Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

An On-Line Approach For Evaluating Trigonometric Functions

by
Rajesh Amin

This thesis investigates the evaluation of trigonometric functions based on an on-line arithmetic approach. On-line algorithms have been developed to evaluate the sine and cosine functions. Error analysis and heuristics are carried out to arrive at a minimal error algorithm based on the series expansion of the sine and cosine function.

A logical design based on the algorithm is presented where the unit is designed as a set of basic modules. A detailed bit slice design of each module is also presented. A simulator was designed as an experimental tool for synthesis of the on-line algorithms, and a tool for performance evaluation.

AN ON-LINE APPROACH FOR EVALUATING TRIGONOMETRIC FUNCTIONS

by
Rajesh Amin

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

January 1994

APPROVAL PAGE

AN ON-LINE APPROACH FOR EVALUATING TRIGONOMETRIC FUNCTIONS

Rajesh Amin

Dr. Edwin Hou, Thesis Adviser — Date
Assistant Professor of Electrical & Computer Engineering, NJIT

Dr. Sotirios Ziavras, Committee Member — Date
Assistant Professor of Electrical & Computer Engineering, NJIT

Dr. M. Zhou, Committee Member — Date
Assistant Professor of Electrical & Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Rajesh Amin

Degree: Master of Science in Electrical Engineering

Date: January 1994

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1994

- Bachelor of Science in Electronics and Communications Engineering,
  Maharaja Sayajirao University, Baroda, India, 1991

Major: Electrical Engineering

This thesis is dedicated to my sister

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my advisor Dr. Edwin Hou for his guidance and motivation throughout this research, without whom this thesis could not have been conceived. His insight, comments, criticisms and suggestions added greatly to the final result. Thanks especially for his support of the thesis writing and for his many readings of it, at different stages.

Special thanks to Dr. Ziavras for his help and advice which made it possible for me to reach higher standards. I also would like to thank him for leading me through my masters program. My thanks also goes to Dr. Zhou for serving as a member of committee.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Introduction

"It has been long recognized that the concept of computer architecture is no longer restricted to the structure of bare machine hardware"[1]. A modern computer is an integrated system consisting of machine hardware, an instruction set, system software, application programs, and user interfaces. Present day computing is driven by real life problems which require fast and accurate solution.

Most scientific problems require extensive numerical computations and their solutions demand complex mathematical formulations and numerous fixed or floating point computations. Most complex problems such as weather forecasting, structural analysis, and random problems in navigation can be transferred to arithmetic computation problems using well known techniques [2]. The resources required by these massive problems are the driving factor that necessitate the enhancement of current day arithmetic units. To satisfy the demands of present day computational problems, more efficient implementation of arithmetic units and faster computational algorithms are needed. Numerous hardware techniques have been introduced, such as parallel processing, pipelining etc.[1]. These techniques are utilized to develop high speed and efficient arithmetic units based on architectures such as superscalar or superpipelining, where one or more results can be obtained in each clock cycle. However, the basic arithmetic pipeline structures are limited by the time required to add or shift operands. Methods such as carry look ahead addition or carry save addition have been developed to alleviate the carry propagate

bottleneck of addition[3]. However, these parallel implementations still require that both operands must be completely resided in the registers before the computation can start. This limitation can be eliminated by a fast evolving technique called on-line arithmetic which uses serial addition instead of conventional parallel arithmetic.

On-line arithmetic is a process for performing arithmetic on a serial basis. All on-line arithmetic processors accept inputs and generate outputs in a most significant digit first format. Redundant number representations [4] are used for the digits to avoid carry propagation in addition. These methods allow the arithmetic unit to produce partial results starting from most significant bits of the input operands. That is, for every bit of input, you produce an output after a small delay. On-line arithmetic processors can be pipelined directly to perform complicated calculations and with their serial dataflow characteristic, they may be internally pipelined at rates limited only by the time required to calculate a single digit. Elimination of carry propagation allows on-line operations to be overlapped. Application specific systems benefit especially because arithmetic operations can be overlapped by starting operations as soon as digits become available from previous operations.

1.2 Objectives

The main objective of this thesis is the development of iterative algorithms for the computation of trigonometric functions such as, cosine and sine, based on on-line arithmetic techniques. Specifically, the algorithm is to be digit on-line algorithm and the on-line delay of this algorithm is to be at the most four. Delay should be limited to this value for efficient pipeline implementation of the algorithm.

Secondly, a modular logic design of the underlying hardware is presented. This provides us with an understanding of the hardware requirements and form the basis of comparison with other algorithms on similar functions.

Finally, a software simulation of the algorithm is presented. This simulation would act as an acid-test to verify the correctness of the algorithm.

1.3 Thesis Overview

The rest of the thesis is organized as follows. Chapter 2 reviews some basic concepts and background materials on the on-line algorithms. The CORDIC procedure for evaluating trigonometric functions and a comparison with the on-line approach is discussed in chapter 3. The implementation of CORDIC is also briefly discussed in this chapter. Chapter 4 deals with the error analysis and heuristics carried out to achieve an optimal series approximation. The hardware implementation along with a logic design, for the implementation of the trigonometric functions are presented in chapter 5. Chapter 6 discusses the software simulation of the on-line algorithm.

CHAPTER 2

BACKGROUND

2.1 Redundant Number System

2.1.1 Signed Digit Number System (Radix > 2)

In a conventional number system with an integer radix r>1 each digit is allowed to assume exactly r values: 0, 1, . . . , r-1. In a redundant number representation with the same radix r each digit is allowed to assume more than r values.

