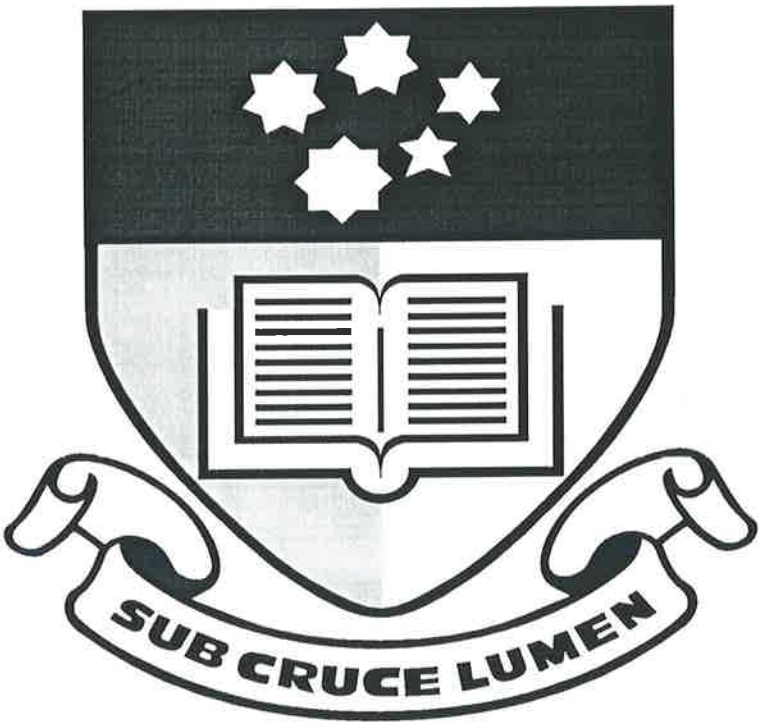
# VLSI IMPLEMENTATION

# OF

# HEART SOUNDS

# MAXIMUM ENTROPY SPECTRAL ESTIMATION




Mohammad Saeed Tahmasbi, B.Eng.

Thesis submitted for the degree of

## MASTER OF ENGINEERING SCIENCE

in the Department of Electrical Engineering
and Applied Mathematics
The University of Adelaide
Adelaide, Australia

December 1994

Awarded 1995

# Contents

# List Of Figures

# List Of Tables

# Publications And Abstracts

During the course of my study the following paper has been presented at a learned society:

An oral presentation "*Architecture And Design Of 16-Bit Processor; Maximum Entropy Spectral Analyser.*"

Authors: M. S. Tahmasbi, K. Eshraghian and J. Mazumdar.

Engineering And The Physical Sciences In Medicine Conference, Sept. 12-15 1994, Perth, Australia.

Co-Sponsored By:

Australasian College Of Physical Sciences & Engineering In Medicine.

College Of Biomedical Engineers, Institution Of Engineers, Australia.

Society For Medical & Biological Engineering (WA) Inc.

# Declaration

I declare that this thesis contains no material which been accepted for the award of any other degree or diploma in any university.

To the best of my knowledge and belief, this thesis contains no material previously published or written by any other person, except where due reference is given in the text of the thesis.

I consent to this thesis being made available for photocopying or loan.

*SIGNED* ..... *DATE* 13/12/94 .....

# Acknowledgements

I wish to express my appreciation to my supervisors, Prof. K. Eshraghian and Prof. J. Mazumdar for their generous encouragement and continual support which has led to this research. I also wish to extend my sincere gratitude to my friend Mr. Alireza Moini for his extensive co-operation and our many discussions in various stages of the design and test of this project.

My thanks are also extended to Dr. A. Bouzerdoum from whom I learned the first principles and theories of statistical signal processing. Last but not least my thanks go to Dr. N. Burgess for reading, correction and constructive suggestions on my project progress reports.

# List Of Mathematical Symbols

$H(U)$ = entropy of partition U

$P(E)$ = probability of event $E$

$\langle X(n) \rangle$ = time average of random variable $X(n)$

$\langle X_n X_{n+m} \rangle$ = time autocorrelation sequence

$E[X_n]$ = expected value of $X(n)$

$R_{XX}(m)$ = autocorrelation function of $X(n)$

$B$ = bias of an estimator

$\sigma_\zeta^2$ = variance of estimator $\zeta$

$H(x_1, x_2, \ldots, x_r)$ = joint entropy of several random variables

$f(x)$ = probability density function of $X$

$f(x_1, \ldots, x_r)$ = joint density of random variables

$H_X$ = entropy rate of stationary process $X(n)$

$H(z)$ = system transfer function

$S_{YY}(\omega)$ = power spectrum of process $Y(n)$

$FPE(m)$ = final prediction error criterion

$AIC(m)$ = Akaike information criterion

spectral analysis, instrumentation, and so on. The aim of this study is to provide a more efficient tool for the spectral analysis of the heart sounds, the third heart sound in particular. In recent times the stethoscope has been reinstated to its rightful important position because the information gained from electrocardiography (ECG), phonocardiography (PCG), and echocardiography offers solid proof of the significance of the heart sounds and murmurs, making the discipline of cardiac ausculation even more valuable as a diagnostic tool. However, the main deficiency of using traditional stethoscope, now an integral part of medicine, in medical diagnosis lies in the limited frequency recognition and intensity resolution of human ear. Although human ear is capable of functioning over a broad range of frequency variation, unfortunately the usual range of cardiac sound is too low-pitched to be easily heard. In addition, not everyone has the same capacity to hear sounds. Some examiners will be able to detect vibrations of extremely low frequencies, while others will not. The problem becomes more obvious considering the addition of environmental noise and other unwanted biological signals like breathing sound, etc.

More advanced technology can be applied to overcome these limitations. One obvious application of technology to the heart sound analysis has been spectral analysis. Although the Fast Fourier Transform (FFT) methods provide a good solution for the spectral analysis of the first and second heart sounds, still when it comes to the analysis of biological signals of short duration like third heart sound, it suffers from limitation in frequency resolution. However, the efficiency of maximum entropy spectral estimation in producing sharper and more pronounced peaks in the power density spectrum has proved satisfactory in dealing with short length of data records.

Due to the shortcomings of the F.F.T. technique in the spectral analysis of the sampled signals of short duration, it was decided to design a special purpose processor for spectral anal-

ysis based on "Maximum Entropy" method. The processor will have application in different areas, but it was particularly designed to work as the central processing unit of the "Biomonitor system", to be used for spectral analysis of the heart sounds. Parallelism and pipelining was the focus for the architecture of the processor. A separate multiplier/accumulator, and a high-speed ALU speeds the program. In addition to the direct speed increase provided by fast multiplication in hardware, using the pipelining technique, the next input can be loaded while the previous product is being calculated. Parallelism on the other hand, helps when more identical operations, performed at the same time, could speed progress.

This project deals with the design methodologies of the architecture of a 16-bit processor based on "Maximum Entropy" method which provides a suitable platform for further study in the field of spectral analysis of the heart sounds.

## 1.2 Physical Characteristics Of Cardiac Structure

The function of the heart is to transfer sufficient blood from the low-pressure venous system to the arterial side of the circulation under the proper pressure to maintain the circulatory needs of the body. The heart is an efficient force pump that few, if any, mechanical pumps can equal without "downtime" for maintenance. In an engineering sense, the heart is made up of two separate pump systems [28]. The right atrium and ventricle act as a single unit (the right heart) to move venous blood from the great veins to the pulmonary circulation. The left atrium and ventricle (the left heart) act together in a similar manner to pump blood from the pulmonary system to the high-pressure system circulation. This directional flow of blood, as well as some of the important structural features of the heart, are shown in Fig. 1.1. The terms "right heart" and "left heart", although physiologically correct, are not descriptive of their position in the body. Due to the normal rotation of the heart on its longitudinal axis, the right ventricle is in front of the left and occupies a position immediately behind the sternum,

whereas the left ventricle is rotated so that it faces toward the left side and the back of the thorax [33].

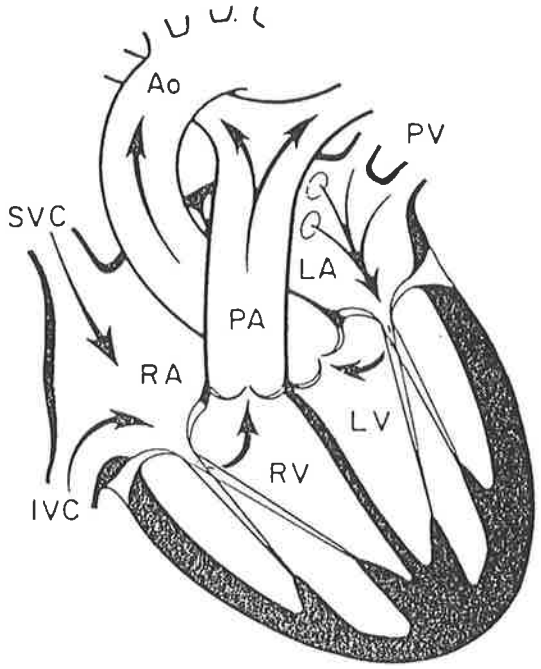



**Figure 1.1--Blood flow through the heart. SVC= superior vena cava; IVC= inferior vena cava; RA= right atrium; LA= left atrium; PA= pulmonary artery; PV= pulmonary veins; RV= right ventricle; LV= left ventricle; Ao= aorta [17].**

## 1.3 Heart Valves

The dynamic effect of cardiac contraction is surprisingly effective in moving blood through the heart even in the absence of the competent valves. Nevertheless, these thin, delicate valves, which guard the entrance and exit of each ventricle, greatly enhance the efficiency of the cardiac pump. The heart valves are mechanical devices that permit the flow of blood in one direction only. The two cuspid (atrio-ventricular) valves are located between the atria and ventricles, while the two semi-lunar valves are located at the entrance to the pulmonary artery and the great aorta [21]. In spite of their fragile appearance, the cusps of these valves are deceptively strong and resilient. Their movements are essentially passive even though some of the leaflets have been shown to contain muscle fibres. The crescent-shaped (semilu-

nar) cusps of the pulmonic and aortic valves permit these structures to open maximally during ventricular ejection and still provide a perfect seal when closed during diastole [33].

## 1.4 Mechanical Events Of The Cardiac Cycle [15]

Activation of the myocardium is followed by cardiac contraction. In the intact heart, this leads to a series of events that are associated with its function as a pump. It is convenient to relate these activities to the changes in pressure that take place inside the chambers of the heart and the great vessels during the cardiac cycle. Representative pressure pulses from the left atrium, left ventricle, and aorta are diagrammatically shown in Fig. 1.2, along with a graphic representation of the electric activity of the heart lines. The left heart pressure relationships are discussed in the analysis of the cardiac cycle that follows. In general, the pressure relationships on the right side of the heart are the same as those shown for the left side, although the pressures will be lower. The mechanical events are also similar on both sides of the heart [15].

Blood flows from an area of higher pressure to an area of lower pressure. The pressure developed in a heart chamber is related primarily to the chamber's size. For example, if the chamber size decreases, the pressure increases. The pressure in the atria is called atrial pressure, that in the ventricles is called ventricular pressure, and pressure in the aorta and pulmonary trunk is referred to as arterial pressure. In a normal heartbeat, the two atria contract while the two ventricles relax. Then, when the two ventricles contract, the two atria relax. The term systole refers to the phase of contraction; diastole is the phase of relaxation. A cardiac cycle, or complete heartbeat, consists of a systole and a diastole of both atria plus the systole and diastole of both ventricles.

The cardiac cycle represents a combination of mechanical, electrical and valvular events whose interrelationship is complex but essential to understanding of how the heart functions

and how disease processes affect it. At rest, the normal adult heart beats at a rate of about 70 to 75 per minute. Blood flows from the atria to the ventricles and from the ventricles to the large arteries at a velocity which is determined by the pressure differences between the chambers. Normally the valves offer no resistance and open or close as a function of the relative pressures exerted by the flowing stream and the energy imparted by the contractions of the atrial and ventricular musculature.

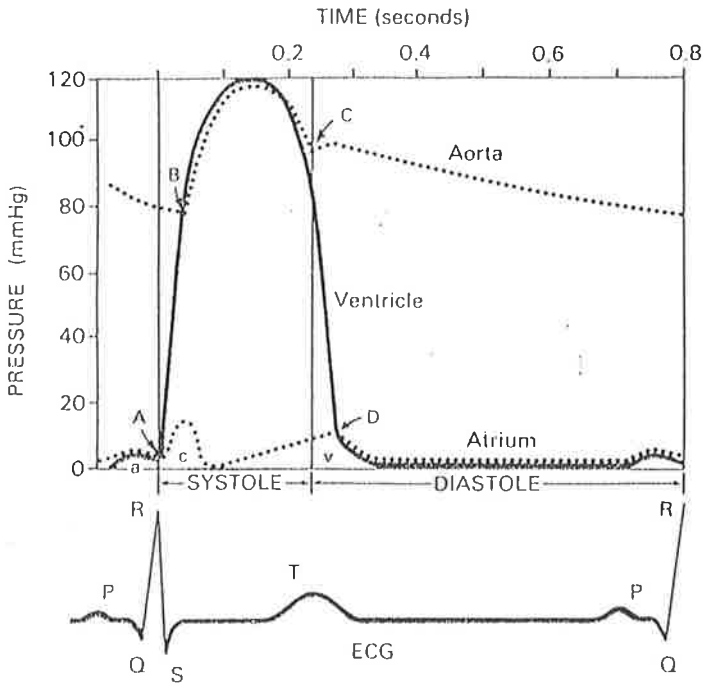



**Figure 1.2--Diagram showing a representative pressure pulse from aorta, left ventricle, and left atrium. An ECG also is shown [31].**

At a rate of 75 per minute, the complete cycle for filling and emptying of the chamber would occupy 0.8 sec or 800 msec (Fig. 1.2). The cardiac cycle is divided into systole and diastole. Left ventricular systole (i.e., the contractile period of the left heart) extends from the early rise of the ventricular pressure and the closure of the AV valve (Fig. 1.2.A) to the closure of the aortic valve and the beginning of diastole (Fig. 1.2.C). During most of the period from B to C, the ventricular pressure is higher than the aortic, the aortic valve is open and the ventricle ejects blood into the arterial system. At C the aortic valve closes.

Diastole, which is the period of ventricular relaxation and filling, begins with the closing of the aortic valve (C). When the ventricular pressure falls below the atrial, the AV valve opens (D) and the ventricle begins to fill. Diastole ends when the ventricle again contracts and the new cycle begins.

## 1.5 Electrical Events Of The Cardiac Cycle

In order for the cardiac muscle to contract, there must be a preceding action potential which initiates the electrical and ionic events that culminate in ventricular systole. The ECG, which is recorded at the body surface, is a graphical representation of the summed voltage changes produced by electric depolarization and repolarization of the heart. These electrical impulses begin at the sinoatrial (SA) node in the right atrium, spread over the entire heart and initiate the contraction wave. The electrical phase of the cardiac cycle begins with excitation of the atrium (i.e., atrial depolarization), denoted on the ECG by an initial upward positive deflection called the P wave (Fig. 1.2), which triggers atrial contraction.

After completion of the P wave, the ECG trace returns to base level, i.e., the isoelectric line. About 0.16 to 0.22 seconds following the onset of the P wave, a second series of negative and positive waves are seen [33]. A negative Q wave usually precedes a positive R and a negative S wave. This QRS complex is caused by electrical depolarization of the ventricles and is quickly followed by ventricular contraction. After a short interval, a positive T wave appears which corresponds to repolarization of the ventricular muscle mass. The ECG then returns to the isoelectric line and usually there is electrical silence for the remainder of diastole. The S-T segment is an important phase of the record because it is specifically distorted in some heart diseases.

It should be emphasized that the action potential is the indispensable forerunner to cardiac contraction. The heart has a spontaneous, intrinsic rhythmicity and automaticity, and contrac-

tion is inevitably coupled to excitation.

## 1.6 Heart Sounds [34]

The cardiac structure vibrations associated with cardiac mechanical events generate acoustic waves that are transmitted to the chest. These acoustic waves are usually classified in four different groups, known as heart sounds, and contain information of the vibratory source. The two primary heart sounds are usually heard as a "lup-dup", a low-pitched first sound, followed by a quicker, higher-pitched second. The intensity of heart sounds, as heard at chest wall, depends upon several factors, i.e., the rate of the rise of the ventricular pressure, the physical characteristics of the ventricles and valves, the volume contained in the heart, the position of the AV valve leaflets at the beginning of ventricular systole and the transmission characteristics of the chest wall. The relationship of the heart sounds to other events of the cardiac cycle is shown in Fig. 1.3. The major components of the heart sounds are associated with the abrupt acceleration and deceleration of blood in and near the heart, but there is not full agreement on the relative significance of valve activity and muscle vibration.

**The First Heart Sound (S1)** is associated with the closure of the mitral and tricuspid valves at the start of ventricular systole and the two components can sometimes be distinguished. If so, the first component of S1 is mitral in origin and the second component tricuspid.

**The Second Heart Sound (S2)** is usually of higher frequency and shorter duration than the first. It marks the end of ventricular systole and the beginning of diastole and is associated with the closure of the semilunar valves. It consists of two components, aortic and pulmonic. Normally, the aortic valve closes several milliseconds before the pulmonic and the time difference is accentuated during the inspiratory phase of respiration.

This respiratory delay in closing of the pulmonary valve produces a "physiological" split-

ting of the second heart sound and is mainly due to the sudden decrease in intrathoracic pressure associated with inspiration. This in turn, causes a temporary increase in venous return, and an increase in right heart volume resulting in increased right ventricular output, a temporary prolongation of ejection time and a delay in pulmonary valve closure. At the same time, pulmonary venous return to the left heart is diminished so that left ventricular stroke volume decreases and the aortic valve closes slightly earlier.

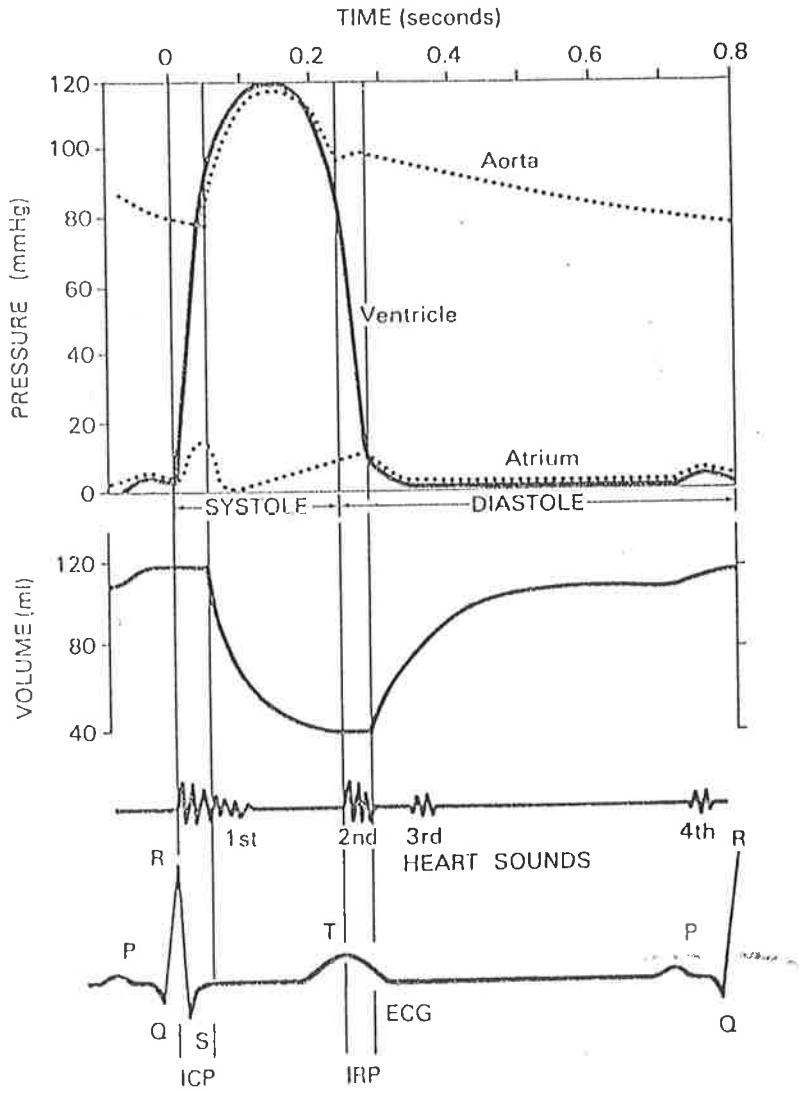



**Figure 1.3--Mechanical and electrical events of the cardiac cycle showing also ventricular volume curve and heart sounds [31].**

**A Third Heart Sound (S3),** shown in Fig. 1.3, is associated with the passive rapid-filling phase. Because of the thinness of their skin of the chest, a physiological third sound may be

present and audible in younger individuals; however, if it occurs after the age of 40 years, it is generally considered abnormal. It may occur in fever, cardiac failure and certain other cardiac disorders.

**The Fourth Heart Sound (S4)** is associated with the active rapid filling phase (Fig. 1.3). While it can often be recorded by phonocardiography, it is generally not audible. When it does occur, it is usually recorded at the peak of atrial contraction and may be associated with increased atrial pressures.

## 1.7 Heart Murmurs

Disturbances of normal blood flow patterns in the heart and great vessels often result in abnormal sound, producing vibrations in the auditory frequency range known as murmurs. They are classified on the basis of their timing as systolic, diastolic and continuous murmurs. If the aortic or pulmonary valve is diseased or deformed, the increased turbulence through the narrowed or distorted orifice results in the systolic murmur characteristic of aortic or pulmonary valve disease [34].

If the AV closure is incomplete because of disease of the mitral or tricuspid valves, the valve will become incompetent and blood will regurgitate into the atrium producing a blowing "whoosh" noise following the first heart sound. If this systolic murmur persists throughout systole it is sometimes referred to as a pansystolic or holosystolic murmur.

Abnormal heart sounds which occur during diastole are associated either with an abnormality of AV valve opening (usually mitral) or an abnormality of semilunar valve closure (usually aortic). A murmur originating at the aortic valve and heard in early diastole may be produced by incomplete closure of the aortic valve at the end of systole. Such an abnormality could be due to fibrosis or stiffening of the valve in the open position or destruction of valve

leaflets. The defect causes regurgitation of blood back into the left ventricle at the end of systole through the incompetent valve, producing the diastolic murmur of aortic regurgitation or aortic insufficiency.

Stenosis of the mitral valve may cause abnormal heart sounds during early or late diastole. The early component of this murmur is often initiated with an opening snap of the mitral valve; the late component may be associated with the atrial systole, just before the onset of ventricular systole, and is referred to as a "presystolic" murmur.

## • Summary

This chapter introduced the basic operation of the heart with a focus on the generation of heart sounds. First the cardiac structure and its physical characteristics were presented. In subsequent sections, mechanical and electrical events in the cardiac cycle were discussed in detail. Finally the main heart sounds, classified in four different categories, were introduced and their relationships with other mechanical or electrical events of the cardiac cycle were reviewed briefly. Furthermore, this chapter gave an overview on the objectives, as well as the significance, of this study. The mathematical approach for the spectral analysis of the heart sounds was mentioned here and is developed in the next chapter, which results in an AR model for the signal and a recursive algorithm.

# Chapter 2

# MAXIMUM ENTROPY SPECTRAL ESTIMATION

## 2.1 Introduction

The Maximum Entropy Method (MEM) for spectral analysis was suggested by Burg (1967) [5, 6]. Its mathematical properties have been discussed in detail by Lacoss (1971) [16], Burg (1972), and Ulrych (1972) who found that the MEM is, in general, superior to the more conventional methods of spectral estimation [36].

The application of entropy can be divided into two categories. The first deals with problems involving the determination of unknown distributions. The available information is in the form of known expected values or other statistical functions, and the solution is based on the principle of maximum entropy: we determine the unknown distribution so as to maximize the entropy $H(U)$ of some partition $U$ subject to the given constraints. In the second category, we are given $H(U)$ and wish to construct various random variables so as to maxi-

mize their expected values. The solution involves the construction of optimum mappings of the random variables under consideration, into the given probability space.

The probability $P(E)$ of an event $E$ can be interpreted as a measure of our uncertainty about the occurrence or non-occurrence of $E$ in a single performance of the experiment $S$. If $P(E)$ is close to one, then we are almost certain that $E$ will occur; if $P(E)$ is close to zero, then we are reasonably certain that $E$ will not occur; our uncertainty is maximum if $P(E) = 0 \cdot 5$. If $U$ is a partition of $S$, i.e., $U$ is a collection of mutually exclusive events $E_i$ whose union equals $S$, then the measure of uncertainty about $U$ will be denoted by $H(U)$ and will be called the entropy of the partition.

The function $H(U)$ must satisfy a certain number of conditions. The following is a typical set of such conditions [C.E. Shannon and W. Weaver] [32]

1. $H(U)$ is a continuous function of $P_i = P(E_i)$
2. if $P_1 = \ldots = P_N = 1/N$, then $H(U)$ is an increasing function of $N$.
3. If a new partition $B$ is formed by subdividing one of the sets of $U$, then $H(B) \geq H(U)$.

It can be shown that the sum

$$H(U) = -P_1 \log P_1 - \ldots - P_N \log P_N \qquad \text{(EQ 1)}$$

satisfies these conditions and it is unique within a constant factor.

## 2.2 Uncertainty, Information, And Principle Of Insufficient Reason

In the heuristic interpretation of entropy, the number $H(U)$ is a measure of our uncertainty about the events $E_i$ of partition $U$ prior to the performance of the experiment. If the experiment is performed and the results concerning $E_i$ become known, then the uncertainty is removed. We can therefore say that the experiments provides information about the events $E_i$

equal to the entropy of their partition. Thus uncertainty equals information and both are measured by the sum in (EQ 1).

If $E_i$ are $N$ events of a partition $U$ of $S$ and nothing is known about their probabilities, then

$$P(E_i) = 1/N$$

Although conceptually the maximum entropy principle is equivalent to the principle of insufficient reason, operationally the maximum entropy method simplifies the analysis drastically when, as in the case in most applications, the constraints are phrased in terms of probabilities in the space $S^n$ of repeated trials. In such cases the equivalence still holds, although less obvious.

The maximum entropy method is thus a valuable tool on the solution of applied problems. It is used, in fact, even in deterministic problems involving the estimation of unknown parameters from insufficient data. The maximum entropy principle is then accepted as a smoothness criterion [23].

## 2.3 Basic Concepts Of Estimation Theory

The characterization of a random process is usually made by its averages [24]. However, it is often necessary to estimate averages of the random process model from a single sample sequence of the random process, i.e., a sequence $X(n)$ which is assumed to be a realization of a random process defined by the set of random variables $\{X(n)\}$. The time average of a random process is defined as

$$\langle X(n) \rangle = \lim_{N \to \infty} \frac{1}{2N+1} \sum_{n=-N}^{N} X(n)$$

Similarly, the time autocorrelation sequence is defined as

$$\langle X_n X_{n+m} \rangle = \lim_{N \to \infty} \frac{1}{2N+1} \sum_{n=-N}^{N} X(n)\, X^*(n+m)$$

It can be shown that the above limits exist if $\{X(n)\}$ is a stationary process with finite mean. A random process is said to be stationary in the strict sense if all its probability functions or statistical averages are independent of time. As defined above, the time averages are functions of an infinite set of random variables and thus are viewed as random variables themselves. However, under a condition known as ergodicity, the time averages defined by above equations are equal to statistical averages. That is

$$\langle X(n) \rangle = E[X_n] = m_X$$

and

$$\langle X(n)\, X^*(n+m) \rangle = E[X_n X^*_{n+m}] = R_{XX}(m)$$

In general, a random process for which time averages equal statistical averages is called an ergodic process.

In order to make the computation of the estimates possible, we must base our estimate on a finite segment of the sample sequence $X(n)$. When we consider ergodic processes, it is possible to compute estimates of the various desired averages of the random variables $\{X_n\}$ from a finite segment of a single sample sequence. The estimate $\hat{\zeta}$ of the parameter $\zeta$ is thus a function of the random variables $X_n$, $0 \le n \le N-1$; i.e.,

$$\hat{\zeta} = F[X_0, X_1, \ldots, X_{N-1}]$$

and therefore $\hat{\zeta}$ is also a random variable.

The bias of an estimator is defined as the true value of the parameter minus the expected value of the estimate, i.e.,

$$bias \quad = \zeta - E[\hat{\zeta}] \equiv B$$

An unbiased estimator is the one for which the bias is 0. This then means that the expected value of the estimate is the true value. The variance of the estimator is defined by

$$var[\zeta] = E\left[(\zeta - E[\hat{\zeta}])^2\right] = \sigma_\zeta^2$$

An estimator is said to be consistent if as the number of observations becomes larger, the bias and the variance both tend to zero.

## 2.4 Principles Of Maximum Entropy

In the literature the explanation of the method of Maximum Entropy in spectral estimation begins usually with the assumption that it is desirable to maximize the logarithmic integral $H$ of the unknown spectrum $S(\omega)$ of a process $S(n)$ and it leads to the conclusion that $S(\omega)$ must be an all pole rational function.

If $S$ is a probability space, the axiomatic definition of entropy is in fact a number assigned to each partition of $S$, i.e., the entropy $H(A)$ of a partition $A$ is the sum [10]

$$H(A) = -\sum_{i=1}^{N} P_i \ln P_i \qquad \text{where } P_i = P(A_i)$$

Since $P_1 + \ldots + P_N = 1$ it can be shown that $H(A)$ is maximum if $P_i = 1/N$ for $i = 1, \ldots, N$ and this maximum is equal to $\ln N$, i.e.

$$0 \le H(A) \le \ln N$$

Suppose that the random variable $X$ is of discrete type which can take the values $x_i$ with probabilities $P_i$. Then the entropy $H(x)$ of random variable $X$ is by definition the entropy of the partition formed by events $\{X = x_i\}$

$$H(X) = -\sum_i P_i \ln P_i \qquad P_i = P\{X = x_i\}$$

Correspondingly the entropy of a continuous variable $X$ is defined as

$$H(X) = -\int_{-\infty}^{\infty} f(x) \cdot \ln f(x)\, dx$$

where $f(x)$ is the probability density function of $X$. As it is obvious from the above integral, the entropy $H(x)$ is in fact the expected value of random variable $\ln f(x)$ or

$$H(X) = -E\{\ln f(x)\}$$

The joint entropy of several random variables is defined similarly

$$H(x_1, x_2, \ldots, x_r) = -E\{\ln f(x_1, x_2, \ldots, x_r)\} \qquad \text{(EQ 2)}$$

If $A$ is non-singular r by r matrix and we form the random variable $Y$ as following

$$Y = AX \qquad \text{where: } X = (X_1, \ldots, X_r) \quad \& \quad Y = (Y_1, \ldots, Y_r)$$

Then the joint density of these random variables is given by

$$f(Y_1, \ldots, Y_r) = \frac{1}{|A|} \cdot f(X_1, \ldots, X_r)$$

Applying (EQ 2), we obtain

$$H(Y_1, \ldots, Y_r) = H(X_1, \ldots, X_r) + \ln|A| \qquad \text{(EQ 3)}$$

In the application of the concept of entropy to the problem in spectral estimation, the quantity of interest is the entropy rate $H_X$ of a stationary process $X(n)$. This quantity is defined as

$$H_X = \lim_{r \to \infty} \left( \frac{1}{r+1} \cdot H(x_0, x_1, \ldots, x_r) \right) \qquad \text{(EQ 4)}$$

where $H(x_0, x_1, \ldots, x_r)$ is the joint entropy of the random variables $X(n), X(n-1), \ldots, X(n-r)$.

Suppose that $X(n)$ is the input to the stable causal system $H(z)$. Furthermore, suppose that $H(z)$ is minimum phase, i.e., $H(z)$ and its inverse are analytic for $|z| > 1$. If $X(n)$ is applied at $n = 0$, then the resulting response is

$$\bar{Y}(n) = \sum_{k=0}^{n} X(n-k) \cdot h(k) \qquad n = 0, 1, \ldots \qquad \text{(EQ 5)}$$

Note that $\bar{Y}(n)$ is not stationary. However, it tends to the stationary process $Y(n)$ as $n \to \infty$. (EQ 5) is a linear transformation of the random variable $X(0), X(1), \ldots, X(n)$ into the random variable $Y(0), Y(1), \ldots, Y(n)$ obtained with the transformation matrix

$$A = \begin{bmatrix} h(0) & 0 & \ldots & 0 \\ h(1) & h(0) & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ h(n) & h(n-1) & \ldots & h(0) \end{bmatrix} \qquad |A| = h^{(n+1)}(0)$$

Considering (EQ 3), this results to

$$H\left(\bar{Y}_0, \ldots, \bar{Y}_n\right) = H(X_0, \ldots, X_n) + (n+1) \cdot \ln h(0)$$

And from (EQ 4), we obtain

$$H_Y = H_X + \ln h(0) \qquad \text{(EQ 6)}$$

It is possible to write the term $\ln h(0)$ in (EQ 6) in term of $H\left(e^{j\omega T}\right)$, in fact since

$$\left|H\left(e^{j\omega T}\right)\right|^2 = H\left(e^{j\omega T}\right) \cdot H\left(e^{-j\omega T}\right)$$

it follows with $e^{j\omega T} = z$ and $jTz \cdot d\omega = dz$ that

$$jT\int_{-\omega_0}^{\omega_0} \ln\left|H\left(e^{j\omega T}\right)\right|^2 d\omega = \oint \frac{1}{z}\ln\left(H(z)\cdot H(1/z)\right) dz \qquad \omega_0 = \frac{\pi}{T}$$

where the integral is along the unit circle. But

$$\oint \frac{1}{z}\ln H(z)\, dz = \oint \frac{1}{z}\ln\left(H(1/z)\right) dz$$

therefore

$$\oint \frac{1}{z}\ln H(z)\, dz = \frac{jT}{2}\cdot\int_{-\omega_0}^{\omega_0} \ln\left|H\left(e^{j\omega T}\right)\right|^2 d\omega \qquad \text{(EQ 7)}$$

The function $\ln H(z)$ is analytic for $|z|>1$ by assumption, therefore, the circle of integration can be made arbitrarily large. Based on initial value theorem

$$h(0) = \lim_{z\to\infty} H(z)$$

and it can be concluded that

$$\oint \frac{1}{z}\ln\left(H(z)\right) dz = \ln h(0)\cdot\oint \frac{dz}{z} = 2\pi j\cdot \ln h(0)$$

in which we used the result of Cauchy Residue theorem

$$\frac{1}{2\pi j}\oint_C \frac{f(z)}{z-z_0}dz = f(z_0) \qquad \text{if } z_0 \text{ is inside circle } C$$
$$= 0 \qquad \text{if } z_0 \text{ is outside } C$$

where $C$ is a closed path in the z-plane and $f(z)$ is a function of complex variable $z$. Considering (EQ 7) we obtain

$$\ln h(0) = \frac{1}{4\omega_0}\int_{-\omega_0}^{\omega_0} \ln\left|H\left(e^{j\omega T}\right)\right|^2 \cdot d\omega \qquad \omega_0 = \frac{\pi}{T} \qquad \text{(EQ 8)}$$

Using (EQ 6) and (EQ 8), the entropy rate of the output of the system can be expressed in the final form as following:

$$H_y = H_x + \frac{1}{4\omega_0}\int_{-\omega_0}^{\omega_0} \ln\left|H\left(e^{j\omega T}\right)\right|^2 d\omega \qquad \omega_0 = \frac{\pi}{T} \tag{EQ 9}$$

### 2.4.1 Entropy Of A Normal Process

If the probability density function of a random variable $x$ is defined as

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/2\sigma^2}$$

then $x$ is called a normal process and we have

$$H(x) = -E\{\ln f(x)\} = E\{x^2/2\sigma^2\} + \ln\sigma\sqrt{2\pi} = \ln\sigma\sqrt{2\pi e}$$

Suppose that $V(n)$ is stationary white noise with average power $E\{V^2(n)\} = \sigma^2$. In this case, the random variables $V(n), V(n-1), \ldots, V(n-r)$ are normal independent with variance $\sigma^2$. Hence, their joint density can be expressed as:

$$f(V_0, V_1, \ldots, V_r) = f(v_0) \ldots f(v_r)$$

and form (EQ 2).

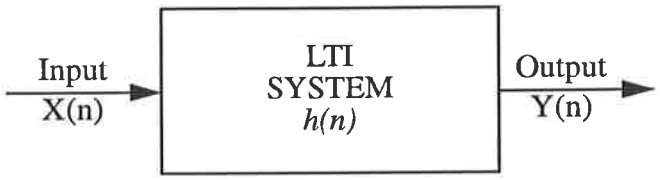
$$H(V_0, \ldots, V_r) = -E\{\ln [f(V_0) \ldots f(V_r)]\} = (r+1)\ln\sigma\sqrt{2\pi e}$$

dividing by $r+1$, it can be seen that the entropy rate of a normal white-noise process $V(n)$ is given by:

$$H_V = \ln\sigma\sqrt{2\pi e}$$

### 2.4.2 Input Output Correlation Functions Of Linear Digital Filters

Suppose that a signal $X(n)$ with known auto-correlation $R_{xx}(m)$ is applied to a linear time invariant system (LTI) with impulse response $h(n)$, producing the output signal $Y(n)$.




$$Y(n) = h(n) \otimes X(n) = \sum_{k=-\infty}^{\infty} h(k) \cdot X(n-k)$$

The auto-correlation of the output signal $Y(n)$, can be obtained by using the above equation and the properties of convolution. Thus we have

$$\begin{aligned} R_{yy}(m) &= Y(m) \otimes Y(-m) \\ &= [h(m) \otimes X(m)] \otimes [h(-m) \otimes X(-m)] \\ &= [h(m) \otimes h(-m)] \otimes [X(m) \otimes X(-m)] \\ &= R_{hh}(m) \otimes R_{xx}(m) \end{aligned} \qquad \text{(EQ 10)}$$

where $R_{xx}(m)$ is the auto-correlation sequence of the input signal $\{X(n)\}$, $R_{yy}(m)$ is the auto-correlation sequence of the output $\{y(n)\}$, $R_{hh}(m)$ is the auto-correlation sequence of the impulse response $\{h(n)\}$. Since (EQ 10) involves the convolution operation, the z-transform of this equation yields

$$\begin{aligned} S_{yy}(z) &= S_{hh}(z) \cdot S_{xx}(z) \\ &= H(z) \cdot H\left(z^{-1}\right) \cdot S_{xx}(z) \end{aligned}$$

Evaluating $S_{yy}(z)$ on the unit circle, we can obtain the energy density spectrum of the output signal as

$$S_{yy}(\omega) = |H(\omega)|^2 \cdot S_{xx}(\omega) \qquad \text{(EQ 11)}$$

### 2.4.3 Entropy Rate And Power Spectrum:

For an arbitrary normal process $N(n)$ we have the auto-correlation function defined as

$$R_{nn}(m) = E\{N(n+m) \cdot N(n)\}$$

and the power spectrum is

$$S_{NN}(\omega) = \sum_{m=-\infty}^{\infty} R_{nn}(m) \cdot e^{-j\omega m}$$

Since $N(n)$ is a normal process, all its statistics including $H_N$ can be expressed in terms of its auto-correlation $R_{nn}(m)$. Hence, if another process $Y(n)$ has the same auto-correlation function, or equivalently, the same power spectrum as $N(z)$, then its entropy rate $H_Y$ will equal $H_N$. If $H(z)$ is a minimum phase function which specifies a system with an impulse response equal to $N(n)$ then $S_{NN}(z)$ can be written as a product

$$S_{NN}(z) = H(z) \cdot H(1/z) \qquad S_{NN}(\omega) = \left|H\left(e^{j\omega}\right)\right|^2$$

Using as input to the system $H(z)$ a white-noise process $V(n)$ with $E\{V^2(n)\} = 1$, we obtain as output a process $Y(n)$ with a spectrum

$$S_{YY}(\omega) = |H(\omega)|^2 \cdot S_{VV}(\omega) = S_{NN}(\omega)$$

Using (EQ 9), the entropy rate of the normal process $N(n)$ can be expressed as

$$H_N = H_Y = H_V + \frac{1}{4\omega_0}\int_{-\omega_0}^{\omega_0} \ln\left|H\left(e^{j\omega T}\right)\right|^2 d\omega$$

$$= \ln\sqrt{2\pi e} + \frac{1}{4\omega_0}\int_{-\omega_0}^{\omega_0} \ln S_{NN}(\omega)\, d\omega \qquad \text{(EQ 12)}$$

## 2.5 Maximum Entropy And Spectral Estimation

Having defined the entropy rate of a random process $S(n)$ in terms of its power spectrum [See (EQ 12)], the estimation of the spectrum of $S(n)$ with the method of maximum entropy involves the maximization of its entropy rate $H_S$ subject to various constraints which are usually expressed as

$$E\{g_k(x)\} = n_k \qquad k = 1, \ldots, m$$

where the functions $g_k(x)$ and the numbers $n_k$ are given.

