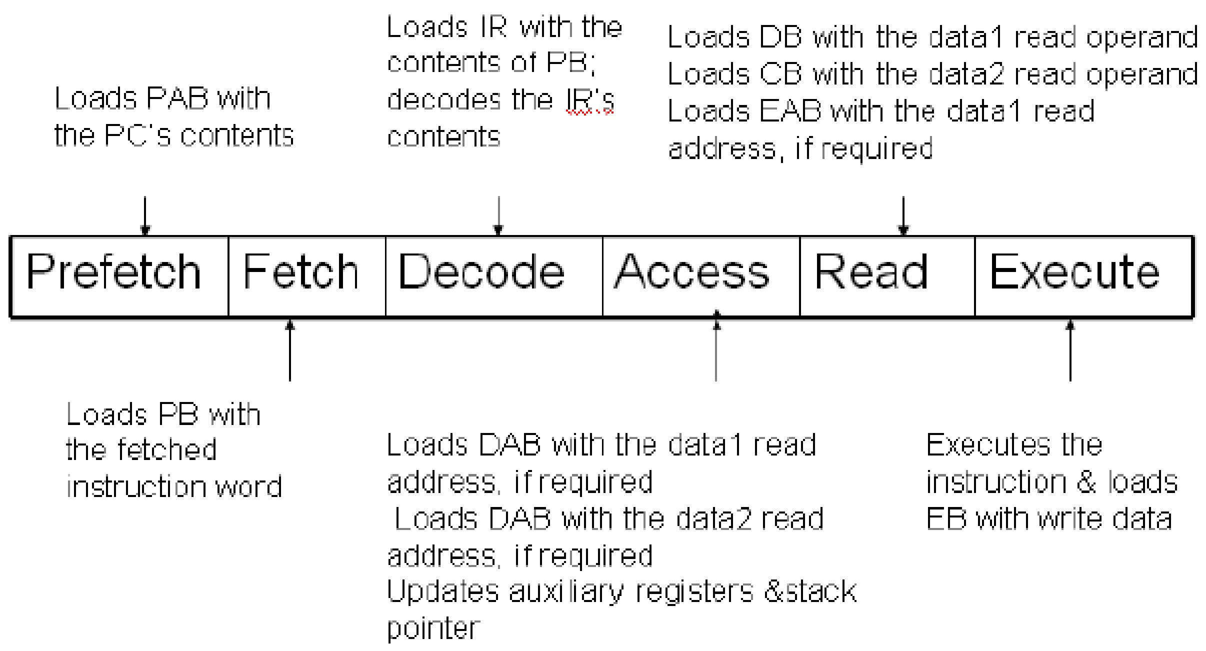

Loads PAB with the PC's contents
Loads IR with the contents of PB; decodes the IR's contents
Loads DB with the data1 read operand
Loads CB with the data2 read operand
Loads EAB with the data1 read address, if required
Prefetch
Fetch
Decode
Access
Read
Execute
Loads PB with the fetched instruction word
Loads DAB with the data1 read address, if required
Loads DAB with the data2 read address, if required
Updates auxiliary registers &stack pointer
Executes the instruction & loads EB with write data

Figure 4.4. Pipeline operation of TMS320C54xx Processors

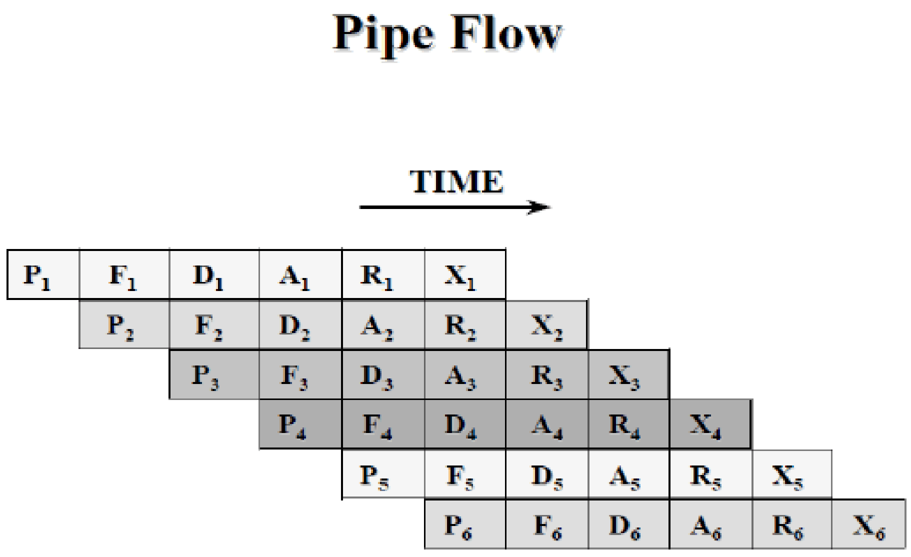


Figure 4.5.Pipe flow diagram

Unit 4

Interfacing Memory & Parallel I/O Peripherals to DSP Devices

Introduction: A typical DSP system has DSP with external memory, input devices and output devices. Since the manufacturers of memory and I/O devices are not same as that of manufacturers of DSP and also since there are variety of memory and I/O devices available, the signals generated by DSP may not suit memory and I/O devices to be connected to DSP. Thus, there is a need for interfacing devices the purpose of it being to use DSP signals to generate the appropriate signals for setting up communication with the memory. DSP with interface is shown in fig. 7.1.

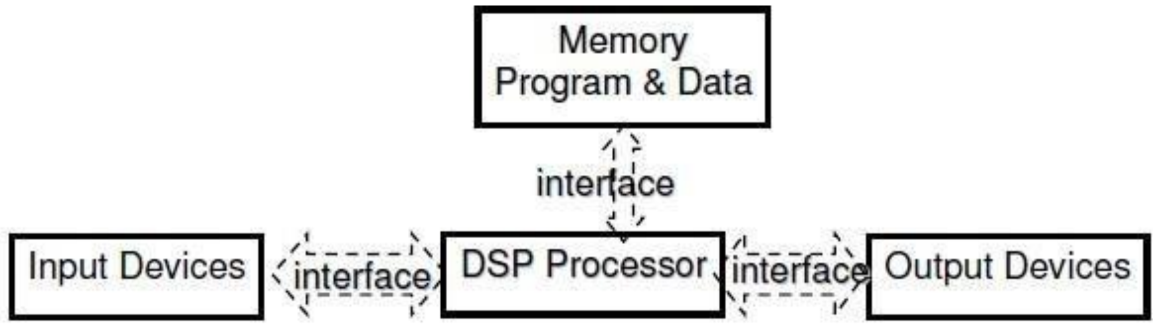


Fig. 7.1: DSP system with interfacing

Memory Space Organization: Memory Space in TMS320C54xx has 192K words of 16 bits each. Memory is divided into Program Memory, Data Memory and I/O Space, each are of 64K words. The actual memory and type of memory depends on particular DSP device of the family. If the memory available on a DSP is not sufficient for an application, it can be interfaced to an external memory as depicted in fig. 7.2. The On- Chip Memory are faster than

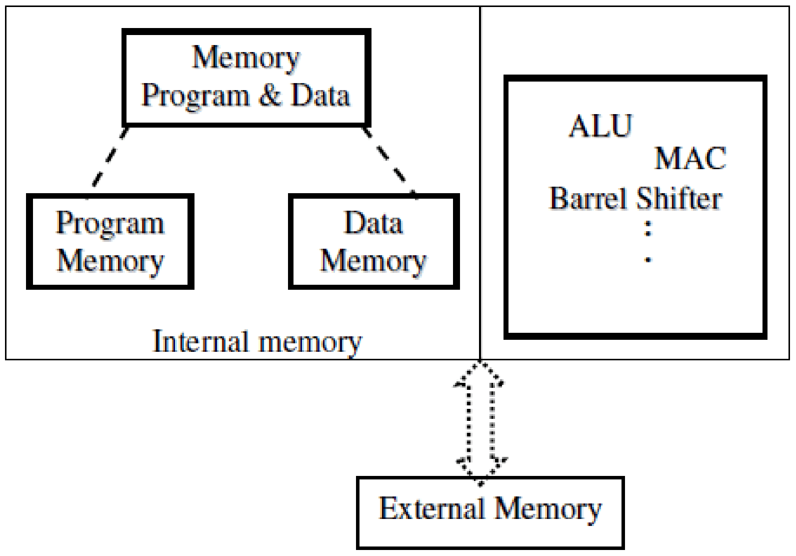


Fig. 7.2: Internal memory and interfacing of external memory

External Memory. There are no interfacing requirements. Because they are on-chip, power consumption is less and size is small. It exhibits better performance by DSP because of better data flow within pipeline. The purpose of such memory is to hold Program / Code / Instructions, to hold constant data such as filter coefficients / filter order, also to hold trigonometric tables / kernels of transforms employed in an algorithm. Not only constants are stored in such memory, they are also used to hold variable data and intermediate results so that the processor need not refer to external memory for the purpose.

External memory is off-chip. They are slower memory. External Interfacing is required to establish the communication between the memory and the DSP. They can be with large memory space. The purpose is being to store variable data and as scratch pad memory. Program memory can be ROM, Dual Access RAM (DARAM), Single Access RAM (SARAM), or a combination of all these. The program memory can be extended externally to 8192K words. That is, 128 pages of 64K words each. The arrangement of memory and DSP in the case of Single Access RAM (SARAM) and Dual Access RAM (DARAM) is shown in fig. 7.3. One set of address bus and data bus is available in the case of SARAM and two sets of address bus and data bus is available in the case of DARAM. The DSP can thus access two memory locations simultaneously.

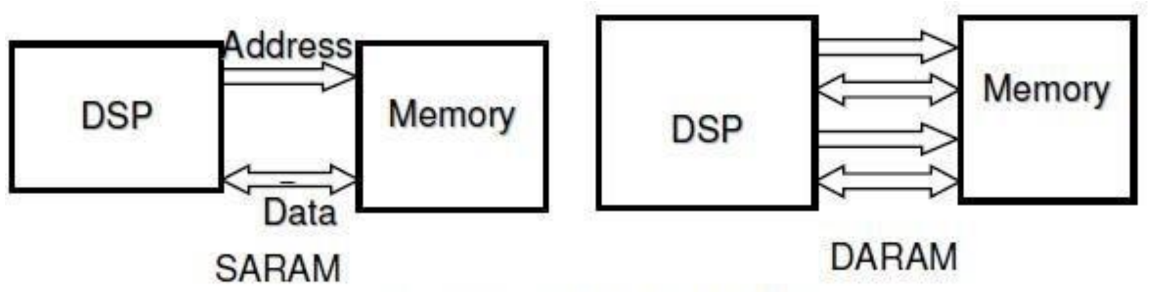
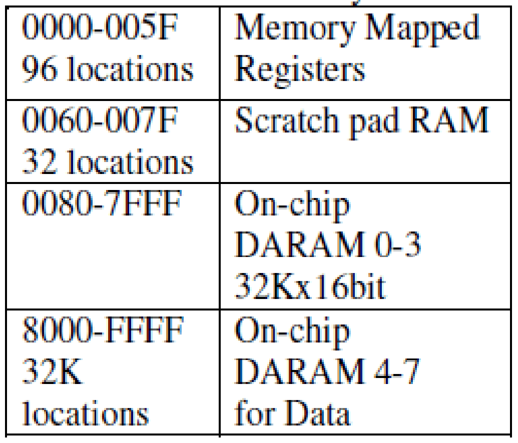


Fig. 7.3: SARAM & DARAM

There are 3 bits available in memory mapped register, PMST for the purpose of on-chip memory mapping. They are microprocessor / microcomputer mode. If this bit is 0, the on-chip ROM is enabled and addressable and if this bit is 1 the on-chip ROM not available. The bit can be manipulated by software / set to the value on this pin at system reset. Second bit is OVLY. It implies RAM Overlay. It enables on-chip DARAM data memory blocks to be mapped into program space. If this bit is 0, on-chip RAM is addressable in data space but not in Program Space and if it is 1, on-chip RAM is mapped into Program & Data Space. The third bit is DROM. It enables on-chip DARAM 4-7 to be mapped into data space. If this bit is 0, on-chip DARAM 4-7 is not mapped into data space and if this bit is 1, on-chip DARAM 4-7 is mapped into Data Space. On-chip data memory is partitioned into several regions as shown in table 7.1. Data memory can be onchip / off-chip.

Table 7.1: Data memory 64 K



0000-005F 96 locations	Memory Mapped Registers
0060-007F 32 locations	Scratch pad RAM
0080-7FFF	On-chip DARAM 0-3 32Kx16bit
8000-FFFF 32K locations	On-chip DARAM 4-7 for Data

The on-chip memory of TMS320C54xx can be both program & data memory. It enhances speed of program execution by using parallelism. That is, multiple data access capability is provided for concurrent memory operations. The number of operations in single memory access is 3 reads & one write. The external memory to DSP can be interfaced with 16 -23 bit Address Bus, 16 bit Data Bus. Interfacing Signals are generated by the DSP to refer to external memory. The signals required by the memory are typically chip Select, Output Enable and Write Enable. For example, TMS320C5416 has 16K ROM, 64K DARAM and 64K SARAM.
Extended external Program Memory is interfaced with 23 address lines i.e., 8192K locations. The external memory thus interfaced is divided into 128 pages, with 64K words per page.

: External Bus Interfacing Signals: In DSP there are 16 external bus interfacing signals. The signal is characterized as single bit i.e., single line or multiple bits i.e., Multiple lines / bus. It can be synchronous / asynchronous with clock. The signal can be
active low / active high. It can be output / input Signal. The signal carrying line / lines Can be unidirectional / bidirectional Signal. The characteristics of the signal depend on
the purpose it serves. The signals available in TMS320C54xx are listed in table 7.2 (a) & table 7.2 (b).

Table 7.2 (a) External Bus Interfacing Signals

1	A0-A19	20 bit Address Bus
2	D0-D15	16 bit Data Bus
3	$\overline{\text{DS}}$	Data Space Select
4	$\overline{\text{PS}}$	Program Space Select
5	$\overline{\text{IS}}$	I/O Space Select
6	R / $\overline{\text{W}}$	Read/Write Signal
7	$\overline{\text{MSTRB}}$	Memory Strobe
8	$\overline{\text{IOTRB}}$	I/O Strobe

In external bus interfacing signals, address bus and data bus are multi-lines bus. Address bus is unidirectional and carries address of the location refereed. Data bus is bidirectional and carries data to

or from DSP. When data lines are not in use, they are tri-stated. Data Space Select, Program Space Select, I/O Space Select are meant for data space, program space or I/O space selection. These interfacing signals are all active low. They are active during the entire operation of data memory / program memory / I/O space reference. Read/Write Signal determines if the DSP is reading the external device or writing.

Read/Write Signal is low when DSP is writing and high when DSP is reading. Strobe Interfacing Signals, Memory Strobe and I/O Strobe both are active low. They remain low during the entire read & write operations of memory and I/O operations respectively. External Bus Interfacing Signals from 1-8 are all are unidirectional except Data Bus which is bidirectional. Address Lines are outgoing signals and all other control signals are also outgoing signals.

Table 7.2 (b) External Bus Interfacing Signals

9	READY	Data Ready Signal
10	$\overline{\text{HOLD}}$	Hold Request
11	$\overline{\text{HLDA}}$	Hold Acknowledge
12	$\overline{\text{MSC}}$	Micro State Complete
13	$\overline{\text{IRQ}}$	Interrupt Request
14	$\overline{\text{IACK}}$	Interrupt Acknowledge
15	XF	External Flag Output
16	$\overline{\text{BIO}}$	Branch Control Input

Data Ready signal is used when a slow device is to be interfaced. Hold Request and Hold Acknowledge are used in conjunction with DMA controller. There are two Interrupt related signals: Interrupt Request and Interrupt Acknowledge. Both are active low. Interrupt Request typically for data exchange. For example, between ADC / another Processor. TMS320C5416 has 14 hardware interrupts for the purpose of User interrupt, Mc-BSP, DMA and timer. The External Flag is active high, asynchronous and outgoing control signal. It initiates an action or informs about the completion of a transaction to the peripheral device. Branch Control Input is a active low, asynchronous, incoming control signal. A low on this signal makes the DSP to respond or attend to the peripheral device. It informs about the completion of a transaction to the DSP.

The Memory Interface: The memory is organized as several locations of certain number of bitumber of locations decides the address bus width and memory capacity. The number of bits per locations decides the data bus width and hence word length. Each location has unique address. The demand of an application may be such that memory capacity required is more than that available in a memory IC. That means there are insufficient words in memory IC. Or the word length required may be more than that is available in a memory IC. Thus, there may be insufficient word length. In both the cases, more number of memory ICs are required.

Typical signals in a memory device are address bus to carry address of referred memory location. Data bus carries data to or from referred memory location. Chip Select Signal selects one or more memory ICs among many memory ICs in the system. Write Enable enables writing of data available on data bus to a memory location. Output Enable signal enables the availability of data from a memory location onto the data bus. The address bus is unidirectional, carries address into the memory IC. Data bus is bidirectional. Chip Select, Write Enable and Output Enable control signals are active high or low and they carry signals into the memory ICs. The task of the memory interface is to use DSP signals and generate the appropriate signals for setting up communication with the memory. The logical spacing of interface is shown in fig. 7.4.

