# INTEL 8086 MICROPROCESSOR BASED DIGITAL FILTER REALIZATION

A DISSERTATION

*submitted in partial fulfilment of the requirements for the award of the Degree*

of

MASTER OF ENGINEERING

*in*

ELECTRICAL ENGINEERING

(System Engineering & Operational Research)

*by*

SAROJ AMBARDAR

DEPARTMENT OF ELECTRICAL ENGINEERING
UNIVERSITY OF ROORKEE
ROORKEE U.P. (INDIA)

August, 1982

## CERTIFICATE

Certified that the dissertation entitled "INTEL 8086 MICROPROCESSOR BASED DIGITAL FILTER REALIZATION" which is being submitted by Ms Saroj Ambardar in partial fulfilment for the award of the degree of MASTER OF ENGINEERING in ELECTRICAL ENGINEERING (System Engineering & Operational Research) of the University of Roorkee, Roorkee, is a record of student's own work carried out by her under my supervision and guidance. The matter embodied in this dissertation has not been submitted for the award of any other degree or diploma.

This is further to certify that she has worked for a period of about 7 months from Jan. 1982 to August 1982 for preparing this dissertation at this University.

[illegible] 16.8.1982

(M. K. VASANTA)
Reader
Electrical Engineering Department
University of Roorkee
Roorkee

Dated August 16, 1982

## A C K N O W L E D G E M E N T S

I wish to express my indebtedness and grateful thanks to Shri M. K. Vasanta, of the Department of Electrical Engineering for the invaluable guidance provided by him in carrying out the work contained in these pages. His guidance has been a constant source of encouragement and inspiration throughout the course of this study.

I also thank all those who helped me directly or indirectly in preparing this dissertation.

Saroj Ambardar

(SAROJ AMBARDAR)

## A B S T R A C T

Digital filtering is a major subdivision of Digital signal processing. The practical realizations of a Digital filter have been discussed in this dissertation. To avoid coefficient sensitivity problems, the Z-Transfer function of a Digital filter is implemented as a cascaded or parallel combination of second order modules. Each module in itself can be one of the four Direct structures.

The significant break-through in the area of IC technology have opened up new options for the implementation of Digital filters. The present day research is centred around the microprocessor based design of a Digital filter. Digital filters are now implemented making use of 16-bit word length microprocessors. Intel 8086 has been considered in this dissertation. The Assembly language of 8086 is used to implement all the Digital filter modules. The software programs are given for K'th order Digital filter using N second order modules in cascade. It has also been shown that the same sub-routines developed for second order modules can be used for parallel structure implementation.

# CONTENTS

# CHAPTER - 0

# I N T R O D U C T I O N

## 0.1 HISTORICAL INTRODUCTION

As man becomes more certain of his control of physical things, an ever more important part of his work is the manipulation of symbols he uses to describe and control these physical things. The field of waveform manipulation or signal processing as in radio, radar, sonar, seismology etc., is one of the keystones of science and technology. The techniques and applications of this field are as old as Newton and Gauss and as new as digital computer and integrated circuits.

During the decade of 1960-70, it became practical to represent information-bearing waveforms digitally and to do signal processing on the digital representation of the waveform. The availability of high speed digital computers fostered the development of increasingly complex and sophisticated signal processing algorithms. The significant break through in the area of integrated circuit technology promise economical implementations of very complex digital signal processing systems.

Fig. (0.1) illustrates one view of how the field has emerged and spread out. Digital filtering is one of the major subdivisions of Digital signal processing. Digital filtering processing algorithms have been used primarily in

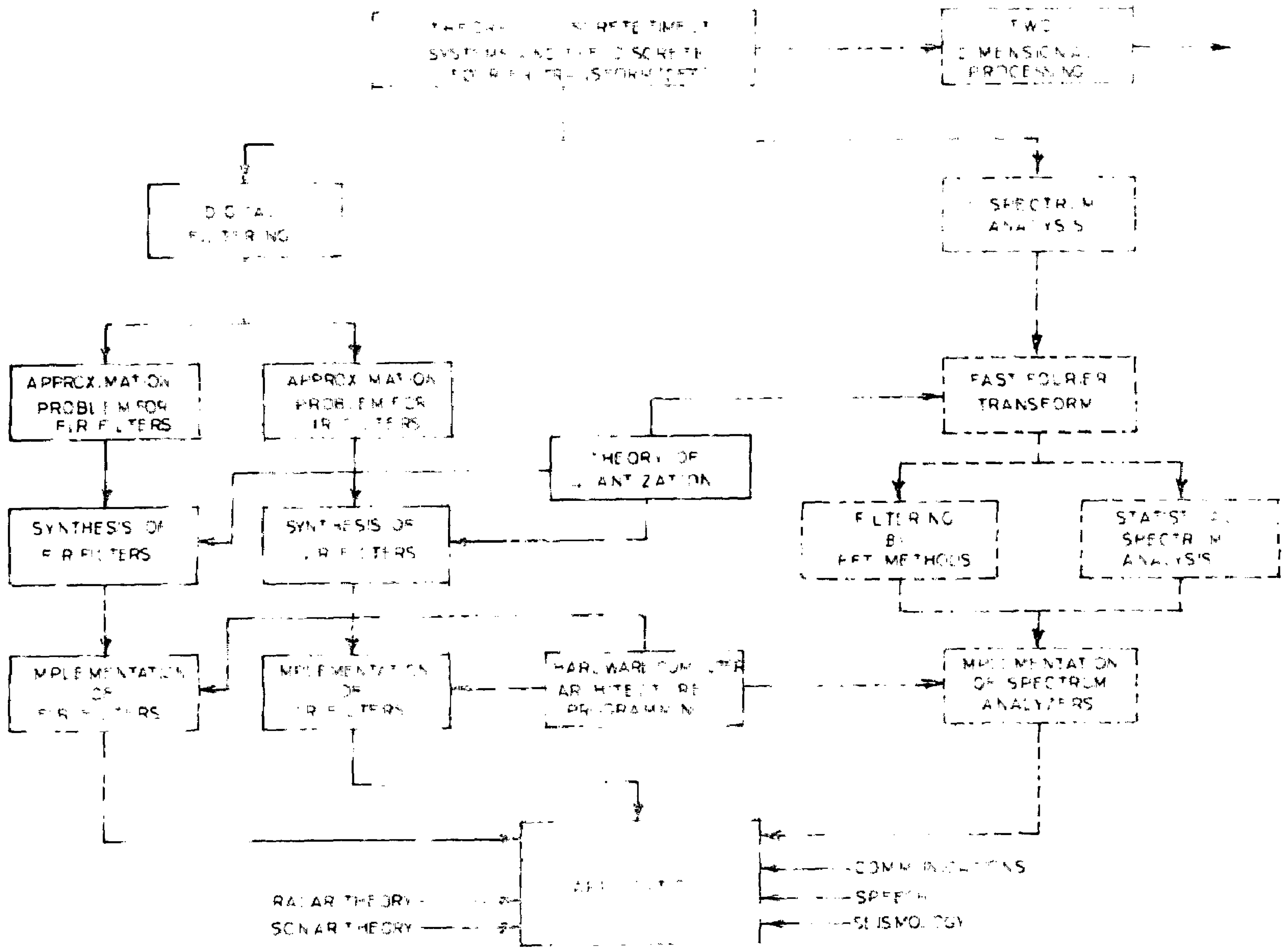

THEORY OF DISCRETE TIME LTI SYSTEMS AND THE DISCRETE FOURIER TRANSFORM (DFT)
TWO DIMENSIONAL PROCESSING
DIGITAL FILTERING
SPECTRUM ANALYSIS
APPROXIMATION PROBLEM FOR FIR FILTERS
APPROXIMATION PROBLEM FOR IIR FILTERS
FAST FOURIER TRANSFORM
THEORY OF QUANTIZATION
SYNTHESIS OF FIR FILTERS
SYNTHESIS OF IIR FILTERS
FILTERING BY FFT METHODS
STATISTICAL SPECTRUM ANALYSIS
IMPLEMENTATION OF FIR FILTERS
IMPLEMENTATION OF IIR FILTERS
HARDWARE COMPUTER ARCHITECTURE PROGRAMMING
IMPLEMENTATION OF SPECTRUM ANALYZERS
RADAR THEORY
SONAR THEORY
COMMUNICATIONS
SPEECH
SEISMOLOGY

FIG (2.1) OVER VIEW OF DIGITAL SIGNAL PROCESSING computer simulation, sampled data analysis and data reduction computations. With the increasing extensive application of digital processors to many systems, more and more importance is place on the development of mathematical tools for its analysis and design. The 'Z-transform' result in considerable simplification and understanding. The work of Kaiser, the first major contribution to the field of Digital signal processing, showed how much of the well-developed theory of the design of filters made of resistors, capacitors and inductors could be translated, with the aid of the Z-transform into straightforward digital filter techniques. At about the same time tremendous impetus was given to this emerging field by the Cooley-Tukey (1965) paper on a fast method of computing the discrete Fourier transform, a method that was subsequently popularized and extended via. many papers in the IEEE Transactions of the Group on Audio and Electroacoustics and other journals. At this time, the development of a formal and quite comprehensive theory of digital filters was well under way.

Perhaps the most interesting aspect of the development of the field of Digital signal processing is the changing relationship between the roles of FIR (finite impulse response) and IIR (infinite impulse response) digital filters. Initially Kaiser analysed FIR filters using window functions, which indicated that IIR filters were more efficient than FIR filters. However, Stockham's work on the FFT method of performing convolution, or more specifically FIR digital filtering, indicated that implementation of high-order FIR filters could be made extremely computationally efficient; thus comparisons between FIR and IIR filters are no longer strongly biased towards the latter. These results also inspired significant research for efficient design for FIR filters.

The Digital filter implementation till now was confined primarily to computer programs for simulation work or for processing relatively small amounts of data. However, with the rapid development of integrated circuit technology and especially the potential for large-scale-integration (LSI) of digital circuits made many of the Digital filters more attractive from the standpoint of cost, size and extreme reliability.

The design of high speed multipliers was of prime concern to many hardware and software implementations of signal processing algorithms. Standard TTL components gave sufficient speed to allow an effective filter to be implemented. Integrated circuits such as the Advanced Micro Devices AM25 LS14 2's complement multiplier was introduced specifically for signal processing applications. Peled and Liu used semiconductor memories for the purpose of fast multiplication, and resulted in significant saving in the cost and power consumption.

The Digit filter may be regarded as a special purpose computer built from an 'off-the shelf' logic family. The design depends upon the sampling rate and flexibility required. For simple sections a special purpose hardwork filter will be more efficient than a general purpose microprocessor. Instruction sets allow greater flexibility. The suitable microprocessor selected depends upon the particular application. Digital filters for different purposes have been implemented making use of 8 bit, 16 bit microprocessors. Much credit here goes to Nagle & Nelson. For most applications 16-bit accuracy is sufficient to avoid quantization problems with filters of moderate order ($n \leq 10$). It has been seen that with Intel 8086 (and other 16 bit microprocessors presently available) a significant improvement over the sampling rates can be achieved as compared with the previous generation of microprocessors, without significant increase in system cost.

## 0.2 OUTLINE OF THE PRESENT WORK

Chapter I discusses the general Transfer function and the various techniques for realizing a Digital filter.

Four Direct structures have been derived and it is in one of these structures that a Digital filter is usually implemented.

The salient features of Intel 8086 microprocessor which is selected for our purpose has been carried out in considerable detail in Chapter II. 8086 microprocessor is a totally new design, than any microprocessor previous offered by the Intel group and has a powerful set of instructions. Memory to memory string operations, hardwired multiplication and division, and flexible addressing modes are some of the significant operations.

Referring to Kaiser's work a second order structure is best suited for implementing higher order filters. Chapter III presents the implementation of a second order 1D module. A flow chart and a main program in 8086 Assembly language and the various subroutines with explanations is given. A K'th order cascaded filter has been discussed. Also a 4th order parallel filter has been given as an example. Use of 1D second order module subroutine is made in tho programs.

Chapter IV discusses the other types of structures used for realization of Digital filters. The sequence of study here is the derivation of the necessary equations, algorithm and the flow chart programs in 8086 Assembly language. A second order module is considered in each of the five cases viz. 2D, 3D, 4D, 1X, 2X.

The dissertation concludes with the summary of the work done alongwith suggestions for future study and development.

---

# CHAPTER - I

# TRANSFER FUNCTION AND REALIZATIONS OF A DIGITAL FILTER

## 1.1 INTRODUCTION

A major subdivision of Digital Signal Processing is Digital filtering - a computational algorithm performed on a sampled input signal resulting in a transformed output signal. Digital filtering processing algorithms have been used in computer simulation, sampled data analysis and data reduction computations. Kaiser [6] shows that the Z-Transform results in considerable simplification and understanding of problems associated with sampled data system. In this chapter the Transfer function of a Digital filter and its pictorial representations are discussed. Also, the various types of realizations of a Digital filter are studied.

## 1.2 Z-TRANSFER FUNCTION

In Linear continuous (Analog) filter theory, linear differential equation is one of the mathematical tools available to describe the Transfer function. Similarly, in linear Digital (Sampled) filter theory the linear difference equation is available as a mathematical tool for analysis and synthesis.

The linear difference equation [7] defines the sampled output pulse amplitude as a function of the present input pulse and any number of past input and output pulses. A general form of the difference equation is

$$Y(nT) = \sum_{i=0}^{N} A_i.X(nT - iT) - \sum_{i=1}^{N} B_i.Y(nT - iT) \quad \ldots \quad (1.1)$$

where X(nT) represents the present input samples and X(iT) are the past input samples. Similarly, Y(nT) and Y(iT) are present output samples and past output samples respectively. $A_i$ and $B_i$ coefficients are constants which determine the response of the filter.

The Z-Transform [1,3,5,7] of the above mentioned general difference equation (1.1) is :

$$Y(z) = X(z).\sum_{i=0}^{N} A_i.Z^{-i} - Y(z).\sum_{i=0}^{M} B_i.Z^{-i} \quad \ldots \quad (1.2)$$

This equation is interpreted as : the present output is dependent on the present and past inputs, each multiplied by the respective coefficients $A_i$ and the past output each multiplied by the respective coefficient $B_i$. Equation (1.2) is represented in the Transfer function form as :

$$D(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^{N} A_i.Z^{-i}}{1 + \sum_{i=1}^{M} B_i.Z^{-i}} \quad \ldots \quad (1.3)$$

Equation (1.3) is the Transfer function representation of a Recursive type of Digital filters. Appendix-I gives the classification of Digital filters.

In order to represent Digital filters in the form of a block diagram, the purpose of which is to graphically depict the way in which a particular system is realized, the terminology 4,8 shown in Fig. (1.1) is recommended.

A first order difference equation is :

$$Y(nT) = A_0.X(nT) + A_1.X(nT - T) - B_1.Y(nT - T) \quad \ldots \quad (1.4)$$

Z-transform of equation (1.4) is :

$$Y(z) = A_0.X(z) + A_1.Z^{-1}.X(z) - B_1.Z^{-1}.Y(z) \quad \ldots \quad (1.5)$$

hence,

$$D(z) = \frac{Y(z)}{X(z)} = \frac{A_0 + A_1.Z^{-1}}{1 + B_1.Z^{-1}} \quad \ldots \quad (1.6)$$

A second order difference equation is represented as :

$$Y(nT) = A_0.X(nT) + A_1.X(nT-T) + A_2.X(nT - 2T)$$
$$- B_1.Y(nT-T) - B_2.Y(nT - T) \quad \ldots \quad (1.7)$$

Z-transform of equation (1.7) is :

$$Y(z) = A_0.X(z) + A_1.Z^{-1}.X(z) + A_2.Z^{-2}.X(z)$$
$$- B_1.Z^{-1}.Y(z) - B_2.Z^{-2}.Y(z) \quad \ldots \quad (1.8)$$

hence,

$$D(z) = \frac{Y(z)}{X(z)} = \frac{A_0 + A_1.Z^{-1} + A_2.Z^{-2}}{1 + B_1.Z^{-1} + B_2.Z^{-2}} \quad \ldots \quad (1.9)$$

Fig. (1.2) represents the block diagram representation of the above derived general first and second order Digital transfer functions of equations (1.6) and (1.9) respectively.

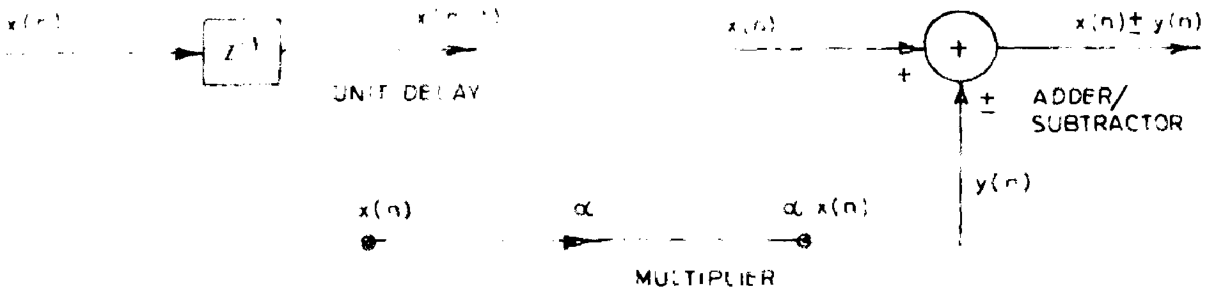
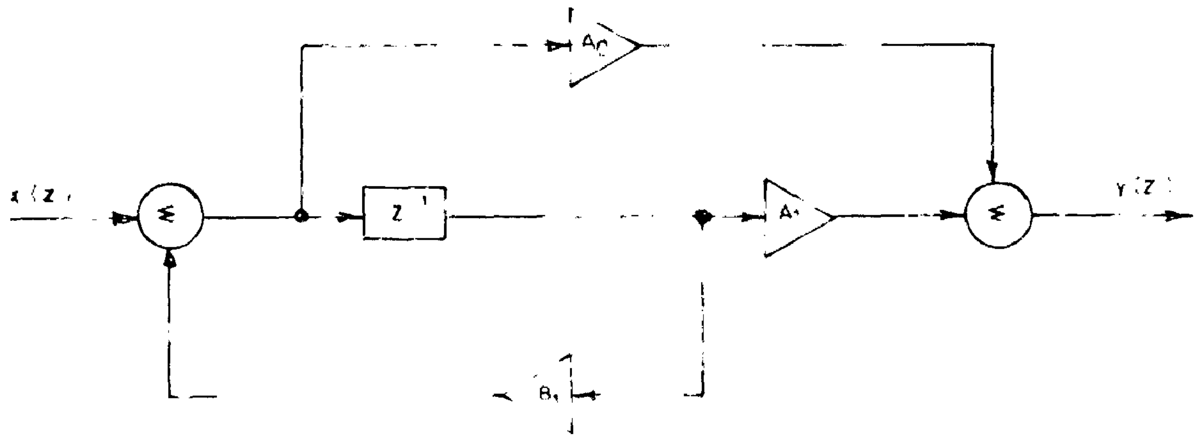
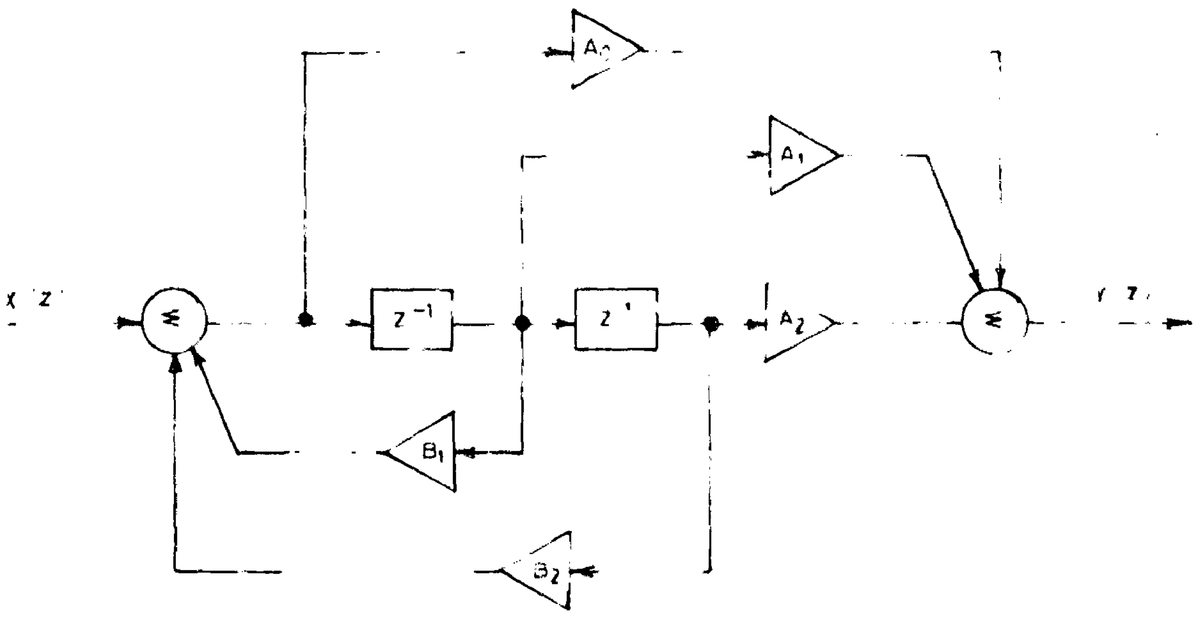


FIG 1.1 BASIC BUILDING BLOCKS



(a) FIRST ORDER IMPLIMENTATION



(b) SECOND ORDER IMPLIMENTATION

FIG 1.2 IMPLIMENTATION OF DIGITAL FILTERS

## 1.3 FILTER REALIZATION TECHNIQUES

This section gives the signal flow diagrams for Digital filter Transfer function in terms of the Digital filter elements namely, the adder, multiplier and the delay. These diagrams are known as realization structures [4,5,8] because it is in one of these forms that the practical realization is usually carried out. These different structures are classified into canonic and non-canonic realizations. By the term canonic realization [2,9] it is meant that the number of delay elements employed is precisely equal to the order of Transfer function (i.e. the highest degree between the numerator and denominator polynomials). The realization techniques include the following forms :

(1) Direct form (a) Direct form (canonical)

(b) Direct form (non-canonical)

(2) Cascade or Series Canonic form

(3) Parallel Canonic form

### 1.3-1 DIRECT FORM REALIZATION

This form includes all those Digital filters in which the real coefficients $A_i$ and $B_i$ of equation (1.3) appear as multipliers in the block diagram implementation. The following sections describe four types of direct structures.

## 1.3 - 1.1 FIRST DIRECT STRUCTURE (1D)

From equation (1.3)

$$D(z) = \frac{\sum_{i=0}^{N} A_i . Z^{-i}}{\sum_{i=0}^{N} B_i . Z^{-i}} \quad \ldots \quad \ldots \quad (1.10)$$

where, $B_0 = 1$ and $M = N$.

Introducing intermediate variable $M(z)$

$$D(z) = \frac{Y(z)}{X(z)} = \frac{Y(z)}{M(z)} . \frac{M(z)}{X(z)} = \frac{\sum_{i=0}^{N} A_i . Z^{-i}}{\sum_{i=0}^{N} B_i . Z^{-i}} \quad \ldots \quad (1.11)$$

Equating numerator and denominator separately

$$\frac{Y(z)}{M(z)} = \sum_{i=0}^{N} A_i . Z^{-i} \quad \text{and} \quad \frac{X(z)}{M(z)} = \sum_{i=0}^{N} B_i . Z^{-i} \quad \ldots \quad (1.12)$$

$$Y(z) = \sum_{i=0}^{N} A_i . Z^{-i} . (M(z) \quad \ldots \quad \ldots \quad (1.13)$$

$$X(z) = \sum_{i=0}^{N} B_i . Z^{-i} . M(z) \quad \ldots \quad \ldots \quad (1.14)$$

$$\text{or } M(z) = X(z) - \sum_{i=1}^{N} B_i . Z^{-i} . M(z) \quad \ldots \quad (1.15)$$

In the time domain equations (1.15) and (1.13) become

$$m(k) = x(k) - \sum_{i=1}^{N} B_i . m(k-i) \quad \ldots \quad (1.16)$$

$$y(k) = \sum_{i=0}^{N} A_i . m(k-i) \quad \ldots \quad \ldots . \quad (1.17)$$

Equations (1.16) and (1.17) define the first Direct structure 1D and is shown in Fig. (1.3). This structure is canonical because it possesses only N time delay elements, the minimum number for the N'th order Transfer function of equation (1.10).

## 1.3 - 1.2 SECOND DIRECT STRUCTURE (2D)

2D realization of Digital filter makes use of the principle of transposition [4,11]. Appendix-II, explains the transpose principle. The transpose of a Digital filter structure is accomplished by reversing the signal flow in all branches of the block diagram but leaving their transmittances the same. The transpose of a filter structure has the same Transfer function as the original structure.

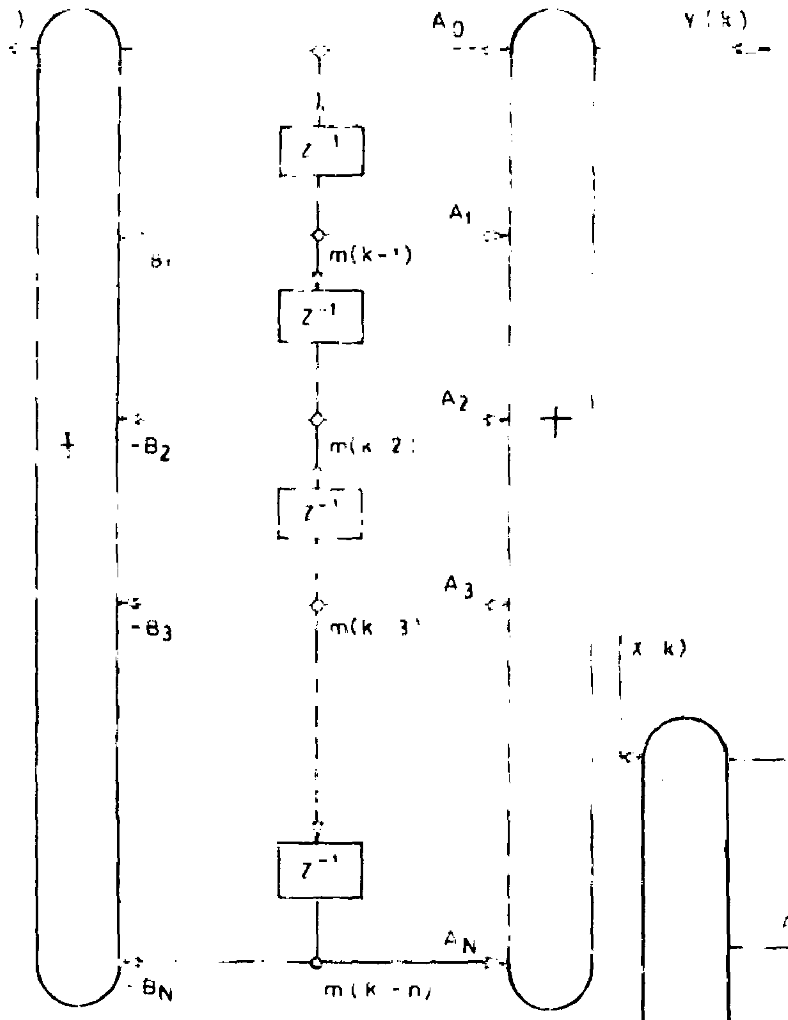
The 2D structure represented in Fig. (1.4) is the transpose of 1D structure. It implements equation (1.10) but requires (n + 1) difference equations (Summing Junctions). The 2D structure difference equations are of the form :

$$p_i(k) = p_{i+1}(k-1) + A_i.x(k) - B_i.y(k); \; i=1, N-1 \quad \ldots \quad (1.18)$$

$$p_N(k) = A_N.x(k) - B_N.y(k) \quad \ldots \quad (1.19)$$

$$y(k) = A_0.x(k) + p_1(k-1) \quad \ldots \quad (1.20)$$

This structure is also canonical because it possesses only N time-delay elements, the minimum number required for an N'th order Transfer function of equation (1.10).



13 1D STRUCTURE

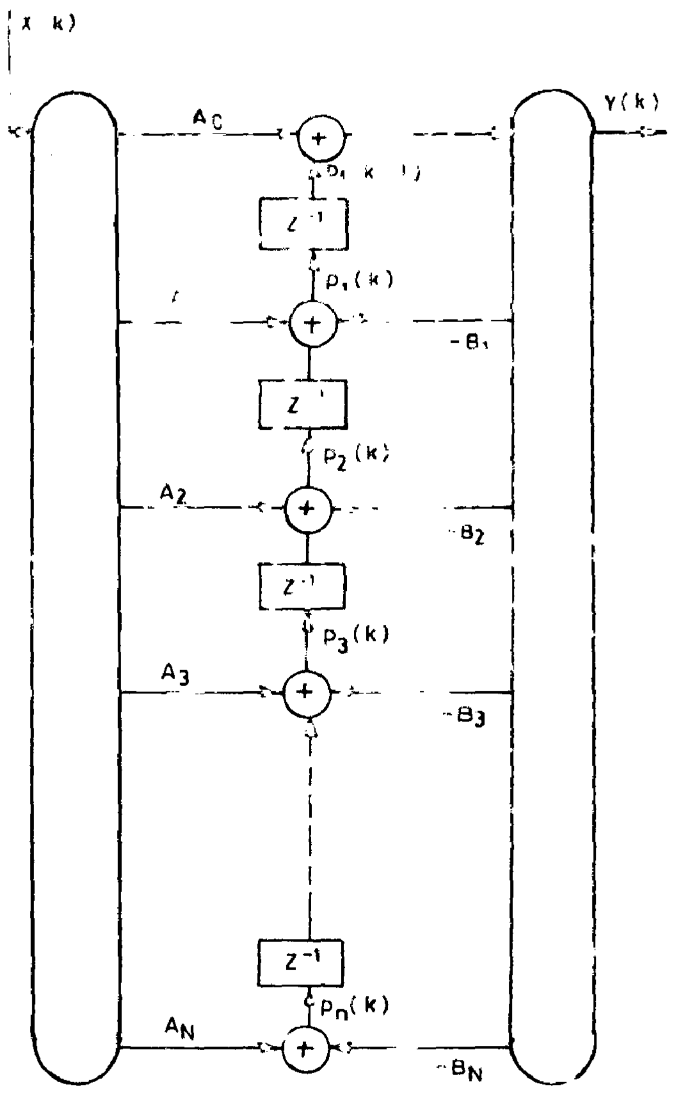


FIG 1.4 2D STRUCTURE

### 1.3 - 1.3 THIRD DIRECT STRUCTURE (3D)

Rewriting equation (1.10)

$$D(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^{N} A_i . Z^{-i}}{\sum_{i=0}^{N} B_i . Z^{-i}} \quad \ldots \quad (1.21)$$

or $$\sum_{i=0}^{N} B_i . Z^{-i} . Y(z) = \sum_{i=0}^{N} A_i . Z^{-i} . X(z) \quad \ldots . \quad (1.22)$$

or $$Y(z) = \sum_{i=0}^{N} A_i . Z^{-i} . X(z) - \sum_{i=1}^{N} B_i . Z^{-i} . Y(z) \quad \ldots \quad (1.23)$$

In the time domain, equation (1.23) becomes

$$y(k) = \sum_{i=0}^{N} A_i . x(k-i) - \sum_{i=1}^{N} B_i . y(k-i) \quad \ldots \quad \ldots \quad (1.24)$$

Equation (1.24) is the difference equation for the 3D Direct structure, which is block diagramed in Fig. (1.5). This structure has only one summing junction, but has 2N time delay elements, hence, a noncanonical form.

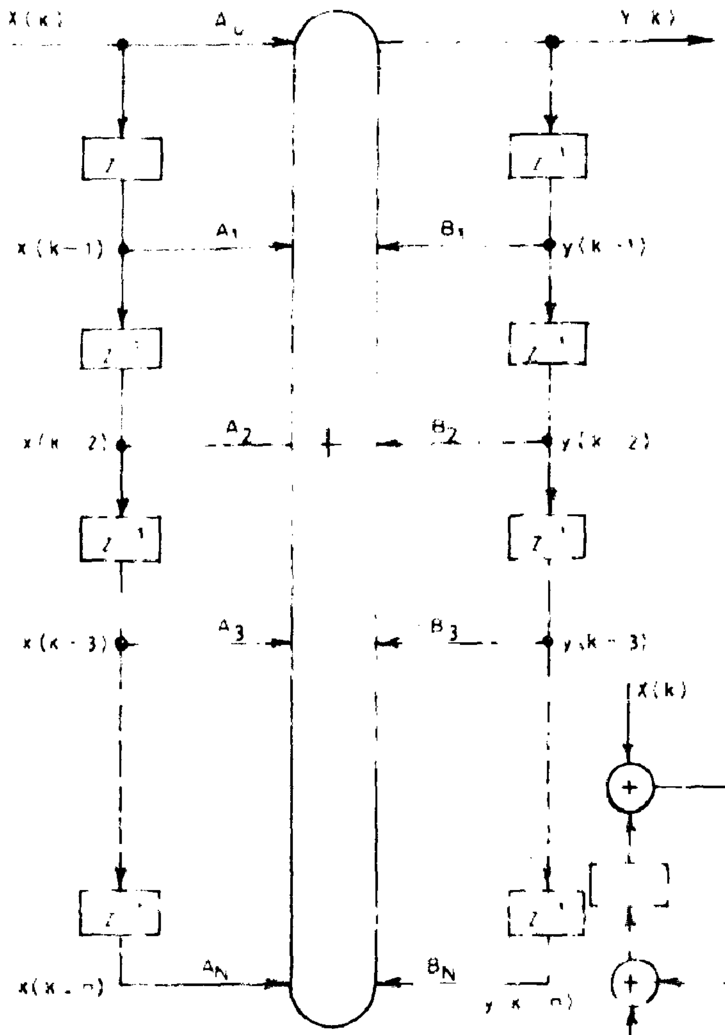
### 1.3 - 1.4 FOURTH DIRECT STRUCTURE (4D)

The 4D Direct structure is the transpose of 3D structure and is shown in Fig. (1.6). This structure has only one signal distribution point, but has 2N difference equations, expressed as follows :

$$r_o(k) = x(k) + r_1(k - 1) \quad \ldots \quad (1.25)$$

$$q_N(k) = A_N . r_o(k) \quad \ldots \quad (1.26)$$

$$r_N(k) = -B_N . r_o(k) \quad \ldots \quad (1.27)$$



IG 1 5 3D STRUCTURE

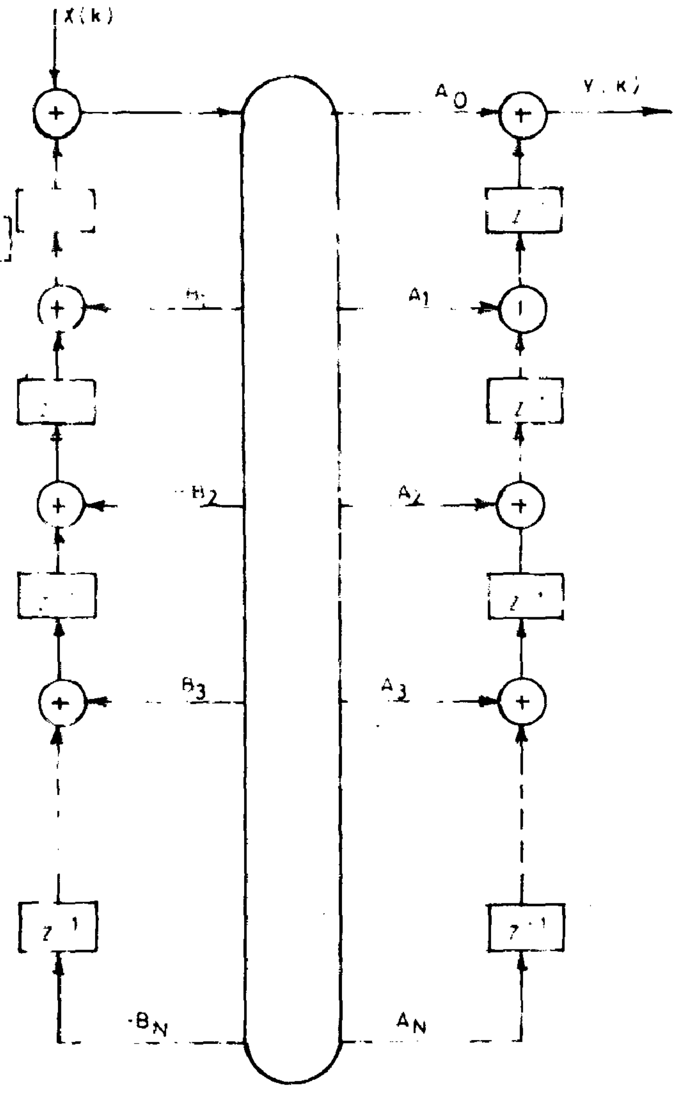
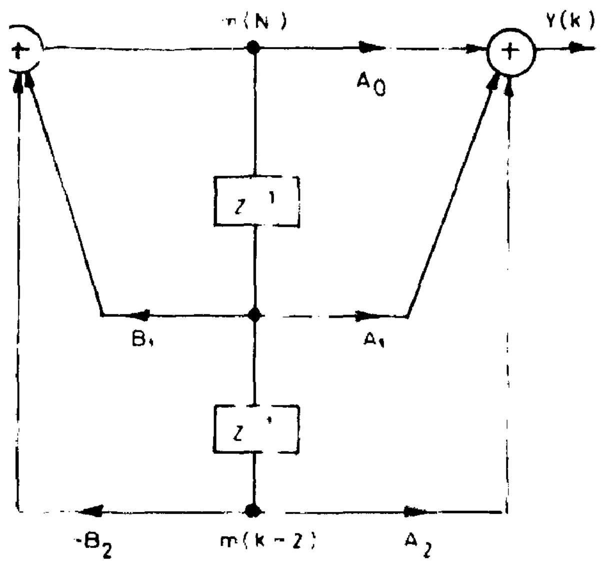
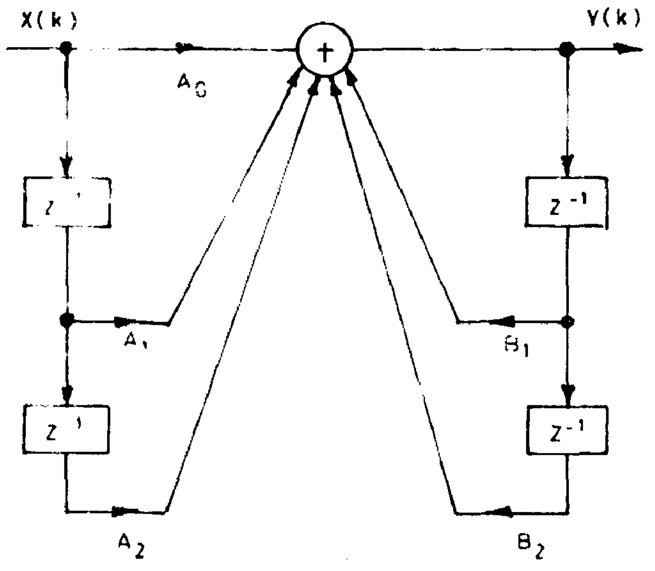


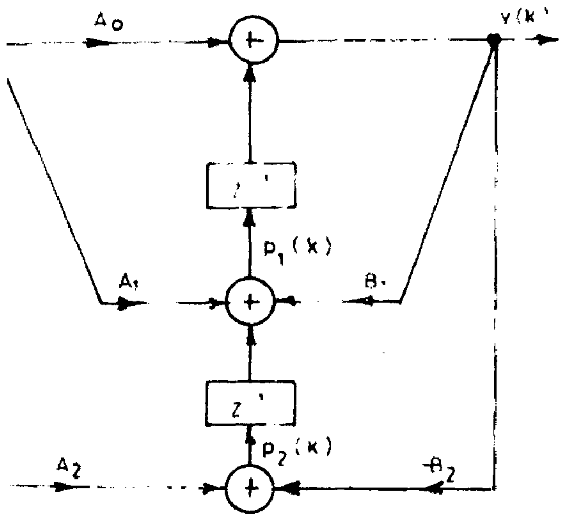
FIG 1 6 4D STRUCTURE



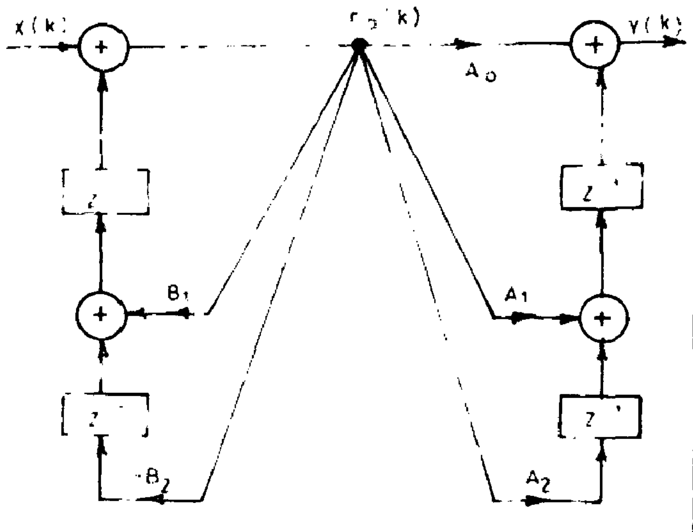
(a) 1D MODULE



(b) 3D MODULE



(c) 2D MODULE



(d) 4D MODULE

FIG 1 7 SECOND ORDER MODULE STRUCTURES

$q_i(k) = A_i . r_o(k) + q_{i+1}(k-1) \quad ; \ i = 1, N-1$ ... (1.28)

$r_i(k) = -B_i . r_o(k) + r_{i+1}(k-1)$ ... (1.29)

A comparison of the various characteristics of the four Direct structures discussed in Section 1.3-1 is summarized in TABLE - 1.1.