The solution of the problem is based on the following inequality

$$-\int_{-\infty}^{\infty} f(x) \cdot \ln f(x)\, dx \le -\int_{-\infty}^{\infty} f(x) \cdot \ln p(x)\, dx$$

with equality iff $f(x) = p(x)$, where $f(x)$ and $p(x)$ are two density functions. The proof of the above comes from the inequality $\ln y \le y - 1$. It can be shown that the optimum density is given by

$$p(x) = \frac{1}{z} \cdot e^{-\lambda_1 g_1(x) - \ldots - \lambda_m g_m(x)}$$

where

$$z = \int_{-\infty}^{\infty} e^{-\lambda_1 g(x) - \ldots - \lambda_m g_m(x)}\, dx$$

and the constants $\lambda_k$ are such that

$$\int_{-\infty}^{\infty} g_k(x) \cdot p(x)\, dx = n_k \qquad k = 1, \ldots, m$$

In practice we are given the $N+1$ values $R(0), R(1), \ldots, R(n)$ (data) of the auto-correlation $R(m)$ of a random process $S(n)$ and we wish to estimate its power spectrum $\bar{s}(\omega)$. The statistics of $S(n)$ are determined in terms of the joint density of random variables $S(n), S(n-1), \ldots, S(n-r)$. Hence, to apply the principle of maximum entropy, we must determine the unknown values of $R(m)$ so as to maximize the entropy $H(S_0, S_1, \ldots, S_r)$ of these random variables and to find the limit as $r \to \infty$. This is equivalent to the maximization of the entropy rate $H_S$ of $S(n)$ subject to the given constraints.

If we denote by $\hat{S}(n)$ the optimum linear predictor of $S(n)$ in terms of its $N$ most recent values [24]

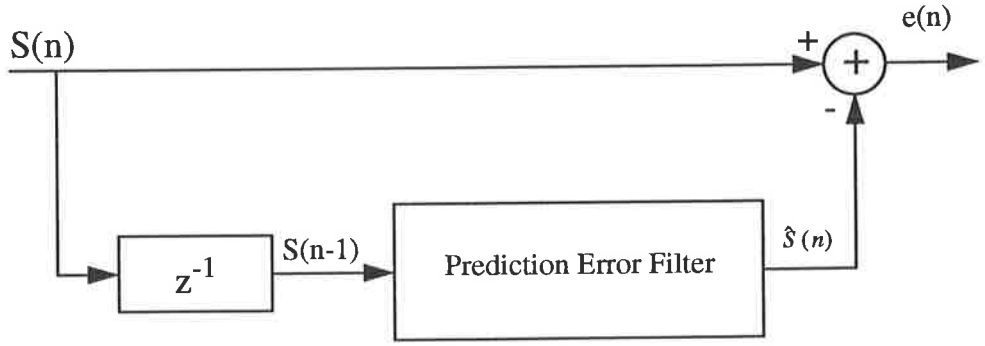
$$\hat{S}(n) = \sum_{k-1}^{N} a_k \cdot S(n-k)$$

then the estimation error

$$e(n) = S(n) - \hat{S}(n)$$

is the output of linear system with input $S(n)$ and system function

$$H(z) = 1 + \sum_{k=1}^{N} a_k \cdot z^{-k}$$



and its average power $P$ is given by

$$E\left[e^2(n)\right] = P = R(0) + \sum_{k=1}^{N} a_k \cdot R(k) \qquad \text{(EQ 13)}$$

From (EQ 9) it follows that

$$H_e = H_S + \frac{1}{2\omega_0}\int_{-\omega_0}^{\omega_0} \ln\left|1 - \sum_{k=1}^{N} a_k \cdot e^{-j\omega kT}\right| d\omega$$

The above integral depends only on the data, hence to maximize $H_S$, it suffices to maximize the entropy rate $H_e$ of the error $e_n$. If $X(n)$ is a stationary process with specified average power $E\{X^2(n)\} = \sigma^2$, then its entropy rate $H_X$ is maximum iff $X(n)$ is normal white noise. By the same argument $e(n)$ is normal white noise with power spectrum

$$S_{ee}(z) = P$$

It follows from the above that the process $S(n)$ satisfies recursive equation

$$S(n) - \sum_{k=1}^{N} a_k \cdot S(n-k) = e(n) \qquad \text{(EQ 14)}$$

where $e(n)$ is white noise. Therefore, $S(n)$ is the output of the filter $1/(H(z))$ with input $e(n)$, hence, its power spectrum equals

$$S_{ss}(z) = \frac{S_{ee}(z)}{H(z) \cdot H(1/z)}$$

substituting $z = e^{j\omega T}$ results

$$\bar{S}(\omega) = \frac{p}{\left|1 - \sum_{k=1}^{N} a_k \cdot e^{-j\omega kT}\right|^2}$$

Therefore, it is concluded that $S(n)$ is an auto-regressive (AR) process and since $e(n)$ is normal, $S(n)$ is also normal.

### 2.5.1 Filter Parameters And Auto-correlation Sequence

When the power spectral density of the random stationary process is rational function, there is a basic relationship which exists between the auto-correlation sequence $R(m)$ and the parameters of the linear filter $1/(H(z))$ that generates the process by filtering the white noise sequence. This relationship may be obtained by multiplying the difference equation in (EQ 14) by $S^*(n-m)$ and taking the expected value of both sides of the resulting equation. Thus we have

$$E[S(n)S^*(n-m)] = \sum_{k=1}^{N} a_k E[S(n-k)S^*(n-m)] + E[e(n)S^*(n-m)]$$

Hence

$$R(r) = \sum_{k=1}^{N} a_k \cdot R(r-k) + R_{es}(r)$$

where $R_{es}(r)$ is the cross correlation sequence between $e(n)$ and $S(n)$.

The cross correlation $R_{es}(r)$ is related to the filter impulse response. That is,

$$\begin{aligned} R_{es}(r) &= E[S^*(n)e(n+m)] \\ &= E\left[\sum_{k=0}^{\infty} h(k)e^*(n-k)e(n+m)\right] \\ &= P \cdot h(-m) \end{aligned}$$

where, in the last step, we have used the fact that the sequence $e(n)$ is white. Hence

$$R_{es}(r) = 0 \text{ for } r>0$$

and form (EQ 13)

$$R(r) = \sum_{k=1}^{N} a_k \cdot R(r-k) \qquad r = 1, \ldots, N$$

$$R(0) = P - \sum_{k=1}^{N} a_k \cdot R(k)$$

Therefore, there is a linear relationship between $R(m)$ and the $\{a_k\}$ parameters. These equations, called the Yule-Walker equations [24], may be expressed in the following matrix form

$$\begin{bmatrix} R(0) & R(-1) & \ldots & R(-N) \\ R(1) & R(0) & \ldots & R(-N+1) \\ \ldots & \ldots & \ldots & \ldots \\ R(N) & R(N-1) & \ldots & R(0) \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ \ldots \\ a_N \end{bmatrix} = \begin{bmatrix} P \\ 0 \\ \ldots \\ 0 \end{bmatrix}$$

This correlation matrix is Toeplitz and can be efficiently inverted by use of Levinson_Durbin algorithm. The key to the Levinson_Durbin method of solution that exploits the Toeplitz property of the matrix is to proceed recursively, beginning with a predictor of order $N = 1$ and to increase the order recursively, using the lower order solution to obtain the solution to the next higher order. An objection that can be raised at this approach becomes evident when we remember the maximum entropy interpretation of an AR representation. Accordingly, the AR coefficients should be estimated in such a way that we do not use any information which is unavailable to us. The estimation of the auto-correlation function coefficients using Yule_Walker estimation, on the other hand, assumes that $S(n) = 0$ for $|n| > N$, an assumption that contradicts the principle of maximum entropy.

Burg suggested a method of estimating the AR parameters or equivalently the prediction error filter coefficients that does not require prior estimates of the auto-correlation function. The Burg method uses a recursion that is very similar to that developed for the Yule_walker estimation. We obtain the desired recursion for the predictor coefficients in the Burg algorithm as [1]

$$a_m(k) = a_{m-1}(k) - a_m(m) \cdot a_{m-1}(m-k) \qquad k = 1, \ldots, m-1$$

where $a_m(k)$ is the kth coefficients of the predictor of order m and $a_m(m)$ is equal to

$$a_m(m) = \frac{-2 \sum_{n=m+1}^{N} b_{m-1}(n) \cdot b'_{m-1}(n)}{\sum_{n=m+1}^{N} \left[ b^2_{m-1}(n) + b'^2_{m-1}(n-1) \right]} \qquad \text{(EQ 15)}$$

The recursion is initiated by

$$b_0(n) = b'(n) = S(n)$$

Therefore, the value for $a_1(1)$ equals

$$a_1(1) = \frac{-2 \sum_{n-2}^{N} S(n) \cdot S(n-1)}{\sum_{n=2}^{N} \left[ S^2(n) + S^2(n-1) \right]}$$

The update equations for the recursion are

$$b_m(n) = b_{m-1}(n) + a_m(m) \cdot b'_{m-1}(n-1) \qquad \text{(EQ 16)}$$

$$b'_m(n) = b'_{m-1}(n-1) + a_m(m) \cdot b_{m-1}(n) \qquad \text{(EQ 17)}$$

The recursive formula for $P_m$ is as follow

$$P_m = P_{m-1}\left(1 - a_m^2(m)\right)$$

where $P_m$ is the average power of $e(n)$ resulted from a filter of order $m$. For $m = 0$, $P_0$ is estimated by

$$P_0 = \frac{\sum_{n=1}^{N} |S(n)|^2}{N}$$

It follows from the equation for computing $a_m(m)$ that $|a_m(m)| \leq 1$ so that $0 \leq P_m < P_{m-1}$. The recursive procedure is summarised in the flow diagram Fig. 2.1. It is also possible by use of

Eqs. (11), (12) and (13) to calculate all the quantities $a_m(m)$ and $P_m$ before evaluation of the remaining filter coefficients.

### 2.5.2 Selection Of Model Order

One of the most important aspects of the use of the AR model is the selection of the order $M$ [9]. As a general rule, a model with a too low order gives a highly smoothed spectrum. On the other hand, if $M$ is too high, it may introduce spurious low-level peaks in the spectrum. One indication of the performance of the AR model is the mean square value of the residual error, which decreases as the order of the AR model is increased. The rate of the decrease can be monitored in order to terminate the process when the rate of decrease becomes relatively slow.

Various researchers have worked on this problem and many experimental results have been achieved among which the papers by Gersch and Sharpe [1973], Ulrych and Bishop [1975], Tong [1975, 1977], Jones [1976], Nuttal [1976], Berryman [1978], Kaveh and Bruzzone [1979] and Kashyap [1980] can be mentioned.

However, the two more known criteria for order selection have been proposed by Akaike [1969, 1974]. The first one is called the final prediction error (FPE) criterion, and is selected to minimize the performance index [2, 3]

$$FPE(m) = P_m\left(\frac{N+M+1}{N-M-1}\right)$$

where $P_m$, the estimated variance of the linear prediction error, is monotonically decreasing, while the term in the brackets is monotonically increasing. The second criterion, called Akaike information criterion (AIC), is based on the minimization of the following

$$AIC(m) = \ln p_m + \frac{2m}{N}$$

The experimental results indicate that the model order selection criteria do not yield definitive results. For example, Ulrych and Bishop [1975], Jones [1976], and Berryman [1978], found that the $FPE(m)$ criterion tends to underestimate the model order. Kashyap [1980] showed that the $AIC$ criterion is statistically inconsistent as $N \to \infty$. In general, experimental results indicate that for small data lengths, the order of the AR model should be in the range $N/3$ to $N/2$ for good results. It is obvious that in the absence of any prior information about the physical process based on which the data is resulted, one should try different model orders and, ultimately, interpret the different results.

## • Summary

In this chapter the necessary background for the statistical signal processing was presented. In the subsequent sections, the theoretical development of the maximum entropy technique and its application within the context of spectral analysis were discussed. Finally the resultant AR model for the signal under analysis, and the Burg method for the estimation of the filter parameters were introduced, which originated the recursive algorithm of the flow diagram of Fig. 2.1. The VLSI implementation of the algorithm is presented in the next chapter along with the basic feature of the chip developed for maximum entropy spectral estimation.

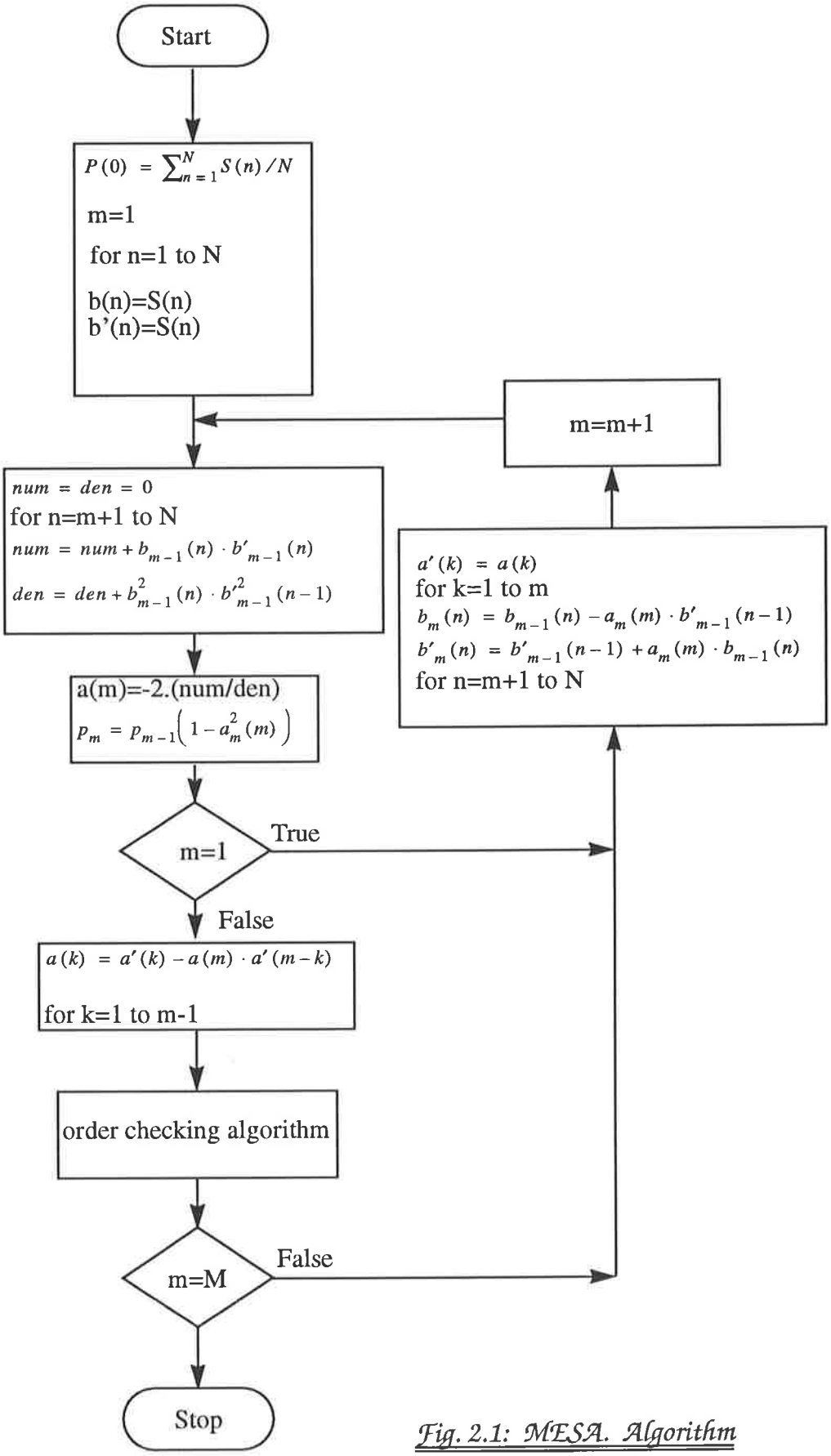



*Fig. 2.1: MESA. Algorithm*

# Chapter 3

# ARCHITECTURAL MAPPING AND ALGORITHM TRANSFORMATION

The single most significant development in digital system design in recent years has been the advent of the microprocessor, a central processing unit integrated on a single chip of silicon. The processing power and economics of the microprocessor have had a tremendous impact on the way digital systems are designed and on their scope of application. Because of some nonideal features of general purpose microprocessors, they can not be applied to DSP problems. Among their limitations, the relatively slow speed of numerical multiplications, and limitations on data input and output can be mentioned. A DSP chip, on the other hand, does many or most of the same things that a general-purpose microprocessor chip does. Specifically, for DSP work, the chip should be exceptionally good at the things that are necessary in DSP networks, e.g., multiplications, summations, and data moves.

Digital signal processing is computationally intensive; it requires many multiplications and additions. A range of approaches are commonly used to implement DSP algorithms. They present various degrees of hardware-software optimization. The single-chip digital signal processor combines software flexibility with DSP hardware power. It removes the bottleneck constraint of conventional microprocessor architecture by high DSP throughput at moderate cost.

### 3.1 DSP Chip Families

A number of methods are used in the design of the DSP processor systems [12]: Bit-Slice, Word-Slice, and microprocessor-plus-memory systems.

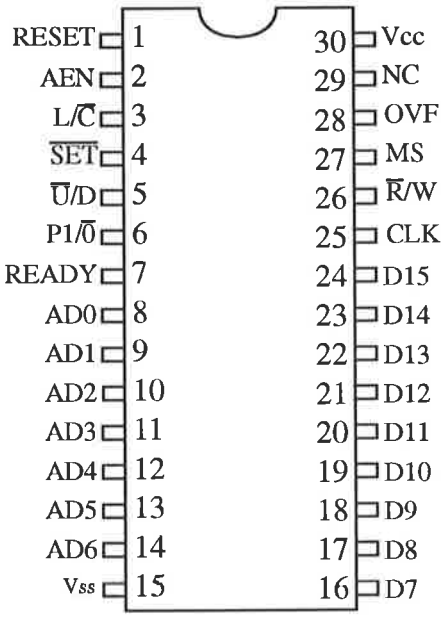
Bit-Slice systems are comprised of small but fast subunits arranged in parallel to build the required word-length. Bit-Slice offers flexibility and high performance, at the cost of high count of components, large power consumption, and complexity of hardware development.

Word-Slice systems, on the other hand, have larger subunits than Bit-Slice ones, and are a natural replacement for Bit-Slice, with comparable performance and far fewer components. The deign of DSP microprocessors and microcomputers follows the Harvard architecture [13] (a historical alternative to the Von Neumann architecture, also defined in the 1940's) with separate data and instruction buses, and that is basically the employed architecture in the present study.

## 3.2 Basic Features Of The Chip

The processor is a single 16-bit chip, implemented with approximately 57000 transistors on about 21.3 $mm^2$ area of chip which could be contained in a 30-pin dual-in-line package. The possible pin layout of the processor is shown in Fig. 3.1. The employed technology in the

design of the processor is CMOS 1.2 micron, and its instruction set consists of 69 instructions. Its architecture allows two levels of pipelining, i.e., while an instruction is being executed, the next instruction is being consequently fetched, decoded, and executed. Many instructions can be executed in parallel, such as load with address generation, multiply with add, and so on.




| Name | Function | Type |
|---|---|---|
| AD6-AD0 | Address/Data Bus | Bidirectional |
| D15-D7 | Data Bus | Bidirectional |
| READY | Job Concluded | Output |
| P1/$\overline{0}$ | Memory Page | Input |
| $\overline{U}$/D | Up/Down Count | Input |
| $\overline{SET}$ | Set Counter | Input |
| L/$\overline{C}$ | Load/Count | Input |
| AEN | Address Enable | Input |
| RESET | System Reset | Input |
| $\overline{R}$/W | Read/Write Control | Input |
| MS | Memory Enable | Input |
| OVF | Overflow Detector | Output |
| CLK | System Clock | Input |
| NC | Not Connected | ..... |
| Vcc | +5v | Input |
| Vss | Ground | Input |

**Figure 3.1--Processor pin configuration**

The processor operates on a single 5v power supply connected at $V_{cc}$; power supply ground is connected to $V_{ss}$. Based on the results of simulation, the frequency of the internal clock, which synchronizes the operation of the processor, can be up to 30 MHz. As it is shown in Fig. 1, some of the terminals have multiple functions. For example in the pin layout we see that the address lines $A_0$ through $A_6$ and data lines $D_0$ through $D_6$ are time multiplexed. In fact these lines serve both as terminals through which the address register of the internal memory (RAM) can be loaded and also as data lines. For this reason, these pins are labelled $AD_0$ to $AD_6$. Address information is provided on the address/data lines at the beginning of each memory reference, and is latched and held during the remainder of the memory

reference to provide address bits $A_0$ to $A_6$. A 7-bit register/counter latches the address information from the address/data pins when clocked by the address register enable signal, AEN. The address register can also work as counter, thus the user has the ability to point to any location of the memory sequentially, by using the address register as a counter or randomly, by loading the address register with the desired address.The remaining package pins provide data, and control signals. They can be described as following (See also Fig. 1)

4. AEN - This signal enables the address register when it is logic 1, otherwise the address register is disabled.

5. L/$\overline{C}$ - This is the load/count control of the address register. When L/$\overline{C}$ is a logic one, the address register can be loaded by data on $D_6$-$D_0$. When L/$\overline{C}$ is a logic zero, the register acts as a counter and the RAM can be addressed sequentially.

6. $\overline{SET}$ - A logic zero on this signal sets the address register to the binary value 1111111 provided that L/$\overline{C}$ is one. $\overline{SET}$ must be logic one, if the address register is to be loaded by a specific data.

7. $\overline{U}$/D - When the address register is in counting mode, a logic zero specifies up-counting while a logic one specifies down-counting.

8. RESET - A logic one on this signal resets the program counter to zero, and this will cause the command in location zero of the ROM to be read into the control unit, which is no operation command (NOP). This command disables all the registers in the chip, thus the processor is prevented to write on the data bus and the internal control lines of the RAM and address register are tri-stated, i.e., the address register and the RAM are under the user control. Note that the content of the program counter (PC) remains zero as long as the RESET is logic one. When RESET is logic zero, the chip is in operating mode, the external data bus pins $D_{15}$-$D_7$ and $AD_6$-$AD_0$ of the chip and external control lines of the RAM

and address register are tri-stated. Therefore the RAM and the address register are under the processor control.

9. READY - when the processor is executing the HALT command, this signal is logic one which shows that the processor is finished with the processing of the data and the results are ready. At this time the address register is pointing to the location of the first predictor's coefficient which is always a one. The content of the program counter (PC) remains unchanged until the processor will be reset.

10. OVF - A logic one on this signal shows that an overflow has occurred. It remains one until the chip will be reset.

## 3.3 Memory Organization

The processor has 256 words (16-bit) of on chip data RAM and 512 words (9-bit) of ROM for microprogram codes. Separate program and data buses enable the processor to perform concurrent data read and write, and program fetches operations. With a clock signal of 20 MHz, the processor will have an ideal instruction cycle time of 50 ns, with most instructions requiring only a single cycle. Thus, it is capable to execute up to 20 million instruction per second. The data RAM can be addressed through eight address lines, seven of which are in fact the outputs of the address register, and the MSB of the address is connected to pin p1/$\overline{0}$ of the chip. This way, in fact the memory is divided into two pages, page zero and page one. The user is only allowed to use memory locations one through 120 (DEC) of page zero to store the original data set. The rest of the memory will be used by the C.P.U. during the processing of the data.

The data RAM can be enabled for read or write operation by using the following control signals

1. MS - Memory select, enables the RAM of the chip for active operation. When MS is logic zero, the data outputs are tri-stated and the RAM is disabled. The data from the last read operation are held in the output latches. When MS is a logic 1, the RAM is allowed to perform read and write operations.

2. $\overline{R}$/W - The $\overline{R}$/W signal specifies whether a read or write operation is to be performed. A logic one on this signal specifies a write operation and places the output latches in a high impedance state, while a logic zero specifies a read operation and enables the output latches. Note that read and write operations take place from and on the same data lines, i.e., the RAM has a common data inputs and outputs.

3. P1/$\overline{0}$ - A logic zero on this signal activates page zero of the RAM for write or read operation, while a logic one is equivalent to pointing to page one of the RAM.

## 3.4 Architecture And Operation of The Processor

A block diagram of the internal architecture of the processor is shown in Fig. 3.2. The processor contains registers with both general and dedicated purpose.

1. A 9-bit program counter (PC).

2. A 9-bit stack register (ST).

3. Eight 16-bit dedicated purpose registers: Z, W, X, Y, MU1, MU2, CO1, CO2.

4. A 63-bit register to store the final partial products in the pipelined Multiplier: MU3.

5. Two 7-bit address registers/counters: A1, A2.

6. A 16-bit temporary register: TEMP.

The 9-bit program counter fetches instructions from any one of 512 possible memory locations. When the Reset pin of the processor is made logic one, the program counter is reset to

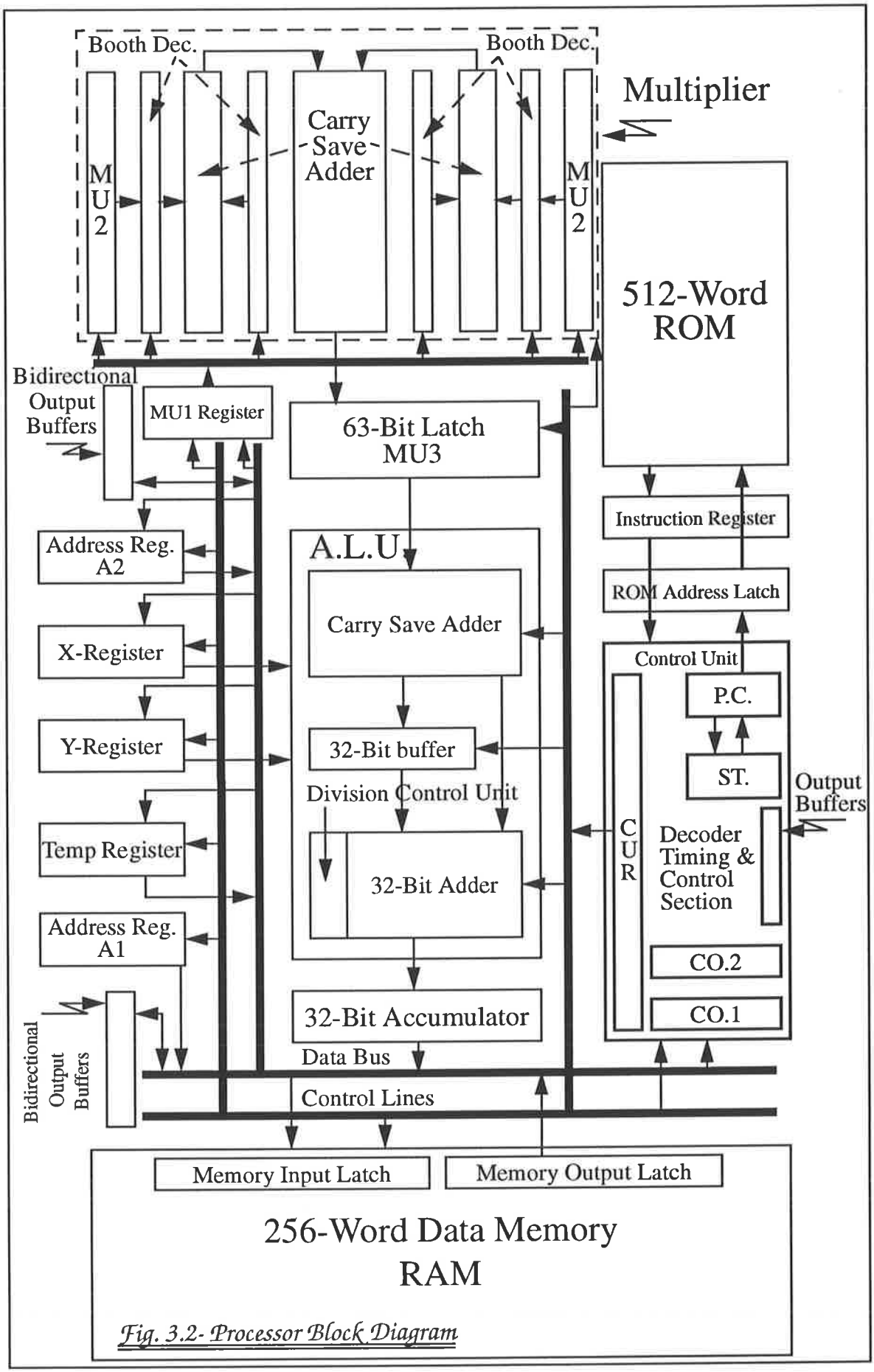



*Fig. 3.2- Processor Block Diagram*

zero; the control unit transfers the contents of the PC to the address latch, providing the address of the first instruction to be executed. Thus, program execution in the processor begins with the instruction in memory location zero, which is a no operation instruction and disables all the registers in the processor unit.

The processor's control unit controls and synchronizes all data transfers and transformations and is the key sequential subsystem in the processor [4, 35]. All the actions attributable to the processor are actions implemented by the control unit. The basic operation of the processor is regulated by the control unit and consists of the sequential fetching and execution of instructions. Each instruction execution cycle has two primary states: the fetch state and the execute state. The fetch state transfers an instruction from the memory (ROM) into the control unit, and the execute state executes the instruction. Because of the pipelined nature of the processor, it does not cycle between fetching and execution of instructions. Therefore, the execution cycle of one instruction is in fact the fetching cycle of the next instruction. At the falling edge of each clock cycle, a new instruction is fetched from the ROM and is processed by the control unit. At the falling edge of the next clock cycle, the control unit updates the contents of the control-unit-register (C.U.R), and fetches the next instruction from the ROM. Therefore, while the control unit is processing a new instruction, the processing unit is executing the previous instruction which can be a data transfer operation, arithmetic operation, etc. As an example, the multiplication of two numbers is shown in Fig. 3.3.

Processor instructions are 1 or 2 bytes in length. The first byte always contains the operation code (OP code). During the instruction fetch, the first byte is transferred from the memory into the instruction register. The PC is automatically incremented so it contains the address of the next instruction if the instruction contains only 1 byte, or the address of the next byte of the present instruction if the instruction consists of 2 bytes. In the case of a mul-

tiple instruction, the timing and control section provides additional operation to read in the additional byte. The timing and control section uses the instruction register output and external control signals to generate the state signals. After all the bytes of an instruction have been fetched, the instruction is executed.

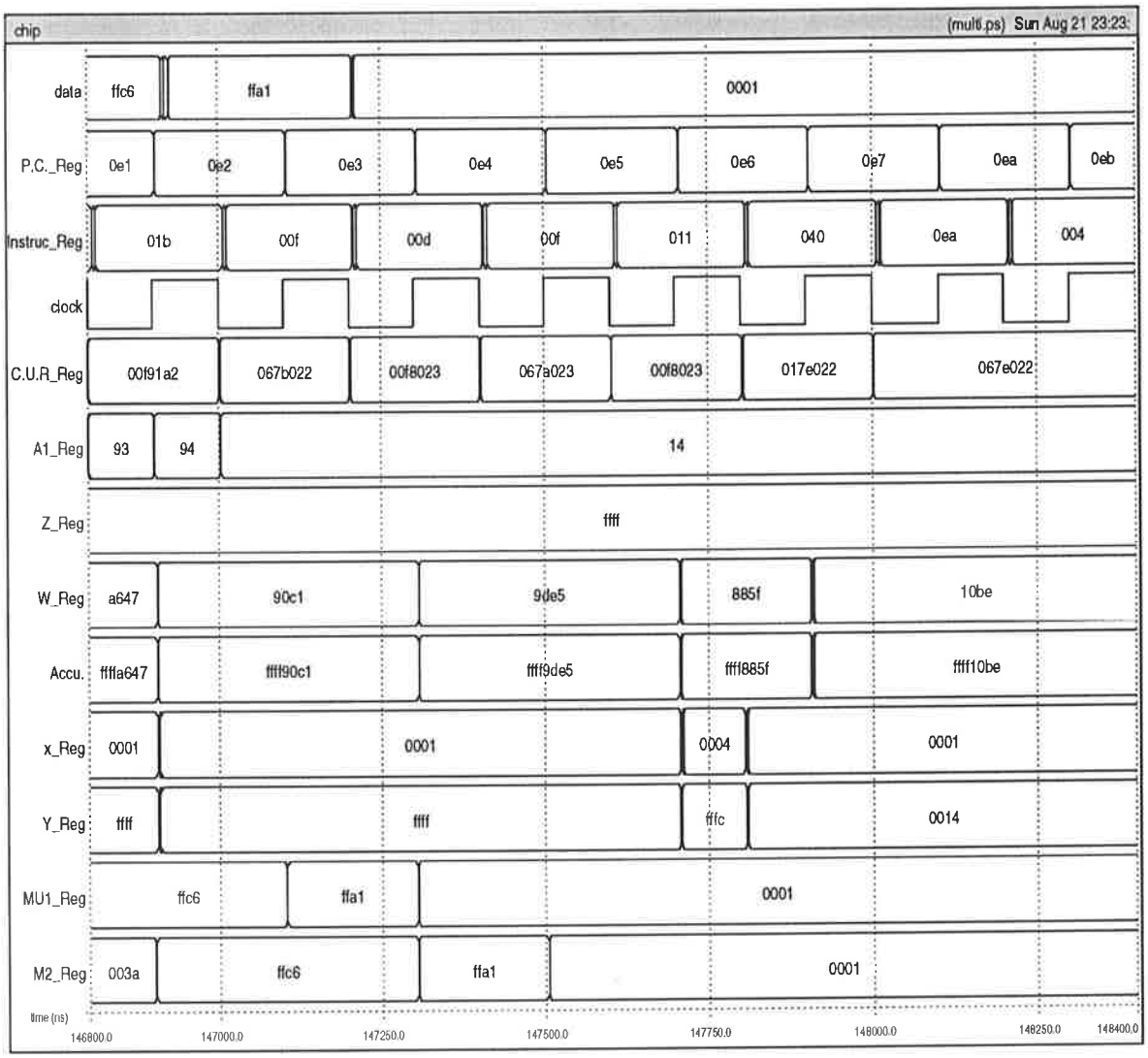



**Fig. 3.3--Multiplication of ffc6(HEX) by ffa1(HEX)**

**Timing description of the multiply operation depicted in Fig. 3:**

**1. At time 146900 ns, the first operand (ffc6) is moved to the register MU1.**

**2. At time 147100 ns, the second operand (ffa1) is moved to the register MU1.**

**3. At time 147300 ns, Mu1*Mu2 is moved to the MU3 and is added to the content of the accumulator, which is ffff9de5(HEX).**

**4. At time 147700 ns, the final result is stored in the accumulator.**

**Note that the data in the accumulator is the complement of the actual result.**

A subroutine call is mainly performed by using the stack register (ST). The 9-bit stack register, ST, maintains a pointer to the location of the ROM to which the control of the program has to return after the execution of the called subroutine is finished. In the current algorithm, it is mainly used for the division subroutine.

The processor's arithmetic unit performs arithmetic operations on data. Depend on which operation is to be performed, the operands for these operations are stored in four registers associated with the ALU: the 16-bit X register, the 16-bit Y register, the 16-bit MU1 register and the 16-bit MU2 register. The register pair ZW is used as the accumulator and serves as the destination register for all the arithmetic operations. The accumulator is loaded from the outputs of the ALU and can transfer data to the internal bus.

Associated with the ALU are four flag registers, which indicate the condition associated with the results of the arithmetic operations. The flags indicate zero, addition overflow, division overflow, and whether the result is more than 16-bit in length.

The processor's internal data bus is 16 bits wide and transfers data among various internal registers, and the data memory (RAM) or to external devices through the multiplexed data bus buffers. The bidirectional three-state data bus buffer isolates the processor internal data bus from the external system data bus. In the output mode, the information on the internal bus is transferred to the external bus through data bus output buffers. The output buffers are floated during input operations. During the input mode, data from the external data bus is transferred to the internal data bus.

The floor plan of the chip and the distribution of the active area within the chip are depicted in Fig. 3.4 and Fig. 3.5 respectively.