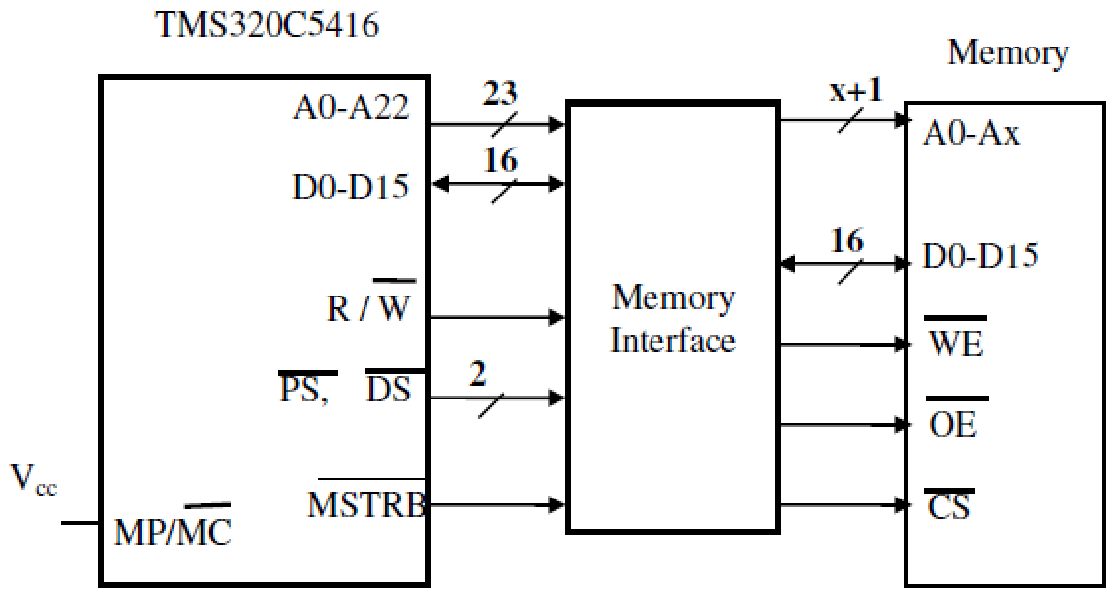


Fig. 7.4 Memory Interface for TMS320C5416

The timing sequence of memory access is shown in fig. 7.5. There are two read operations, both referring to program memory. Read Signal is high and Program Memory Select is low. There is one Write operation referring to external data memory. Data Memory Select is low and Write Signal low. Read and write are to memory device and hence memory strobe is low. Internal program memory reads take one clock cycle and External data memory access require two clock cycles.

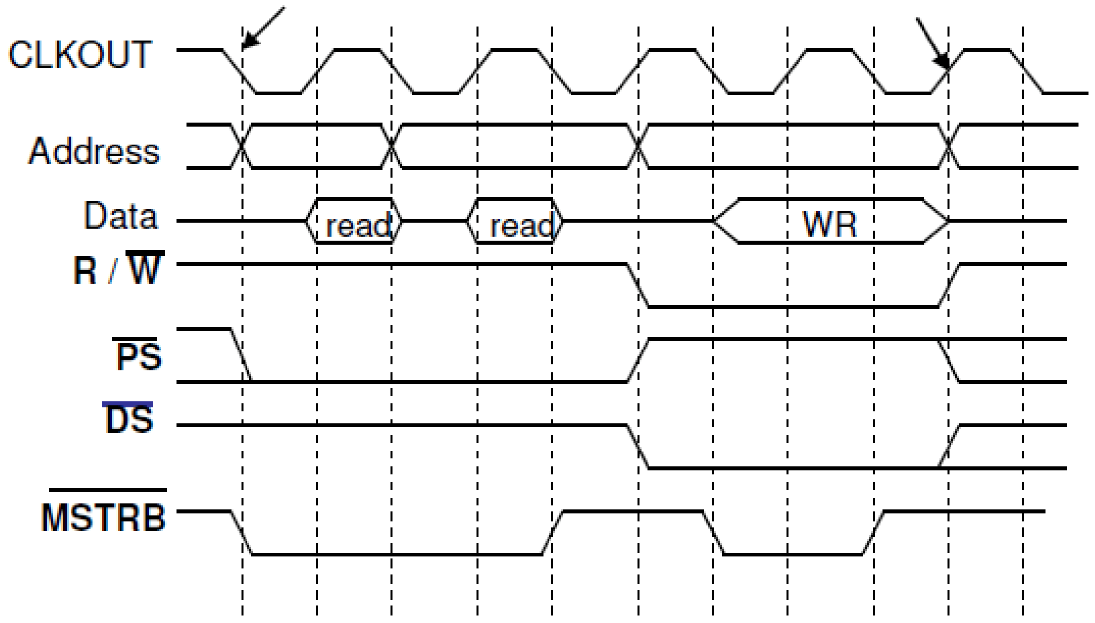


Fig. 7.5 Timing Sequence for External Memory Access

Effects of ‘No decode’ interface are

- Fast memory Access
- ENTIRE Address space is used by the Device that is connected
- Memory responds to 0000-1FFFh and also to all combinations of address bits A13-A19 (In the example quoted)
- Program space select & data space select lines are not used
- SRAM is thus indistinguishable as a program or data Memory

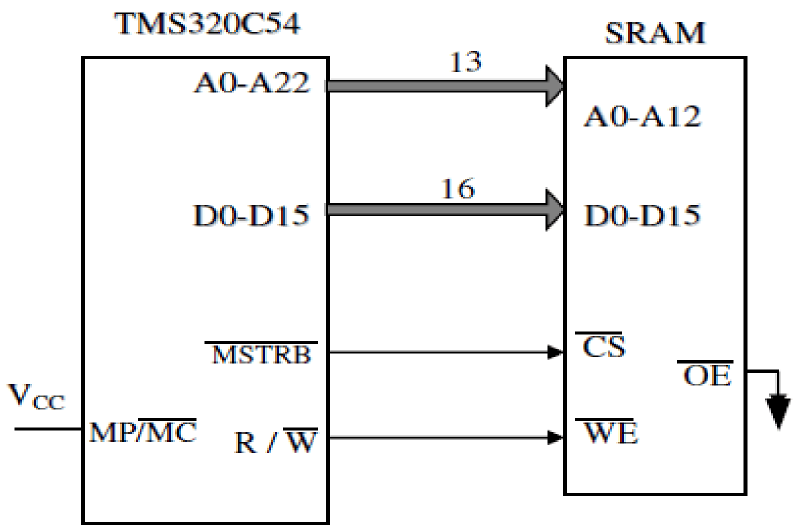


Fig. P7.4: Memory interface without decode circuit

Problem P7.5: Design an interface to connect a 64K x 16 flash memory to a TMS320C54xx device. The Processor address bus to be used is A0-A15. The flash memory has the signals as shown in fig. P7.5.

Solution: Address lines from A0-A15 are used to address 64K locations. All the data lines, D0-D15 are used to carry data word. Data Space Select line is connected to chip enable of memory so that whenever DSP refers to data memory, this flash memory is enabled. When DSP refers to memory and it is a write operation, both memory strobe and read/write signals will be low. They are combined in using OR gate and used as write enable for memory. Memory read is performed by combining memory strobe and XF signals.

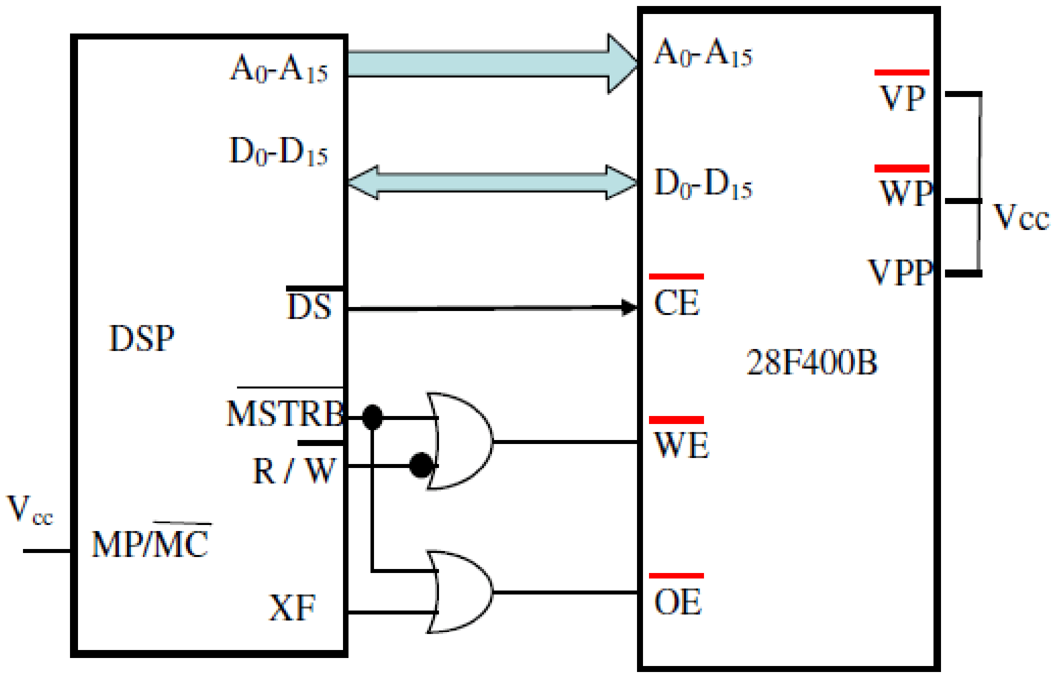


Fig. P7.5: Interfacing flash memory

Parallel I/O Interface: I/O devices are interfaced to DSP using unconditional I/O mode, programmed I/O mode or interrupt I/O mode. Unconditional I/O does not require any handshaking signals. DSP assumes the readiness of the I/O and transfers the data with its own speed. Programmed I/O requires handshaking signals. DSP waits for the readiness of the I/O readiness signal which is one of the handshaking signals. After the
completion of transaction DSP conveys the same to the I/O through another handshaking signal. Interrupt I/O also requires handshaking signals. DSP is interrupted by the I/O indicating the readiness

of the I/O. DSP acknowledges the interrupt, attends to the interrupt. Thus, DSP need not wait for the I/O to respond. It can engage itself in execution as long as there is no interrupt.

: Programmed I /O interface: The timing diagram in the case of programmed I/O is shown in fig. 7.6. I/O strobe and I/O space select are issued by the DSP. Two clock cycles each are required for I/O read and I/O write operations.

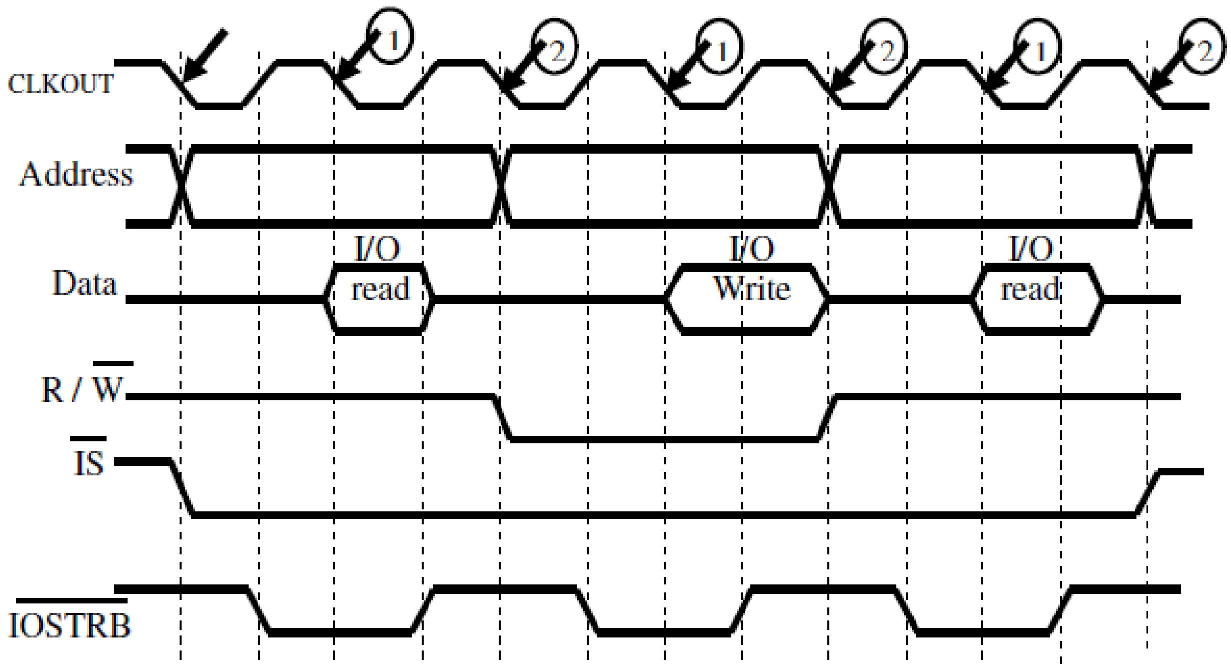


Fig. 7.6: Read-Write-Read Sequence of Operations

An example of interfacing ADC to DSP in programmed I/O mode is shown in fig. 7.7. ADC has a start of conversion (SOC) signal which initiates the conversion. In programmed I/O mode, external flag signal is issued by DSP to start the conversion. ADC issues end of conversion (EOC) after completion of conversion. DSP receives Branch input control by ADC when ADC completes the conversion. The DSP issues address of the ADC, I/O strobe and read / write signal as high to read the data. An address decoder does the translation of this information into active low read signal to ADC. The data is supplied on data bus by ADC and DSP reads the same. After reading,
DSP issues start of conversion once again after the elapse of sample interval. Note that
there are no address lines for ADC. The decoded address selects the ADC. During conversion, DSP waits checking branch input control signal status for zero. The flow chart of the activities in programmed I/O is shown in fig. 7.8.

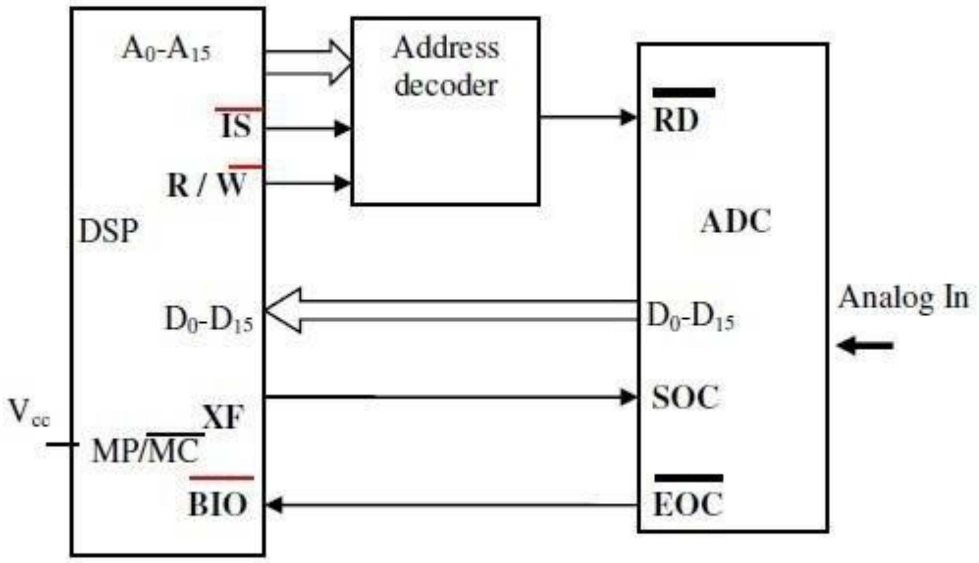


Fig. 7.7: ADC in Programmed I/O mode

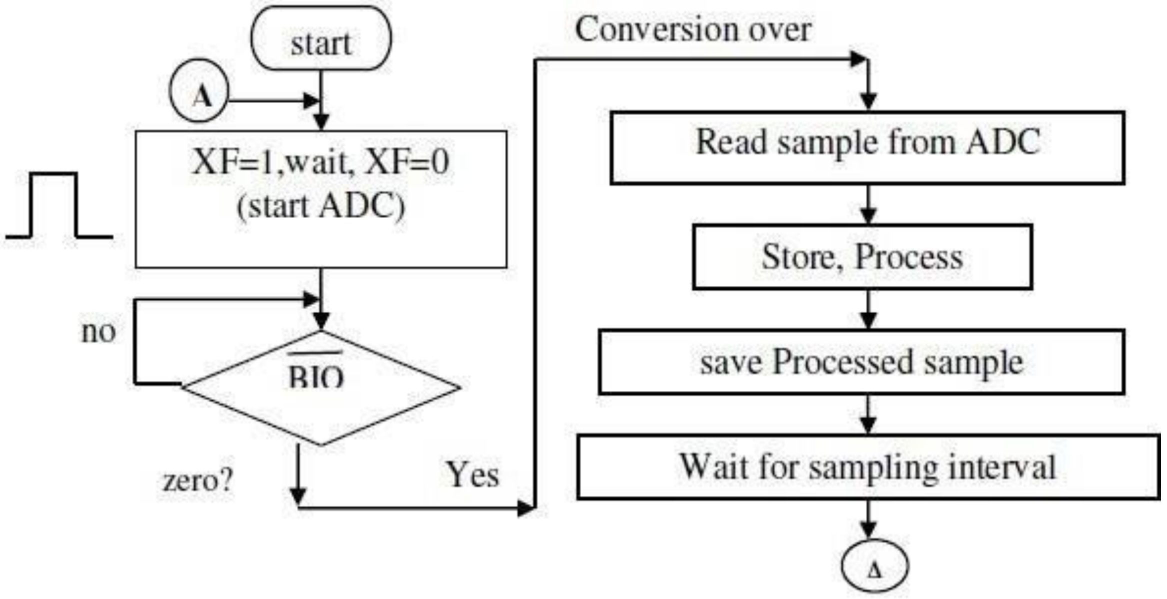


Fig. 7.8: Programmed I/O mode

Interrupt I/O: This mode of interfacing I/O devices also requires handshaking signals. DSP is interrupted by the I/O whenever it is ready. DSP Acknowledges the interrupt, after testing certain conditions, attends to the interrupt. DSP need not wait for the I/O to respond. It can engage itself in execution. There are a variety of interrupts. One of the classifications is maskable and nonmaskable. If maskable, DSP can ignore when that interrupt is masked. Another classification is vectored and non-vectored. If vectored, Interrupt Service subroutine (ISR) is in specific location. In Software Interrupt, instruction is written in the program.