TABLE - 1.1

PROPERTIES OF DIRECT STRUCTURES

| CHARACTERISTICS | 1D | 2D | 3D | 4D |
|---|---|---|---|---|
| Time Delay Elements | N | N | 2N | 2N |
| Multipliers | 2N+1 | 2N+1 | 2N+1 | 2N+1 |
| Summing Junctions | 2 | N+1 | 1 | 2N |
| Signal Distribution Points | N+1 | 2 | 2N | 1 |

As will be explained in a later chapter second order Digital filter is the basic building block for realizing any N'th order Digital filter. The Transfer function of equation (1.10) can be implemented making use of these four structures. Fig. (1.7) illustrates the 1D, 2D, 3D and 4D structures for second order modules.

## 1.3-2 CASCADE FORM REALIZATION

The Cascade or Series canonic form structure for Digital filter is implemented from the Transfer function of equation (1.10), written as a product of factors.

$$D(z) = \prod_{i=1}^{N} H_i(z) \quad \ldots \quad \ldots \quad \ldots \quad (1.30)$$

where,

$$H_i(z) = \frac{A_{0i} + A_{1i}.Z^{-1}}{1 + B_{1i}.Z^{-1}} \quad \text{for first order..} (1.31)$$

$$\text{or } H_i(z) = \frac{A_{0i} + A_{1i}.Z^{-1} + A_{2i}.Z^{-2}}{1 + B_{1i}.Z^{-1} + B_{2i}.Z^{-2}} \quad \text{for second order} \ldots \quad (1.32)$$

The configuration is shown in Fig. (1.8). It consists of a series of lower order filters connected in cascade. The individual second order or first order equations are generally realized in one of the Direct forms. Fig. (1.9) illustrates the use of the Direct structures in cascade. Equations derived for the cascaded structures is same as that derived for the Direct form structures. These structures are compared in TABLE - 1.2.

TABLE - 1.2

PROPERTIES OF CASCADED STRUCTURES

| CHARACTERISTIC | 1D | 2D | 3D | 4D |
|---|---|---|---|---|
| Time Delay Elements | 2N | 2N | 2N+2 | 2N+2 |
| Multipliers | 5N | 5N | 5N | 5N |
| Summing junctions | N+1 | 3N | N | 3N+1 |
| Signal Distributing Points | 3N | N+1 | 3N+1 | N |

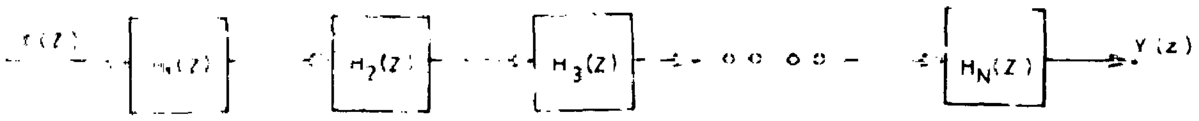


FIG 1 8 CASCADED SECOND ORDER MODULES

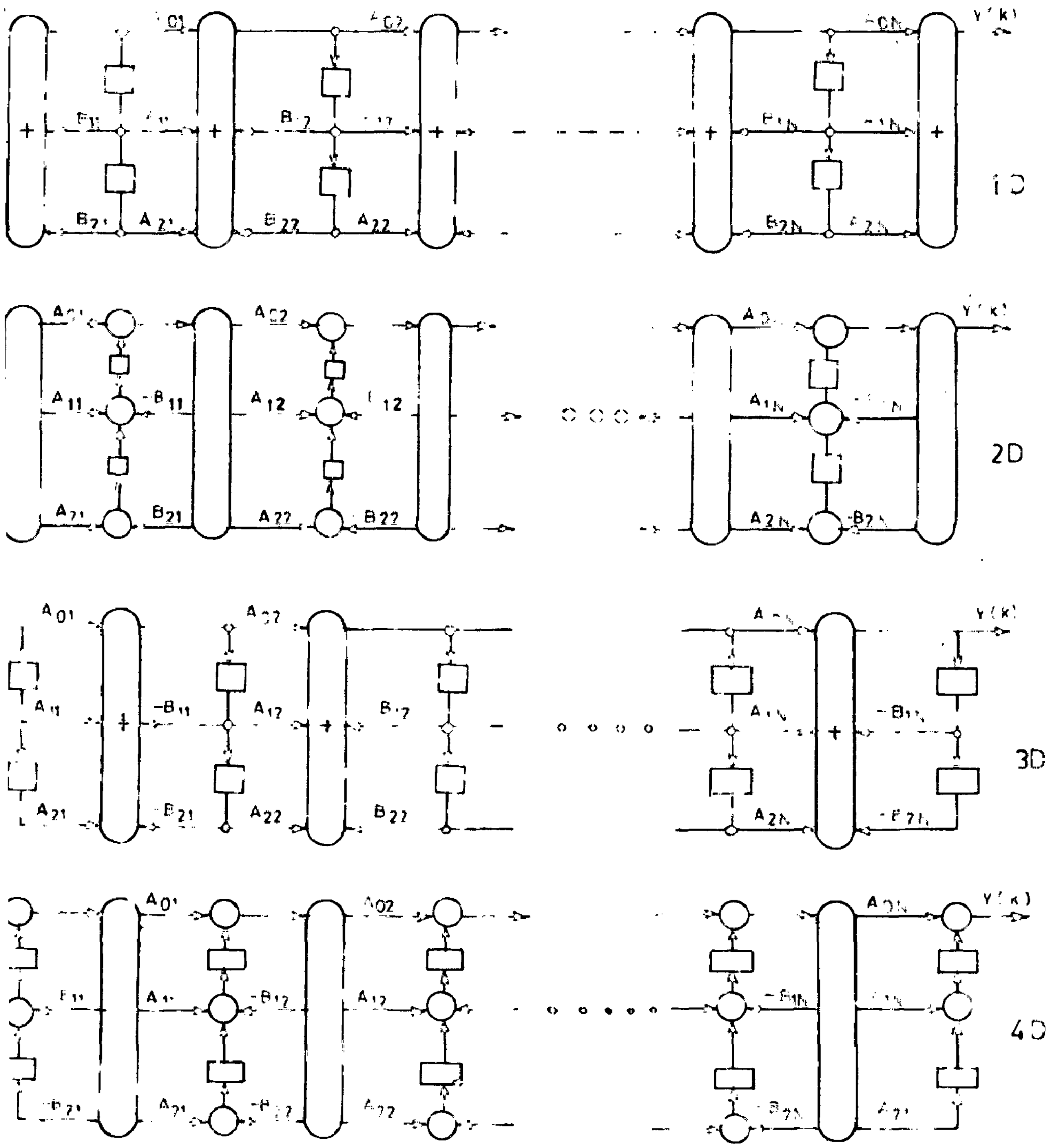


FIG 1 9 CASCADED FILTER STRUCTURES

### 1.3-3 PARALLEL FORM REALIZATION

The Parallel Canonic form structure for Digital filter is implemented by expanding the equation (1.10) in partial fraction form as

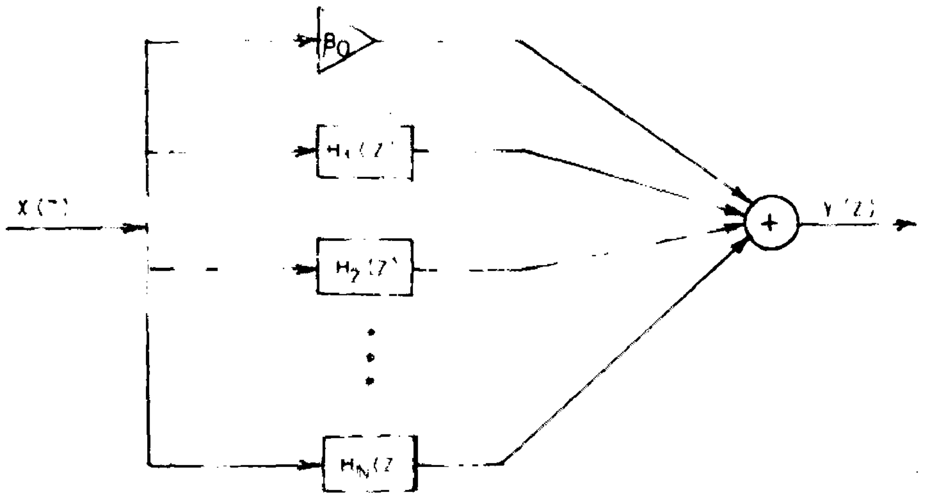
$$D(z) = \beta_0 + \sum_{i=1}^{N} H_i(z) \quad \ldots \quad (1.33)$$

where,

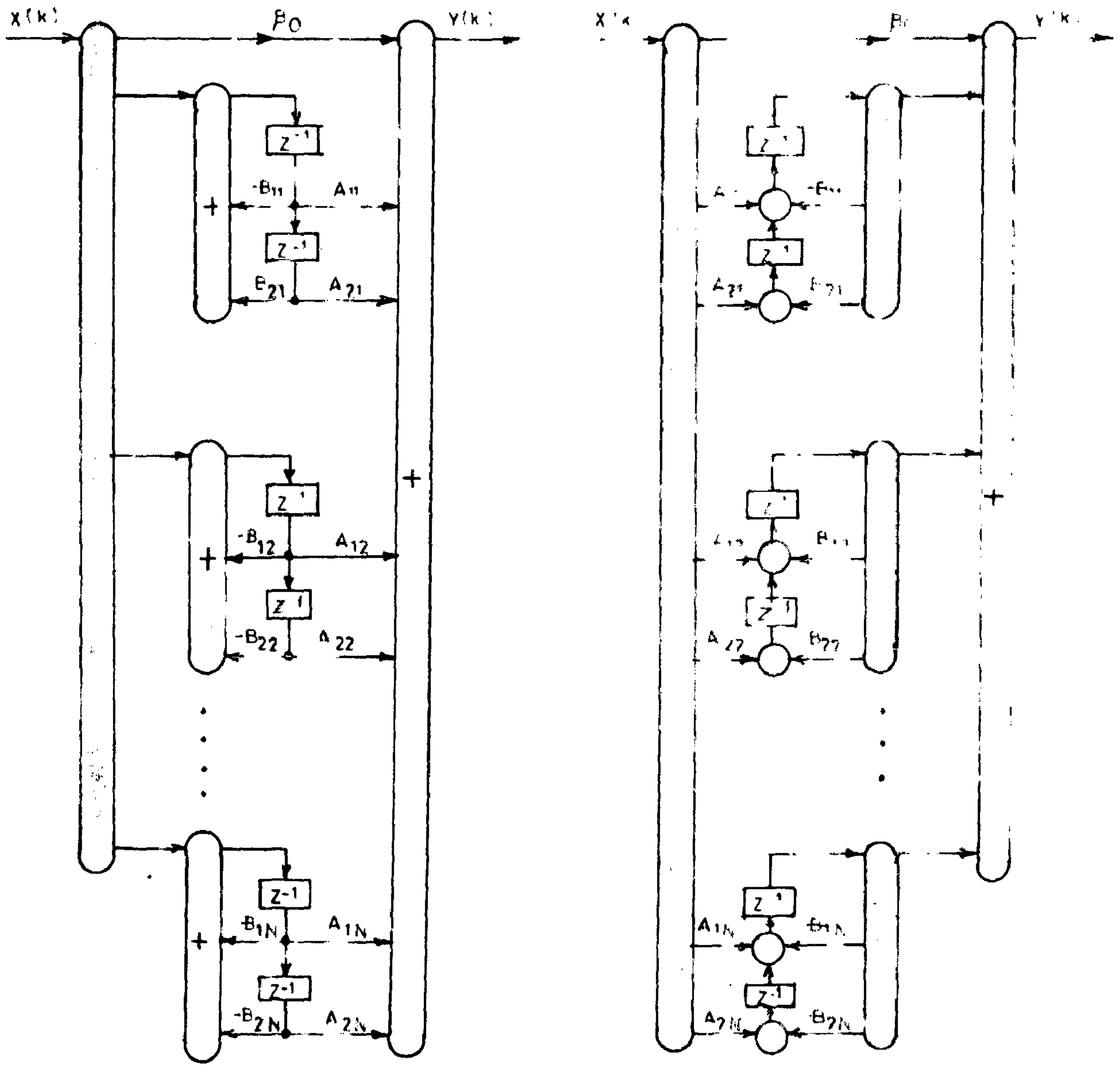
$$H_i(z) = \frac{A_{oi} + A_{1i}.Z^{-1}}{1 + B_{2i}.Z^{-1}} \quad \text{for first order} \quad \ldots \quad (1.34)$$

$$\text{or } H_i(z) = \frac{A_{oi} + A_{1i}.Z^{-1} + A_{2i}.Z^{-2}}{1 + B_i.Z^{-1} + B_i.Z^{-2}} \quad \text{for second order} \ldots \quad (1.35)$$

This configuration is shown in Fig. (1.10) and consists of a group of lower order filters each operating on the input signal with the output parallel bank summed up together. The individual second order or first order sections can be realized in one of the direct forms. If the Direct structures are used some element sharing may be accomplished as was done in the cascade case. Fig. (1.11) shows the direct parallel structure and TABLE - 1.3 compares their characteristics. Equations for the parallel structure is same as that derived for Direct form structure.

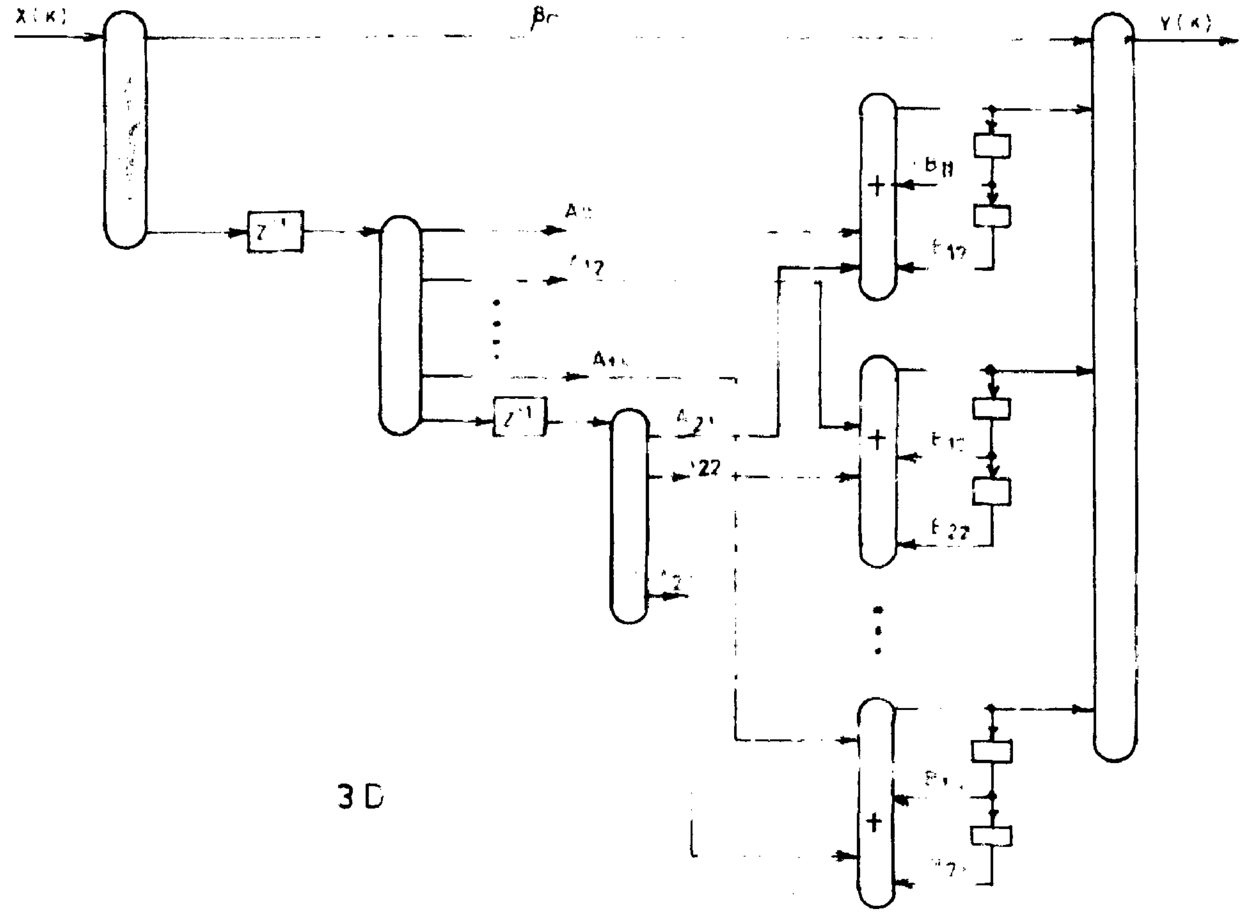


F.G 1.10 PARALLEL SECOND ORDER MODULES



1 [illegible]

2 [illegible]



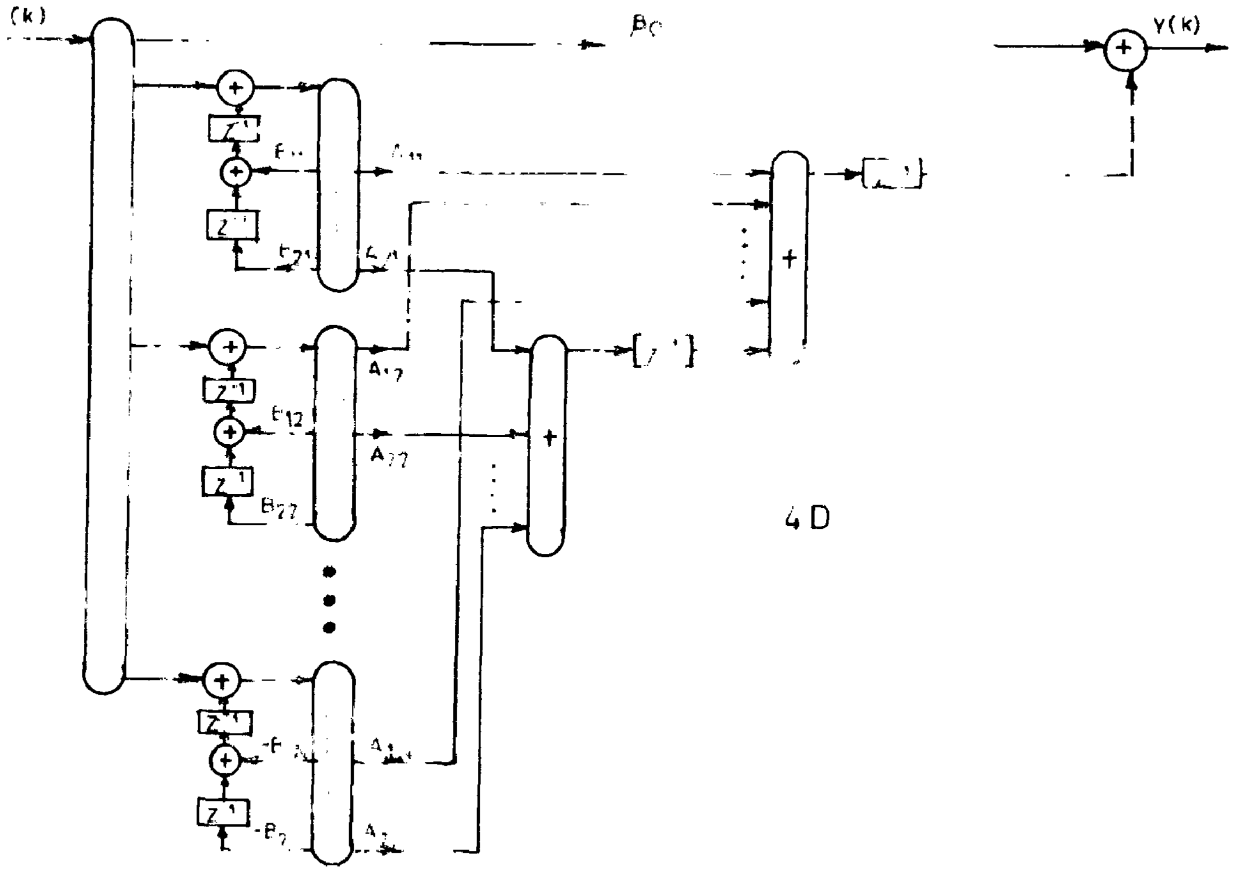


FIG 1.11 PARALLEL FILTER STRUCTURES

TABLE - 1.3

PROPERTIES OF PARALLEL STRUCTURE

| CHARACTERISTIC | 1D | 2D | 3D | 4D |
|---|---|---|---|---|
| Time Delay Elements | 2N | 2N | 2N+1 | 2N+1 |
| Multipliers | 4N+1 | 4N+1 | 4N+1 | 4N+1 |
| Summing Junctions | N+1 | 2N+1 | N+1 | 2N+3 |
| Signal Distributing Points | 2N+1 | N+1 | 2N+3 | N+1 |

## 1.4 SUMMARY

The Z-Transform calculus is the mathematical basis for the analysis and design of Digital filters. Such Digital filters are best understood by emphasizing the relations between the difference equations, the block diagram and filter response function. Various realization types have been discussed and the general equations involved, derived. Also, a comparison of the different characteristic present in each structure is made. The second order Digital filter, a basic module for realization of a N'th order structure, will be discussed in later chapters.

# CHAPTER - II

# SALIENT FEATURES OF INTEL 8086 MICROPROCESSOR

## 2.1 INTRODUCTION

Intel introduced its first microprocessor in November 1971. This was followed with the delivery of 8008 in 1972, the 8080 in 1974, the 8085 in 1976 and 8086 in 1978. Each successive product implementation depended on fabrication innovations, sophisticated software, and throughout this development upward compatibility not envisioned by the first designer was maintained.

The selection of a suitable microprocessor [13] depends primarily on the particular application. Since the characteristics of the various processors are quite different, a number of factors must be considered in making a good choice. The selection process involves investigating the software, hardware and system design of the microprocessor.

In this chapter the salient features of Intel 8086 microprocessor are discussed. The various microprocessor of the Intel group have been compared in Appendix -III for the selection of this suitable microprocessor.

## 2.2 SALIENT FEATURES OF MICROPROCESSOR 8086

Intel 8086 [20,21,22] introduced in June 1978 is the first of the high performance generation of 16 bit microprocessors. It is implemented in N channel depletion load, silicon gate technology (HMOS) and packaged in a 40 pin Cer. DIP package. The 8086 is able to directly address one megabytes (1024 K bytes) of external memory. The detailed pin out of the 8086 is shown in Fig. (2.1).

## 2.2-1 FUNCTIONAL PIN DESCRIPTION

1. $AD_{15} - AD_0$ : 2 - 16, 39, (I/O) Address Data Bus

Time multiplexed memory / IO address ($T_1$) and data ($T_2$, $T_3$, $T_w$, $T_4$) bus,

2. $A_{19}/S_6 - A_{16}/S_3$ : 35 - 38, (OUT) Address / Status

During $T_1$, used as address lines for memory operations. Lines LOW during I/O operation. In $T_2$, $T_3$, $T_W$ and $T_4$ status information is available on these lines. $S_3$ and $S_4$ indicate which of the segment (relocation) register is used (to construct the physical address used in the bus cycle). $S_5$ reflects the state of the interrupt enable flag. $S_6$ is always LOW.

| $S_4$ | $S_3$ | |
|---|---|---|
| 0 | 0 | Extra Segment (Alternate Data) |
| 0 | 1 | Stack Segment |
| 1 | 0 | Code Segment or none |
| 1 | 1 | Data Segment |

3. $\overline{BHE}/S_7$ : 34, (OUT) Bus High Enable / Status

During $T_1$ the Bus high enable signal ($\overline{BHE}$) is used to enable data on the most significant half of data bus

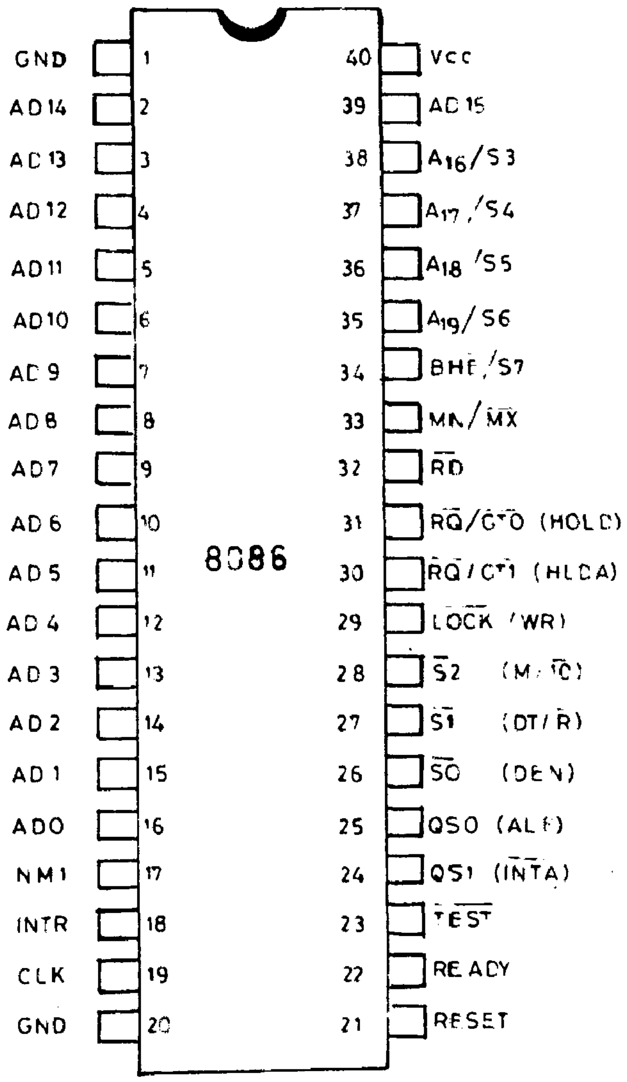


FIG. 2.1 8086 PIN DIAGRAM

(pin $D_{15}$ - $D_8$). $S_7$ is a spare status line whose contents are undefined.

4. MN/$\overline{\text{MX}}$ : 33, (IN) Minimum / Maximum mode

Indicates the system configuration. When this pin is grounded the 8086 treats pins 24 through 31 in maximum mode, when it is strapped to 5V it acts in the minimum mode.

5. $\overline{\text{RD}}$ : 32, (OUT) Read

Indicates that processor is performing a memory or I/O read cycle.

6. $\overline{\text{TEST}}$ : 23 , (IN) Test

$\overline{\text{TEST}}$ input examined by the WAIT (wait for $\overline{\text{TEST}}$) instruction. If the signal goes LOW execution continues, otherwise the processor waits in the 'Idle' state.

7. RESET : 21 , (IN) Reset

Causes the processor to immediately terminate its present activity and starts execution from FFFF0 (H).

8. CLK : 19 , (IN) Clock

Provides basic timing for the processor and bus controller.

9. INTR : 18 , (IN) Interrupt Request

It is a single interrupt request line which can be masked internally by software with the resetting of the Interrupt enable flag status bit. During the interrupt response sequence further interrupts are disabled. A single byte is then expected from interrupting device which is multiplied by 4 and gives the address of service routine pointer stored from 00000 (H) to 003FF (H).

10. NMI : 17 , (IN) Nonmaskable interrupt

Is a single nonmaskable interrupt which has a higher priority than the maskable interrupt request pin and causes a type 2 interrupt.

11. GND : 1, 20 Ground pin

12. VCC : 40 + 5V $\pm$ 10 %

Pin functions which are unique in the minimum mode are defined below.

(1) $\overline{\text{INTA}}$ 24 , (OUT) Interrupt Acknowledge

Is used as a read strobe for interrupt acknowledge cycle. It is active LOW in $T_2$, $T_3$ and $T_W$ states.

(2) ALE : 25 , (OUT) Address Latch Enable

Is provided to latch the address into the 8282/8283 address latch.

(3) $\overline{\text{DEN}}$ : 26, (OUT) Data Enable

Is provided as an output enable for the data bus transceiver.

(4) DT/$\overline{\text{R}}$ : 27 , (OUT) Data Transmit/Receive

This is needed in minimum mode system that desires to use a data bus transceiver. It is used to control the direction of data flow through the transciever.

5. $M/\overline{IO}$ : 28, (OUT) Status line

Is used to distinguish a memory access from an I/O access. HIGH, on this line indicates a memory operation and a LOW indicates an I/O operation.

6. $\overline{WR}$ : 29 , (OUT) Write

Indicates that the processor is performing a write memory or write I/O cycle, depending on the state of the $M/\overline{IO}$ signal.

7. HOLD and HLDA 31, 30 (I/O)

Indicates that another master is requesting to take control of the address and data bus. To be acknowledged, HOLD must be active HIGH. The processor receiving the 'hold' request will issue HLDA (HIGH) as an acknowledgement. When HOLD goes LOW, the processor will LOWer HLDA and the processor start on its next cycle.

When the 8086 is in the maximum mode the functions unique to it are described below.

1. $QS_1$, $QS_0$ : 24, 25, (OUT) Queue Status

Queue status valid during the CLK cycle after which the queue operation is performed. These provide status to allow external tracking of internal 8086 inst . queue.

| $QS_1$ | $QS_0$ | |
|---|---|---|
| 0 | 0 | No operation |
| 0 | 1 | First Byte of Opcode from queue |
| 1 | 0 | Empty the queue |
| 1 | 1 | Subsequent byte from queue |

2. $\overline{S}_2$, $\overline{S}_1$, $\overline{S}_0$ : 26 - 28, (OUT) Status

Status is used by the 8288 Bus Controller to generate all memory and I/O access control signals. These status lines are encoded as

| $\overline{S}_2$ | $\overline{S}_1$ | $\overline{S}_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | Interrupt Acknowledge |
| 0 | 0 | 1 | Read I/O Port |
| 0 | 1 | 0 | Write I/O Port |
| 0 | 1 | 1 | Halt |
| 1 | 0 | 0 | Code Access |
| 1 | 0 | 1 | Read Memory |
| 1 | 1 | 0 | Write Memory |
| 1 | 1 | 1 | Passive |

3. $\overline{\text{LOCK}}$ : 29, (OUT) Lock

It indicates that other system bus masters are not to gain control of the system bus while $\overline{\text{LOCK}}$ is active LOW. It is activated by the 'LOCK' prefix inst . and remains active until the completion of next instruction.

4. $\overline{RQ}$ / $\overline{GT}_0$, $\overline{RQ}$ / $\overline{GT}_1$ 30, 31, (OUT) Request/Grant.

These are used by local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. Each pin is bidirectional with $\overline{RQ}$ / $\overline{GT}_0$ having higher priority than $\overline{RQ}$ / $\overline{GT}_1$.

## 2.2-2 REGISTER ORGANIZATION

The 8086 processor contains a total of thirteen 16-bit registers and nine 1-bit flags. The set of registers Fig. (2.2) can be divided into the following four groups.

General Registers - There are four 16-bit general registers which can be used as either 8- or 16- bit registers. The dual nature of these registers permits them to handle both byte and word quantities with equal ease. They are Accumulator (AX), Base (BX), Count (CX) and Data (DX). The X can be replaced by H or L for referring to high or low order byte respectively.

The AX register is used in arithmetic operations to hold one of the two operands. The BX register can be used to hold an offset address for computing the effective address (EA) of an instruction operand. The CX and DX registers are used for specific purposes (dedicated general registers). These may be used as scratch pad during the evaluation of expressions or for holding the shift count in some shift and rotate instructions.

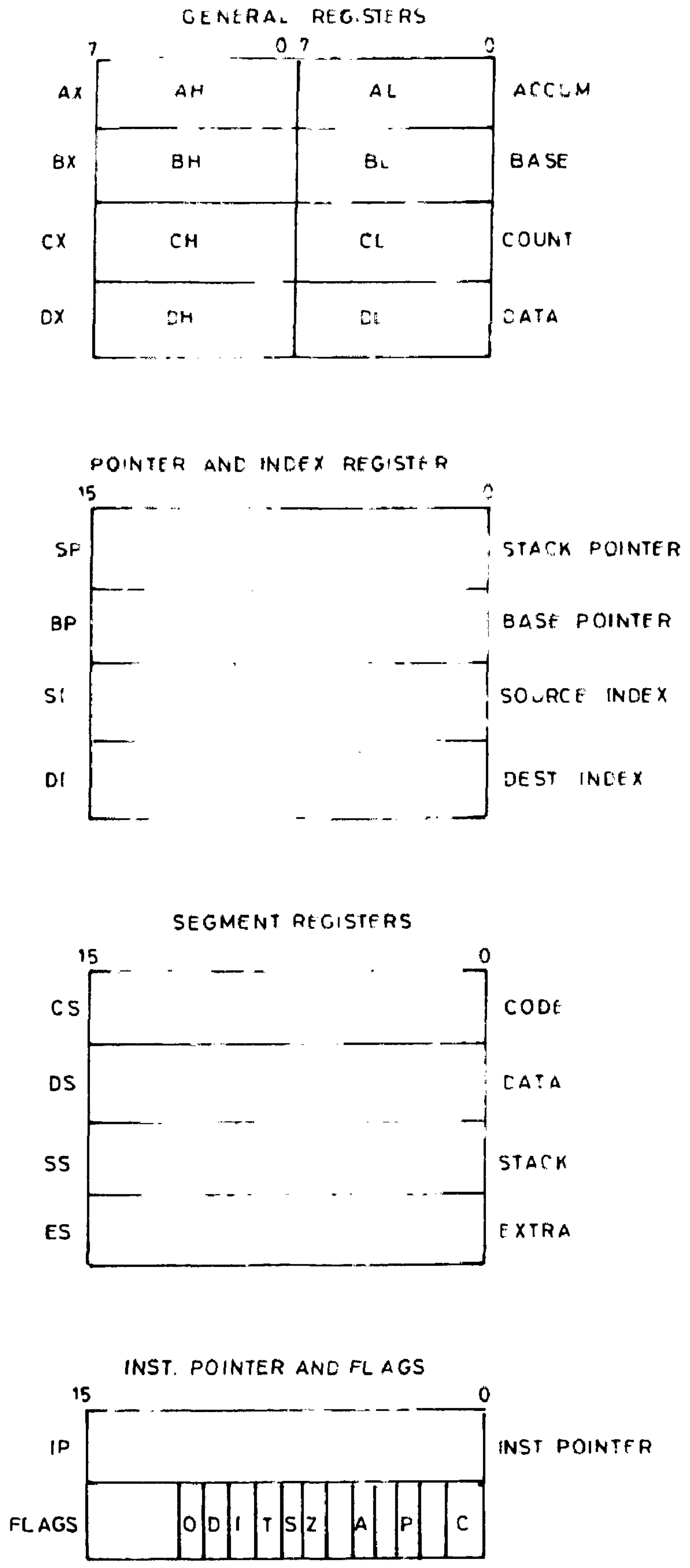


FIG. 2.2 8086 REGISTER STRUCTURE

Pointer and Index Registers - This group consists of the 16-bit registers Stackpointer (SP), Base pointer (BP) Source Index (SI) and Destination Index (DI). These registers usually contain offset addresses for addressing within a segment. They reduce the size of programs by not requiring each instruction to specify frequently used addresses. Another important function is that they provide for dynamic effective-address computations. In order to accomplish this the pointer and index registers participates in arithmetic and logical operations alongwith 16-bit general registers.

Segment Registers - This group consists of four 16-bit registers Code Segment (CS), Data Segment (DS), Stack Segment (SS) and Extra Segment (ES). Each segment can be at most 64K bytes in size. A segment can begin from any location in the memory that is divisible by 16.

The segment registers are used for calculation of physical address (PA). All instruction fetches are taken from the current code segment (CS) using the offset specified in the instruction pointer (IP) register. The (SS) register points to the current stack segment; stack operations are performed on locations in this segment. The (DS) points to current data segment and generally contains program variables. The ES contents define the current extra segment, it has no specific use although it is usually treated as an additional data segment.

Instruction Pointer and Flag Registers - The 16-bit instruction pointer (IP) (analogous to the program counter in the 8080/8085), is not directly accessible to the programmer; it is manipulated with control transfer instructions. There are nine 1-bit flags; six of these Carry (CF), Parity (PF), Auxillary carry (AF), Zero (ZF), Sign (SF) and overflow (OF) flags record processor status information of the latest arithmetic and logical operation and the additional three flags Direction (DF), Interrupt (IF) and Trap (TF) control processor operations.

## 2.2-3 MEMORY ORGANIZATION

The 8086 can address up to 1 Megabyte or 512 K words of memory directly. Logically the memory is organized as a sequence of $2^{20}$ bytes but physically it is organized in two banks each of 512 K bytes Fig. (2.3). One bank is connected to the lower half of the sixteen-bit data bus ($D_7 - D_0$) and contains even addressed bytes. The other bank is connected to the upper half of the data bus ($D_{15} - D_8$) and contains odd addressed bytes. A specific byte within each bank is selected by address lines $A_{19} - A_1$. The most significant address bit $A_0(AD_0)$ and the output signal $\overline{BHE}$ are used to select appropriate bytes to be read from or written into the memory.