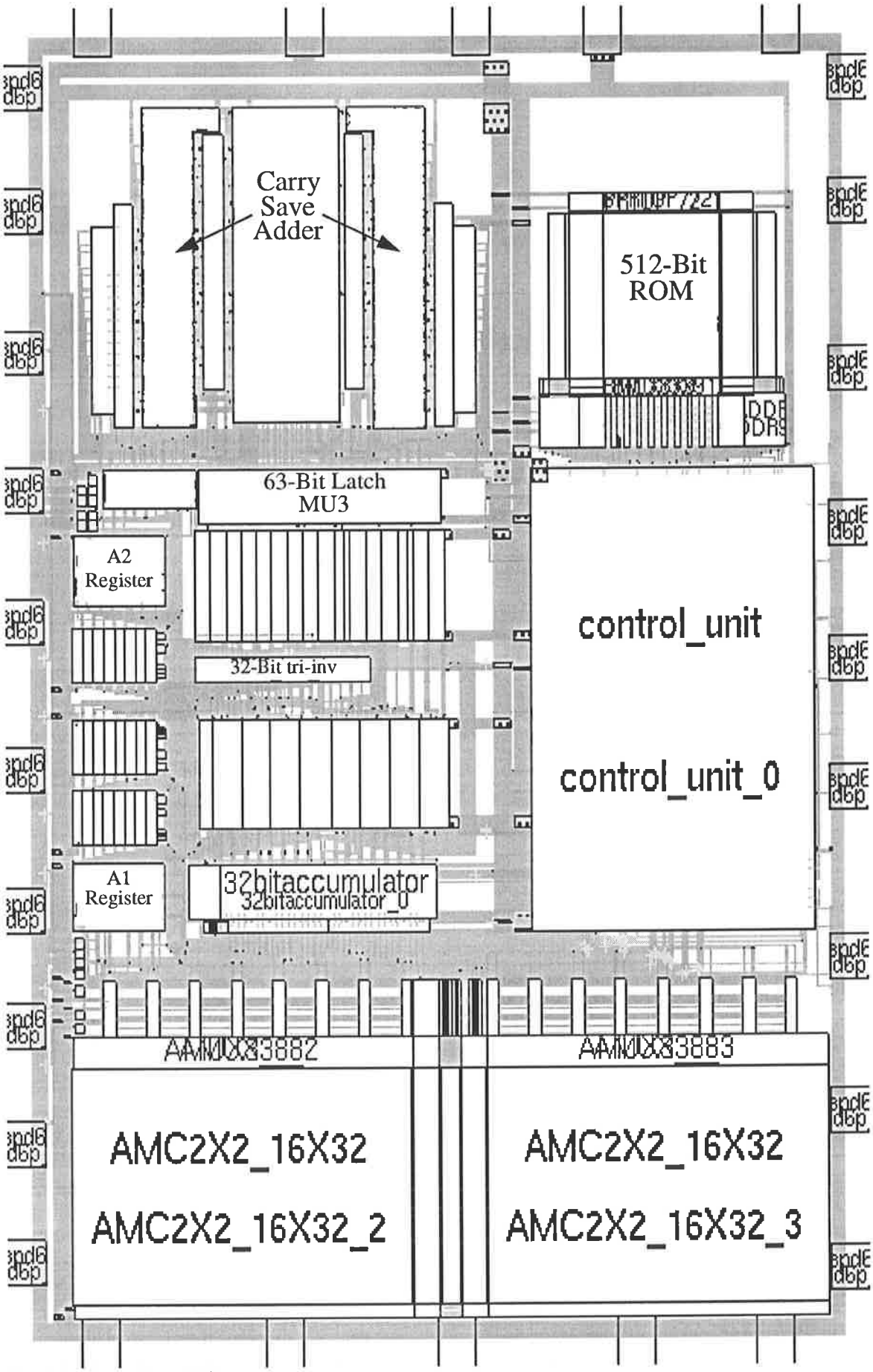



**Fig. 3.4-- Floor Plan of the processor**

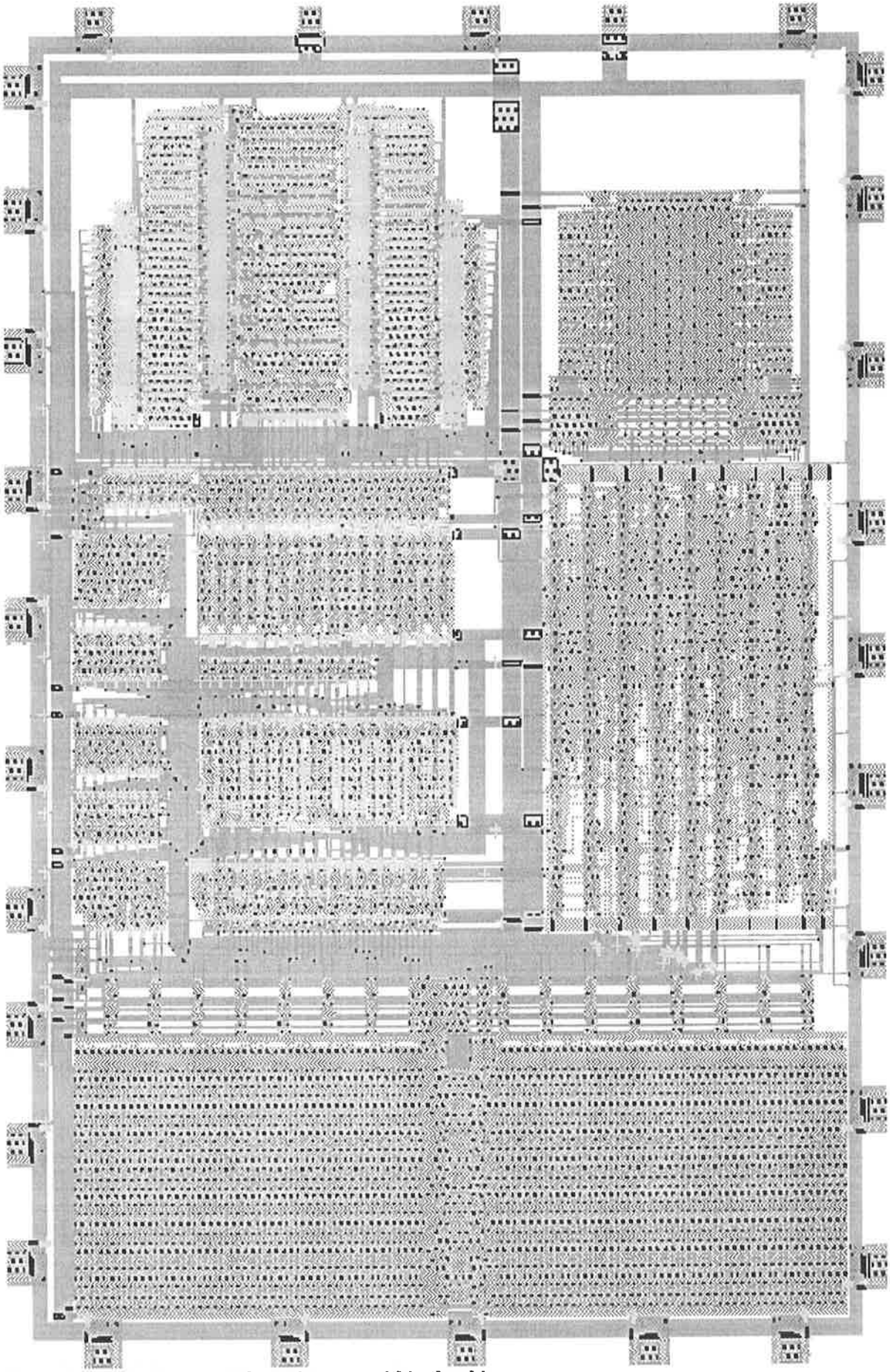


**Figure 3.5-- Distribution of the active area within the chip.**

## 3.5 Instruction Set

The function of the processor is implemented by a sequence of data transfers between registers in the main memory and the processor. Each register which can be manipulated under program control is addressable in some manner, allowing it to be designated for use in data transfer or transformation.

The kinds of individual transfers and transformations which are possible are specified by the processor's instruction set. Each instruction in the set causes one or more data transfers and/or transformations. The control section of the processor decodes the program instructions, and using the system clock, controls what register transfers or transformations take place and when. The instruction set of the processor is grouped in order under four different functional headings[29]

1. Data Transfer Group-- Moves data between registers or between memory locations and registers.
2. Arithmetic Group-- Adds, subtracts, increments, or decrements data in registers.
3. Branch Group-- Initiates conditional or unconditional jumps, subroutine calls and returns.
4. Machine Control Group-- Includes instructions for setting and clearing flags, setting registers, shifting accumulator and Halt.

A summary of the processor instruction set is given in tables one to four. It shows the machine code equivalents (Hex.) as well as assembly language mnemonics. A full description of the instruction set is given in appendix A.

**TABLE 1. Instruction Set--Data Transfer Group**

| Machine Code (HEX) | Mnemonic | Function |
|---|---|---|
| 12 | MV Y, Z | (Y) <- (Z) |
| 22 | MV X, Z | (X) <- (Z) |
| 0B | MV CO1, Z | (CO1) <- (Z) |
| 33 | MV CO2,Z | (CO2) <- (Z) |
| 2B | MV A1, Z | (A1) <- (Z) |
| 2A | MV A2, Z | (A2) <- (Z) |
| 23 | MV MU1, Z | (MU1)<-(Z) & (MU2)<-(MU1) & (MU3)<-(MU1)*(MU2) & ((Z)(W))<-0 |
| 0F | MV1 MU1, TEMP | (MU1)<-(TEMP) & (MU2)<-(MU1) & (MU3)<-(MU1)*(MU2) & Load ((Z)(W)) |
| 0D | MV2 MU1, TEMP | (MU1)<-(TEMP) & (MU2)<-(MU1) |
| 15 | MV X, W | (X) <- (W) |
| 20 | MV MU1, W | (MU1)<-(W) & (MU2)<-(MU1) & (MU3)<-(MU1)*(MU2) & ((Z)(W))<-0 |
| 25 | MV A1, A2 | (A1) <- (A2) |
| 18 | MV1 M, Z | (0A1) <- (Z) & (A1) <- (A1)+1 |
| 19 | MV2 M, Z | (1A1) <- (Z) & (A1) <- (A1)+1 |
| 1C | MV1 M, TEMP | (0A1) <- (TEMP) & (A1) <- (A1)+1 |
| 27 | MV2 M, TEMP | (1A1) <- (TEMP) & (A1) <- (A1)+1 |
| 2D | MV1 M, W | (0A1) <- (W) |
| 1D | MV2 M,W | (1A1) <- (W) |
| 07 | MV TEMP, ML | (TEMP) <- (ML) & (ML) <- (1A1) |
| 0A | MV CO1, ML | (CO1) <- (ML) & (ML) <- (0A1) |
| 32 | MV CO2, ML | (CO2) <- (ML) & (ML) <- (0A1) |
| 24 | MV A2, ML | (A2) <- (ML) & (ML) <- (1A1) |
| 0C | MV1 MU1, ML | (ML)<-(0A1) & (MU1)<-(ML) & (MU2)<-(MU1) & (MU3)<-(MU1)*(MU2) & (A1)<-(A1)+1 & ((Z)(W))<-0 |
| 1F | MV2 MU1, ML | (ML)<-(1A1) & (MU1)<-(ML) & (MU2)<-(MU1) & (MU3)<-(MU1)*(MU2) & (A1)<-(A1)+1 & ((Z)(W))<-0 |
| 08 | MV3 MU1, ML | (ML)<-(1A1) & (MU1)<-(ML) & (MU2)<-(MU1) & (MU3)<-(MU1)*(MU2) & (A1)<-(A1)+1 & Load ((Z)(W)) |
| 0E | MV4 MU1, ML | (ML)<-(0A1) & (MU1)<-(ML) & (MU2)<-(MU1) & (MU3)<-(MU1)*(MU2) & (A1)<-(A1)+1 & Load ((Z)(W)) |
| 1B | MV5 MU1, ML | (ML)<-(0A1) & (MU1)<-(ML) & (MU2)<-(MU1) |

**TABLE 1. Instruction Set--Data Transfer Group**

| Machine Code (HEX) | Mnemonic | Function |
|---|---|---|
| 1E | MV6 MU1, ML | (ML)<-(1A1) & (MU1)<-(ML) & (MU2)<-(MU1) |
| 2C | MV7 MU1, ML | (ML)<-(0A1) & (MU1)<-(ML) & (MU2)<-(MU1) & (A1)<-(A1)+1 |
| 10 | MV X, ML | (ML) <- (0A1) & (X) <- (ML) |
| 16 | MV Y, ML | (ML) <- (0A1) & (X) <- (ML) & (A1) <- (A1) -1 |
| 09 | MV1 ML, M | (ML) <- (0A1) & (A1) <- (A1) +1 |
| 1A | MV2 ML,M | (ML) <- (0A1) |
| 06 | MV3 ML, M | (ML) <- (1A1) |
| 3E | MVI A1, data | (A1) <- (byte2) |

**TABLE 2. Instruction Set--Arithmetic Group**

| Machine Code (HEX) | Mnemonics | Function |
|---|---|---|
| 26 | ADD | (Z) <- (Y)+(X) |
| 17 | SUB | (Z)<-(Y)-(X) |
| 13 | DIV1 | (Z)<-(Y)±(X) |
| 14 | DIV2 | (Z)<-(Y)±(X) |
| 29 | INR A2 | (A2) <- (A2)+1 |
| 28 | DCR A2 | (A2) <- (A2)-1 |
| 05 | INR A1 | (A1) <- (A1)+1 |
| 21 | DCR A1 | (A1) <- (A1)-1 |

**TABLE 3. Instruction Set--Branch Group**

| Machine Code (HEX) | Mnemonics | Function |
|---|---|---|
| 36 | JMP addr | (PC) <- (byte2) |
| 01 | BNCH addr | (PC) <- (byte2) & (ST) <- (PC)+2 |
| 03 | RTN | (PC) <- (ST) |
| 02 | JNE addr | $IF\,[\,(A1) \neq (CO1_{6;0})\,] \Rightarrow$ (PC) <- byte2 |
| 2F | JBG addr | $IF\,[\,(BG) = 1\,] \Rightarrow$ (PC) <- byte2 |
| 34 | JLE addr | $IF\,[\,(CO1) \leq (CO2)\,] \Rightarrow$ (PC) <- byte2 |
| 35 | JLT addr | $IF\,[\,(CO1) < (CO2)\,] \Rightarrow$ (PC) <- byte2 |

**TABLE 3. Instruction Set--Branch Group**

| Machine Code (HEX) | Mnemonics | Function |
|---|---|---|
| 37 | JEQ addr | $IF\,[\,(CO1) = (CO2)\,] \Rightarrow$ (PC) <- byte2 |
| 40 | JF1 addr | $IF\,[\,(F1) = 1] \Rightarrow$ (PC) <- byte2 |
| 41 | JF2 addr | $IF\,[\,(F2) = 1] \Rightarrow$ (PC) <- byte2 |
| 42 | JF3 addr | $IF\,[\,(F3) = 1] \Rightarrow$ (PC) <- byte2 |
| 43 | JF4 addr | $IF\,[\,(F4) = 1] \Rightarrow$ (PC) <- byte2 |

**TABLE 4. Instruction Set--Machine Control Group**

| Machine Code | Mnemonics | Function |
|---|---|---|
| 04 | SET A1 | (A1) <- (1111111) |
| 00 | NOP | No Operation |
| 3F | HLT | Halt |
| 2E | SET OVF | Check Division Overflow |
| 30 | SET F1 | (F1) <- (1) |
| 31 | RST F1 | (F1) <- (0) |
| 38 | SET F2 | (F2) <- (1) |
| 39 | RST F2 | (F2) <- (0) |
| 3A | SET F3 | (F3) <- (1) |
| 3B | RST F3 | (F3) <- (0) |
| 3C | SET F4 | (F4) <- (1) |
| 3D | RST F4 | (F4) <- (0) |
| 11 | SHL | $(Z_0)$ <- $(W_{15})$ ; $(W_0)$ <- (0) |
| 44 | SHD | $(Z_0)$ <- $(W_{15})$ ; $(W_0)$ <- D.C. |

## 3.5 The Stack And Subroutine Execution

One of the important techniques in software design for microprocessor systems is the use of subroutines. It is in fact a task which is required to be carried out, at several points in a program, and requires the execution of the same group of instructions. It is more cost effective in terms of memory usage if the needed group of instructions appears only once but can be executed from several points in a program. Central to this is a storage structure called stack,

which is a collection of registers organized in such a way that the last data written is the first one available to be read. In other words, a stack is a last-in first-out memory, or LIFO memory. Depend on the level of subroutine calls, a stack located in a microprocessor is of fixed depth, typically between four and 16 words. In case of this study, provision is made for only one level of subroutine call, which was primarily needed for the execution of division subroutine.

## 3.6 Algorithm Transformation

The design of microprocessor system requires a knowledge of both hardware and software. The mathematical background of the algorithm was developed in chapter two which resulted in the flow diagram of Fig. 2.1. This flow diagram is the key solution to the question of what are the basic requirements of the processor in order to achieve a reasonably good performance together with the basic goal of logic design which is a system that functions as required and is reliable, easy to maintain, and cost effective. A closer inspection of the proposed algorithm reveals the following facts

- The algorithm is recursive, and because the final result, i.e., the spectrum of the signal is theoretically irrespective of the magnitude of the sampled signal, the initial scaling of the data can result in better computational accuracy as well as avoiding under-flow and/or over-flow without any penalties.
- Among the four basic required arithmetic operations (+, -, $\times$, $\div$), multiplication is the most important one. That is primarily because it is encountered more frequently in the proposed routine, and also because of the fact that some other calculations like the square or mean value of the data can be carried out more efficiently, provided a high speed multiplier is available.

- In each recursion, one division is required that provides a result which is always less than one. Thus, a serial non-restoring divider can meet the desired accuracy and speed specifications.
- The main block in all the arithmetic operations and also some address generations of the memory is the 32-bit adder. A two-level, 32-bit, carry look ahead adder proved to meet the speed demands of other blocks as well as providing an efficient area to speed ratio.

The main routine of the program can be partitioned into several smaller subroutines

1. **Initialization**-- To decrease the possibility of overflow, and also to speed the processing, the processor calculates the mean value of data by successive multiply-accumulate operations. Each datum is multiplied by a one, which is already stored in TEMP register, and the result will be added to the outcome of next multiply operation. Two copies of data are then generated by subtracting the mean value from the original data set and will be stored in page zero and page one of the data RAM respectively. They are shown as strings $b(n)$ and $b'(n)$ in the flow diagram of Fig 2.1.

2. **The Main Loop**-- The two variables $NOM$ and $DEN$ are calculated as

$$DO \quad 30 \quad T = 1, (N-M)$$

$$NOM = NOM + b(T) \cdot b'(T)$$

$$DEN = DEN + b(T) \cdot b(T) + b'(T) \cdot b'(T)$$

$$30 \quad CONTINUE$$

again through successive multiply-accumulate operations. Since for every T

$$(b(T) - b'(T))^2 \geq 0$$

then

$$b(T) \cdot b(T) + b'(T) \cdot b'(T) \geq 2b(T) \cdot b'(T)$$

$$\Rightarrow \sum_T b(T) \cdot b(T) + b'(T) \cdot b'(T) > 2\sum_T b(T) \cdot b'(T)$$

$$\Rightarrow DEN > 2 \cdot NOM \Rightarrow A(M) < 1$$

In order to calculate the value of $A(M)$ accurately, the processor first calculates the value of $DEN$. If $DEN$ is more than 16-bit in length, then both $DEN$ and $NOM$ will be divided by N, where N is the total number of sampled data. The result of any division operation must not be more than 16-bit long, otherwise the OVF signal will be logic one, which shows the sampled data are not properly scaled. The processor multiplies the value of $NOM$ by 4000 (Hex.) and stores the result in the accumulator. Thus, the result of division of $NOM$ by DEN has its radix point after bit 14, i.e., it has 14 meaningful digits before radix point which give the desired accuracy.

Multiplication of two numbers with different radix points can be simply carried out without any pre-adjustment of radix points. However, for addition and subtraction, the operands have to be adjusted in order to have their radix points at the same place. In updating the value of the previously calculated filter coefficients, A(m), and the data in data RAM, using the value of A(M), it is therefore important to consider the fact that A(M) has its radix point after bit 14. Each time, at the end of execution of the main loop, the processor adjusts the value of A(M) so that it has only 9 digits before the radix point. It is therefore possible to assign a broad range of values, both smaller and bigger than zero, to the filter coefficients A(m) when the processor updates these quantities. The processor then provides two copies of filter coefficients in pages zero and one of the RAM respectively. The update equation for the filter coefficients can be written as

$DO \quad 40 \quad T = 1, (M-1)$

$$A(T) = AA(T) - A(M) \cdot AA(M-T) \qquad \text{(EQ 18)}$$

40 *CONTINUE*

To ease the addressing of the memory, the second copy of the filter coefficients in page one of the memory is stored downward, compared to the first copy in page zero, i.e., the address of the last filter coefficient has a value which is smaller than the address of the first one. Therefore, in calculation of (EQ 18), the processor needs to address pages zero and one of the memory just in one direction (Upward). Note that in each recursion one location of the memory will be released and therefore can be used to store the recently calculated filter coefficient. In most cases the addressing of the memory locations is done upward and sequentially by using the address register A1 as a counter. There are however occasions that it is necessary to use counter A2 to keep track of memory locations in page one, e.g., for copying filter coefficients in page one.

3. **Division Subroutine**-- The only subroutine which is used in the program is the division subroutine. The machine codes of this routine are stored in locations 1F0 (Hex.) to the end of the ROM. The algorithm uses a non-restoring method and mainly consists of successive shift and add or subtract operations.

The complete program of the flow diagram of Fig. 2.1, written by using assembly language mnemonics, is given in appendix B.

## • Summary

In this chapter, the operation of the processor was described. First a general overview on the processor was given and its pin configuration was discussed and the function of each terminal was explained briefly. In subsequent paragraphs, the organization of the memory (RAM) and the architecture of the processor were described in more details. The major parts of the processor were listed and the function of each was clarified concisely. The floor plan of the

processor shows how these elements are distributed within the chip and the processor's block diagram depicts the interaction of these components. Particular attention was paid to the operation of the control unit and its pipelined nature in synchronizing all the events within the processor was explained by an example.

Finally the instruction set available in the processor was introduced. The instructions were classified in four functional groups and presented in separate tables. Next chapter is devoted to the design methodologies of major parts of the processor. Mathematical background behind the operation of each block is discussed and its hardware design as well as the simulation results are described in detail.

# Chapter 4

# PROCESSOR BUILDING BLOCKS

## 4.1 Multiplier

For a given resolution, speed is the dominant specification of a multiplier. Therefore DSP multipliers are parallel "array" multipliers rather than the clocked "shift-and-add" of software multiplication. The cost in chip area is reduced somewhat by employment of algorithms (e.g. Booth's algorithm) to eliminate redundant operations when a string of 1's or 0's is encountered. The main benefit is single-cycle multiply speed to match other computational elements and data transfers. An accumulator combined with a multiplier is desirable since it facilitates carrying out terms like:

$$\sum b(n)\,x(n-k)$$

which are frequently encountered in the operation of filters, Fourier analysis, and vector operations. Intermediate pipelining registers enhance throughput because overhead on repet-

itive calculations is lessened. Internal feedback path can make possible, for example, single-cycle computation of the common DSP operation $M \times D + B$. Pipelining, on the other hand increases the bandwidth of the system for a given latency by allowing simultaneous execution of several tasks at the cost of higher gate count due to additional latches. Fig. 1 is a convenient star representation of the concept of the multiplication of two 16-bit operands. The scheme is based on simultaneous generation and reduction of partial products which takes place in two independent steps.

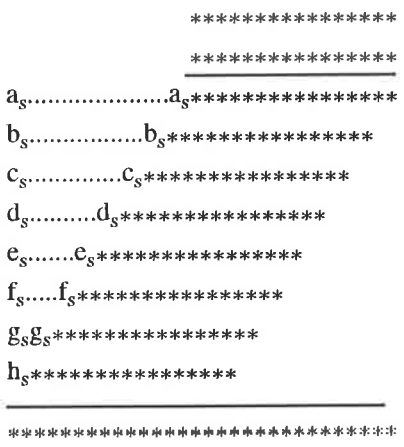



**FIGURE 4.1-- Star representation of multiplication of two 16-bit numbers.**

### 4.1.1 Generation Of The Partial Products And Booth's Algorithm

Partial products can be generated using AND gates and in this way an n-bit multiplier generates n partial products. However it is possible to reduce the number of partial products by using encoding techniques. The modified Booth's algorithm [7] is one of these techniques which reduces the number of partial products by half. The original Booth's algorithm suggests to skip over any string of 1's or 0's. Skipping over a string of 0's is straightforward, but a string of 1's is equal to a 1 followed by n 0's less one. A more commonly used algorithm is modified Booth which differs from original Booth in that it always generates n/2 independent

partial products whereas the former produces a varying number (at most n/2) of dependent partial products.

The modified Booth's algorithm generates 8 partial products for a 16 by 16-bit multiplier by encoding 3-bit groups. Each multiplier is divided into groups of three bits and adjacent groups share a common bit. In using the Booth's algorithm for two's complement numbers, the most significant bit has a weight of $(-2^n)$ and requires the multiplier to be padded by a 0 to the right to form 8 complete 3-bit groups. Of course in two's complement, the sign bit must be extended to the full width of the final result, as shown by the repetitive terms in Fig. 4.1. Fig. 4.2 depicts the encoding scheme of modified Booth's algorithm [37].

| Bit | | | operation | |
|---|---|---|---|---|
| $2^1$ | $2^0$ | $2^{-1}$ | | |
| $m_{i+1}$ | $m_i$ | $m_{i-1}$ | | |
| 0 | 0 | 0 | add zero | +0 |
| 0 | 0 | 1 | add multiplicand | +x |
| 0 | 1 | 0 | add multiplicand | +x |
| 0 | 1 | 1 | add twice the multiplicand | +2x |
| 1 | 0 | 0 | subtract twice the multiplicand | -2x |
| 1 | 0 | 1 | subtract multiplicand | -x |
| 1 | 1 | 0 | subtract multiplicand | -x |
| 1 | 1 | 1 | subtract zero | -0 |

**Figure 4.2--Encoding scheme of the modified Booth's algorithm.**

Based on this scheme, if the shared bit is 1 the subtraction indicated since we prepare for a string of 1's. For the lowest order action, only four actions are possible and derived from bits $m_1m_0$ padded with a 0 to the right. In generating the next partial product, the participating bits are $m_3m_2m_1$. The resulting partial product must be shifted by two with respect to the previous one, and this is true for any partial product so generated.

### 4.1.2 Sign Bit Extension And Add_One Method

Since a modified Booth algorithm is adopted, eight 17 bits partial products are generated. The sign bits of these partial products are located 2 bits apart from each other. Therefore, the sign bit of partial product zero is to be extended to the sign bit position of partial product 7, which is the most significant partial product. Obviously this operation needs a large number of circuits. However using add_one method [40], the sign bit extension can be treated in a simple way.

Based on this method instead of extending the sign bit (bit 17) of partial products, the following three steps can be adopted

4. Invert the sign bit (bit 17) of the partial products.
5. Set a "1" between the sign bit of partial product (i-1) and partial product (i).
6. Set a "1" in the partial product zero sign bit position.

It should be noted that the result of the multiplication of two 16-bit two's complement numbers is always less than or equal to 31-bit long. The proof of this method is as following.

The result of the multiply operation can be found by adding the partial products as is given by

$$pp_0 + pp_1 \cdot 2^2 + pp_2 \cdot 2^4 + pp_3 \cdot 2^6 + pp_4 \cdot 2^8 + pp_5 \cdot 2^{10} + pp_6 \cdot 2^{12} + pp_7 \cdot 2^{14} =$$

$$\left(a_s \sum_{i=16}^{30} 2^i + \sum_{i=0}^{15} a_i \cdot 2^i - a_s \cdot 2^{31}\right) + \left(b_s \sum_{i=16}^{28} 2^i + \sum_{i=0}^{15} b_i \cdot 2^i - b_s \cdot 2^{29}\right) \cdot 2^2 +$$

$$\left(c_s \sum_{i=16}^{26} 2^i + \sum_{i=0}^{15} c_i \cdot 2^i - c_s \cdot 2^{27}\right) \cdot 2^4 + \left(d_s \sum_{i=16}^{24} 2^i + \sum_{i=0}^{15} d_i \cdot 2^i - d_s \cdot 2^{25}\right) \cdot 2^6 +$$

$$\left(e_s \sum_{i=16}^{22} 2^i + \sum_{i=0}^{15} e_i \cdot 2^i - e_s \cdot 2^{23}\right) \cdot 2^8 + \left(f_s \sum_{i=16}^{20} 2^i + \sum_{i=0}^{15} f_i \cdot 2^i - f_s \cdot 2^{21}\right) \cdot 2^{10} +$$

$$\left(g_s \sum_{i=16}^{18} 2^i + \sum_{i=0}^{15} g_i \cdot 2^i - g_s \cdot 2^{19}\right) \cdot 2^{12} + \left(h_s \cdot 2^{16} + \sum_{i=0}^{15} h_i \cdot 2^i - h_s \cdot 2^{17}\right) \cdot 2^{14} \qquad \text{(EQ 19)}$$

Coefficients $a_s$ to $h_s$ are sign bits of partial products zero to seven. The sign extension term in (EQ 19) can be separated and shown by a term referred to as S

$$S = \left(a_s \sum_{i=16}^{30} 2^i - a_s \cdot 2^{31}\right) + \left(b_s \sum_{i=16}^{28} 2^i - b_s \cdot 2^{29}\right) \cdot 2^2 + \left(c_s \sum_{i=16}^{26} 2^i - c_s \cdot 2^{27}\right) \cdot 2^4 + \left(d_s \sum_{i=16}^{24} 2^i - d_s \cdot 2^{25}\right) \cdot 2^6 +$$

$$\left(e_s \sum_{i=16}^{22} 2^i - e_s \cdot 2^{23}\right) \cdot 2^8 + \left(f_s \sum_{16}^{20} 2^i - f_s \cdot 2^{21}\right) \cdot 2^{10} + \left(g_s \sum_{i=16}^{18} 2^i - g_s \cdot 2^{19}\right) \cdot 2^{12} + \left(h_s \cdot 2^{16} - h_s \cdot 2^{17}\right) \cdot 2^{14}$$

Defining new parameters

$$a_s = (1 - \bar{a}_s) \qquad b_s = (1 - \bar{b}_s) \qquad c_s = (1 - \bar{c}_s) \qquad d_s = (1 - \bar{d}_s)$$

$$e_s = (1 - \bar{e}_s) \qquad f_s = (1 - \bar{f}_s) \qquad g_s = (1 - \bar{g}_s) \qquad h_s = (1 - \bar{h}_s)$$

and considering

$$\sum_{i=m}^{n} 2^i = 2^{n+1} - 2^m$$

it is concluded that

$$S = \left(a_s \sum_{i=16}^{30} 2^i - a_s \cdot 2^{31}\right) + \left(b_s \sum_{i=18}^{30} 2^i - b_s \cdot 2^{31}\right) + \left(c_s \sum_{i=20}^{30} 2^i - c_s \cdot 2^{31}\right) + \left(d_s \sum_{i=22}^{30} 2^i - d_s \cdot 2^{31}\right) +$$

$$\left(e_s \sum_{i=24}^{30} 2^i - e_s \cdot 2^{31}\right) + \left(f_s \sum_{i=26}^{30} 2^i - f_s \cdot 2^{31}\right) + \left(g_s \sum_{i=28}^{30} 2^i - g_s \cdot 2^{31}\right) + \left(h_s \cdot 2^{30} - h_s \cdot 2^{31}\right)$$

$$\Rightarrow S = (1 - \bar{a}_s)\left(2^{15} - 1\right) \cdot 2^{16} - (1 - \bar{a}_s) \cdot 2^{31} + (1 - \bar{b}_s)\left(2^{13} - 1\right) \cdot 2^{18} - (1 - \bar{b}_s) \cdot 2^{31} +$$

$$(1 - \bar{c}_s)\left(2^{11} - 1\right) \cdot 2^{20} - (1 - \bar{c}_s) \cdot 2^{31} + (1 - \bar{d}_s)\left(2^9 - 1\right) \cdot 2^{22} - (1 - \bar{d}_s) \cdot 2^{31} +$$

on the other hand and in spite of its difficult physical realization, is another option in reducing the propagation stages. Moreover, a 4-2 compressor unit is employed instead of conventional full adders which are usually used in multiplier arrays. As is shown in the block diagram of Fig. 4.4, the multiplier consists of three major blocks

1. Booth encoding block: generates 8 partial products according to the modified Booth's algorithm.
2. Compressor: reduces four lines of partial products to two. In this block the Wallace tree structure has been applied to reduce the propagation delay.
3. 32 bits carry_look_ahead adder: Performs the final addition and generates the final result.

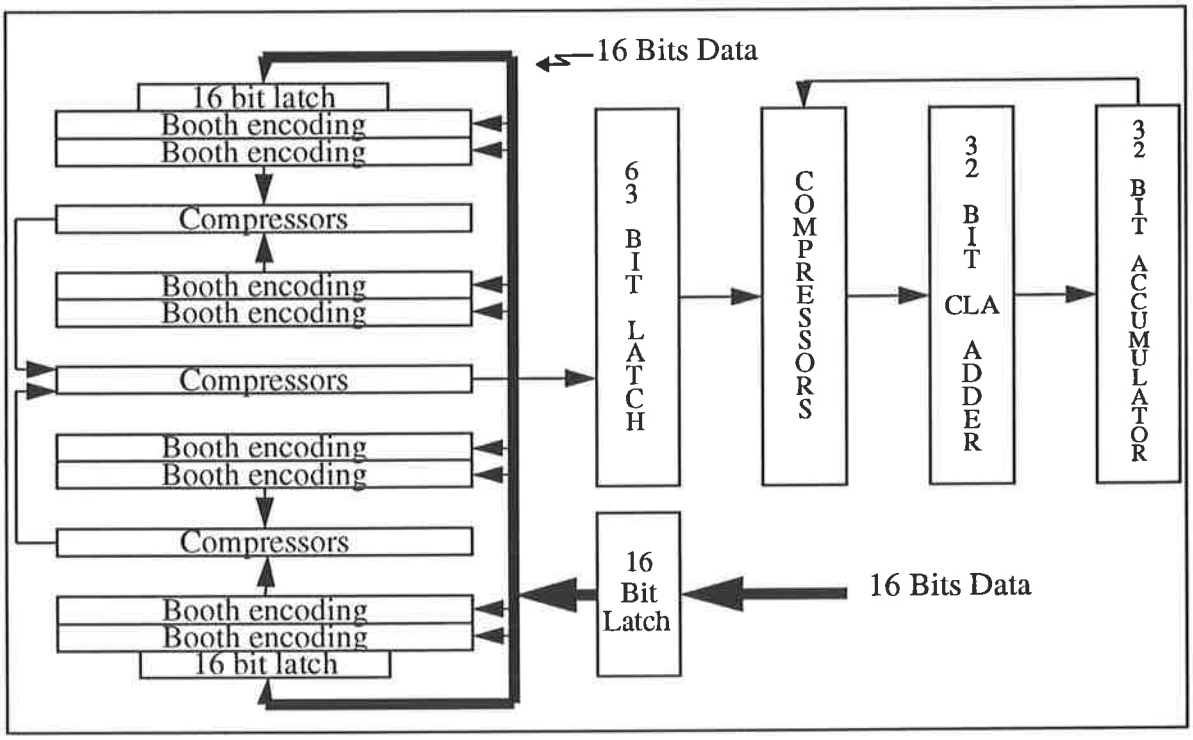



**Figure 4.4--Block diagram of the 16 by 16 multiplier.**

The circuit diagram for a 4-2 compressor [22] is shown in Fig. 4.5(a), and its equivalent circuit is shown in Fig. 4.5(b). It has five inputs and three outputs and is capable of compressing four partial products (x1, x2, x3, x4) into two new partial products (sum, co), simultane-

ously. Note that the generation of "Cout" is independent of the arrival of 'cin' therefore no carry signal will propagate through the array. Only two stages of this compressor are needed to sum up 8 partial products and three stages for multiply accumulator.

The Booth encoding block consists of two major parts [37]. The first part is the control section which generates the control signals N, x1 and x2 by considering the logic levels of 3-bit groups of the multiplier. These signals control the operation of all the other similar blocks which act on the input multiplicand. The circuit diagram of the control section is shown in

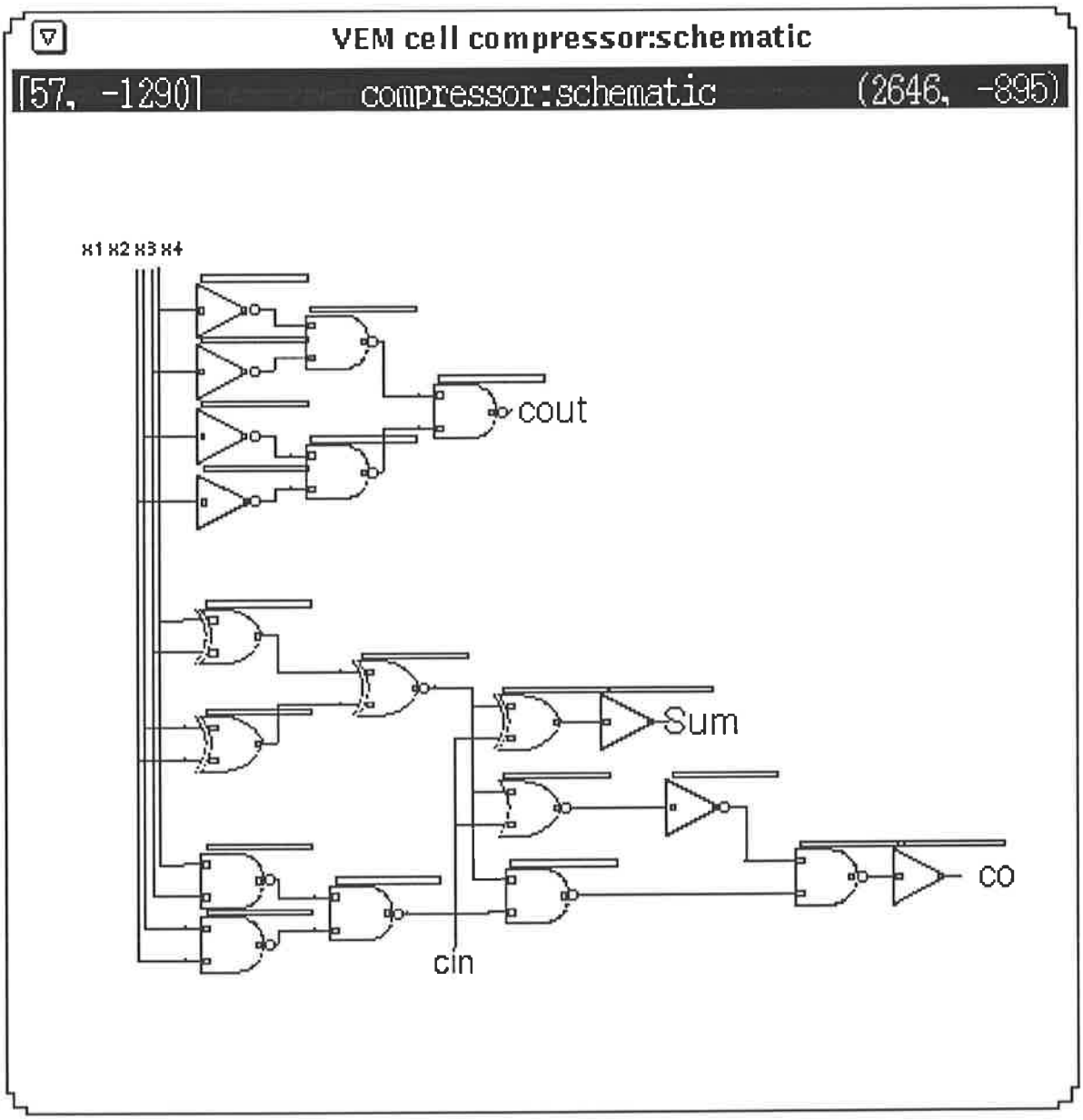



**Figure 4.5(a)--Circuit diagram of a 4-2 compressor.**

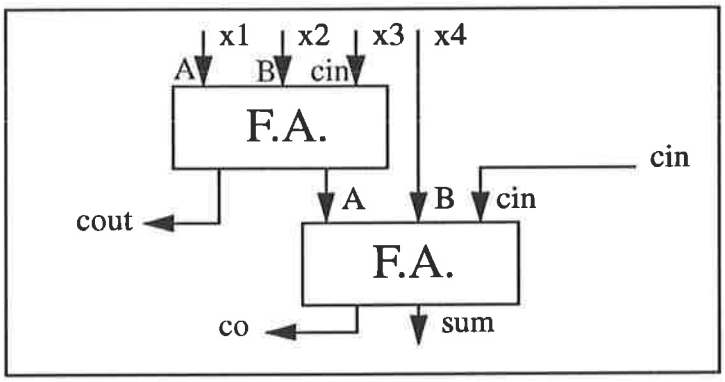



Figure 4.5(b)--Equivalent circuit of a 4-2 compressor.