Avizienis[6] described a method where each digit of a positional constant radix number with an integer radix r is allowed to assume q values,

$$r+2 \leq q \leq 2r-1 \tag{2.1}$$

This is possible because both positive and negative digit values are allowed. Redundancy in the number representation allows fast addition and subtraction in which each sum digit is the function only of the digits in two adjacent digital positions of the operands. Such operations are called totally parallel addition and subtraction. The requirements for totally parallel addition and subtraction and a unique representation for the zero value are satisfied by a class of redundant representations with radices r > 2 which are called signed digit representations. Each digit of a number in signed digit representation can assume both positive and negative integer values. The number of digit values in a radix r > 2 representation ranges from a required minimum of r+2 to the allowable maximum of 2r-1.

The purpose of signed digit representation is to allow addition and subtraction of two numbers where no carry propagation is required; that is, the time for the operation is independent of the length of the operands and is equal to the time required for the addition or subtraction of two digits. A signed digit

number is represented by n+m+1 digits z_i (i= -n, . . . , -1, 0, 1, . . ., m) and has the value Z as shown in equation 2.2.

$$Z = \sum_{-n}^{m} z_i \, r^{-i} \tag{2.2}$$

Consider the addition of two digits, z_i, y_i, the sum digit $s_i = f(z_i, y_i, t_i)$ where t_i is the transfer digit from the (i+1) th position on the right and $t_{i-1} = f(z_i, y_i)$. The addition of the two digits is performed in two successive steps. First, an outgoing transfer digit t_{i-1} and an interim sum digit w_i are formed:

$$z_i + y_i = rt_{i-1} + w_i, \tag{2.3}$$

then the sum digit s_i is formed from,

$$s_i = w_i + t_i. \tag{2.4}$$

The requirement for the unique representation of zero is satisfied by the condition,

$$| z_i | \leq r-1. \tag{2.5}$$

For a two operand operation, the condition(equation 2.5) establishes values for $t_i = \{-1,0,1\}$ and the condition $| w_i | \leq r-2$ sets the upper limit for the magnitude of the interim sum (this also restricts the radix to $r > 2$). The relationship between the greatest value w_{max} and smallest value w_{min} of w_i is

$$w_{max} - w_{min} \geq r - 1, \tag{2.6}$$

and the set of allowable values for w_i is unique and consists of 2r-3 integers from -(r-2) to (r-2).

Since $t_i \in \{ -1, 0, 1\}$, the required values of the sum digit s_i consists of a sequence of r+2 integers:

$$s_i \in \{w_{min}-1, w_{min}, \ldots, -1, 0, 1, 2, \ldots, w_{max}, w_{max}+1\}). \tag{2.7}$$

For odd radix r_o, the minimum required set is,

$$z_i \in \{ -(r_o + 1)/2, \ldots, -1, 0, 1 , \ldots, (r_o + 1)/2\}, \tag{2.8}$$

for even radix r_e, minimum required set is,

$$z_i \in \{-(r_e/2 + 1), \ldots, -1, 0, 1, \ldots, r_e/2 + 1\}. \quad (2.9)$$

Also, since $| z_i | \leq r-1$ we can have different sets of $z_i \in \{-a, -(a+1), .., -1, 0, 1, ..(a-1), a\}$, where, $(r_o + 1)/2 \leq a \leq r_o - 1$ or $r_e/2 + 1 \leq a \leq r_e - 1$. If $a = r_o - 1$ or $a = r_e - 1$, then there is maximum redundancy, and if $a = (r_o+1)/2$ or $a = r_e/2+1$, then condition exists for minimum redundancy.

The following gives an example for addition of two signed digit numbers as the totally parallel addition of all corresponding digits.

Example: Signed Digit Addition (Radix = 10)

The allowed digit values for t_i and w_i are:

ti: -1, 0, 1 and,

w_i: 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -6.

$$t_i = \begin{cases} 0 & \text{if } -6 < (z_i + y_i) < 5 \\ 1 & \text{if } (z_i + y_i) > 5 \\ -1 & \text{if } (z_i + y_i) < -6 \end{cases}$$

$Z = 1.\bar{3}65\bar{1}\bar{4}$ value = 0.76486

$Y = 0.\bar{4}053\bar{1}$ value = -0.39471

The addition procedure S = Z + Y based on equations (2.3) and (2.4) is illustrated in the following table where $\bar{x} \equiv -x$.

Table 1: Addition Procedure for Radix 10.

i	0	1	2	3	4	5
augend z_i	1.	-3	6	5	-1	-4
addend y_i	0.	-4	0	5	3	-1
step 1 ($rt_{i-1} + w_i$)	0+1	-10+3	10+(-4)	10+0	0+2	0+(-5)
step2 (t_i)	-1	1	1	0	0	
sum s_i	0.	4	-3	0	2	5

The sum is $S = 0.4\bar{3}02\bar{5}$; value = 0.37015

2.1.2 Modified Signed Digit Representations (Radix = 2)

The digit addition rules for signed digit may be modified to allow the propagation of the transfer digit over two digital position to the left[4]. If this type of transfer addition is allowed the radix r = 2 may be used and only r + 1 values are required for the sum digit. Two transfer additions are executed in three steps

$$z_i' + y_i' = rt_{i-1}' + w_i' , \tag{2.10}$$

$$w_i' + t_i' = rt_{i-1}'' + w_i'' , \tag{2.11}$$

$$s_i' = t_i'' + w_i'' , \tag{2.12}$$

the digits z_i , y_i , and s_i are digits of a modified signed digit representation. For example, with radix r = 2, the required values are -1, 0, 1;

- if no redundancy exists and each sum digit s_i is the function of all the addend digits z_i and augend digits y_i to the right, i.e.

  $s_i = f(z_i , y_i , z_{i+1} , .., z_m , y_m)$;
- if each sum digit assumes r + 1 values, we have $s_i = f(z_i , y_i , z_{i+1}, y_{i+1} , z_{i+2} , y_{i+2})$ and the operations are two transfer additions;
- if each sum digit assumes r + 2 values or more, then we have $s_i = f(z_i , y_i , z_{i+1}, y_{i+1})$ and the operation is a single transfer addition.