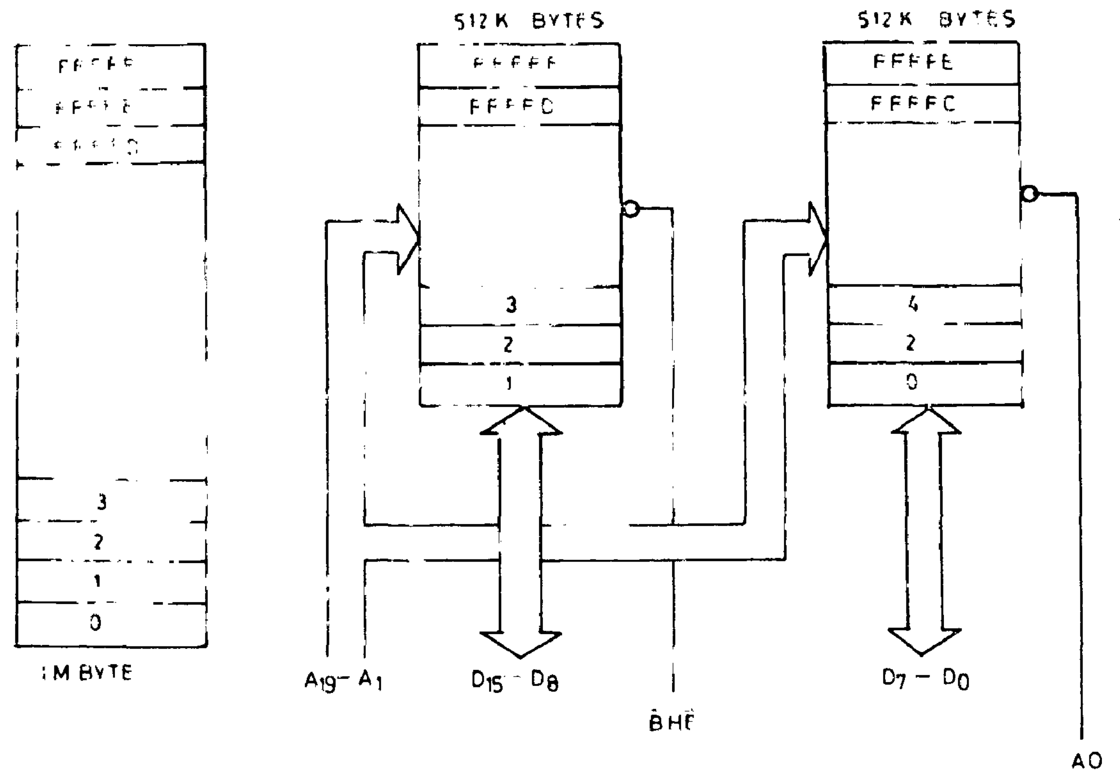
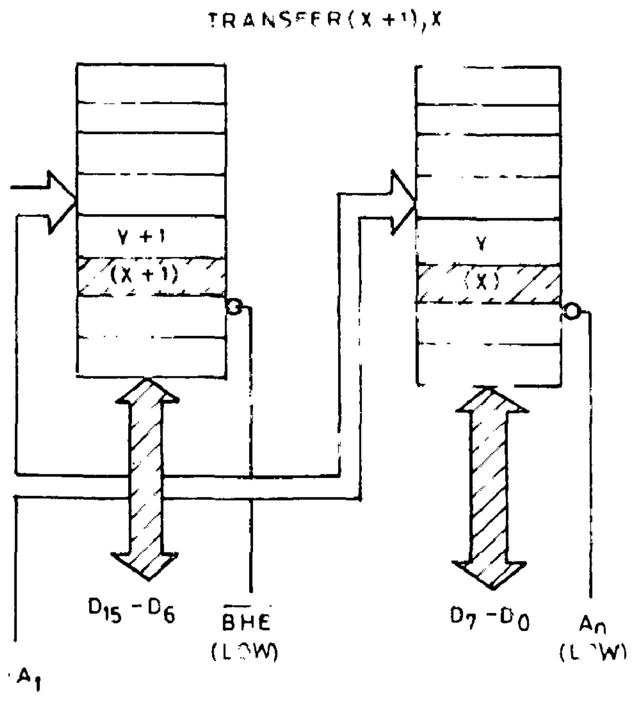


FIG 2.3 8086 MEMORY ORGANISATION



2 4(a) EVEN ADDRESS WORD TRANSFER

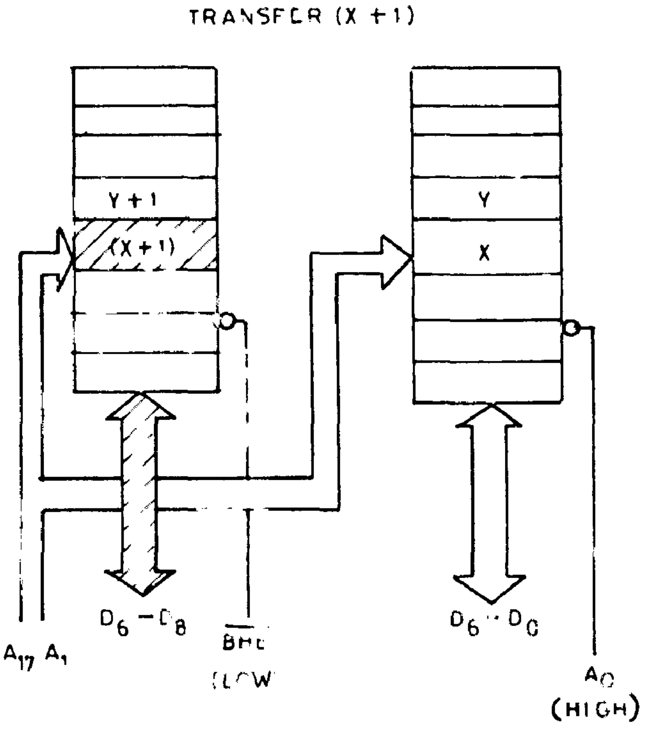


FIG 2 4(b) ODD ADDRESS BYTE TRANSFER

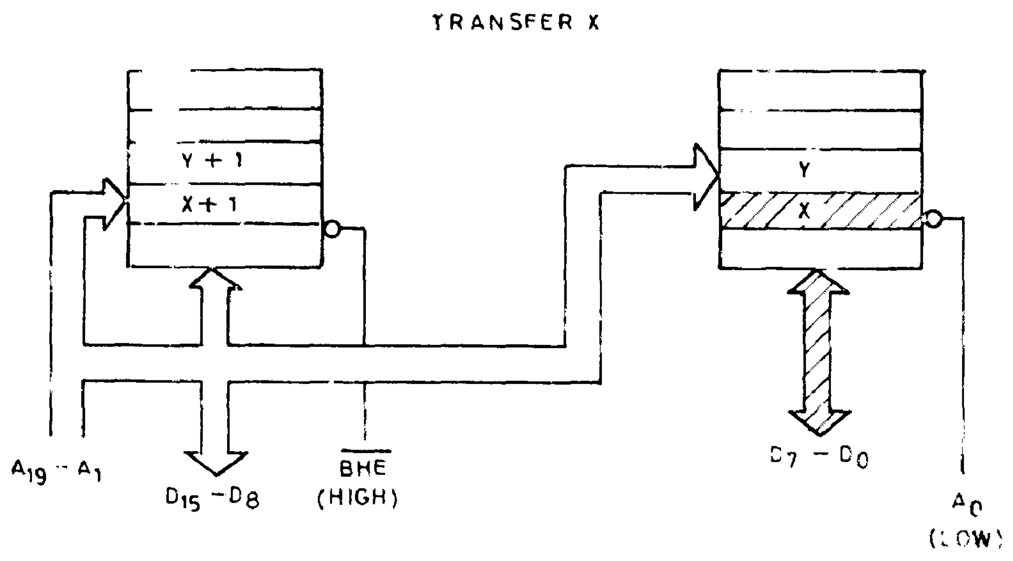
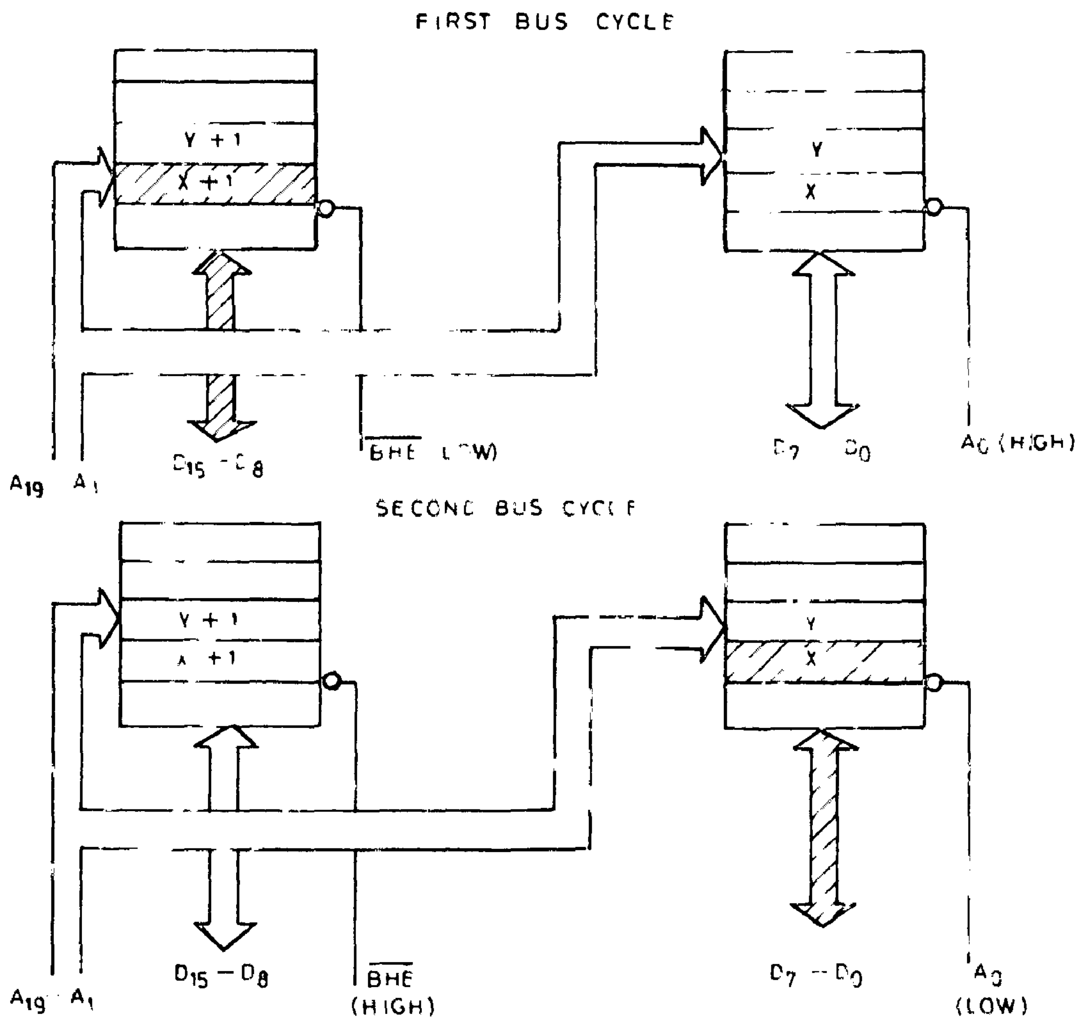


FIG. 2.4(c) EVEN ADDRESS BYTE TRANSFER



ODD ADDRESSING WORD TRANSFER

FIG 2 4(d) ADDRESSING TRANSFER

TABLE - 2.1 describes the use of $\overline{BHE}$ and $A_0$ combination.

TABLE - 2.1

| $\overline{BHE}$ | $A_0$ | |
|---|---|---|
| 0 | 0 | One 16 bit word |
| 0 | 1 | One byte from / to edd address |
| 1 | 0 | One type from / to even address |
| 1 | 1 | None |

Organization of N bytes of memory is shown in Fig. (2.4). The low bank consists of only even-address bytes and the high bank consists of only odd-address bytes.

(1) A word is to be fetched from an even byte location. For this $\overline{BHE}$ $A_0$ = 0 0 ; low byte of word falling on even-address byte and high byte on odd-address byte.

(2) A word is to be fetched from odd-address. This shall require two machine cycles. In the first, odd byte shall be read and in the next machine cycle the even byte shall be read.

(3) A byte is to be fetched from even-address location. For this $\overline{BHE}$ $A_0$ = 1 0 and data shall be transferred on $D_0$ - $D_7$ lines.

(4) A byte is to be fetched from odd-address. For this $\overline{BHE}$ $A_0$ = 0 1 and data shall be transferred on $D_8 - D_{15}$ lines.

The memory can be further logically divided into code, data, alternate data and stack segments of upto 64 K bytes each, with each segment falling on 16 byte boundary (Fig. 2.5).

Certain memory locations are reserved for specific processor operations. Locations 00000 (H) through 003FF (H) are reserved for interrupt operations. Each of the 256 possible interrupts have their service routine pointed by a 4-byte pointer element. Following RESET, the processor will jump to FFFF0 (H). FFFF0 (H) through FFFFF(H) are reserved for operation including a jump to the initial program loading routine (Fig. 2.6).

It is useful to think of every memory location as having two kinds of addresses, physical and logical. A physical address is the 20-bit value that uniquely identifies each byte location in the megabyte memory space. Physical address may range from 0(H) through FFFFF (H). All exchanges between the CPU and memory components use this physical address.

Programs deal with logical, rather than physical addresses and allow code to be developed without prior

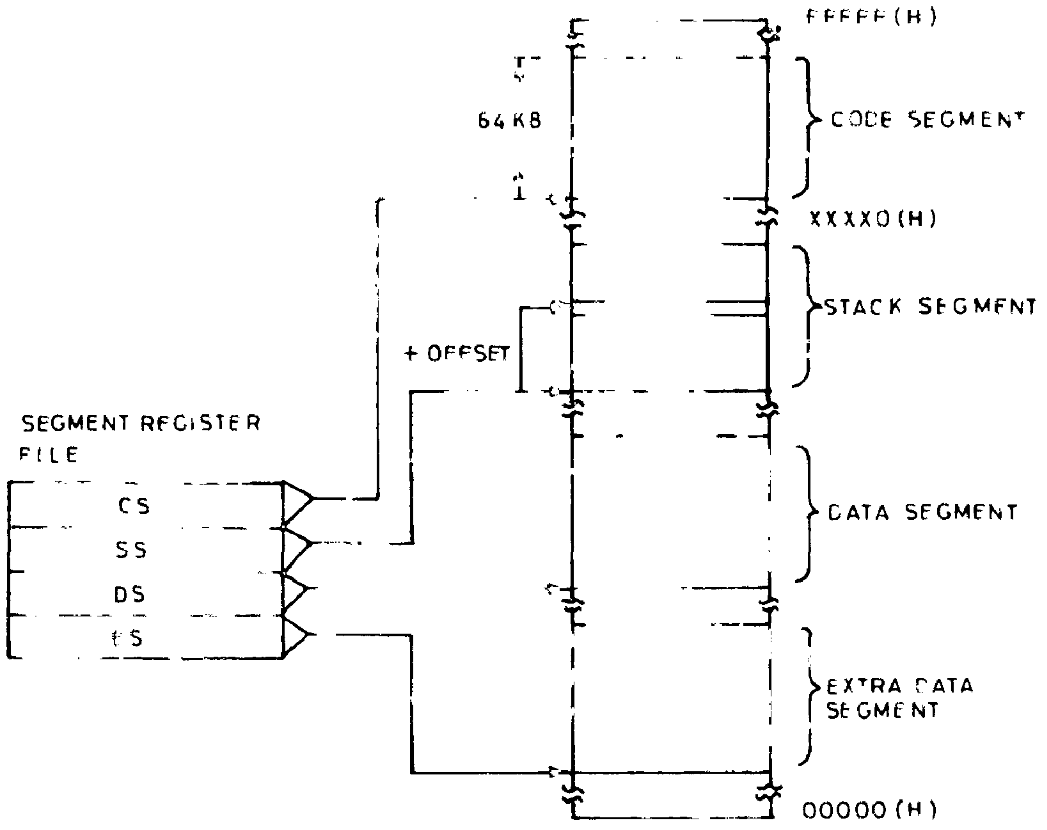


FIG 2.5 MEMORY ORGANIZATION

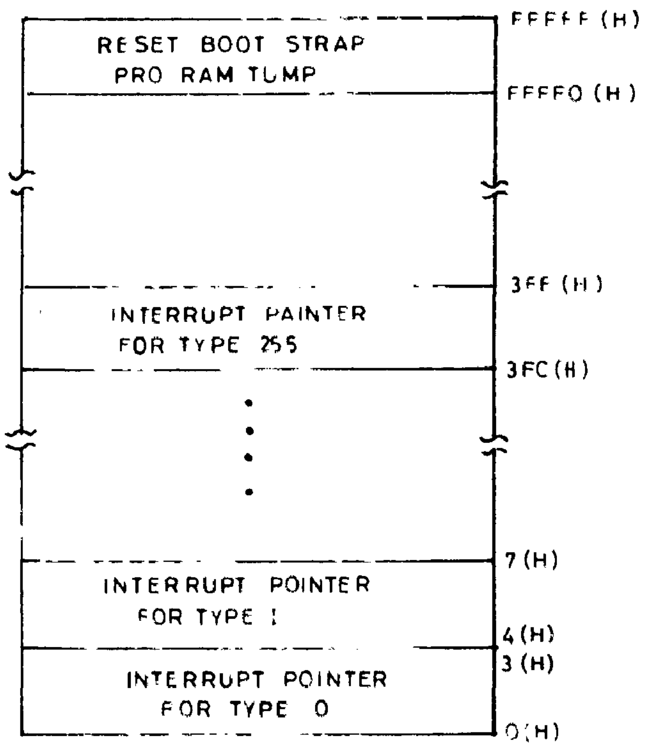

FFFFF (H)
RESET BOOT STRAP PRO RAM TUMP
FFFF0 (H)
3FF (H)
INTERRUPT PAINTER FOR TYPE 255
3FC (H)
7 (H)
INTERRUPT POINTER FOR TYPE 1
4 (H)
3 (H)
INTERRUPT POINTER FOR TYPE 0
0 (H)

FIG.2 6 RESERVED MEMORY LOCATIONS

knowledge of where the code is to be located in memory. A logical address consists of a segment base value and an offset value. For any given memory location, the segment base value locates the first byte of the containing segment and the offset value is the distance, in bytes, of the target location from the beginning of the segment. Segment base and offset values are unsigned 16-bit quantities. Many different logical addresses can map to the same physical location.

A physical address is generated from a logical address by shifting the segment base value four bit positions and adding the offset. Calculation of the offset of a memory variable is based on the addressing mode specified in the instruction; the result is called the operand's effective address (EA).

## 2.2.4 ADDRESSING MODES

Following are the different ways of calculating effective address (EA) and are shown in Fig. (2.7).

Direct Addressing - It is the simplest memory addressing mode. No registers are involved; the EA is taken directly from the displacement field of the instruction. Direct addressing is used to access simple variables. The EA is added to the DS segment to get the physical address.

$$(EA) = DISP$$
$$(PA) = (DS) * 16 + DISP$$

Example : The instruction MOV AX , VALUE specifies that the contents of VALUE are to be moved to 16-bit accumulator. Let address associated with VALUE be 021(H), then the physical address from which the low data byte will be fetched is

$$= 1000(H) * 16 + 021(H)$$

$$= 10000(H) + 021(H) = 10021(H)$$

The high byte will be fetched from the physical address 10022(H).

Based Addressing - Here the effective address is the sum of a displacement value and the content of register BX or BP. If BP is specified as a base register, the BIU is directed to obtain the operand from the current stack segment. This makes based addressing a very convenient way to access stack data.

Based addressing provides a straightforward way to address structures which may be located at different places in memory. A base register can be pointed at the base of the structure and elements of the structure addressed by their displacements from the base. Different copies of the same structure can be accessed by simply changing the base register.

$$(EA) = (BX/BP) + DISP$$

$$\text{for BX}, \quad (PA) = (DS) * 16 + (BX) + DISP$$

$$\text{for BP}, \quad (PA) = (SS) * 16 + (BP) + DISP$$

Example : The instruction MUL BETA (BX) implies that the contents of AX are to be multiplied by the contents of (EA). The (EA) is computed as DISP + (BX) where DISP is the 16 bit address of BETA. The 32 bit product will be placed in registers DX (high word) and AX (low word).

Indexed Addressing - The effective address is calculated from the sum of a displacement plus content of an index register (SI or DI, SP or BP). Indexed addressing often is used to access elements in an array. Also it is assumed that the operand resides in the current data segment and hence DS register is used for computing physical address.

$$(EA) = (IX) + DISP$$

$$(PA) = (DS) * 16 + (IX) + DISP$$

Based Index (Indirect) Addressing - This addressing generates an effective address that is the sum of a base register, an index register and a displacement. Based index addressing is a very flexible mode because two address components can be varied at execution time. It provides a convenient way for a procedure to address an array allocated on a stack. Arrays contained in structures and matrices (two dimensional arrays) also could be accessed with based index addressing.

$$(EA) = (BX/BP) + (IX) + DISP$$

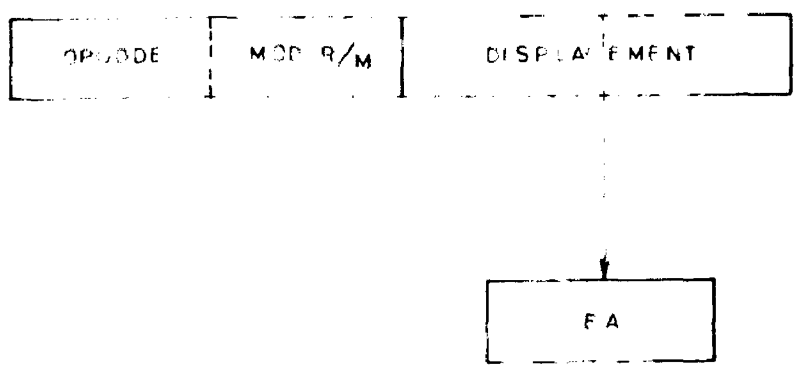
$$(PA) = (BX) + (IX) + DISP + (DS) * 16$$

$$(PA) = (BP) + (IX) + DISP + (SS) * 16$$

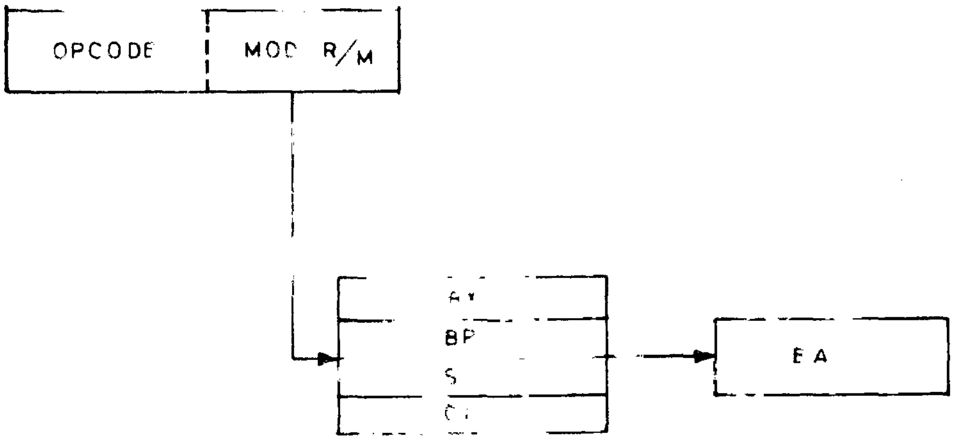
String Addressing - String instructions do not use the normal memory addressing modes to access their operands. Instead, the index registers are used implicitly, when a string instruction is executed, SI is assumed to point to the first byte or word of the source string, and DI is assumed to point to the first byte or word of the destination string. In a repeated string operation the CPU automatically adjust SI and DI to obtain subsequent bytes or words.

I/O Port Addressing - If an I/O port is memory mapped, any of the memory operand addressing modes may be used to access the port. String instructions also can be used to transfer data to memory-mapped ports with appropriate hardware interface.

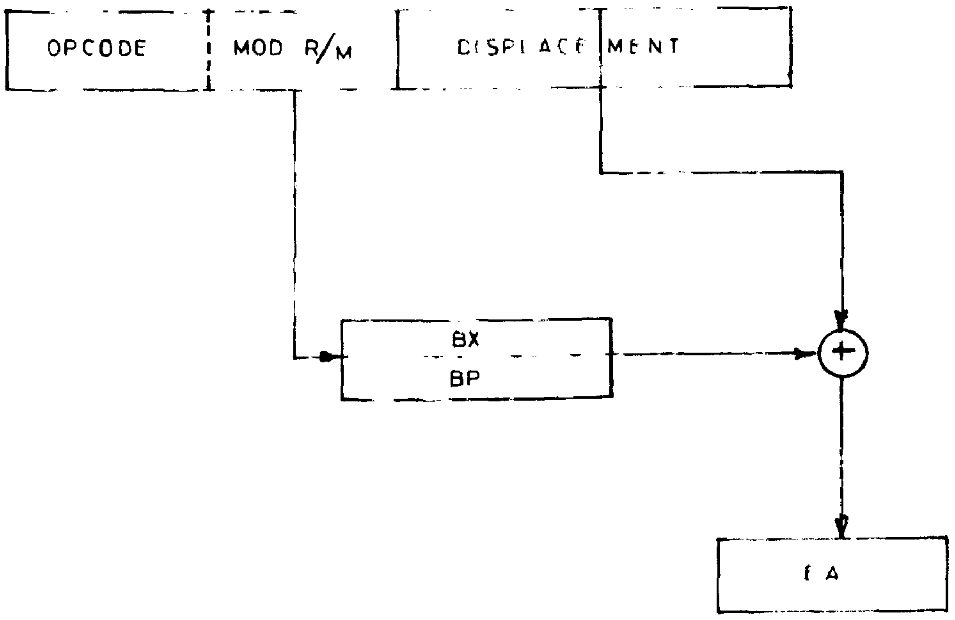
Two different addressing modes can be used to access ports located in the I/O space. In direct port addressing, the port number is an 8 bit immediate operand. This allows fixed access to ports numbered 0-255. Indirect port addressing is similar to register indirect addressing of memory operands. The port number is taken from register DX and can range from 0 to 65,535.



DIRECT ADDRESSING

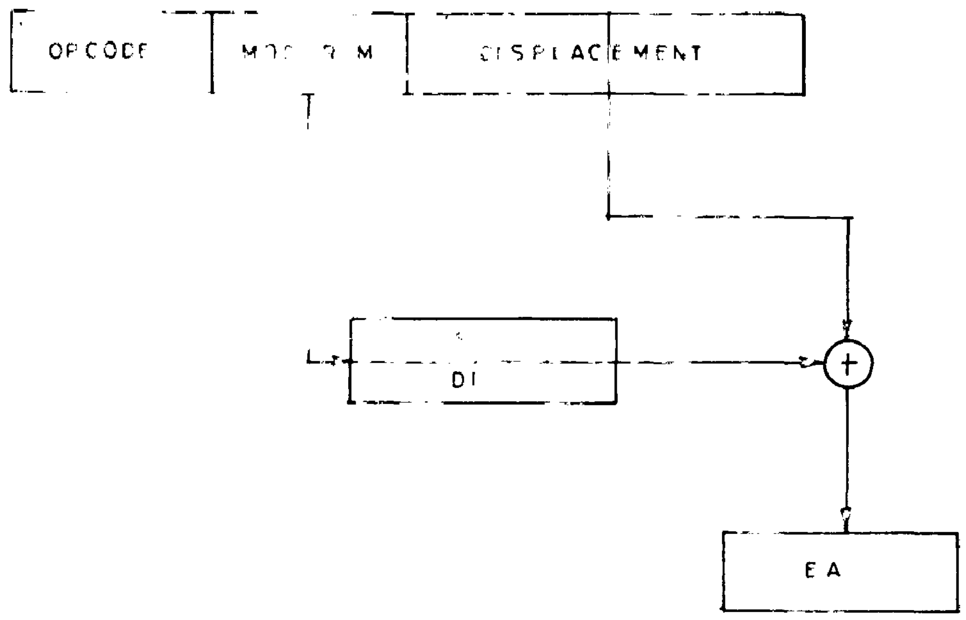


REGISTER INDIRECT ADDRESSING

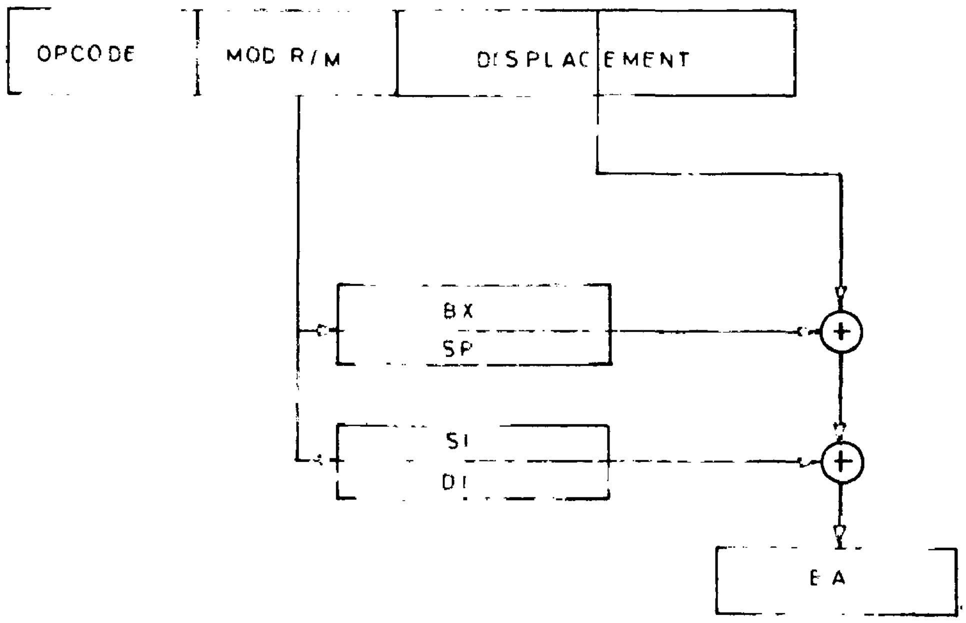


BASED ADDRESSING

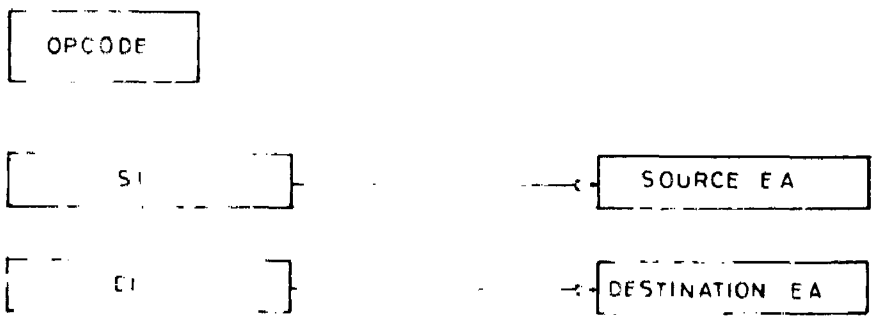
CONTD.



INDEXED ADDRESSING

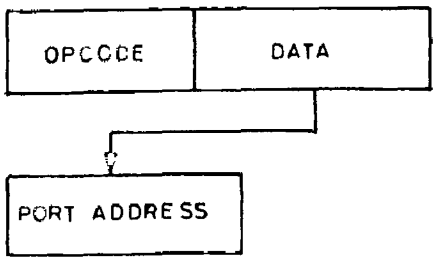


BASED INDEX ADDRESSING

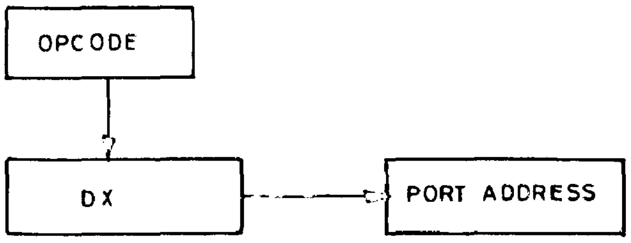


STRING OPERAND ADDRESSING

CONTD.



DIRECT PORT ADDRESSING



INDIRECT PORT ADDRESSING

FIG. 2.7 CALCULATION OF EFFECTIVE ADDRESS

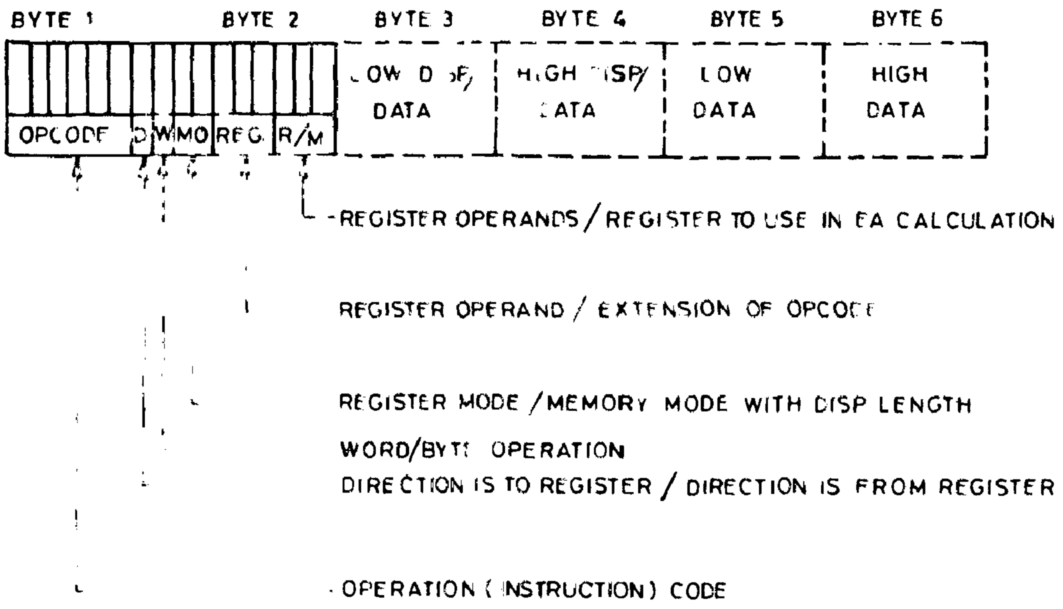

BYTE 1
BYTE 2
BYTE 3
BYTE 4
BYTE 5
BYTE 6
LOW DISP/ DATA
HIGH DISP/ DATA
LOW DATA
HIGH DATA
OPCODE
D
W
MOD
REG.
R/M
REGISTER OPERANDS / REGISTER TO USE IN EA CALCULATION
REGISTER OPERAND / EXTENSION OF OPCODE
REGISTER MODE / MEMORY MODE WITH DISP LENGTH
WORD/BYTE OPERATION
DIRECTION IS TO REGISTER / DIRECTION IS FROM REGISTER
OPERATION (INSTRUCTION) CODE

FIG. 2.8 TYPICAL 8086 MACHINE INST. FORMAT

## 2.2-5 INSTRUCTION SET

The 8086 instruction set is divided in six groups

| | | | |
|---|---|---|---|
| (1) | Data Transfer | (4) | String Manipulation |
| (2) | Arithmetic | (5) | Control Transfer |
| (3) | Bit Manipulation | (6) | Processor Control |

These instructions treat different type of operands uniformly. Nearly every instruction can operate on either byte or word data. Register, memory and immediate operands may be specified interchangeably in most instructions. The instruction set can be viewed as existing at two levels; the assembly level and the machine level. These two levels address two different requirements; efficiency and simplicity The numerous forms of machine level instructions allow these instructions to make very efficient use of storage. The assembly-level instructions simplify the programmer's view of the instruction set.

To pack instructions into memory as densely as possible the 8086 CPU utilizes an efficient coding techniques. Machine instructions vary from one to six bytes in length. One byte instructions, which generally operate on single registers or flags, are simple to identify. The key to decoding longer instructions are in the first two bytes. The format of these bytes can vary, but most instructions follow the format shown in Fig. (2.8).

The first six bits of a multibyte instruction generally contain an opcode that identifies the basic instruction type ADD, XOR, etc. The following bit, called the D field, generally specifies the 'direction' of the operation :

1 = the REG field in the 2nd byte identifies the destination operand.

0 = the REG field identifies the source operand

The w field distinguishes between byte and word operations :

0 = byte, 1 = word

One of the three additional single bit fields, S, V or Z appears in some instruction formats. S is used in conjunction with W to indicate sign extension of immediate fields in arithmetic instructions. V distinguishes between single and variable bit shifts and rotates. Z is used as a compare bit with the zero flag in conditional repeat and loop instructions. All single bit field settings are summarized in TABLE 2.2.

TABLE 2.2
SINGLE BIT FIELD ENCODING

| FIELD | VALUE | FUNCTION |
|---|---|---|
| S | 0 | No sign extension |
| | 1 | Sign extension 8-bit immediate data to 16-bit if w =1 |
| W | 0 | Inst. operates on byte data |
| | 1 | Inst. operates on word data |
| D | 0 | Inst. source specified in REG field |
| | 1 | Inst. Destination specified in REG field |
| V | 0 | Shift/rotate count is one |
| | 1 | Shift/rotate count specified in CX register |
| Z | 0 | Repeat/Loop while zero flag is clear |
| | 1 | Repeat/Loop while zero flag is set. |

The second byte usually identifies the instruction's operands. The mode (MOD) field indicates whether one of the operands is in memory or whether both operands are registers TABLE - 2.3. The register (REG) field identifies a register that is one of the instruction operands. TABLE - 2.4. In a number of instructions, chiefly the immediate-to-memory variety, REG is used as an extension of the opcode to identify the type of operation. The encoding of the R/M (register/memory) field in TABLE - 2.5, depends upon how the mode field is set. If MOD = 11 (register to-register mode), then R/M identifies the second register operand. If MOD selects memory mode, then R/M indicates how the effective address of the memory operand is to be calculated.

Bytes 3 through 6 of an instruction are optional fields that usually contain the displacement value of a memory operand and/or the actual value of an immediate constant operand. There may be one or two displacement bytes. The MOD field indicates how many displacement bytes are present. Following Intel convention, if the displacement is two bytes the most significant byte is stored second in the instruction. If the displacement is only a single byte the 8086 automatically sign-extends this quantity to 16 bits before using the information in further address calculations. Immediate values always follow any displace-

TABLE - 2.3

MODE FIELD ENCODING

| CODE | EXPLANATION |
|---|---|
| 00 | Memory mode, no displacement follows (except when R/M is 110) |
| 01 | Memory mode, 8-bit displacement follows |
| 10 | Memory mode, 16-bit displacement follows |
| 11 | Register mode, no displacement |

TABLE - 2.4

REGISTER/FIELD ENCODING

| REGISTER | W = 0 | W = 1 |
|---|---|---|
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

TABLE - 2.5

R/M FIELD ENCODING

| MOD = 11 | | | EFFECTIVE ADDRESS CALCULATIONS | | | |
|---|---|---|---|---|---|---|
| R/M | W = 0 | W = 1 | R/M | MOD 00 | MOD 01 D8 means Sign extended | MOD 10 |
| 000 | AL | AX | 000 | (BX) + (SI) | (BX) + (SI) + D8 | (BX) + (SI) + D16 |
| 001 | CL | CX | 001 | (BX) + (DI) | (BX) + (DI) + D8 | (BX) + (DI) + D16 |
| 010 | DL | DX | 010 | (BP) + (SI) | (BP) + (SI) + D8 | (BP) + (SI) + D16 |
| 011 | BL | BX | 011 | (BP) + (DI) | (BP) + (DI) + D8 | (BP) + (DI) + D16 |
| 100 | AH | SP | 100 | (SI) | (SI) + D8 | (SI) + D16 |
| 101 | CH | BP | 101 | (DI) | (DI) + D8 | (DI) + D16 |
| 110 | DH | SI | 110 | Direct Address | (BP) + D8 | (BP) + D16 |
| 111 | BH | DI | 111 | (BX) | (BX) + D8 | (BX) + D16 |

ment values that may be present. The second byte of a two byte immediate value is the most significant.