Fig. 4.6(a). The second part is essentially a multiplexer which outputs one of the signals multiplicand(n), multiplicand(n-1), zero or their complements as proposed by the control signals x1, x2 and N. The circuit diagram of the multiplexer block is depicted in Fig. 4.6(b).

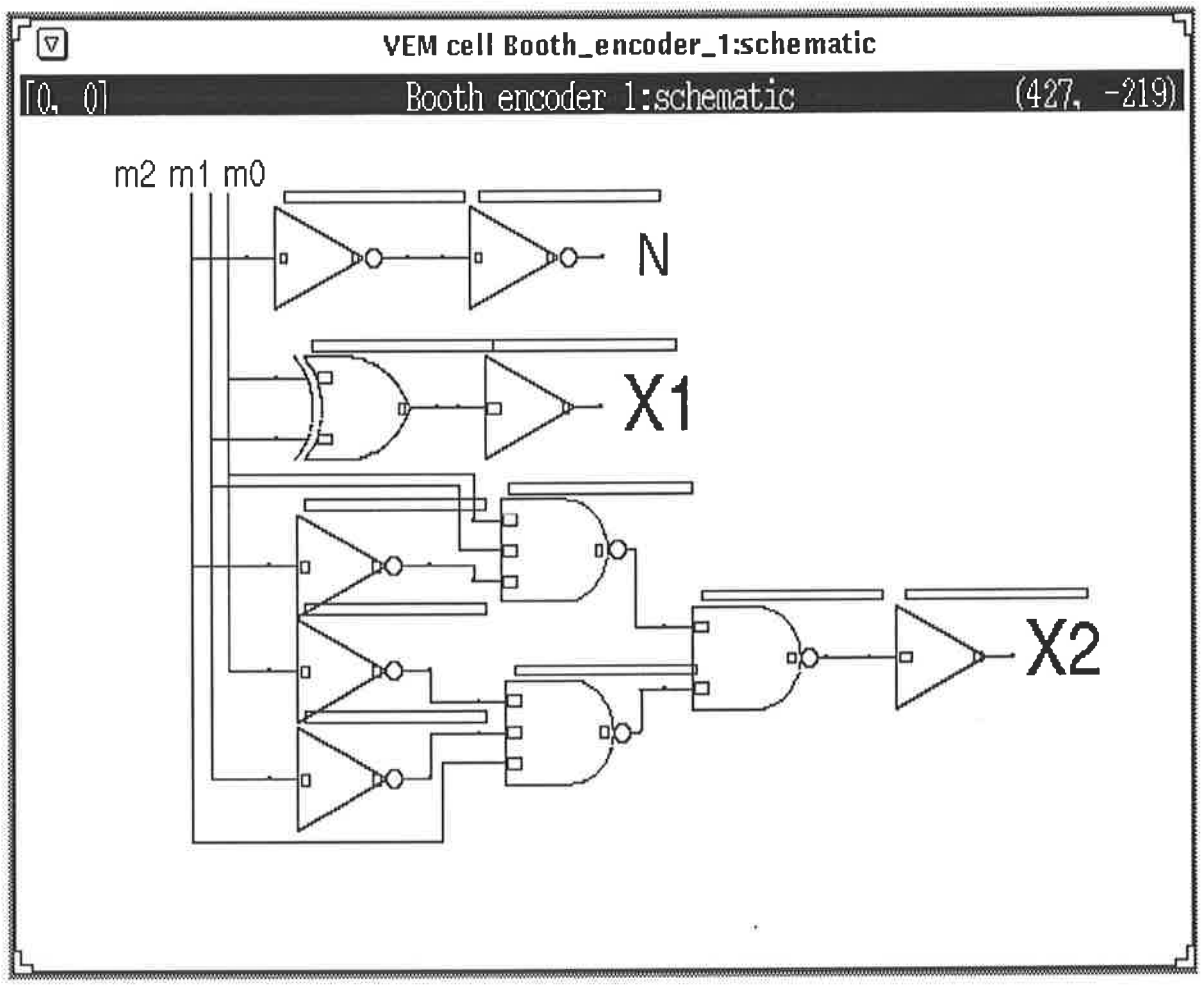



**Figure 4.6(a)--Control section of the Booth encoding block.**

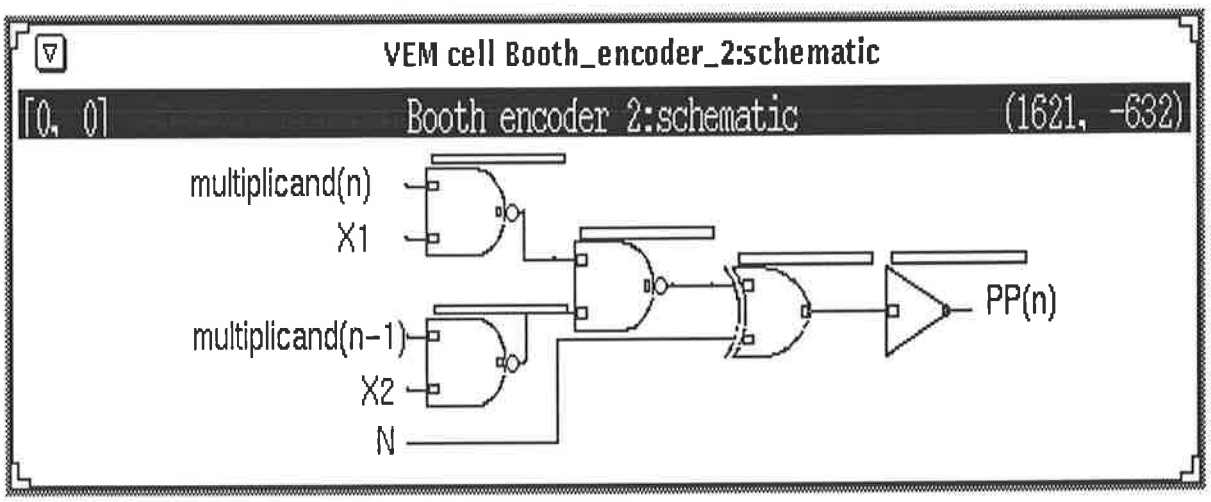



**Figure 4.6(b)--The circuit diagram of the multiplexer used in Booth Encoding Block**

A complementary pass transistor logic (CPL) is applied to the Booth encoding blocks [41]. It consists of complementary inputs and outputs, NMOS pass transistor logic network and CMOS output inverters. It is necessary to use output inverters to amplify the output of the pass transistors which their high level is less than supply voltage by threshold voltage of the pass transistors. To acquire better noise margin, the threshold voltage of the output inverters has to be less than Vcc/2. The basic circuit modules, shown in Fig. 4.7, are used to construct the Booth encoding blocks shown in Fig. 5.

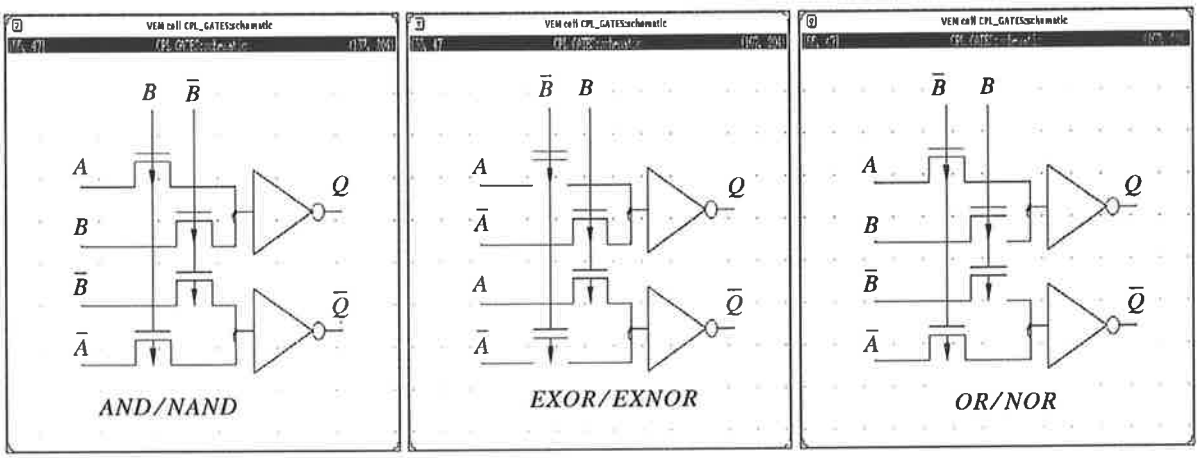



**Figure 4.7-- CPL Circuit modules**

### 4.1.4 Multiplier Simulation Result And Performance Estimation

If a system performs only one task at a time the bandwidth is defined as the inverse of the latency. In general, bandwidth is the number of tasks that can be performed in each specific time interval. Pipelining technique, in addition to the parallelism, can result in increasing the bandwidth of the system while keeping the latency constant. The increased bandwidth is achieved by dividing the combinational logic into several stages separated by latches. Therefore increasing the bandwidth by pipelining results in higher gate count due to additional latches.

Referring to Fig. 4.4 it can be inferred that both multiplicand and multiplier can be loaded in two clock cycles. At clock cycle three the result is ready at the output of the 63-bit latch to be added to the current data in the accumulator and also the second multiplicand can be loaded into the 16-bit input latch. At clock cycle four the second multiplier can be loaded and the final result of the first multiply accumulate operation is ready at the input of the 32-bit accumulator. Fig. 4.8 depicts a typical simulation sequence of the multiplier circuitry. In this simulation the multiplier is tested against the most common operation in DSP programs, as well as the algorithm of Fig. 2.1, which is the calculation of terms like

$$(A_1) \cdot (B_1) + (A_2) \cdot (B_2)$$

Timing description of the simulation is as follows

1. At time 8300 ns, the first number (1C) is moved to register MU1 and accumulator is reset to zero.
2. At time 8500 ns, the second number (11) is moved to register MU1.
3. At time 8700 ns, the third number (-1C -> two's complement -> FFE4) is moved to register MU1 and the contents of registers MU3 and accumulator are updated.

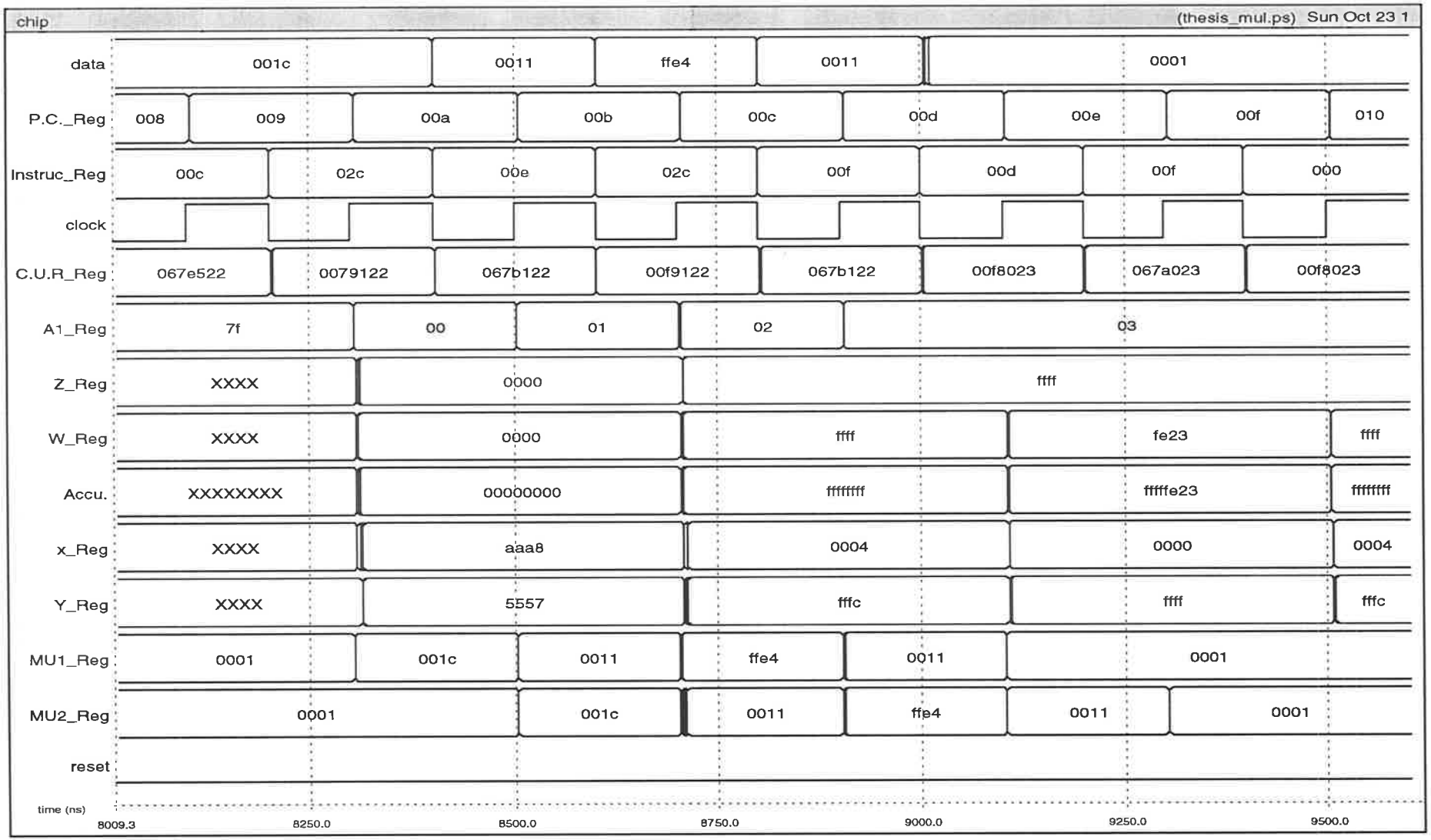



**Figure 4.8-- Simulation result of the calculation of $(1C) \cdot (11) + (-1C) \cdot (11)$ , all the numbers are in Hexadecimal base.**

4. At time 8900 ns, the fourth number (11) is moved to register MU1.

So far all the four numbers have been moved to the multiplier. In order to get the final result, which is obviously zero, it is always necessary to terminate the sequence of data transfers to register MU1 by three extra data move operations. These additional operations put the final result in the accumulator, and prepare the multiplier for the next sequence of multiply-accumulate operations by loading registers MU1 and MU2 with a 1. In this simulation the final result is moved to the accumulator at time 9500 ns. Note that the content of the accumulator is in fact the complement of the actual result. The sequence of instructions required for this simulation is as following

| | |
|---|---|
| MV1 MU1, ML | (Instruction Register: 0C) |
| MV7 MU1, ML | (Instruction Register: 2C) |
| MV4 MU1, ML | (Instruction Register: 0E) |
| MV7 MU1, ML | (Instruction Register: 2C) |
| MV1 MU1, TEMP | (Instruction Register: 0F) |
| MV2 MU1, TEMP | (Instruction Register: 0D) |
| MV1 MU1, TEMP | (Instruction Register: 0F) |
| NOP | (Instruction Register: 00) |

Note that due to the pipelined nature of the processor, the instruction register always shows the machine code of the next instruction to be executed.

The maximum rate at which this pipeline can operate depends on the maximum and minimum propagation delay of the combinational logic. The minimum clock period $\Delta t$ required for each stage is equal to:

$$\Delta t > t_{max} - t_{min} + t_g$$

where

$t_{max}$ = maximum propagation delay of the stage

$t_{min}$ = minimum propagation delay of the stage

$t_g$ = gate width or set up time required for the data to be valid at the input of the latches in order to be stored properly.

The maximum rate is determined by the slowest block and is equal to: $1/\Delta t_{max}$.

The result of the simulation shows that the minimum clock period required is about 15 ns.

## 4.2 32-Bit Carry_Look_Ahead Adder

### 4.2.1 Algorithm And Hardware Implementation

Due to the simplicity and modularity that make it particularly acceptable to integrated circuit implementation, carry_look_ahead is one of the most popular methods of addition. To show the hardware implementation of the algorithm the equations for a 4-bit slice can be written as follow [25, 26]

Sum equations

$$S_0 = A_0 \oplus B_0 \oplus C_0$$

$$S_1 = A_1 \oplus B_1 \oplus C_1$$

$$S2 = A_2 \oplus B_2 \oplus C_2$$

$$S3 = A_3 \oplus B_3 \oplus C_3$$

Or in general

$$S_i = A_i \oplus B_i \oplus C_i$$

Carry equations

$$C1 = A_0B_0 + C_0(A_0 + B_0)$$

$$C_2 = A_1B_1 + C_1(A1 + B1)$$

$$C_3 = A_2B_2 + C_2(A2 + B2)$$

$$C_4 = A_3B_3 + C_3(A_3 + B_3)$$

Or in general $$C_{i+1} = A_iB_i + C_i(A_i + B_i)$$

If we define the Generate term $G_i$ as $G_i = A_i B_i$ and propagate term $P_i$ as $P_i = A_i \oplus B_i$ then

$$C_1 = G_0 + P_0C_0$$

$$C_2 = G_1 + P_1C_1$$

Substitute $C_1$ into the $c_2$ equation we have

$$C_2 = G_1 + P_1G_0 + P_1P_0C_0$$

and in the same way $C_3$ and $C_4$ can be expressed as follows

$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$

$$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

Generalizing the above procedure, the carry look ahead equation can be derived as

$$C_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots + P_iP_{i-1}\ldots P_0C_0$$

Based on this equation a carry to any bit position can be computed in two gate delays, however because of the fan in limitation it can't be realized in practice. The solution to this problem is to have several levels of carry_look_ahead. To illustrate this concept the equation for $C_4$ can be rewritten as follow

$$C_4 = G'_0 + P'_0C_0$$

where

$G'_0$ =Group generate=$G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$

and

$P'_0$ =Group propagate=$P_3P_2P_1P_0C_0$

With a fan in of 4, one level of carry_look_ahead is enough for 16 bits. Therefore two levels of carry_look_ahead are employed for 32-bit addition. The hardware implementation of the adder is shown in Fig. 4.9 for a 4-bit slice [39].

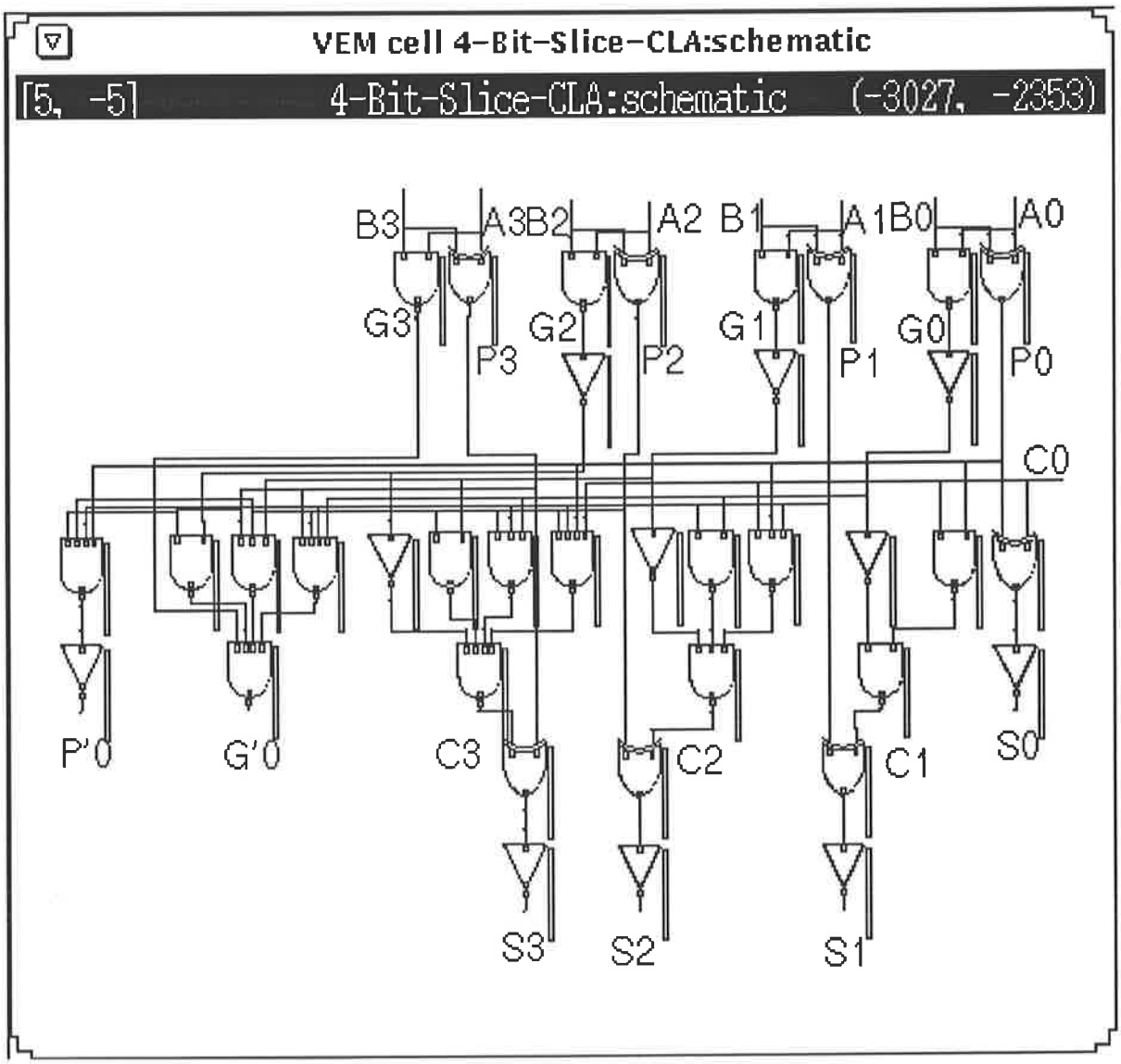



**Figure 4.9--Circuit diagram of 4-bit adder slice**

The logic equations of this circuit have been optimised by using a logic optimser software in order to reduce the number of gates, and as a result, increasing the speed as well as reducing the layout area of the block.

The first level of carries are generated using the following equations

$$C_4 = G'_0 + P'_0 C_0$$

$$C_8 = G'_1 + P'_1 G'_0 + P'_1 P'_0 C_0$$

$$C_{12} = G'_2 + P'_2 G'_1 + P'_2 P'_1 G'_0 + P'_2 P'_1 P'_0 C_0$$

The second level generate and propagate terms are

$$G'' = G'_3 + P'_3 G'_2 + P'_3 P'_2 G'_1 + P'_3 P'_2 P'_1 G'_0$$

$$P'' = P'_3 P'_2 P'_1 P'_0$$

The only second level carry in case of 32-bit adder is

$$C_{16} = G'' + P'' C_0$$

Figure 4.10 shows the gate implementation of the 4 group carry generator. It should be mentioned that in the actual design of this block, the carry generator of each slice is included in the previous slice. Although this scheme results in four different modules and reduces the modularity of the design, but it increases the speed of the adder. This in turn affects the speed of the whole system due to the importance of the adder in many operations of the processor. The increase in speed is primarily because of the resultant simplicity of the placement and routing of the four sub-modules so generated. The input carry signal of each slice is readily available as the output carry of the previous slice. Therefore there is no demand for long metal lines to distribute the carry signals between different blocks which can result in addi-

tional delay due to increasing wiring capacitance. The scheme also reduces the required area for 32-bit adder and provides a rectangular shape block which is consistent with the rest of the system.

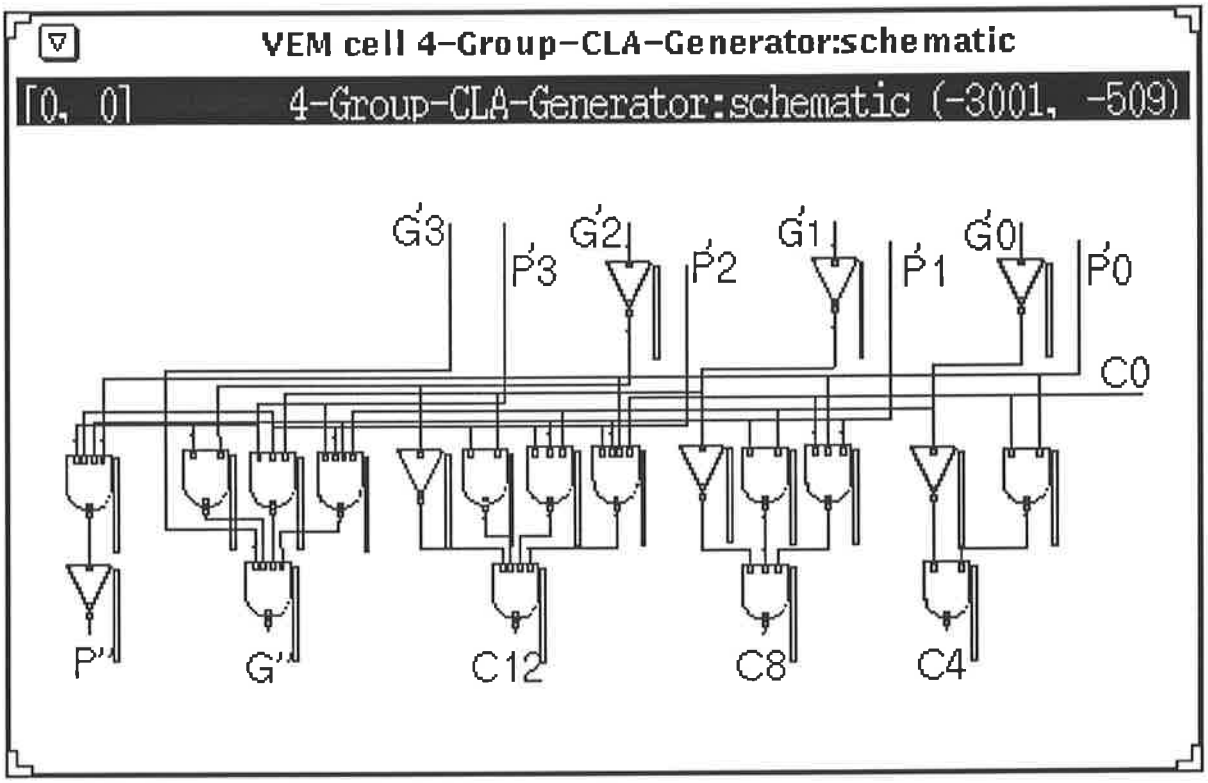



**Figure 4.10--Four group Carry_Look_Ahead generator**

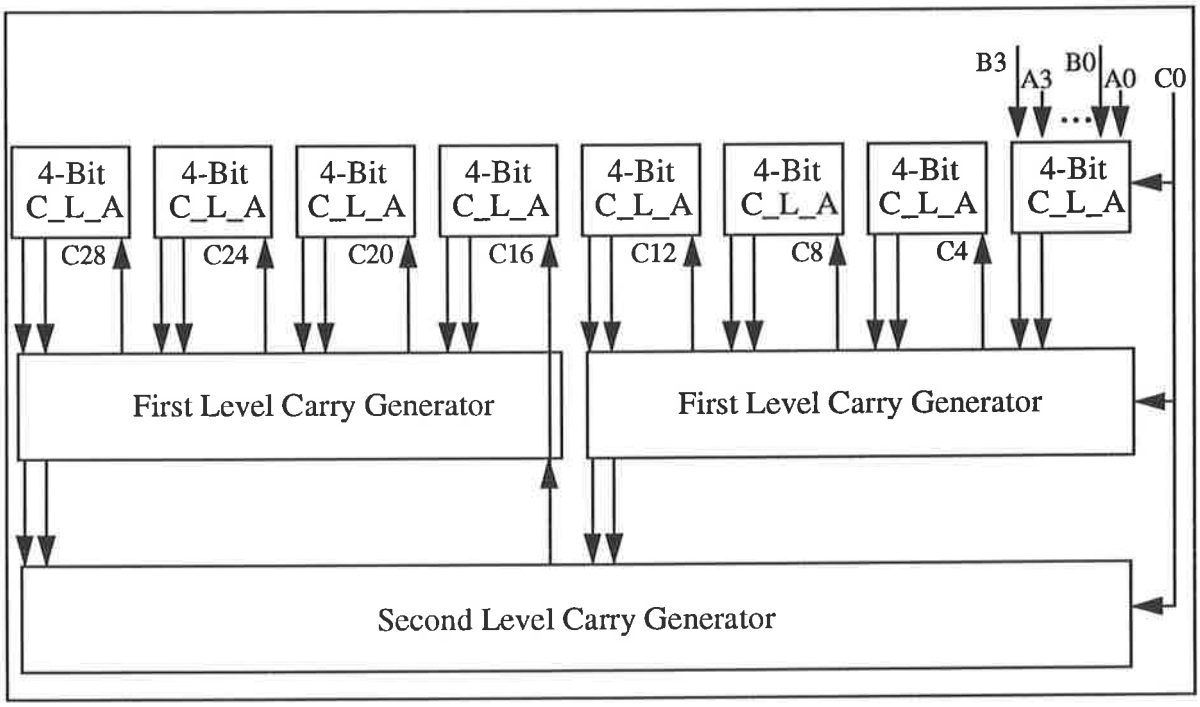



**Figure 4.11--Block diagram of 32-bit Carry_Look_Ahead adder**

The block diagram of the final adder is depicted in Fig. 4.11. An array of 32 Exclusive-OR gates is placed at one of the inputs of the adder which can be used to generate the two's complement of the input data. This gives the possibility to use this block as a 32-bit adder/subtracter simply by using the other input of the Ex-Or gates as the control line. Fig. 4.12 shows the floor plan of the 32-bit adder, and its layout is depicted in Fig. 4.13.

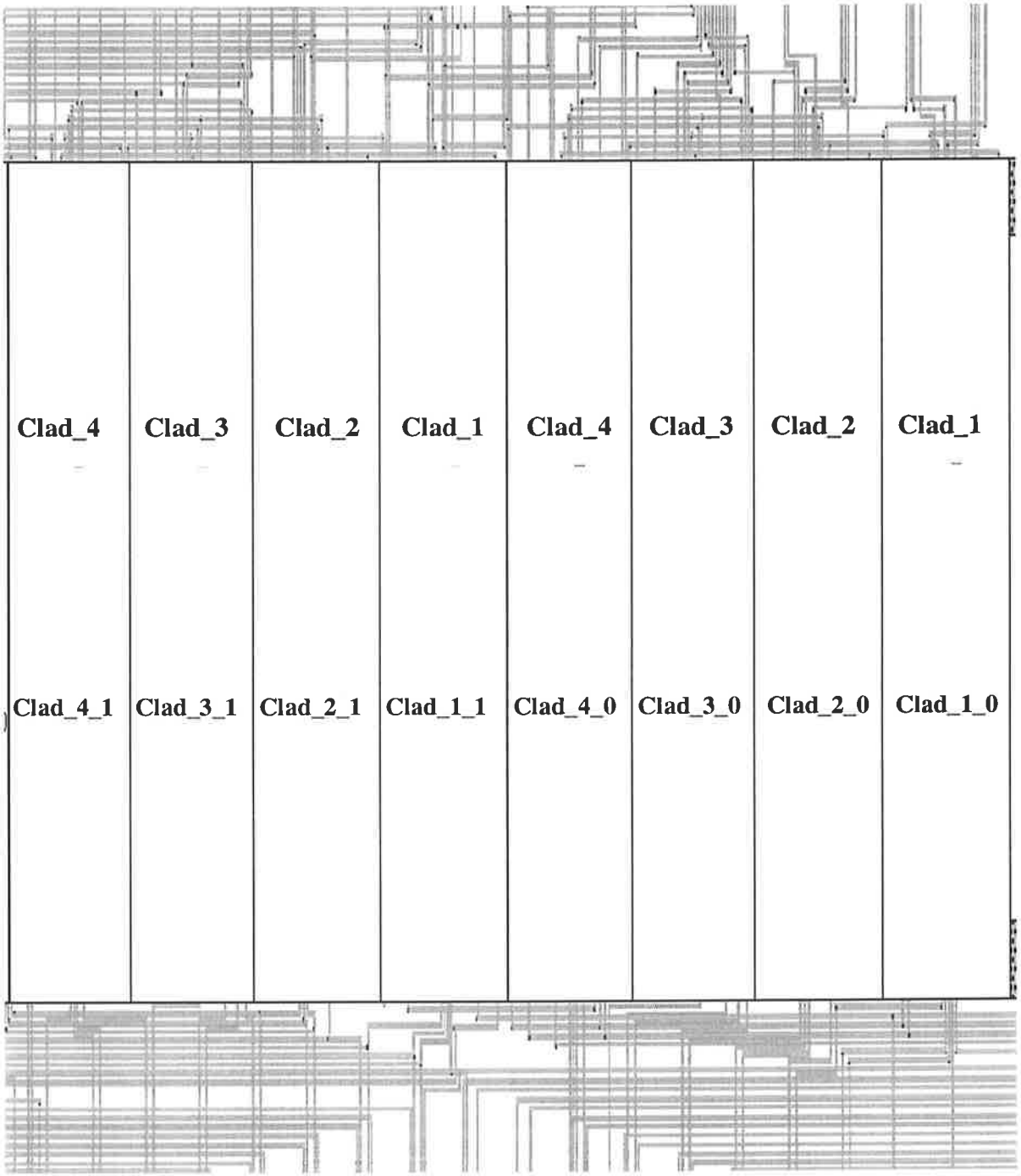



**Figure 4.12--Floor plan of the 32-bit Carry_Look_Ahead adder.**

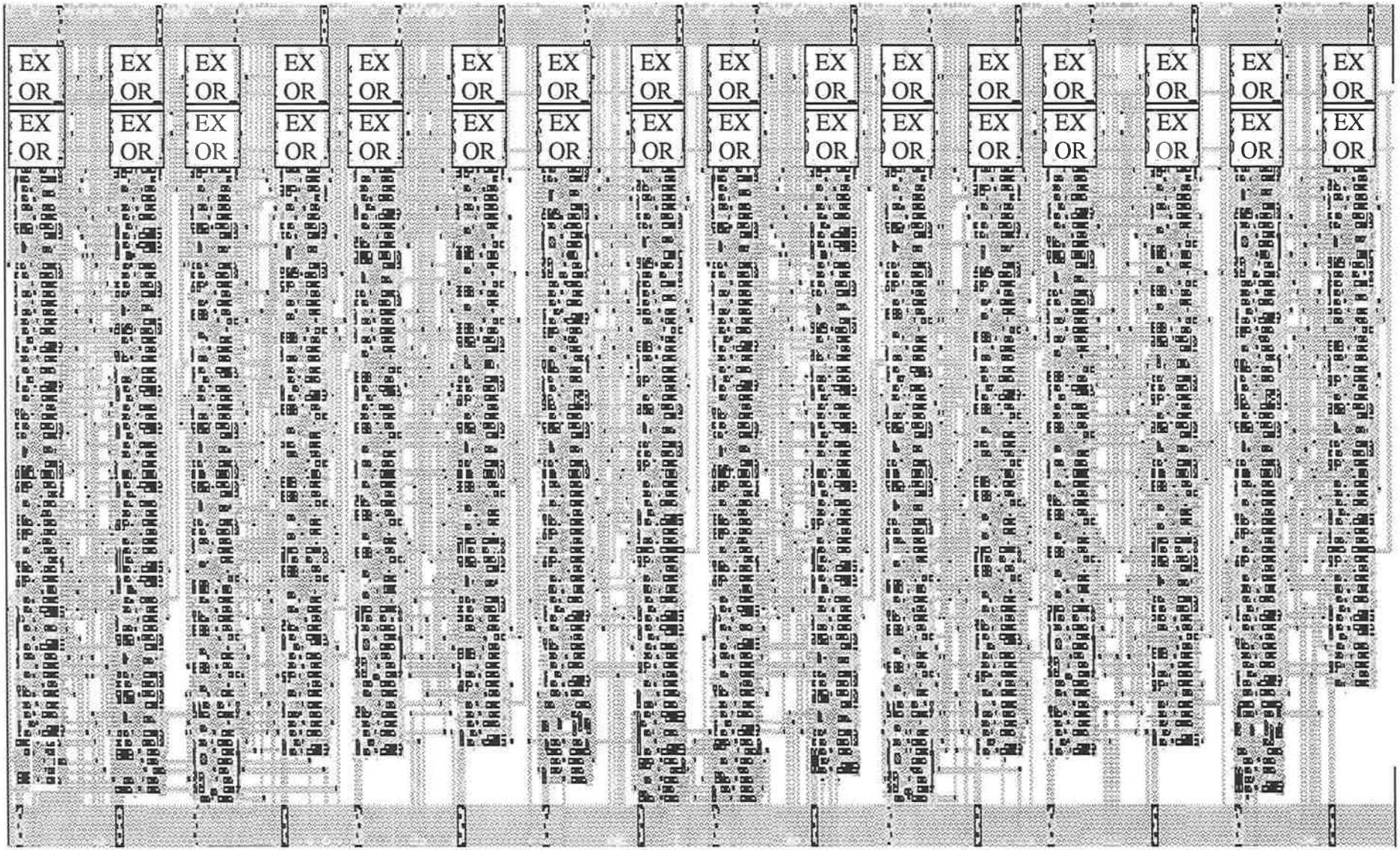



**Figure 4.13--Layout design of the 32-bit Carry_Look_Ahead adder.**

### 4.2.2 Carry_Look_Ahead Adder Simulation Result

The result of simulation shows that the maximum delay between inputs and outputs of the adder is about 11 ns. From the analysis of the circuit of Fig. 4.9, it is obvious that the propagate terms $P_i$ are readily available just in one gate delay. This is the arrival of the carry $c_i$ that determines the final status of the sum $s_i$. If the carry is zero, then the value of the sum remains unchanged, however if the carry is a logic one then it inverts the value of the sum.

Among the carry signals $c_i$, maximum propagation delay belongs to $c_{31}$. Therefore the worst situation occurs when this carry signal attempts to change the sum $s_{31}$, i.e., when $c_{31}$ changes to one as a result of propagation of the carry signal $c_0$. A typical situation in which this condition occurs is the addition of 80000000 (Hex.) with 7FFFFFFF (Hex.), when the carry $c_0$ is one. The picture of Fig. 4.14 shows the simulation result for the worst case analysis of the 32-bit adder. As it was predicted, the longest delay is associated with $s_{31}$ which settles to logic one at time 18.5 ns. The fastest output however is always $s_0$.