Example: Signed Digit Addition (Radix = 2)

The allowed digit values are:

$w_i' , w_i'' \in \{-1, 0, 1\}$; $t_i' , t_i'' \in \{-1, 0, 1\}$

$S = Z + Y$

$$t_i' = \begin{cases} 0 & \text{if } -1 \le (z_i + y_i) \le 1 \\ 1 & \text{if } (z_i + y_i) > 1 \\ -1 & \text{if } (z_i + y_i) < -1 \end{cases} \qquad t_i'' = \begin{cases} 0 & \text{if } -1 \le (w_i' + t_i') \le 1 \\ 1 & \text{if } (w_i' + t_i') > 1 \\ -1 & \text{if } (w_i' + t_i') < -1 \end{cases}$$

$Z = 1.\bar{1}01\bar{1}1$ binary value = 0.1001

$Y = 0.10\bar{1}0\bar{1}$ binary value = 0.0011

The procedure of addition based on equations (2.10)-(2.12) is illustrated in the following table:

Table 2: The Radix-2 Addition Procedure.

i	-1	0	1	2	3	4	5
augend z		1.	-1	0	1	-1	0
addend y		0.	1	-1	-1	1	0
$z_i + y_i$		1	0	-1	0	0	0
t_i'	0	0	0	0	0	0	0
w_i'		1	0	-1	0	0	0
t_i''	0	0	0	0	0	0	0
w_i''	0	1	0	-1	0	0	0
s_i	0	1	0	-1	0	0	0

The sum is $S = 1.0\bar{1}000;$ binary value = 0.11000

2.2 On-Line Arithmetic

2.2.1 Overview

On-line arithmetic is a process for performing arithmetic on a serial (i.e. digit by digit) basis. All on-line arithmetic processors accept inputs and generate outputs in a most significant digit first format. To obtain the jth digit of the result from an on-line algorithm, it is necessary and sufficient to have the operands available up to the $(j + \delta)^{th}$ digit. The index difference δ, called on-line delay(typically 1 to 4), is a small positive constant and it is algorithmic dependent.

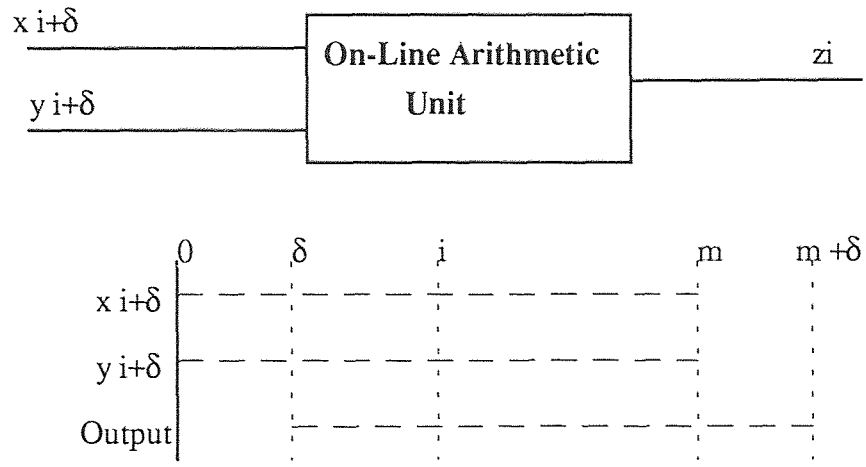

Figure 1 Input/Output Characteristics of On-Line Arithmetic

As indicated in Figure 1, the first digit of the output is produced after a delay δ. Subsequently, one digit of the result is produced upon receiving one digit of each of the operands and m is the precision of the result.

Figure 2 demonstrates the difference in evaluation time between on-line and conventional arithmetic for an expression:

$$a = [\ (p+q)^{1/2} * (m - o)^2\ ] + (x\ /\ y) \tag{2.13}$$

It is easy to perceive that a conventional (non-redundant) number system is not feasible for on-line arithmetic. If we were to use a non-redundant number system, then even for simple operations like addition and subtraction, there is an on-line delay δ = m due to carry propagation. Hence it is required for an on-line algorithm to use redundant numbers where the time required to compute one output digit is independent of the length of the input operands. The on-line representation of a number A is defined as