Some of the instructions from the instruction set given in Appendix-IV are explained here with examples.

<u>The Data Transfer Group</u> included MOV, PUSH, POP, XCHG, IN, OUT, LOAD and STORE instructions.

Example : MOV CX, TOP (BX)

This instruction shall move the contents of TOP + (BX) to the CX register and will occupy 4 bytes.

w = 1, it is a word operation

d = 1, the destination is a register

mod = 10,

DISP = disp. high, disp. low

destination register CX = 001

r/m field = 111

EA = (BX) + (TOP) address

PA = DS * 16 + (BX) + (TOP) address

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 0 0 0 1 0 | 1 | 1 | 1 0 | 0 0 1 | 1 1 1 | DISP | DISP. |
| | d | w | mod field | dest. reg. CX | r/m field | | |

<u>The Arithmetic Group</u> includes ADD, ADC, DEC, INC, MUL, IDIV etc. instructions which can operate on signed and unsigned numbers.

Example : ADI AX , 005F(H)

This is an add immediate the value 005F to AX register and occupies 4 bytes

S : W = 01 16 bits of immediate data from operation

mod = 11,

r/m is register field

r/m = 000, ; destination register (AX) = 000

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 0 0 0 0 0 0 | 0 | 1 | 11 | 000 | 000 | Data | Data if SW=01 |
| | d | w | mod field | dest. reg. | r/m field | | |

The Logical Group includes AND, OR, NOT, ROTATE, SHIFT, TEST instructions

Example : SHL ALPHA

The contents of memory location ALPHA (16 bit value) shift by 'n', where 'n' is the shift count stored in the CX register.

It occupies 4 consecutive bytes

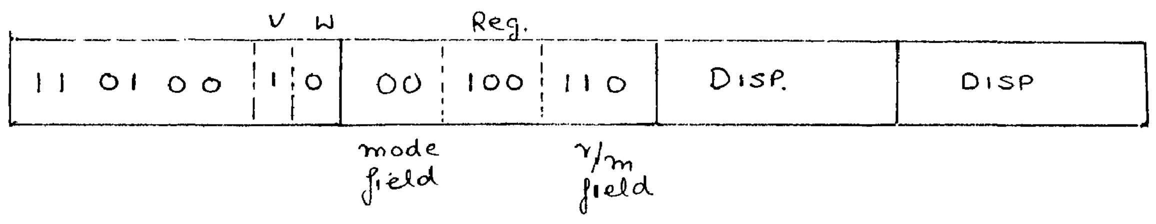
V = 1 as the shifting is to be done 'n' times

W = 1 word operation

mod = 00

r/m = 110

EA = disp. high; disp. low.



<u>The String Manipulation group</u> is a set of very useful instruction used for moving blocks of data. MOVS Fig. (2.9 ) is a single byte instruction with the least significant bit (the w-field) specifying whether a word or a byte is to be moved. The source is picked-up from the address specified in SI register and the destination from the address specified in the DI registers. After transferring the source byte (s) to destination, the SI and DI index registers are incremented/ decremented by 1 or 2 depending upon whether W = 0 or 1. The direction flag DF is used to determine whether the index register are to be decremented (DF = 1 by STD) or incremented (DF = 0 by CLD) after data has been moved. If a REP (single byte) instruction precedes the MOVS instruction the latter is executed repetitively while the CX register remains non zero. Each time the MOVS instruction is executed, CX is decremented by 1, and if not zero, the instruction is executed again. Four more useful instructions for string manipulation are CMPS, SCAS, LODS, STOS.

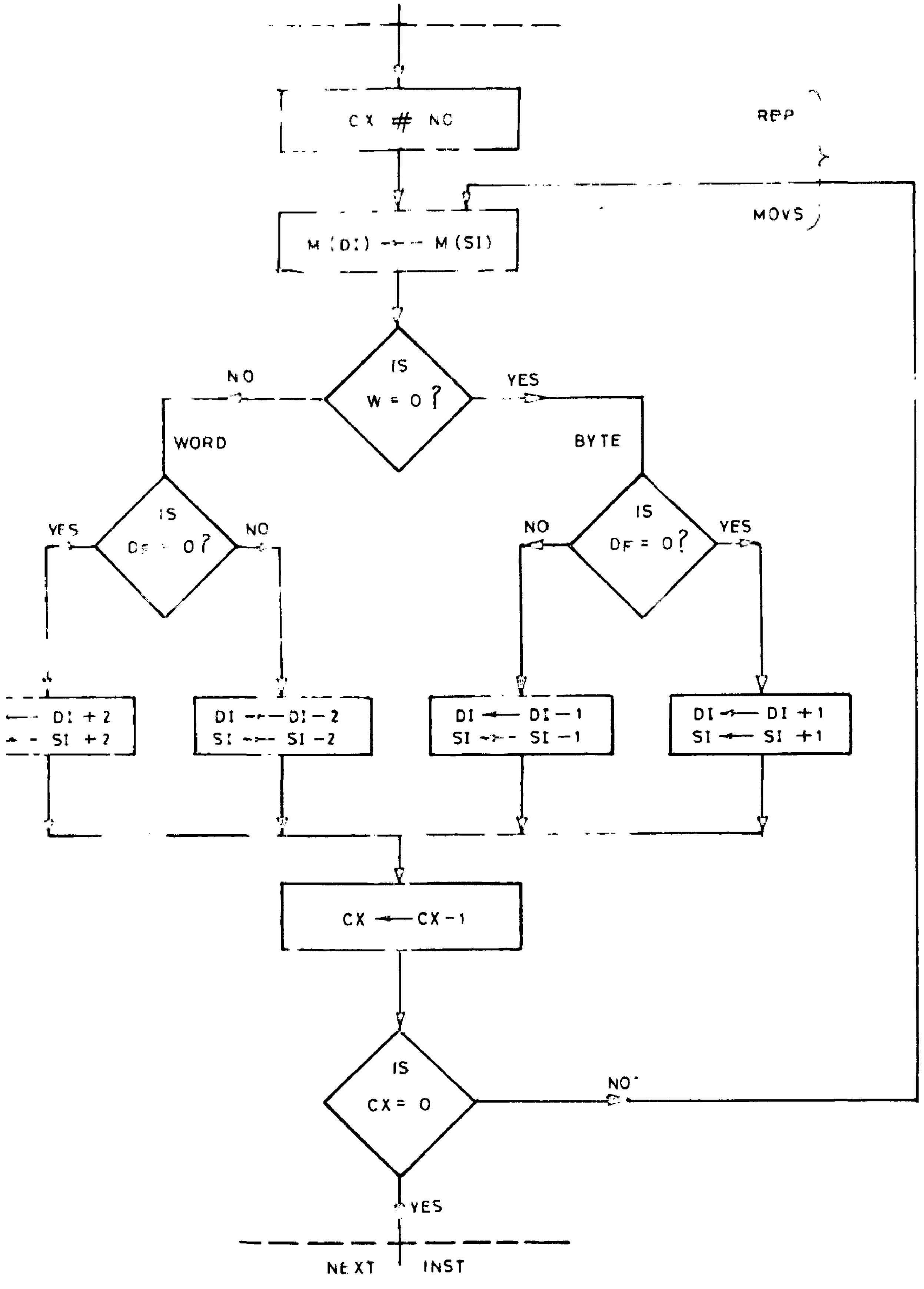


FIG 2 9 STRING MANIPULATION INSTRUCTIONS REP MOVS

```
Example      CLD

             REP.

             MOVS     NEW,     OLD

         W  =  1 ;  transfer word.
```

The words starting from location OLD shall be transferred to locations starting from NEW. The REPeat instruction preceding the MOVS shall repeat till CX $\neq$ 0.

For OLD(EA) = (SI) * 16 + DISP

and NEW(EA) = (DI) * 16 + DISP

The Control Transfer group comprises of CALL, RET, JMP, LOOP, INT instructions.

```
Example :     LOOP     NUMBER
```

Decrement CX by 1 and transfer to NUMBER if CX $\neq$ 0. In this case, the 16 bit displacement is calculated by extending 8 bit displacement. This is a two byte instruction so can only loop $\pm$ 128 bytes from the present location, otherwise use JMP.

The Processor Control group has instructions for carry, direction, and interrupt flags : HALT, WAIT, LOCK and ESC.

Example :

```
LOCK

ADI     NUMBER   FB8C(H)
```

The bus shall be locked till the end of the instruction. No external device shall be able to take over the bus till ADI is executed.

## 2.3 SUMMARY

In this chapter a study of the salient features of Intel 8086 microprocessor has been carried out in considerable detail. This study indicates that the CPU of 8086 is more powerful than any microprocessor previously offered by Intel group. 8086 is totally a new design and has a power set of instructions discussed in Appendix - IV. Memory to memory string operation is available for efficient character data manipulation, hence useful for reducing the complexibility of the program. The various types of addressing modes are useful in solving many problems. The hardwire multiplication and division of signed and unsigned binary numbers are quite powerful instructions.

In the next chapter use is made of the 8086 Assembly language to implement the Digital filter structure described earlier in Chapter I.

# CHAPTER - III

# REALIZATION OF DIGITAL FILTERS USING SECOND ORDER 1D STRUCTURE AS BASIC MODULES

## 3.1 INTRODUCTION

Filters have been implemented in hardwired logic, special purpose computers and general purpose computers. The high speed 16-bit microcomputers with built in multiplication hardware has created a new option for implementing Digital filter [23, 24], with high sampling rate. In this chapter the Intel 8086 microcomputer is used to implement the individual second order 1D module. Also the Intel 8086 is used to implement Digital filters by cascaded and paralleled second order module.

## 3.2 WHY A SECOND ORDER MODULE

The three basic forms for realizing linear Digital filters of the Recursive type are the Direct, Cascade and Parallel forms. As far as the stability question goes the two variations of the Direct form Fig. (1.3) and Fig. (1.5) are entirely equivalent, with the configuration of Fig. (1.3) requiring fewer delay elements. The stability result derived indicates clearly that the coefficient accuracy problem will be by far the most acute for the Direct form realization. For any reasonably complex filter with steep transitions between pass and stop bands the use of Direct form should be avoided.

The choice between the utilization of either the cascade or parallel forms is not clear cut but depends

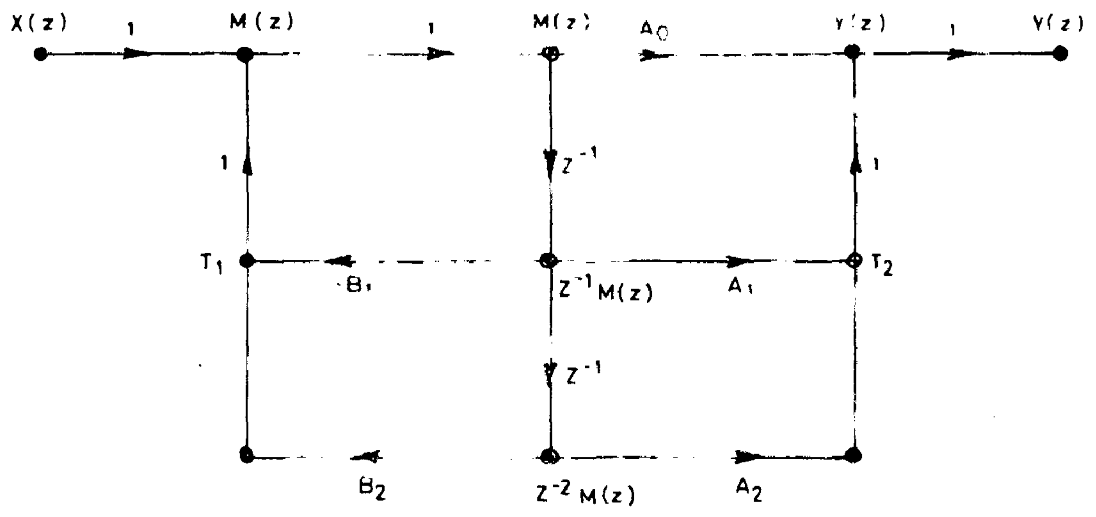


FIG 3.1 SIGNAL FLOW DIAGRAM IN Z-DOMAIN OF A SECOND ORDER DIGITAL FILTER

somewhat on the initial form of the continuous filter and on the digitilization scheme [6] used. In any case the denominator of D(z) must be known in factored form. In order to avoid coefficient sensitivity problems, the transfer function D(z) of equation (1.3) is implemented as a cascade or parallel combination of second order modules.

## 3.3 SECOND ORDER DIGITAL FILTER - 1D STRUCTURE

Second order Digital filter has the form

$$D(z) = \frac{A_0 + A_1 . Z^{-1} + A_2 . Z^{-2}}{1 + B_1 . Z^{-1} + B_2 . Z^{-2}} \quad \ldots \qquad (3.1)$$

and can be represented by any form explained in Section 1.3-1 with N = 2 Fig.(1.7). In this chapter 1D structure is specifically chosen for explanation and implementation using Intel 8086 microprocessor instruction set.

### 3.3-1 MATHEMATICAL DERIVATION

The Transfer function of second order Digital filter is given in equation (3.1). Introducing an intermediate variable M(z)

$$D(z) = \frac{Y(z)}{M(z)} . \frac{M(z)}{X(z)} \quad \ldots \quad \ldots \qquad (3.2)$$

Therefore

$$\frac{Y(z)}{M(z)} . \frac{M(z)}{X(z)} = \frac{A_0 + A_1 . Z^{-1} + A_2 . Z^{-2}}{1 + B_1 . Z^{-1} + B_2 . Z^{-2}} \quad \ldots (3.3)$$

Equating the neumerator and denominator separately

$$\frac{Y(z)}{M(z)} = A_o + A_1.Z^{-1} + A_2.Z^{-2} \quad \ldots \quad (3.4)$$

and $$\frac{X(z)}{M(z)} = 1 + B_1.Z^{-1} + B_2.Z^{-2} \quad \ldots \quad (3.5)$$

From equation (3.4)

$$Y(z) = A_o.M(z) + A_1.Z^{-1}.M(z) + A_2.Z^{-2}.M(z) \quad \ldots \quad (3.6)$$

and from equation (3.5)

$$M(z) = X(z) - B_1.Z^{-1}.M(z) - B_2.Z^{-2}.M(z) \quad \ldots \quad (3.7)$$

Equations (3.6) and (3.7) in time domain are

$$y(k) = A_o.m(k) + A_1.m(k-1) + A_2.m(k-2) \quad \ldots \quad (3.8)$$

$$m(k) = x(k) \quad - B_1.m(k-1) - B_2.m(k-2) \quad \ldots \quad (3.9)$$

Equations (3.8) and (3.9) clearly show that m(k) is to be found out before y(k) is calculated.

Let $T_1 = -B_1.m(k-1) - B_2.m(k-2) \quad \ldots \quad (3.10)$

and $T_2 = A_1.m(k-1) + A_2.m(k-2) \quad \ldots \quad (3.11)$

Equations (3.8) and (3.9) becomes

$$y(k) = A_o.m(k) + T_2 \quad \ldots \quad (3.12)$$

$$m(k) = x(k) + T_1 \quad \ldots \quad (3.13)$$

Equations (3.6) through (3.13) define first direct structure, 1D, for second order Digital filter. Fig. (3.1) is the signal flow diagram in Z domain for the second order Digital filter in 1D form, and uses these equations. This signal flow diagram can be used to implement, through hardwired logic a second order Digital filter using summers, multipliers and two delay elements [4,8].

It is to be noted that the intermediate variable $T_1$ and $T_2$ depend on the previous samples and therefore can be evaluated in the interval $(K-1)T < t < KT$ and shall be available before KT sampling point. The output $y(k)$ may rapidly be calculated upon the receipt of input $x(k)$.

## 3.3-2 ALGORITHM FOR IMPLEMENTATION

Equations (3.6) through (3.13) can also be used for obtaining the algorithm for the implementation in a microprocessor. The information can be grouped as follows :

| | | |
|---|---|---|
| OUTPUT | $m(k) = x(k) + T_1$ | ... (3.13) |
| | $y(k) = A_0.m(k) + T_2$ | ... (3.12) |
| DELAY | $m(k-2) \longleftarrow m(k-1)$ | |
| | $m(k-1) \longleftarrow m(k)$ | |
| PRECALCULATIONS | $T_1 = -B_1.m(k-1) - B_2.m(k-2)$ | ... (3.10) |
| in the interval $(K-1)T < t < KT$ | $T_2 = A_1.m(k-1) + A_2.m(k-2)$ | ... (3.11) |

It is to be noted that the above algorithm when implemented gives the maximum sampling rate possible in a microprocessor.

## 3.3-3 SOFTWARE PROGRAM USING INTEL 8086 INSTRUCTION SET

The flow chart representing the process of 1D Structure derived above is shown in Fig. (3.2). Steps involved are

| | | |
|---|---|---|
| (1) | Initialization | ; A/D, D/A converters and all variables. |
| (2) | Input X(k) | ; From A/D converter. |
| (3) | Compute MØ and Y(k) | |
| (4) | Output Y(k) | ; To D/A converter. |
| (5) | Perform Time Delay | |
| (6) | Compute $T_1$ and $T_2$ | ; Precalculation of $T_1$ & $T_2$. |

The Assembly language software program is given in PROGRAM-3.1. The following salient features of the software program written are to be noted.

1. Input/Output are connected through A/D and D/A converters for Memory Mapped I/O operations.

2. The value of X(k) is inputed from A/D converter through CPU initiated Polled I/O transfer.

3. The value of the constants $A_0$, $A_1$, $A_2$, $B_1$ and $B_2$ are stored as half values. This is explained in detail in Appendix - V. Thus the VALUE STORED = $\lfloor$'Value' * $2^{14}$ + 0.5$\rfloor$ where $\lfloor x \rfloor$ means largest integer smaller than or equal to X. 'Value' in the paranthesis is the value of the constant which is assumed to lie between -1 and 2.

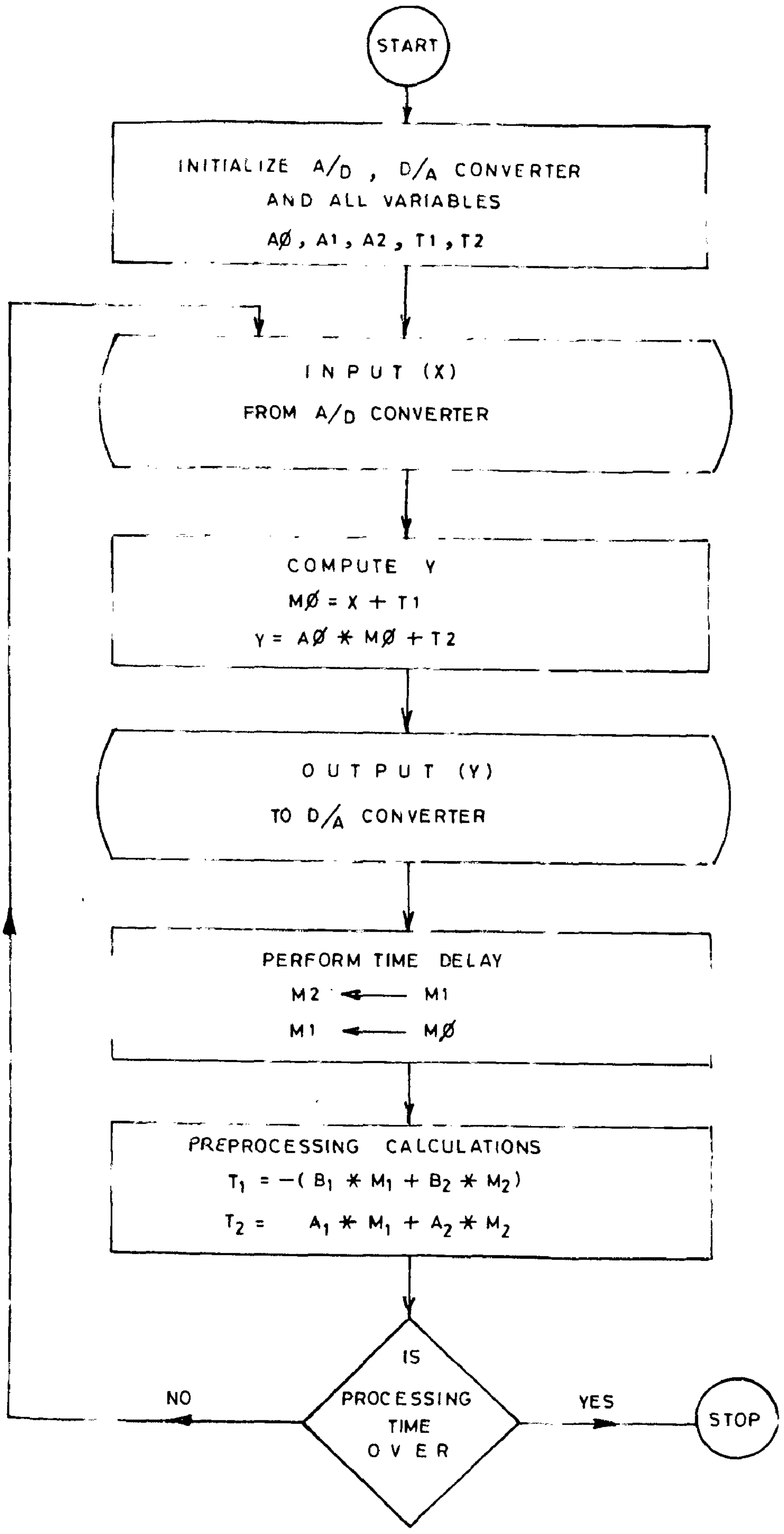


FIG. 3. 2 FLOW CHART OF SECOND ORDER MODULE - ID

4. This program is written exclusively for second order Digital filter. Once started it continues to sample input X(k) at maximum sampling rate and outputs Y(k) immediately thereafter, this continues till the processor is instructed to stop. This is done through an input PORT4.

PROGRAM - 3.1

FILTER SECOND ORDER 1D STRUCTURE

```
; INITIALIZATION CLEAR MØ, M1, M2, T1 AND T2

           CLD
           MOV     AX , # Ø   ; CLEAR ACCUMULATOR
           MOV     CX , # 6   ; STORE 6 * N IN CX REGISTER
           LEA     DI , MØ    ; DI POINTS TOWARDS ADDR LOC M1Ø
           REP                ; STORES CONTENTS OF AX
         • STOS W             ; IN LOCATIONS

; INPUT X FROM A/D CONVERTER THROUGH POLLED I/O TRANSFER
; A/D CONNECTED FOR MEMORY MAPPED I/O OPERATION

CONT :     MOV     PORT3 , AX ; ISSUE START CONVERSION PULSE
                              ; TO A/D CONVERTER
IN-LP :    MOV     AX , PORT2 ; READ 'BUSY' SIGNAL FROM A/D
           AND     AX , 8ØØØ(H)
           JZ      IN-LP      ; WAIT UNTIL READY
           MOV     AX , PORT1 ; X IS NOW IN A/D CONVERTER

; COMPUTE OUTPUT SAMPLE Y
; NOTE THAT ADJUSTMENTS ARE NECESSARY SINCE CONSTANTS
; ARE STORED AS HALF VALUES

OUTP-1D :  ADD     AX , T1    ; MØ IS NOW IN AX : X + T1
           MOV     MØ , AX    ; STORE MØ IN ITS LOCATION
           IMUL    AØ         ; MØ * AØ/2 IN DX,AX
           SAL     DX , 1     ; MØ * AØ IN DX
           ADD     DX,, T2    ; AØ * MØ+T2 IN DX = Y
           MOV     AX , DX    ; Y IN AX

; OUTPUT Y IN AX TO OUT PORTØ , PORTØ BEING THE ADDRESS
; ASSIGNED TO D/A CONVERTER IN MEMORY MAPPED I/O MODE

           MOV     PORTØ , AX
```

```
; PERFORM DELAY OPERATION SO THAT M2 ←  M1 AND M1 ←  MØ.

DELAY-1D : LEA  DI , T1-2        ; DI POINTS TO M2
           LEA  SI , M2-2        ; SI POINTS TO M1
           MOV  CX , ## 2        ; COUNT DATA MOVE
           STD                   ; SET D FLAG FOR AUTODECREMENT
           REP
           MOVS
           CLD                   ; CLEARS D FLAG FOR AUTOINCREMENT

; PREPROCESSING CALCULATIONS BEGINS HERE TO CALCULATE T1 AND T2
; T1 = -(B1 * M1 + B2 * M2), T2 = A1 * M1 + A2 * M2

PRE-1D :   LEA  SI , A1          ; SI POINTS TO COEFF A1
           LODW                  ; A1/2 IS LOADED TO AX AND
                                 ; SI ← SI + 2
           IMUL M1               ; A1 * M1/2 IN DX, AX
           MOV  BX , DX          ; SAVE A1 * M1/2 IN BX
           LODW                  ; A2/2 IS NOW LOADED TO AX
                                 ; AND SI ← SI + 2
           IMUL M2               ; A2 * M2/2 IN DX, AX
           ADD  BX , DX          ; T2/2 IS NOW IN BX
           SAL  BX , 1           ; T2 IS NOW IN BX
           MOV  T2 , BX          ; STORE NEW VALUE OF T2
           LODW                  ; B1/2 IS NOW IN AX AND SI ← SI+2
           IMUL M1               ; B1 * M1/2 IS IN DX, AX
           MOV  BX , DX          ; SAVE B1 * M1/2 IN BX
           LODW                  ; B2/2 IS IN AX NOW
           IMUL M2               ; M2 * B2/2 IN DX, AX
           ADD  BX , DX          ; -T1/2 IS IN BX
           SAL  BX , 1           ; BX THEN CONTAINS -T.
           NOT  BX               ; NOT AND INC INSTRUCTIONS
           INC  BX               ; TOGETHER NEGATES BX, SO T1 IN BX
           MOV  T1 , BX          ; STORE NEW VALUE IN T1

; IT IS ASSUMED INPUT DEVICE PORT4 SHALL CONTAIN NO FFFF IF
; PROCESS CONTINUES, OTHERWISE STOP.

           MOV  AX , PORT4       ;
           NOT  AX
           JZ   CONT
           HALT
```

## 3.4 CASCADE STRUCTURE OF K'TH ORDER DIGITAL FILTER — N SECOND ORDER 1D MODULES IN CASCADE

In order to avoid coefficient sensitivity problems, the Digital filter Transfer function is implemented using a cascade of second order modules.

$$D(z) = \frac{\sum_{i=1}^{N} (A_{oi} + A_{1i}.Z^{-1} + A_{2i}.Z^{-2})}{\sum_{i=1}^{N} (1 + B_{1i}.Z^{-1} + B_{2i}.Z^{-2})} \quad \ldots \quad (3.14)$$

where N is the smallest integer greater than or equal to K/2. If the numerator and denominator factors are paired and the modules ordered in cascade, then

$$D(z) = \sum_{i=1}^{N} H_i(z) \quad \ldots \quad (3.15)$$

where $$H_i(z) = \frac{A_{oi} + A_{1i}.Z^{-1} + A_{2i}.Z^{-2}}{1 + B_{1i}.Z^{-1} + B_{2i}.Z^{-2}} \quad \ldots \quad (3.16)$$

Equations (3.14) and (3.15) are the same as discussed in Section (1.3-2). The problems encountered in pairing and ordering in cascaded second order modules has been extensively studied in the literature [26,27,28] which provides guidelines for designing filters.

The cascaded block diagram in Z-domain for equation (3.1) is shown in Fig. (3.3). The signal flow diagram for i'th cascaded block is shown in Fig. (3.4), this is similar to

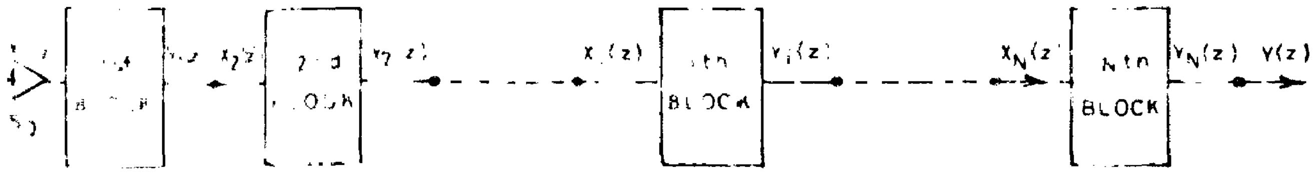


FIG 3 3  BLOCK DIAGRAM OF N-STAGES IN CASCADE

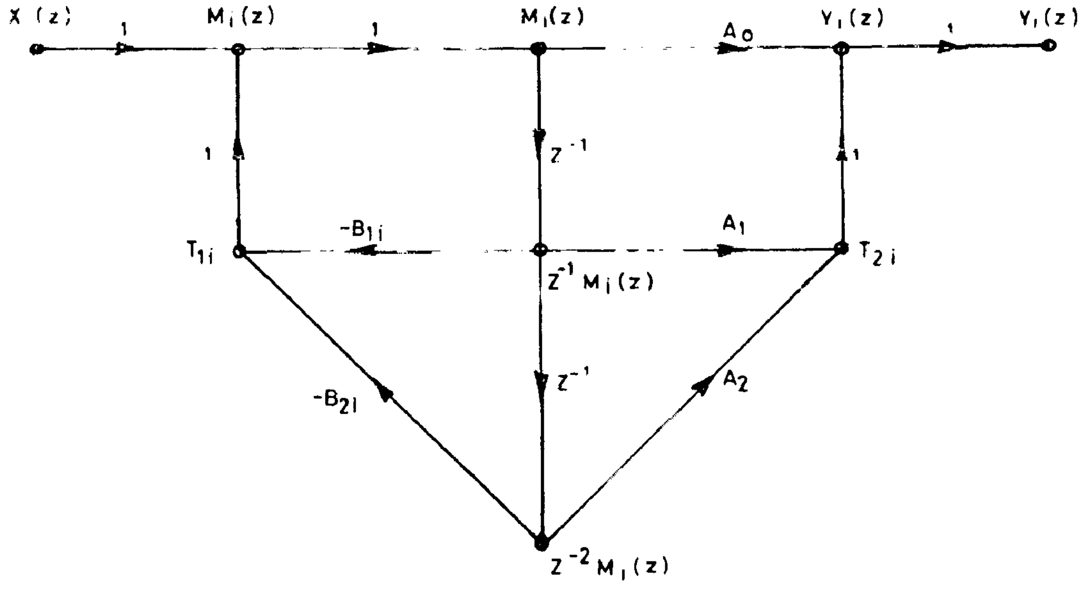


FIG.3 4  SIGNAL FLOW DIAGRAM IN Z- DOMAIN FOR ith CASCADED BLOCK

Fig. (3.1) except for introducing 'i' for i'th block identification, which implements equation (3.16).

### 3.4-1 ALGORITHM FOR I'TH MODULE

Using the equations derived from Section (3.3), the following equations can be written for i'th stage.

OUTPUT : $m_i(k) = x_i(k) + T_{1i}$ ... (3.17)

$y_i(k) = A_{oi}.m_i(k) + T_{2i}$ ... (3.18)

DELAY : $m_i(k-2) \leftarrow m_i(k-1)$ ... (3.19)

$m_i(k-1) \leftarrow m_i(k)$ ... (3.20)

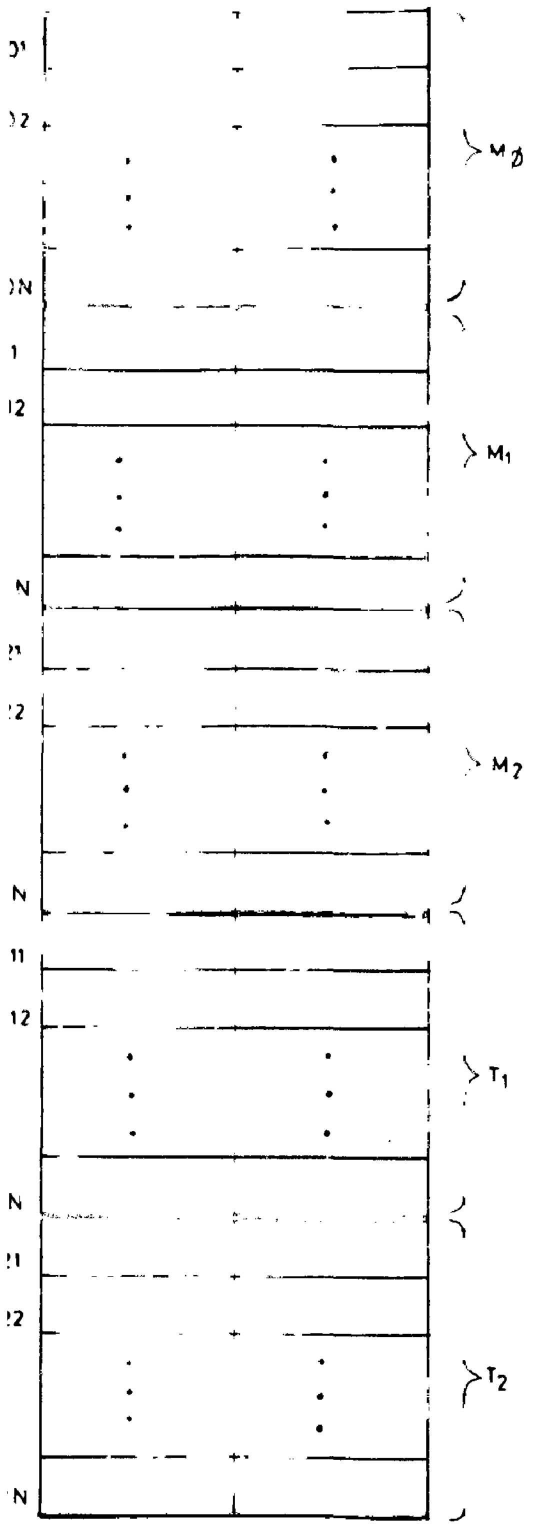
PRECALCULATION: $T_{1i} = -B_{1i}.m_i(k-1) - B_{2i}.m_i(k-2)$ ..(3.21)

$T_{2i} = A_{1i}.m_i(k-1) + B_{2i}.m_i(k-2)$ ,,(3.22)

These equations are valid for all $i = 1$ to N.

### 3.4-2 MEMORY ORGANIZATION

From equations (3.17) through (3.22) for all $i = 1$ to $N$ it is obvious that coefficients ($A_{oi}$, $A_{1i}$ etc.), delayed value of the intermediate variables ($m_i(k)$, $m_i(k-1)$ etc.) and temporary storage variables ($T_{1i}$, $T_{2i}$ etc.) are to be stored in the RAM memory interfaced with Intel 8086 microprocessor. They are to be arranged in a particular way so that the String Manipulation instructions can be effectively used. The arrangement is shown in Fig. (3.5).



3 5 MEMORY ORGANIZATION OF VARIABLES

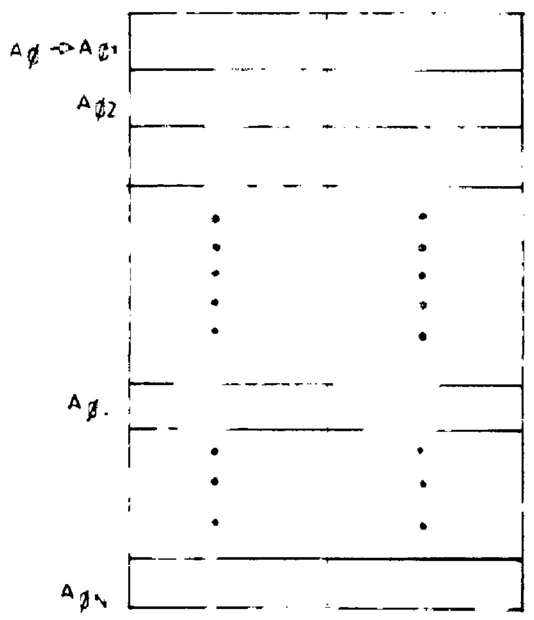


FIG 3 6(a) MEMORY ORGANIZATION OF CONSTANTS $A_{0i}$ COEFFICIENTS

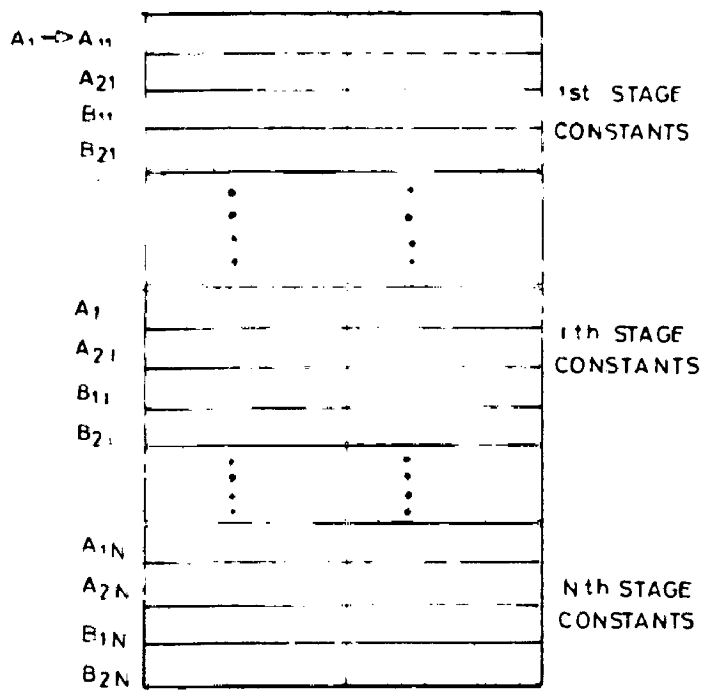


FIG.3.6(b) MEMORY ORGANIZATION OF CONSTANTS COEFFICIENTS $A_{1i}$, $A_{2i}$, $B_{1i}$, $B_{2i}$

The displacement variable MØ initially points towards MØ1 with index zero. After performing the desired calculation the index is incremented by two and by the index addressing modes available, the pointer is changed to MØ2. Thus MØ pointer points to $m_i(k)$ of all the cascaded stages, i = 1 to . Similarly, M1 is the displacement pointer initially pointed towards $m_i(k-1)$ of all the stages, i = 1 to N. This follows immediately after N values of MØ. Soon after N values of $m_i(k-1)$, the storage of second delay values $m_i(k-2)$ should start. M2 is the displacement pointer address $m_i(k-2)$, i = 1 tó N. Thus (M2 - 2) gives the address of the last location of first delay storage which stores $m_N(k-1)$. T1 is the pointer for temporary storage $T_{1i}$, i = 1 to N and this follows soon after the second delay storage values. Thus T1 - 2 gives the address of the last location of second delay storage which stores $m_N(k-2)$. After all $T_{1i}$ are stored, $T_{2i}$ variables are stored consecutively starting from T2 displacement address.

Fig. 3.6 gives the memory organisation for constant coefficients. These are stored as half values as explained in Appendix - V.