## 4.3 Divider

Based on their iterative operator, division algorithms can be grouped into two classes. In the first class, subtraction is the basic iterative operation and their execution time is generally proportional to the operand (divisor) length. This group can be further partitioned into many algorithms such as non-restoring division, which is comparatively slow but fast enough to be a suitable candidate for this study. The second class however is the one where multiplication is the iterative operator. This group of algorithms are faster and converge quadratically, i.e., their execution time are proportional to $\log_2$ of the divisor length.

There are usually two numbers in fixed point division, a divisor $V$ and a dividend $D$. The

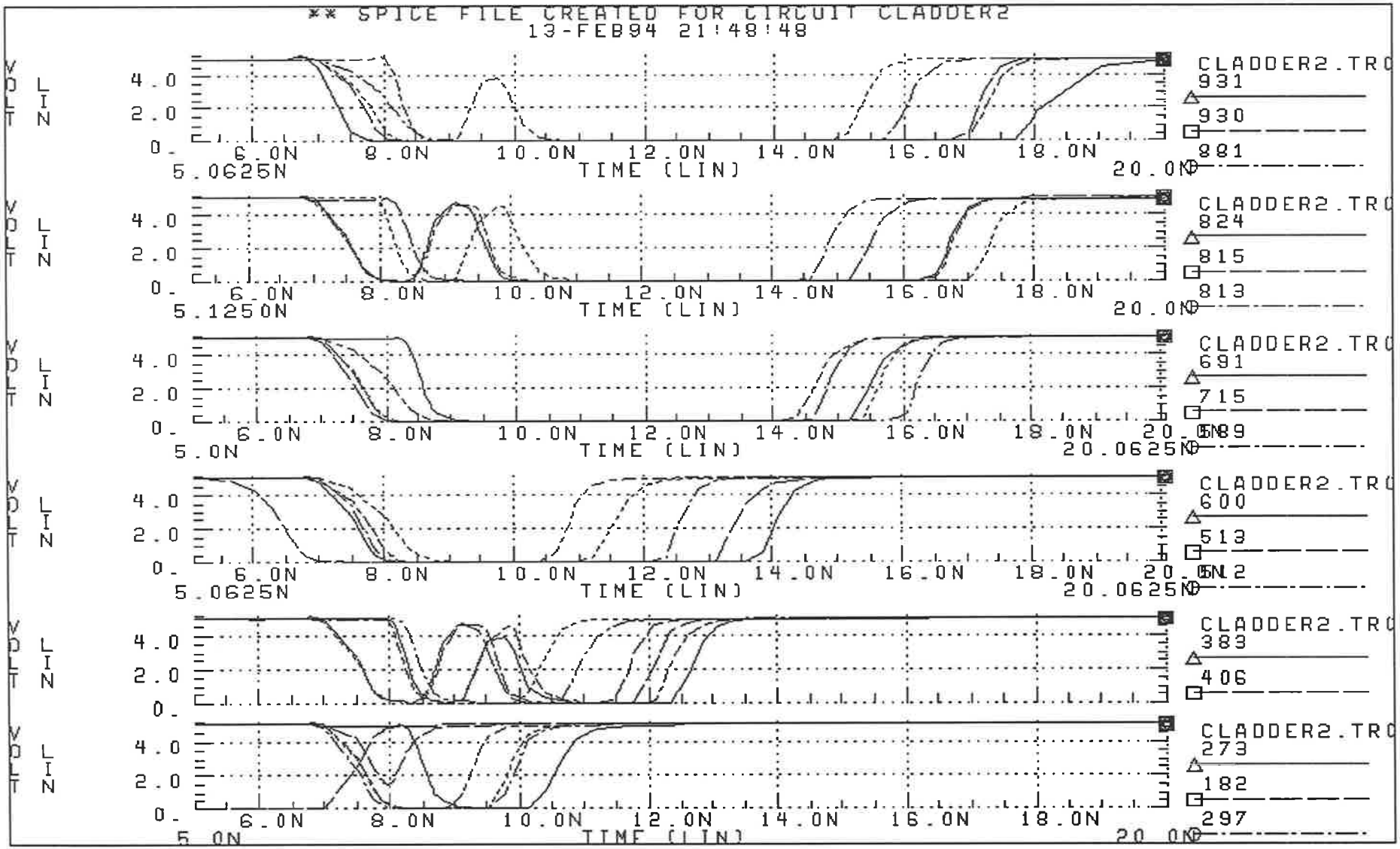



**Figure 4.14--Logic levels of signal $S_{31}$ to $S_0$ are shown from top to bottom respectively.**

third number $Q$, called the quotient, is to be calculated in such a way that

$$D = Q \times V + R$$

In the above equation, $R$ is the remainder and required to be of smaller magnitude of $V$, i.e., $0 \leq |R| < V$. Division circuits are usually designed to compute the quotient $Q$ of some specific length. In case of this study, only the quotient $Q$ is required and the remainder $R$ is discarded. In division algorithms, usually dividend $D$ is considered to be the first partial remainder and in our case it is always smaller than the divisor $V$. To get the required accuracy, it is therefore necessary to pad the dividend by a number of zeros to the right. Generally speaking, the final remainder $R$ may be used to generate additional quotient digits by second division operation of $R$ by $V$. In this way successive division instructions can be used to generate a more accurate result.

### 4.3.1. Restoring Binary Division [30]

One of the simplest methods for division is the sequential shift and subtract method which is similar to the conventional pencil-and-paper technique. To explain this approach, and for simplicity, let's suppose that the dividend $D$ and the divisor $V$ are positive integer binary numbers. The quotient bits $q_{n-1}, \cdots, q_1, q_0$ are computed one bit at a time. At each step, the divisor $V$ is shifted one bit to the right and is compared with the dividend $D$ or the current partial remainder $R_i$. If the shifted divisor is less than or equal to the partial remainder, then the new quotient bit is set to 1; otherwise the new quotient bit will be zero. The following partial remainder can be calculated using the relation

$$R_{i+1} \leftarrow R_i - q_i 2^{-i} V$$

where $i$ is the iteration number. In hardware implementation it is usually more convenient to

shift the partial remainder to the left rather than shifting the divisor to the right. Furthermore, the new quotient bit $q_i$ is determined by subtracting $V$ from $2R_i$. If the result is negative then $q_i = 0$; otherwise $q_i = 1$. It should be noted that whenever $q_i = 0$, then the result of the subtraction is $2R_i - V$, however the new partial remainder $R_{i+1}$ must be equal to $2R_i$. It is therefore necessary to add the divisor $V$ to the result of subtraction in order to restore the result back to its original value $2R_i$.

### 4.3.2 Non-restoring Binary Division [30]

In restoring algorithm, the machine may need up to $2n-1$ cycle to compute all the quotient bits, i.e., there are $n$ cycles for the trial subtractions, and there may be additional $n-1$ cycle for the restoration. Non-restoring division is a faster algorithm which effectively eliminates the restoring phase. To explain this method, consider that in restoring method in every step the operation

$$R_{i+1} \leftarrow 2R_i - V$$

is performed. When the result of this subtraction is negative, a restoring addition is performed as follows

$$R_{i+1} \leftarrow R_{i+1} + V \text{ therefore } \Rightarrow R_{i+1} = 2R_i$$

To calculate the partial remainder $R_{i+2}$, again we have

$$R_{i+2} \leftarrow (2R_{i+1} - V) = (4R_i - V)$$

However the non-restoring algorithm suggests that before the restoring addition is performed the resultant partial remainder be shifted and then the restoring addition takes place, that is

$$R_{i+1} = 2R_i - V$$

and

$$R_{i+2} \leftarrow [2(2R_i - V) + V] = 4R_i - V$$

Apparently the result of non-restoring method is exactly the same as restoring algorithm with the difference that one subtraction operation is saved, and hence results in a faster algorithm.

Similar to restoring method, in non-restoring algorithm the quotient bit $q_i$ at each step determines the next operation to be performed. However, unlike restoring method, in non-restoring algorithm the quotient digits are selected from the set $\{-1, 1\}$ [39]. If $q_i = 1$ then the next operation will be subtraction; otherwise the next operation is addition. Once the quotient bits $q_i \quad i = 1, \ldots, n-1$ are computed, the result of the division operation can be represented in signed digit format as following

$$Q = \sum_{i=0}^{n-1} q_i \cdot 2^i \qquad q_i = \{1, -1\} \qquad \text{(EQ 20)}$$

Based on the fact that $-1$ can not be represented in the binary system used by processors, the following scheme is adopted

$$(q_i = -1) \Rightarrow 0$$

$$(q_i = 1) \Rightarrow 1$$

However this arises another problem since the result will be interpreted as a different number by a standard binary machine. Fortunately, the signed digit numbers generated by non-restoring algorithm can be converted to standard binary format by using a very simple algorithm and considering that

$$1 \cdot 2^n - 1 \cdot 2^{n-1} = 0 \cdot 2^n + 1 \cdot 2^{n-1}$$

Based on this algorithm the next three steps should be followed

1. Shift the result left one bit position.
2. Complement the most significant bit (MSB) of the result.
3. Set a 1 into the least significant bit (LSB) of the result.

If the ith bit of the extracted number $K$ is represented as $K_i$, then there is a basic relationship between $K_i$ and $q_i$ as

$$\begin{aligned} K_0 &= 1 && K_i = \{0, 1\} \\ q_i &= 2K_{i+1} - 1 \quad \text{if} && i = 0, \ldots, n-2 \\ q_i &= 1 - 2K_{i+1} \quad \text{if} && i = n-1 \end{aligned} \qquad \text{(EQ 21)}$$

It is important to note that in practice the bit length of $K$ and $Q$ must be the same. This implies that the shift operation of step one will cause the MSB of the result to be lost, therefore step two is redundant. It still gives the correct result if

$$K_n = K_{n-1}$$

otherwise an overflow signal detects the case. In order to prove that the algorithm is valid for two's complement numbers, we have to show that $K$ is in fact the two's complement representation of the quotient $Q$. If we replace $q_i$ in (EQ 20) by its equivalent value in (EQ 21) we get

$$Q = \sum_{i=0}^{n-1} q_i \cdot 2^i = (1 - 2K_n) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (2K_{i+1} - 1) \cdot 2^i$$

$$= 2^{n-1} - 2^n \cdot K_n + \sum_{i=0}^{n-2} 2K_{i+1} \cdot 2^i - \sum_{i=0}^{n-2} 2^i$$

Since $$2^{n-1} - \sum_{i=0}^{n-2} 2^i = 1$$

then $$Q = -2^n \cdot K_n + \sum_{i=0}^{n-2} K_{i+1} \cdot 2^{i+1} + 1$$

or $$Q = -2^n \cdot K_n + \sum_{i=1}^{n-1} K_i \cdot 2^i + K_0$$

The last equation shows that the binary two's complement number $K$ is equal to the quotient $Q$. The basic rule in calculation of quotient bits $q_i$ is to bring the partial remainder as close to zero as possible. If the dividend is considered to be the first partial remainder, then $q_i$ can be determined by using table five.

**TABLE 5. Quotient bits determination**

| Sign of Partial Remainder | Sign of Divisor | Next Operation | $q_i$ |
|---|---|---|---|
| + | + | Subtraction | 1 |
| + | - | Addition | 0 |
| - | + | Addition | 0 |
| - | - | Subtraction | 1 |

Table five [39] implies that the next quotient bit can be generated by following relation

$$q_{i-1} = (\text{SIGN OF DIVISOR}) \oplus (\overline{\text{SIGN OF } R_i})$$

where $R_i$ is the current partial remainder.

### 4.3.3 Hardware Implementation

The block diagram of the 32 by 16-bit divider is shown in Fig. 4.15. The 32-bit carry look ahead adder/subtracter designed for the multiplier is modified so that this block can be shared between multiplier and divider. The employed multiplexer enables the system to use the adder as a 32-bit or 16-bit adder/subtracter, or as part of the divider in which case the 16-bit adder is governed by the control circuitry of the divider that determines whether an addition or a subtraction is to be performed.

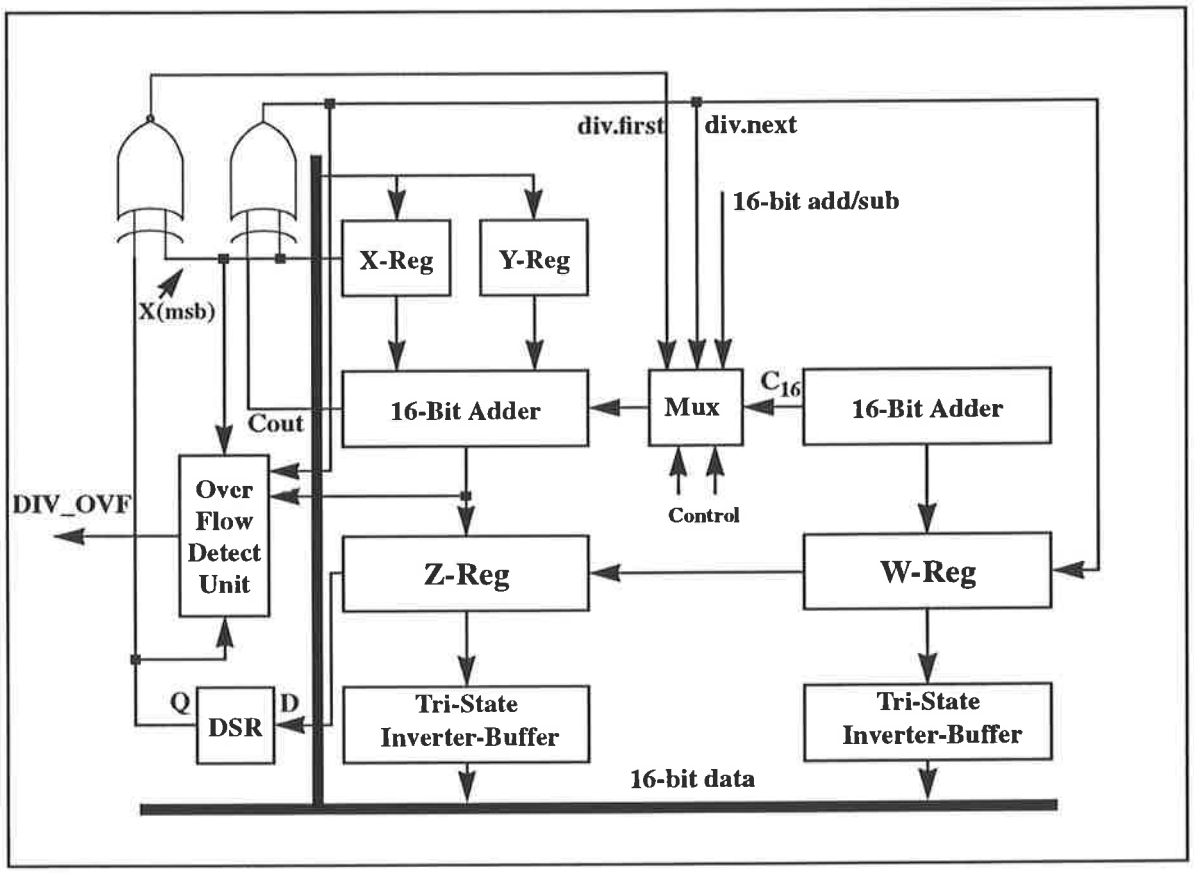



**Figure 4.15--Block diagram of 32 by 16-bit non-restoring divider**

At the beginning of the division operation the 32-bit dividend is stored in accumulator which is composed of registers Z and W. The most significant portion of the dividend is in the Z register, and the least significant portion is in the W register. Register X is loaded with 16-bit divisor and the quotient bits $q_i$ are generated one at a time and are left shifted into register W. It should be noted that the output carry of add/subtract unit is used to generate the

next quotient bit. It is legitimate because in two's complement addition/subtraction the carry out is always of opposite sign to the sign bit of the result, provided that no overflow has occurred. Therefore it is possible to merge the left shift operation of the dividend and the left shift operation of the new quotient bit into one shift operation. The result of division is always of 16-bit length and will be stored in W register. The flow diagram of the algorithm is shown in Fig. 4.16 [14]. The microprogram of the non-restoring division is given in Appendix B.

### 4.3.4 Overflow Detection [39]

For our 16-bit processor, the dividend is 32 bits, and the divisor is 16 bits. The quotient $Q$ must be at most of 16 bits length [19], otherwise an overflow flag will be set which shows a division overflow has occurred. Using two's complement representation for the quotient $Q$, the below restrictions should be imposed in order to avoid overflow.

$$|Q| < 2^{15} \quad \text{for positive quotients,}$$

$$|Q| \leq 2^{15} \quad \text{for negative quotients,}$$

replacing $Q$ by its equivalent value

$$|Q| = \left|\frac{Z \cdot 2^{16} + W}{X}\right| < 2^{15}$$

$$\Rightarrow |Z \cdot 2^{16} + W| < 2^{15} \cdot |X| \Rightarrow \left|Z \cdot 2 + \frac{W}{2^{15}}\right| < |X|$$

where $W$ is the least significant portion of the dividend and is always positive. Considering that in fixed point division the remainder $R$ must satisfy the following relation

$$0 \leq |R| < divisor$$

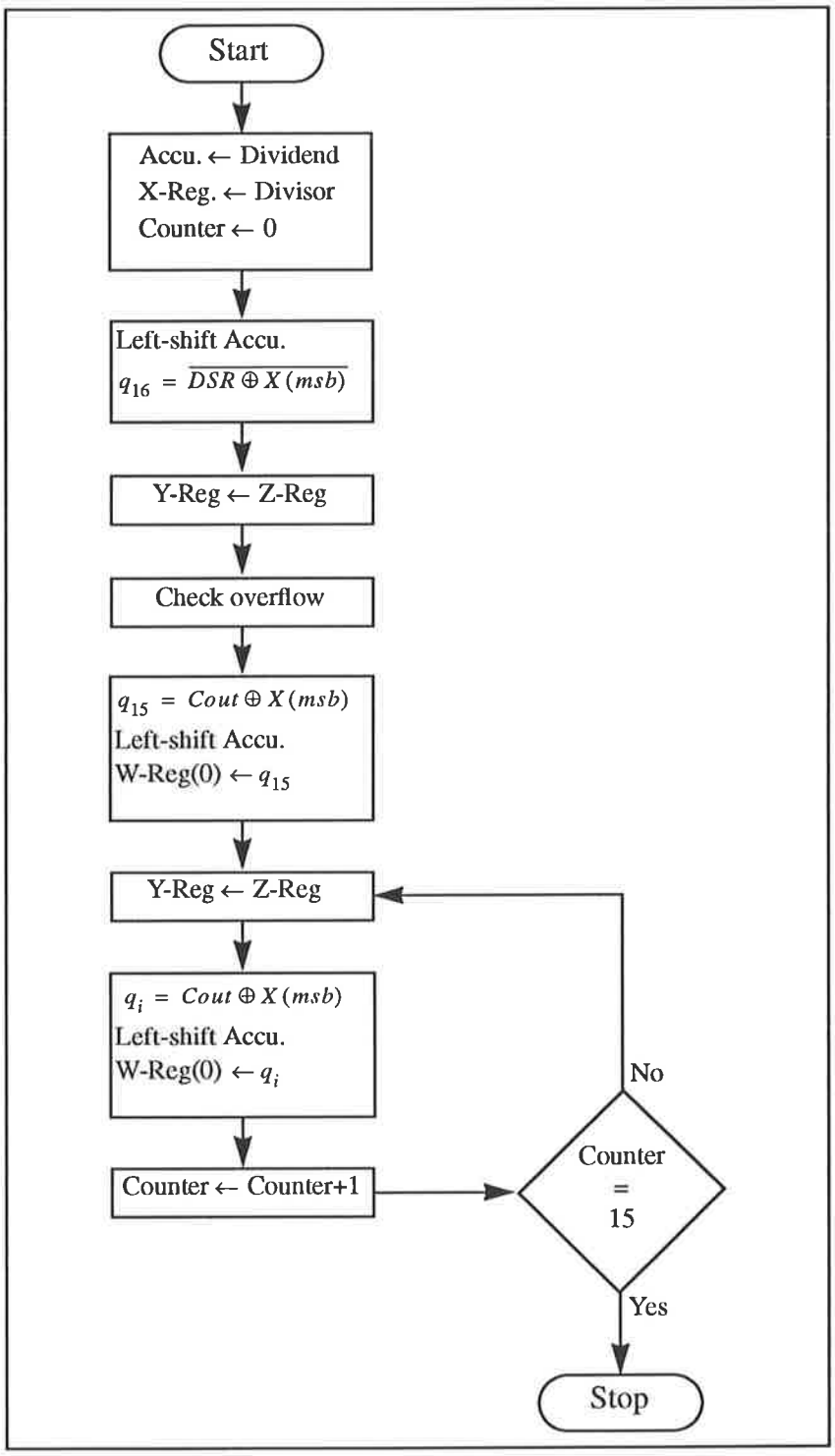



**Figure 4.16--Non-restoring division algorithm for two's complement numbers.**

the following two conditions can be discerned

$$\frac{W}{2^{15}} = 1 \quad \text{if} \quad W(msb) = 1$$

$$\frac{W}{2^{15}} = 0 \quad \text{if} \quad W(msb) = 0$$

or in general $$\frac{W}{2^{15}} = W(msb)$$

therefore $$\left|2 \cdot Z + \frac{W}{2^{15}}\right| = |2 \cdot Z + W(msb)|$$

The right hand side of the above equation represents the content of the Z register when left shifted with the most significant bit of the W register. The above analysis implies that the conditions to avoid overflow can be expressed as

$$|X| > |2 \cdot Z + W(msb)| \quad \text{for positive quotients,}$$

$$|X| \geq |2 \cdot Z + W(msb)| \quad \text{for negative quotients,}$$

It is therefore possible to detect the division overflow right after the first cycle of shift and add/subtract operations.

The sign of the quotient is depicted in table 6 as a function of the dividend and the divisor signs and the relative magnitude of Z and X. The overflow conditions are identified by numbers in square brackets. The zero flag of the processor is used to detect the condition where $|X| = |Z|$. The zero flag is logic one whenever the result of any 16-bit add/subtract operation is zero. The division overflow signal (DIV_OVF) is therefore a function of signals DSR, X(msb), Q(msb), and ZF (Zero Flag). The truth table for DIV_OVF signal is based on table 6 and is depicted in table 7.

**TABLE 6. Division overflow detection.**

| Case | DSR | X(msb) | Operation | Q(msb) | | |
|---|---|---|---|---|---|---|
| | | | | $\|X\| > \|Z\|$ | $\|X\| < \|Z\|$ | $\|X\| = \|Z\|$ |
| 1 | 0 | 0 | Z-X | 0 | [1] | [1] |
| 2 | 1 | 0 | Z+X | 1 | [0] | 1 |
| 3 | 0 | 1 | Z+X | 1 | [0] | [0] |
| 4 | 1 | 1 | Z-X | 0 | [1] | [0] |

**TABLE 7. Truth table for division overflow signal DIV_OVF.**

| ZF | DSR | X(msb) | Q(msb) | DIV_OVF |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | don't care |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | don't care |
| 1 | 1 | 0 | 0 | don't care |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | don't care |

A four variable Karnaugh map of table 7 gives the following simplified expression for the DIV_OVF signal.

$$DIV\_OVF = Z \cdot \overline{DSR} + Z \cdot X(msb) + DSR \cdot X(msb) \cdot Q(msb) + DSR \cdot \overline{X(msb)} \cdot \overline{Q(msb)} + \overline{DSR} \cdot X(msb) \cdot \overline{Q(msb)} + \overline{DSR} \cdot \overline{X(msb)} \cdot Q(msb)$$

The circuit diagram of division overflow detector is displayed in Fig. 4.17. The logic of this circuit is further simplified by using a logic optimiser software.

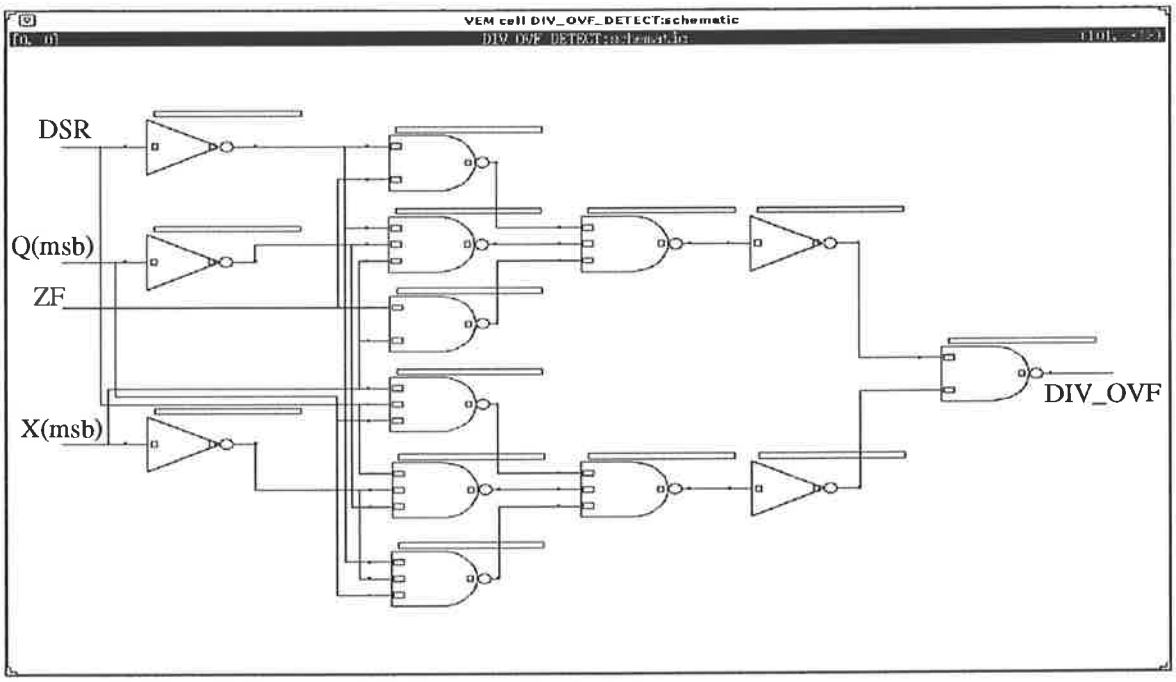



**Figure 4.17--Circuit diagram of division overflow detector.**

### 4.3.5 Divider Simulation Result

Fig. 4.18 shows the actual simulation of the processor during the analysis of a sampled data. In this picture only part of the simulation is depicted in which a division operation is involved. The processor is computing one of the predictor filter coefficients by dividing the variable "num" by the variable "den" (see Fig. 2.1). As usual the variable "num" is of smaller magnitude than the variable "den", therefore its value is multiplied by 4000 (Hex.) and the result is placed in the accumulator. This way the dividend is padded by 14 zeros to the right, and therefore the resultant quotient in register W has its radix point after digit 14.

The variable "num" is originally equal to BF6 (Hex.), and when padded with additional zeros to the right it forms the actual dividend of this division operation, which is equal to 2FD8000 (Hex.). The most significant portion of the dividend in the Z register is FD02 (Hex.), and the least significant portion in the W register is 7FFF (Hex.). Note that the processor stores the complement of the dividend into the accumulator which in this case is equal to FD027FFF (Hex.). The variable "den" is the divisor of this simulation. It resides in the X register and its value is equal to 24F3 (Hex.), which as depicted in Fig. 4.18 remains unchanged during the execution of the division routine. The result of this simulation shows that each division routine demands 100 clock cycle to generate the final result and return it to the main routine. The quotient Q is stored in register W, which in this case is equal to

$$Q=0.01010010110111 \text{ (Bin.)} =0.323669433$$

The accuracy of this division operation is equal to $\pm\frac{1}{2^{14}} = \pm 0.00006$ which is reasonably enough for the application of this study. Note that the division overflow signal is checked by the control unit only after the first shift and add/subtract operation.

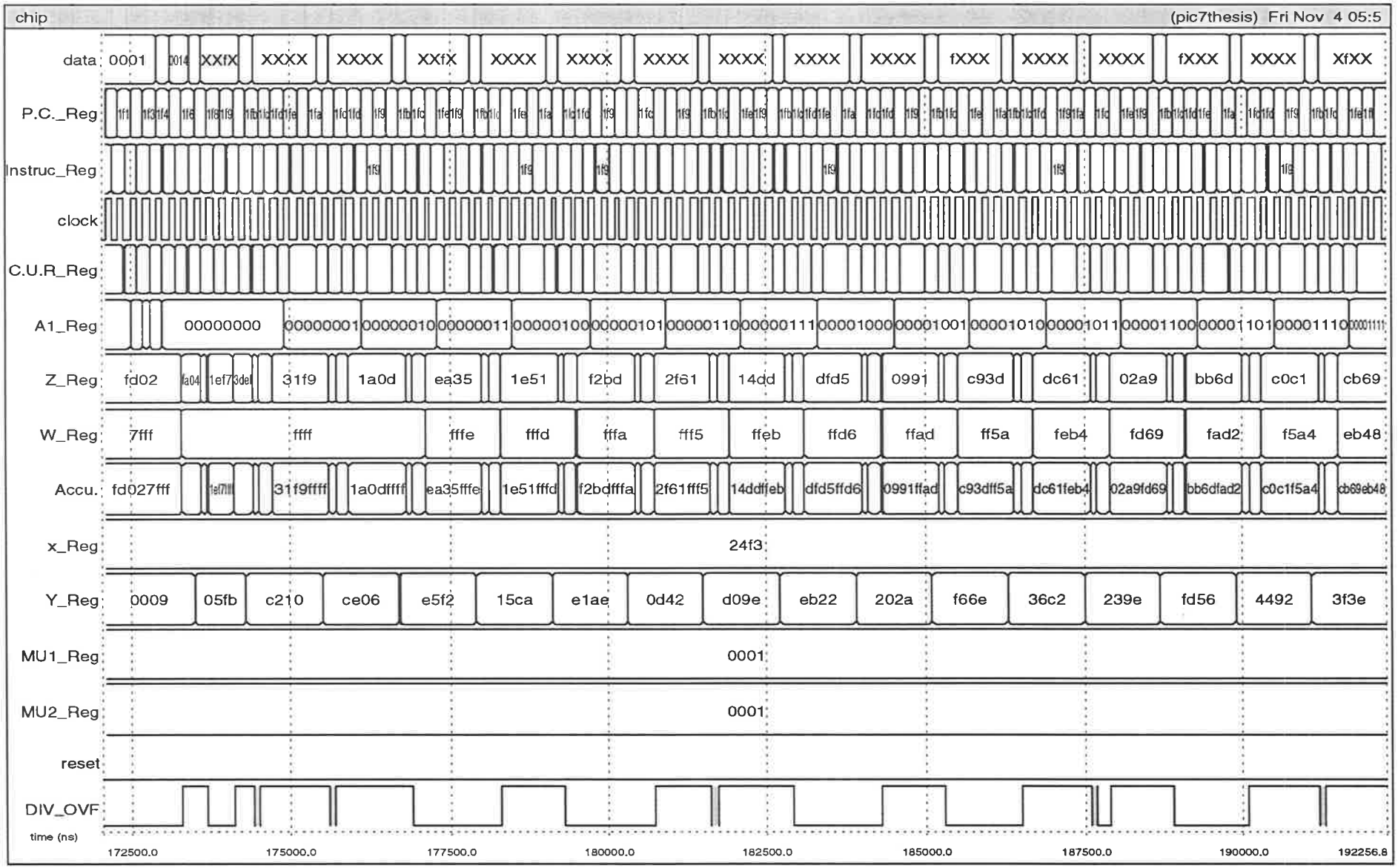



**Figure 4.18--Simulation result for division operation.**

## 4.4 Control Unit

The circuit design of a processor can be divided into two separate parts. The first part is the design of the digital circuits that perform the data processing operations. The data processing unit is a network of functional units capable of performing certain operations on data. The second part is pertained to the design of the control circuits. The function of the control section is to regulate the operation of the processor. It decodes the instructions and causes the proper events to occur in the correct order. The control section of a processor consists of a group of registers and Flip Flops and the timing circuitry necessary to make them operate properly.

There are two kinds of informations stored in the processor; data and control information. Control information supplies command signals that administer the operations in the data processing section to execute the required task. The main clock signal is applied to all registers and Flip Flops in the processor, but their status remain unchanged unless they are enabled by a control signal. The signals that control the load input of the registers or enable them to write on the data bus and also the select input of the multiplexers are generated in the control subsystem. The interaction between the control unit and the data processing section is shown in Fig. 4.19 [18].

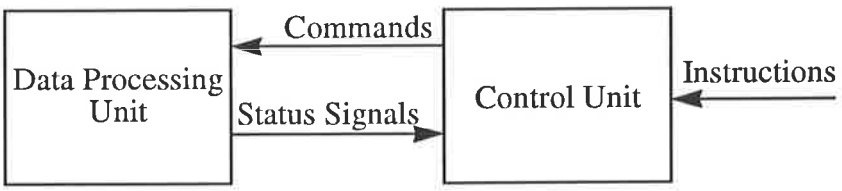



**Figure 4.19--Relationship between control unit and data processor**

The control logic is primarily a sequential circuit with both internal and external status signals. It uses the informations which it receives from the data processor to determine the flow

of the controlled operations. At any given time, the state of the sequential circuit determines a set of commands to be executed. Depending on the status signals, the sequential control goes to the next state to complete the current instruction or initiates the next operation.

### 4.4.1 Design Methodologies

Historically, two general approaches to control unit design have been developed [14]. The first of these regards the control unit as a sequential logic circuit to generate specific fixed sequences of control signals. In this approach the control unit is implemented with gates and Flip Flops. As such, the main goal is to minimize the number of components used and to maximize their speed of operation. It has the advantage that it can be optimized to produce a fast mode of operation. The disadvantage however is that once constructed, any change in the operation of the circuit can be applied only through the redesigning and rewiring of the unit. This approach is therefore known as hardwired organization of the control unit.

In contrast to the hardwired approach, the second method of control unit design is called the microprogrammed regulation. The sequence of instructions required to perform a specific operation comprises a microprogram for that particular operation. The control signals of this microprogram are stored in the form of 0's and 1's in a special memory. In microprogrammed control, since the signals are implemented in software rather than the hardware, any change in the design can be made by modifying the contents of the control memory. The disadvantage nevertheless is that microprogram control units are slower because of the extra time required to fetch the microinstructions from the control memory.

Generally speaking, the design of the control unit can be done in a more systematic way by using the microprogrammed approach. The control signals can be organized into words (microinstructions) that have a well defined format. However, in VLSI design, the availabil-

ity of the automatic placement and routing softwares as well as the logic synthesizers have made the hardwired control unit design quite flexible. A high level language can be used to efficiently define the behaviour of the system and any further modification is a matter of software revision and recompilation rather than redesigning and rewiring of the hardware. It is therefore the hardwired design approach which was chosen for the design of the control unit.

### 4.4.2 Hardware Implementation

The block diagram of Fig. 4.20 shows the pipelined architecture of the control unit. The machine codes of the instructions in the instruction register (IR) are interpreted by the decoder under the supervision of the timing and control section. If the instruction requires only one cycle for its execution, then the decoder generates the required control signals which are transferred to the control unit register (C.U.R.). In almost all of the two cycle instructions the processor unit must execute the no operation command (NOP). These instructions are recognized by the timing and control section and force the decoder to set the register C.U.R. with no operation command variables. During the time that the processor unit is executing one instruction, the decoder is translating the next instruction. As depicted in Fig. 4.20, there are several control signals used by the control unit that manipulate the flow of the stored program. The first group of these signals are initiated in the processor unit and enable the processed data to affect the control unit, allowing data-dependent decisions to be made. These signals are comprised of "ADD_OVF", "DIV_OVF", "BG", data lines, and RAM address lines of register A1. The status of the flag signals "Flag1" to "Flag6" and "Jump" are functions of the content of the instruction register. The signals "Flag5", "Flag6" and "Jump" are particularly responsible for providing additional cycles to complete the execution of two cycle instructions.

Along with the other flags, the following registers are part of the control unit and provide

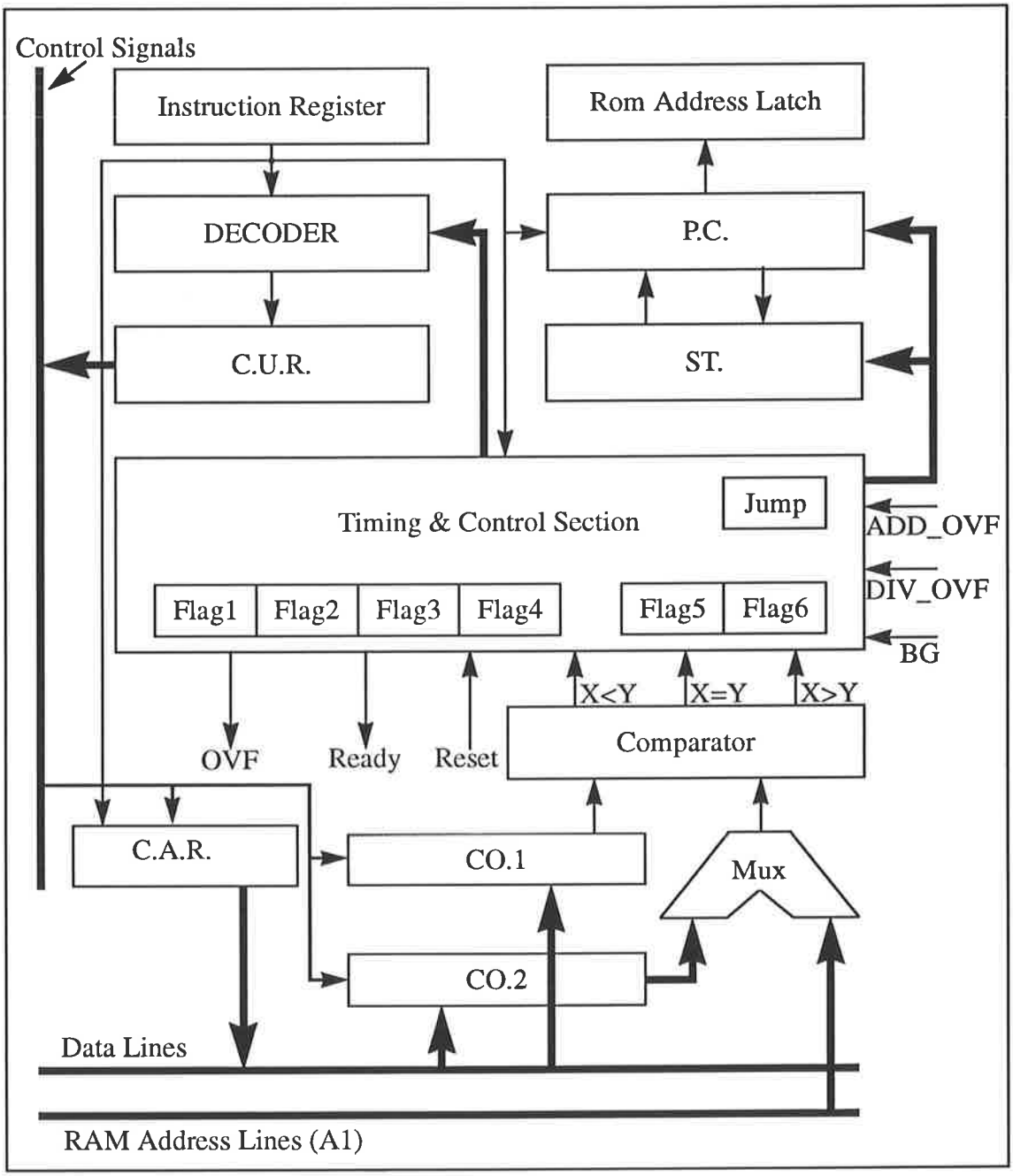



**Figure 4.20--Block diagram of the control unit**

additional facilities in instruction sequencing

1. **The program counter (P.C.).** This is a register that holds the memory address of the next instruction word to be executed.