$$A_j = A_{j-1} + a_{j+\delta}\ r^{-\delta - j} \tag{2.14}$$

and,

$$A_0 = \sum_{i=1}^{\delta} a_i r^{-i} \tag{2.15}$$

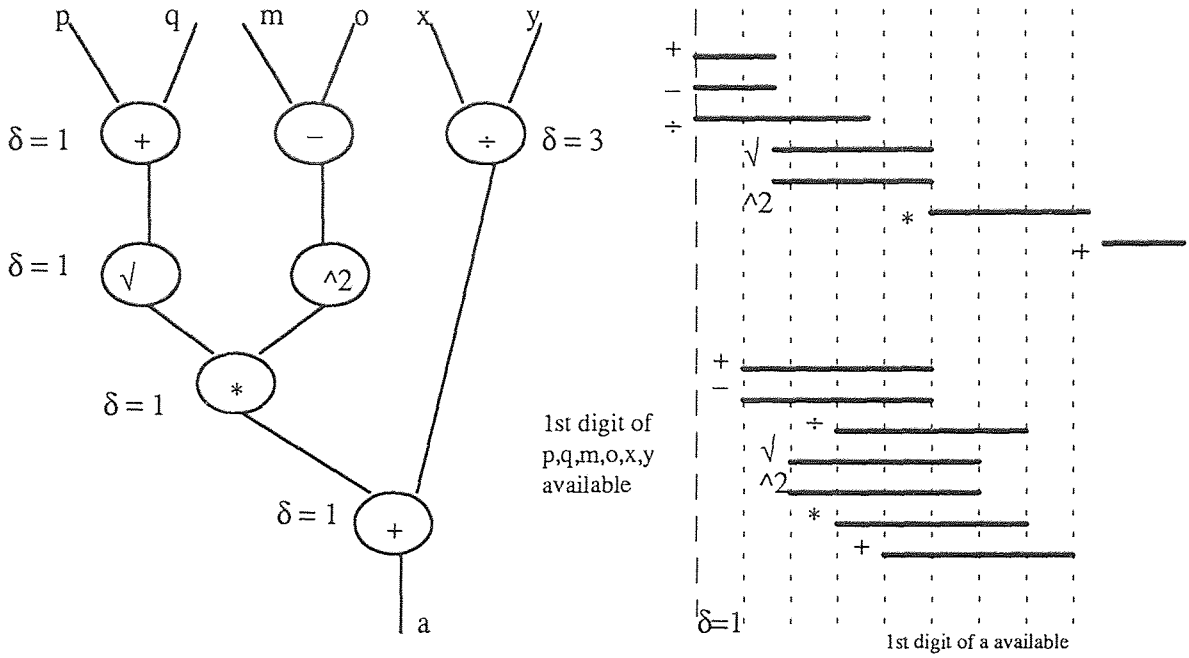

Figure 2 Comparison of Evaluation Time: On-Line and Conventional

The digits a_i belong to a redundant digit set $\{-\mu, \ldots., -1, 0, 1, \ldots., \mu)$ where $r/2 \leq \mu \leq r-1$ determines the amount of redundancy.

Example: For binary data the redundant digit set consists of {-1, 0, 1}.

2.2.2 Fixed Point On-Line Addition

Presented below is an example of an on-line add algorithm for adding X_k and Y_k [5].

Let $X_k = X_{k-1} + x_k.r^{-k}$ and $Y_k = Y_{k-1} + y_k.r^{-k}$ denote the values of the addends, while $S_k = S_{k-1} + s_k.r^{-k}$ denotes the value of the sum, at step k in a radix-r redundant number system.

[1] Initialization:

$w_j = 0 \;\; ; \;\; s_{-2} = s_{-1} = 0$

[2] Recurrence:

for j = 0, 1, 2, 3,. . . . . , m+1 do:

$w_j = r\,(w_{j-1} - s_{j-2}) + r^{-\delta}\,(x_j + y_j)$

[3] Selection Function:

$s_j = S(w_j)$

where the selection function S is done by rounding

$$S(w_j) = \begin{cases} \text{sign}(w_j)\,\lfloor\, |(w_j)| + 1/2 \,\rfloor & \text{if } |(w_j)| \le \mu\,; \\ \text{sign}(w_j)\,\lfloor\, |(w_j)| \,\rfloor & \text{otherwise} \end{cases}$$

The on-line delay is $\delta = 1$ for r >= 4 and μ = r-1. For r = 2, $\delta = 2$.

2.2.3 Fixed Point On-Line Multiply

We now give an example of an on-line multiply algorithm for multiplying X_k and Y_k [5].

Let $X_k = X_{k-1} + x_k r^{-k}$ and $Y_k = Y_{k-1} + y_k r^{-k}$ denote the values of the multiplier and multiplicand, while $D_k = D_{k-1} + d_k r^{-k}$ denotes the value of the partial product, at step k in a radix-r redundant number system.

[1] Initialization:

$w_j = 0\ ;\ x_0 = y_0 = 0\ ;\ d_0 = 0$

[2] Recurrence:

for j = 0,1, 2, 3,. . . . . , m+1 do:

$w_j = r\,(w_{j-1} - d_{j-1}) + (X_j y_j + Y_{j-1} x_j)$

[3] Selection Function:

$d_j = S(w_j) = \text{sign}\, w_j .\, \lfloor |w_j| + 1/2 \rfloor$

where the selection function S is done by rounding and after m steps the product is $P = XY = D_m + (W_m - d_m)r^{-m}$.

2.2.4 Implementation

On-line arithmetic algorithms can be implemented in two ways:

1. Linear array organization of on-line units and
2. Pipelined implementation.

In a linear array organization of on-line units, inputs are given parallel to each module where input operands are divided in sub operands depending on the precision of the internal digits of each module. As the input operand digits start coming to the first module(most-significant bits) it calculates the partial output and also passes the transition result to the next module. Hence the unit starts producing output result as soon as the input operand bits are available (plus a small delay), unlike the conventional computation where the unit has to wait for all the bits of the operand and then starts the computation.

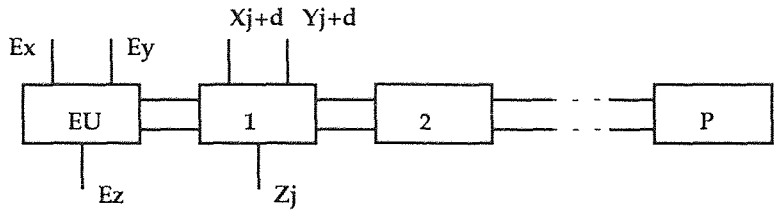

Figure 3 Linear Array Organization of On-Line Unit

The pipelined implementation is similar to the linear array implementation but here the input operands are given in a pipelined fashion instead of single input operation as in the linear array. A pipelined on-line unit consists of $(n+\delta)$ stages with the stage delay t_d. In the steady state, the unit is computing up to n different results and the last stage producing the last digit of the $(i-n)^{th}$ result. To implement the recurrence of an on-line algorithm, the working precision that increases with the number of steps must be provided. If the result is to be computed to a maximum precision of n digits, the recurrence requires at the jth

step a precision of the j digits for $j < n/2$ and a precision of n-j digits for $j \geq n/2$. Therefore, n simultaneous operations in various stages of completion require a total working precision of about $n^2/4$ digits. This requires that a one-dimensional array of modules, shown in figure 3, would not be suitable for pipelined inputs since the modules (their internal precision) and the inter-module bandwidth would depend on the relative position in the array. A two dimensional array typical for pipelined inputs is shown in the figure 4.