Note that by properly loading SI and DI registers with proper starting indices, the pointer displacement address can be used along with indexed addressing modes to identify any address in the corresponding pointer blocks. For example, AØ pointer address can point to any address AØ1 to AØN. Similarly A1 pointer address can address all the constants of N stages (Fig. 3.6b). Similarly, M2 pointer address can be used along with index addressing modes to identify all the addresses from M21(=M2) to M2N.

## 3.4-3 IDENTIFICATION OF DIFFERENT SUBROUTINES

The operations involved in the cascaded modules can be broken up into different parts giving rise to the following subroutines.

(1) SUBROUTINE INITIALIZATION called INIT-1D.

This initializes $m_{oi}$, $m_{1i}$, $m_{2i}$, $T_{1i}$, $T_{2i}$ locations by clearing all the memory locations given in Fig. (3.5).

(2) SUBROUTINE INPUT called INP-1D.

As explained earlier CPU initiated Polled I/O transfer is used for inputting X(k) and storing it in AX register through Memory Mapped I/O connection.

(3) SUBROUTINE OUTPUT called OUT-1D.

This calculates equations (3.17) and (3.18) for all i = 1 to N. This subroutine is entered only after passing X(k) value in AX register and the number of stages N in CX register. The calculated value Y(k) of the last stage is returned in AX register when the subroutine is executed.

(4) SUBROUTINE DELAY called DEL-1D.

This subroutine implements the transfers given in equations (3.19) and (3.20) for all i = 1 to N. The String Manipulation block move instructions (REP MOVS) is very useful here.

(5) SUBROUTINE PREPROCESSING called PRE-1D..

This subroutine calculates all the temporary storage values $T_{1i}$ and $T_{2i}$, i = 1 to N for each sampling period and updates the information during $(K-1) < t < KT$. Again String Manipulation instructions and LOOP instruction simplifies the software program to a very great extent.

## 3.4-4 SUBROUTINE FOR INITIALIZATION

The R.T.L. (Register Transfer Logic) flow chart for initialization subroutine is shown in Fig. 3.7, and the corresponding subroutine program is given in PROGRAM - 3.2. In PROGRAM - 3.2 the String Manipulation instruction STOSW along with REPeat instruction is used to implement the last three blocks of the flow chart. Thus, REP STOSW clears all the memory locations MØ to MØN, M1 to M1N, M2 to M2N, T1 to T1N and T2 to T2N. One more block of N word locations will also be cleared as 6*N has been stored in CX register N should be known, and 6*N should be loaded into the CX register before clearing the memory locations.

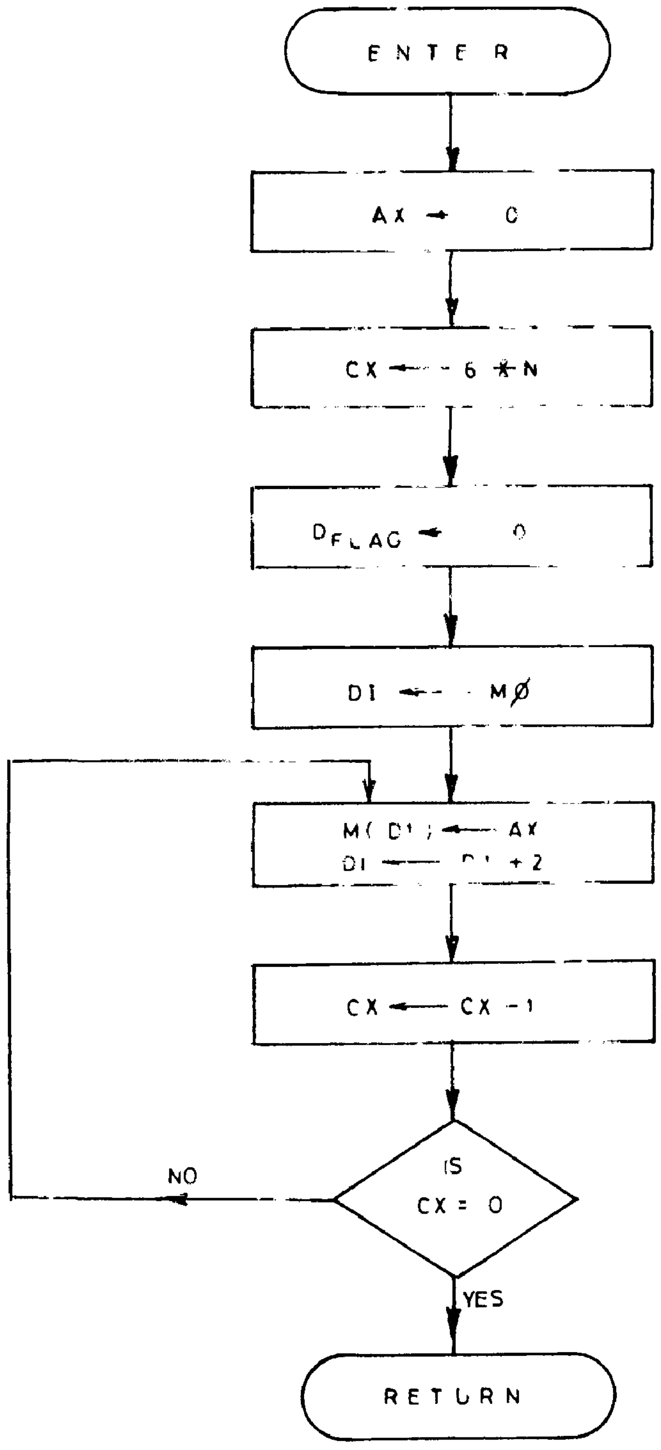


FIG.3.7 R.T.L. FLOW CHART OF SUBROUTINE INITILIZATION-ID

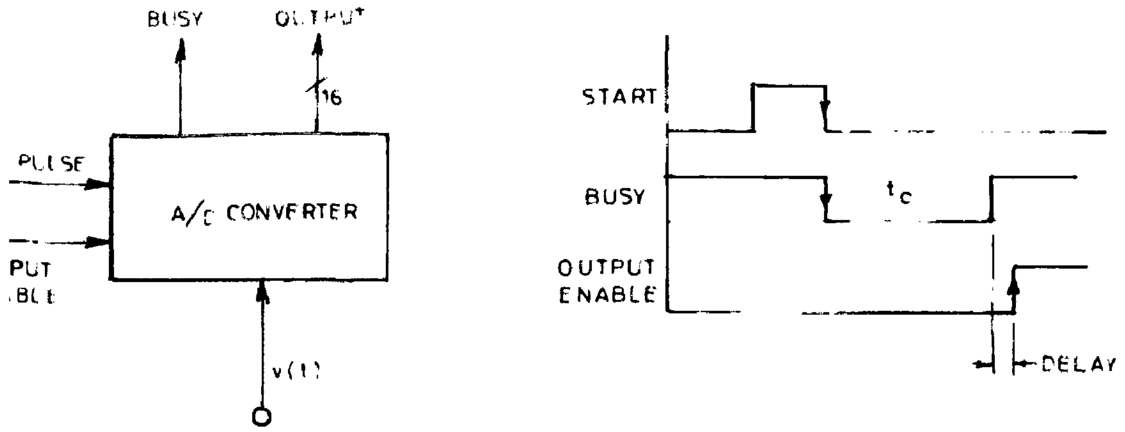
PROGRAM - 3.2

```
INIT-1D    :    MOV    AX , # Ø      ;  CLEAR ACCUMULATOR
                MOV    CX , # 6*N    ;  STORE 6*N IN CX
                CLD
                LEA    DI , MØ       ;  DI POINTS TOWARDS M1Ø
                REP
                STOSW                ;  STORES CONTENTS OF AX
                                     ;  IN LOCATIONS
                RET
```

3.4-5 <u>SUBROUTINE INPUT - 1D</u>

Here, it is assumed that the I/O operation is Memory Mapped, i.e., I/O devices may be placed in the memory space. An advantage of Memory-Mapped I/O is that it provides additional programming flexibility.

A/D converters are devices that convert analog input data into digital form. The block diagram of a tristate A/D converter is shown in Fig. (3.8a). The analog input voltage is converted into its 16-bit equivalent digital output. The output appears at the OUTPUT terminals only when Output-Enable goes LOW from HIGH. When Output-Enable is LOW the 16 output terminals are in tristate condition. Start-pulse is a control input terminal, when it is LOW the A/D converter is dead — not working and when this Start-pulse goes from HIGH to LOW the A/D converter starts the conversion process. The A/D conversion is not instantaneous and takes some time.



(a) Block diagram (b) Timing diagram

FIG 3 8 A/D CONVERTER

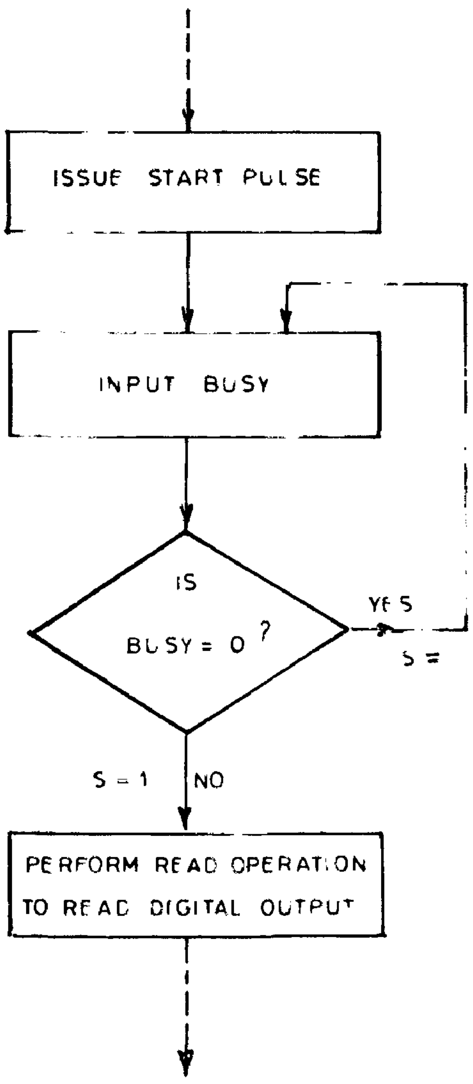


FIG. 3 9 A10 OPERATION IN POLLED I/O CONDITION

During the conversion process the A/D converter is said to be BUSY and is indicated by Busy-control output signal. This BUSY is normally HIGH goes LOW at the start of A/D conversion, remains LOW for '$t_c$' sec., till the conversion is complete and the required data is ready for transfer.

To perform the operations as in Fig. (3.9) different signals of Fig. (3.8b) are to be issued by proper interfacing of Fig. (3.8a) of A/D converter with the given microprocessor This is shown in Fig. (3.10) and the corresponding subroutine program is given in PROGRAM - 3.3. Twenty-bit address bus is got by making use of 8282 latches (3 Nos.). The ALE issued out of microprocessor latches the address in first ($T_1$) state The 8282 propogates the address through to the outputs while ALE is high and latches the address on the falling edge of ALE.

PROGRAM - 3.3

```
INP-1D :     MOV   PORT3, AX      ;  ISSUE START CONVERSION PULSE
IN-LP  :     MOV   AX , PORT2     ;  READ BUSY SIGNAL FROM A/D
             AND   AX , 8000(H)
             JZ    IN-LP          ;  WAIT UNTIL READY
             MOV   AX , PORT1     ;  X IS NOW IN A/D
             RET
```

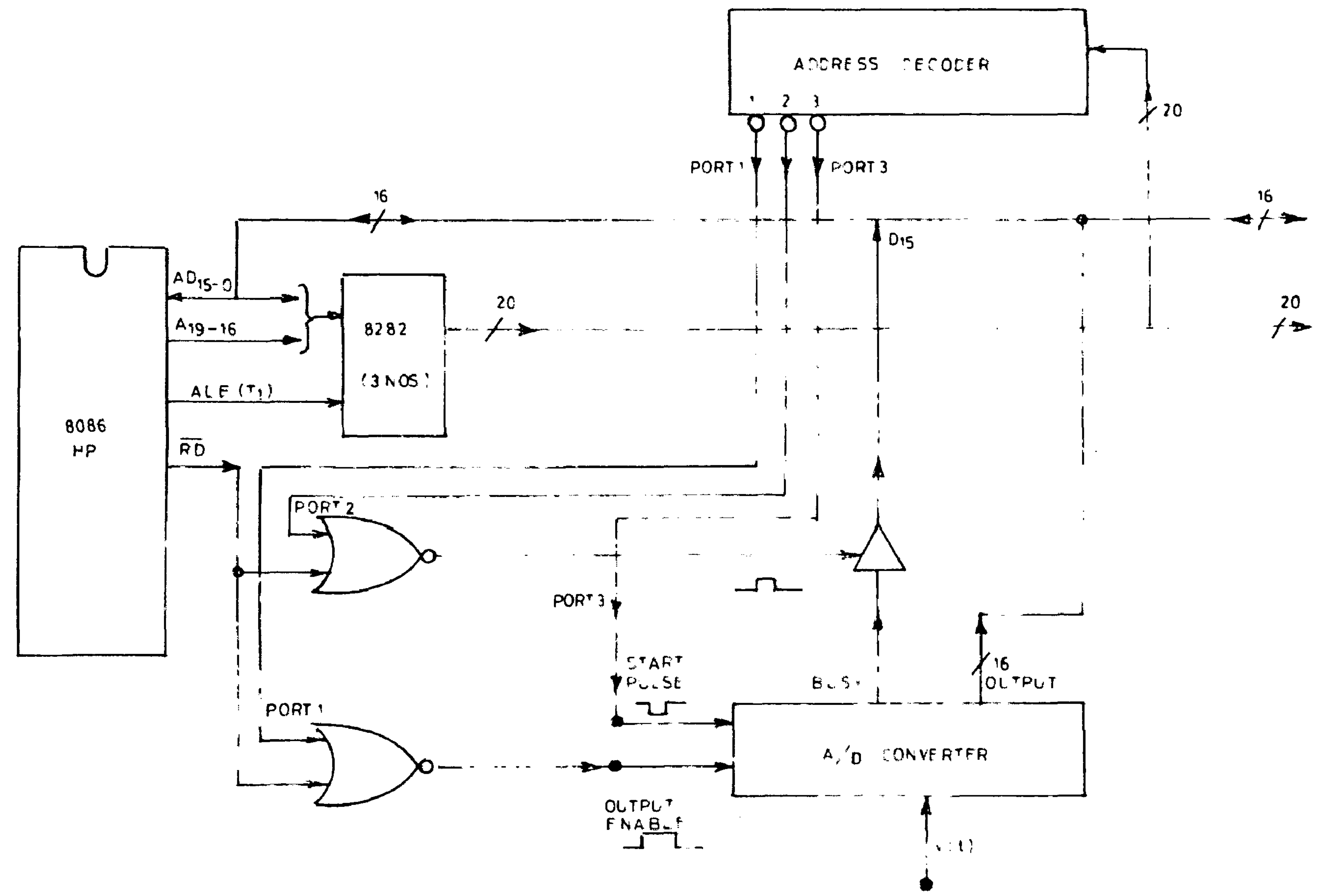


FIG 3.10 INTERFACING CIRCUIT FOR A/D CONVERTER

## 3.4-6 SUBROUTINE OUTPUT - 1D

The R.T.L. flowchart is shown in Fig. (3.11) and the corresponding program is given in PROGRAM - 3.4. The following points are to be noted while reading the flowchart.

(1) X(K) is passed in AX register before entering this subroutine.

(2) N, the number of cascades second order modules, is passed in CX register before entering the subroutine.

(3) The coefficients are assumed to be arranged as shown in Fig. (3.6a) and stored as half values as explained in Appendix - V.

(4) MØ, M1, M2, T1 and T2 pointers points to the first address of the corresponding block, Sixteen bit operations are assumed.

PROGRAM - 3.4

```
OUT-1D :   MOV  SI , # Ø       ; STAGE INDEX
           LEA  DI , MØ        ; M(K) POINTER POINTS TO FIRST
                               ; ADDRESS
           CLD
OLP-1D :   ADD  AX , T1 [SI]   ; MØ is NOW IN AX
           STOSW               ; STORE IN MØ LOCATION
                               ; DI ← DI + 2
           IMUL AØ [SI]        ; MØ * AØ/2 IN DX, AX
           SAL  DX , 1         ; TRUNCATE AND MULTIPLY BY 2
                               ; TO GET MØ * AØ IN DX
           ADD  DX , T2 [SI]   ; Y(K) IN DX
           MOV  AX , DX        ; Y(K) NOW IN AX READY FOR
                               ; NEXT STAGE
           ADD  SI, # 2        ; MOVE INDEX TO POINT NEXT
                               ; STAGE LOCATIONS
           LOOP OLP-1D         ; LOOP BACK TO CALCULATE
                               ; NEXT STAGE
           RET
```

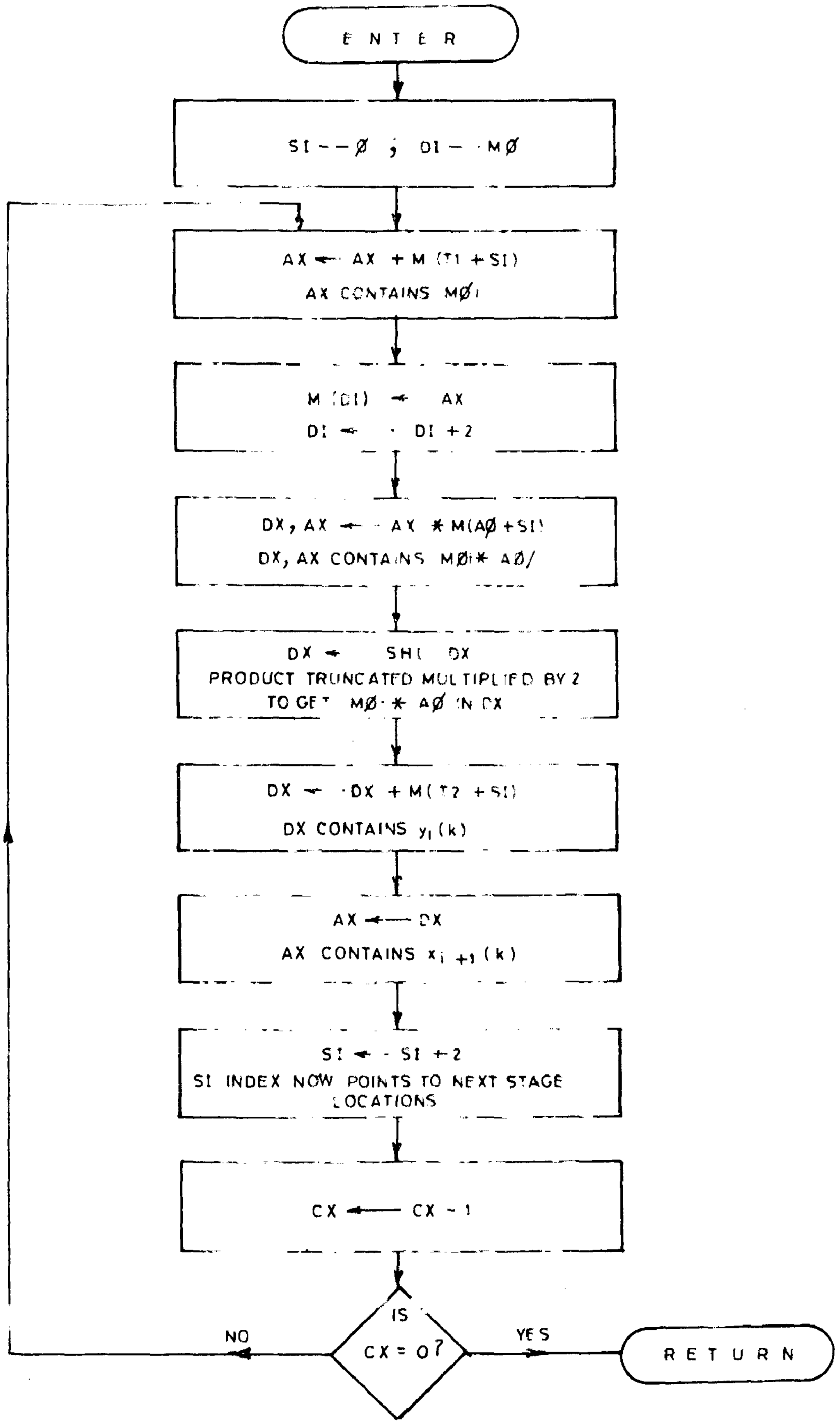


FIG. 3.11 R.T.L. FLOW CHART FOR OUTPUT-1D SUBROUTINE

### 3.4-7 SUBROUTINE DELAY - 1D

The R.T.L. flowchart for this subroutine is shown in Fig. (3.12) and the corresponding program in PROGRAM - 3.5. The following points must be noted.

(1) MØ, M1, M2, T1 & T2 displacement addresses points to the first address of each block as given in Fig. (3.5). From Fig. (3.5) it is clear that M2-2 points to M1N namely m(k-1) of the N'th stage and T1-2 points to m(k-2) of the last stage (≐ M2N).

(2) N, the number of cascade stages of second order modules is passed in CX register before entering the delay subroutine.

(3) The String Manipulation block move instructions alongwith REPeat instruction performs the complete transfer operations so that all $m_i(k-1)$ are transferred to $m_i(k-2)$ locations and thereafter all $m_i(k)$ are transferred to $m_i(k-1)$ locations. Thus, 2N locations are to be transferred from one block to the other. Hence, the count in CX register must be multiplied by 2 before executing the block move instructions.

PROGRAM - 3.5

```
; CX CONTAINS N NOS OF CASCADED STAGES BEFORE ENTERING
DEL-1D :    LEA  DI , (T1-2)  ; POINTS TO M2
            LEA  SI , (M2-2)  ; POINTS TO M1
            STD               ; SETS DIR.FLAG FOR AUTODECREMENT
            SAL  CX , 1       ; DOUBLE LOOPCOUNT FOR TWO MOVS
            REP               ; PERFORMS
            MOVS              ; BLOCK MOVE OPERATION
            RET
```

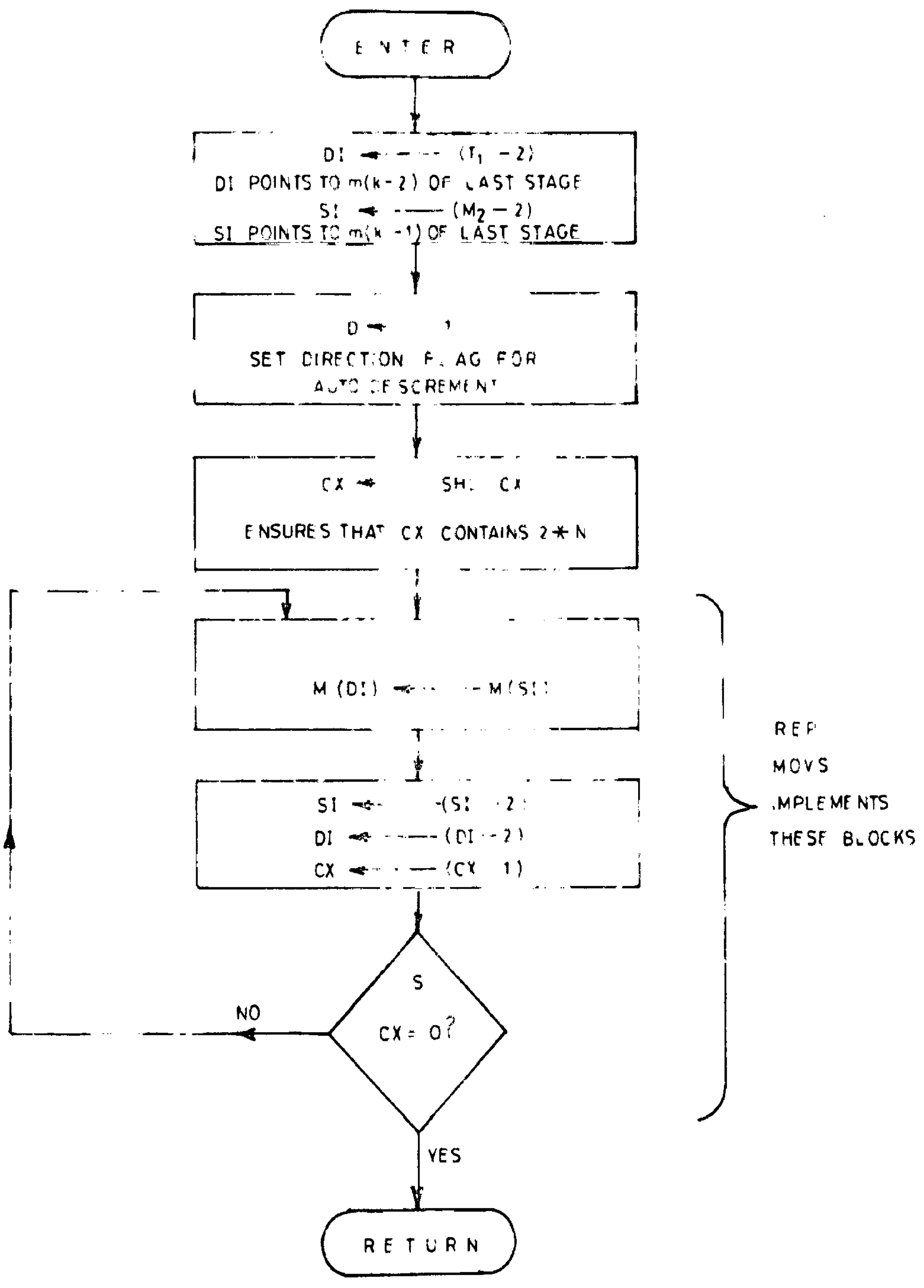

ENTER
DI ← (T1 − 2)
DI POINTS TO m(k−2) OF LAST STAGE
SI ← (M2 − 2)
SI POINTS TO m(k−1) OF LAST STAGE
D ← 1
SET DIRECTION FLAG FOR AUTO DESCREMENT
CX ← SHL CX
ENSURES THAT CX CONTAINS 2*N
M(DI) ← M(SI)
SI ← (SI − 2)
DI ← (DI − 2)
CX ← (CX − 1)
S
CX = 0?
NO
YES
RETURN
REP MOVS IMPLEMENTS THESE BLOCKS

FIG 3.12 R.T.L FLOW CHART FOR DELAY-1D SUBROUTINE

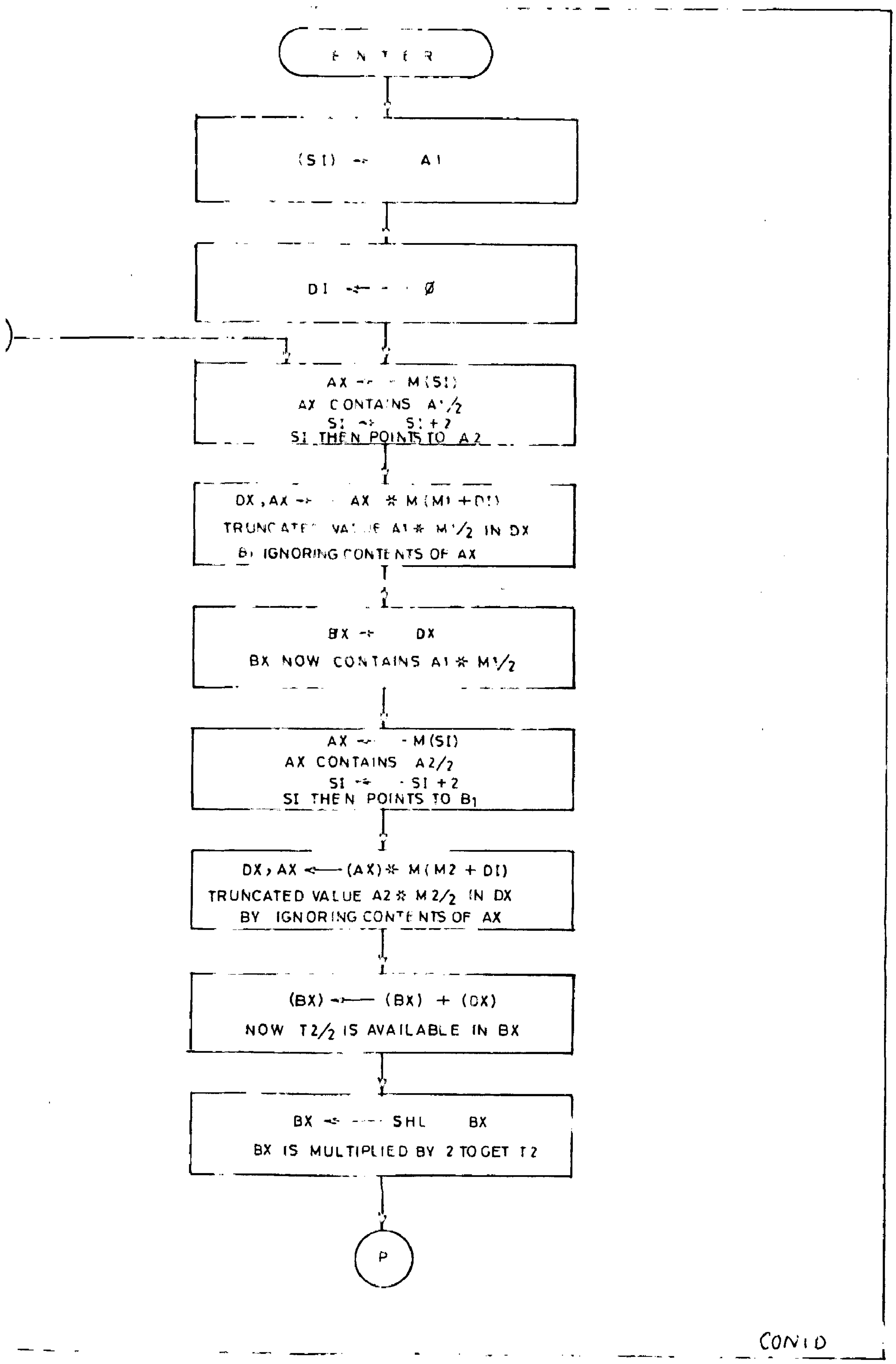
3.4-8 SUBROUTINE PROCESSING - 1D

The R.T.L. flowchart is shown in Fig. (3.13) and the corresponding program in PROGRAM - 3.6. As before, N, number of cascaded stages should be passed in CX before entering th subroutine. Also M1, M2, T1, T2 displacement addresses points to the first address of each block as given in Fig. (3.5

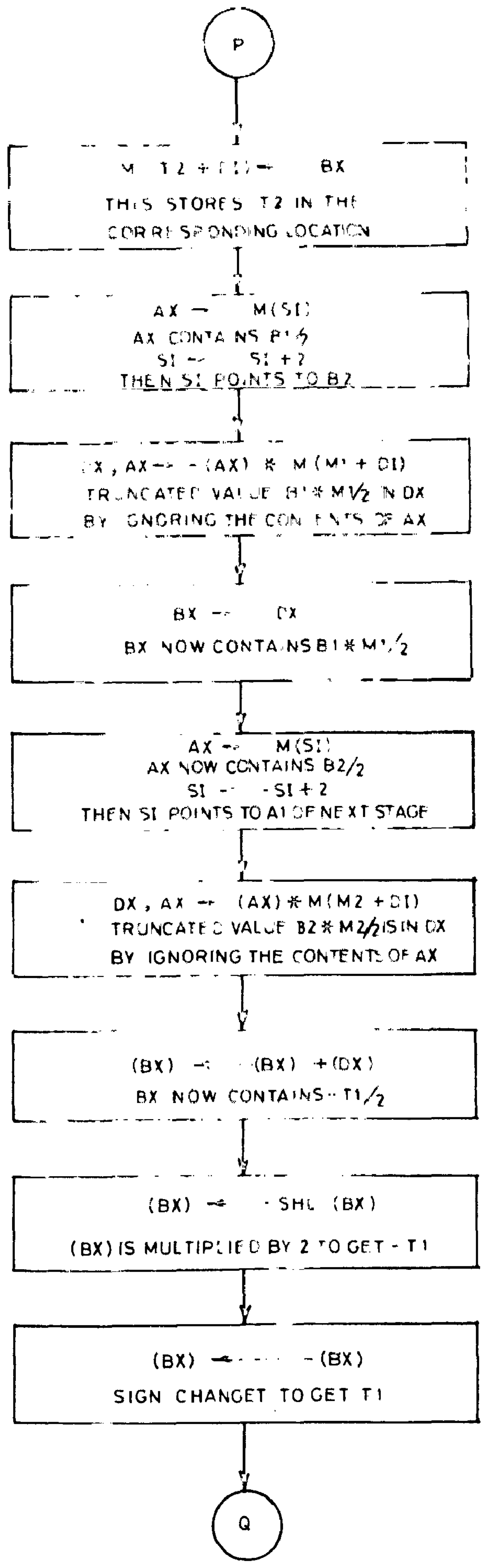
A1 displacement address points to the first address of the coefficients, A11, A21, B11, B21 etc., as shown in Fig. (3.6b). The coefficients are stored in these locations as half values explained in Appendix - V. Again the use of String Manipulation instructions and LOOP instruction simplifies writing the Assembly language Program shown in PROGRAM - 3.6.

PROGRAM - 3.6

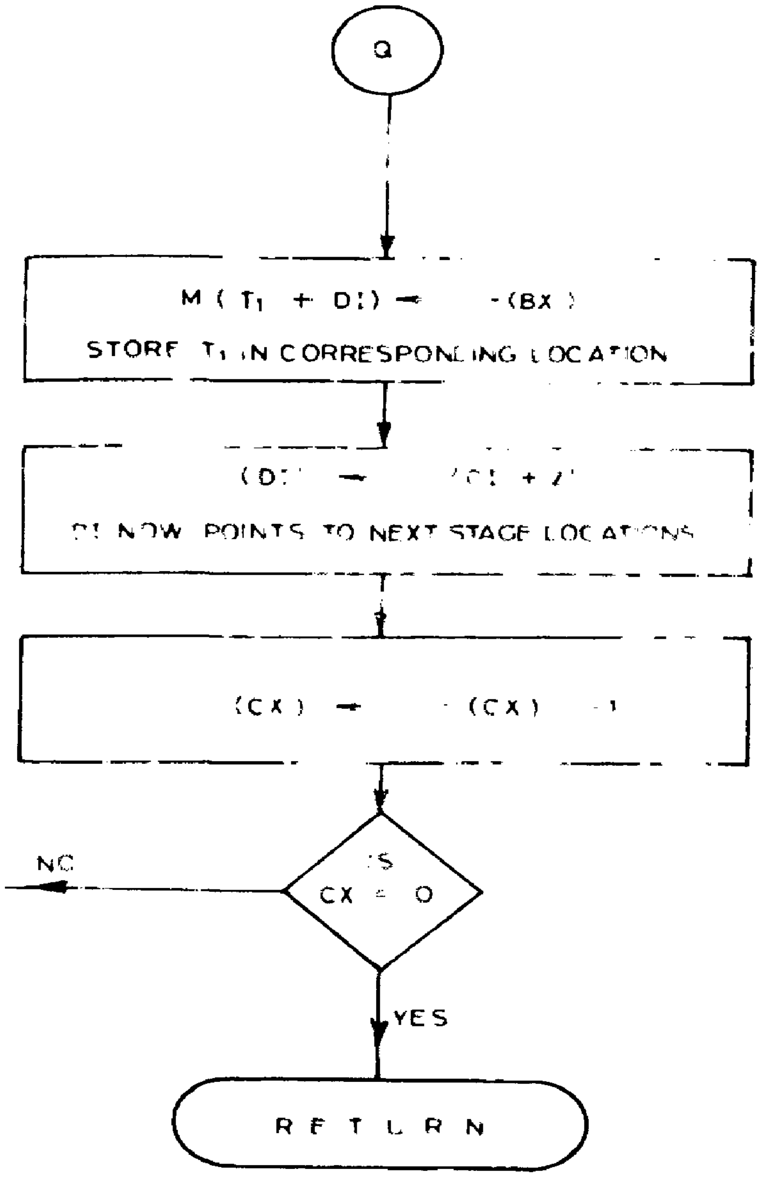
```
PRE-1D : LEA   SI , A1         ; POINTS TO FIG. 3.6(b) COEFS
         MOV   DI , # Ø        ; INDEX TO POINT CURRENT STAGE
                               ; CALCULATIONS
         CLD
PLP-1D : LODSW                 ; A1/2 IS NOW IN AX AND SI ← SI +2
         IMUL  M1 [DI]         ; A1 * M1/2 IN DX , AX
         MOV   BX ,DX          ; A1 * M1/2 IN BX     SAVE
         LODSW                 ; A2/2 IS NOW IN AX AND SI ← SI +2
         IMUL  M2 [DI]         ; A2 * M2/2 IS NOW IN DX, AX
         ADD   BX , DX         ; T2/2 IS NOW IN BX
         SAL   BX , 1          ; T2 IS NOW IN BX
         MOV   T2 [DI], BX     ; STORE T2 IN ITS LOCATION
         LODSW                 ; B1/2 IS NOW IN AX AND SI ← SI +2
         IMUL  M1 [DI]         ; B1 * M1/2 IN DX, AX
         MOV   BX , DX         ; B1 * M1/2 IS NOW IN BX
         LODSW                 ; B2/2 IS NOW IN AX AND SI ← SI +2
                               ; SI THEN POINTS TO NEXT STAGE
                               ; CONSTANT A1.
         IMUL  M2 [DI]         ; B2 * M2/2 IN DX, AX
         ADD   BX , DX         ; -T1/2 IS NOW IN BX
```


ENTER
(SI) → A1
DI ← Ø
AX ← M(SI)
AX CONTAINS A1/2
SI ← SI + 2
SI THEN POINTS TO A2
DX, AX ← AX * M(M1 + DI)
TRUNCATED VALUE A1 * M1/2 IN DX
BY IGNORING CONTENTS OF AX
BX ← DX
BX NOW CONTAINS A1 * M1/2
AX ← M(SI)
AX CONTAINS A2/2
SI ← SI + 2
SI THEN POINTS TO B1
DX, AX ← (AX) * M(M2 + DI)
TRUNCATED VALUE A2 * M2/2 IN DX
BY IGNORING CONTENTS OF AX
(BX) ← (BX) + (DX)
NOW T2/2 IS AVAILABLE IN BX
BX ← SHL BX
BX IS MULTIPLIED BY 2 TO GET T2
P

CONTD


P
M(T2 + DI) → BX
THIS STORES T2 IN THE CORRESPONDING LOCATION
AX → M(SI)
AX CONTAINS B1/2
SI → SI + 2
THEN SI POINTS TO B2
DX , AX → (AX) * M(M1 + DI)
TRUNCATED VALUE B1 * M1/2 IN DX
BY IGNORING THE CONTENTS OF AX
BX → DX
BX NOW CONTAINS B1 * M1/2
AX → M(SI)
AX NOW CONTAINS B2/2
SI → SI + 2
THEN SI POINTS TO A1 OF NEXT STAGE
DX , AX → (AX) * M(M2 + DI)
TRUNCATED VALUE B2 * M2/2 IS IN DX
BY IGNORING THE CONTENTS OF AX
(BX) → (BX) + (DX)
BX NOW CONTAINS - T1/2
(BX) ← SHL (BX)
(BX) IS MULTIPLIED BY 2 TO GET - T1
(BX) ← - (BX)
SIGN CHANGET TO GET T1
Q

CONTD.