2. **The Stack Pointer (ST.).** This register stores the return address while the processor is executing a subroutine.

3. **Control Address Register (C.A.R.).** This register provides the required capability for immediate addressing. It holds the second byte of the immediate address instruction to be transferred to the memory address register A1.

4. **Registers CO.1 and CO.2.** These registers are primarily used for data comparison and decision making based on the acquired result, utilizing the built in comparator of the control unit.

### 4.4.3 Control Unit Specifications

The two formal tools for describing the behaviour of the control unit are flowcharts and description languages [14]. The BDS language explanation of the control unit is given in Appendix C. It provides a combinational logic implementation of the control unit, which can be used to compose the required sequential element by providing the necessary feedback paths and input/output registers. In the following sections the flowchart description of the control unit provide an in-depth perception of the operation of this block.

- **ROM Address Generation**

Most of the instructions in our program have a unique successor, in which case it is natural to store the next instruction in the next memory address. Hence the program counter (P.C.) is simply increased by one to produce the address of the next instruction to be executed. However, it is sometimes necessary to select one of the several possible actions or to repeat a set of instructions for a specific number of times. In such circumstances the current instruction specifies the address of the next instruction and alters the flow of the program control by moving the address of the next instruction into the program counter (P.C.). The address of the next instruction is stored as the second byte of the current instruction. Therefore all the conditional and unconditional branch and subroutine call instructions of the processor are two

cycle instructions. The flow diagram of Fig. 4.21 indicates how the instruction register (IR) and the "jump" signal affect the contents of the program counter (P.C.).

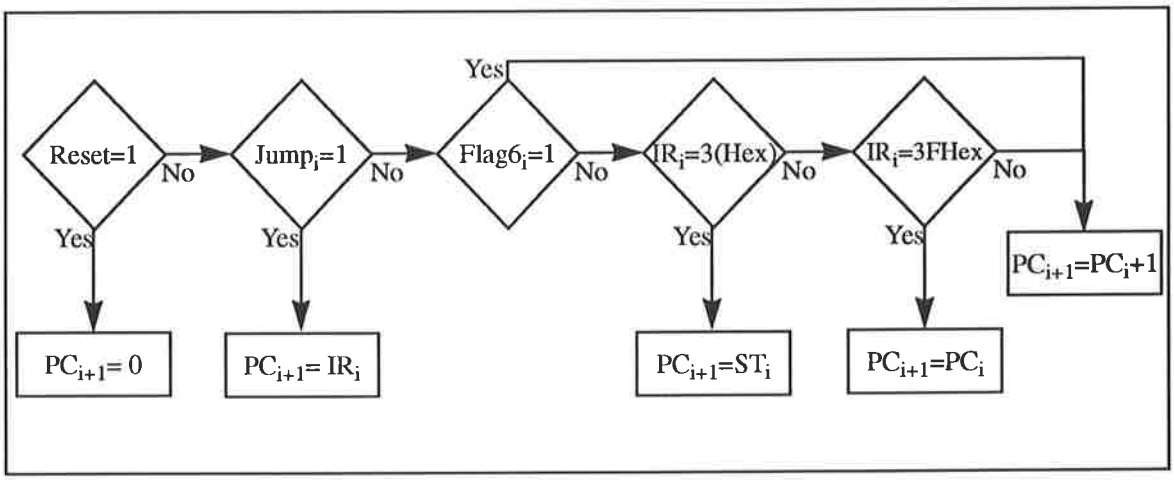



**Figure 4.21--Program Counter Routine**

The status of the signal "Flag6" indicates whether the content of the instruction register is an address or a command. If "Flag6" is logic one, then IR represents an address and NOP instruction is performed by the processor unit. If "Flag6" is logic 0, then the content of IR is regarded as an instruction which alters the contents of the control unit register (C.U.R.). The flow diagram of Fig. 4.22 shows which commands influence the status of the signal Flag6.

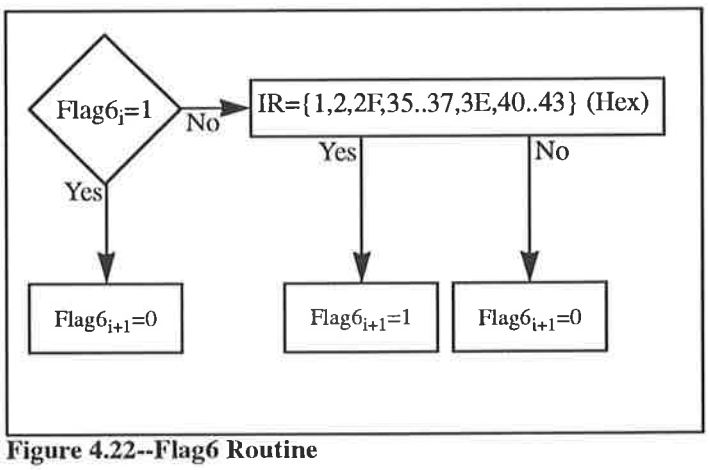



**Figure 4.22--Flag6 Routine**

The "Jump" signal is a flag in the control unit that determines whether the required condition for the Branch instructions is satisfied. If "Jump" is logic one, then the content of IR is the address of the next instruction to be executed. The flow diagram of Fig. 4.23 depicts the

"Jump" routine.

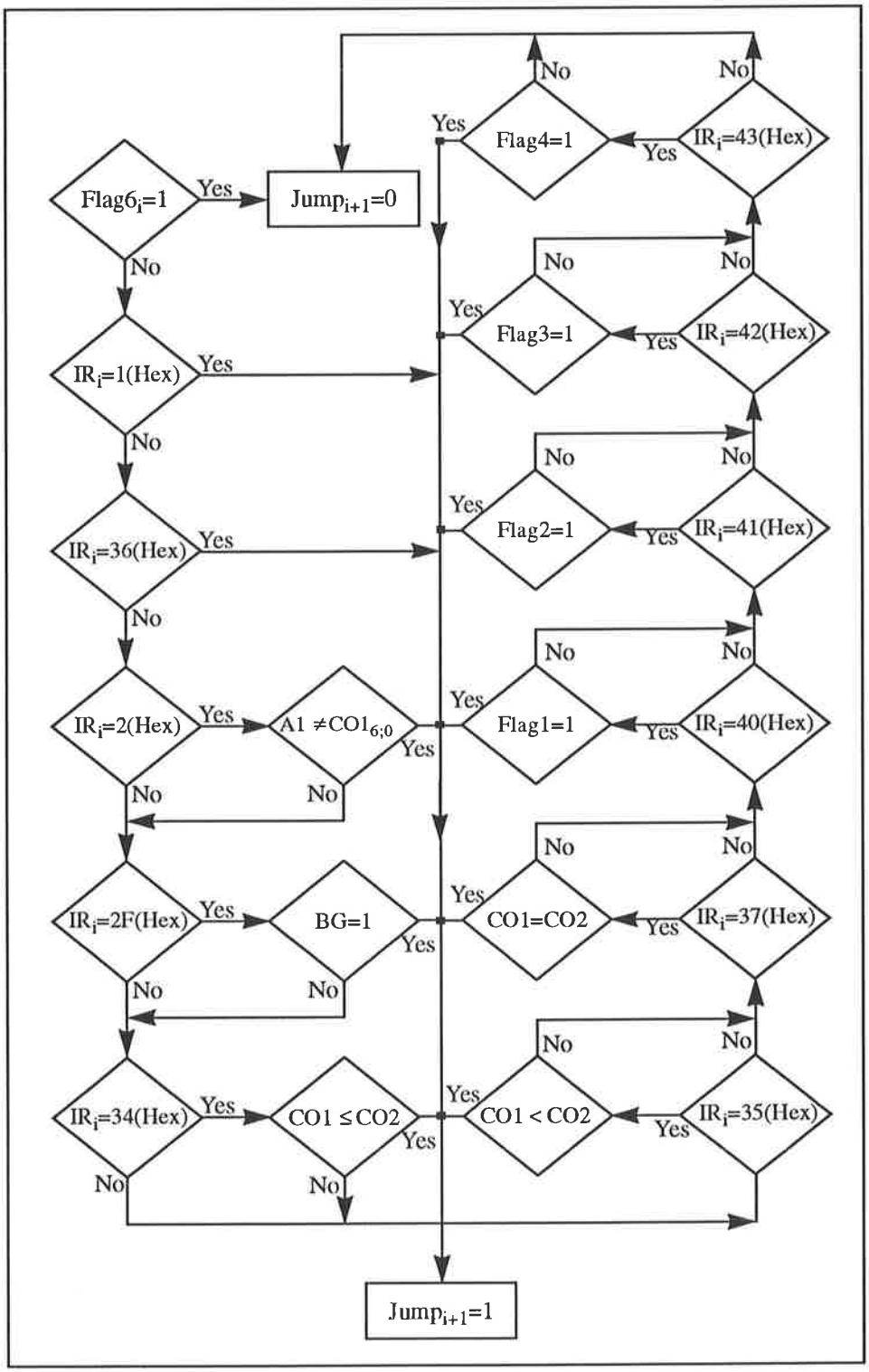
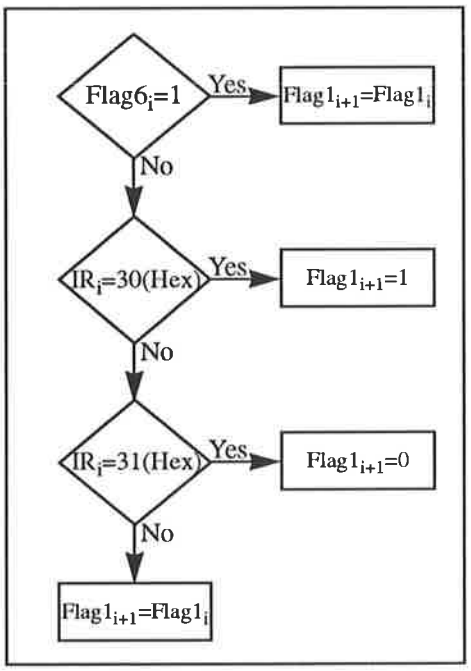


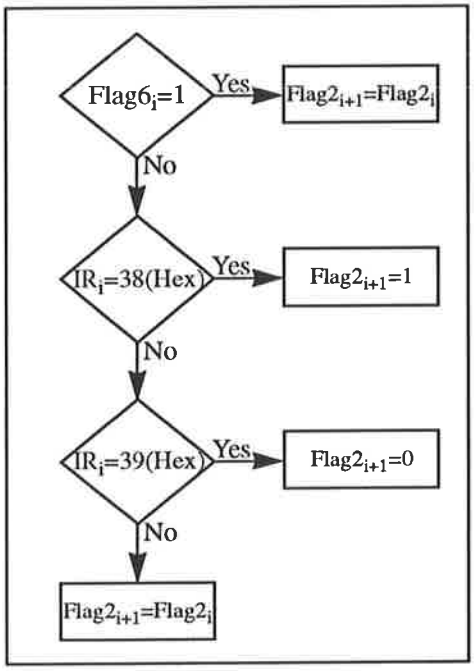

**Figure 4.23--Jump Routine**

The flag signals "Flag1" to "Flag4" can be set or reset according to any arbitrary condition. They are primarily introduced for the order checking algorithm. These flags can be subsequently used to alter the flow of the program control according to the flow diagrams of Fig.

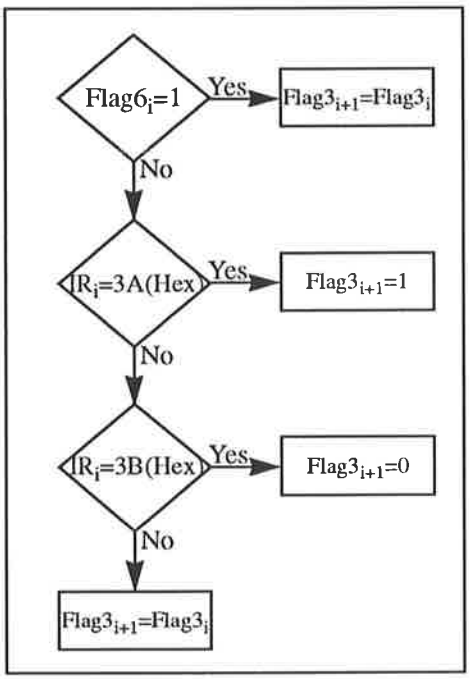
4.24.



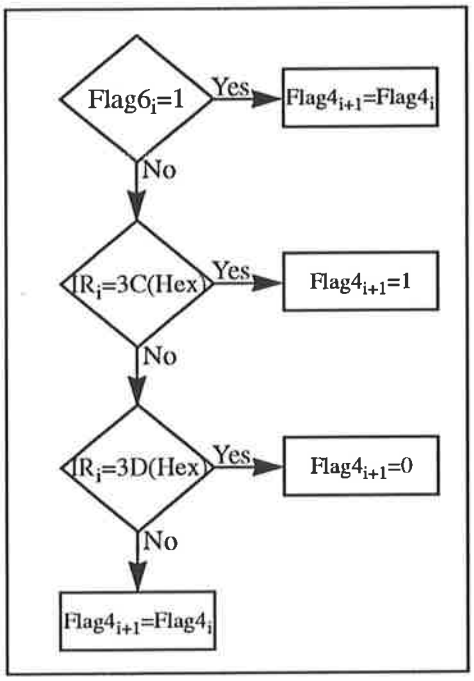

**(a)Flag1 Routine**




**(b)Flag2 Routine**




**(c)Flag3 Routine**




**(d)Flag4 Routine**

**Figure 4.24--Routine (a)Flag1 (b)Flag2 (c)Flag3 (d)Flag4**

- **Subroutine Call**

Often it is necessary to implement a temporary transfer of control from the main program to

a subprogram. This control transfer is initiated by the main program and is known as subroutine call. In order for the control to be transferred back to the main program, the address of the next instruction of the main program is stored in the register (ST) of the control unit. The last instruction of the subprogram should transfer the content of (ST) back to (P.C.) (see Fig. 4.21). The flow diagram of Fig. 4.25 shows how the subroutine call instruction affects the register (ST).

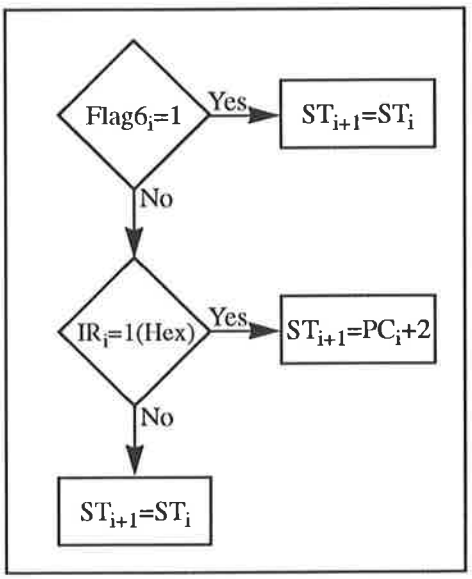



**Figure 4.25--Stack Pointer Routine**

## • Immediate Addressing

The content of the instruction register (IR) can be interpreted either as an instruction to be executed by the processor unit or an address. If "Flag6" is logic one, then IR is an address to be transferred either to program counter (P.C.) or to the memory address register (A1). Furthermore, the status of "Flag5" determines whether IR contains the second byte of the "Immediate Address" instruction. If "Flag5" is logic one, then the content of IR is transferred to register C.A.R. and subsequently to the address register (A1). The flow diagram of Fig. 4.26 describes how these operations are performed.

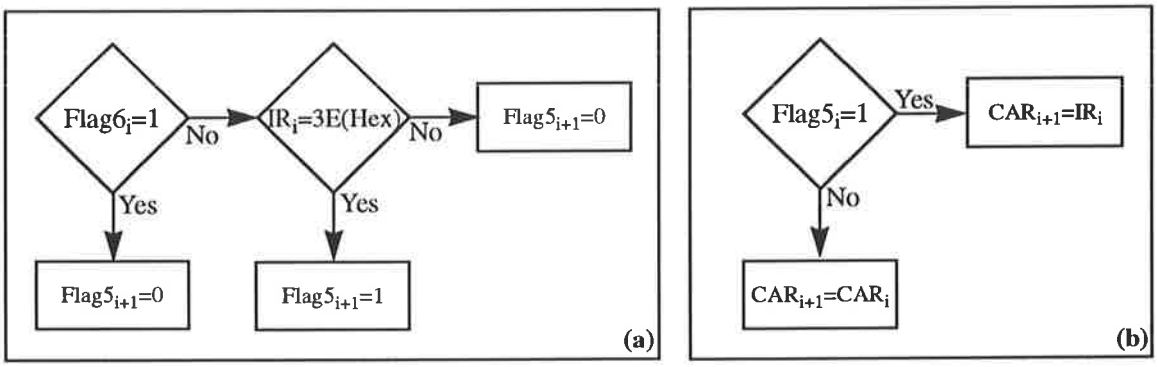



**Figure 4.26--(a)Flag5 Routine (b) Immediate Address Routine**

There are a number of occasions in which the data processing unit must perform the no operation command (NOP). These situations are depicted in the flow diagram of Fig. 4.27 along with the influence of status flags "Flag5" and "Flag6" on the control unit register (C.U.R.).

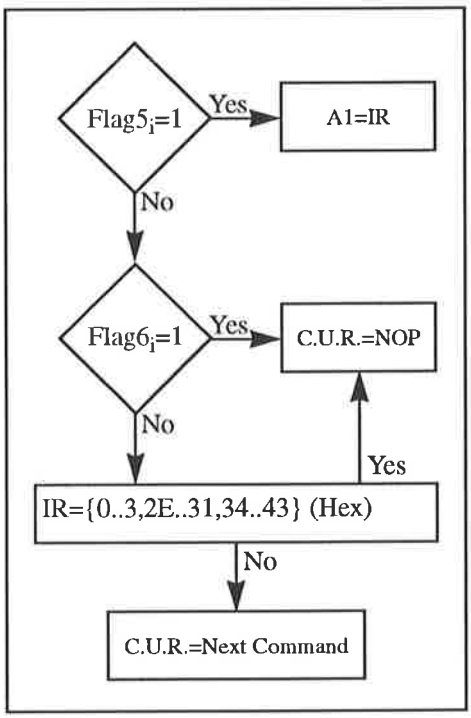



**Figure 4.27--Next Command Routine**

- **Processor Status Signals**

The outgoing signals "Ready" and "OVF" indicate the status of the processor to the outside world. The "Reset" line is a signal which is received from a supervisory controller and can be used to synchronize the operation of the processor with other devices of the system. The flow diagrams of Fig. 4.28 shows how these signals are manipulated by the content of the instruction register and the computational results of data processing unit.

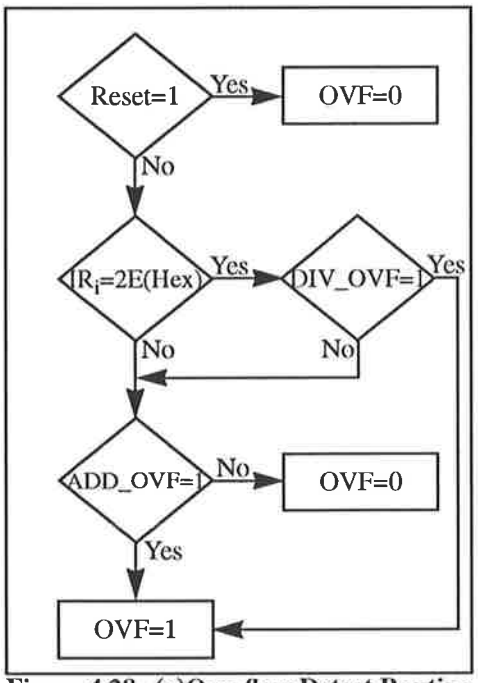



Figure 4.28--(a)Overflow Detect Routine

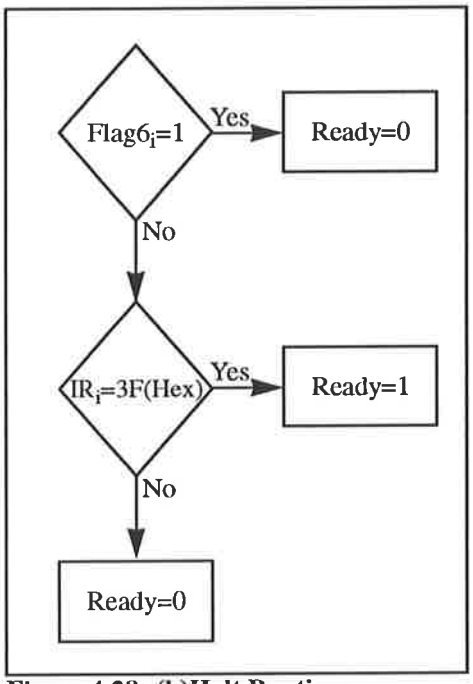



Figure 4.28--(b)Halt Routine

## 4.5 Random Access Memory Design

Random access memories are storage elements in which the access time is independent of the physical location of data. They can be classified into static and dynamic [27] groups depend on the structure used in the design of the memory cells [20]. Both types of memory may be further divided into synchronous and asynchronous categories. Synchronous RAMs are those which need a clock signal for their operation. However, asynchronous structures reflect upon the change of the logic level of their address lines. Due to its design simplicity and reliability, the static synchronous structure is the most commonly used technique in the

design of random access memories and also the one which was adopted for this study.

### 4.5.1 Hardware Description

The RAM of the processor is capable to accommodate up to 256 words of 16-bit length data. Figure 4.29 shows the transistor level circuit diagram of a typical RAM cell [42] together with the required supporting circuits in block diagram form. The floor plan of the RAM is depicted in Fig. 4.30. The RAM has separate data inputs and outputs which are tied together by using tri-state output signals in order to realize a single data bus arrangement. The column decoder is a one of four decoder which controls the operation of multiplexers. Multiplexers are designed by using n-channel pass transistors, therefore they can operate both as multiplexer and demultiplexer. At any time, one of $2^6 = 64$ rows can be selected for either read or write operation through the row decoder. The least significant bits $a_0$ and $a_1$ of the address lines determine the selected column and the rest most significant bit lines, $a_2$ to $a_7$, are used by the row decoder to access the determined row. Associated with the row decoder are row drivers. They are designed in such a way that each block of row driver buffers two outputs of the row decoder. The precharge circuits on the top prepare the bit lines of the RAM cell at the beginning of each read or write cycle by pulling them up to logic one. Each read or write operation requires only one clock cycle to complete, during which the bit lines normally run as complementary signals. The transistor circuit diagram of the RAM cell in Fig 4.29 forms two cross-coupled inverters that are accessible via two n-channel pass transistors [37]. Sense amplifiers reflect upon small changes of the voltage on bit lines that results when a particular cell is selected for read operation, and therefore speed up the read operation of the RAM. The control section of the RAM regulates all the operations within the RAM by generating the required control signals and using the following inputs which are accessible to the user.

1. The "CS" signal enables the RAM for active operation when it is logic one, otherwise the

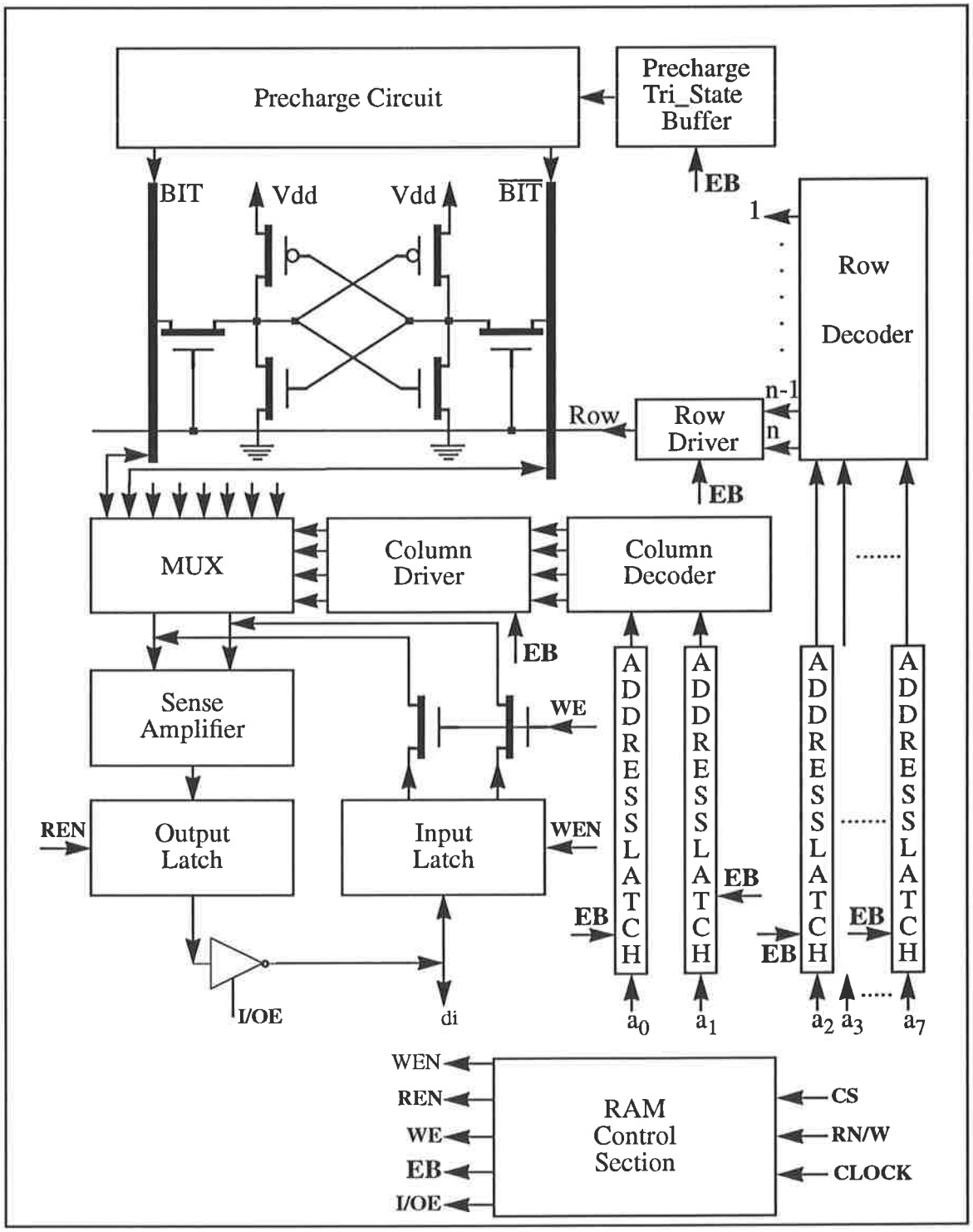



**Figure 4.29--Block diagram and supporting circuits for a typical RAM cell.**

RAM is disabled and the data outputs are tri-stated.

2. The "RN/W" signal determines whether a read or write operation is to be performed. The polarity of this signal is compatible with I/OE signal so that whenever a write operation is initiated, the data output is tri-stated.

3. The CLOCK signal synchronizes the RAM operation with the rest of the system.

The simulation result of Fig. 4.31 shows how the output signals of control section affect the operation of the RAM.

### 4.5.2 Read Operation

To increase the density, the transistors of the RAM cell are of minimum size. Due to the logic degradation of n-channel transistors in passing logic one, and because the p-channel transistors in the RAM cell are very small, this structure is not good enough to pull up the bit lines. Therefore the scheme for reading a RAM cell is pulling up the bit lines through a more efficient precharge circuit. Depend on the datum stored in a RAM cell, one of the bit lines will be pulled down during the read operation while the other remains high. This variation in the voltage level is sensed and amplified by the corresponding sense amplifier and will be reflected to the output latch. The control signal "I/OE" is logic one during the write operation, allowing the data to be put on the data bus. The precharge circuits use n-channel transistors to pull up the bit lines. This results in bit lines being charged to a voltage which is less than "Vdd" by threshold voltage level of the n-channel transistors. This subsequently results in dramatic increase of speed [37] as the logic levels of the bit lines are closer to the threshold voltage of the sense amplifiers, and therefore it takes less time for the sense amplifiers to detect the voltage drop on the bit lines.

### 4.5.3 Write Operation

The basic goal of a RAM write operation is to insert two complement signals into the RAM cell through the bit lines so that to modify the internal logic levels of the cell to the new ones. As in the read operation, the write operation also starts with precharging the bit lines to logic one when the clock signal is low. The write enable transistors of Fig. 4.29 are manipulated by

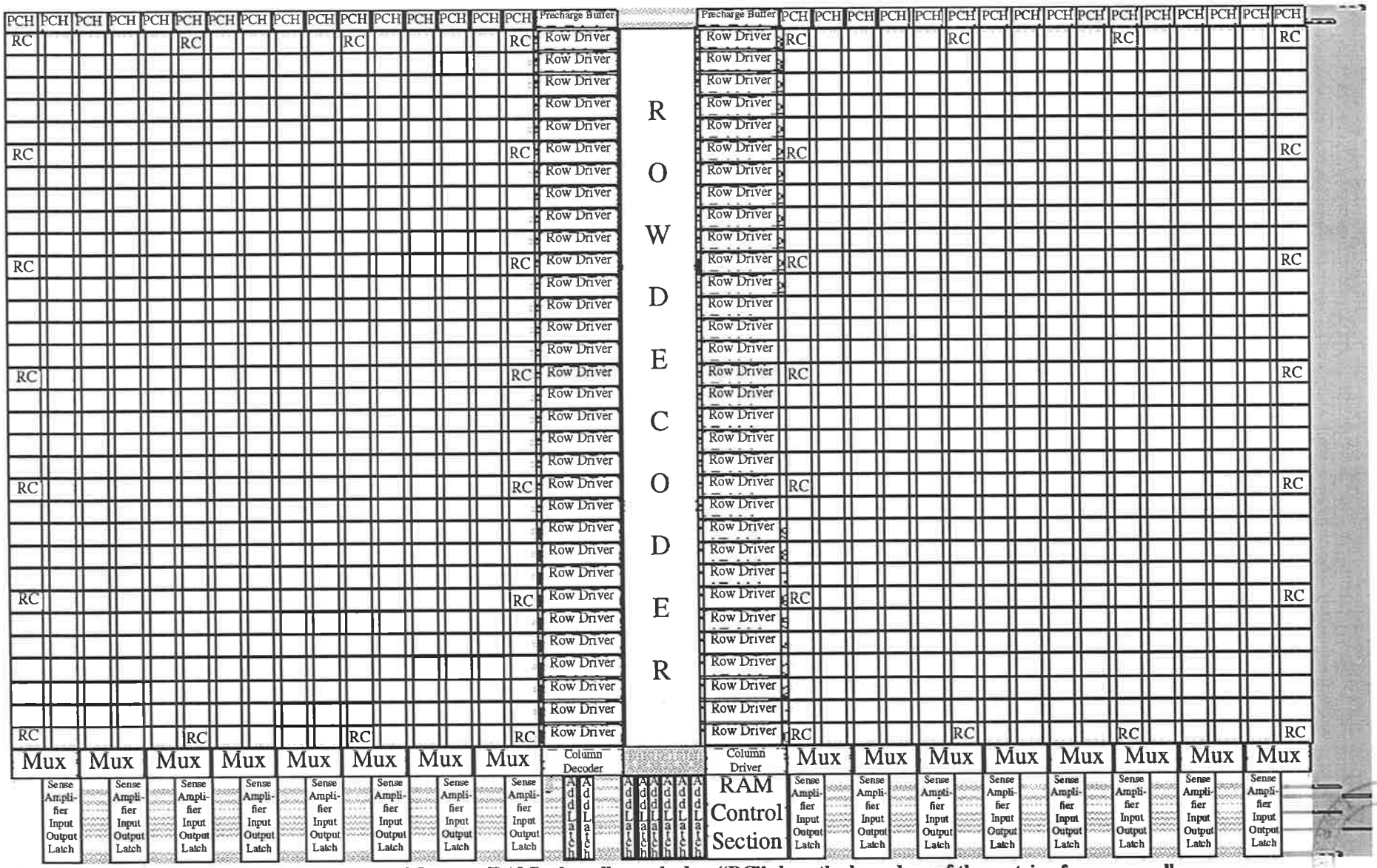



**Figure 4.30--Floor plan of the Random Access Memory (RAM), the cells marked as "RC" show the boundary of the matrix of memory cells.**

the control signal "WE". This signal is logic high during the write operation, allowing the data and its complement to move to the bit lines. Subsequently, one of the bit lines is driven to ground level while the other remains high. This results in the change of status of the selected cell when the row select signal is activated.

### 4.5.4 RAM Simulation Result

The diagram of Fig. 4.31 shows the result of simulation of the random access memory. The simulation consists of two write operations, followed by two read operations of the same locations of the memory. The first write operation stores the number 0000 (Hex) in address 00 (Hex) of the RAM, whereas the second one stores the number 1111 (Hex) in location FC (Hex). Note that the two least significant bits of the address are always zero in this simulation, therefore the bit lines "BIT" and "BITN" show how the other bit lines are affected by read and write operations. The result of simulation shows that the required time for memory read operation is about 16.5 ns from the rising edge of the clock signal. This time is less for memory write operation and is about 8.5 ns.

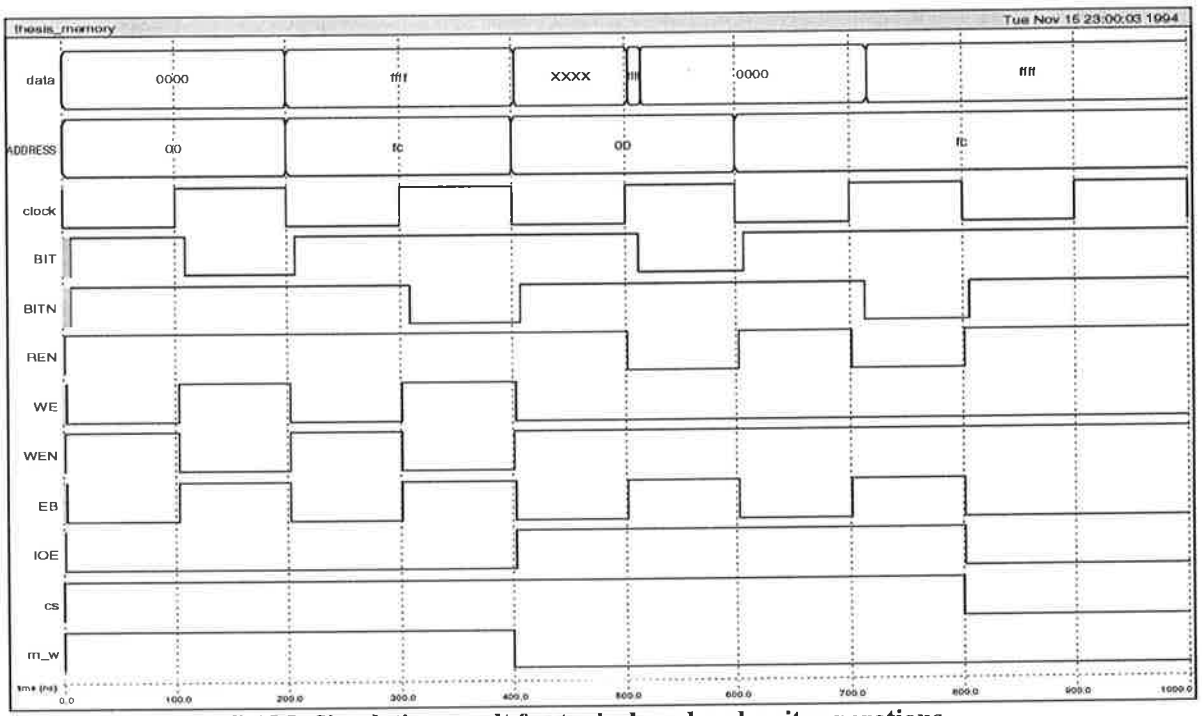



**Figure 4.31--Static RAM- Simulation result for typical read and write operations**

## 4.6 Read Only Memory Design

Read only memories or ROMs are storage elements in which the information can be read out, but can not be written in as readily as in RAMs. Depend on the technique used to manipulate the information in the memory, ROMs can be divided into the following categories [20]

- Mask Programmable ROM
- PROM
- EPROM
- EEPROM

The packing density of a mask programmable ROM, or simply ROM, can be at least several times more than that of a static RAM. The employed ROM for our processor is of mask programmable type and has the capacity of storing 512 words of 9 bits length. It can be programmed with contact mask, therefore it is called "contact" ROM in contrast with "active" ROM.

### 4.6.1 Structure Of The ROM

The implementation of the ROM requires one transistor per storage bit. The structure is static (in a sense that decoders, sense amplifiers, etc. are static circuits) and usually carried out by using NOR arrays [37], as shown in Fig. 4.32. The stored information can be held constantly even when the power is off. The address information is stored in duplicate into the address latches provided on the left and right side of the ROM (see Fig. 4.32). The left and right column decoders are one of four decoder and each of them controls four select lines of the multiplexers. The least significant bits $a_0$, $a_1$,and $a_2$ of the address lines ascertain the selected column and the rest of the address lines are used by the row decoders to determine one of the

$2^6 = 64$ accessible rows. The left row decoder decodes the even lines, whereas the right one decodes the odd lines. The control block generates the required control signals which synchronize the operation of the ROM. As the ROM must be always in active operation mode, the input control signals "CS" and "OEN" are permanently connected to the Vdd line. This results in output tri_state buffers being always enabled and one set of data being written on the output data lines $C_i$ on each rising edge of the clock signal. The floor plan of the ROM is depicted in Fig. 4.33.