In Hardware interrupt, a hardware pin, on the DSP IC will receive an interrupt by the external device. Hardware interrupt is also referred to as external interrupt and software interrupt is referred to as internal interrupt. Internal interrupt may also be due to execution of certain instruction can causing interrupt. In TMS320C54xx there are total of 30 interrupts. Reset, Non-maskable, Timer Interrupt, HPI, one each, 14 Software Interrupts, 4 External user Interrupts, 6 Mc-BSP related Interrupts and 2 DMA related Interrupts. Host Port Interface (HPI) is a 8 bit parallel port. It is possible to interface to a Host Processor using HPI. Information exchange is through on-chip memory of DSP
which is also accessible Host processor.

Registers used in managing interrupts are Interrupt flag Register (IFR) and Interrupt Mask Register (IMR). IFR maintains pending external & internal interrupts. One in any bit position implies pending interrupt. Once an interrupt is received, the orresponding bit is set. IMR is used to mask or unmask an interrupt. One implies that the corresponding interrupt is unmasked. Both these registers are Memory Mapped Registers. One flag, Global enable bit (INTM), in ST1 register is used to enable or disable all interrupts globally. If INTM is zero, all unmasked interrupts are enabled. If it is one, all maskable interrupts are disabled.

When an interrupt is received by the DSP, it checks if the interrupt is maskable. If the interrupt is non-maskable, DSP issues the interrupt acknowledgement and thus serves the interrupt. If the interrupt is hardware interrupt, global enable bit is set so that no other interrupts are entertained by the DSP. If the interrupt is maskable, status of the INTM is checked. If INTM is 1, DSP does not respond to the interrupt and it continues with program execution. If the INTM is 0, bit in IMR register corresponding to the interrupt is checked. If that bit is 0, implying that the interrupt is masked, DSP does not respond to the interrupt and continues with its program execution. If the interrupt is unmasked, then DSP issues interrupt acknowledgement. Before branching to the interrupt service routine, DSP saves the PC onto the stack. The same will be reloaded after attending the interrupt so as to return to the program that has been interrupted. The response of DSP to an Interrupt is shown in flow chart in fig. 7.9.

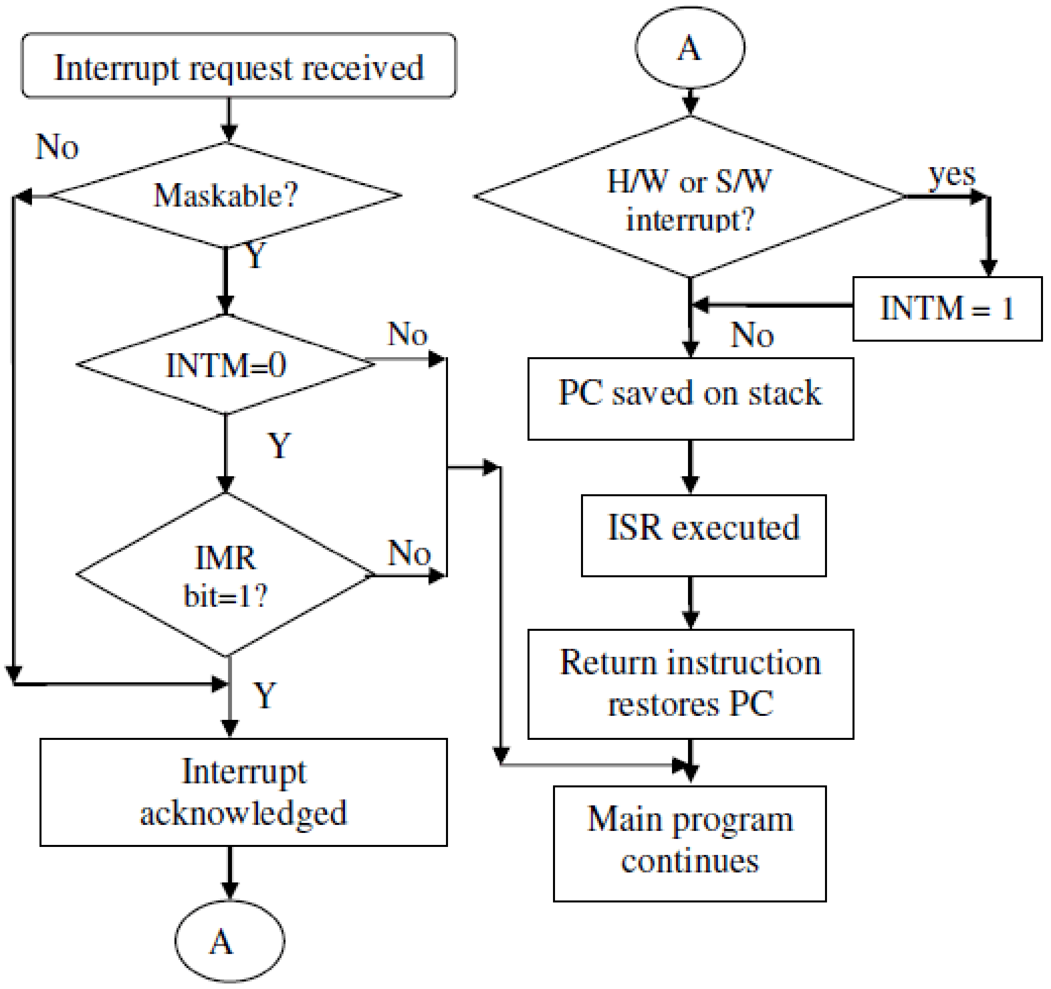


Fig. 7.9: Response of DSP to interrupt

: Direct Memory Access (DMA) operation: In any application, there is data transfer between DSP and memory and also DSP and I/O device, as shown in fig. 7.10. However, there may be need for transfer of large amount of data between two memory regions or between memory and I/O. DSP can be involved in such transfer, as shown in fig. 7.11. Since amount of data is large, it will engage DSP in data transfer task for a long time. DSP thus will not get utilized for the purpose it is meant for, i.e., data manipulation. The intervention of DSP has to be avoided for two reasons: to utilize DSP for useful signal processing task and to increase the speed of transfer by direct data transfer between memory or memory and I/O. The direct data transfer is referred to as direct memory access (DMA). The arrangement expected is shown in fig. 7.12. DMA controller helps in data transfer instead of DSP.

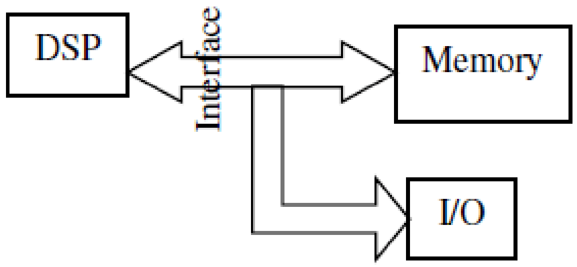


Fig. 7.10: Interface between DSP and external devices

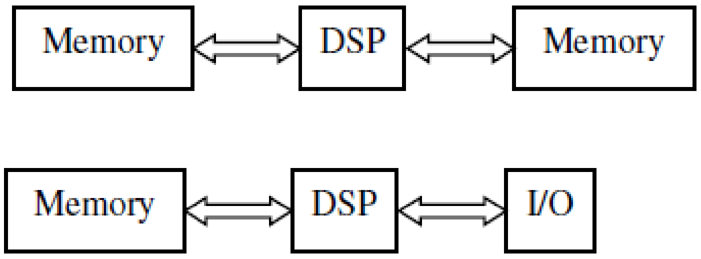


Fig. 7.11: Data transfer with intervention by DSP

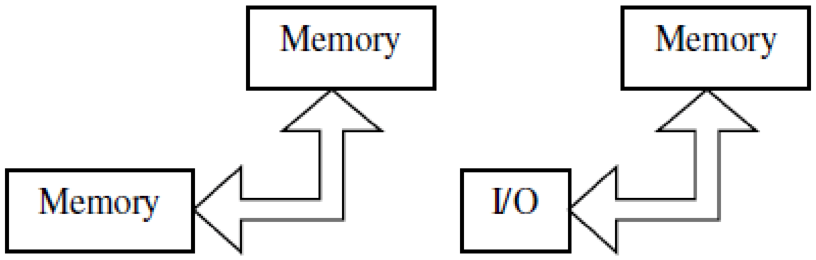


Fig. 7.12: data transfer without intervention by DSP

In DMA, data transfer can be between memory and peripherals which are either internal
or external devices. DMA controller manages DMA operation. Thus DSP is relieved of the task of data transfer. Because of direct transfer, speed of transfer is high. In TMS320C54xx, there are up to 6 independent programmable DMA channels. Each channel is between certain source & destination. One channel at a time can be used for

data transfer and not all six simultaneously. These channels can be prioritized. The speed of transfer measured in terms of number of clock cycles for one DMA transfer depends on several factors such as source and destination location, external interface conditions, number of active DMA channels, wait states and bank switching time. The time for data transfer between two internal memory is 4 cycles for each word.

Requirements of maintaining a channel are source & Destination address for a channel, separately for each channel. Data transfer is in the form of block, with each block having frames of 16 / 32 bits. Block size, frame size, data are programmable. Along with these, mode of transfer and assignment of priorities to different channels are also to be maintained for the purpose of data transfer.

There are five, channel context registers for each DMA channel. They are Source Address Register (DMSRC), Destination Address Register (DMDST), Element Count Register (DMCTR), Sync select & Frame Count register (DMSFC), Transfer Mode Control Register (DMMCR). There are four reload registers. The context register DMSRC & DMDST are source & destination address holders. DMCTR is for holding number of data elements in a frame. DMSFC is to convey sync event to use to trigger DMA transfer, word size for transfer and for holding frame count. DMMCR Controls transfer mode by specifying source and destination spaces as program memory, data memoryor I/O space. Source address reload & Destination address reload are useful in reloading source address and destination address. Similarly, count reload and frame count reload are used in reloading count and frame count. Additional registers for DMA that are common to all channels are Source Program page address, DMSRCP, Destination Program page address, DMDSTP, Element index address register, Frame index address register.

Number of memory mapped registers for DMA are 6x(5+4) and some common registers for all channels, amounting to total of 62 registers required. However, only 3 (+1 for priority related) are available. They are DMA Priority & Enable Control Register (DMPREC), DMA sub bank Address Register (DMSA), DMA sub bank Data Register with auto increment (DMSDI) and DMA sub bank Data Register (DMSDN). To access each of the DMA Registers Register sub addressing Technique is employed. The schematic of the arrangement is shown in fig. 7.13. A set of DMA registers of all channels (62) are made available in set of memory locations called sub bank. This voids the need for 62 memory mapped registers. Contents of either DMSDI or DMSDN indicate the code (1’s & 0’s) to be written for a DMA register and contents of DMSA refers to the unique sub address of DMA register to be accessed. Mux routes either DMSDI or DMSDN to the sub bank. The memory location to be written

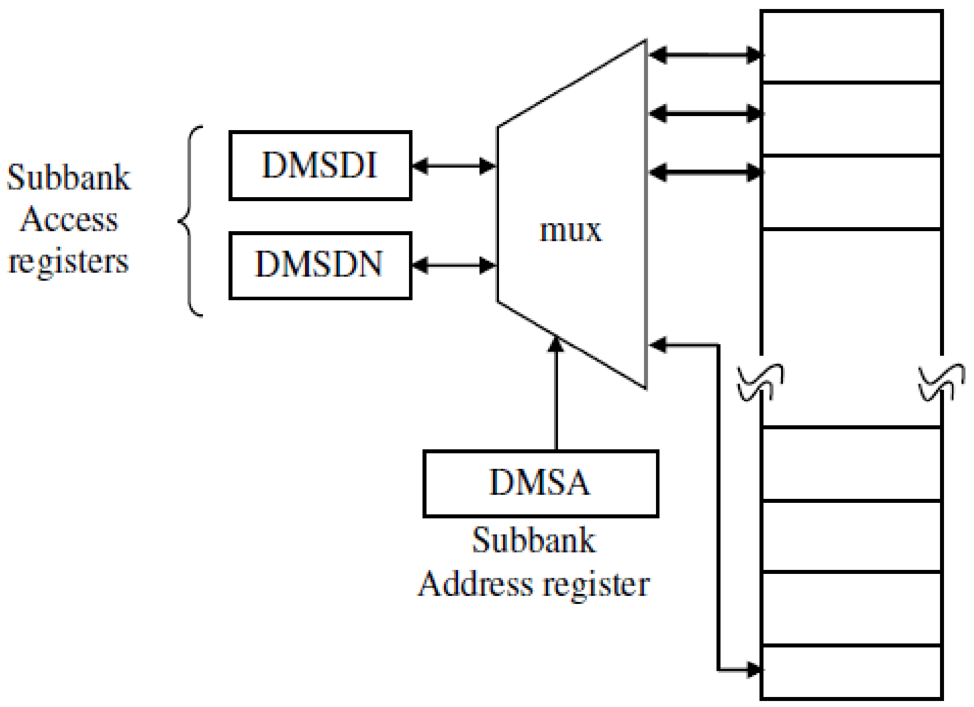


Fig. 7.13: Register Subaddress Technique

DMSDI is used when an automatic increment of the sub address is required after each access. Thus it can be used to configure the entire set of registers. DMSDN is used when single DMA register access is required. The following examples bring out clearly the method of accessing the DMA registers and transfer of data in DMA mode.

UNIT-5

Implementation of Basic DSP Algorithms

Introduction:

In this unit, we deal with implementations of DSP algorithms & write programs to implement the core algorithms only. However, these programs can be combined with input/output routines to create applications that work with a specific hardware.

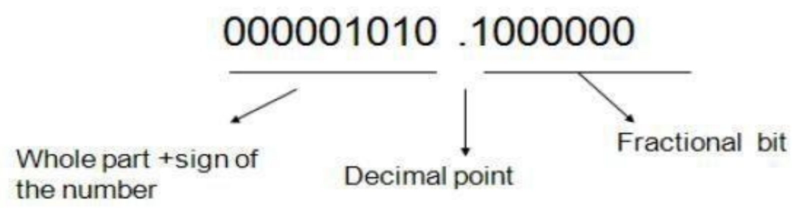
- Q-notation
- FIR filters
- IIR filters
- Interpolation filters
- Decimation filters

The Q-notation:

DSP algorithm implementations deal with signals and coefficients. To use a fixed point DSP device efficiently, one must consider representing filter coefficients and signal samples using fixed-point2's complement representation. Ex: N=16, Range: -2N-1 to +2N-1 -1(-32768 to 32767).Typically, filter coefficients are fractional numbers.

To represent such numbers, the Q-notation has been developed. The Q-notation specifies the number of fractional bits.

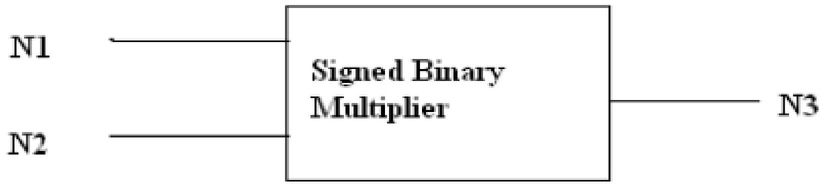
Ex: Q7



A commonly used notation for DSP implementations is Q15. In the Q15 representation, the least significant 15 bits represent the fractional part of a number. In a processor where 16 bits are used to represent numbers, the Q15 notation uses the MSB to represent the sign of the number and the rest of the bits represent the value of the number.
In general, the value of a 16-bit Q15 number N represented as:

$b_{15}\ldots\ldots\ldots\ldots b_1 b_0$
$N = -b_{15} + b_{14}2^{-1} + \ldots\ldots\ldots\ldots + b_0 2^{-15}$
Range:-1 to $1 - 2^{-15}$

Multiplication of numbers represented using the Q-notation is important for DSP implementations. Figure 5.1(a) shows typical cases encountered in such implementations.