Q
M ( T1 + DI) → (BX)
STORE T1 IN CORRESPONDING LOCATION
(DI) → (DI + 2)
DI NOW POINTS TO NEXT STAGE LOCATIONS
(CX) → (CX) - 1
IS CX = 0
NO
YES
RETURN

FLOW CHART FOR PREPROCESSING - 1D SUBROUTINE

```
        SAL   BX , 1       ; -T1 IN BX
        NEG   BX           ; T1 IS NOW IN BX
        MOV   T1 [DI], BX; STORE T1 IN ITS LOCATION
        ADD   DI , # 2     ; DI THEN POINTS TO NEXT STAGE
                           ; LOCATIONS
         LOOP PLP-1D       ; CX IS DECREMENTED AND LOOPS
                           ; BACK IF NOT ZERO
         RET
```

3.4-9 MAIN PROGRAM

The flowchart for the main program is shown in Fig. (3.14) and the corresponding program in PROGRAM - 3.7. The main program is written in the same way as in PROGRAM - 3.1 but by utilizing the subroutine programs (3.2 to 3.6) explained in earlier sections in sequence. Fig. (3.3) is implemented this main program.

PROGRAM - 3.7

```
NFILTR  : CALL   INIT-1D       ; IN THE PROGRAM PROPER CONSTANTS
                               ; BE LOADED IN  CX REGISTER
SAMPLE  : CALL   INP-1D        ; INPUTS X(K) INTO AX
          IMUL   SØ            ; X(K) * SØ/2 IN DX, AX
          SAL    DX , 1        ; X(K) * SØ IN DX
          MOV    AX , DX       ; AX NOW CONTAINS SCALED X(K)
          MOV    CX , # N      ; NUMERICAL 'N' BE LOADED TO CX
          CALL   OUT-1D        ; COMPUTE Y(K) AND MAKE IT
                               ; AVAILABLE IN AX
          MOV    PORTØ , AX    ; PORTØ IS THE OUTPUT PORT ADDR.
          MOV    CX , # N      ; LOAD CX AGAIN WITH NUMERICAL
                               ; VALUE OF 'N'
          CALL   DEL-1D        ; BLOCK MOVE M2 <- M1; M1 <- MØ
          MOV    CX , # N      ; LOAD CX WITH 'N' FOR PRE-
                               ; CALCULATIONS
          CALL   PRE-1D        ; CALCULATES T1 AND T2 OF ALL
                               ; STAGES
```

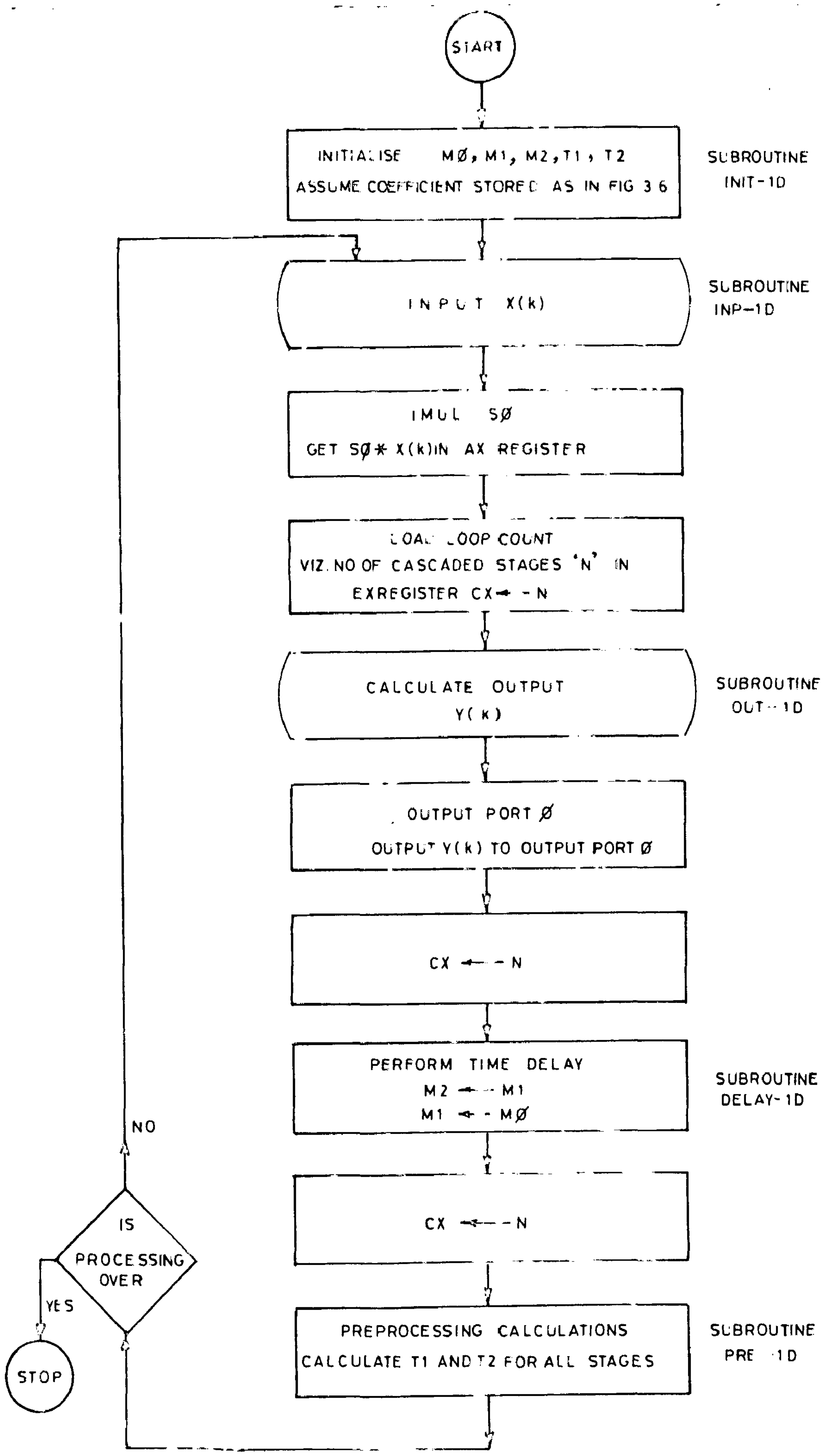


FIG. 3.14 FLOW CHART FOR MAIN PROGRAM

```
MOV   AX , PORT4      ; READ PORT4 FOR PROCESSING
                      ; OVER OR NOT
NOT   AX              ; PORT4 CONTAINS FFFF(H)
                      ; TO CONTINUE
JZ    SAMPLE
HALT
```

## 3.5 PARALLEL IMPLEMENTATION OF K'TH ORDER DIGITAL FILTER

Another method of avoiding coefficient sensitivity is to implement the filter as a sum of partial fraction of the given Transfer function. Equation for Parallel Canonic form is

$$D(z) = \beta_0 + \sum_{i=1}^{N} H_i(z) \qquad \ldots \qquad \ldots \quad (3.23)$$

where $H_i(z) = \beta_i \cdot \dfrac{A_{oi} + A_{1i}\cdot Z^{-1} + A_{2i}\cdot Z^{-2}}{1 + B_{1i}\cdot Z^{-1} + B_{2i}\cdot Z^{-2}}$ ... (3.24)

where, $A_{oi} = 0$ and is introduced to make the second order Z-Transfer function identical to equation (3.1). This ensures that the subroutine developed for cascaded structure can be made use of judiciously in parallelstructure implementation The coefficients of Fig. (3.15) are adjusted such that $\beta_i = 2^K$, this ensures that the results obtained from output subroutine program can be easily modified by shifting the result left by K-bits, which is equivalent to multiplying by $2^K$.

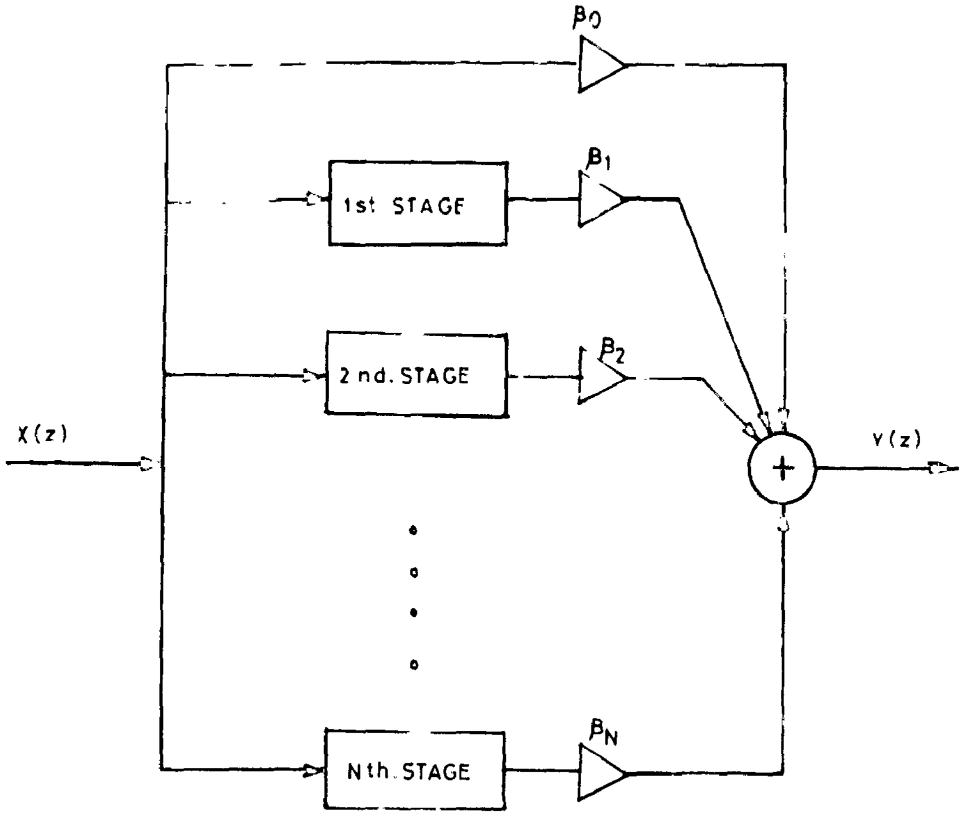


FIG. 3.15 BLOCK DIAGRAM REPRESENTATION OF PARALLEL STRUCTURAL

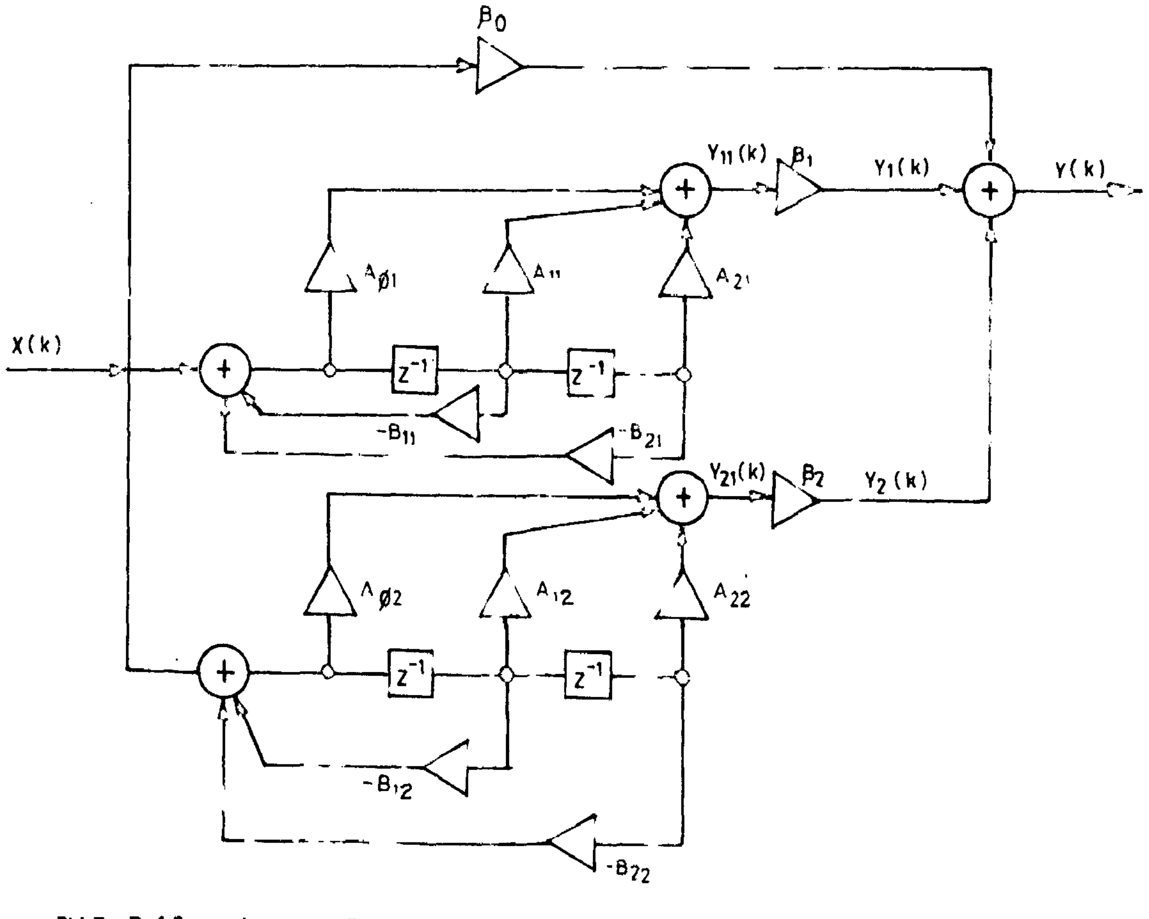


FIG 3.16 4th ORDER PARALLEL FILTER USING TWO SECOND ORDER 1D MODULES

Fig. (3.15) gives the block diagram representation of the parallel N'th order Digital filter. As a specific example [29] for parallel structure implementation consider a 4th order Digital filter. Transfer function is as

$$D(z) = \beta_o - 8. \frac{A_{o1} + A_{11}.Z^{-1} + A_{21}.Z^{-2}}{1 + B_{11}.Z^{-1} + B_{21}.Z^{-2}}$$

$$-4 . \frac{A_{o2} + A_{12}.Z^{-1} + A_{22}.Z^{-2}}{1 + B_{12}.Z^{-1} + B_{22}.Z^{-2}} \qquad \ldots \quad (3.25)$$

Where, $A_{o1} = A_{o2} = 0$, and all the constant $A_{11}$, $A_{21}$, etc. lie between -2 and +2 so that they can be stored as half values and $\beta_1 = -8$, $\beta_2 = -4$. It can be realized shown in Fig. (3.16). The complete program for implementing equation (3.25) is shown in PROGRAM - 3.8. The comments in the comment field of each instruction in the program are self-explanatory.

PROGRAM - 3.8

```
MAIN PROGRAM FOR 4TH ORDER DIGITAL FILTER PARALLEL STRUCTURE

; THE EQUN BE MADE AVAILABLE AS GIVEN IN EQUN 3.25
; THE COEFFICIENTS ARE STORED AS HALF VALUES  AS
; EXPLAINED IN FIG. 3.6A, FIG. 3.6B WITH AØ1 = AØ2 = Ø
; CALLS ALL SUBROUTINES DEVELOPED FOR CASCADED STRUCTURE
```

```
FILTR4 : CALL   INIT-1D       ; CLEAR M1, M2, T1 AND T2
CONT   : CALL   INP-1D        ; GET X(K) FROM A/D
                              ; X(K) IS NOW IN AX
         MOV    BX , AX       ; X(K) IS NOW IN BX
         IMUL   BØ            ; BØ * X(K)/2 IN DX , AX
         SAL    DX , 1        ; BØ * X(K) IN DX
         MOV    TEMP , DX     ; TEMP LOCATION NOW CONTAINS THE
                              ; FIRST TERM OF OUTPUT Y(K)
         MOV    AX , BX       ; AX AND BX HAS X(K)
         MOV    CX , # 1      ; CALCULATE FIRST STAGE OUTPUT
                              ; FROM HERE ONWARDS
         CALL   OUTP-1D       ; CALCULATE Y11(K).
                              ; Y11(K) IS IN AX AND SI <- SI+2
                              ; WHEN RETURNED
         SAL    AX , 3        ; -Y1(K) IS NOW IN AX(=8.Y11(K))
         NEG    AX            ; Y1(K) IN AX NOW
         ADD    AX , TEMP     ; AX NOW CONTAINS SUM OF TWO TERMS
                              ; OF OUTPUT = BoX(K) + Y1(K)
         MOV    TEMP , AX     ; (TEMP) = BoX(K) + Y1(K)
         MOV    AX , BX       ; AX AND BX BOTH CONTAIN X(K)
         MOV    CX , # 1      ; CALCULATE SECOND STAGE OUTPUT
                              ; HERE ONWARDS
; DO NOT CALL OUTP-1D ROUTINE OMIT FIRST TWO INSTRUCTIONS
; BECAUSE INDEX SI MUST POINT TO SECOND STAGE LOCATIONS WITH
; DI POINTING TO MØ + 2.  SO CALL FROM OPL-1D

         CALL   OPL-1D        ; Y21(K) IS RETURNED IN AX
         SAL    AX , 2        ; -Y2(K) IS NOW IN AX
         NEG    AX            ; Y2(K) IS IN AX NOW
         ADD    AX , TEMP     ; AX <- Bo.X(K) + Y1(K) + Y2(K)
         MOV    PORTØ , AX    ; MOVE OUT TO D/A IN PORTØ
         MOV    CX , # 2      ; DELAY-1D INITIALIZATION
         CALL   DEL-1D        ; M2 <- M1 AND M1 <- MØ FOR
                              ; TWO STAGES
         MOV    CX , # 2      ; PRE-1D INITIALIZATION
         CALL   PRE-1D        ; CALCULATES T1 AND T2 FOR
                              ; TWO STAGES
         MOV    AX , PORT4    ; INPUT PROCESSING OVER
                              ; OR NOT SIGNAL ISSUED
         NOT    AX            ; SIGNAL IS FFFF(H) CONTINUE
         JZ     CONT          ; JUMP TO START IF AX IS ZERO
         HALT
```

## 3.6 SUMMARY

In order to avoid coefficient sensitivity problems a Digital filter is implemented as a cascade or parallel combination of second order modules. In this chapter the second order 1D module has been extensively dealt with. The mathematical derivation, algorithm and the software program using Intel 8086 instruction set has been derived. The K'th order Digital filter using N second order 1D modules in cascade and in parallels have also been discussed : the various subroutines for these structures are written. Finally, a main program for the cascade and parallel (4th order) structure using these subroutines written.

# CHAPTER - IV

## IMPLEMENTATION OF SECOND ORDER DIGITAL FILTER THROUGH OTHER STRUCTURES

### 4.1 INTRODUCTION

The Transfer function for the i'th stage second order module is rewritten as

$$D(z) = \frac{A_{oi} + A_{1i}.Z^{-1} + A_{2i}.Z^{-2}}{1 + B_{1i}.Z^{-1} + B_{2i}.Z^{-2}} \quad \ldots \quad (4.1)$$

Equation (4.1) can be implemented by any of the realization structures discussed earlier in section 1.3. Fig. (4.1) is a flow chart that models all the second order modules implemented by these Direct structures [25].

In this chapter the necessary mathematical equations, algorithm and finally the subroutine programs for 2D, 3D and 4D second order structures will be discussed. Also two other structures viz. 1X and 2X crosscoupled structures [30] will be used to implement the above equation (4.1).

### 4.2 2D STRUCTURE

Cross Multiplying both sides of equation (4.1)

$$Y(z) + B_1.Z^{-1}.Y(z) + B_2.Z^{-2}.Y(z) = A_o.X(z) + A_1.Z^{-1}.X(z) + A_2.Z^{-2}.X(z) \quad \ldots \quad (4.2)$$

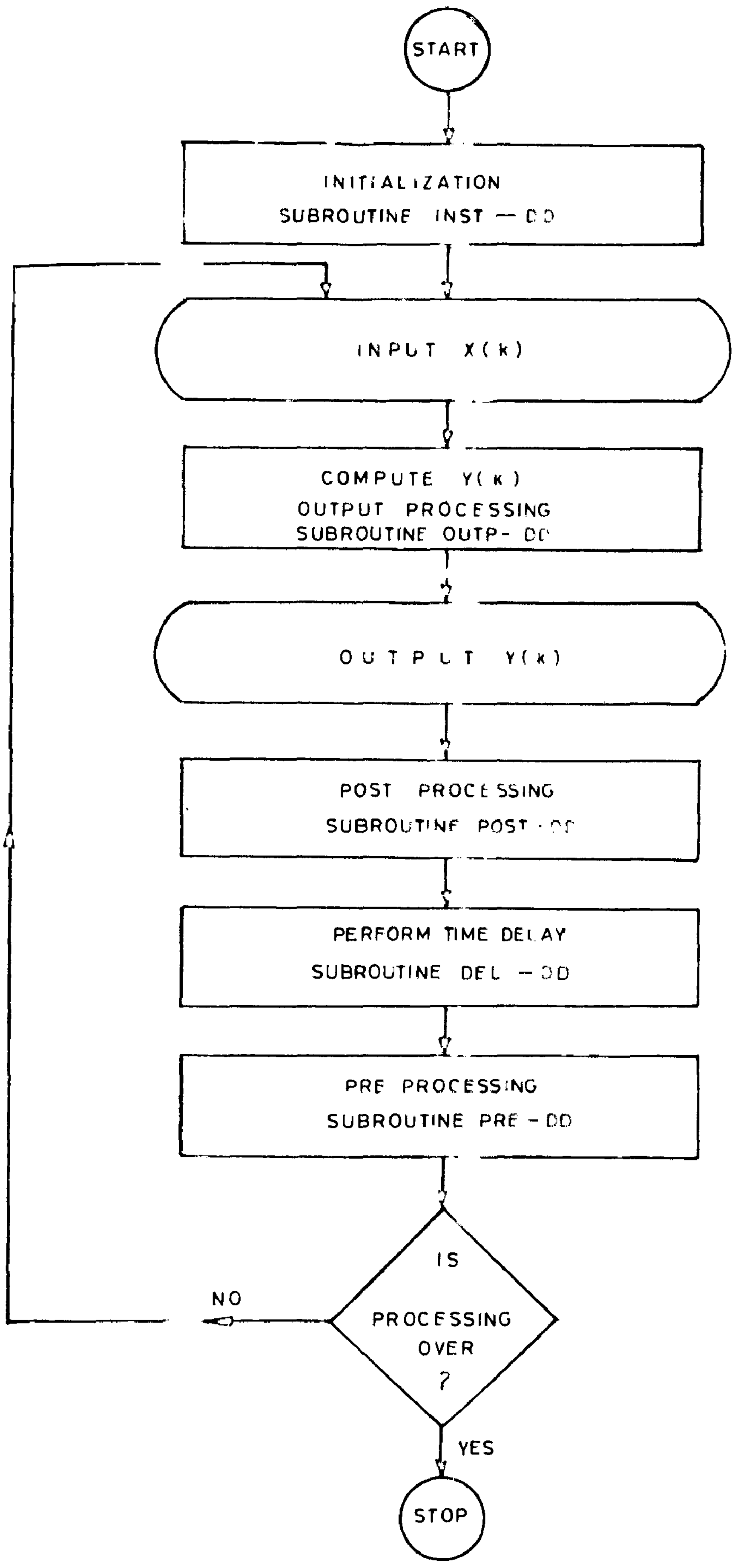


FIG 4.1 GENERAL FLOW CHART OF SECOND ORDER MODULES

$$Y(z) = A_o.X(z) + A_1.Z^{-1}.X(z) + A_2.Z^{-2}.X(z)$$
$$- B_1.Z^{-1}.Y(z) - B_2.Z^{-2}.Y(z) \quad \ldots \quad (4.3)$$

Arranging in the powers of $Z^{-1}$ and $Z^{-2}$

$$Y(z) = A_o.X(z) + (A_1.X(z) - B_1.Y(z)).Z^{-1}$$
$$+ (A_2.X(z) - B_2.Y(z)).Z^{-2} \quad \ldots \quad (4.4)$$

Let $P_2(z) = A_2.X(z) - B_2.Y(z)$ ... (4.5)

So $P_2(z).Z^{-1} = (A_2.X(z) - B_2.Y(z)).Z^{-1}$ ... (4.6)

Also let $P_1(z) = (A_1.X(z) - B_1.Y(z)) + P_2(z).Z^{-1}$ .. (4.7)

So $P_1(z).Z^{-1} = (A_1.X(z) - B_1.Y(z)).Z^{-1} + P_2(z).Z^{-2}$ ... (4.8)

Substituting the value of $P_1(z).Z^{-1}$ and $P_2(z).Z^{-1}$ in equation (4.4)

$$Y(z) = A_o.X(z) + P_1(z).Z^{-1} \quad \ldots \quad (4.9)$$
$$P_1(z) = A_1.X(z) - B_1.Y(z) + P_2(z).Z^{-1} \quad \ldots \quad (4.10)$$

and $P_2(z) = A_2.X(z) - B_2.Y(z)$ ... (4.11)

In the time domain

$$y(k) = A_o.x(k) + p_1(k-1) \quad \ldots \quad (4.12)$$
$$p_1(k) = A_1.x(k) - B_1.y(k) + p_2(k-1) \quad \ldots \quad (4.13)$$
$$p_2(k) = A_2.x(k) - B_2.y(k) \quad \ldots \quad (4.14)$$

From equations (4.12) through (4.14) y(k) is to be found out first. The values of $p_1(k)$ and $p_2(k)$ are calculated during $KT < t < KT + T$. A stepwise procedure is

| | |
|---|---|
| OUTPUT | $y(k) = A_0 \cdot x(k) + p_1(k-1)$ |
| POST PROCESSING in the interval $KT < t < KT + T$ | $p_1(k) = A_1 . x(k) - B_1 . y(k) + p_2(k-1)$ |
| | $p_2(k) = A_2 . x(k) - B_2 . y(k)$ |
| DELAY | $p_2(k-1) \longleftarrow p_2(k)$ |
| | $p_1(k-1) \longleftarrow p_1(k)$ |

The flow chart of Fig. (4.1) represents the process. Precalculation is not needed. The steps involved are,

| | | |
|---|---|---|
| (1) | Initialization; | A/D, D/A converters and all variables |
| (2) | Input X(k) ; | From A/D converter |
| (3) | Compute Y(k) | |
| (4) | Output Y(k) ; | To D/A converter |
| (5) | Perform Time Delay | |
| (6) | Compute P1 and P2; | Post calculation of P1 and P2 |

The subroutines for 2D structure making use of the instruction set of 8086 microprocessor are given in PROGRAM - 4.1.

PROGRAM - 4.1

```
FILTER SECOND ORDER 2D STRUCTURE
; SUBROUTINE INITIALIZATION AND SUBROUTINE INPUT ARE SAME
; AS IN PROGRAM - 3.2 FIG. (3.7) AND PROGRAM - 3.3 FIG. (3.8)
; RESPECTIVELY.
; SUBROUTINE OUTPUT COMPUTES OUTPUT SAMPLE Y = AØ.X + P12

; X PASSED IN AX Y RETURNED IN AX.LOOP COUNT IN CX.

OUT - 2D  ; MOV   SI #  Ø      ;STAGE INDEX
            LEA   DI , X(K)    ;POINTS TO X
OLP - 2D  : STOSW              ;SAVE X
            IMUL  AØ [SI]      ;X * AØ / 2 IN DX
            SAL   DX , 1       ;X * AØ IN DX
            ADD   DX , P11[SI] ;Y
            MOV   AX , DX      ;Y IN AX READY FOR NEXT STAGE
            ADD   SI # 2       ;MOVE INDEX TO POINT NEXT STAGE
                               ;LOCATION
            LOOP  OLP-2D       ;USE COUNT IN CX
            RET

;OUTPUT Y  IN AX TO OUTPUT PORTØ. PORTØ BEING THE ADDRESS
;ASSIGNED TO D/A CONVERTER IN MEMORY MAPPED I/O MODE
            MOV    PORTØ , AX
; COMPUTE DELAY P12 <--- P1 ,    P22 <---- P2
DEL-2D      LEA   DI, P1       ;P(k)
            LEA   SI           ;P(k-1)
            REP MOVS           ; PERFORM BLOCK MOVE
            RET
;PREPROCESSING 2D NOT USED IN 2D MODULE SECOND ORDER
;STRUCTURE
            PRE-2D    RET
;POST PROCESSING - 2D CALCULATIONS BEGIN  HERE TO CALCULATE
;P1 = A1*X - B1*Y + P22 AND P2 = A2*X - B2*Y
;LOOP COUNT IN CX
POST - 2D : LEA   SI , A1      ;COEFFICIENT POINTER
            LEA   BX , X       ;POINTS TO INPUTS
            MOV   DI # Ø       ;STAGE INDEX
POLP-2D :   LODSW              ;A1/2 IN AX AND SI <--  SI + 2
            IMUL BX [DI]       ;A1 * X/2 IN DX,;  AX AND
                               ;BX <-- DX + 2
            PUSH  DX           ;SAVE
            LODSW              ;B1/2
            IMUL  2 BX [DI]    ;B1  * Y/2 IN DX, AX and
                               ;AND BX <--- BX + 2
            POP   AX
            SUB   AX , DX      ;A1  * X - B1  * Y)/2
            SAL   AX , 1       ;A1  * X - B1  * Y
            ADD   AX , P21[DI] ;COMPUTE P1
            MOV   P1[DI], AX   ;STORE P1
            LODSW .            ;A2/2 IN AX AND SI <--- SI + 2
            IMUL  [BX][DI ]    ;X + A2/2 IN DX, AX
                               ;AND BX <--- BX + 2
```

```
        PUSH   DX
        LODSW                 ;B2/2 IN AX AND SI <- SI + 2
        IMUL   2 [DX] [DI]    ;Y * B2/2
        POP    AX
        SUB    AX , DX        ;P2/2 = (X * A2 - Y * B2)/2
        SAL    AX , 1         ;P2
        MOV    P2 [DI]        ;STORE P2
        ADD    DI #2          ;MOVE INDEX TO POINT NEXT
                              ;STAGE LOCATION
        LOOP   POLP-2D        ;USE COUNT IN CX
        RET

2D CONSTANT STORAGE FOR N STAGES

  AØ  : DW AØ1, AØ2,,,,,  AØN   ; AØ FOR N STAGES
  A1  : DW A11, B11, A21, B21   ; STAGE 1 COEFFICIENTS
        DW A12, B12, A22, B22   ; STAGE 2 COEFFICIENTS
        .
        .
        .
        .
        DW A1N, B1N, A2N, B2N   ; STAGE N COEFFICIENTS

2D TEMPORARY STORAGE FOR N STORAGE

  X     DW   (N+1) DUPØ         ; INPUTS/OUTPUTS
  P1    DW       N DUPØ
  P2    DW       N DUPØ
```

## 4.3 <u>3D STRUCTURE</u>

Equation (4.1) can be written as

$$Y(z) = (A_o + A_1.Z^{-1} + A_2.Z^{-2}).X(z) - (B_1.Z^{-1} + B_2.Z^{-2}).Y(z) \quad \ldots \quad (4.15)$$

or

$$Y(z) = A_o.X(z) + A_1.Z^{-1}.X(z) + A_2.Z^{-2}.X(z) - B_1.Z^{-1}.Y(z) - B_2.Z^{-2}.Y(z) \quad .. \quad (4.16)$$

In the time domain

$$y(k) = A_o.x(k) + A_1.x(k-1) + A_2.x(k-2) - B_1.y(k-1) - B_2.y(k-2) \quad \ldots \quad (4.17)$$

Let $T_3 = A_1.x(k-1) + A_2.x(k-2) - B_1.y(k-1) - B_2.y(k-2)$ ... (4.18)

Equation (4.17) is represented as

$$y(k) = A_o.x(k) + T_3 \qquad \ldots \qquad (4.19)$$

It is to be noted that the intermediate variable $T_3$ depends on the previous samples and is evaluated in the interval $KT - T < t < KT$ and shall be available before KT sampling point. The output y(k) can be calculated upon the receipt of input x(k). A stepwise procedure is

| | |
|---|---|
| OUTPUT | : $y(k) = A_o.x(k) + T_3$ |
| PREPROCESSING in the interval $KT-T < t < KT$ | ; $T_3 = A_1.x(k-1) + A_2.x(k-2) - B_1.y(k-1) - B_2.y(k-2)$ |
| DELAY | : $x(k-1) \longleftarrow x(k)$ , $x(k-2) \longleftarrow x(k-1)$ |
| | $y(k-1) \longleftarrow y(k)$ , $y(k-2) \longleftarrow y(k-1)$ |

The flow chart of Fig. 4.1 represents the process. Post calculation is not needed. Following are the steps involved.

(1) Initialization; A/D D/A converters and all
; variables

(2) Compute X(k) ; From A/D converter

(3) Compute Y(k)

(4) Output Y(k) ; To D/a converter

(5) Perform Time Delay

(6) Compute $T_3$ ; Precalculation of $T_3$

Various subroutines in the 3D implementation are given in PROGRAM - 4.2.

PROGRAM - 4.2

```
FILTER SECOND ORDER 3D STRUCTURE

;SUBROUTINE INITIALIZATION AND SUBROUTINE INPUT ARE SAME

;AS IN PROGRAM - 3.2 FIG. (3.7) AND PROGRAM - 3.3 FIG. (3.8)

; RESPECTIVELY.

;SUBROUTINE OUTPUT COMPUTES Y = AØ * X + T3

;LOOP COUNT IN CX

OUT - 3D  :    LEA    DI, X1          ; POINTS TO X
               MOV    SI , # Ø        ; STAGE INDEX

CLP - 3D  :    STOSW                  ; SAVE X , Y
               IMUL   AØ [SI]         ; AØ * X/2 IN DX
               SAL    DX , 1          ; AØ * X
               ADD    DX , T3 [SI]    ; COMPUTE Y
               MOV    AX , DX         ; RETURN Y IN AX
               ADD    SI , # 2        ; POINTS TO NEXT STAGE
               LOOP   CLP-3D
               STOSW                  ; SAVE LAST Y
               RET

;OUTPUT Y IN AX TO OUTPUT PORTØ. PORTØ BEING THE ADDRESS
;ASSIGNED TO D/A CONVERTER IN MEMORY MAPPED I/O MODE

               MOV    PORTØ , AX

;COMPUTE DELAY X2 <-- X1.  LOOP COUNT IN  CX

DEL - 3D  :    LEA    SI , X1         ; POINTS TO X(k)
               LEA    DI , X2         ; POINTS TO X(k-1)
```

```
; DEL - 3D SUBROUTINE CONTINUES

                INC   CX        ; MOVE X VALUES AND Y
                REP             ; PERFORMS
                MOVS            ; BLOCK MOVE OPERATION
                RET             ;


; PREPROCESSING 3D CALCULATIONS BEGIN HERE TO CALCULATE
; T3 = A1.X1 + A2.X2 - B1.Y1 - B2.Y2
; LOOP COUNT IN CX.
PRE
PRE - 3D :      LEA    SI , A1 ; COEFFICIENT POINTER
                MOV    DI , #Ø; INDEX

PLP - 3D :      LODSW           ; A1/2
                IMUL  X1 [DI]   ; X1 * A1/2 IN DX
                MOV  BX , DX    ; PARTIAL SUM IN BX
                LODSW           ; A2/2
                IMUL X2 [DI]    ; X2 * A2/2 IN DX
                ADD    BX , DX ; PARTIAL SUM
                LODSW           ; B1/2
                IMUL   X1+2 [DI]; Y1 * B1/2 IN DX
                SUB    BX; DX   ; TOTAL
                LODSW           ; B2/2
                IMUL   X2+2 [DI] ; Y2 * B2/2
                SUB    BX , DX ; T3/2
                SAL    BX , 1   ; T3
                MOV    T3 [DI] , BX   ; STORE
                ADD    DI , # 2       ; MOVE INDEX TO POINT
                                      ; NEXT STAGE LOCATION
                LOOP   PLP-3D         ; USE COUNT IN CX
                RET

; POST PROCESSING 3D NOT USED IN 3D SECOND ORDER MODULE

                POST-3D
                RET


3D CONSTANT STORAGE FOR N STAGES

   AØ : DW AØ1, AØ2 ............... AØN   ; AØ FOR N STAGES
   A1 : DW A11, B11, A21, B21             ; STAGE 1 COEFFICIENTS
   A2 : DW A12, B12, A22, B22             ; STAGE 2 COEFFICIENTS
   .
   .
   .
   .
         DW A1N, B1N, A2N, B2N            ; STAGE N COEFFICIENTS
3D TEMPORARY STORAGE FOR N STAGES

  X1 DW (N+1) DUPØ                        ; X(k), Y(k)
  X2 DW (N+1) DUPØ                        ; X(k-1), Y(k-1)
  T3 DW  N    DUPØ
```

## 4.4 4D STRUCTURE

Introducing intermediate variable $R_o(z)$ in equation (4.1)

$$\frac{Y(z)}{R_o(z)} \circ \frac{R_o(z)}{X(z)} = \frac{A_o + A_1.Z^{-1} + A_2.Z^{-2}}{1 + B_1.Z^{-1} + B_2.Z^{-2}} \quad \ldots \quad (4.20)$$

$$Y(z) = A_o.R_o(z) + A_1.Z^{-1}.R_o(z) + A_2.Z^{-2}.R_o(z) \quad \ldots (4.21)$$

and $X(z) = R_o(z) + B_1.Z^{-1}.R_o(z) + B_2.Z^{-2}.R_o(z)$ ... (4.22)

Hence $R_o(z) = X(z) - B_1.Z^{-1}.R_o(z) - B_2.Z^{-2}.R_o(z)$ ... (4.23)

Let $R_1(z) = -B_1.R_o(z) - B_2.Z^{-1}.R_o(z)$ ... (4.24)

So $R_o(z) = X(z) + R_1(z).Z^{-1}$ ... (4.25)

Let $Q_1(z) = A_1.R_o(z) + A_2.Z^{-1}.R_o(z)$ ... (4.26)

So $Y(z) = A_o.R_o(z) + Q_1(z).Z^{-1}$ ... (4.27)

Rewriting equations (4.23) through (4.27)

$R_o(z) = X(z) + R_1(z).Z^{-1}$ ... (4.25)

$Y(z) = A_o.R_o(z) + Q_1(z).Z^{-1}$ ... (4.27)

~~$R_1(z) = -B_1.R_o(z) - B_2.R_o(z).Z^{-1}$~~ ... (4.24)

$Q_1(z) = A_1.R_o(z) + A_2.R_o(z).Z^{-1}$ ... (4.26)

In the time domain this set of equations is

$r_o(k) = x(k) + r_1(k-1)$

$y(k) = A_o.r_o(k) + q_1(k-1)$

$r_1(k) = B_1.r_o(k) - B_2.r_o(k-1)$

$q_1(k) = A_1.r_o(k) + A_2.r_o(k-1)$

From above it is clear that $y(k)$ is to be calculated first and $r_1(k)$ and $q_1(k)$ can be calculated in the interval $KT \leq t < KT + T$. A stepwise procedure is

OUTPUT : $r_o(k) = x(k) + r_1(k-1)$

$y(k) = A_o \cdot r_o(k) + q_1(k-1)$

POSTCALCULATION in the interval $KT < t < KT + T$ : $r_1(k) = -B_1 \cdot r_o(k) - B_2 \cdot r_o(k-1)$

$q_1(k) = A_1 \cdot r_o(k) + A_2 \cdot r_o(k-1)$

DELAY : $r_o(k-1) \longleftarrow r_o(k)$ ;

$r_1(k-1) \longleftarrow r_1(k)$

$q_1(k-1) \longleftarrow q_1(k)$

Flow chart of Fig. (4.1) represent the 4D structure and the program is given in PROGRAM - 4.3. No precalculation is needed.