This array, if implemented with d digit wide modules, requires $\lceil n/d \rceil$ rows with a variable number of modules per row. The total number of d-digit modules required for maximum precision of n digits is approximately $(n/d)^2/4$.

2.2.5 Characteristics and Features

The following depicts some of the important characteristics and features of on-line arithmetic :

1. Produces results most significant digit first.
2. Digit cycle time is independent of data wordlength.
3. Higher computational rates by allowing overlap at the digit level between successive operations.
4. Variable precision.
5. Overlapped operand alignment in floating point operations.
6. Error control.
7. Minimum interconnection complexity between the processing units - one digit per operand.
8. Low interconnection bandwidth.
9. Modularity.
10. High concurrency.
11. VLSI realizability.

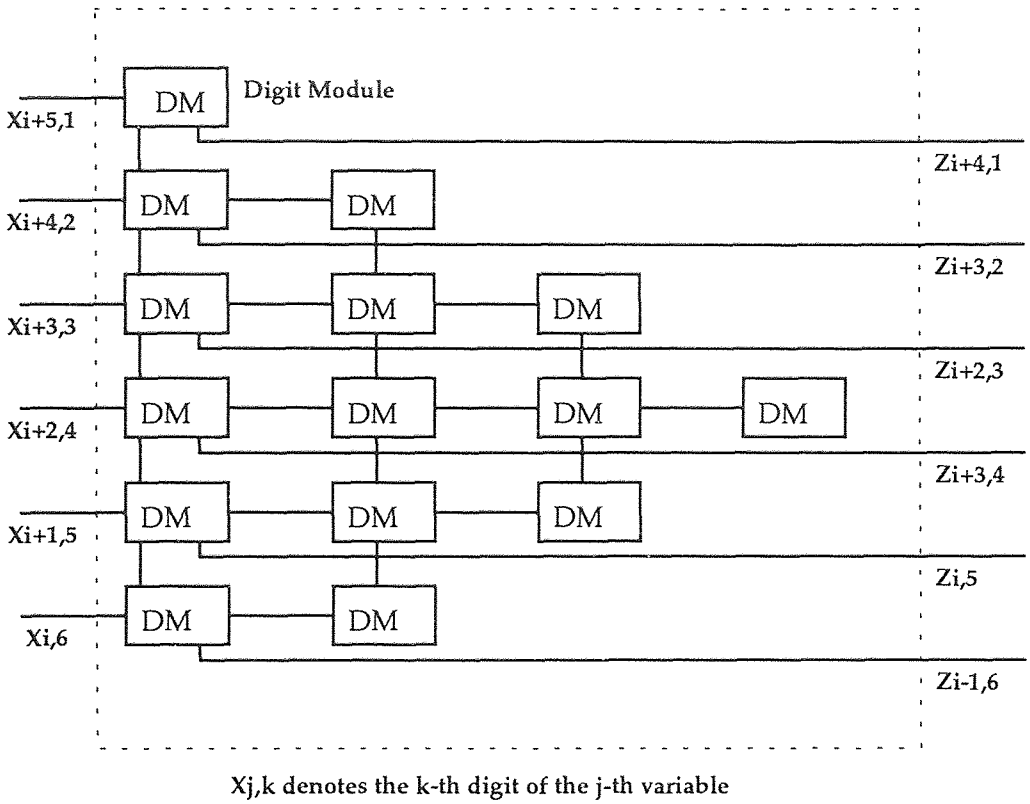

Figure 4 Two-Dimensional Pipelined On-Line Unit (n= 5, d = 1, δ = 1)

On-line arithmetic is highly attractive in high speed multi-module structures for parallel and pipelined computations. Compared to conventional arithmetic where high speed multi-operand processing requires full precision bandwidth between arithmetic units, on-line requires a bandwidth of only one digit per operand which presents a very feasible and cost effective alternative. Also due to its highly modular characteristics it can be easily realized in terms of VLSI design. The main results indicate that the on-line approach offers a speed-up factor of 2 to 16 with respect to conventional arithmetic while preserving limited interconnection bandwidth, decentralized control, and uniform structure. These features are highly attractive for reconfigurable networks.

The principal disadvantage lies in the fact that the use of redundant number system is mandatory where conversions to and from the conventional system is an overhead. Moreover the inherently serial operation makes on-line arithmetic unsuitable for isolated arithmetic operations and comparisons.

CHAPTER 3

TRIGONOMETRIC FUNCTION EVALUATION

3.1 The CORDIC Procedure

3.1.1 CORDIC Algorithm

The COordinate Rotation DIgital Computer (CORDIC) algorithm to evaluate trigonometric functions was first introduced by Volder[6]. Consider the problem of rotating a vector R(r, β), where r is the magnitude and β is the angle made by the vector with the positive X-axis, through a specified positive angle ϕ. Assume that the original vector is expressed in terms of its coordinates X andY and we wish to find the coordinates X' and Y' of the rotated vector.