N1(16 bit)	N2(16 bit)	N3(16 bit)
Q0	Q0	Q0
Q0	Q15	Q15
Q15	Q15	Q30

Figure 5.1Multiplication of numbers represented using Q-notation

Program to multiply two Q15 numbers

i.e N1×N2 = N1*N2

Where

N1 &N2 are 16-bit numbers in Q15 notation
N1×N2 is the 16-bit result in Q15 notation

```
                          .mmregs                ; memory mapped registers
                          .data                  ; sequential locations
N1:                       .word    4000h         ; N1=0.5 (Q15 numbers)
N2:                       .word    2000h         ; N2=0.25 (Q15 numbers)
N1×N2                     .space   10h           ; space for N1×N2
                          .text
                          .ref           _c_int00
                          .sect          ".vectors "
RESET:          b         _c_int00               ; reset vector
                nop

                         nop

          :_int00
                         STM   #N1,AR2         ;AR2 points to N1
                         LD    *AR2+, T        ;T  reg =N1
                         MPY    *AR2+, A       ;A= N1 *N2 in Q30 notation
                         ADD   #1, 14, A       ;round the result
                         STH   A, 1, *AR2      ;save N1 *N2 as Q15 number
                         NOP
                         NOP
                         .end
```

FIR Filters:

A finite impulse response (FIR) filter of order N can be described by the difference equation.

$$y[n] = \sum_{m=0}^{m=N-1} h(m)x(n-m)$$

The expanded form is y(n)=h(N-1)x(n-(N-1))+h(N-2)x(n-(N-2))+ ...h(1)x(n-1)+h(0)x(n)

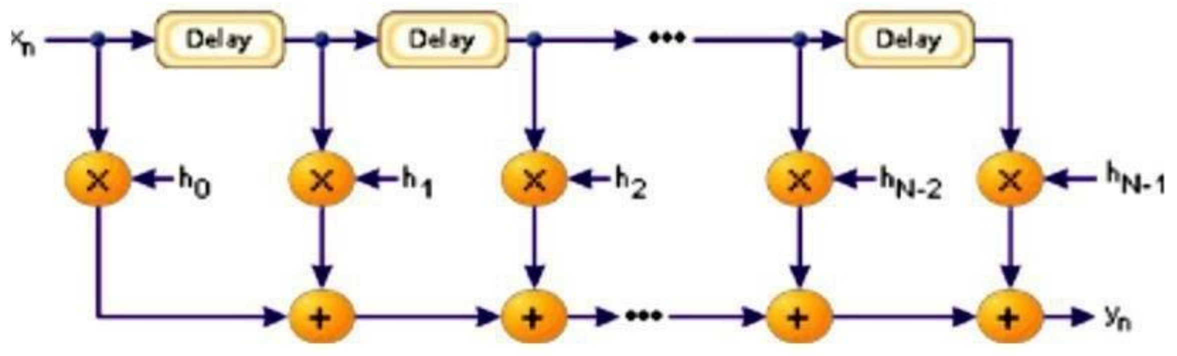


Figure 5.2 A FIR filter implementation block diagram

The implementation requires signal delay for each sample to compute the next output, y(n+1), is given as y(n+1)=h(N-1)x(n-(N-2))+h(N-2)x(n-(N-3))+ ...h(1)x(n)+h(0)x(n+1) Figure 5.3 shows the memory organization for the implementation of the filter. The filter Coefficients and the signal samples are stored in two circular buffers each of a size equal to the filter. AR2 is used to point to the samples and AR3 to the coefficients. In order to start with the last product, the pointer register AR2 must be initialized to access the signal sample x(2-(N-1)), and the pointer register AR3 to access the filter coefficient h(N-1). As each product is computed and added to the previous result, the pointers advance circularly. At the end of the computation, the signal sample pointer is at the oldest sample, which is replaced with the newest sample to proceed with the next output computation.

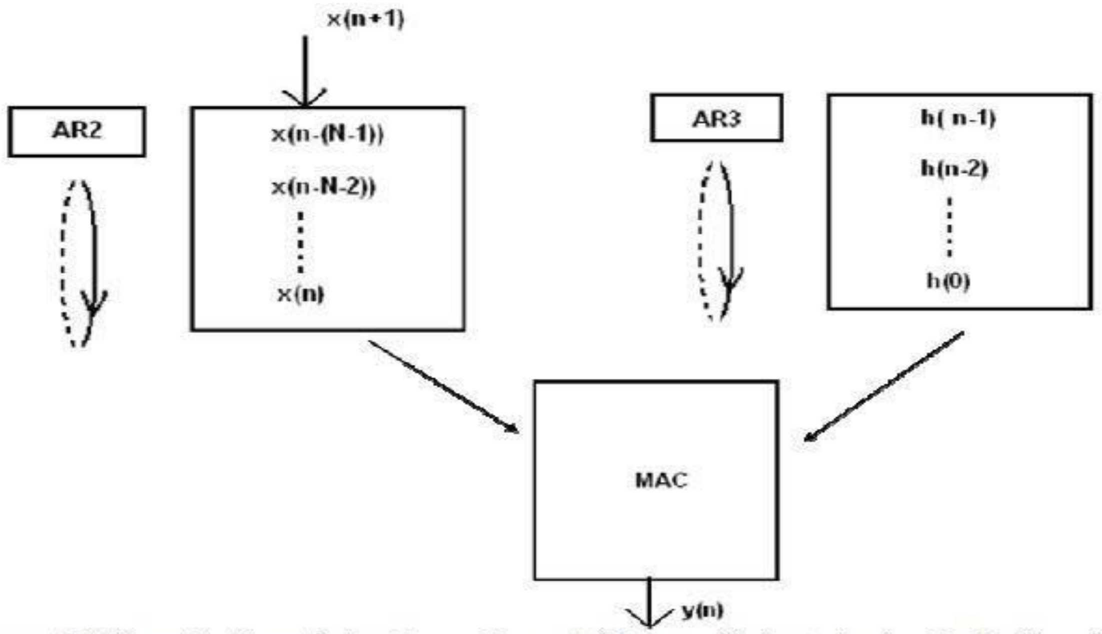


Figure 5.3 Organization of signal samples and filter coefficients in circular buffers for a FIR filter implementation.

Program to implement an FIR filter:

It implements the following equation;

y(n)=h(N-1)x(n-(N-1))+h(N-2)x(n-(N-2))+ ...h(1)x(n-1)+h(0)x(n)

Where N = Number of filter coefficients = 16.

h(N-1), h(N-2),...h(0) etc are filter coefficients (q15numbers) .

The coefficients are available in file: coeff_fir.dat.

x(n-(N-1)),x(n-(N-2),...x(n) are signal samples(integers).

The input x(n) is received from the data file: data_in.dat.

The computed output y(n) is placed in a data buffer.

```
                .mmregs
                .def _c_int00
                .sect "samples"
InSamples       .include "data_in.dat"        ; Allocate space for x(n)s
OutSamples      .bss y, 200,1              ; Allocate space for y(n)s
SampleCnt       .set 200                   ; Number of samples to
                                                   filter
                .bss CoefBuf, 16, 1           ; Memory for coeff circular
                                                    buffer
                .bss SampleBuf, 16, 1   ; Memory for sample  circular buffer
            .sect "FirCoeff"              ; Filter coeff (seq locations)
FirCoeff       .include "coff_fir.dat“
Nm1            .set 15                        ; N – 1


                .text
 _c_int00:
                STM #OutSamples, AR6     ; clear o/p sample buffer
                RPT #SampleCnt
                ST #0, *AR6+
                STM #InSamples, AR5      ; AR5 points to InSamples buffer
                STM #OutSamples, AR6      ; AR6 points to OutSample buffer
                STM #SampleCnt, AR4     ; AR4 = Number of samples to
                                                   filter

                CALL fir_init                    ; Init for filter calculations
                SSBX SXM                ; Select sign extension mode

loop:
                LD *AR5+,A             ; A = next input sample (integer)
                CALL fir_filter                  ; Call Filter Routine
                STH A,1,*AR6+                    ; Store filtered sample (integer)
                BANZ  loop,*AR4-   ; Repeat till all samples filtered
                 nop
                 nop
                 nop
```

FIR Filter Initialization Routine

```
; This routine sets AR2 as the pointer for the sample circular buffer
; AR3 as the pointer for coefficient circular buffer.
; BK = Number of filter taps - 1.
; AR0 = 1 = circular buffer pointer increment
```

```
fir_init:

        ST #CoefBuf,AR3             ; AR3 is the CB Coeff Pointer
        ST #SampleBuf,AR2                   ; AR2 is the CB sample pointer
        STM #Nm1,BK                         ; BK = number of filter taps
        RPT #Nm1
        MVPD #FirCoeff, *AR3+%              ; Place coeff in circular buffer
        RPT #Nm1 - 1                        ; Clear circular sample buffer
        ST #0h,*AR2+%
        STM #1,AR0                  ; AR0 = 1 = CB pointer increment
        RET
        nop
        nop
        nop
```

FIR Filter Routine

; Enter with A=the current sample x(n)-an integer, AR2 pointing to the location for the current sample x(n),andAR3pointingtotheq15coefficienth(N-1). Exit with A = y(n) as q15 number.

```
fir_filter:

            STL A, *AR2+0%              ; Place x(n)in the sample buffer
            RPTZ A, #Nm1                        ; A = 0
            MAC *AR3+0%,*AR2+0%,A       ; A = filtered sum (q15)
            RET
            nop
            nop
            nop
            .end
```

IIR Filters:

An infinite impulse response (IIR) filter is represented by a transfer function, which is a ratio of two polynomials in z. To implement such a filter, the difference equation representing the transfer function can be derived and implemented using multiply and add operations. To show such an implementation, we consider a second order transfer function given by

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}}$$

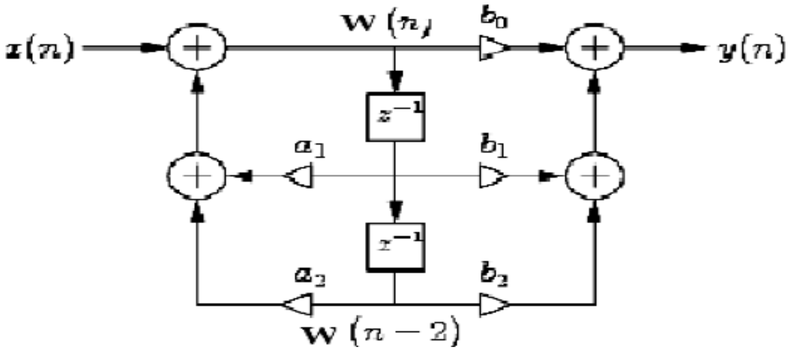


Figure5.4 Block diagram of second order IIR filter

$$\begin{aligned} w(n) &= x(n) + a_1 w(n-1) + a_2 w(n-2) \\ y(n) &= b_0 w(n) + b_1 w(n-1) + b_2 w(n-2) \end{aligned}$$

Program for IIR filter:

The transfer function is

H(z) = [b0 + b1.z(-1)+ b2.z**(-2)]/[1- a1.z**(-1)- a2.z**(-2)]**

Which is equivalent to the equations:

w(n) = x(n) + a1.w(n-1) + a2.w(n-2)

y(n) = b0.w(n) + b1.w(n-1) + b2.w(n-2)

Where w(n), w(n-1), and w(n-2) are the intermediate variables used in computations (integers).a1, a2, b0, b1, and b2 are the filter coefficients (q15 numbers). x(n) is the input sample (integer). Input samples are placed in the buffer, In Samples, from a data file, data_in.dat y(n) is the computed output (integer). The output samples are placed in a buffer, Out Samples.

```
                    .mmregs
                    .def _c_int00
                    .sect "samples"
InSamples           .include "data_in.dat"      ; Allocate space for x(n)s
OutSamples          .bss y,200,1                ; Allocate buffer for y(n)s
SampleCnt           .set 200                    ; Number of samples to filter

; Intermediate variables (sequential locations)
wn          .word 0                    ;initial w(n)
wnm1 .word 0                    ;initial w(n-1) =0
wnm2 .word 0                    ;initial w(n-2)=0


            .sect "coeff"
; Filter coefficients (sequential locations)
b0          .word 3431                 ; b0 = 0.104
b1          .word -3356         ; b1 = -0.102
b2          .word 3431                 ; b2 = 0.104
a1          .word -32767        ; a1 = -1
a2          .word 20072         ; a2 = 0.612


            .text

_c_int00:

        STM #OutSamples, AR6     ; Clear output sample buffer
        RPT #SampleCnt
        ST #0, *AR6+
        STM #InSamples, AR5              ; AR5 points to InSamples buffer
        STM #OutSamples, AR6     ; AR6 points to OutSample buffer
        STM #SampleCnt, AR4      ; AR4 = Number of samples to filter

 loop:
        LD *AR5+,15,A                    ; A = next input sample (q15)
        CALL iir_filter                  ; Call Filter Routine
        STH A,1,*AR6+                    ; Store filtered sample (integer)
        BANZ loop,*AR4-          ; Repeat till all samples filtered
        nop
        nop
        nop
```

```
IIR Filter Subroutine
; Enter with A = x(n) as q15 number
; Exit with A = y(n) as q15 number
; Uses AR2 and AR3

iir_filter:
                SSBX SXM    ; Select sign extension mode
        ;w(n)=x(n)+ a1.w(n-1)+ a2.w(n-2)

        STM #a2,AR2                         ; AR2 points to a2
        STM #wnm2, AR3                  ; AR3 points to w(n-2)
        MAC *AR2-,*AR3-,A                   ; A = x(n)+ a2.w(n-2)
                                            ; AR2 points to a1 & AR3 to w(n-
1)
        MAC *AR2-,*AR3-,A                   ; A = x(n)+ a1.w(n-1)+ a2.w(n-2)
                                            ; AR2 points to b2 & AR3 to w(n)
        STH A,1,*AR3                        ; Save w(n)

;y(n)=b0.w(n)+ b1.w(n-1)+ b2.w(n-2)

        LD #0,A                 ; A = 0
        STM #wnm2,AR3           ; AR3 points to w(n-2)
        MAC *AR2-,*AR3-,A       ; A = b2.w(n-2)
                                ; AR2 points to b1 & AR3 to w(n-1)
        DELAY *AR3              ; w(n-1) -> w(n-2)
        MAC *AR2-,*AR3-,A  ; A = b1.w(n-1)+ b2.w(n-2)
                                ; AR2 points to b0 & AR3 to w(n)
        DELAY *AR3              ; w(n) -> w(n-1)

 MAC *AR2,*AR3,A ; A = b0.w(n)+ b1.w(n-1)+ b2.w(n-2)
 RET                         ; Return
 Nop
 Nop
 Nop
 .end
```

Interpolation Filters:

An *interpolation filter* is used to increase the sampling rate. The interpolation process involves inserting samples between the incoming samples to create additional samples to increase the sampling rate for the output. One way to implement an interpolation filter is to first insert zeros between samples of the original sample sequence. The zero-inserted sequence is then passed through an appropriate lowpass digital FIR filter to generate the interpolated sequence. The interpolation process is depicted in Figure 5.5

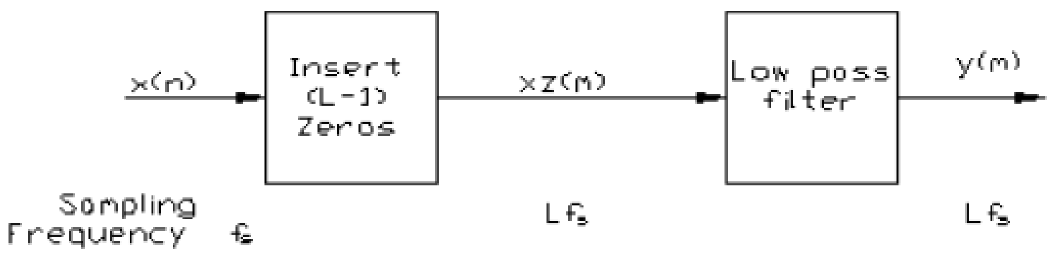


Figure 5.5 :The interpolation process

Example:

X(n) = [0 2 4 6 8 10]	;input sequence
Xz(n) = [0 0 2 0 4 0 6 0 8 0 10 0]	;zero inserted sequence
h(n) = [0.5 1 0.5]	;impulse sequence
Y(n) = [0 0 1 2 3 4 5 6 7 8 9 10 5 0]	;interpolated sequence y(n)

The kind of interpolation carried out in the examples is called *linear interpolation* because the convolving sequence h(n) is derived based on linear interpolation of samples. Further, in this case, the h(n) selected is just a second-order filter and therefore uses just two adjacent samples to interpolate a sample. A higher-order filter can be used to base interpolation on more input samples. To implement an ideal interpolation. Figure 5.6 shows how an interpolating filter using a 15-tap FIR filter and an interpolation factor of 5 can be implemented. In this example, each incoming samples is followed by four zeros to increase the number of samples by a factor of 5.

The interpolated samples are computed using a program similar to the one used for a FIR filter implementation. One drawback of using the implementation strategy depicted in Figure 5.7 is that there are many multiplies in which one of the multiplying elements is zero. Such multiplies need not be included in computation if the computation is rearranged to take advantage of this fact. One such scheme, based on generating what are called *poly-phase sub-filters*, is available for reducing the computation. For a case where the number of filter coefficients N is a multiple of the interpolating factor L, the scheme implements the interpolation filter using the equation.

Figure 5.7 shows a scheme that uses poly-phase sub-filters to implement the interpolating filter using the 15-tap FIR filter and an interpolation factor of 5. In this implementation, the 15 filter taps are arranged as shown and divided into five 3-tap sub filters. The input samples x(n), x(n-1) and x(n-2) are used five times to generate the five output samples. This implementation requires 15 multiplies as opposed to 75 in the direct implementation of Figure 5.7.