~~PROGRAM - 4.3~~

```
FILTER SECOND ORDER 4D STRUCTURE
; INITIALIZATION SUBROUTINE AND INPUT SUBROUTINE ARE SAME
; AS IN. PROGRAM - 3.2 FIG. (3.7) AND PROGRAM - 3.3 FIG. (3.8)
; RESPECTIVELY.
; SUBROUTINE OUTPUT COMPUTES RØ = X + R11 AND OUTPUT SAMPLE
; Y = AØ * RØ + Q.11. PASS X IN AX, RETURN Y IN AX
; LOOP COUNT IN CX.
```

```
OUT - 4D :    LEA   DI , RØ        ; POINT TO RØ
              MOV   SI #Ø          ; STAGE INDEX
OLP - 4D ; .  ADD   AX , R1[SI]    ; RØ = X + R11
              STOSW                ; STORE IN LOCATION
              IMUL  AØ[SI]         ; RØ * AØ / 2 IN DX
              SAL   DX , 1         ; RØ * AØ
              ADD   DX , Q1[SI]    ; Y = AØ * RØ + Q11
              MOV   AX,  DX        ; RETURN IN AX
              ADD   SI , # 2       ; MOVE INDEX TO POINT NEXT
                                   ; STAGE LOCATIONS
              LOOP OLP-4D          ; USE COUNT IN CX
              RET

; OUTPUT Y IN AX TO OUTPUT PORTØ.  PORTØ BEING THE ADDRESS
;ASSIGNED TO D/A CONVERTER IN MEMORY MAPPED I/O MODE

              MOV   PORTØ ; AX

;PREPROCESSING 4D NOT USED IN 4D SECOND ORDER STRUCTURE

              PRE-4D RET

;DELAY 4-D CALCULATIONS BEGIN      RØ1 <- RØ   AND SO ON

DEL-4D    :   LEA   SI , RØ          ; SI POINTS TO RØ
              LEA   DI , RØ1         ; DI POINTS TO RØ1
              INC   CX
              CLD
              REP                    ; PERFORMS
              MOVS                   ; BLOCK MOVE
              RET

~~; POST PROCESSING 4-D CALCULATIONS BEGIN HERE TO CALCULATE~~
; R1 = -B1.RØ - B2.RØ1,   Q1 = A1.RØ + A2.RØ1.
; LOOP COUNT IN CX

POST-4D   :   LEA   SI , B1          ; COEFFICIENT POINTER
              MOV   DI , # Ø         ; STAGE INDEX
POLP-4D   :   LODSW                  ; B1/2
              IMUL  RØ [DI]          ; RØ * B1/2 IN DX, AX
              MOV   BX , DX
              LODSW                  ; B2/2
              IMUL  RØ1[DI]          ; RØ1 * B2/2 IN DX, AX
              ADD        BX , DX     ; - R1/2
              SAL        BX , 1      ; - R1
              NEG        BX          ; R1
              MOV        R1[DI], BX; STORE R1 IN LOCATION
```

```
        LODSW                        ; A1/2
        IMUL   RØ[DI]                ; RØ * A1/2
        MOV    BX , DX
        LODSW                        ; A2/2
        IMUL   RØ1[DI]               ; RØ1 * A2/2
        ADD    DX , DX               ; Q1/2
        SAL    BX/, 1                ; Q1
        MOV    Q1[DI] , DX           ; STORE Q1 IN LOCATIONS
        ADD    DI , # 2              ; MOVE INDEX TO POINT NEXT STAGE
                                     ; LOCATION
        LOOP   POLP-4D               ; USE COUNT IN CX
        RET
```

4D CONSTANT STORAGE FOR N STAGES

```
AØ : DW AØ1, AØ2 .....    AØN ; AØ FOR N STAGES
A1 : DW A11, B11, A21, B21    ; STAGE 1 COEFFICIENTS
     DW A12, B12, A22, B22    ; STAGE 2 COEFFICIENTS
     .
     .
     .
     .
     DW A1N, B1N, A2N, B2N : STAGE N COEFFICIENTS
```

4D TEMPORARY STORAGE FOR N STAGES

```
RØ  : DW    N DUP(Ø)           ; RØ(k)
RØ1 : DW    N DUP(Ø)           ; RØ(k-1)
R1  : DW    N DUP(Ø)           ; R1(k)
Q1  : DW    N DUP(Ø)           ; Q1(k)
```

## 4.5 1X AND 2X STRUCTURES

Another method of realizing a Digital filter is the cross coupled structure of Fig. 4.2. These derivations of these structures is extensively dealt with in [30].

The difference equations (in time domain) employed for 1X Structure :

$$y(k) = A_0 . x(k) + s_2(k-1) \quad \ldots \quad (4.28)$$

$$s_1(k) = g_1 . s_1(k-1) - g_2 . s_2(k-1) + g_3 . x(k) \quad .. \quad (4.29)$$

$$s_2(k) = g_1 . s_1(k-1) + g_2 . s_2(k-1) + g_3 . x(k) \quad .. \quad (4.30)$$

where the $g_i$ comes from

$$D(z) = A_o + \frac{A}{Z + p} + \frac{A^*}{Z + p^*}$$

This is a canonical structure, since two delay elements are used for implementing a second order module. A step-wise procedure is

OUTPUT : $y(k) = A_o.x(k) + s_2(k-1)$

POSTCALCULATION in the interval $KT < t < KT + T$ :

$$s_1(k) = g_1.s_1(k-1) - g_2.s_2(k-1) + g_3.x(k)$$

$$s_2(k) = g_1.s_1(k-1) + g_2.s_2(k-1) + g_4.x(k)$$

DELAY :

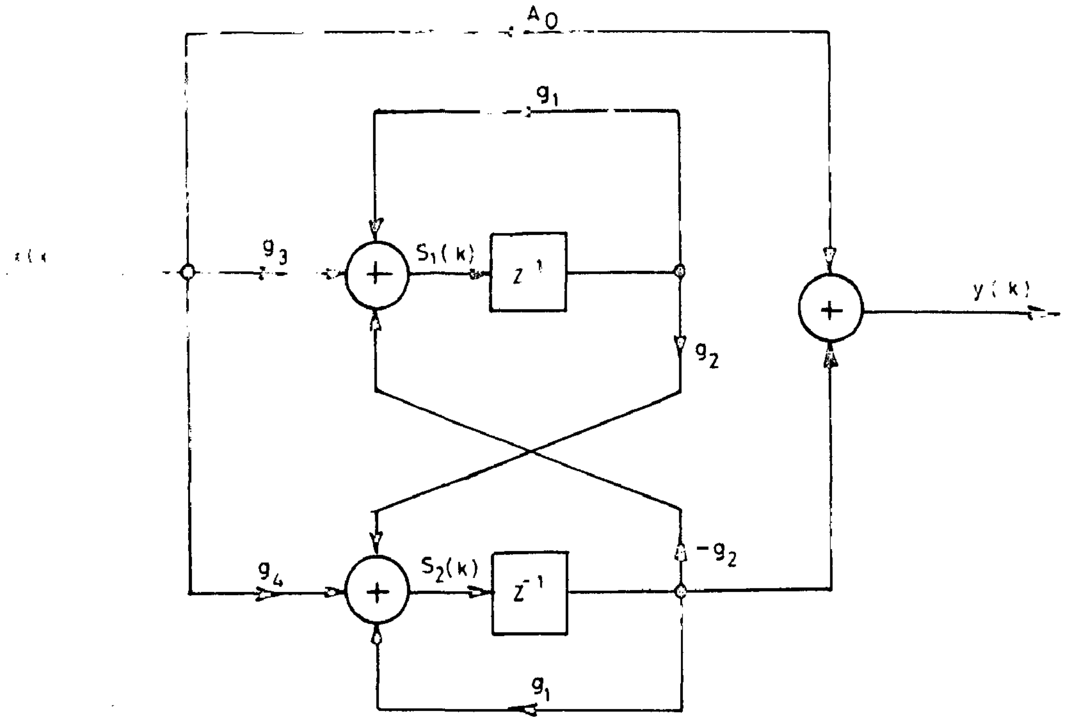
$$s_1(k-1) \longleftarrow s_1(k)$$

$$s_2(k-1) \longleftarrow s_2(k)$$

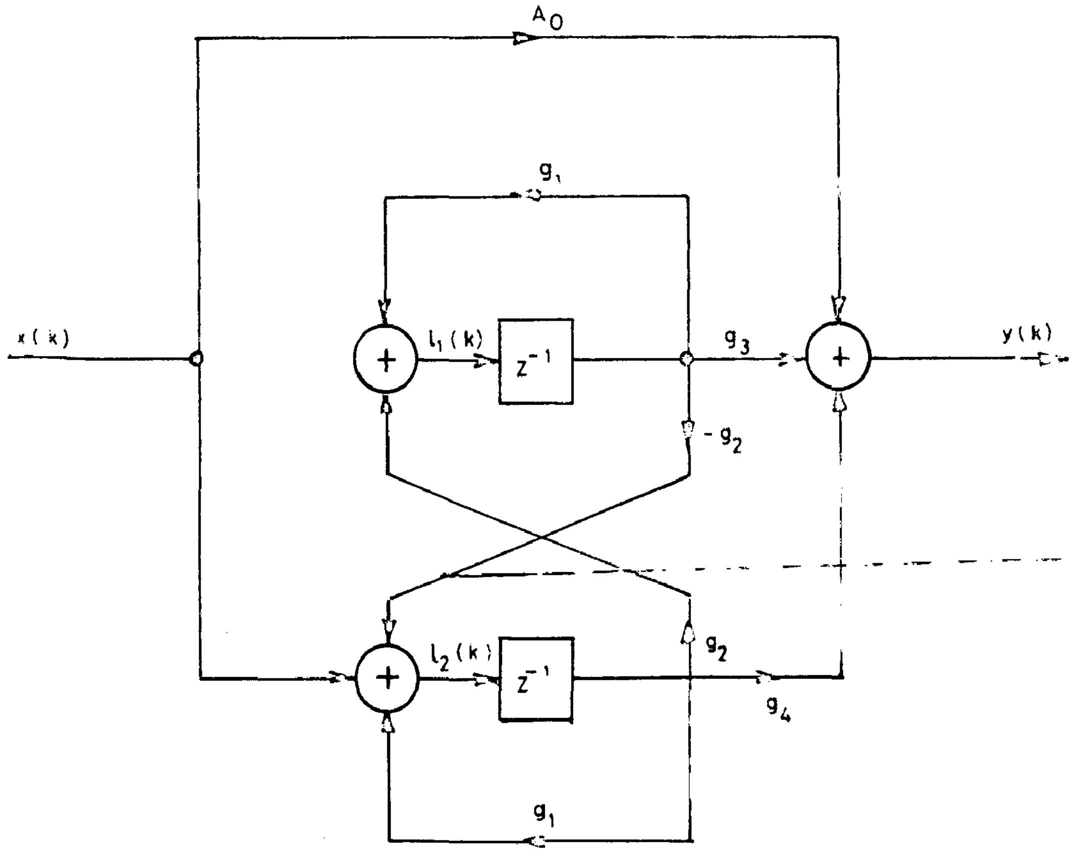
The flow chart of Fig. (4.1) also represents the 1X structure and the program is given in PROGRAM - 4.4.

PROGRAM - 4.4

```
FILTER SECOND ORDER 1X STRUCTURE
; INITIALIZATION AND INPUT SUBROUTINES ARE SAME
; AS IN PROGRAM - 3.2 FIG. (3.7) AND PROGRAM - 3.3 FIG. (3.8)
; RESPECTIVELY.
```



1X STRUCTURE



2X STRUCTURE

FIG 4.2 1X AND 2X STRUCTURES

```
; SUBROUTINE OUTPUT COMPUTES Y = AØ*X+ S2

; PASS X IN AX, RETURN Y IN AX.  LOOP COUNT IN CX

  OUTP - 1X : LEA    DI , X        ; POINT TO X
              MOV    SI , #Ø       ; STAGE INDEX
  OLP  - 1X : STOSW                ; SAVE X
              IMUL   AØ SI         ; X * AØ/2
              SAL    DX , 1        ; X * AØ
              ADD    DX , S2[SI]   ; Y = AØ*X + S2
              MOV    AX , DX       ; RETURN IN AX
              ADD    SI , # 2      ; MOVE INDEX ~~TO~~ POINT
                                   ; NEXT STAGE LOCATIONS
              LOOP   OLP-1X        ; USE COUNT IN CX
              RET

; OUTPUT Y IN AX TO OUTPUT PORTØ.  PORTØ BEING THE ADDRESS
; ASSIGNED TO D/A CONVERTER IN MEMORY MAPPED I/O MODE

              MOV    PORTØ , AX

; PREPROCESSING1X NOT  USED IN 1X SECOND ORDER STRUCTURE

              PRE-1X RET

; DELAY 1X CALCULATIONS BEGIN S1(k-1) <-- S1(k); S2(k-1) <--
; S2(k).  LOOP COUNT IN CX.

DEL - 1X : LEA    SI, S1       ; SOURCE
           LEA    DI, S11      ; DESTINATION
           ADD    CX, CX       ; DOUBLE COUNT FOR X1 AND X2
           REP                 ; PERFORM
           MOVS                ; BLOCK MOVE
           RET

; POSTCALCULATIONS 1X BEGIN.  S1 = G1*S11 - G2*S21 + G3*X ,
; S2 = G1*S21 - G2*S11 + G4*X.

POST-1X  : LEA    SI, G1       ; COEFFICIENT POINTER
           MOV    DI, #Ø       ; STAGE INDEX
POLP-1X  : LODSW               ; G1/2
           IMUL   S11[DI]      ; S11*G1/2
           MOV    BX, DX
           LODSW               ; G2/2
           IMUL   S21[DI]      ; S21*G2/2
           SUB    BX, DX
           LODSW               ; G3/2
           IMUL   X[DI]        ; X*G3/2
           ADD    BX, DX       ; S1/2
           SAL    BX, 1        ; S1
           MOV    S1[DI] , BX  ; STORE S1
```

```
; POSTCALCULATIONS SUBROUTINE FOR 1X MODULE CONTINUES

                LODSW               ; G1/2
                IMUL S21[DI]        ; S21*G1/2
                MOV BX, DX
                LODSW               ; G2/2
                IMUL  S11[DI]       ; S11*G2/2
                ADD  BX, DX
                LODSW               ; G4/2
                IMUL  X[DI]         ; X*G4/2
                ADD  BX, DX         ; X2/2
                SAL  DX, 1          ; S2
                MOV  S2[DI], BX     ; STORE S2
                ADD  DI # 2         ; MOV IN NEXT TO POINT
                                    ; NEXT STAGE LOCATIONS
                LOOP POLP-1X        ; USE COUNT IN CX
                RET

; 1X CONSTANT STORAGE FOR N STAGES
  AO : DW AØ1, AØ2 .... AØN        ; AØ FOR N STAGES
  G1 : DW G11, G21, G31 , G11, G21, G41  ; STAGE 1 COEFFICIENTS
          .
          .
          .
          .
       DW G1N, G2N, G3N, G1N, G2N, G4N; STAGE N COEFFICIENTS
; 1X DATA STORAGES FOR N STAGES
  X :  DW   N DUP(Ø)                ; INPUTS
  S1:  DW   N DUP(Ø) S1(k)          ; S1(k)
  S2 : DW   N DUP(Ø)                ; S2(k)
  S11: DW   N DUP(Ø) S              ; S1(k-1)
  S12: DW   N DUP(Ø)                ; S2(k-1)
```

B. 2X STRUCTURE :

This structure is the transpose (Appendix - II) of 1X structure. Equations involved are

$$y(k) = A_o.x(k) + g_3.l_1(k-1) + g_4.l_2(k-1) \quad \ldots (4.31)$$

$$l_1(k) = g_1.l_1(k-1) + g_2.l_2(k-1) \quad \ldots \quad \ldots (4.32)$$

$$l_2(k) = x(k) + g_1.l_2(k-1) - g_2.l_1(k-1) \quad \ldots (4.33)$$

A Stepwise procedure is

OUTPUT : $y(k) = A_0.x(k) + L_3$

POSTCALCULATIONS in the interval KT t KT + T ; $l_1(k) = g_1.l_1(k-1) + g_2.l_2(k-1)$

$l_2(k) = x(k) + g_1.l_2(k-1) - g_2.l_1(k-1)$

PRECALCULATIONS : $L_3 = g_3.l_1(k-1) + g_4.l_2(k-1)$

DELAY : Not necessary

Fig. (4.1) also represents this 2X structure and the program is given in PROGRAM - 4.5.

PROGRAM - 4.5

```
FILTER SECOND ORDER 2X STRUCTURE

; INITIALIZATION AND INPUT SUBROUTINES ARE SAME

; AS IN PROGRAM - 3.2 FIG. (3.7) AND PROGRAM - 3.3 FIG. (3.8)

; RESPECTIVELY.

; SUBROUTINE OUTPUT COMPUTES Y = AØ.X +L3 . PASS X IN AX

; LOOP COUNT IN CX.

 OUTP - 2X: LEA    DI, X         ; POINT TO X
            MOV    SI,  # Ø      ; STAGE INDEX
 OLP - 2X : STOSW                ; SAVE INPUTS TO STAGES
            IMUL   AØ[SI]        ; X*AØ/2
            SAL    DX, 1         ; X*AØ
            ADD    DX, L3[SI]    ; COMPUTE Y
            MOV    AX, DX        ; RETURN IN AX
            ADD    SI, # 2       ; MOVE INDEX TO POINT
                                 ; NEXT STAGE LOCATIONS
            LOOP OLP-2X          ; USE COUNT IN CX
            RETURN
```

```
; POSTCALCULATIONS 2X BEGIN L1 = G1.L1(k-1) + G2.L2(k-1),
; L2 = X + G1.L2 (k-1) - G2.L1(k-1). LOOP COUNT IN CX.

  POST-2X   : LEA    SI, G1          ; COEFFICIENT POINTER
              MOV    DI, #Ø          ; STAGE INDEX
  POLP-2X   : LODSW                  ; G1/2
              IMUL   L11[DI]         ; L11*G1/2
              MOV    BX, DX
              LODSW                  ; G2/2
              IMUL   L21 [DI]        ; L 21*g2/2
              ADD    BX, DX          ; L1/2
              SAL    BX, 1           ; L1
              MOV    L1[DI], BX      ; STORE L1(k)
              SUB    SI , # 4        ; BACK POINTER UPTO G1/2
              LODSW                  ; G1/2
              IMUL   L21 [DI]        ; G1*L21/2
              MOV    BX, DX
              LODSW                  ; G2/2
              IMUL   L11 [ DI]       ; G2* L11/2
              SUB    BX, DX          ; PARTIAL SUM
              SAL    BX, 1           ; L2 - X
              ADD    BX, X[DI]       ; L2
              MOV    L2[DI], BX      : STORE L2(k)
              ADD    DI, #  2        ; MOVE INDEX TO POINT
                                     ; NEXT STAGE LOCATIONS
              LOOP POLP-2X           ; USE COUNT IN CX
              RET

; DELAY 1X CALCULATION NOT NECESSARY FOR 2X

              DEL-2X RET

; PREPROCESSING CALCULATIONS 2X BEGIN.
; L3 = G3*L1(k-1) + G4*L2(k-1).  LOOP COUNT IN CX.

  PRE-2X    : LEA    SI, G3          ; COEFFICIENT POINTER
              MOV    DI, # Ø         ; STAGE INDEX
  PLP-2X    : LODSW                  ; G3/2
              IMUL L11[ DI]          ; G3*L11/2
              MOV  BX, DX
              LODSW                  ; G4/2
              IMUL L3  [DI]          ; G4*L21 /2
              ADD  BX, DX            ; L3/2
              SAL  BX, 1             ; L3
              MOV  L3[DI] , BX       ; STORE L3(k)
              ADD  D1  , #  2        ; MOVE INDEX TO POINT
                                     ; NEXT STAGE LOCATIONS
              LOOP-PLP-2X.           ; LOOSE COUNT IN CX
              RET
```

```
2X CONSTANT STORAGE  FOR N STAGES

G1: DW G11, G12              ; STAGE 1
    DW G21  G22              ; STAGE 2

    DW GN1, GN2              ; STAGE N

G3  DW G13, G14              ; STAGE 1 COEFFICIENT
    DW G23, G24              ; STAGE 2 COEFFICIENT

    DW GN3, GN4              ; STAGE N COEFFICIENT

AØ :DW AØ1, AØ2     ...AØN

2X TEMPORARY STORAGE FOR N STAGES

X : DW  N  DUP(Ø)            ; INPUTS TO STAGES
L11 DW  N  DUP(Ø)            ; L1(k) OR L1(k-1)
T12 DW  N  DUP(Ø)            ; L2(k) OR L2(k-1)
L3  DW  N  DUP(Ø)            ; L3(k)
```

## 4.6 SUMMARY

In this chapter, all the other realization techniques discussed in Section 1.3 are used to implement a second order module. The necessary equations for the Algorithm are derived and the various subroutines implementing the main flow chart of Fig. (4.1) written. Finally, two cross coupled structures are used to implement the same flow chart of Fig. (4.1).

# CHAPTER - V

# CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

## 5.1 CONCLUSIONS

Digital filters have been implemented in harwired logic, special purpose computers and general purpose computers. The recent advent of 16-bit microcomputers with built in multiplication hardware has created a new option for implementing Digital filters. A typical 16-bit microcomputer, Intel 8086 has been selected here. There is a significant improvement in the sampling rate because of the availability of multiplication instruction in the Instruction set of 8086 microprocessor.

In this dissertation, the various realization techniques for Digital filters are discussed and their characteristics compared. All the Direct form structures suffer extreme coefficient sensitivity as N, the order of the filter grows large. In order to avoid coefficient sensitivity, the Digital filter Transfer function is implemented as a cascaded or parallel combination of second order modules. The second order module in itself can be any one of the Direct form structures.

The salient features of the Intel 8086 microprocessor has been studied in considerable detail. Using the Instruction set of 8086 microprocessor the Assembly language programs for the various realizations of a second order module written. The mathematical equations for the corresponding algorithms are derived prior to the R.T.L. flowchart model. Main programs for the cascade and parallel Digital filters are also written making use of the various subroutines and each stage in itself is a second order 1D structure.

## 5.2 SUGGESTIONS FOR FUTURE WORK

Because of the non-availability of the Intel 8086 microcomputer kit the various programs could not be tested. These programs can be tested as and when a kit is made available in the Department.

All of 16-bit microprocessors viz. Intel 8086 Motorola MC 68000, Texas Instruments TMS 9900, Zilog Z8000, Fairchild 9445 are similar in basic word size and arithmetic capabilities. The programs written down using the Intel 8086 microprocessor can be modified for the rest of the microprocessors.

In this dissertation Intel 8086 microprocessor is used to implement Digital filters - one section of Digital signal processing. It can be extended to other sections of Digital signal processing. This leads to the microcomputer study of Sampled Data Systems.

## R E F E R E N C E S

1. Rabiner, L. R. and Gold, B. 'Theory and Application of Digital Signal Processing', Englewood Cliffs, N.J.; Prentice-Hall, Inc., 1975.

2. Bogner, R. E. and Constantinides, A. G. 'Introduction to Digital Filtering', New York : John Wiley, 1975.

3. Tretter, S. A. 'Introduction to Discrete Time Signal Processing', New York : John Wiley, 1976.

4. Oppenheim, A. V. and Schafer, R.W. 'Digital Signal Processing', Englewood Cliffs, N. J. : Prentice Hall, Inc., 1975.

5. Lam, H. Y-F. 'Analog and Digital Filters - Design and Realisation', Englewood Cliffs, N. J. : Prentice Hall, Inc., 1979.

6. Kaiser, J. F. 'Digital Filters' in Kuo, F. F. and Kaiser, J. F. eds., System Analysis by Digital Computer, Chap. 7, New York : John Wiley, 1966.

7. Rader, C. M. and Gold, B. 'Digital Filter Design Techniques in the Frequency Domain', Proc. IEEE, Vol. 55, Feb. 1967, pp. 149-171.

8. Rabiner, L. R., Cooley, J. W. et.al 'Terminology in Digital Signal Processing' IEEE Trans. on Audio Electroacoust., Vol. AU-20. Dec. 1972, pp 322-377.

9. Jackson, L. B., Kaiser, J. F., and Mc donald, H. S. 'An approach to the Implementation of Digital Filters', IEEE Trans. on Audio Electroacoust., Vol. AU-16, Sept. 1968, pp. 413-421.

10. Rabiner, L. R., Kaiser, J. F., Herrmann, O. and Dolan, M. T. 'Some Comparisons between FIR and IIR Digital Filters'. Bell System Tech. J. Vol. 53, 1974, pp. 305-331.

11. Fettweis, A. 'A general Theorem for Signal Flow Networks with Applications', 'Arch. Elek. Übertragung' Vol. 25 1971. pp. 557-561. Reprinted in Lawrence, R. R. & Rader, C. M. 'Selected Papers in Digital Signal Processing' New York : John Wiley, 1972.

12. Seviora, R. E. and Sabaltash, M. 'A Tellegen's Theorem for Digital Filters', IEEE Trans. Circuit Theory, Vol. CT-18, Jan. 1971, pp 201-203.

13. Weissberger, A. J. 'MOS/LSI Microprocessor Selection', Electronic Design, Vol. 22 No. 12, June 1974.

14. Ogdin, J. L. 'Microcomputers : Promises and Practices', IEEE Intercon. 74, Session 17, N.Y., March 1974.

15. Cushman, R. H. 'The Intel 8080 : First of the Second Generation Microprocessors', EDN, Vol. 19, No. 9, May 1974.

16. George, A. 'Get minicomputer feature at ten times the 8080 speed with the 8086', Electronic Design, Vol. 26, Sept. 1978, pp. 60-66.

17. Katz, B. J., Morse, S. P., Pohlam, W. P. and Bruce, W. R. '8086 microcomputer bridges the gap between 8 and 16 bit designs', Electronics International, Vol. 51, Feb. 1978, pp. 99-104.

18. Morse, S. P., Pohlman, W. B., and Ravenel, B. W. 'The Intel 8086 microprocessor, a 16 bit Evolution of the 8080' IEEE Trans. Computer, Vol. 11 No. 6, June 1978,

19. Morse, S. P., Ravenel, B. W., Mazor, S. and Pohlman, W. B., 'Intel Microprocessors - 8008 to 8086', IEEE ~~Trans.~~ Computer, Vol. 13 No 6 Feb. 1980, pp. 42-60.

20. Garland, M. 'Introduction to Microprocessor System Design', Tokko: Mc Graw Hill Kogakusha, 1979.

21. Mathur, A. P. 'Introduction to Microprocessors', New Delhi : Tata Mc Graw Hill, 1980.

22. MCS-86 User's Manual, Intel Corp., Santa Clara, CA, 1979.

23. Neale, D. F., and Wilson, D. R. 'Application of Microprocessors in Signal Processing', Warsaw, Polish-English Seminar on Real Time Process Control, Jan. 1977.

24. Nagle, Jr., H. T., and Carrol, C. C. 'Organizing a Special Purpose Computer to realize Digital Filters for Sampled data Systems', IEEE Trans. Audio Electroacoust.. Vol. AU-16, Sept. 1978.

25. Nagle, Jr., H. T. and Nelson, V. P., 'Digital Filter Implementation on 16-bit Microcomputers', IEEE Micro, Feb. 1981, pp. 23-41.

26. Hwang, S. Y. 'An Optimization of Cascade Fixed Point Digital Filters', IEEE Trans. Circuits and Systems, Vol. CAS-21, No. 5, Jan. 1974, pp 163-168

27. Lee, W. S. 'Optimization of Digital Filters for Low Round off Noise,' IEEE Trans. Circuits and Systems, Vol. CAS-21, No. 5, May 1974, pp. 423-431

28. Liu, B. and Peled, A. 'Heuristic Optimization of the Cascade Realization of Fixed Point Digital Filters IEEE Trans. Acoustics, Speech and Signal Processi Vol. ASSP-23, No. 5, Oct. 1975, pp. 464-473.

29. Nagle, Jr., H. T., and Carroll, C. C. 'Realization of Digital Controllers', Proc. IFAC Symp. Auto Control in Space, Armenia, USSR, Aug. 1974, pp. 23-35.

30. Jackson, L. B., Lindgren, A. G., and Kim, Y. "Optimal Synthesis for Second Order State Space Structure for Digital Filters", IEEE Trans. Circuits and Systems, Vol. CAS-26, No. 3, March 1979, pp. 149-152.

APPENDIX - I

## CLASSIFICATION OF DIGITAL FILTERS

The term Digital filtering refers to a computational algorithm performed on a sampled input signal resulting in a transformed output signal. The computational process can correspond to high pass filtering, low pass filtering, band pass filtering, integration, differentiation etc. The process is assumed to be linear, that means the principle of superposition applies to the input output relationship. The input signal is a sequence of numbers from either an Analog-to-Digital (A/D) converter or a direct digital input source. The output signal is either a direct digital sequence or a regenerated analog signal from a Digital-to-Analog (D/A) converter.

The unique advantages offered by Digital filters are :

1. The performance from unit to unit is stable and repeatable.
2. Arbitrarily high precision is achieved that is limited only by the number of bits carried in memory and by the input and output resolution capabilities
3. No impedance matching problems exist in the digital domain.
4. Critical filter frequency can be placed without restriction but it influences the required precision
5. Component value variation problems are non existent
6. Greater flexibility is achieved since filter response can be changed by varying the proper arithmetic coefficients,

7. The intrinsic possibility of time sharing major implementation section exist.

8. Small size results from integrated circuit implementation.

9. Periodic calibration as is required with analog circuits is eliminated.

10. Performance limitations of physical analog components are avoided.

Two general types[1,5,8] of configurations of Digital filters are :

(a) Recursive Digital filter

(b) Non-Recursive Digital filter

The Recursive Digital filter is a discrete time filter which is realized via a recursion relation. It means the output samples of the filter are explicitly determined as a weighted sum of past output samples as well as past and/or present input samples. For example

$$y(n) = a_0.x(n) + a_1.x(n-1) + a_2.x(n-2) - b_1.y(n-1) - b_2.y(n-2) \quad \ldots \quad (A1.1)$$

The Non-Recursive Digital filter is a discrete time filter for which the output samples of the filter are explicitly determined as a weighted sum of past and present input samples only. For example,

$$y(n) = a_0.x(n) + a_1.x(n-1) + a_2.x(n-2) \quad \ldots \quad (A.1.2)$$

Thus Recursive Digital filters are those filters which possess a transfer function as given by equation (1.3)

$$D(z) = \frac{\sum_{i=0}^{M} A_i . Z^{-1}}{1 + \sum_{i=1}^{N} B_i . Z^{-1}} \quad .... \quad .... \quad (A1.3)$$

It has all common factors cancelled, The denominator coefficients are identically nonzero. The zeros and poles are located on the $Z^{-1}$ plane. The Non Recursive Digital filters however possess a transfer function which is a polynomial of $Z^{-1}$ and all common factor in equation (1.3) are cancelled. this case, the transfer function is of the form

$$D(z) = D_0 + D_1 . Z^{-1} + D_2 . Z^{-2} \ldots D_N \; Z^{-N} \;.. \quad (A1 \cdot 4)$$

This equation is a finite degree polynomial, no poles can appear in any finite part of the $Z^{-1}$ plane. Non Recursive filter, as a result is always stable. (This of course is consistent with the absence of feed back).

Consider the general transfer function of equation (A1.3) which is reproduced here for convenience in the factor■d form.

$$D(z) = \frac{\prod_{i=1}^{M} (1 - Z_i \; Z^{-1})}{\prod_{i=1}^{N} (1 - p_k \; Z^{-1})} \quad \ldots \quad (A1.5)$$

where $p_1$, $p_2$ ... $p_N$ are the poles and $Z_1$, $Z_2$ ... $Z_M$ are the zeros.

Any filter whose transfer function is given by (A1.5) with $N \geq 1$ is called an infinite impulse response (IIR) digital filter, because there does not exist a finite integer L such that

$$d(n) = 0 \quad \text{for } n > L$$

where d(n) is the impulse response of the filter. For IIR digital filters assume $M \leq N$. This assumption holds true for almost all cases of practical interest. A partial fraction expansion of (A1.5) is

$$D(z) = \xi_0 + \frac{\xi_1}{1 - p_1 Z^{-1}} + \frac{\xi_2}{1 - p_2 \cdot Z^{-1}} + \cdots \frac{\xi_M}{1 - p_N \cdot Z^{-1}}$$

where $\xi_0 = \alpha$ if $N = M$

$= 0$ if $N > M$

and $\xi_i = (1 - p_i Z^{-1}) D(z)/Z = p_i$ for $i = 1, 2 \ldots N$

Hence, the corresponding impulse response is

$$h(n) = \left[\xi_1 \cdot p_1^{n} + \xi_2 \cdot p_2^{n} + \ldots \xi_N \cdot p_N^{n}\right] u(n) + \xi_0 \cdot \delta(n)$$

Clearly, the necessary and sufficient conditions for the impulse response above to satisfy the stability criteria of

$$\sum_{n=-\infty}^{\infty} |h(n)| < \infty$$

is that

$$|p_i| < 1 \text{ for } i = 1, 2 \; .... \; N$$

That is, all the pole locations of the digital filter are within the unit circle in the Z-plane.

When the transfer function of a Digital filter is given by equation

$$D(z) = D_o + D_1 . Z^{-1} + ...... \; D_M . Z^{-M} \qquad ... \qquad (A1.6)$$

which is equivalent to the case when $N = 0$, the Digital filter is said to be of finite impulse response (FIR) type. This name is used because the impulse response of equation (A1.6) has the property that $h(n) = 0$ for $n > M$ and for $n < 0$. That is the corresponding impulse response is of finite duration. In this case, there are no poles and this type of filter is always stable.

From the above it is clear that Transfer function in equation (A1.3) represents an IIR Digital filter while the Transfer function of equation (A1.4) represents an FIR Digital filter. The FIR filters are all stable and casual while the IIR filter is stable if the poles of $D(z)$ are within the unit circle in the Z plane, and casual if $B_o$ is the first nonzero coefficient in the denominator. As we are concerned with casual filters it is convenient to assume $B_o = 1$.

The major differences are listed in[10] . IIR Digital filters cannot have perfect linear phase characteristics while FIR filters are always designed to have linear phase characteristic. Implementation of an FIR Digital filter requires more computations and more digital components; hence FIR filters are more expensive than IIR filters The amount of computation and hardware needed to perform a filtering process is usually an important practical consideration. In general IIR Digital filters require lesser computations and/or hardware to achieve a particular filtering function than those required by the corresponding FIR Digital filters. FIR Digital filters are called for to perform tasks not possible and/or not practical by IIR Digital filters such as linear phase filters, and multirate filters where the input signals and the corresponding output signals are sampled at different rates.

Although IIR Digital filters are generally realized recursively and FIR filters nonrecursively, IIR filters can be realized nonrecursively and FIR filters can be realized recursively.

## PRINCIPLE OF TRANSPOSE AND TRANSPOSITION THEOREM

Tellegen's theorem is an important basic theorem of conventional network theory. As Digital filter networks are not subject to Kirchoff's laws, Tellegen's theorem in its most general form does not apply. A restricted form of Tellegen's theorem referred to as the difference form, can be derived [5,11,12] rom this a number of useful properties of digital networks can be developed. In classical networks Tellegen's theorem is in the form of a relationship between the voltage distribution in one network and the current distribution in a second network, where the only relationship between the networks is that they have the same topology but otherwise unrelated. In a similar manner, if we consider every flowgraph to have a branch in each direction between every pair of nodes, with the transmission of some of the branches being zero, then any two flow graphs with the same number of nodes can be considered to be topologically equivalent.

Consider two signal flow-gfaphs with the same topology Let N denote the number of network nodes. The network node variables, branch outputs and source node values in the first network are denoted by $w_k$, $v_{jk}$ and $x_j$ respectively and in the second network by $w'_k$, $v'_{jk}$ and $x'_j$. Then, the Tellegen's theorem is

$$\sum_{k=1}^{N} \sum_{j=1}^{N} (w_k . v'_{jk} - w'_k . v_{jk}) + \overset{N}{\underset{k=1}{}} (w'_k . x_k - w'_k x_k) = 0 \qquad \ldots \quad (A2.1)$$

Proof: The proof of equation (A2.1) follows almost directly from the definition of a signal flow graph. The branch outputs are related to the node variables and source inputs by

$$w_k = \sum_{j=1}^{N} V_{jk} + \sum_{j=1}^{M} S_{jk} \qquad \ldots \quad (A2.2)$$

Adopting the convention that each network is drawn in such a way that each network node has associated with it a source node connected to it by a branch with unity transmittance. Also, this source node is not connected to any other network nodes.

With the convention regarding source nodes equation (A2.2) changes to

$$w_k = \sum_{j=1}^{N} v_{jk} + x_k \qquad \ldots . \quad (A2.3)$$

Writing the identify

$$\overset{N}{\underset{k=1}{}} (w_k . w'_k - w'_k . w_k) = 0 \qquad \ldots \quad (A2.4)$$

Equation (A2.1) follows in a straightforward manner by substituting equation (A2.3) into equation (A2.4).

Equation (A2.1) is referred to as Tellegen's theorem for signal flow graphs or for digital filters. If variables $W_k$ $W'_k$, $V_{jk}$, $V'_{jk}$, $X_k$ and $X'_k$ are derived through a linear operation from $w_k$, $w'_k$, $v_{jk}$, $v'_{jk}$, $x_k$ and $x_k$ respectively, then

$$\sum_{k=1}^{N} \sum_{k=1}^{N} ( W \cdot V'_{jk} - W'_k \cdot V_{jk}) + \sum_{k=1}^{N} (W_k \cdot X'_k - W'_k \cdot X_k) = 0 \qquad \ldots \quad (A2.5)$$

Thus Tellegen's theorem applies either to the sequence values or to the Z-transforms.

For passive analog networks consisting of interconnections of resistors, inductors and capacitors, the notion of reciprocity plays an important role. For digital networks there exists corresponding notions of reciprocity and inter-reciprocity consider a given network excited by two different sets of sources. The Z-transforms of the source node values for the two different sets will be denoted by $X_k$ and $X'_k$. The value of the node variables of the k'th node when the network is excited by the unprimed sources will be denoted by $W_k$. When the network is excited by the primed sources, this variable will be denoted by $W'_k$. The network is said to satisfy reciprocity if for any two signal distributions.

$$\sum_{k=1}^{N} ( W_k \cdot X'_k - W'_k \cdot X_k) = 0 \qquad \ldots \quad (A2.6)$$

As a consequence of reciprocity, if we excite the graph at network node 'a' and observe the output at node 'b'; then for a reciprocal graph, the same excitation at node 'b' will result in the same output at node 'a'.

Most digital networks are not reciprocal. A related concept that is more useful with regard to digital network is that of interreciprocity. In this case we consider two distinct signal flow graphs. Let $X_k$ denote the source node values and $W_k$ denote the node variables for one network and $X'_k$ and $W'_k$ the source node values and network node variables for the second network. Then the two networks are said to be interreciprocal if

$$\sum_{k=1}^{N} (W_k X'_k - W'_k X_k) = 0 \quad \ldots \quad (A2.7)$$

Equations (A2.7) is similar to equation (A2.6), here for reciprocity the primed and unprimed network differ only in the sources, whereas for interreciprocity both the sources an and branch transmittances can differ in the primed and unprimed networks. A network that is reciprocal is also interreciprocal with its lf.

TRANSPOSITION THEOREM

A property of digital networks is that they are interreciprocal with their transpose. The transpose of a flow graph is generated by reversing the directions of all the branches but leaving their transmittances the same.