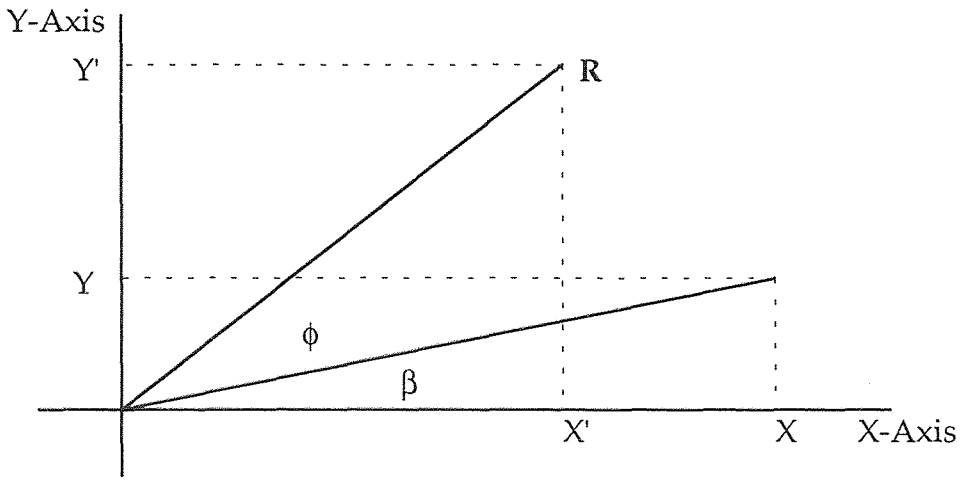

Figure 5 Coordinate Rotation of Vector R

From figure 5, we have,

$$X' = X\cos\phi - Y\sin\phi$$

$$Y' = X\sin\phi - Y\cos\phi \qquad (3.1)$$

and,

$$X'/\cos\phi = X - Y\tan\phi$$

$$Y'/\sin\phi = Y + X\tan\phi \qquad (3.2)$$

The CORDIC algorithm is an iterative procedure, in which each step, a vector is rotated in one or the other direction through an angle $a_j = \tan^{-1} 2^{-i}$. The direction for each a_j is chosen as positive (anti-clockwise) if $(\phi - \sum a_j) > 0$ and as negative(clockwise) if $(\phi - \sum a_j) < 0$ where j = 0 to i -1. Thus if we let ϕ_i denote the total angle through which the vector has been rotated through step i then, $\phi_i = \pm a_0 \pm a_1 \pm a_2 \ldots . \pm a_i$. This sum converges for all angles of magnitude less than 100° (1.74 radians). The iterative rotation through $\pm a_i$ can be used to compute X' and Y' iteratively with

$$X_{i+1} = X_i - Y_i \tan a_i$$

$$Y_{i+1} = Y_i - X_i \tan a_i. \qquad (3.3)$$

If $(\phi - \sum a_j) < 0$ then,

$$X_{i+1} = X_i + Y_i * 2^{-i}$$

$$Y_{i+1} = Y_i - X_i * 2^{-i}, \qquad (3.4)$$

and if $(\phi - \sum a_j) > 0$

$$X_{i+1} = X_i - Y_i * 2^{-i}$$

$$Y_{i+1} = Y_i + X_i * 2^{-i}. \qquad (3.5)$$

Continuation of the iterations until ϕ_i approximates ϕ therefore produces X and Y components which are Kc times as large as the true X' and Y', where Kc is given by

$$\begin{aligned} Kc &= 1/\cos a_0 * 1/\cos a_1 * 1/\cos a_2 \ldots \ldots \\ &= \sqrt{(1 + 2^{-0})} * \sqrt{(1 + 2^{-1})} * \sqrt{(1 + 2^{-2})} \ldots \ldots \\ &= 1.646760255\ldots \end{aligned} \qquad (3.6)$$

which is the Volders Value through i = 24. Division by this factor forms the true X' and Y' values from the X and Y values formed by repeated application of equations (3.3)-(3.5).

3.1.2 Implementation

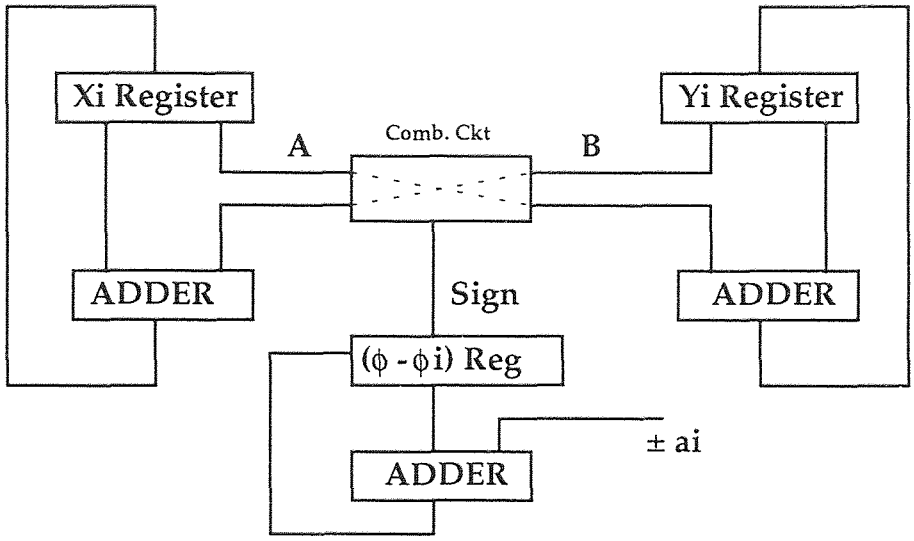

Figure 6 Implementation of CORDIC

The hardware requirements for performing these steps are quite simple. We need a shift register for X_i and Y_i and two adders. We also need to accumulate $\phi - \phi_i$ and to test its sign. Figure 6 shows the block diagram of these elements. The crossover paths A and B carry the shifted (and possibly complemented) values of X_i and Y_i to be added or subtracted as specified by equations (3.3)-(3.5). The sign of a_i is determined by the sign bit of the $(\phi - \phi_i)$ register, and the various values of a_i must be provided on successive steps. These values are stored in ROM.

The structure of the algorithm can be easily adapted for other purposes besides the rotation described above. Some of them are:

- Given a vector (r,ϕ), we find its components X' and Y' by setting Xo = r and Yo = 0 and rotating through ϕ.