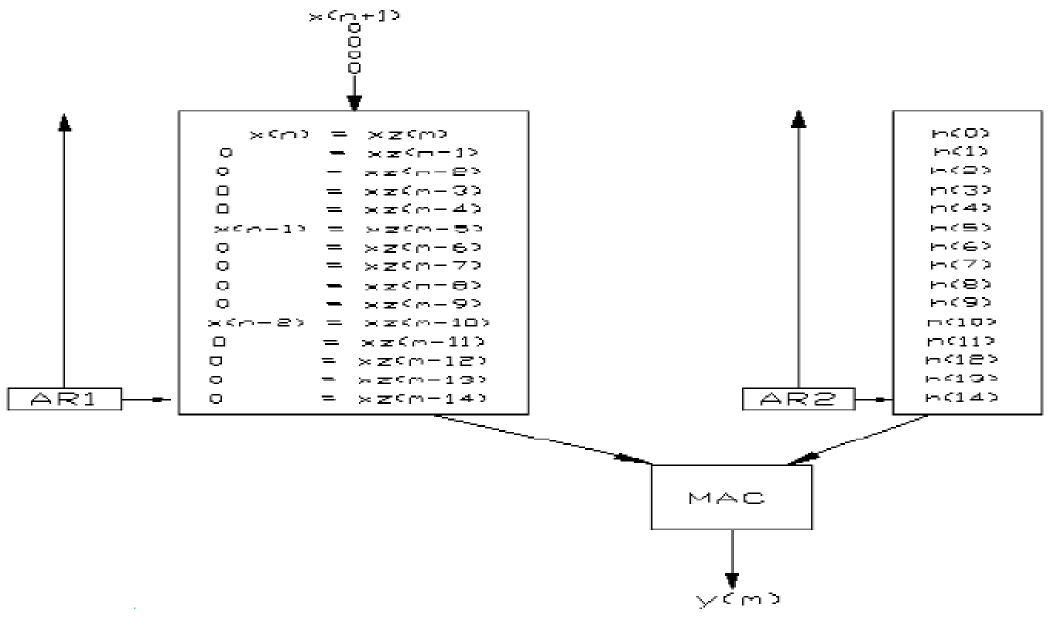


Figure 5.6 interpolating filter using a 15-tap FIR filter and an interpolation factor of 5

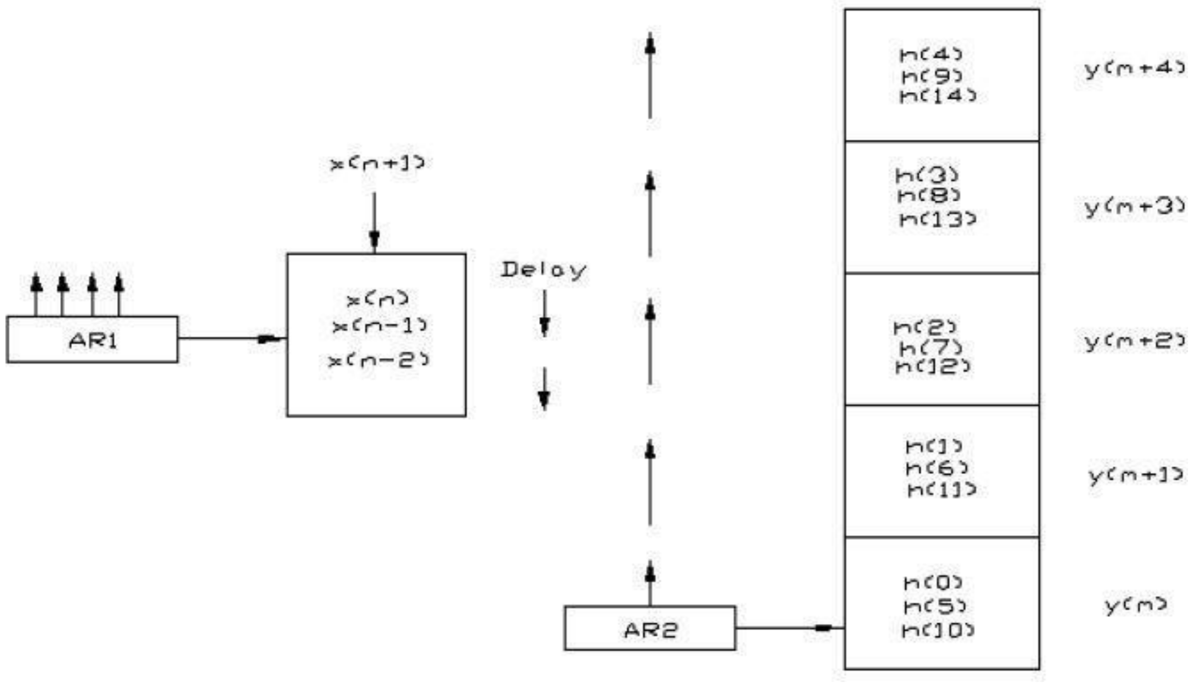


Figure5.7: A scheme that uses poly-phase sub-filters to implement the interpolating filter Using the 15-tap FIR filter and an interpolation factor of 5

$$y(m+i) = \sum_{K=0}^{N/L-1} h(KL+i)x(n-K)$$

Where i = 0,1,2,…. (L-1) and m = nL.

Decimation Filters:

A decimation filter is used to decrease the sampling rate. The decrease in sampling rate can be achieved by simply dropping samples. For instance, if every other sample of a sampled sequence is dropped, the sampling the rate of the resulting sequence will be half that of the original sequence. The problem with dropping samples is that the new sequence may violate the sampling theorem, which requires that the sampling frequency must be greater than two times the highest frequency contents of the signal.

To circumvent the problem of violating the sampling theorem, the signal to be decimated is first filtered using a low pass filter. The cutoff frequency of the filter is chosen so that it is less than half the final sampling frequency. The filtered signal can be decimated by dropping samples. In fact, the samples that are to be dropped need not be computed at all. Thus, the implementation of a decimator is just a FIR filter implementation in which some of the outputs are not calculated.

Figure 5.8 shows a block diagram of a decimation filter. Digital decimation can be implemented as depicted in Figure 5.9 for an example of a decimation filter with decimation factor of 3. It uses a low pass FIR filter with 5 taps. The computation is similar to that of a FIR filter. However, after computing each output sample, the signal array is delayed by three sample intervals by bringing the next three samples into the circular buffer to replace the three oldest samples.

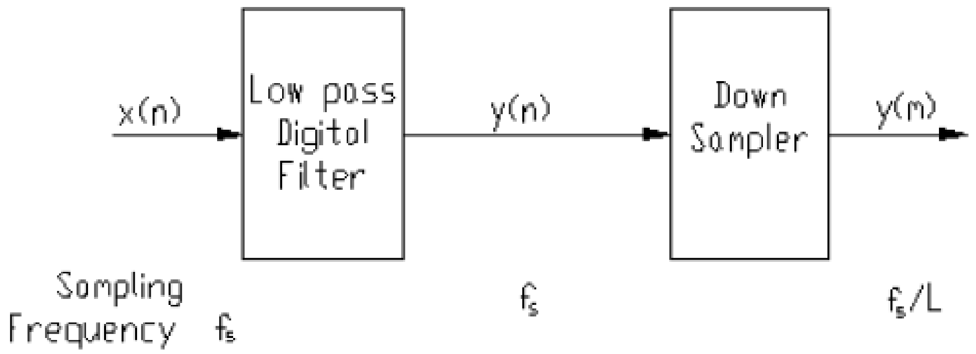


Figure 5.8: The decimation process

$$y(m) = y(nL) = \sum_{K=0} h(K)x(nL-K):$$

Where n = 0,1,2,…..
L=decimation factor
N=filter size

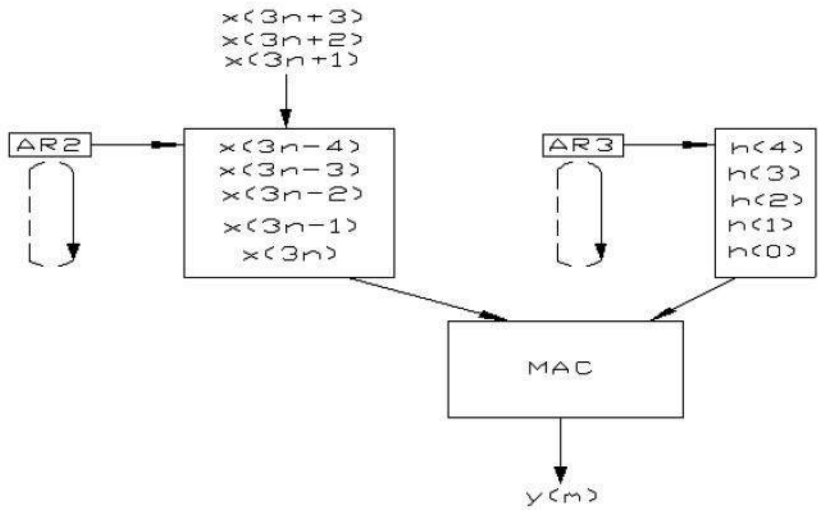


Figure 5.9: Implementation of decimation filter

<u>Implementation of decimation filter</u>

It implements the following equation:
y(m) = h(4)x(3n-4) + h(3)x(3n-3) + h(2)x(3n-2) + h(1)x(3n-1) + h(0)x(3n) followed by the equation
y(m+1) = h(4)x(3n-1) + h(3)x(3n) + h(2)x(3n+1) + h(1)x(3n+2) + h(0)x(3n+3)
and so on for a decimation factor of 3 and a filter length of 5.

```
                .mmregs
                    .def _c_int00

                    .sect "samples"
InSamples        .include "data_in.dat"          ; Allocate space for x(n)s
OutSamples       .bss y,80,1                 ; Allocate space for y(n)s
SampleCnt            .set 240                    ; Number of samples to decimate

                    .sect "FirCoeff"          ; Filter coeff (sequential)
FirCoeff        .include "coeff_dec.dat“
Nm1             .set 4                        ; Number of filter taps - 1

                      .bss CoefBuf, 5, 1      ; Memory for coeff circular buffer
```

```
                 .bss SampleBuf, 5, 1 ; Memory for sample circular buffer
                .text
_c_int00:
                STM #OutSamples, AR6      ; Clear output sample buffer
                RPT #SampleCnt
                ST #0, *AR6+

                STM #InSamples, AR5        ; AR5 points to InSamples buffer
                STM #OutSamples, AR6      ; AR6 points to OutSample buffer
                STM #SampleCnt, AR4     ; AR4 = Number of samples to filter
                CALL dec_init            ; Init for filter calculations
loop:
           CALL dec_filter              ; Call Filter Routine
                STH A,1,*AR6+            ; Store filtered sample (integer)
                BANZ loop,*AR4-          ; Repeat till all samples filtered
                nop
                nop
                nop
```

Decimation Filter Initialization Routine

This routine sets AR2 as the pointer for the sample circular buffer, and AR3 as the pointer for coefficient circular buffer.
BK = Number of filter taps. ; AR0 = 1 = circular buffer pointer increment.

```
dec_init :
          ST #CoefBuf,AR3                    ; AR3 is the CB Coeff Pointer
          ST #SampleBuf,AR2          ; AR2 is the CB sample pointer
          STM #Nm1,BK                        ; BK = number of filter taps
          RPT #Nm1
          MVPD #FirCoeff, *AR3+%             ; Place coeff in circular buffer
          RPT #Nm1                    ; Clear circular sample buffer
          ST #0h,*AR2+%
          STM #1,AR0;                ; AR0 = 1 = CB pointer increment
           RET                               ; Return
           nop
           nop
           nop
```

FIR Filter Routine

Enter with A = x(n), AR2 pointing to the circular sample buffer, and AR3 to the circular coeff buffer. AR0 = 1.
Exit with A = y (n) as q15 number.

```
dec_filter :
          LD *AR5+,A                          ; Place next 3 input samples
          STL A, *AR2+0%                ; into the signal buffer
          LD *AR5+,A
          STL A, *AR2+0%
          LD *AR5+,A
          STL A, *AR2+0%
          RPTZ A, #Nm1                  ; A = 0
          MAC *AR3+0%,*AR2+0%,A         ; A = filtered signal
          RET                               ; Return
          nop
          nop
          nop
         .end
```

Problems:

1. What values are represented by the 16-bit fixed point number N=4000h in Q15 & Q7 notations?

Solution:

Q15 notation: **0.100 0000 0000 0000 (N=0.5)**

Q7 notation: **0100 0000 0.000 0000 (N=+128)**

Implementation of FFT algorithms

Introduction: The N point Discrete Fourier Transform (DFT) of x(n) is a discrete signal of length N is given by eq(6.1)

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \quad ; \qquad k = 0 \ldots N-1 \qquad (6.1)$$

$W_N^{kn} = e^{-j2\pi kn/N}$ is the twiddle factor

The Inverse DFT (IDFT) is given by eq(2)

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-kn}; \quad n = 0 \ldots N-1 \qquad (6.2)$$

By referring to eq (6.1) and eq (6.2), the difference between DFT & IDFT are seen to be the sign of the argument for the exponent and multiplication factor, 1/N. The computational complexity in computing DFT / I DFT is thus same (except for the additional multiplication factor in IDFT). The computational complexity in computing each X(k) and all the x(k) is shown in table 6.1.

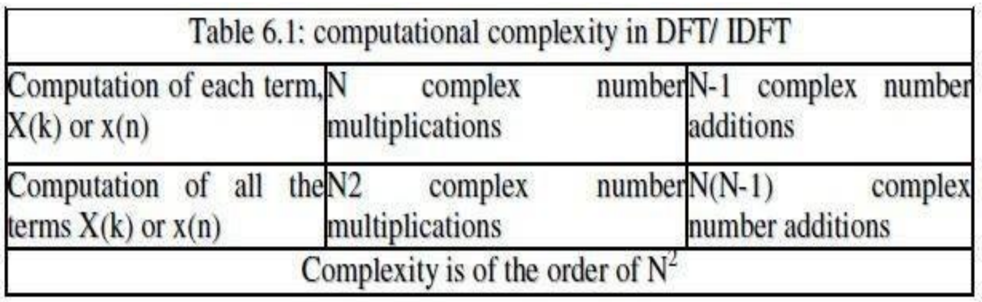


Table 6.1: computational complexity in DFT/ IDFT

Computation of each term, X(k) or x(n)	N complex number multiplications	N-1 complex number additions
Computation of all the terms X(k) or x(n)	N2 complex number multiplications	N(N-1) complex number additions
Complexity is of the order of N^2		

In a typical Signal Processing System, shown in fig 6.1 signal is processed using DSP in the DFT domain. After processing, IDFT is taken to get the signal in its original domain. Though certain amount of time is required for forward and inverse transform, it is because of the advantages of transformed domain manipulation, the signal processing is carried out in DFT domain. The transformed domain manipulations are sometimes simpler. They are also more useful and powerful than time domain manipulation. For example, convolution in time domain requires one of the signals to be folded, shifted and multiplied by another signal, cumulatively. Instead, when the signals to be convolved are transformed to DFT domain, the two DFT are multiplied and inverse transform is taken. Thus, it simplifies the process of convolution.

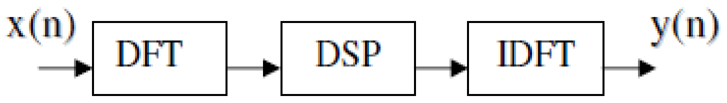


Fig 6.1: DSP System

An FFT Algorithm for DFT Computation: As DFT / IDFT are part of signal processing system, there is a need for fast computation of DFT / IDFT. There are algorithms available for fast computation of DFT/ IDFT. There are referred to as Fast Fourier Transform (FFT) algorithms. There are two FFT algorithms: Decimation-In-Time
FFT (DITFFT) and Decimation-In-Frequency FFT (DIFFFT). The computational complexity of both the algorithms are of the order of log2(N). From the hardware / software implementation viewpoint the algorithms have similar structure throughout the
computation. In-place computation is possible reducing the requirement of large memory locations. The features of FFT are tabulated in the table 6.2.

Table 6.2: Features of FFT

Features	DITFFT	DIFFFT
Sequence which is decimated by factor 2	Time domain sequence	DFT sequence
Input sequence	Bit reversed order	Proper order
Output sequence	Proper order	Bit reversed order

Consider an example of computation of 2 point DFT. The signal flow graph of 2 point DITFFT Computation is shown in fig 6.2. The input / output relations is as in eq (6.3) which are arrived at from eq(6.1).

$$\begin{aligned} X(0) &= x(0)W_2^0 + x(1)W_2^0 = x(0) + x(1) \\ X(1) &= x(0)W_2^0 + x(1)W_2^1 = x(0) - x(1) \end{aligned} \qquad (6.3)$$

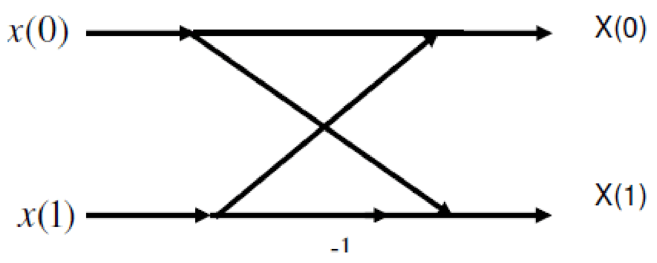


Fig 6.2: Signal Flow graph for N=2

Similarly, the Butterfly structure in general for DITFFT algorithm is shown in fig. 6.3. The signal flow graph for N=8 point DITFFT is shown in fig. 4. The relation between input and output of any Butterfly structure is shown in eq (6.4) and eq(6.5).