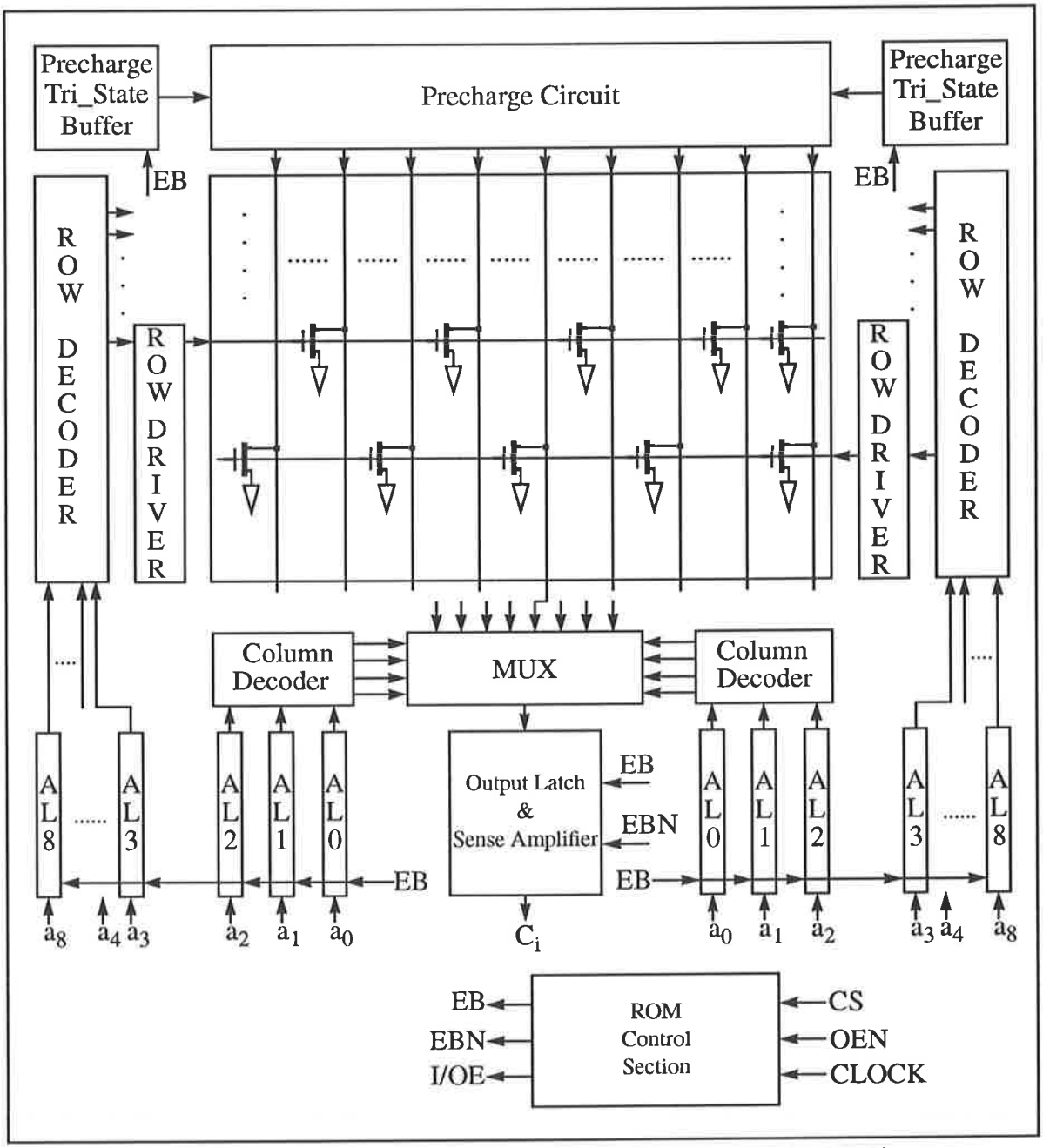



**Figure 4.32--Block diagram and supporting circuits of the ROM; (AL = Address Latch)**

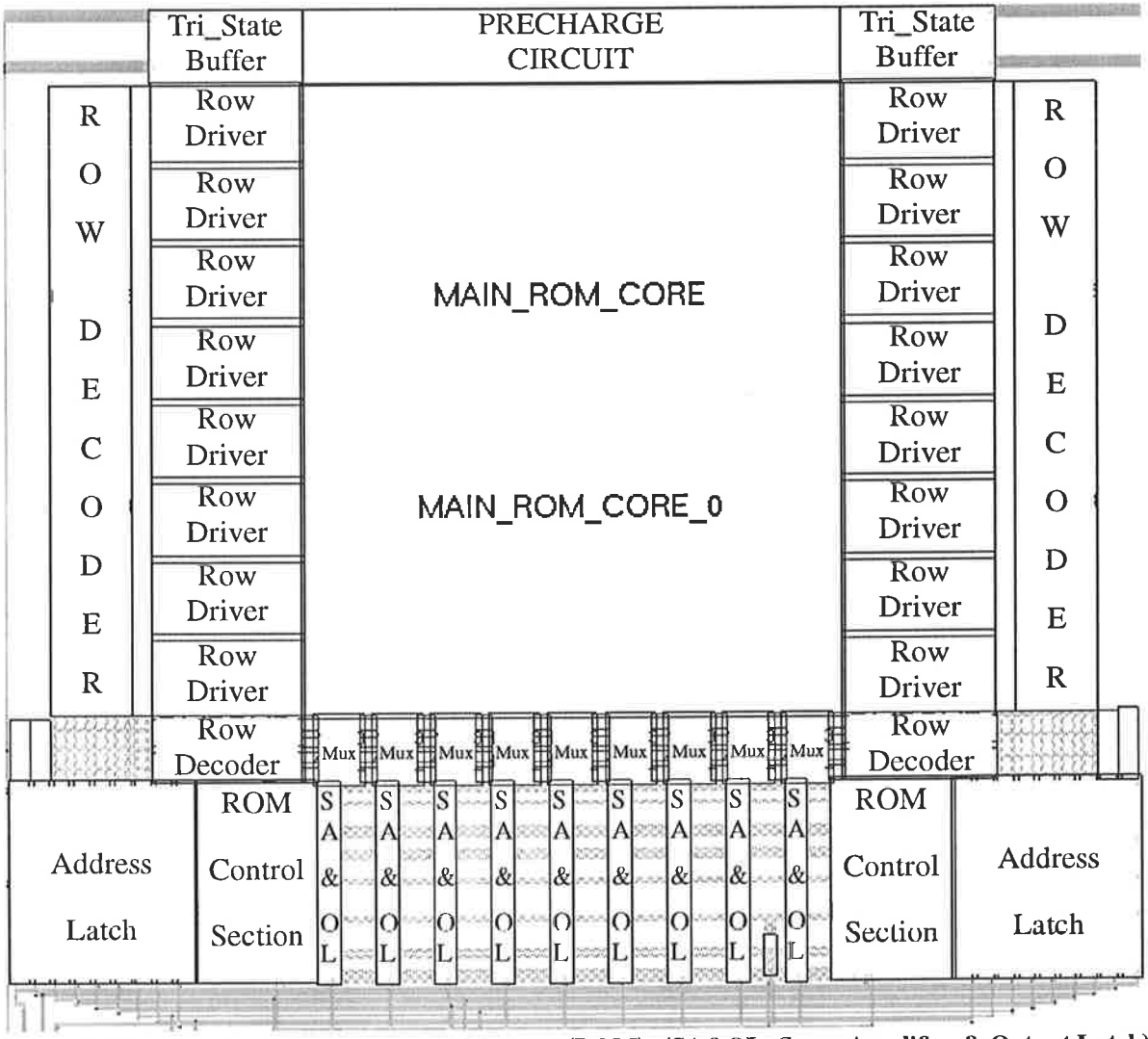



**Figure 4.33--Floor plan of the Read Only Memory (ROM); (SA&OL=Sense Amplifier & Output Latch)**

### 4.6.2 ROM Simulation Result

The ROM is programmed to store the machine codes of the algorithm of Fig. 2.1. The content of the ROM is depicted in table 8. In this table the first column shows the starting address for the data on each row in decimal format. Consecutive data is continued on a new row and in this case a plus sign is placed in the first column. The data are shown in Hexadecimal format. The locations which are not specified in table 8 (locations 340 to 495) are filled by pattern 00 (Hex.), and therefore initiate a no operation (NOP) command.

The simulation result of Fig. 4.34 shows the read operation of the first four locations of the

**TABLE 8. Machine codes in Hexadecimal format stored in the ROM.**

| 00 | 00 | 04 | 06 | 07 | 1E | 1E | 3C | 04 | 05 | 09 | 0A | 05 | 0C | 0D | 0E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| + | 02 | 0D | 0D | 0E | 0D | 0E | 0D | 0F | 0D | 0F | 04 | 05 | 09 | 10 | 01 |
| + | 1F0 | 15 | 04 | 05 | 09 | 0A | 09 | 16 | 17 | 09 | 18 | 09 | 16 | 17 | 19 |
| + | 18 | 02 | 29 | 09 | 16 | 17 | 19 | 18 | 04 | 21 | 1C | 36 | 9E | 04 | 06 |
| + | 16 | 1A | 10 | 26 | 18 | 04 | 21 | 06 | 1F | 0D | 0F | 0D | 0F | 11 | 11 |
| + | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 05 | 1A | 24 | 25 | 18 | 04 | 05 |
| + | 1A | 0A | 16 | 24 | 17 | 12 | 04 | 06 | 10 | 26 | 2B | 33 | 1A | 07 | 25 |
| + | 27 | 12 | 26 | 28 | 35 | 64 | 04 | 06 | 07 | 04 | 06 | 24 | 10 | 05 | 1A |
| + | 16 | 26 | 12 | 21 | 1A | 10 | 17 | 0B | 25 | 06 | 0C | 16 | 04 | 21 | 06 |
| + | 1E | 0F | 0D | 0F | 11 | 22 | 17 | 25 | 18 | 1F | 16 | 1A | 1B | 0F | 0D |
| + | 0F | 11 | 22 | 17 | 19 | 29 | 02 | 81 | 04 | 21 | 1A | 10 | 04 | 05 | 1A |
| + | 16 | 17 | 0B | 3E | 01 | 1A | 0C | 1B | 0E | 02 | AC | 1B | 0E | 1B | 3E |
| + | 01 | 06 | 08 | 1E | 02 | B6 | 08 | 1B | 0F | 0D | 0F | 2F | C3 | 36 | CA |
| + | 30 | 04 | 05 | 1A | 10 | 01 | 1F0 | 04 | 21 | 1D | 04 | 21 | 1A | 10 | 04 |
| + | 05 | 1A | 16 | 17 | 0B | 3E | 01 | 1A | 1F | 1B | 08 | 02 | DB | 1B | 08 |
| + | 1B | 0F | 0D | 0F | 11 | 40 | EA | 36 | F1 | 04 | 05 | 1A | 10 | 01 | 1F0 |
| + | 31 | 04 | 1A | 20 | 1B | 0F | 0D | 0F | 04 | 21 | 06 | 10 | 01 | 1F0 | 11 |
| + | 04 | 21 | 1D | 04 | 06 | 0A | 21 | 1A | 32 | 37 | 3A | 04 | 21 | 1A | 10 |
| + | 04 | 05 | 1A | 0A | 16 | 17 | 2A | 29 | 25 | 05 | 06 | 0C | 16 | 04 | 21 |
| + | 06 | 1E | 0F | 0D | 0F | 11 | 22 | 17 | 25 | 18 | 29 | 02 | 117 | 43 | 12C |
| + | 04 | 21 | 1A | 0A | 21 | 06 | 32 | 35 | 3A | 04 | 05 | 1A | 0A | 16 | 21 |
| + | 1A | 10 | 17 | 2A | 2B | 1C | 1A | 10 | 16 | 17 | 12 | 05 | 05 | 05 | 10 |
| + | 17 | 18 | 02 | 148 | 10 | 17 | 05 | 18 | 25 | 3F | | | | | |
| 496 | 04 | 05 | 06 | 0A | 11 | 12 | 13 | 2E | 11 | 12 | 14 | 11 | 05 | 02 | 1F9 |
| + | 03 | | | | | | | | | | | | | | |

ROM. When the clock signal is logic zero, the precharge circuits are turned on and pull up the bit lines to logic one. At this time the word select lines are disabled in order to prevent

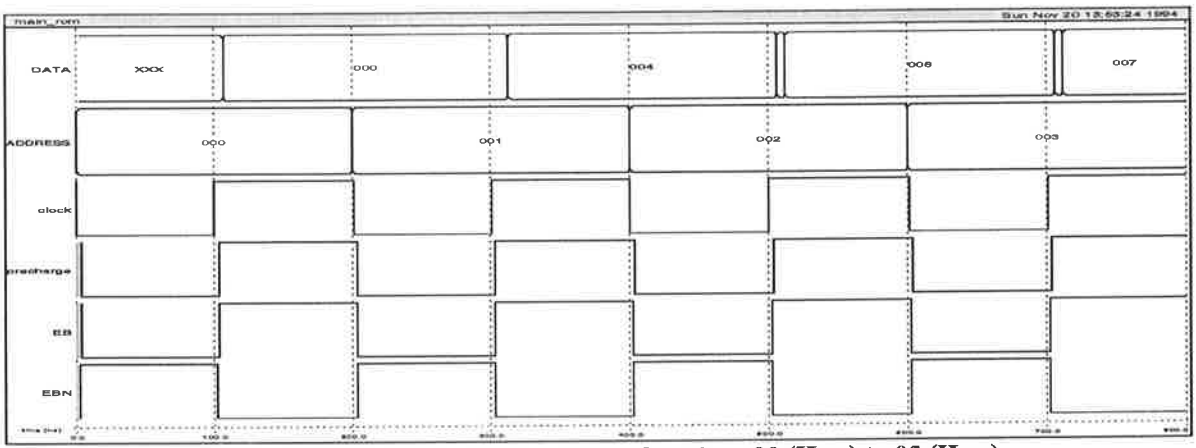



**Figure 4.34--Static Rom-Read operation of the address locations 00 (Hex.) to 03 (Hex.)**

any DC power loss. At the rising edge of the clock signal the pull-up circuits are turned off and one word line is active, allowing the data of the specified row to be transferred to the output latches. The status of the bit lines depend on the existence of the polysilicon contacts on the gate of the corresponding transistors. If a contact exists then the transistor is on which consequently pulls down the corresponding bit line, otherwise the bit line remains high. The additional signals shown in Fig. 4.34 are control lines of input/output latches and precharge circuits. The result of the simulation shows that the maximum required time for reading an arbitrary location of the ROM is about 12.5 ns from the rising edge of the clock signal.

- **Summary**

In this chapter the mathematical backgrounds required for understanding the operation of basic blocks of the processor were discussed, and the algorithm based on which each unit operates was developed. The hardware design and specifications of each segment are mainly governed by the developed algorithm. Furthermore, the performance of each element is influenced by the floor plan design of that particular element. In order to improve the overall performance of the processor, the layout of each unit designed in such a way to be consistent with other elements of the processor, so that the placement and routing of different blocks can be done easier and the employed area can be minimized. The speed of operation of each component is also an important factor, which of course is not independent of other features such as the shape and size of the block, and considered to be in agreement with the basic principle of high speed DSP hardware design while maintaining the required physical harmony.

Additionally, the employed methodologies for the simulation of individual units were discussed and the simulation results were depicted. The overall performance of the processor is examined in the next chapter along with the performance of the maximum entropy algorithm, by introducing a synthetic data generated by a computer program.

# Chapter 5

# CONCLUSIONS AND THE FUTURE DEVELOPMENTS

Recent advances in the VLSI technology has made possible the design of handy medical equipments which can provide crucial informations about the patient condition, primarily by replacing some of the traditional diagnostic instruments with more efficient and reliable tools. In this study a special purpose DSP processor with an application in the spectral analysis of the heart sounds was designed and simulated. The algorithm based on which the processor works, was devised by Andersen [1] from the university of Copenhagen, and constructed upon the Burg method of estimating the AR parameters. The architecture of the processor takes advantage of available DSP techniques in processor design such as parallel-

ism and pipelining.

In order to verify the hardware of the processor as well as the employed maximum entropy algorithm, trials with synthetic data were carried out. The synthetic data was simply generated by a computer program so that it looked similar to the third heart sound in the time domain [11]. This was achieved by the summing of three sinusoids with frequencies of 10, 20, and 40 Hz and relative amplitudes of 1.0, 0.5 and 0.25 respectively. Normally distributed random noise was added to simulate the actual data more realistically. The length of the data was 100 samples which can comfortably be accommodated in the built in RAM of the processor. The graph of Fig. 5.1(a) shows the sketch of the synthetic data in the time domain, whereas the one in Fig. 5.1(b) presents the augmented normal noise.

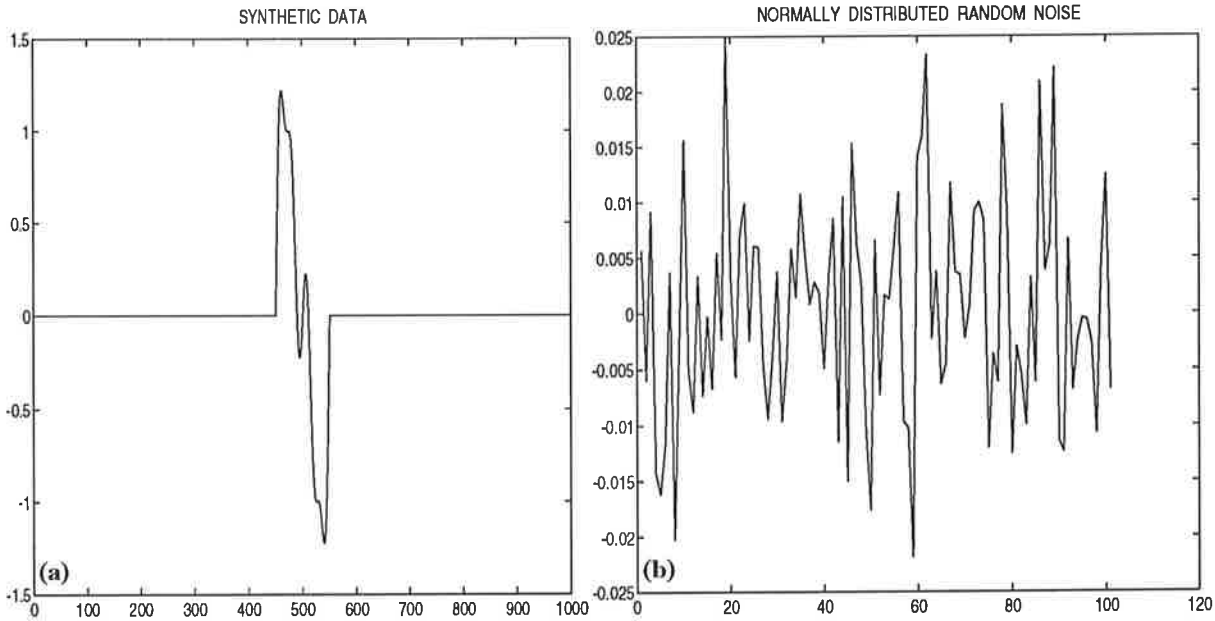



**Figure 5.1--(a) The synthetic data (b) The added normally distributed random noise**

The filter coefficients generated by the processor were taken to the Matlab package where the F.F.T. was applied to them and the final graph of Fig. 5.2 was developed. From the above and similar experiments it is obvious that "Maximum Entropy" method offers superior resolution compared to F.F.T. specially with short observation lengths of simulated data. As it is depict-

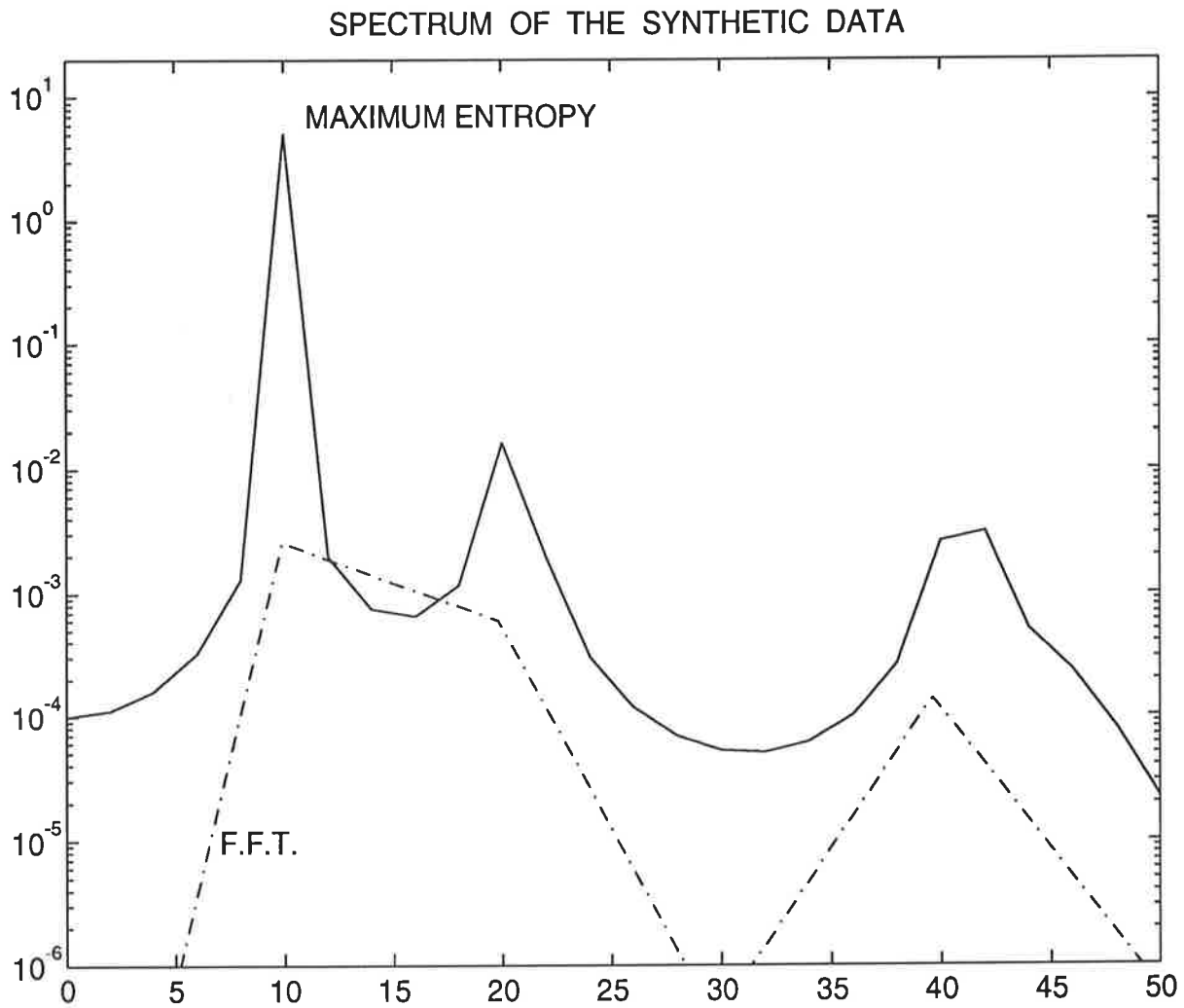



**Figure 5.2--Spectrum of the synthetic data of Fig. 5.1(a)**

ed in Fig. 5.2 the F.F.T. did not resolve the 10 Hz peak, whereas the maximum entropy approach produced two sharp peaks at 10 and 20 Hz.

With 16-bit data length and provision of handling 32-bit intermediate results in case of successive multiply/divide operations, the processor provides the desired high level of accuracy in almost all phases of the processing. The potential danger however is when it comes to reforming the original stored data as proposed by the following two equations in the main loop of the flow diagram of Fig. 2.1.

$$b_m(n) = b_{m-1}(n) - a_m(m) \cdot b'_{m-1}(n-1)$$

$$b'_m(n) = b'_{m-1}(n-1) + a_m(m) \cdot b_{m-1}(n)$$

Since a multiply operation is involved in the calculation of the above equations, the results are 32-bit in length with the most significant bits stored in register Z and the least significant bits retained in register W. The current program of the processor discards the least significant part in the W register which is always less than one, assuming that the absolute value of the most significant part is much larger than one. This is however only true for the prescaled initial data with which the processor starts, and there is no guarantee that the intermediate results comply with the above assumption.

One way to overcome this problem is to modify the software so that to rescale the intermediate results before the next step, considering that the resultant filter coefficients are independent of the amplitude of the sampled signal. The other solution is to provide the processor with double precision handling of the results. This however demands some extra memory locations as well as modification of the software. Furthermore, it increases the total processing time of the data. The best solution is to modify the architecture of the ALU from fixed point to floating point. This scheme not only increases the overall accuracy of the processing, but also provides the required facility for incorporating F.F.T. algorithm with maximum entropy method which gives the final spectrum as proposed by the following equation of chapter two, rewritten here for reference.

$$\bar{S}(\omega) = \frac{p}{\left|1 - \sum_{k=1}^{N} a_k \cdot e^{-j\omega kT}\right|^2}$$

The performance of the ALU can be further improved by introducing a reciprocator. A reciprocator is fairly inexpensive hardware, which provides a fast divider if combined with a fast multiplier.

In the design of the processor the following CAD tools were used

1. MAGIC layout editor was the main editor used in the hierarchical design of different blocks of the processor.

2. OCTTOOLS is a package of variety of tools. It is mainly used for gate and transistor level design and functional simulation of sub-modules and random logics, as well as the optimization of logic circuits. The possibility of changing the OCTTOOLS layouts to MAGIC format makes it an extremely useful tool to be used in conjunction with MAGIC.

3. Detailed simulation of the fairly small but usually substantial submodules which affect the entire performance of the processor was carried out by using the HSPICE simulator.

4. IRSIM is an event driven switch level simulator which provides a fast and efficient tool for the simulation of larger blocks. Furthermore, it yields a fairly accurate perception of the system's timing and is the simulator which was used to test the performance of the whole chip.

# Appendix A

# Instruction Set

The control unit of the processor is capable to distinguish 69 different instructions. The instruction set is grouped in order under four different functional headings as

- Data Transfer Group

This group of instructions transfers data to and from registers and memory. Unless indicated otherwise, all instructions in this group are one cycle commands as they need only one machine cycle for their execution.

MOV Y, Z (Move Register)
(Y) <- (Z)
The content of register Z is moved to register Y

MOV X, Z (Move Register)
(X) <- (Z)
The content of register Z is moved to register X

MOV CO1, Z (Move Register)
(CO1) <- (Z)
The content of register Z is moved to register CO1

MOV CO2, Z (Move Register)
(CO2) <- (Z)
The content of register Z is moved to register CO2

MOV A1, Z (Move Register)

(A1) <- (Z)
The content of register Z is moved to register A1.

MOV A2, Z (Move Register)
(A2) <- (Z)
The content of register Z is moved to register A2.

MOV MU1, Z (Move Register)
(MU1) <- (Z) & (MU2) <- (MU1) & (MU3) <- (MU1)*(MU2) &
((Z) (W)) <- 0

The content of register Z is moved to register MU1. The content of register MU1 is moved to register MU2. Register MU3 is loaded with MU1*MU2. Accumulator is reset to zero.

MOV1 MU1, TEMP (Move Register)
(MU1) <- (TEMP) & (MU2) <- (MU1) & (MU3) <- (MU1)*(MU2) &
Load ((Z) (W))
The content of register TEMP is moved to register MU1. The content of register MU1 is moved to register MU2. Register MU3 is loaded with MU1*Mu2. Accumulator is loaded with the outputs of 32-bit adder.

MOV X, W (Move Register)
(X) <- (W)
The content of register W is moved to register X.

MOV2 MU1, TEMP (Move Register)
(MU1) <- (TEMP) & (MU2) <- (MU1)
The content of register TEMP is moved to register MU1. The content of register MU1 is moved to register MU2.

MOV MU1, W (Move Register)
(MU1) <- (W) & (MU2) <- (MU1) & (MU3) <- MU1*MU2 & ((Z) (W)) <- 0
The content of register W is moved to register MU1. The content of register MU1 is moved to register MU2. Register MU3 is loaded with MU1*MU2. Accumulator is reset to zero.

MOV A1, A2 (Move Register)
(A1) <- (A2)
The content of register A2 is moved to register A1.

MOV1 M, Z (Move to memory)
(0A1) <- (Z) & (A1) <- (A1) + 1
The content of register Z is moved to the memory location (page 0), whose address is in register A1. Register A1 is incremented by one.

MOV1 M, TEMP (Move to memory)
(0A1) <- (TEMP) & (A1) <- (A1) + 1
The content of register TEMP is moved to the memory location (page 0), whose address is in register A1. Register A1 is incremented by one.

MOV2 M, TEMP (Move to memory)
(1A1) <- (TEMP) & (A1) <- (A1) + 1
The content of register TEMP is moved to the memory location (page 1), whose address is in register A1. Register A1 is incremented by one.

MOV1 M, W (Move to memory)

(0A1) <- (W)
The content of register W is moved to the memory location (page 0), whose address is in register A1.

MOV2 M, W (Move to memory)
(1A1) <- (W)
The content of register W is moved to the memory location (page 1), whose address is in register A1.

MOV2 M, Z (Move to memory)
(1A1) <- (Z) & (A1) <- (A1) + 1
The content of register Z is moved to the memory location (page 1), whose address is in register A1. Register A1 is incremented by one.

MOV TEMP, ML (Move from memory)
(TEMP) <- (ML) & (ML) <- (1A1)
The content of memory location (page 1), whose address is in register A1, is moved to register ML (Memory Latch). The content of register ML is moved to register TEMP.

MOV CO1, ML (Move from memory)
(CO1) <- (ML) & (ML) <- (0A1)
The content of memory location (page 0), whose address is in register A1, is moved to register ML (Memory Latch). The content of register ML is moved to register CO1.

MOV CO2, ML (Move from memory)
(CO2) <- (ML) & (ML) <- (0A1)
The content of memory location (page 0), whose address is in register A1, is moved to register ML (Memory Latch). The content of register ML is moved to register CO2.

MOV A2, ML (Move from memory)
(A2) <- (ML) & (ML) <- (1A1)
The content of memory location (page 1), whose address is in register A1, is moved to register ML (Memory Latch). The content of register ML is moved to register A2.

MOV1 MU1, ML (Move from memory)
(ML) <- (0A1) & (MU1) <- (ML) & (MU2) <- (MU1) &
(MU3) <- (MU1)*(MU2) & (A1) <- (A1) + 1 & ((Z) (W)) <- 0
The content of memory location (page 0), whose address is in register A1, is moved to register ML (Memory Latch). The content of register ML is moved to register MU1. The content of register MU1 is moved to register MU2. Register MU3 is loaded with MU1*MU2. Register A1 is incremented by one. Accumulator is reset to zero.

MOV2 MU1, ML (Move from memory)
(ML) <- (1A1) & (MU1) <- (ML) & (MU2) <- (MU1) &
(MU3) <- (MU1)*(MU2) & (A1) <- (A1) + 1 & ((Z) (W)) <- 0
The content of memory location (page 1), whose address is in register A1, is moved to register ML (Memory Latch). The content of register ML is moved to register MU1. The content of register MU1 is moved to register MU2. Register MU3 is loaded with MU1*MU2. Register A1 is incremented by one. Accumulator is reset to zero.

MOV3 MU1, ML (Move from memory)
(ML) <- (1A1) & (MU1) <- (ML) & (MU2) <- (MU1) &
(MU3) <- (MU1)*(MU2) & (A1) <- (A1) + 1 & Load ((Z) (W))
The content of memory location (page 1), whose address is in register A1, is moved to regis-

ter ML (Memory Latch). The content of register ML is moved to register MU1. The content of register MU1 is moved to register MU2. Register MU3 is loaded with MU1*MU2. Register A1 is incremented by one. Accumulator is loaded with the outputs of 32 bit adder.

MOV4 MU1, ML (Move from memory)
(ML) <- (0A1) & (MU1) <- (ML) & (MU2) <- (MU1) &
(MU3) <- (MU1)*(MU2) & (A1) <- (A1) + 1 & Load ((Z) (W))
The content of memory location (page 0), whose address is in register A1, is moved to register ML (Memory Latch). The content of register ML is moved to register MU1. The content of register MU1 is moved to register MU2. Register MU3 is loaded with MU1*MU2. Register A1 is incremented by one. Accumulator is loaded with the outputs of 32-bit adder.

MOV5 MU1, ML (Move from memory)
(ML) <- (0A1) & (MU1) <- (ML) & (MU2) <- (MU1)
The content of memory location (page 0), whose address is in register A1, is moved to register ML (Memory Latch). The content of register ML is moved to register MU1. The content of register MU1 is moved to register MU2.

MOV6 MU1, ML (Move from memory)
(ML) <- (1A1) & (MU1) <- (ML) & (MU2) <- (MU1)
The content of memory location (page 1), whose address is in register A1, is moved to register ML (Memory Latch). The content of register ML is moved to register MU1. The content of register MU1 is moved to register MU2.

MOV7 MU1, ML (Move from memory)
(ML) <- (0A1) & (MU1) <- (ML) & (MU2) <- (MU1) & (A1) <- (A1) + 1
The content of memory location (page 0), whose address is in register A1, is moved to register ML (Memory Latch). The content of register ML is moved to register MU1. The content of register MU1 is moved to register MU2. Register A1 is incremented by one.

MOV X, ML (Move from memory)
(ML) <- (0A1) & (X) <- (ML)
The content of the memory location (page 0), whose address is in register A1, is moved to register ML (Memory latch). The content of register ML is moved to register X.

MOV Y, ML (Move from memory)
(ML) <- (0A1) & (Y) <- (ML) & (A1) <- (A1) -1
The content of the memory location (page 0), whose address is in register A1, is moved to register ML (Memory latch). The content of register ML is moved to register Y. Register A1 is decremented by one.

MOV1 ML, M (Read memory)
(ML) <- (0A1) & (A1) <- (A1) + 1
The content of the memory location (page 0), whose address is in register A1, is moved to register ML (Memory latch). Register A1 is incremented by one.

MOV2 ML, M (Read memory)
(ML) <- (0A1)
The content of the memory location (page 0), whose address is in register A1, is moved to register ML (Memory latch).

MOV3 ML, M (Read memory)
(ML) <- (1A1)
The content of the memory location (page 1), whose address is in register A1, is moved to

register ML (Memory latch).

MVI A1, data (Move Immediate)
(A1) <- (byte 2)
The content of byte 2 of the instruction is moved to register A1. This is a two cycle instruction.

- Arithmetic Group

This group of instructions performs arithmetic operations on data in registers X and Y. Note that any data transfer to register MU1 initiates a multiply-accumulate operation which is a two cycle operation. Multiply operation has come under "Data Transfer Group". All the other arithmetic operations are one cycle operation. All the instruction in this group affect the registers BG, Division Overflow and Add/Subtract Overflow.

ADD (Add registers X and Y)
(Z) <- (Y) + (X)
The content of register X is added to the content of the register Y. The result is placed in the register Z.

SUB (Subtract register X from Y)
(Z) <- (Y) - (X)
The content of register X is subtracted from the content of register Y. The result is placed in the register Z.

DIV1 (Add/Subtract X and Y)
(Z) <- (Y)± (X)
Depend on the control signals of the division control block, the content of the register X is added to or subtracted from the content of the register Y. The result is placed in the register Z.

DIV2 (Add/Subtract X and Y)
(Z) <- (Y)± (X)
Depend on the control signals of the division control block, the content of the register X is added to or subtracted from the content of the register Y. The result is placed in the register Z.

INR A2 (Increment Register)
(A2) <- (A2) + 1
The content of register A2 is incremented by one.

DCR A2 (Decrement Register)
(A2) <- (A2) - 1
The content of register A2 is decremented by one.

INR A1 (Increment Register)
(A1) <- (A1) + 1
The content of register A1 is incremented by one.

DCR A1 (Decrement Register)
(A1) <- (A1) - 1
The content of register A1 is decremented by one.

- Branch Group

This group of instructions alter normal sequential flow of the program. The two types of branch instructions are unconditional and conditional. Unconditional transfers simply per-

form the specified operation on register PC (the program counter). Conditional transfers examine the status of one of the four processor flags or the equivalence of register CO1 with CO2 or A1 to determine if the specified branch is to be executed. Unless indicated otherwise, all instructions in this group are two cycle commands.

JMP addr (Jump)
(PC) <- (byte 2)
Control is transferred to the instruction whose address is specified in byte2 of the current instruction.

BNCH addr (Call Subroutine)
(PC) <- (byte 2)
(ST) <- (PC) + 2
Control is transferred to the instruction whose address is specified in byte 2 of the current instruction. The content of register PC is incremented by 2 and moved to register ST (Stack).

RTN (Return)
(PC) <- (ST)
The content of register ST is moved to register PC. Control is transferred to the instruction whose address is specified in register ST. This is a one cycle instruction.

JNE (Conditional Jump)
If A1 is not equal to $CO1_{6:0}$ Then
(PC) <- byte2
If the specified condition is true, control is transferred to the instruction whose address is specified in byte 2 of the current instruction; otherwise, control continues sequentially.

JBG (Conditional Jump)
If register BG is equal to one Then
(PC) <- byte2
If the specified condition is true, control is transferred to the instruction whose address is specified in byte 2 of the current instruction; otherwise, control continues sequentially.

JLE (Conditional Jump)
If register CO1 is less than or equal to register CO2 Then
(PC) <- byte2
If the specified condition is true, control is transferred to the instruction whose address is specified in byte 2 of the current instruction; otherwise, control continues sequentially.

JLT (Conditional Jump)
If register CO1 is less than register CO2 Then
(PC) <- byte2
If the specified condition is true, control is transferred to the instruction whose address is specified in byte 2 of the current instruction; otherwise, control continues sequentially.

JEQ (Conditional Jump)
If register CO1 is equal to register CO2 Then
(PC) <- byte2
If the specified condition is true, control is transferred to the instruction whose address is specified in byte 2 of the current instruction; otherwise, control continues sequentially.

JF1 (Conditional Jump)
If Flag1 is logic one Then
(PC) <- byte2
If the specified condition is true, control is transferred to the instruction whose address is

specified in byte 2 of the current instruction; otherwise, control continues sequentially.

JF2 (Conditional Jump)
If Flag2 is logic one Then
(PC) <- byte2
If the specified condition is true, control is transferred to the instruction whose address is specified in byte 2 of the current instruction; otherwise, control continues sequentially.

JF3 (Conditional Jump)
If Flag3 is logic one Then
(PC) <- byte2
If the specified condition is true, control is transferred to the instruction whose address is specified in byte 2 of the current instruction; otherwise, control continues sequentially.

JF4 (Conditional Jump)
If Flag4 is logic one Then
(PC) <- byte2
If the specified condition is true, control is transferred to the instruction whose address is specified in byte 2 of the current instruction; otherwise, control continues sequentially.

- Machine Control Group

This group of instructions alters internal control flags, set registers and perform the shift operation on the accumulator.

SET A1 (Set Register)
(A1) <- (1111111)
The content of register A1 is changed to 1111111.

NOP (No Operation)
No operation is performed. The registers and flags are unaffected.

HLT (Halt)
The processor is stopped. The registers and flags are unaffected. The processor remains in this state until be reset.

SET OVF (Check Division Overflow)
Set overflow flag, if division overflow has occurred.

SET F1 (Set Flag F1)

RST F1 (Reset Flag F1)

SET F2 (Set Flag F2)

RST F2 (Reset Flag F2)

SET F3 (Set Flag F3)

RST F3 (Reset Flag F3)

SET F4 (Set Flag F4)

RST F4 (Reset Flag F4)

SHL (Shift Left With Zero)

$(Z_0) <- (W_{15})$ ; $(W_0) <- 0$

The content of the accumulator is shifted left one position. The low order bit of register Z is set to the value shifted out of the high order bit position of register W. The high order bit of register Z is lost. The low order bit of register W is set to zero.

SHD (Division Shift Left)

$(Z_0) <- (W_{15})$ ; $(W_0) <-$ D.C.

The content of the accumulator is shifted left one position. The low order bit of register Z is set to the value shifted out of the high order bit position of register W. The high order bit of register Z is lost. The low order bit of register W is set to the value determined by division control block.