- Given a vector (r,ϕ), we find $\sin\phi$ and $\cos\phi$ by setting Xo = 1 and Yo = 0 and rotating through ϕ, to form $X' = \cos\phi$ and $Y' = \sin\phi$ (after multiplying by scaling factor 1/Kc).
- Given an angle ϕ, we find $\tan\phi$ by forming $\sin\phi$ and $\cos\phi$ as shown previously and then finding their quotient.
- Given the components X and Y, we find (r,ϕ) by choosing successive a_i so that Y_{i+1} becomes zero as the rotation continues. Then $r = X_{i+1}/Kc$ and $\phi = \phi_i$, the accumulated sum of $\pm a_i$.
- Given X_1 and Y_1, to find $\phi = \tan^{-1} Y_1/X_1$, set $X = X_1$ and $Y = Y_1$. Rotate the vector by choosing signs for successive a_i so that Y_{i+1} becomes zero. Then $\phi = \phi_i$.

Larger values must be prescaled to reduce them to size for which the procedure converges.

3.2 On-Line Trigonometric Functions Over Conventional CORDIC

As discussed in section 3.1 the CORDIC method [6] is straightforward and easy to implement. Its principal advantage is that it requires neither floating point nor multiplication hardware. Also it requires only a small number of operations. However, the conventional implementation of a CORDIC module has a lot more disadvantages when compared to modern techniques.

- The first advantage which on-line implementation of trigonometric functions has over CORDIC method is that the usage of redundancy greatly improves the speed. That is because the time required to compute one output digit is independent of the length of operands.
- Secondly, CORDIC requires variable shifting which is time consuming and expensive, this can be eliminated or alleviated using an on-line approach.

- Thirdly, the on-line approach does not require table lookup, which is a major bottleneck while the CORDIC method requires ROM's to store angle constants. Using ROM's makes the implementation slow and requires large chip area. Moreover, the on-line approach uses a regular implementation of modules such as add, multiply. etc., hence the design is highly modular and interconnections are simpler. This is a highly desired feature in highly complex systems such as matrix triangularization and singular value decomposition (SVD [7]) which uses angle calculation algorithm as their basis. The modularity feature of the on-line approach can be utilized in building systems which are internally pipelined and consequently increases the overall system throughput. They are also highly cost-effective.
- Other features of on-line approach include realization of a variable precision arithmetic. Floating point operand alignment can be performed in overlapping manner with significand operation. Moreover, in contrast to conventional CORDIC method we can control error, i.e. if error occurs during computation at the j^{th} step, we can restrict the precision of result to j and still get a partial output.

CHAPTER 4

ERROR ANALYSIS

4.1 Introduction

In this chapter, we investigate the possibility of using a series expansion approach to evaluate trigonometric functions. The first question that was encountered was whether this method is feasible. This means, we need to evaluate the error produced by a series expansion approach where error is defined as $| \cos(x) - f_{cos}(x) |$. In this chapter, we determine how the size of the input operand and the number of terms in the series expansion is related to this error.

To calculate the error, several programs were written to graph the error of the sine and cosine series from their actual values. At the same time, calculations were carried out to find out the significance of each terms in the series. Also these calculations involve finding out the number of terms needed to be performed (i.e. number of x^n / n! terms to be added) before the output remains unaffected for the number of precision bits considered.

For example, consider the cosine series expansion:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots .+ (-1)^n \frac{x^{2n}}{2n!} + \ldots \qquad (4.1)$$

With a 16 bit input and output precision, the first step would be to find how many terms from the above series would affect the result within the precision. Table 3 provides summarizes the results.

From the table, we selected at most nine terms of the series as an approximation to the cosine function.