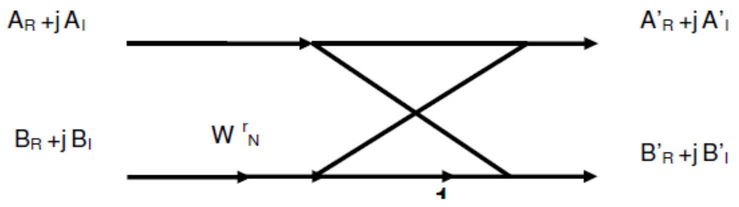


Fig 6. 3: Butterfly structure for N point DITFFT Computation

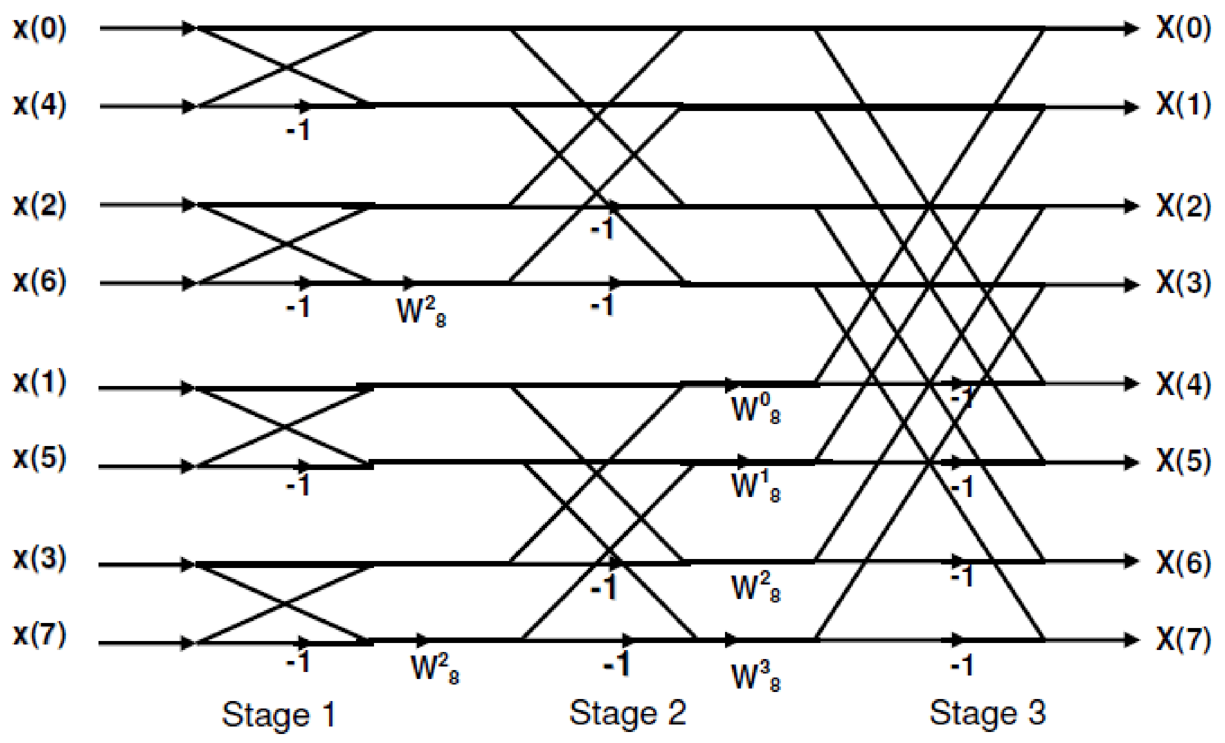


Fig 6.4. Signal flow graph of 8 point DITFFT Computation

$$A'_R + jA'_I = A_R + jA_I + (B_R + jB_I)(W^r_R + jW^r_I) \qquad (6.4)$$

$$B'_R + jB'_I = A_R + jA_I - (B_R + jB_I)(W^r_R + jW^r_I) \qquad (6.5)$$

Separating the real and imaginary parts, the four equations to be realized in implementation of DITFFT Butterfly structure are as in eq(6.6).

$$\left\{\begin{array}{l} A'_R = A_R + B_R W^r_R - B_I W^r_I \\ A'_I = A_I + B_I W^r_R + B_R W^r_I \\ B'_R = A_R - B_R W^r_R + B_I W^r_I \\ B'_I = A_I - B_I W^r_R - B_R W^r_I \end{array}\right\} \qquad (6.6)$$

Observe that with N=2^M, the number of stages in signal flow graph=M, number of multiplications = (N/2)log2(N) and number of additions = (N/2)log2(N). Number of Butterfly Structures per stage = N/2. They are identical and hence in-place computation is possible. Also reusability of hardware designed for implementing Butterfly structure is

possible. However in case FFT is to be computed for a input sequence of length other than 2^M the sequence is extended to N=2^M by appending additional zeros. The process will not alter the information content of the signal. It improves frequency resolution. To make the point clear, consider a sequence whose spectrum is shown in fig. 6.5.

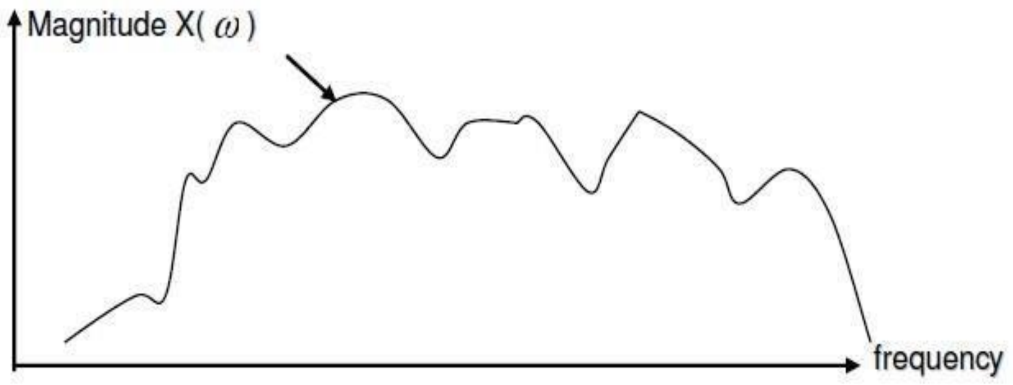


Fig 6.5: Spectrum of x(n)

The spectrum is sampled to get DFT with only N=10. The same is shown in fig 6.

The variations in the spectrum are not traced or caught by the DFT with N=10. For example, dip in the spectrum near sample no. 2, between sample no.7 & 8 are not represented in DFT. By increasing N=16, the DFT plot is shown in fig. 6.7. As depicted in fig 6.7, the approximation to the spectrum with N=16 is better than with N=10. Thus, increasing N to a suitable value as required by an algorithm improves frequency resolution.

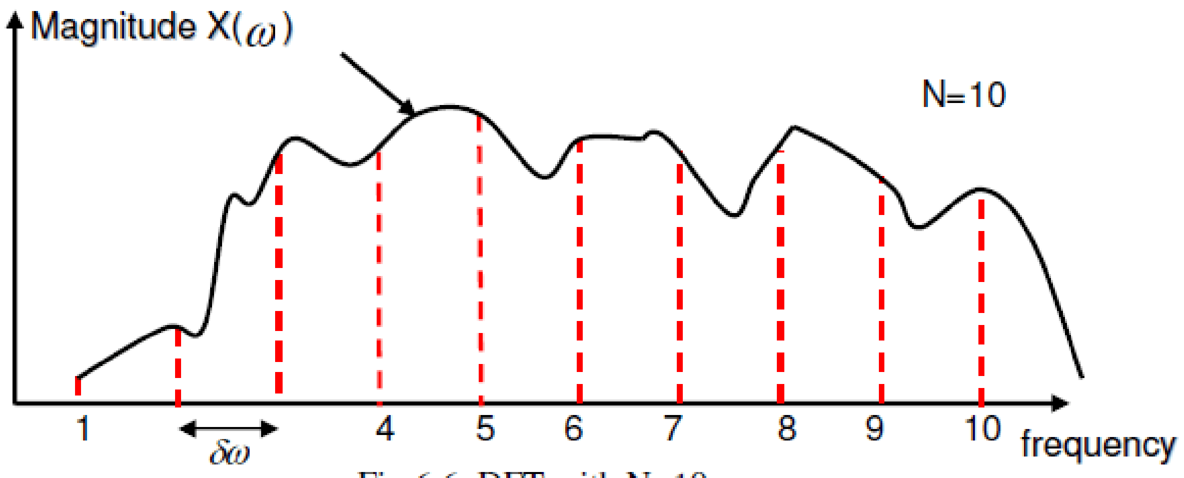


Fig 6.6: DFT with N=10

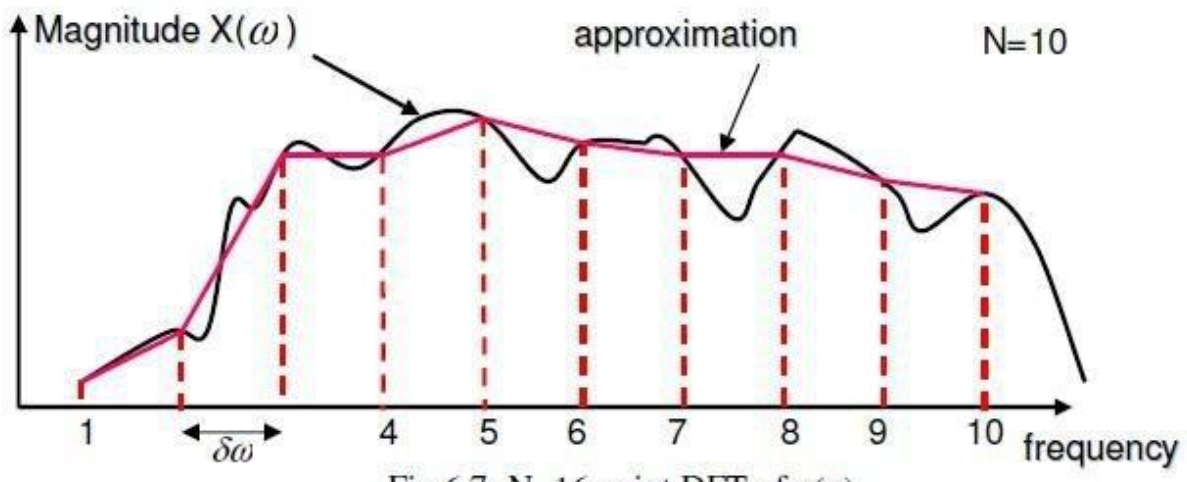
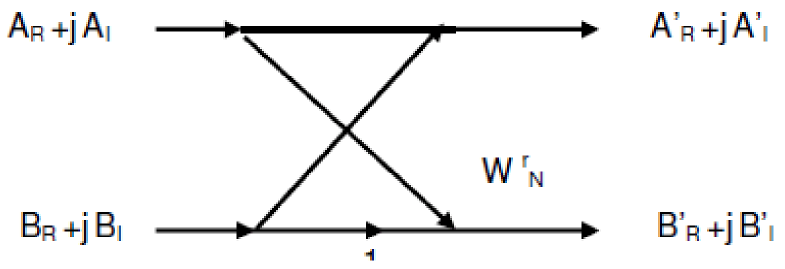


Fig 6.7: N=16 point DFT of x(n)

Problem P6.1: What minimum size FFT must be used to compute a DFT of 40 points? What must be done to samples before the chosen FFT is applied? What is the frequency resolution achieved?

Solution:
Minimum size FFT for a 40 point sequence is 64 point FFT. Sequence is extended to 64 by appending additional 24 zeros. The process improves frequency resolution from

$$\delta\omega = 2\pi/40 \text{ to } \delta\omega = 2\pi/64 \quad \text{(P6.1)}$$



Overflow and Scaling: In any processing system, number of bits per data in signal processing is fixed and it is limited by the DSP processor used. Limited number of bits leads to overflow and it results in erroneous answer. InQ15 notation, the range of numbers that can be represented is -1 to 1. If the value of a number exceeds these limits, there will be underflow / overflow. Data is scaled down to avoid overflow.

However, it is an additional multiplication operation. Scaling operation is simplified by selecting scaling factor of 2^-n. And scaling can be achieved by right shifting data by n bits. Scaling factor is defined as the reciprocal of maximum possible number in the operation. Multiply all the numbers at the beginning of the operation by scaling factor so that the maximum number to be processed is not more than 1. In the case of DITFFT computation, consider for example,

$$A'_I = A_I + B_I W_R^r + B_R W_I^r$$
$$= A_I + B_I \cos\theta + B_R \sin\theta \qquad (6.7)$$
$$\text{where } \theta = 2\pi kn/N$$

To find the maximum possible value for LHS term, Differentiate and equate to zero

$$\frac{\partial A_I}{\partial\theta} = -B_I \sin\theta + B_R \cos\theta = 0$$
$$\Rightarrow B_I \sin\theta = B_R \cos\theta \qquad (6.8)$$
$$\Rightarrow \tan\theta = B_R/B_I$$
$$\therefore \sin\theta = \frac{B_R}{\sqrt{B_R^2 + B_I^2}} \qquad \text{Similarly,} \quad \cos\theta = \frac{B_I}{\sqrt{B_R^2 + B_I^2}}$$

Substituting them in eq(6.7),

$$A_I' = A_I + \sqrt{B_R^2 + B_I^2}$$
$$A'_{I,\max} = 1 + \sqrt{2} = 2.414$$

Thus scaling factor is 1/2.414=0.414. A scaling factor of 0.4 is taken so that it can be implemented by shifting the data by 2 positions to the right. The symbolic representation
of Butterfly Structure is shown in fig. 6.8. The complete signal flow graph with scaling factor is shown in fig. 6.9.

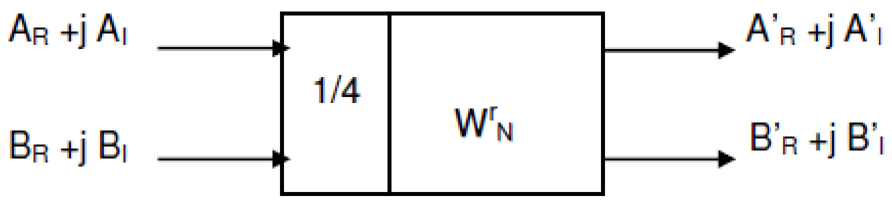


Fig 6.8: Symbolic representation of Butterfly structure with scaling factor

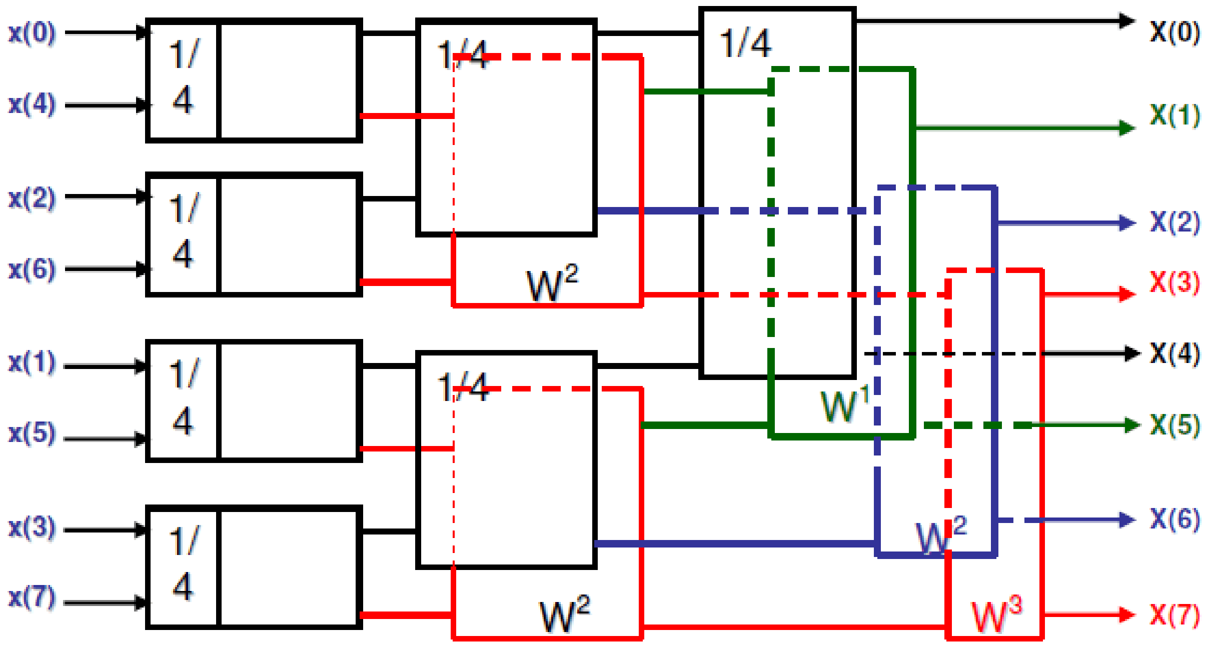


Fig 6.9: Signal flow graph with Scaling

Bit-Reversed Index Generation: As noted in table 6.2, DITFFT algorithm requires input in bit reversed order. The input sequence can be arranged in bit reverse order by reverse carry add operation. Add half of DFT size (=N/2) to the present bit reversed ndex to get next bit reverse index. And employ reverse carry propagation while adding bits from left to right. The original index and bit reverse index for N=8 is listed in table 6.3

Table 6.3: Original & bit reverse indices

Original Index	Bit Reversed Index
000	000
001	100
010	010
011	110
100	001
101	101
110	011
111	111

Consider an example of computing bit reverse index. The present bit reversed index be 110. The next bit reversed index is

```
1 1 0
1 0 0  (N/2=4)
-------
0 0 1
```

There are addressing modes in DSP supporting bit reverse indexing, which do the computation of reverse index.