# Appendix B

# Assembly Language Program For Maximum Entropy Spectral Estimation

```
!C    This routine finds the maximum entropy estimate of the predictor filter coefficients for
      the spectral density. The algorithm is based on the Burg method, and was devised by
      Andersen of the university of Copenhagen. The first column either shows a comment
      (!C) or the address of the instruction in the ROM in hexadecimal base.

!C        Set up routine: load registers MU1, MU2, and TEMP with a one, and set flag F4,
          therefore the user defines the model order (M).
          NOP
          SET     A1
          MV3     ML, M
          MV      TEMP, ML
          MV6     MU1, ML
          MV6     MU1, ML
          SET     F4

!C        Calculate the mean value of the sampled data.
          SET     A1
          INR     A1
          MV1     ML, M
          MV      CO1, ML
```

```
        INR     A1
        MV1     MU1, ML
0D      MV2     MU1, TEMP
        MV4     MU1, ML
        JNE     0D (Hex.)
        MV2     MU1, TEMP
        MV4     MU1, ML
        MV2     MU1, TEMP
        MV4     MU1, ML
        MV2     MU1, TEMP
        MV1     MU1, TEMP
        MV2     MU1, TEMP
        MV1     MU1, TEMP
        SET     A1
        INR     A1
        MV1     ML, M
        MV      X, ML
        BNCH    1F0 (Hex.)
        MV      X, W
```

!C Generate two copies of data by subtracting the mean value from each datum and storing the result in both pages, zero and one, of data RAM.

```
        SET     A1
        INR     A1
        MV1     ML, M
        MV      CO1, ML
        MV1     ML, M
        MV      Y,ML
        SUB
        MV1     ML, M
        MV1     M, Z
29      MV1     ML, M
        MV      Y, ML
        SUB
        MV2     M, Z
        MV1     M, Z
        JNE     29 (Hex.)
        MV1     ML, M
        MV      Y, ML
        SUB
        MV2     M, Z
        MV1     M, Z
```

!C Set M=1.

```
        SET     A1
        DCR     A1
        MV1     M, TEMP
```

!C Go to address 9E (Hex.); calculate A(M).

```
        JMP     9E (Hex.)
```

```
!C      M=M+1
3A      SET     A1
        MV3     ML, M
        MV      Y, ML
        MV2     ML, M
        MV      X, ML
        ADD
        MV1     M, Z

!C      Store A(M)
        SET     A1
        DCR     A1
        MV3     ML, M
        MV2     MU1, ML
        MV2     MU1, TEMP
        MV1     MU1, TEMP
        MV2     MU1, TEMP
        MV1     MU1, TEMP
        SHL
        SHL
        SHL
        SHL
        SHL
        SHL
        SHL
        SHL
        SHL
        SHL
        INR     A1
        MV2     ML, M
        MV      A2, ML
        MV      A1, A2
        MV1     M, Z

!C      Generate another copy of predictor filter coefficients and store in page 1 of data
        RAM.
        SET     A1
        INR     A1
        MV2     ML, M
        MV      CO1, ML
        MV      Y, ML
        MV      A2, ML
        SUB
        MV      Y, Z
        SET     A1
        MV3     ML, M
        MV      X, ML
        ADD
64      MV      A1, Z
        MV      CO2, Z
```

```
        MV2     ML, M
        MV      TEMP, ML
        MV      A1, A2
        MV      M, TEMP
        MV      Y, Z
        ADD
        DCR     A2
        JLT     64
        SET     A1
        MV3     ML, M
        MV      TEMP, ML
```

!C Update the data values in pages zero and one of the data RAM, using the current value of A(M).

```
        SET     A1
        MV3     ML, M
        MV      A2, ML
        MV      X, ML
        INR     A1
        MV2     ML, M
        MV      Y, ML
        ADD
        MV      Y, Z
        DCR     A1
        MV2     ML, M
        MV      X, ML
        SUB
        MV      CO1, Z
        MV      A1, A2
81      MV3     ML, M
        MV1     MU1, ML
        MV      Y, ML
        SET     A1
        DCR     A1
        MV3     ML, M
        MV6     MU1, ML
        MV1     MU1, TEMP
        MV2     MU1, TEMP
        MV1     MU1, TEMP
        SHL
        MV      X, Z
        SUB
        MV      A1, A2
        MV1     M, Z
        MV2     MU1, ML
        MV      Y, ML
        MV2     ML, M
        MV5     MU1, ML
        MV1     MU1, TEMP
        MV2     MU1, TEMP
```

```
        MV1     MU1, TEMP
        SHL
        MV      X, Z
        SUB
        MV2     M, Z
        INR     A2
        JNE     81 (Hex.)

!C      Calculate the value of A(M).

!C      Calculate DEN.
9E      SET     A1
        DCR     A1
        MV2     ML, M
        MV      X, ML
        SET     A1
        INR     A1
        MV2     ML, M
        MV      Y, ML
        SUB
        MV      CO1, Z
        MVI     A1, 01 (Hex.)
        MV2     ML, M
        MV1     MU1, ML
AC      MV5     MU1, ML
        MV4     MU1, ML
        JNE     AC (Hex.)
        MV5     MU1, ML
        MV4     MU1, ML
        MV5     MU1, ML
        MVI     A1, 01 (Hex.)
        MV3     ML, M
B6      MV3     MU1, ML
        MV6     MU1, ML
        JNE     B6 (Hex.)
        MV3     MU1, ML
        MV5     MU1, ML
        MV1     MU1, TEMP
        MV2     MU1, TEMP
        MV1     MU1, TEMP

!C      Calculate NOM.
        JBG     C3 (Hex.)
        JMP     CA (Hex.)
C3      SET     F1
        SET     A1
        INR     A1
        MV2     ML, M
        MV      X, ML
        BNCH    1F0 (Hex.)
```

```
CA      SET     A1
        DCR     A1
        MV2     M, W
        SET     A1
        DCR     A1
        MV2     ML, M
        MV      X, ML
        SET     A1
        INR     A1
        MV2     ML, M
        MV      Y, ML
        SUB
        MV      CO1, Z
        MVI     A1, 01 (Hex.)
        MV2     ML, M
        MV2     MU1, ML
DB      MV5     MU1, ML
        MV3     MU1, ML
        JNE     DB (Hex.)
        MV5     MU1, ML
        MV3     MU1, ML
        MV5     MU1, ML
        MV1     MU1, TEMP
        MV2     MU1, TEMP
        MV1     MU1, TEMP
        SHL
        JF1     EA (Hex.)
        JMP     F1 (Hex.)
EA      SET     A1
        INR     A1
        MV2     ML, M
        MV      X, ML
        BNCH    1F0 (Hex.)
        RST     F1
F1      SET     A1
        MV2     ML, M
        MV      MU1, W
        MV5     MU1, ML
        MV1     MU1, TEMP
        MV2     MU1, TEMP
        MV1     MU1, TEMP
        SET     A1
        DCR     A1
        MV3     ML, M
        MV      X, ML
        BNCH    1F0 (Hex.)
        SHL
        SET     A1
        DCR     A1
        MV2     M, W
```

```
!C      If M=1, then go to address 3A (Hex.); M=M+1.
        SET     A1
        MV3     ML, M
        MV      CO1, ML
        DCR     A1
        MV2     ML, M
        MV      CO2, ML
        JEQ     3A (Hex.)

!C      Update the values of predictor filter coefficients, using the current value of A(M).
        SET     A1
        DCR     A1
        MV2     ML, M
        MV      X, ML
        SET     A1
        INR     A1
        MV2     ML, M
        MV      CO1, ML
        MV      Y, ML
        SUB
        MV      A2, Z
        INR     A2
        MV      A1, A2
117     INR     A1
        MV3     ML, M
        MV1     MU1, ML
        MV      Y, ML
        SET     A1
        DCR     A1
        MV3     ML, M
        MV6     MU1, ML
        MV1     MU1, TEMP
        MV2     MU1, TEMP
        MV1     MU1, TEMP
        SHL
        MV      X, Z
        SUB
        MV      A1, A2
        MV1     M, Z
        INR     A2
        JNE     117 (Hex.)

!C      If F4=1, then go to address 12C (Hex.).
        JF4     12C (Hex.)

!C      If "M is less than the desired model order", then go to address 3A (Hex.).
12C     SET     A1
        DCR     A1
        MV2     ML, M
        MV      CO1, ML
```

```
        DCR     A1
        MV3     ML, M
        MV      CO2, ML
        JLT     3A (Hex.)

!C      Translate "A = the predictor filter coefficients sequence" one position forward.
        SET     A1
        INR     A1
        MV2     ML, M
        MV      CO1, ML
        MV      Y, ML
        DCR     A1
        MV2     ML, M
        MV      X, ML
        SUB
        MV      A2, Z
        MV      A1, Z
        MV1     M, TEMP
        MV2     ML, M
        MV      X, ML
        MV      Y, M
        SUB
        MV      Y, Z
        INR     A1
        INR     A1
148     INR     A1
        MV      X, ML
        SUB
        MV1     M, Z
        JNE     148 (Hex.)
        MV      X, ML
        SUB
        INR     A1
        MV1     M, Z
        MV      A2, A1
        HLT

!C      Division routine, starts at location 1F0 (Hex.)
1F0     SET     A1
        INR     A1
        MV3     ML, M
        MV      CO1, ML
        SHL
        MV      Y, Z
        DIV1
        SET     OVF
        SHL
1F9     MV      Y, Z
        DIV2
        SHL
```

```
INR     A1
JNE     1F9 (Hex.)
RTN
```

# Appendix C

# BDS Language Description Of The Control Unit

BDSYN is an interpreter for the hardware description of combinational logic. The input to the BDSYN is the description of the combinational logic in textual format, and the output is a collection of logic functions which realize the specified function. It is a subset of the functional simulation language BDS which is used for high level simulation. The output of the BDSYN is in BLIF (Berkeley Logic Interchange Format) and is a multiple-level logic representation of the specified function. [Chapter 6, BDSYN user's manual]

What is coming in the following is the BDS language description of the control unit of the processor.

```
!This file provides the BDS language description of the control unit.

MODEL CONTROL_UNIT
```

```
CON<3:0>,                !CONTROL LINES: CO1, CO2, 32BIT_SUB, 16BIT_SUB
WZ_ENABLE<1:0>,          !CONTROL LINES: W_ENABLE, Z_ENABLE
SHIFT_LOAD<1:0>,         !CONTROL LINES: SHIFT, LOAD
ENABLE<5:0>,             !ENABLE LINES: TRI_1EN, TRI_2EN, X_ENABLE,
                         !Y_ENABLE, M_ENABLE, 63_ENABLE

MS<0>,                   !MEMORY SELECT
RN_W<0>,                 !W/R OF THE MEMORY
L_CN<0>,                 !LOAD/COUNT CONTROL OF A1: LOAD IF 1, COUNT
                         !IF 0

UN_D<0>,                 !UP/DOWN CONTROL OF A1: UP_COUNT IF 0,
                         !DOWN_COUNT IF 1

COUNT_EN<0>,             !ENABLE OF THE ADDRESS REGISTER/COUNTER A1
P1_0N<0>,                !MEMORY (RAM) PAGE NUMBER
SN<0>,                   !IF 0 SET ADDRESS REGISTER/COUNTER A1
C1WN<0>,                 !WRITE ENABLE OF THE ADDRESS REGISTER/
                         !COUNTER A2

COUNT1_EN<0>,            !ENABLE OF THE ADDRESS REGISTER/COUNTER A2
C1L_CN<0>,               !LOAD/COUNT CONTROL OF A2: LOAD IF 1, COUNT
                         !IF 0

C1UN_D<0>,               !UP/DOWN CONTROL OF A2: UP_COUNT IF 0,
                         !DOWN_COUNT IF 1

TEMP_REG<1:0>,           !CONTROL LINES OF TEMP REGISTER: TEMP_EN
                         !TEMPW

CONREG1_EN<0>,           !ENABLE OF CONTROL REGISTER ONE (CO1)
CONREG2_EN<0>,           !ENABLE OF CONTROL REGISTER TWO (CO2)
ADDESS_EN<0>,            !ENABLE OF ADDRESS REGISTER OF CONTROL
                         !UNIT, NOTE THAT THIS SIGNAL IS DEFINED IN
                         !IMMEDIATE_ADDRESS ROUTINE

ADDESS_OUT<7:0>,         !OUTPUT OF THE ADDRESS REGISTER OF THE
                         !CONTROL UNIT

STACK_OUT<8:0>,          !OUTPUT OF THE STACK POINTER
PC_OUT<8:0>,             !PROGRAM COUNTER OUTPUT
JUMP_OUT<0>,             !STATE VARIABLE FOR JUMP/SUBROUTINE
                         !HANDLING

FLAG6_OUT<0>,            !OUTPUT STATE VARIABLE FOR TWO CYCLE
                         !COMMANDS

FLAG1_OUT<0>,            !FLAG1 OUTPUT
FLAG2_OUT<0>,            !FLAG2 OUTPUT
FLAG3_OUT<0>,            !FLAG3_OUTPUT
FLAG4_OUT<0>,            !FLAG4_OUTPUT
FLAG5_OUT<0>,            !FLAG5_OUTPUT DEVOTED TO IMMEDIATE
                         !ADDRESSING

OVRFOW_EN<0>,            !CONTROLS THE CLOCK OF THE OVERFLOW
                         !FLIP_FLOP OF CONTROL UNIT

READY<0>,                !HALT SIGNAL OF THE PROCESSOR I
OVRFOW<0>                !ANY KIND OF OVERFLOW

=

BIG_NUM<0>               !IF ONE THEN ACCU. IS MORE THAN 16 BIT
```

```
DIVOVFOW<0>,          !DIVISION OVERFLOW SIGNAL
ADDOVFOW<0>,          !ADD/SUB OVERFLOW SIGNAL
COMMAND<8:0>,         !MACHINE CODE FROM CONTROL MEMORY (ROM)
INPUT_DATA1<15:0>,    !INPUT DATA ONE FROM C.P.U.
INPUT_DATA2<15:0>,    !INPUT DATA TWO FROM C.P.U.
ADDRESS<6:0>,         !ADDRESS TO MEMORY EXCLUDING P1_ON
PC_IN<8:0>,           !PROGRAM COUNTER INPUT
JUMP_IN<0>,           !INPUT STATE VARIABLE FOR JUMP/SUBROUTINE
                      !HANDLING
FLAG6_IN<0>,          !INPUT STATE VARIABLE FOR TWO CYCLE
                      !COMMANDS
FLAG1_IN<0>,          !FLAG1 INPUT
FLAG2_IN<0>,          !FLAG2 INPUT
FLAG3_IN<0>,          !FLAG3 INPUT
FLAG4_IN<0>,          !FLAG4 INPUT
FLAG5_IN<0>,          !FLAG5 INPUT DEVOTED TO IMMEDIATE
                      !ADDRESSING
OVCON<0>,             !THIS IS CONNECTED TO THE OUTPUT Q OF THE
                      !FLIP_FLOP USED IN ROUTINE OVERFLOW
RESET<0>,             !RESET CONTROL OF P.C., TO BE CONNECTED TO
                      !RESET LINE OF THE PROCESSOR
ADDESS_IN<7:0>,       !INPUT OF THE ADDRESS REGISTER OF THE
                      !CONTROL UNIT
STACK_IN<8:0>;        !INPUT OF THE STACK POINTER


SYNONYM ADDRESS_DATA<6:0>=INPUT_DATA1<6:0>;

ROUTINE PROGRAM_COUNTER;
    IF RESET EQL 1 THEN
        PC_OUT=0
    ELSE IF JUMP_IN EQL 1 THEN
        PC_OUT=COMMAND
    ELSE IF FLAG6_IN EQL 1 THEN
        PC_OUT=PC_IN+1
    ELSE IF COMMAND EQL 3 THEN
        PC_OUT=STACK_IN
    ELSE IF COMMAND EQL 63 THEN
        PC_OUT=PC_IN
    ELSE
        PC_OUT=PC_IN+1;
ENDROUTINE;


ROUTINE HLT;
    IF FLAG6_IN EQL 1 THEN
        READY=0
    ELSE IF COMMAND EQL 63 THEN
        READY=1
    ELSE
```

```
        READY=0;
ENDROUTINE;


ROUTINE OVERFLOW;
    IF RESET EQL 1 THEN
        OVRFOW=0
    ELSE IF FLAG6_IN EQL 1 THEN
        OVRFOW=0
    ELSE IF (COMMAND EQL 46) AND (DIVOVFOW EQL 1) THEN
        OVRFOW=1
    ELSE IF ADDOVFOW EQL 1 THEN
        OVRFOW=1
    ELSE
        OVRFOW=0;

!NOTE: THE OVERFLOW LINE OF THIS ROUTINE WILL BE CONNECTED TO A
!D_FLIP_FLOP AND THE CLOCK OF THE FLIP_FLOP WILL BE CONTROLED
!BY RESET AND Q (OR Q̄) OF THE FLIP_FLOP.

    IF (RESET EQL 1) OR (OVCON EQL 0) THEN
        OVRFOW_EN=0
    ELSE
        OVRFOW_EN=1;
ENDROUTINE;


ROUTINE STACK;
    IF FLAG6_IN EQL 1 THEN
        STACK_OUT=STACK_IN
    ELSE IF COMMAND EQL 1 THEN
        STACK_OUT=PC_IN+2
    ELSE
        STACK_OUT=STACK_IN;
ENDROUTINE;


ROUTINE FLAG;
    IF FLAG6_IN EQL 1 THEN
        FLAG1_OUT=FLAG1_IN
    ELSE IF COMMAND EQL 48 THEN
        FLAG1_OUT=1
    ELSE IF COMMAND EQL 49 THEN
        FLAG1_OUT=0
    ELSE
        FLAG1_OUT=FLAG1_IN;
    IF FLAG6_IN EQL 1 THEN
        FLAG2_OUT=FLAG2_IN
    ELSE IF COMMAND EQL 56 THEN
        FLAG2_OUT=1
```

```
    ELSE IF COMMAND EQL 57 THEN
        FLAG2_OUT=0
    ELSE
        FLAG2_OUT=FLAG2_IN;
    IF FLAG6_IN EQL 1 THEN
        FLAG3_OUT=FLAG3_IN
    ELSE IF COMMAND EQL 58 THEN
        FLAG3_OUT=1
    ELSE IF COMMAND EQL 59 THEN
        FLAG3_OUT=0
    ELSE
        FLAG3_OUT=FLAG3_IN;
    IF FLAG6_IN EQL 1 THEN
        FLAG4_OUT=FLAG4_IN
    ELSE IF COMMAND EQL 60 THEN
        FLAG4_OUT=1
    ELSE IF COMMAND EQL 61 THEN
        FLAG4_OUT=0
    ELSE
        FLAG4_OUT=FLAG4_IN;
ENDROUTINE;


ROUTINE JUMP;
  IF FLAG6_IN EQL 1 THEN
        JUMP_OUT=0
  ELSE IF (COMMAND EQL 1) OR (COMMAND EQL 54) TIIEN
        JUMP_OUT=1
  ELSE IF (COMMAND EQL 2) AND (ADDRESS NEQ ADDRESS_DATA) THEN
        JUMP_OUT=1
  ELSE IF (COMMAND EQL 47) AND (BIG_NUM EQL 1) THEN
        JUMP_OUT=1
  ELSE IF (COMMAND EQL 52) AND (INPUT_DATA1 LEQ INPUT_DATA2) THEN
        JUMP_OUT=1
  ELSE IF (COMMAND EQL 53) AND (INPUT_DATA1 LSS INPUT_DATA2) THEN
        JUMP_OUT=1
  ELSE IF (COMMAND EQL 55) AND (INPUT_DATA1 EQL INPUT_DATA2) THEN
        JUMP_OUT=1
  ELSE IF (COMMAND EQL 64) AND (FLAG1_IN EQL 1) THEN
        JUMP_OUT=1
  ELSE IF (COMMAND EQL 65) AND (FLAG2_IN EQL 1) THEN
        JUMP_OUT=1
  ELSE IF (COMMAND EQL 66) AND (FLAG3_IN EQL 1) THEN
        JUMP_OUT=1
  ELSE IF (COMMAND EQL 67) AND (FLAG4_IN EQL 1) THEN
        JUMP_OUT=1
  ELSE
        JUMP_OUT=0;
ENDROUTINE;
```

```
ROUTINE IMMEDIATE_ADDRESS;
    IF FLAG6_IN EQL 1 THEN
        FLAG5_OUT=0
    ELSE IF COMMAND EQL 62 THEN
        FLAG5_OUT=1
    ELSE
        FLAG5_OUT=0;
    IF FLAG5_IN EQL 1 THEN
        ADDESS_OUT=COMMAND
    ELSE
        ADDESS_OUT=ADDESS_IN;
    IF FLAG5_IN EQL 1 THEN
        ADDESS_EN=0

!IMPORTANT NOTE: THIS ENABLE LINE IS IN FACT WRITE ENABLE OF
!THE REGISTER ADDESS

    ELSE
        ADDESS_EN=1;
ENDROUTINE;


ROUTINE FLAG_6;
    IF FLAG6_IN EQL 1 THEN
        FLAG6_OUT=0
    ELSE
    SELECT COMMAND FROM
        [1,2,47,52,53,54,55,62,64,65,66,67]:FLAG6_OUT=1;
        [OTHERWISE]:FLAG6_OUT=0;
    ENDSELECT;
ENDROUTINE;


ROUTINE NEXT_COMMAND;
    IF FLAG5_IN EQL 1 THEN BEGIN
        CON=DONT_CARE;
        WZ_ENABLE=11#2;
        SHIFT_LOAD=DONT_CARE;
        ENABLE=111111#2;
        MS=0;
        RN_W=DONT_CARE;
        L_CN=1;
        UN_D=DONT_CARE;
        COUNT_EN=1;
        P1_0N=DONT_CARE;
        SN=1;
        C1WN=1;
        COUNT1_EN=0;
        C1L_CN=DONT_CARE;
        C1UN_D=DONT_CARE;
```

```
            TEMP_REG=10#2;
            CONREG1_EN=1;
            CONREG2_EN=1;
            END
ELSE IF FLAG6_IN EQL 1 THEN BEGIN
            CON=DONT_CARE;
            WZ_ENABLE=11#2;
            SHIFT_LOAD=DONT_CARE;
            ENABLE=111111#2;
            MS=0;
            RN_W=DONT_CARE;
            L_CN=DONT_CARE;
            UN_D=DONT_CARE;
            COUNT_EN=0;
            P1_0N=DONT_CARE;
            SN=DONT_CARE;
            C1WN=1;
            COUNT1_EN=0;
            C1L_CN=DONT_CARE;
            C1UN_D=DONT_CARE;
            TEMP_REG=10#2;
            CONREG1_EN=1;
            CONREG2_EN=1;
            END
ELSE BEGIN
            SELECT COMMAND FROM
                        [0,1,2,3,46,47,48,49,52,53,54,55,56,57,58,59,60,
                        61,62,63,64,65,66,67]:BEGIN
                        CON=DONT_CARE;
                        WZ_ENABLE=11#2;
                        SHIFT_LOAD=DONT_CARE;
                        ENABLE=111111#2;
                        MS=0;
                        RN_W=DONT_CARE;
                        L_CN=DONT_CARE;
                        UN_D=DONT_CARE;
                        COUNT_EN=0;
                        P1_0N=DONT_CARE;
                        SN=DONT_CARE;
                        C1WN=1;
                        COUNT1_EN=0;
                        C1L_CN=DONT_CARE;
                        C1UN_D=DONT_CARE;
                        TEMP_REG=10#2;
                        CONREG1_EN=1;
                        CONREG2_EN=1;
                        END;
                        [4]:BEGIN
                        CON=DONT_CARE;
                        WZ_ENABLE=11#2;
```

```
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=1;
UN_D=DONT_CARE;
COUNT_EN=1;
P1_0N=DONT_CARE;
SN=0;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[5]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=0;
UN_D=0;
COUNT_EN=1;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[6]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=1;
RN_W=0;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=1;
SN=DONT_CARE;
C1WN=1;
```

```
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[7]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=1;
RN_W=0;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=1;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=00#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[8]:BEGIN
CON<0>=0;
CON<1>=0;
CON<2>=0;
CON<3>=DONT_CARE;
WZ_ENABLE=00#2;
SHIFT_LOAD=01#2;
ENABLE=111100#2;
MS=1;
RN_W=0;
L_CN=0;
UN_D=0;
COUNT_EN=1;
P1_0N=1;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
```

```
[9]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=1;
RN_W=0;
L_CN=0;
UN_D=0;
COUNT_EN=1;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[10]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=1;
RN_W=0;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=0;
CONREG2_EN=1;
END;
[11]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=011111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
```

```
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=0;
CONREG2_EN=1;
END;
[12]:BEGIN
CON<0>=0;
CON<1>=0;
CON<2>=0;
CON<3>=DONT_CARE;
WZ_ENABLE=00#2;
SHIFT_LOAD=00#2;
ENABLE=111100#2;
MS=1;
RN_W=0;
L_CN=0;
UN_D=0;
COUNT_EN=1;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[13]:BEGIN
CON<0>=0;
CON<1>=0;
CON<2>=0;
CON<3>=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111101#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
```

```
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=11#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[14]:BEGIN
CON<0>=0;
CON<1>=0;
CON<2>=0;
CON<3>=DONT_CARE;
WZ_ENABLE=00#2;
SHIFT_LOAD=01#2;
ENABLE=111100#2;
MS=1;
RN_W=0;
L_CN=0;
UN_D=0;
COUNT_EN=1;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[15]:BEGIN
CON<0>=0;
CON<1>=0;
CON<2>=0;
CON<3>=DONT_CARE;
WZ_ENABLE=00#2;
SHIFT_LOAD=01#2;
ENABLE=111100#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=11#2;
CONREG1_EN=1;
```

```
CONREG2_EN=1;
END;
[16]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=110111#2;
MS=1;
RN_W=0;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[17]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=00#2;
SHIFT_LOAD=10#2;
ENABLE=111111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[18]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=011011#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
```

```
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[19]:BEGIN
CON<0>=1;
CON<1>=1;
CON<2>=DONT_CARE;
CON<3>=DONT_CARE;
WZ_ENABLE=10#2;
SHIFT_LOAD=01#2;
ENABLE=111111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[20]:BEGIN
CON<0>=1;
CON<1>=0;
CON<2>=DONT_CARE;
CON<3>=DONT_CARE;
WZ_ENABLE=10#2;
SHIFT_LOAD=01#2;
ENABLE=111111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
```

```
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[21]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=100111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[22]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111011#2;
MS=1;
RN_W=0;
L_CN=0;
UN_D=1;
COUNT_EN=1;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[23]:BEGIN
CON<0>=0;
```

```
CON<1>=1;
CON<2>=DONT_CARE;
CON<3>=1;
WZ_ENABLE=10#2;
SHIFT_LOAD=01#2;
ENABLE=111111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=0;
UN_D=1;
COUNT_EN=1;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[24]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=011111#2;
MS=1;
RN_W=1;
L_CN=0;
UN_D=0;
COUNT_EN=1;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[25]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=011111#2;
MS=1;
RN_W=1;
L_CN=0;
UN_D=0;
```

```
COUNT_EN=1;
P1_0N=1;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[26]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=1;
RN_W=0;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[27]:BEGIN
CON<0>=0;
CON<1>=0;
CON<2>=0;
CON<3>=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111101#2;
MS=1;
RN_W=0;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
```

```
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[28]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=1;
RN_W=1;
L_CN=0;
UN_D=0;
COUNT_EN=1;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=11#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[29]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=101111#2;
MS=1;
RN_W=1;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=1;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[30]:BEGIN
CON<0>=0;
CON<1>=0;
CON<2>=0;
CON<3>=DONT_CARE;
WZ_ENABLE=11#2;
```

```
SHIFT_LOAD=DONT_CARE;
ENABLE=111101#2;
MS=1;
RN_W=0;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=1;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[31]:BEGIN
CON<0>=0;
CON<1>=0;
CON<2>=0;
CON<3>=DONT_CARE;
WZ_ENABLE=00#2;
SHIFT_LOAD=00#2;
ENABLE=111100#2;
MS=1;
RN_W=0;
L_CN=0;
UN_D=0;
COUNT_EN=1;
P1_0N=1;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[32]:BEGIN
CON<0>=0;
CON<1>=0;
CON<2>=0;
CON<3>=DONT_CARE;
WZ_ENABLE=00#2;
SHIFT_LOAD=00#2;
ENABLE=101100#2;
MS=0;
RN_W=DONT_CARE;
```

```
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[33]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=0;
UN_D=1;
COUNT_EN=1;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[34]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=010111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
```

```
CONREG1_EN=1;
CONREG2_EN=1;
END;
[35]:BEGIN
CON<0>=0;
CON<1>=0;
CON<2>=0;
CON<3>=DONT_CARE;
WZ_ENABLE=00#2;
SHIFT_LOAD=00#2;
ENABLE=011100#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[36]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=1;
RN_W=0;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=1;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=1;
C1L_CN=1;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[37]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
```

```
ENABLE=111111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=1;
UN_D=DONT_CARE;
COUNT_EN=1;
P1_0N=DONT_CARE;
SN=1;
C1WN=0;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[38]:BEGIN
CON<0>=0;
CON<1>=1;
CON<2>=DONT_CARE;
CON<3>=0;
WZ_ENABLE=10#2;
SHIFT_LOAD=01#2;
ENABLE=111111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[39]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=1;
RN_W=1;
L_CN=0;
UN_D=0;
COUNT_EN=1;
P1_0N=1;
```

```
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=11#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[40]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=1;
C1L_CN=0;
C1UN_D=1;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[41]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=1;
C1L_CN=0;
C1UN_D=0;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[42]:BEGIN
```

```
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=011111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=DONT_CARE;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=1;
C1L_CN=1;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[43]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=011111#2;
MS=0;
RN_W=DONT_CARE;
L_CN=1;
UN_D=DONT_CARE;
COUNT_EN=1;
P1_0N=DONT_CARE;
SN=1;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[44]:BEGIN
CON<0>=0;
CON<1>=0;
CON<2>=0;
CON<3>=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111101#2;
MS=1;
RN_W=0;
L_CN=0;
```

```
UN_D=0;
COUNT_EN=1;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[45]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=101111#2;
MS=1;
RN_W=1;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
CONREG2_EN=1;
END;
[50]:BEGIN
CON=DONT_CARE;
WZ_ENABLE=11#2;
SHIFT_LOAD=DONT_CARE;
ENABLE=111111#2;
MS=1;
RN_W=0;
L_CN=DONT_CARE;
UN_D=DONT_CARE;
COUNT_EN=0;
P1_0N=0;
SN=DONT_CARE;
C1WN=1;
COUNT1_EN=0;
C1L_CN=DONT_CARE;
C1UN_D=DONT_CARE;
TEMP_REG=10#2;
CONREG1_EN=1;
```

```
                                    CONREG2_EN=0;
                                    END;
                                    [51]:BEGIN
                                    CON=DONT_CARE;
                                    WZ_ENABLE=11#2;
                                    SHIFT_LOAD=DONT_CARE;
                                    ENABLE=011111#2;
                                    MS=0;
                                    RN_W=DONT_CARE;
                                    L_CN=DONT_CARE;
                                    UN_D=DONT_CARE;
                                    COUNT_EN=0;
                                    P1_0N=0;
                                    SN=DONT_CARE;
                                    C1WN=1;
                                    COUNT1_EN=0;
                                    C1L_CN=DONT_CARE;
                                    C1UN_D=DONT_CARE;
                                    TEMP_REG=10#2;
                                    CONREG1_EN=1;
                                    CONREG2_EN=0;
                                    END;
                                    [OTHERWISE]:BEGIN
                                    CON=DONT_CARE;
                                    WZ_ENABLE=DONT_CARE;
                                    SHIFT_LOAD=DONT_CARE;
                                    ENABLE=DONT_CARE;
                                    MS=DONT_CARE;
                                    RN_W=DONT_CARE;
                                    L_CN=DONT_CARE;
                                    UN_D=DONT_CARE;
                                    COUNT_EN=DONT_CARE;
                                    P1_0N=DONT_CARE;
                                    SN=DONT_CARE;
                                    C1WN=DONT_CARE;
                                    COUNT1_EN=DONT_CARE;
                                    C1L_CN=DONT_CARE;
                                    C1UN_D=DONT_CARE;
                                    TEMP_REG=DONT_CARE;
                                    CONREG1_EN=DONT_CARE;
                                    CONREG2_EN=DONT_CARE;
                                    END;
                ENDSELECT;
            END;
        ENDROUTINE;
ENDMODEL;
```

# Bibliography


[1] ANDERSEN, N. - On the calculation of filter coefficients for maximum entropy spectral analysis. Geophysics, Vol. 39, pp. 69-72, Feb. 1974.

[2] AKAIKE, A. - A new look at the statistical model identification. IEEE Trans. Autom. Control, Vol. AC-19, P.P. 716-723, Dec. 1974.

[3] AKAIKE, A. - Fitting autoregressive models for prediction. Ann. Inst. Statist. Math., 1969, 21, pp. 243-247.

[4] ANTONAKOS, J.L. - The 68000 microprocessor; hardware and software principles and applications. 2nd ed., Merrill, New York, 1993.

[5] BURG, J.P. - Maximum entropy spectral analysis. Proceedings of the 37th meeting of the society of exploration geophysicists, Oklahoma, 1967.

[6] BURG, J.P. - A new analysis technique for time series data. NATO Advanced study institute on signal processing with emphasis on underwater acoustics. Aug. 12-23, 1968.

[7] BOOTH, A.D. - A signed binary multiplication technique. Quart. J. Mech. Appl. Math., Vol. 4, pp. 236-240, 1951.

[8] CHASSIANG, R. - Digital signal processing with C and the TMS320c30. John Wiley & Sons, Inc. 1992.

[9] COHEN, A. - Biomedical signal processing; Vol. 1, Time and frequency domains analysis. CRC Press, Florida, 1986.

[10] EWING, G., MAZUMDAR, J., VOJDANI, B., GOLDBLATT, E., VOLLENHOVEN, V. - A comparative study of the maximum entropy method and the fast fourier transform for the spectral analysis of the third heart sound in children. Australian physical & engineering sciences in medicine. Vol. 9 No. 3, 1986.

[11] EWING, G.J. - A new approach to the analysis of the third heart sound. Thesis submitted for M.Sc. degree. The university of Adelaide, Australia, 1989.

[12] HIGGINS, R.J. - Digital signal processing in VLSI. Prentice-Hall, Inc. N.J., 07632, 1990.

[13] HUTCHINS, B.A., PARKS, T.W. - A digital signal processing laboratory using the TMS320C25. Texas Instruments, Prentice-Hall, Englewood Cliffs, N.J. 07632, 1990.

[14] HAYES, J.P., Computer architecture and organization. McGraw-Hill, New York, 1978.

[15] LUISADA, A.A. - The sounds of the normal heart. Warren H. Green, Inc. St. Louis, Missouri, U.S.A., 1972, pp. 32-38.

[16] LACOSS, R.T. - Data adaptive spectral analysis methods. Geophysics, Aug. 1971.

[17] LITTLE, R.C. - Physiology of the heart and circulation. 3rd ed., Year book medical publisher, Chicago, 1985.

[18] MANO, M.M. - Computer engineering hardware design. Prentice-Hall, Inc., N.J. 07632, 1988.

[19] MANO, M.M. - Computer system architecture, Prentice-Hall, Inc., N.J., 1976.

[20] MUROGA, S. - VLSI System design; when and how to design very-large-scale integrated circuits. John Wiley & Sons, Inc., U.S.A., 1982.

[21] MAZUMDAR, J. - An introduction to mathematical physiology and biology. Cambridge university press, 1989.

[22] NAGAMATSU, M., TANAKA, S., MORI, J., HIRANO, K., NOGUCHI, T., HATANAKA, K. - A 15-ns $32 \times 32 - b$ CMOS multiplier with an improved parallel structure. IEEE JSSC, Vol. 25, No. 2, April 1990, pp. 494-497.

[23] PAPOULIS, A. - Signal analysis. McGraw-Hill, New York, 1977.

[24] PROAKIS, J.G., MANOLAKIS, D.G. - Digital signal processing; Principles, Algorithms, and Applications. 2nd ed., Macmillan, New York, 1992.

[25] PUCKNELL, D.A. - Fundamental of digital logic design with VLSI circuit applications. Prentice-Hall, Australia, 1990.

[26] PUCKNELL, D.A., ESHRAGHIAN, K. - Basic VLSI design; system and circuits. 2nd ed., Prentice-Hall, Australia, 1988.

[27] RIDEOUT, V.L. - One-device cells for dynamic random-access memories. IEEE Trans. on Electron Devices, Vol. ED-26, Jun. 1979, pp. 839-852.

[28] STRACKEE, J., WESTERHOF, N. - The physics of heart and circulation. Institute of physics, Philadelphia, PA, 1993, pp. 207-219.

[29] SHORT, K.L. - Microprocessors and programmed logic. Prentice-Hall, Inc. N.J. 07632, 1981.

[30] STONE, H.S., LOOMIS, H.H. - Introduction to computer architecture. 2nd ed., Science Research Associates, Inc., U.S.A., 1980, pp. 63-71.

[31] SMITH, J.J. - Circulatory physiology; the essentials. 2nd ed.,Williams & Wilkins, 1984.

[32] SHANNON, C.E., WEAVER, W. - The mathematical theory of communication. University of Illinois press, 1949.

[33] TORTORA, G.J., ANAGNOSTAKOS, N.P. - Principles of anatomy and physiology. 5th ed., Harper & Row, 1987, pp. 461-468.

[34] TILKIAN, A.G., CONOVER, M.B. - Understanding heart sounds and murmurs with an introduction to lung sounds. 2nd ed., W.B. Saunders Company, 1984, pp. 35-70, 89.

[35] TRIEBEL, W.A., SINGH, A. - 16-bit Microprocessors; architecture, software, and interface techniques. Prentice-Hall, N.J., 1985.

[36] ULRYCH, T.J., BISHOP, T.N. - Maximum entropy spectral analysis and autoregressive decomposition. Review of geophysics and space physics, Feb. 1975, Vol. 13, 183-200.

[37] WESTE, N.E., ESHRAGHIAN, K., - Principles of CMOS VLSI design; A system perspective. 2nd ed., Addison-Wesley, 1993.

[38] WALLACE, C.S. - A suggestion for fast multipliers. IEEE Trans. Electron. Comput. Vol. EC-13, Feb. 1964, pp. 14-17.

[39] WASER, S., FLYYN, M.J. - Introduction to arithmetic for digital systems designers. CBS college publishing, 1982.

[40] YAMAUCHI, H., NIKADO, T., NAKASHIMA, T., KOBAYASHI, Y., SAKAI, T. - 10 ns $8 \times 8$ multiplier LSI using super self-aligned process technology. IEEE JSCC, Vol. SC-18, No. 2, April 1983, pp. 204-210.

[41] YANO, K., YAMANAKA, T., NISHIDA, T., SAITO, M., SHIMOHIGASHI, K., SHIMIZU, A. - A 3.8-ns CMOS $16 \times 16 - b$ multiplier using complementary pass transistor logic. IEEE JSSC, Vol. 25, No. 2, April 1990, pp. 388-394.

[42] YAMAGUCHI, K., NAMBU, H., KANETANI, K., IDEI, Y., HOMMA, N., HIRAMOTO, T., TAMBA, N., WATANBE, K., ODAKA, M., IKEDA, T., OHHATA, K., SAKURAI, Y. - A 1.5-ns access time, 78-um$^2$ memory-cell size, 64-kb ECL-CMOS SRAM. IEEE JSSC, Vol. 27, No. 2, Feb. 1992, pp. 167-174.