Consider a digital network where $W_k$ denotes the node variable for the k'th node. The transmission from node 'j' to node 'k' is denoted by '$F_{jk}$'.

$$S_o . V_{jk} = F_{jk} . W_j$$

In the transposed network, the node variable of the k'th node is denoted by $W'_k$ and the branch transmittance between nodes 'j' and 'k' is denoted by $F'_{jk}$, so that

$$V'_{jk} = F'_{jk} . W'_j$$

By definition of the transposed network $F'_{jk} = F_{jk}$

To prove that a network and its transpose are inter-reciprocal - i.e. to show that equation (A2.7) holds for the above conditions, we utilize the fact that a network and its its transpose have the same topology so that Tellegen's theorem equation (A2.5) holds.

Thus,

$$\sum_{j=1}^{N} \sum_{k=1}^{N} (W_k . V'_{jk} - W'_k . V_{jk}) + \sum_{k=1}^{N} (W_k . X'_k - W'_k . X_k) = 0$$

... (A2.8)

Substituting value of $V_{jk}$ and $V'_{jk}$ in (A2.8) we obtain

$$\sum_{j=1}^{N} \sum_{k=1}^{N} (W_k . W'_j . F'_{jk} - W'_k . W_j . F_{jk})$$

$$+ \sum_{k=1}^{N} (W_k . X'_k - W'_k . X_k) = 0$$

or $$\sum_{j=1}^{N} \sum_{k=1}^{N} W_k . W'_j . F'_{jk} - \sum_{j=1}^{N} \sum_{k=1}^{N} W'_k . W_j . F_{jk}$$

$$+ \sum_{k=1}^{N} ( W_k . X'_k - W'_k - W'_k . X_k ) = 0 \quad \ldots \text{(A2.9)}$$

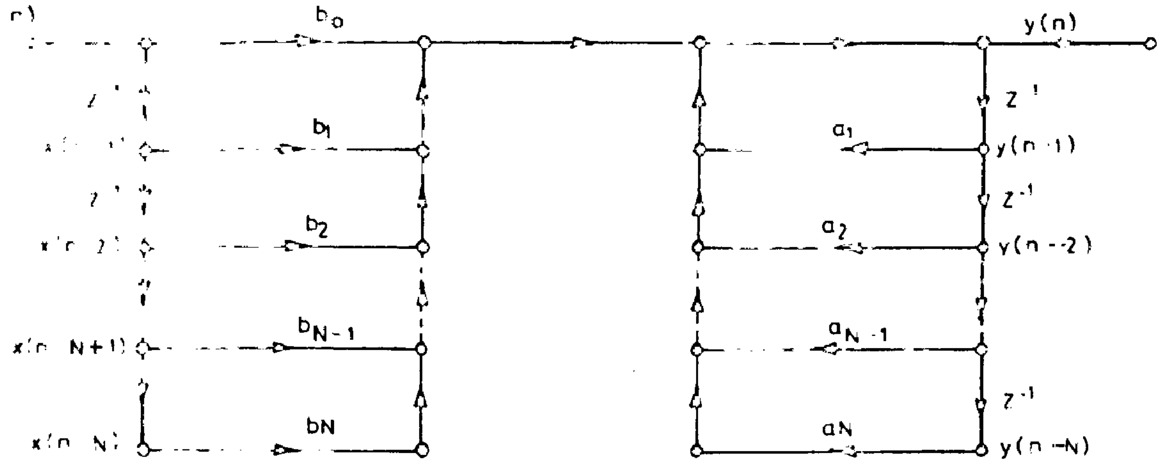
Interchanging the indicies of summation in the first double sum of equation (A2.9).

$$\sum_{j=1}^{N} \sum_{k=1}^{N} ( W'_k . W_j . F'_{jk} - W'_k . W_j . F_{jk} )$$

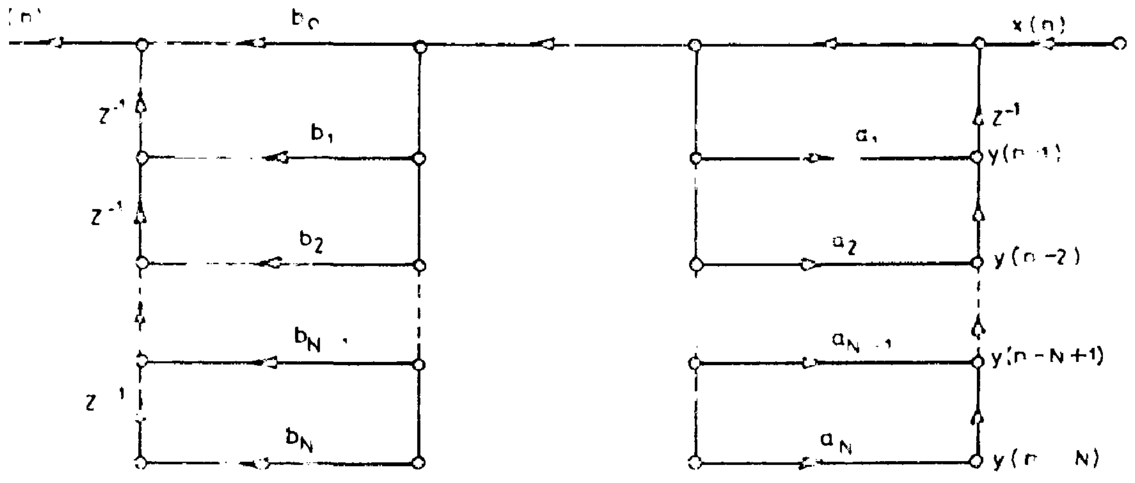
$$+ \sum_{k=1}^{N} ( W_k . X'_k - W'_k . X_k ) = 0 \quad \ldots \text{(A2.10)}$$

Since, the primed and unprimed networks are transposes, $F'_{jk} = F_{jk}$, and therefore the double sum is zero and $\overset{N}{\underset{k=1}{}} ( W_k . X'_k - W'_k . X_k ) = 0$, which proves that a network and its transpose are interreciprocal.

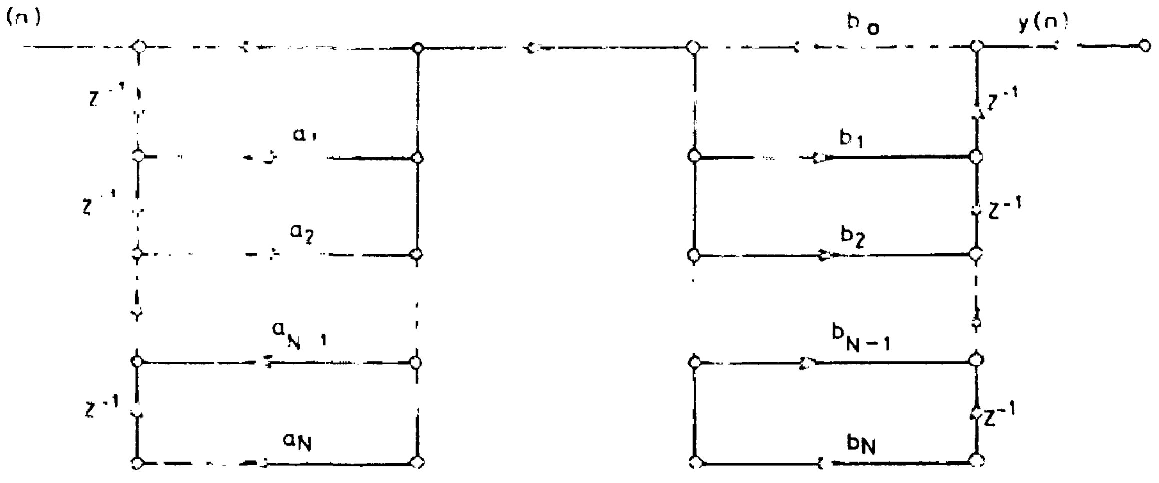
For single input - single output networks, a network and its transpose have the same transfer function. For a 2nd order section the diagramatic changes are shown in Fig. (A2.1).



(a) Nth ORDER SINGLE INPUT OUTPUT NETWORK



(b) SIGNAL FLOW AND INPUT OUTPUT REVERSED



(c) TRANSPOSED Nth ORDER NETWORK

FIG. A 2.1

APPENDIX - III

## RISE OF INTEL MICROPROCESSOR 8008 TO MICROPROCESSOR 8086 AND THEIR COMPARISON

The Intel 8008 [14,17,19] was the first 8 bit, monolithic, p channel MOS device to be developed. The 8008 processor architecture is quite simple compared to that of today's microprocessors. The instruction set is small but symmetrical with only a few operand addressing modes available. The addressable memory space is 16K bytes which seemed to be lot back in 1970 when memories were expensive and LSI devices were slow. The memory size limitation was imposed by the lack of available pins.

The microprocessor does not have instructions with direct addresses since two CPU registers must be used to reference main storage. Also, some operations such as moving data from one place in storage to another, are somewhat awkward. Another problem area is that associated with an interrupt. Interrupt processing was not a requirement for the 8008, only the most primitive mechanism conceivable - not incrementing the program counter was provided. Such a mechanism permits an interrupting device to jam an instruction into the processor's instruction stream. Since memory is addressed during the interrupt, two of the scratchpad registers are to be reserved as interrupt registers. This reduces the effective number of the registers in the scratch pad file from seven to five. There is no instruction for disabling the interrupt mechanism; thus this function must be realized with external hardware. Finally, the single 8 bit bus into processor requires a large amount of support hardware. If a single IC is produced which will replace these components, this processor will be valuable in many more applications.

The Intel 8080 [16,17,19] an 8 bit, monolithic, n channel MOS device is a second generation microprocessor with many improvements over its predecessor, the 8008 [15] The 8080 was the first processor designed specifically for the microprocessor market. The main objective of the 8080 was to obtain a ten-to-one improvement in throughput eliminate many of the 8008 shortcomings that had by 1973 become apparent and provide new processing capabilities not found in the 8008. The latter included handling of the 16 bit data types, BCD arithmetic, enhanced operand addressing modes, and improved interrupt processing. Memory costs had come down and processing speed was approaching TTL, so larger memory spaces seemed more practical and direct addressing of more than 16K bytes was achieved. Symmetry was not a goal because the benefits to be gained from making the extensions symmetric would not have justified the resulting increase in chip size and opcode space. Most of the external logic required to support the 8008 is on the 8080 CPU, and all the important interfacing signals are generated on designated processor pins.

The 8080 architecture is significantly different from that of 8008. The byte handling facilities are augmented with a limited number of 16-bit facilities. The memory space is 64K bytes, the address bus 16 bits wide, so an entire address can be sent down the bus in one memory cycle. The 8080 extends the 32 port capacity of the 8008 to 256 input ports and 256 output ports. The 8080 processor contains a file of seven 8-bit general registers, a 16-bit program counter and stack pointer and five 1-bit flags. ~~The stack is contained in memory~~ (RAM) instead of on the chip, a strategy which removes the restriction of only seven levels of nested subroutines. The programmer can directly access the stack pointer in 8080, unlike in the 8008. A fifth flag, Auxiliary Carry, augments the 8008 flag set. It indicates whether a carry was generated out of the four low order bits. This flag, in conjunction with a decimal adjust instruction, makes possible packed BCD addition. The 8080 includes the entire 8008 instruction set as a subset. The added instructions provide some new operand addressing modes and some 16 bit data manipulation facilities. The 8080 has an interrupt mechanism identical to that of 8008, but includes instructions for enabling and disabling the mechanism.

The 8080 is packaged in a 40 pin DIP and has separate address and data buses having tnstate outputs. As a result of the separate data and address buses, a micro computer is formed with as few as six TTL packages. A disadvantage, however, is that 8080 requires three separate power supplies.

In 1976 advances in technology allowed Intel to consider enhancing the 8080. The objective was a processor set utilizing a single power supply and requiring fewer chips (the 8080 required both an oscillator chip and a system controller chip). The new processor, called the Intel 8085 was constrained to be compatible with the 8080 at the machine-code level. This meant that extensions to the instruction set could use only the 12 unused opcodes of the 8080. Architecturally, the 8085 turned out to be not much more than a repackaging of the 8080. The major differences were added features such as on chip oscillator, power on reset, vectored interrupts, decoded control lines, a serial I/O port and a single power supply. Two new instructions RIM and SIM, were added to handle the serial port and the interrupt mask.

The Intel 8086 was designed to provide an order of magnitude [17,18,19] increase in processing throughput over the 8080. The processor was to be compatible with the 8080 at the assembly language level, so that existing 8080 software could be reassembled and correctly executed on the 8086. To allow this, the 8080 register and instruction set were to appear as logical subsets of the 8086 registers and instructions.

The goals of the 8086 architecture were symmetric extension of existing 8080 features and the addition of processing capabilities not found in the 8080. New features and capabilities included 16 bit arithmetic, signed 8-16-bit arithmetic (including multiply and divide), efficient interruptible byte string operations, improved bit manipulation facilities and mechanisms to provide for re-entrant code, position-independent code, and dynamically relocatable programs. By 1977 memory had become inexpensive and microprocessors were being used in applications requiring large amounts of code and data. Another achievement was the direct addressing of more than 64K bytes and support of multiprocessor configurations.

The 8086 processor architecture comprises a memory structure, a register structure, an instruction set, and an external interface. The 8086 external interface consists of interrupts, multiprocessor synchronization and resource sharing, this all goes way beyond the facilities provided in the 8080. The 8086 can access upto one million bytes of memory and upto 64K input/output ports. The I/O space consists of 64 K ports a 256-fold increase over the 8080. The processor contains a total of thirteen 16-bit registers and nine 1-bit flags. The 8080 register set is a subset of the 8086 register set as shown in Fig. (A3.1). The 8086 instruction set is not a superset of the 8080/8085 instruction set. Most of the 8080/8085 instructions are included in the 8086 while some of the infrequently used ones (e.g. conditional calls and returns) are not. The operand addressing modes of the 8080 have been greatly enhanced. Significant new operations includes : (a) multiplication and division of signed and unsigned binary numbers as well as unpacked decimal numbers, (b) move, scan and compare operations for strings upto 64 K bytes in length, (c) non-destructive bit testing, (d) byte translation from one code to another, (e) software generated interrupts and (f) a group of instructions that can help coordinate the activities of multiprocessor systems.

The mere six years of microprocessor evolution has yielded a three orders of magnitude performance improvement. TABLE - 2.1 and TABLE 2.2 trace the comparison of these processors in respect of features, and technology.

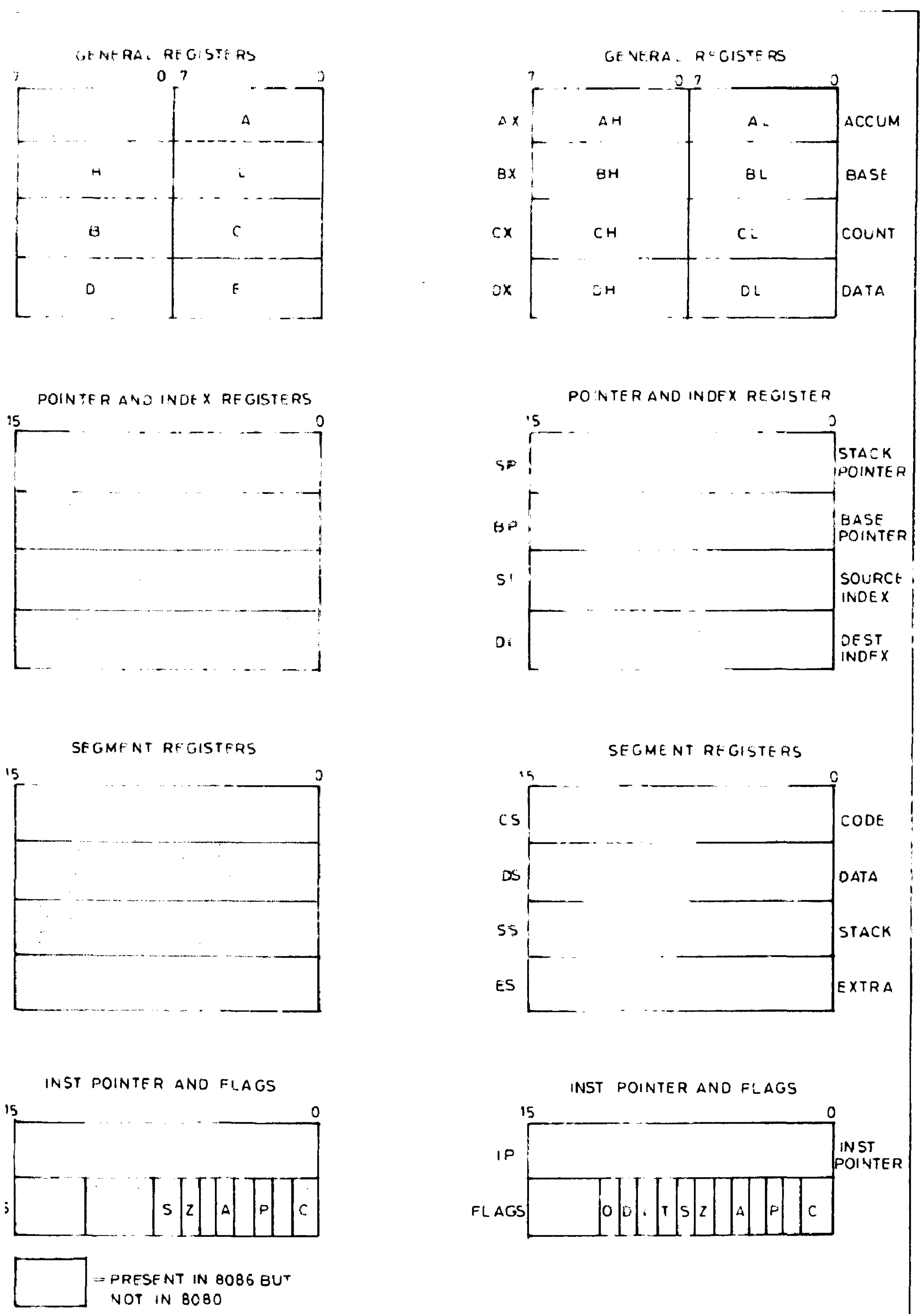


FIG A3.1 8080 SUBSET OF 8086 REGISTER

TABLE - A2.1

FEATURE COMPARISON - INTEL MICROPROCESSORS 1972-78

| | FEATURE | 8008 | 8080 | 8085 | 8086 |
|---|---|---|---|---|---|
| 1. | Introduction date | 1972 | 1974 | 1976 | 1978 |
| 2. | No. of instructions | 48 | 72 | 85 | 97 |
| 3. | No. of flags | 4 | 5 | 5 | 9 |
| 4. | Maximum memory size | 16 K bytes | 64 K bytes | 64 K bytes | 1 M bytes |
| 5. | I/O ports | 8 input<br>24 output | 256 input<br>256 output | 256 input<br>256 output | 64 K input<br>64 K output |
| 6. | No. of pins | 18 | 40 | 40 | 40 |
| 7. | Address Bus width | 8* | 16 | 16 | 20* |
| 8. | Data Bus width | 8* | 8 | 8 | 16* |
| 9. | Data types | 8 bit unsigned | 8 bit unsigned<br>16 bit unsigned Ltd.<br>Packed BCD Ltd. | 8 bit unsigned<br>16 bit " Ltd.<br>Packed BCD Ltd. | 8-16 bit unsigned<br>8-16 bit "<br>Packed BCD<br>Unpacked BCD |

TABLE - A2.1 (Contd.)

| | FEATURE | 8008 | 8080 | 8085 | 8086 |
|---|---|---|---|---|---|
| 10. | Addressing modes | Register<br>Immediate** | Memory direct Ltd.<br>Memory indirect Ltd.<br>Register<br>Immediate** | Memory direct Ltd.<br>Memory indirect Ltd.<br>Register<br>Immediate** | Memory direct<br>Memory indirect<br>Register<br>Immediate<br>Indexing |

* Address and Data bit multiplexed

** Memory can be addressed as a special case by using register M.

TABLE - A2.2

TECHNOLOGY COMPARISON - INTEL MICROPROCESSORS 1972-78

| | CHARACTERISTICS | 8008 | 8080 | 8085 | 8086 |
|---|---|---|---|---|---|
| 1. | Silicon gate technology | P-Channel Enhancement Load Device | N-Channel E.L.D | N-Channel Depletion Load Device | Scaled N-Channel (HMOS) D.L.D. |
| 2. | Clock Rate | 0.5 — 0.8 MHz | 2 — 3 MHz | 3 — 5 MHz | 5 — 8 MHz |
| 3. | Minimum gate delay | 30 nanos | 15 ns | 5 ns | 3 ns |
| 4. | Typical speed power product | 100 pj | 40 pj | 10 pj | 2 pj |
| 5. | Approximate Nos. of Transistors | 2000 | 4500 | 6500 | 20000 |
| 6. | Average Transistor density | 8.4 | 7.5 | 5.7 | 2.5 |

APPENDIX - IV

INSTRUCTION SET OF 8086 MICROPROCESSOR

The instruction set of 8086 microprocessor can be studied under the following heads :

(A) Data Transfer Instructions

(B) Arithmetic Instructions

(C) Bit Manipulation Instructions

(D) String Instructions

(E) Program Transfer Instructions

(F) Processor Control Instructions

The fourteen data transfer instructions can be studied under the following four heads. The flags in this case remain unaltered.

(A) DATA TRANSFER INSTRUCTION

(a) General Purpose

1. MOV Move byte or word

| | | | | |
|---|---|---|---|---|
| (a) | Reg / memory, reg | 100010dw | mod reg r/m | |
| (b) | Reg16/memory16, Seg.reg | 100011d0 | mod 0 reg r/m | |
| (c) | Acc, memory | 101000dw | Addr low | Addr high |
| (d) | Reg, immed | 1011w reg | data | data if w=1 |
| (e) | Reg/ Mem, immed | 1100011w | mod 000 r/m data | data if w= |

| | | |
|---|---|---|
| 2. | PUSH | PUSH word onto stack |
| (a) | Register | 01010 reg |
| (b) | Seg-reg (CS legal) | 000reg110 |
| (c) | Memory 16/reg16 | 11111111 mod110r/m |
| 3. | POP | POP word off stack |
| (a) | Register | 01011reg |
| (b) | Seg-reg (CS-illegal) | 000reg111 |
| (c) | Memory/reg | 10001111 mod000r/m |
| 4. | XCHG | Exchange byte or word |
| (a) | Reg/mem with register | 1000011w modregr/m |
| (b) | Reg, acc | 10010 reg |
| 5. | XLAT | Translate byte |
| (a) | Translate byte to AL | 11010111 |

(b) Input/Output

| | | |
|---|---|---|
| 6. | IN | Input byte or word |
| (a) | Acc, immed | 1110010w Port |
| (b) | Acc, DX | 1110110w |
| 7. | OUT | Output byte or word |
| (a) | Acc, immed | 1110011w Port |
| (b) | Acc, DX | 1110111w |

(c) Address Object

| | | | | |
|---|---|---|---|---|
| 8. | LEA | Load Effective Address | | |
| | (a) | Load EA to reg | 10001101 | modregr/m |
| 9. | LDS | Load pointer using DS | | |
| | (a) | Load pointer to DS | 11000101 | modregr/m |
| 10. | LES | Load pointer using ES | | |
| | (a) | Load pointer to ES | 11000100 | modregr/m |

(d) Flag Transfer

| | | | |
|---|---|---|---|
| 11. | PUSHF | Push flags onto stack | |
| | (a) | Load AH with flags | 10011100 |
| 12. | POPF | Pop flags off stack | |
| | (a) | Pop flags | 10011101 |
| 13. | SAHF | Store AH register in flags | |
| | (a) | Store AH into flags | 10011110 |
| 14. | LAHF | Load AH register from flags | |
| | (a) | Load AH with flags | 10011111 |

B. ARITHMETIC INSTRUCTIONS

8086 arithmetic operations may be performed on 4 types of numbers unsigned, binary, signed binary, unsigned packed decimal and unsigned unpacked decimal. Following is the effect of the flags.

CF Carry flag : ADC and SBB incorporate the Carry flag in their operations. The Carry flag is set(a) if an addition results in a carry out of the high order bit of the result and (b) if a subtraction results in a borrow into the high order bit of the result. Otherwise the CF is cleared.

AF Auxiliary Carry flag : The AF is set (a) if an addition results in a carry out of the lower order half byte of the result and (b) if a subtraction results in a borrow into the lower order half byte of the result. The AF is provided for the decimal adjust instructions.

SF Sign flag : Arithmetic and Logical instruction set the Sign flag equal to the high order bit (7 or 15) of the result. Programs performing unsigned operations ignore SF.

ZF Zeroflag : If the result of an Arithmetic or Logical Operation is zero, the ZF is set, otherwise ZF is cleared.

PF Parity flag : If the low order eight bits of an Arithmetic or Logical result contains an even number of 1-bits, then the PF is set, otherwise it is cleared. It also checks ASCII characters for correct parity.

OF Overflow flag : If the result of an operation is too large a positive number, or too small a negative number to fit in the destination operand (excluding the sign bit) then the OF is set, otherwise it is cleared. OF indicates signed arithmetic overflow.

(a) Addition

1. ADD Add byte or word

(a) Reg /memory with register to either
000000dw modregr/m

(b) Immed , reg/mem 100000sw mod000r/m Data

(c) Immed , acc 0000010w Data Data w = 1

2. ADC Add byte or word with carry

(a) Reg / mem with register to either
000100dw modregr/m

(b) Immed , reg / mem 100000sw mod010r/m Data

(c) Immed , acc 0001010w Data Data w =1

3. INC Increment byte or word by one

(a) Register 01000reg

(b) Reg/mem 1111111w mod000r/m

4. AAA ASCII adjust for addition

(a) ASCII adjust for add 00110111

5. DAA Decimal adjust for addition

(a) Decimal adjust for add 00100111

(b) Subtraction

6. SUB Subtract byte or word

(a) Reg/mem and register to either

001010dw modregr/m

| | | |
|---|---|---|
| (b) | Immed , reg/mem | 100000sw mod011r/m Data |
| (c) | Immed , acc | 0010110w Data Data if w=1 |

7. SBB Subtract byte or word with borrow

| | | |
|---|---|---|
| (a) | Reg/memory and register to either | 000110dw modregr/m |
| (b) | Immed , reg/mem | 100000sw mod011r/m Data |
| (c) | Immed , acc | 0001110w Data Data if w=1 |

8. DEC Decrement byte or word by one

| | | |
|---|---|---|
| (a) | Reg/mem | 111111wmod001r/m |
| (b) | register | 01001reg |

9. NEG Negate byte or word

| | | |
|---|---|---|
| (a) | Change sign | 1111011w mod011r/m |

10. CMP Compare byte or word

| | | |
|---|---|---|
| (a) | Reg/mem , reg | 001110dw modregr/m |
| (b) | Immed , reg/mem | 100000sw mod111r/m Data |
| (c) | Immed , acc | 0011110w Data Data if w=1 |

11. AAS ASCII Adjust for subtraction 00111111

12. DAS Decimal adjust for subtraction 00101111

(c) Multiplication

13. MUL Multiply byte or word unsigned
1111011w mod100r/m

14. IMUL Integer multiply byte or word
1111011wmod101r/m

15. AAM ASCII Adjust for multiply
11010100 00001010

(d) Division

16. DIV Divide byte or word unsigned
1111011w mod110r/m

17. IDIV Integer divide byte or word
1111011w mod111r/m

18. AAD ASCII adjust for division
11010101 00001010

19. CBW Convert byte to word 10011000

20. CWD Convert word to double word 10011001

(c) <u>BIT MANIPULATION INSTRUCTIONS</u>

8086 provides three groups of bit manipulating instructions.

(a) Logicals - Here NOT has no effect on the flags. AND OR, XOR, TEST affect the flags as : The OF and CF are always cleared, the contents of the AF is always undefined following execution of a logical instruction. The SF, ZF, PF are always posted to reflect the result of operation and can be tested by conditional jimp instruction.

1. NOT Invert 1111011w mod010r/m

2. AND 'And' byte or word

(a) Reg/mem , register

001000dw modreg/rm

(b) Immed , reg/mem
1000000w mod100r/m Data

(c) Immed , Acc
0010010w Data Data if w = 1

3. OR 'Inclusive or' byte or word

(a) Reg/mem ,register
000010dw modregr/m

(b) Immed , reg/mem
100000w mod001r/m Data

(c) Immed , acc
0000110w Data Data if w = 1

4. XOR 'Exclusive or' byte or word

(a) Reg/mem , register
001100dw modregr/m

(b) Immed , reg/mem
1000000w mod110r/m Data

(c) Immed , acc 0011010w Data Data if w = 1

5. TEST 'Test' byte or word

(a) Reg/mem , register
1000010w modregr/m

(b) Immed , reg/mem
1111011w mod000r/m Data

(c) Immed , acc
0011010w Data Data if w = 1

(b) Shift - Bits are shifted arithmetically or logically. Upto 255 shifts may be performed according to the value of the count eperand coded in the instruction. The count may be specified as a constant 1, or as reg. CL allowing the shift count to be a variable supplied at execution time. Arithmetic shifts may be used to multiply or divide binary numbers by powers of two. Logical shifts can be used to isolate bits. Shift instructions affect the flags as follows: AF is always undefined following a shift operation. PF, SF, ZF are updated. CF contains the value of the last bit shifted out of the destination operand. OF is undefined following a multibit shift. In a single bit shift, OF is set if the value of the high order (sign) bit retains the original value, otherwise OF is cleared.

| | | |
|---|---|---|
| 6. | SHL/SHA or word | Shift logical arithmetic left byte 110100vw mod100r/m |
| 7. | SHR | Shift logical right byte or word 110100vw mod101r/m |
| 8. | SAR | Shift arithmetic right byte or word 110100vw mod111r/m |

(c) Rotate - Here the CF may act as an extension of the operand in two of the rotate instructions, allowing a bit to be isolated in CF and then tested by a jump if carry or jump if not carry instruction.

| | | |
|---|---|---|
| 9. | ROL | Rotate left byte or word<br>110100vw mod000r/m |
| 10. | ROR | Rotate right byte or word<br>110100vw mod001r/m |
| 11. | RCL | Rotate through carry left byte or word<br>110100vw mod010r/m |
| 12. | RCR | Rotate through carry right byte or word<br>110100vw mod011r/m |

D. <u>STRING INSTRUCTIONS</u>

String instructions do not use the normal memory addressing modes to access their operands. Instead Index registers are used implicitly. Following are the string instructions which allow strings of bytes or words to be operated on, o ne element at a time.

| | | | |
|---|---|---|---|
| 1. | MOVS | Move byte or word string | 1010010w |
| 2. | CMPS | Compare byte or word string | 1010011w |
| 3. | SCAS | Scan byte or word string | 1010111w |
| 4. | LODS | Load byte or word string | 1010110w |
| 5. | STOS | Store byte or word string | 1010101w |
| 6. | REP | Repeat | 1111001z |

E. <u>PROGRAM TRANSFER INSTRUCTIONS</u>

The sequence of execution of instructions in a program is determined by the CS & IP. CS contains the base address of the current code segment (64 K portion of memory) from which instructions are presently being fetched. IP is used as an offset from the beginning of the code sediment. The combination of CS & IP points to the memory location from which the next instruction is to be fetched.

The program transfer instructions operate on the IP and CS thereby changing their contents; This changing causes **normal sequential** execution to be altered . When a program transferred occurs the queue no longer contains the correct instructions and the BIU obtains the next instructions from memory using the new IP and CS values. and passes the instruction directly to the EU and then begins refilling the queue from the new locations. The flags are not effected except in interrupt related instructions.

(a) Unconditional Transfer

(1) CALL Call procedure

(a) Direct within segment

```
11101000  Disp low    Disp High
```

(b) Indirect within segment

```
11111111  mod010r/m
```

(c) Direct inter segment

```
10011010 offset low   offset high
         seg low      seg high
```

(d) Indirect intersegment

```
11111111  mod011r/m
```

2. RET Return from procedure

(a) within segment 11000011

(b) within seg. adding Immed. to SP
11000010 Data low
Data High

(c) Intersegment 11001011

(d) Interseg. adding immed. to SP
11001010 Data low
Data high

3. JMP Unconditional jump

(a) Direct within segment
11101001 displacement low
displacement high

(b) Direct within seg. - short
11101011 disp.

(c) Indirect within segment
11111111 mod100r/m

(d) Direct intersegment
11101010 offset low offset high
seg. low seg. high

(e) Indirect intersegment
11111111 mod101r/m

(b) Conditional Transfer

4. JO Jump if overflow 01110000 Disp

5. JNO Jump if not overflow 01110001 displacement

6. JB/JNAE/JC Jump on below/not above or equal/carry
01110010 displacement

| | | |
|---|---|---|
| 7. | JNE/JAE/JNC | Jump if not equal/not carrý/above or equal |
| | | 01110101 disp |
| 8. | JE/JZ | Jump if equal/zero |
| | | 01110100 disp |
| 9. | JL/JNGE | Jump if less/not greater or equal |
| | | 01111100 disp |
| 10. | JLE/JNG | Jump if less or equal/not greater |
| | | 01111110 disp |
| 11. | JBE/JNA | Jump if above or equal/not above |
| | | 01110110 disp |
| 12. | JP/JPE | Jump if parity/parity even |
| | | 01111010 disp |
| 13. | JS | Jump if sign |
| | | 01111000 disp |
| 14. | JNL/JGE | Jump if not less/greater or equal |
| | | 01111101 disp |
| 15. | JNLE/JG | Jump if not less or equal/greater |
| | | 01111111 disp |
| 16. | JNBE/JA | Jump if not below or equal/above |
| | | 01110111 disp |
| 17, | JNP/JPO | Jump if not parity/parity odd |
| | | 01111011 disp |
| 18. | JNS | Jump if not sign |
| | | 01111001 disp |

| | | | | |
|---|---|---|---|---|
| 19. | LOOP | Loop CX times | 11100010 | disp |
| 20. | LOOPZ/LOOPE | Loop while zero/equal | 11100001 | disp |
| 21. | LOOPNZ/LOOPNE | Loop while not zero/not equal | 11100000 | disp |
| 22. | JCXZ | Jump on CX zero | 11100011 | disp |

(d) Interrupt

| | | | | |
|---|---|---|---|---|
| 23. | Type specified | | 11001101 | type |
| 24. | Type 3 | | 11001100 | |
| 25. | INTO | Interrupt on overflow | 11001110 | |
| 26. | IRET | Interrupt return | 11001111 | |

PROCESSOR CONTROL INSTRUCTION

These instructions allow programs to control various CPU functions. There are three groups (a) Flag operation - this updates flags, (b) External Synchronization - used for synchronizing the 8086 with external events, (c) No operation causes CPU to do nothing. Except for the flag operation none of the processor control instructions affect the flags.

| | | | | |
|---|---|---|---|---|
| 1. | CLC | Clear carry | 11111000 | |
| 2. | CMC | Complement carry | 11110101 | |
| 3. | STC | Set carry | 11111001 | |
| 4. | CLD | Clear direction | 11111100 | |
| 5. | STD | Set direction | 11111101 | |
| 6. | CLI | Clear interrupt | 11111010 | |
| 7. | STI | Set interrupt | 11111011 | |
| 8. | HALT | Halt | 11110100 | |
| 9. | WAIT | Wait | 10011011 | |
| 10. | ESC | Escape to external device | 11011xxx | modxxxr/m |
| 11. | LOCK | Bus lock prefix | 11110000 | |

Consider multiplication in the two's complement number system. An n-bit multiplicand (perhaps a signal variable) is multiplied by an n-bit multiplier (perhaps a filter coefficient). The product has 2n bits and may be used as another multiplicand in a later multiplication so it is quantized (truncated here) back to n bits.

Suppose the multiplier is 'X' and the coefficient is 'a', then

$$\begin{array}{r} X \\ * \quad a \\ \hline aX \end{array}$$

and the product is quantized $Q\,[\;]$:

$$Q\left[aX\right] = \left\lfloor aX / 2^n \right\rfloor$$

The computer hardware actually handles the integers of equation (15.1) so that in hardware

$$\begin{array}{r} X * 2^{n-1} \\ * \quad a * 2^{n-1} \\ \hline aX * 2^{2n-2} \end{array}$$

is quantized as $Q\left[aX\right] = \left\lfloor aX * 2^{2n-2} / 2^n \right\rfloor = \left\lfloor aX * 2^{n-2} \right\rfloor$

Consequently, the product must be multiplied by 2 (shifted one place left) so that the final truncated term is

$$Q\left[aX\right] = \left\lfloor aX * 2^{n-1} \right\rfloor$$

Code sequence in Intel 8086 programing language is as

```
LEA     SI , A        ;   COEFF. POINTER
LODW                  ;   A / 2  LOADED
IMUL    X             ;   AX / 4 IN DX REGISTER
SAL     DX , 2        ;   AX IN DX REGISTER
```

APPENDIX - V

## COEFFICIENT REPRESENTATION

Intel 8086 represents all the numbers in the two's complement number system, so

$$N = (S\ M_{14}\ M_{13}\ \ldots\ M_1\ M_0)\ 2\ \text{cns} \quad \ldots (A5.1)$$

where,

$$-2^{15} \leq N \leq 2^{15} - 1$$

If we consider all numbers to be scaled, such that

$$N = (S.\ M_{14}\ M_{13}\ \ldots\ M_1\ M_0)\ 2\ \text{cns}$$

thus,

$$-1 \leq N \leq 1 - 2^{-15}$$

As a result, coefficients in the range $1 \leq N < 2$ cannot be represented. Therefore, all coefficients will be stored as half their actual value, and

$$\text{VALUE STORED} = \lfloor \text{Value} * 2^{14} + 0.5 \rfloor$$

and a left shift (multiply by 2) operation will be performed in each routine to compensate for this change. The symbol $\lfloor X \rfloor$ means the largest integer less than X.

As an example suppose coefficient SØ = .4383164 is to be stored in the Intel 8086. microprocessor.

$$\begin{aligned} \text{VALUE STORED} &= \lfloor SØ * 2^{14} + .5 \rfloor \\ &= \lfloor .4383164 * 16384 + .5 \rfloor \\ &= \lfloor 7181.3759 + .5 \rfloor \\ &= \lfloor 7181.8759 \rfloor = 7181 \end{aligned}$$

The following operations are actually performed

(1) Load the coefficient 'a' into AX register

$$AX = a * 2^{n-2}$$

(2) Multiply by the variable 'X'

$$\begin{aligned} DX,\ AX &= (a * 2^{n-2}) * (X * 2^{n-1}) \\ &= a\,X * 2^{2n-3} \\ &= (a\,X\,/\,4) * 2^{2n-1} \end{aligned}$$

the product is now in the DX, AX register.

(3) Shift DX Register left 2 places (quantize to 16 bits and multiply by 4).

$$\begin{aligned} DX &= \lfloor (a\,X\,/\,4) * 2^{2n-1}\,/\,2^{n} \rfloor * 4 \\ &= \lfloor a\,X\,/\,4 * 2^{2n-1} \rfloor * 4 \\ &= a\,X * 2^{n-1} \end{aligned}$$

The operation left justifies the register DX and fills in two zeros in the least significant bits. The DX register now contains the truncated, properly scaled result.

On computers with double register shifting, one would perform the double left shift first.

$$\begin{aligned} \text{Double register} &= (a\,X\,/\,4 * 2^{2n-1}) * 4 \\ &= a\,X * 2^{2n-1} \end{aligned}$$

and then truncate to the n most significant bits.

$$\begin{aligned} \text{For Single register} &= (a\,X * 2^{2n-1})\,/\,2^{n} \\ &= (a\,X * 2^{n-1}) \end{aligned}$$

which is more accurate than above.