Implementation of FFT on TMS32OC54xx: The main program flow for the implementation of DITFFT is shown in fig. 6.10. The subroutines used are _clear to clear all the memory locations reserved for the results. _bitrev stores the data sequence x (n) in bit reverse order. _butterfly computes the four equations of computing real and imaginary parts of butterfly structure. _spectrum computes the spectrum of x (n). The Butterfly subroutine is invoked 12 times and the other subroutines are invoked only once.

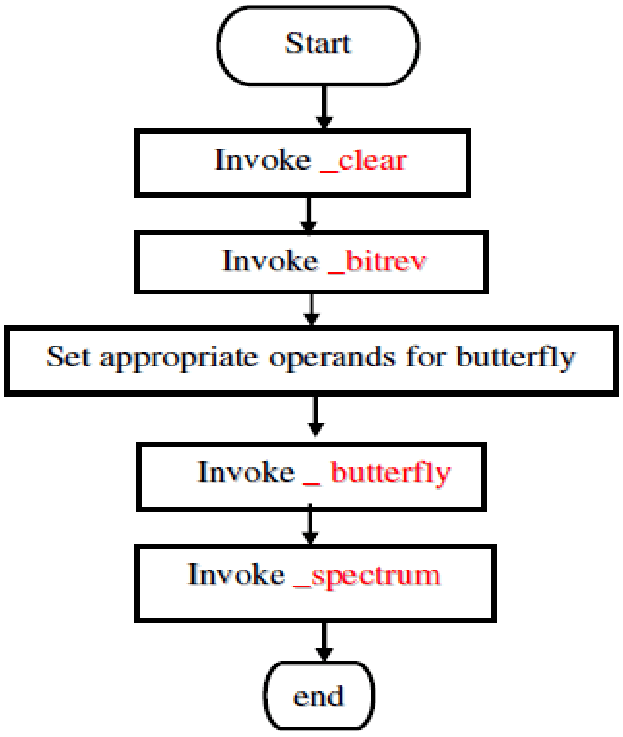


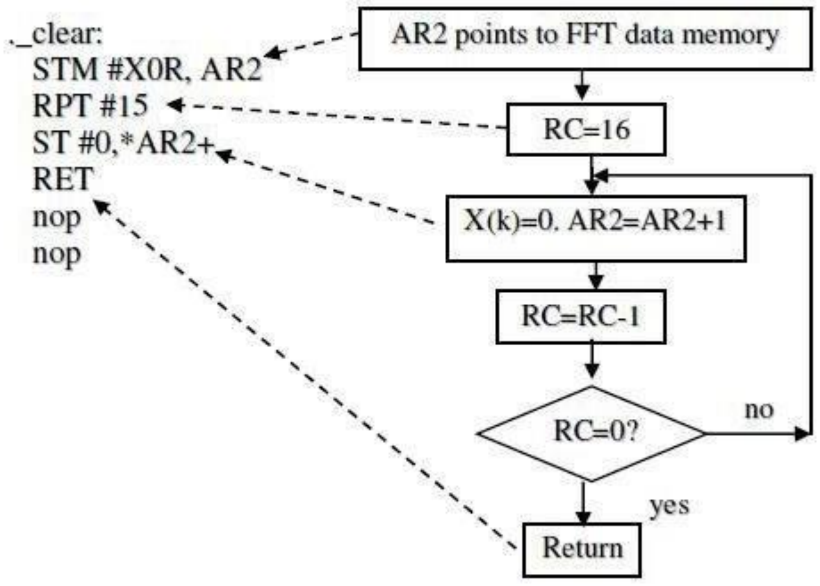
Fig. 6.10: Main Program Flow

The program is as follows

```
 .mmregs
 .def _c_int00
 .data
 ; Reserve 8 locations for x(n)
 ;x(n)                        Q15 notation     decimal value

xn0    .word  0            ; 0h                 0.0
xn1    . word 16384        ; 4000h              0.5
xn2    .word 23170         ; 5A82h              0.707
xn3    . word - 24576      ; E000h             -0.25
xn4    .word 12345   ; 3039h             0.3767
xn5    .word 30000         ; 7530h              0.9155
xn6    .word 10940   ; 2ABCh             0.334
xn7    .word 12345         ; 3039h              0.3767

; Reserve 16 locations for X(k)
X0R    .word 0             ;real part of X(0) =0
X0Im   .word 0       ;imaginary part of X(0) =0
X1R   .word 0
X1Im  .word 0
X2R    .word 0
X2Im  .word 0
X3R   .word 0
X3Im  .word 0
X4R   .word 0
X4Im  .word 0
X5R   . word 0
X5Im  .word 0
X6R   .word 0
X6Im  .word 0
X7R   .word 0
X7Im  .word 0

; 8 locations for W08 to W38, twiddle factors
W08R          .word  32767          ;cos(0)=1
W08Im         .word 0               ;-sin(0)=0
W18R          .word  23170          ;cos(pi/4)= 0.707
W18Im         .word -23170          ;-sin(pi/4)= -0.707
W28R          .word  0              ;cos(pi/2)= 0
W28Im         .word -32767          ;-sin(pi/2)= -1
W38R          .word -23170          ;cos(3pi/4)= -0.707
W38Im         .word -23170          ;-sin(3pi/4)= -0.707

; 8 locations for Spectrum
S0     .word  0            ;Frequency content at 0
S1     .word  0            ;Frequency content at fs/8
S2     .word  0            ;Frequency content at 2fs/8
S3     .word  0            ;Frequency content at 3fs/8
S4     .word  0            ;Frequency content at 4fs/
S5     .word  0            ;Frequency content at 5fs/8
S6     .word  0            ;Frequency content at 6fs/8
S7     .word  0            ;Frequency content at 7fs/8
```

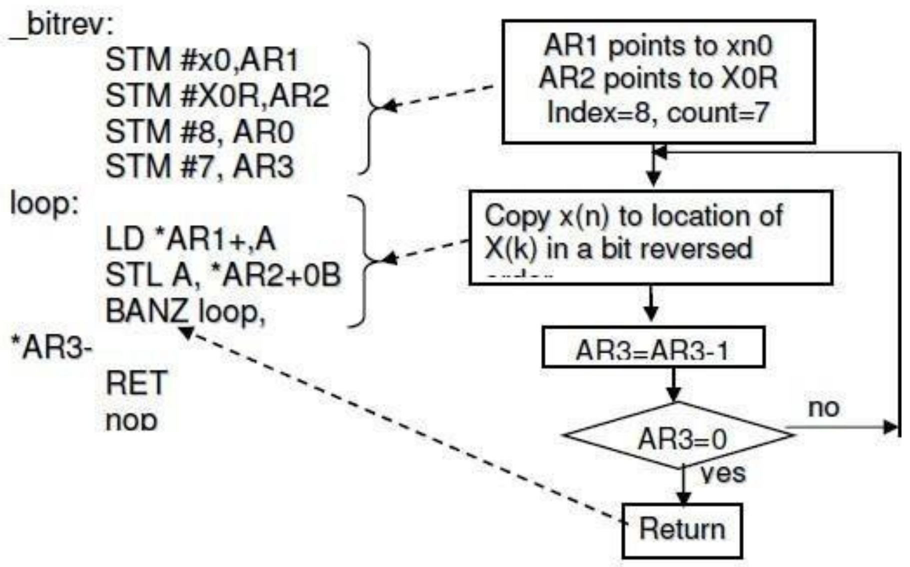
```
;temporary locations
TEMP1          .word 0
TEMP2          .word 0
;MAIN PROGRAM
. text
_c_int00:
          SSBX SXM    ; set sign extension mode bit of ST1
          CALL  _clear
          CALL  _bitrev
```

Clear subroutine is shown in fig. 6.11. Sixteen locations meant for final results are cleared. AR2 is used as pointer to the locations. Bit reverse subroutine is shown in fig. 6.12. Here, AR1 is used as pointer to x(n). AR2 is used as pointer to X(k) locations. AR0 is loaded with 8 and used in bit reverse addressing. Instead of N/2 =4, it is loaded with N=8 because each X(k) requires two locations, one for real part and the other for imaginary part. Thus, x(n) is stored in alternate locations, which are meant for real part of X(k). AR3 is used to keep track of number of transfers.



```
._clear:
  STM #X0R, AR2
  RPT #15
  ST #0,*AR2+
  RET
  nop
  nop
```

Fig. 6.11: Clear subroutine



```
_bitrev:
        STM #x0,AR1
        STM #X0R,AR2
        STM #8, AR0
        STM #7, AR3
loop:
        LD *AR1+,A
        STL A, *AR2+0B
        BANZ loop,
*AR3-
        RET
        nop
```

Fig. 6.12: Bit Reverse Subroutine

Butterfly subroutine is invoked 12 times. Part of the subroutine is shown in fig. 6.13. Real part and imaginary of A and B input data of butterfly structure is divided by 4 which
is the scaling factor. Real part of A data which is divided by 2 is stored in temp location. It is used
further in computation of eq (3) and eq (4) of butterfly. Division is carried out by shifting the data to the right by two places. AR5 points to real part of A input data, AR2 points to real part of B input data and AR3 points to real part of twiddle factor while
invoking the butterfly subroutine. After all the four equations are computed, the pointers
are in the same position as they were when the subroutine is invoked. Thus, the results
are stored such that in-place computation is achieved. Fig. 6.14 through 6.17 show the
butterfly subroutine for the computation of 4 equations.

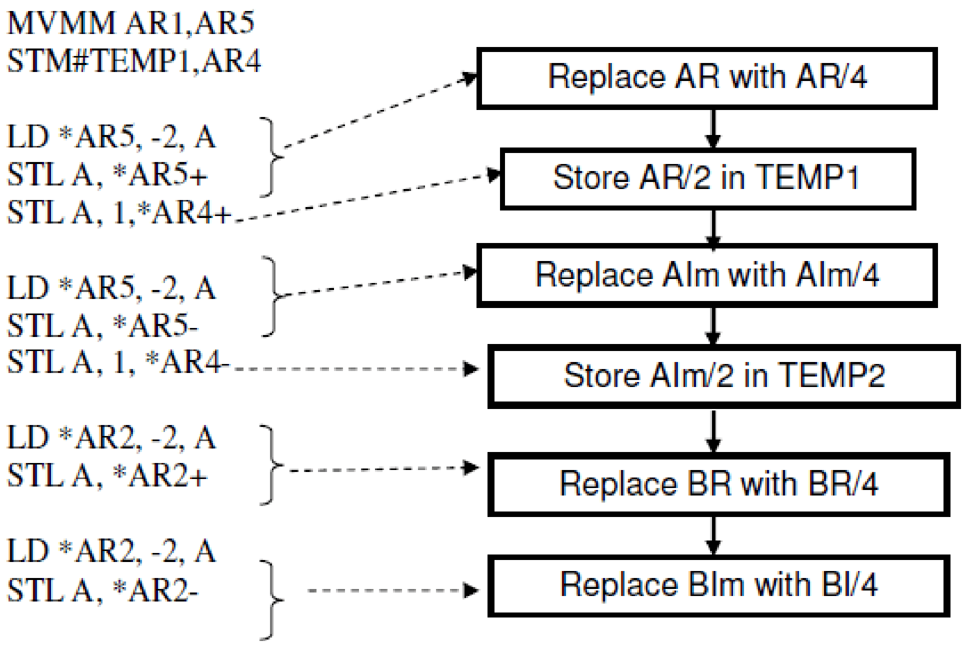


Fig. 6.13: Butterfly Subroutine

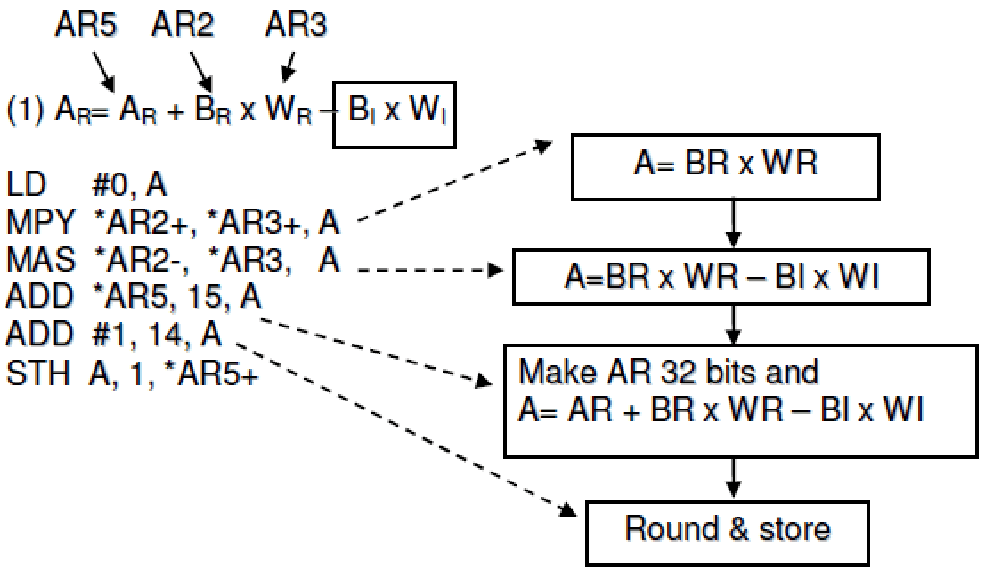
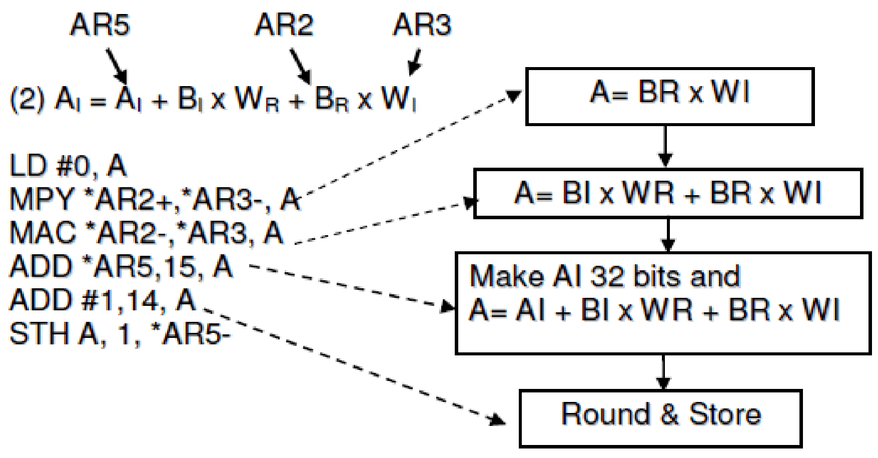


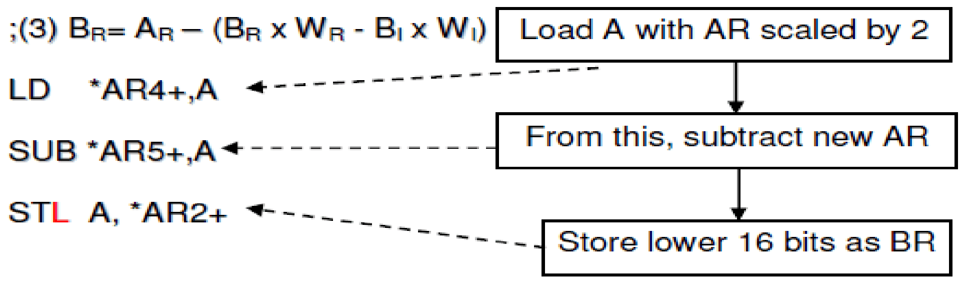
Fig. 6.14: Real part of A output of Butterfly



$(2)\ A_I = A_I + B_I \times W_R + B_R \times W_I$

```
LD #0, A
MPY *AR2+,*AR3-, A
MAC *AR2-,*AR3, A
ADD *AR5,15, A
ADD #1,14, A
STH A, 1, *AR5-
```

Fig. 6.15: Imaginary part of A output of Butterfly



$;(3)\ B_R = A_R - (B_R \times W_R - B_I \times W_I)$

```
LD   *AR4+,A
SUB *AR5+,A
STL  A, *AR2+
```

Load A with AR scaled by 2
From this, subtract new AR
Store lower 16 bits as BR

Fig. 6.16: Real part of B output of Butterfly

$;(4)\ B_I = A_I - (B_I \times W_R + B_R \times W_I)$

```
LD   *AR4 -, A
SUB *AR5-, A
STL  A, *AR2-

RET
nop
nop
```

Fig. 6.17: Imaginary part of B output of Butterfly

Figure 6.18 depicts the part of the main program that invokes butterfly subroutine by supplying appropriate inputs, A and B to the subroutine. The associated butterfly structure is also shown for quick reference. Figures 6.19 and 6.20 depict the main program for the computation of 2nd and 3rd stage of butterfly.

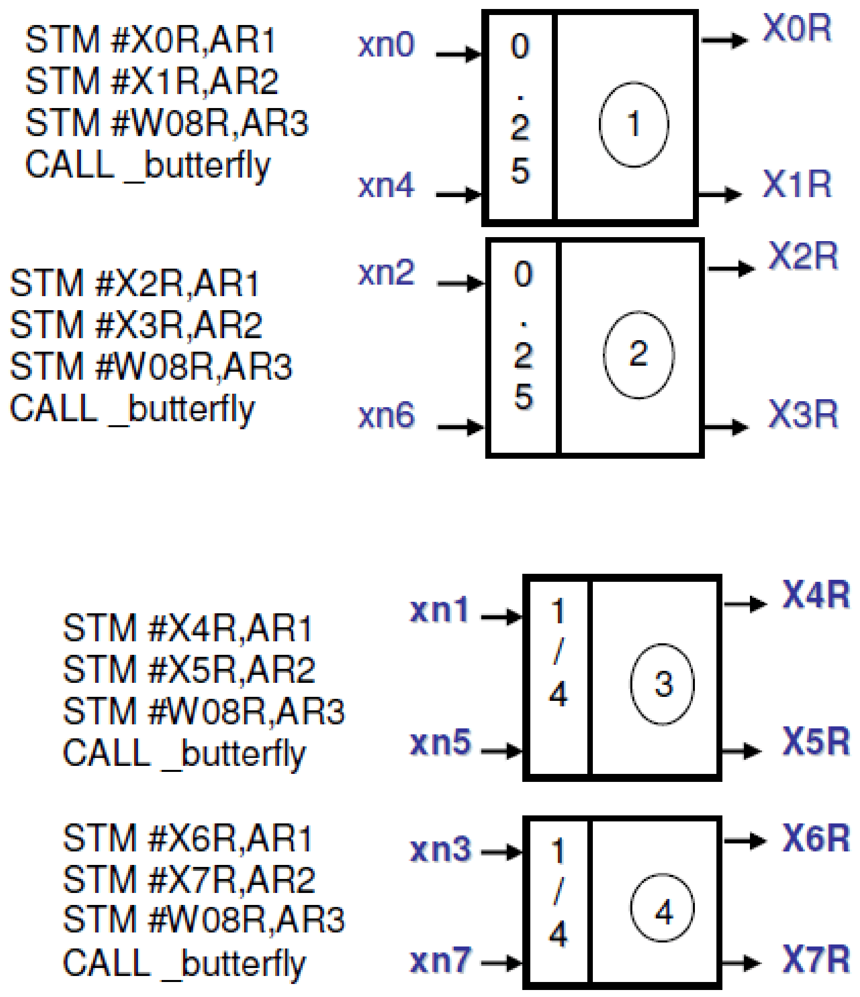
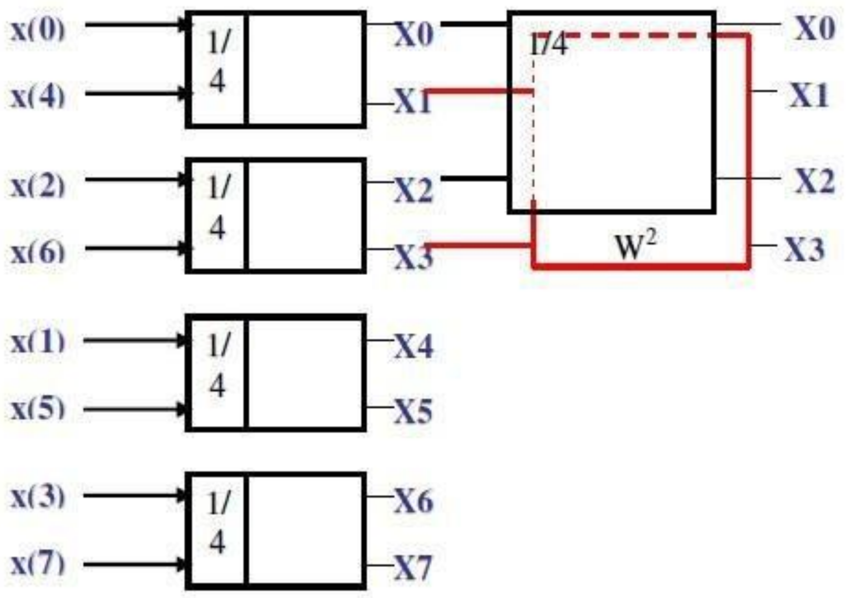
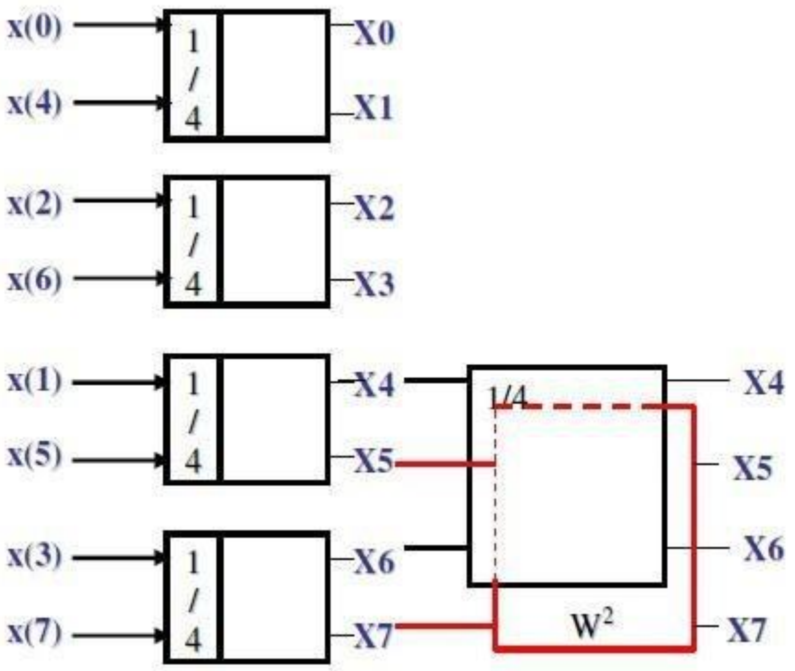


Fig. 6.18: First stage of Signal Flow graph of DITFFT



```
STM #X0R,AR1
STM #X2R,AR2
STM #W08R,AR3
CALL _butterfly
```

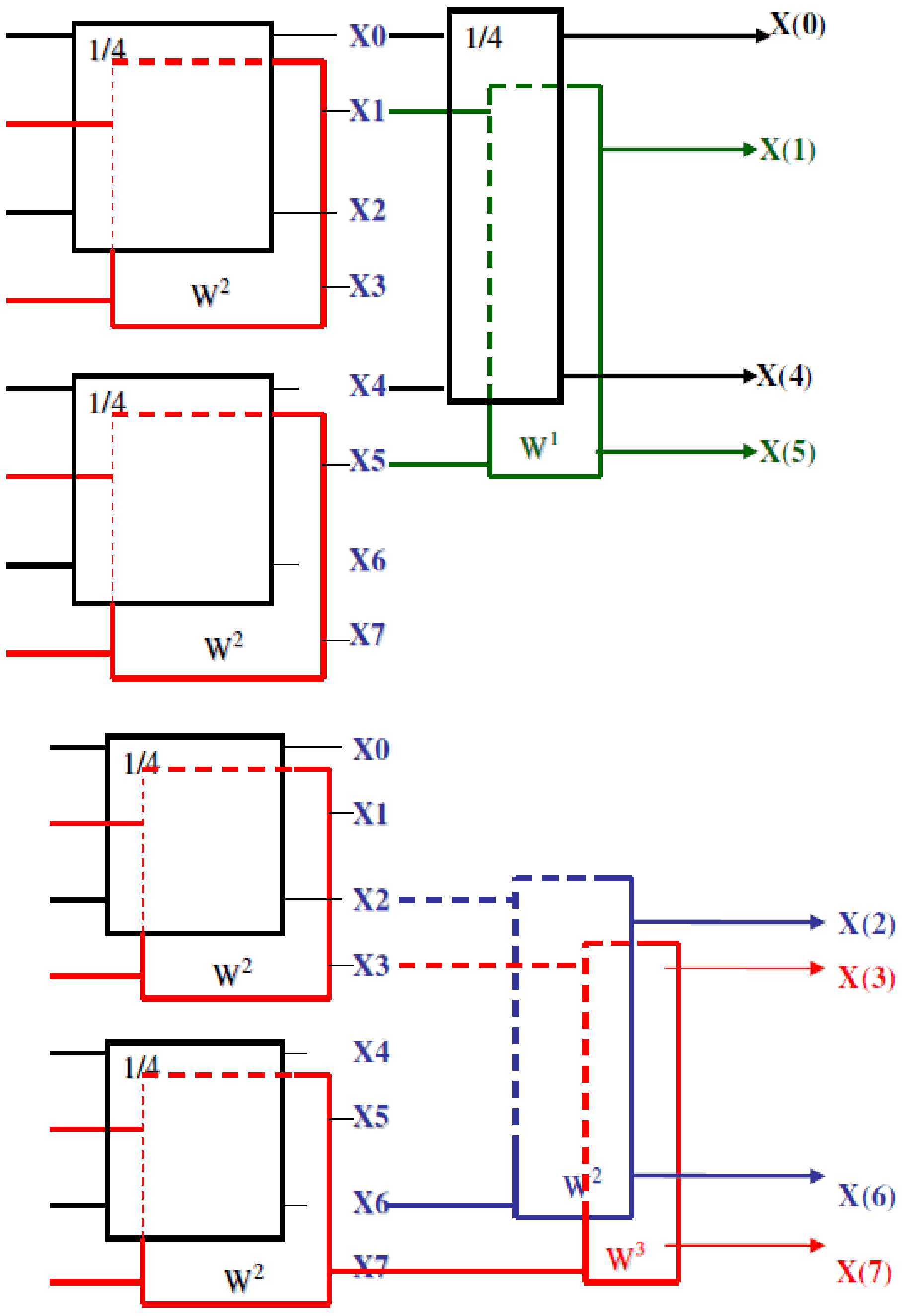
```
STM #X1R,AR1
STM #X3R,AR2
STM #W28R,AR3
CALL _butterfly
```



```
STM #X4R,AR1
STM #X6R,AR2
STM #W08R,AR3
CALL _butterfly
```

```
STM #X5R,AR1
STM #X7R,AR2
STM #W28R,AR3
CALL _butterfly
```

Fig. 6.19: Second stage of Signal Flow graph of DITFFT



```
STM #X0R,AR1
STM #X4R,AR2
STM #W08R,AR3
CALL _butterfly
```

```
STM #X1R,AR1
STM #X5R,AR2
STM #W18R,AR3
CALL _butterfly
```

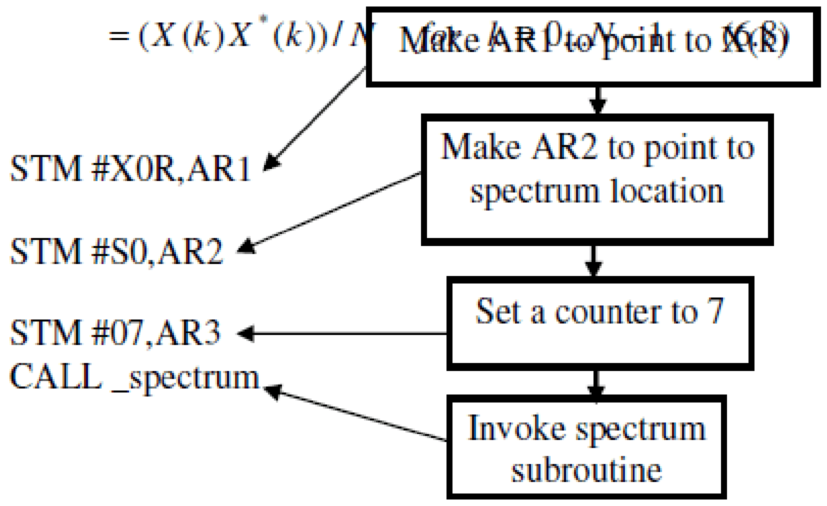
```
STM #X2R,AR1
STM #X6R,AR2
STM #W28R,AR3
CALL _butterfly
```

```
STM #X3R,AR1
STM #X7R,AR2
STM #W38R,AR3
CALL _butterfly
```

Fig. 6.20: Third stage of Signal Flow graph of DITFFT

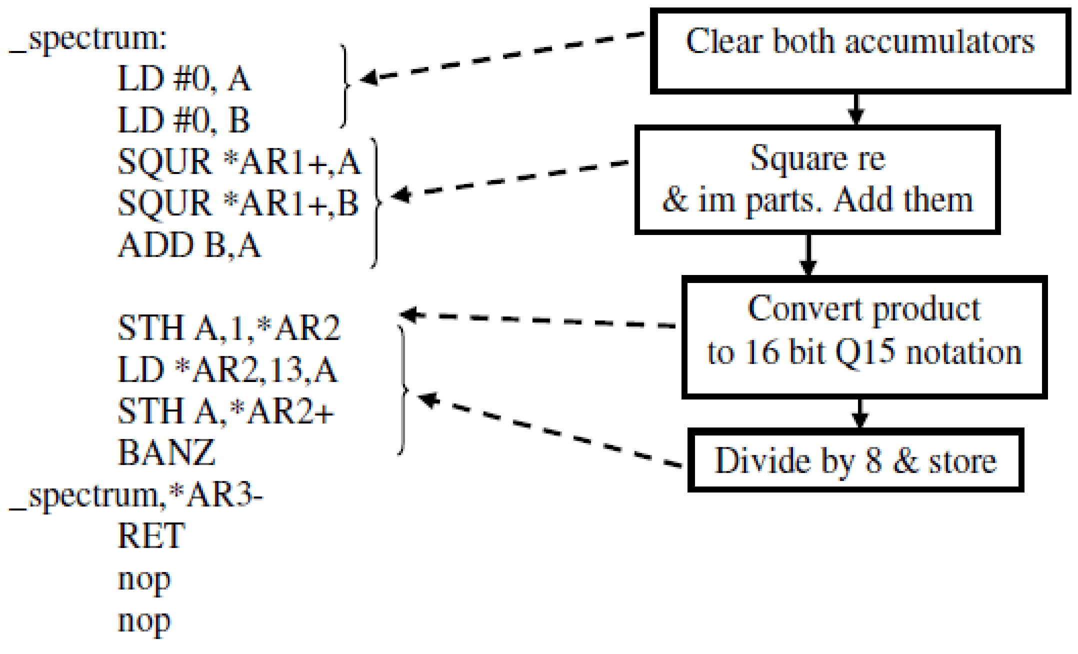
After the computation of X(k), spectrum is computed using the eq(6.8). The pointer AR1 is made to point to X(k). AR2 is made to point to location meant for spectrum. AR3 is loaded with keeps track of number of computation to be performed. The initialization of the pointer registers before invoking the spectrum subroutine is shown in fig. 6.21. The subroutine is shown in fig. 6.22. In the subroutine, square of real and imaginary parts are computed and they are added. The result is converted to Q15 notation and stored.

$$S(k) = (X^2(k))/N$$
$$= (X(k)X^*(k))/N \quad \text{for } k = 0,\dots,N-1 \qquad (6.8)$$


Make AR1 to point to X(k)
Make AR2 to point to spectrum location
Set a counter to 7
Invoke spectrum subroutine

```
STM #X0R,AR1
STM #S0,AR2
STM #07,AR3
CALL _spectrum
```

Fig. 6.21: Initialization for Spectrum Computation


Clear both accumulators
Square re & im parts. Add them
Convert product to 16 bit Q15 notation
Divide by 8 & store

```
_spectrum:
        LD #0, A
        LD #0, B
        SQUR *AR1+,A
        SQUR *AR1+,B
        ADD B,A

        STH A,1,*AR2
        LD *AR2,13,A
        STH A,*AR2+
        BANZ
_spectrum,*AR3-
        RET
        nop
        nop
```

Fig. 6.22: Subroutine for Spectrum Computation

Problems:

1. Derive equations to implement a Butterfly encountered in a DIFFFT implementation.

Solution:

Butterfly structure for DIFFFT:

The input / output relations are

$$A'_R + jA'_I = A_R + jA_I + B_R + jB_I$$
$$B'_R + jB'_I = (A_R + jA_I - (B_R + jB_I))(W^r_R + jW^r_I)$$

Separating the real and imaginary parts,

$$\therefore\ A'_R = A_R + B_R \quad \& \qquad A'_I = A_I + B_I$$
$$B'_R = (A_R - B_R)W^r_R - (A_I - B_I)W^r_I$$
$$B'_I = (A_R - B_R)W^r_I + (A_I - B_I)W^r_R$$

2. How many add/subtract and multiply operations are needed to implement a general butterfly of DITFFT?

Solution:

Referring to 4 equations required in implementing DITFFT Butterfly structure, Add//subttrractt operations 06 and Multiply operations 04

3. Derive the optimum scaling factor for the DIFFFT Butterfly structure.

Solution: The four equations of Butterfly structure are

Differentiating 4th relation and setting it to zero, (any equation may be considered)

$$\frac{\partial B'_I}{\partial \theta} = (A_R - B_R)\cos\theta - (A_I - B_I)\sin\theta = 0$$
$$\Rightarrow (A_R - B_R)\cos\theta = (A_I - B_I)\sin\theta$$
$$\therefore\ \tan\theta = \frac{A_R - B_R}{A_I - B_I} \qquad \text{P6.5.2}$$

$$A'_R = A_R + B_R$$
$$A'_I = A_I + B_I$$
$$B'_R = (A_R - B_R)W^r_R - (A_I - B_I)W^r_I$$
$$B'_I = (A_R - B_R)W^r_I + (A_I - B_I)W^r_R \qquad \text{P6.5.1}$$

$$\sin\theta = \frac{(A_R - B_R)}{\sqrt{(A_R - B_R)^2 + (A_I - B_I)^2}}$$
$$\& \quad \cos\theta = \frac{(A_I - B_I)}{\sqrt{(A_R - B_R)^2 + (A_I - B_I)^2}}$$

$$\therefore\ B'_{I,\max} = \sqrt{(A_R - B_R) + (A_I - B_I)}$$
$$= \sqrt{2} \qquad \text{P6.5.3}$$

Thus scaling factor is 0.707. To achieve multiplication by right shift, it is chosen as 0.5.