facing 22

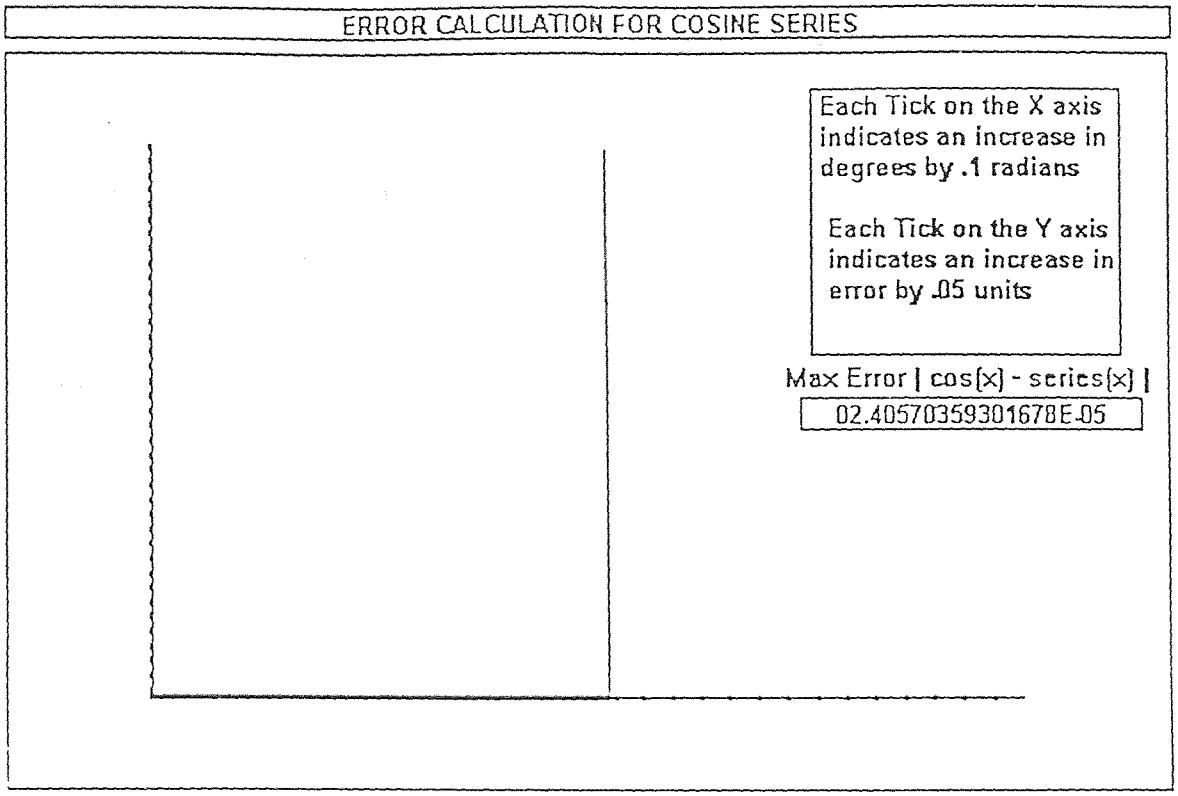

Figure 7 Error Curve For Cosine Series $1-X^2/2! + X^4/4! + (-1)^nX^{2n}/2n + \ldots$

Table 3 Significant Terms Analysis

ANGLE	NUMBER OF TERMS	SERIES
15°	5	$(1 - x^2/2! + \ldots\ldots + x^8/8!)$
30°	7	$(1 - x^2/2! + \ldots\ldots + x^{12}/12!)$
45°	8	$(1 - x^2/2! + \ldots\ldots + x^{14}/14!)$
60°	9	$(1 - x^2/2! + \ldots\ldots + x^{16}/16!)$

The next step is to take the truncated series and analyze its error at different values (0 to 90 degrees). The curve in figure 7 shows how the error $| \cos(x) - f_{cos}(x) |$ takes a constant path curve with the error remaining more or less constant between 0 to 90 degrees. This indicates the behavior of the series expansion in different input regions. The maximum error from the curve is 0.000024057036.

After determining the characteristics of the series approach and its feasibility, we carried out analysis (see Appendix C) to derive a series which has the denominators as only powers of two.

For example, in our case for the cosine function we determine the series as approximation

$$f_{\cos}(x) = 1 - \frac{x^2}{2^1} + \frac{x^4}{2^4} - \frac{x^6}{2^7} . \qquad (4.2)$$

To prove that this is a good approximation, we determine the characteristic of this series with an error analysis as before. Furthermore, routines were written to graph the deviation error $| x - f^{-1}_{cos}(x) |$.

Figure 8 shows a normal error curve ($| \cos(x) - f_{cos}(x) |$) for the equation 4.2. It shows the exponentially increasing behavior, due to heavy truncation on elements of the series plus the modified denominator values.

Figure 9 demonstrates the deviation error characteristics. This means that if the original value for cos(60) is 0.5, then to get 0.5 as output with the modified

facing 23

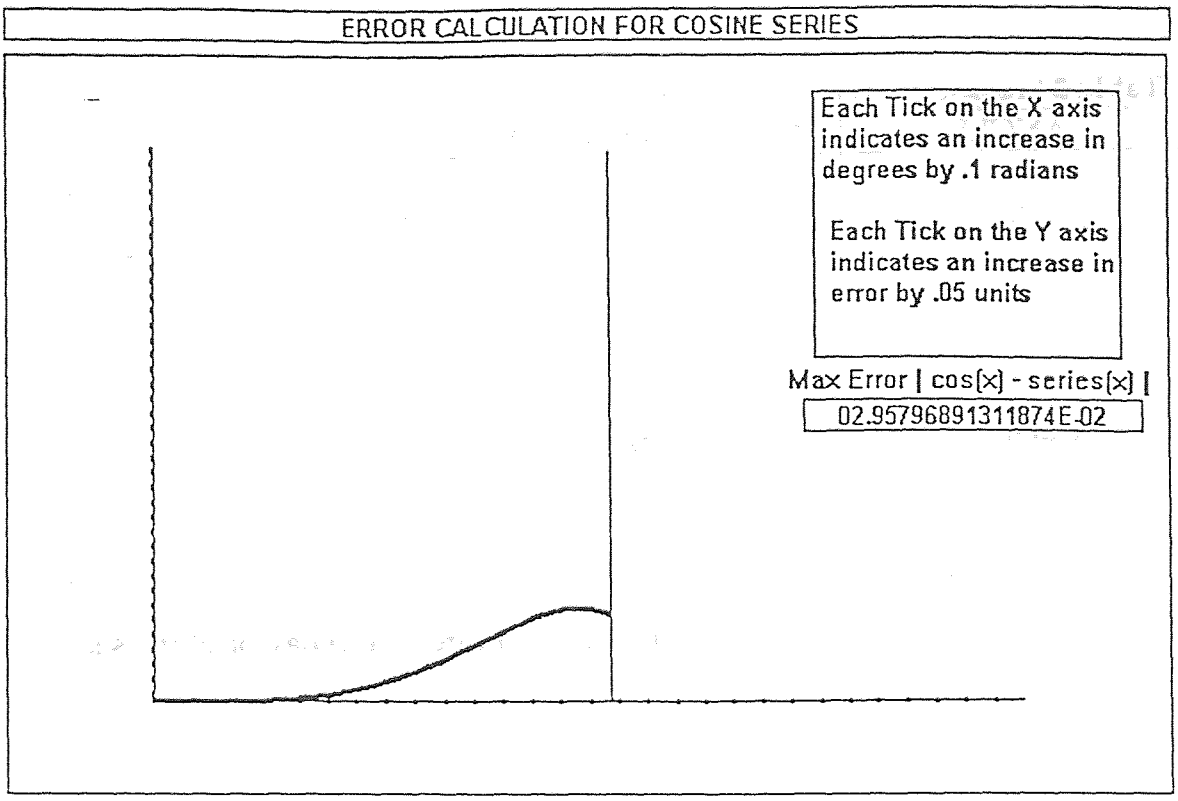

Figure 8 Error Curve For Modified Cosine Series $1-X^2/2 + X^4/16 - X^6/128$

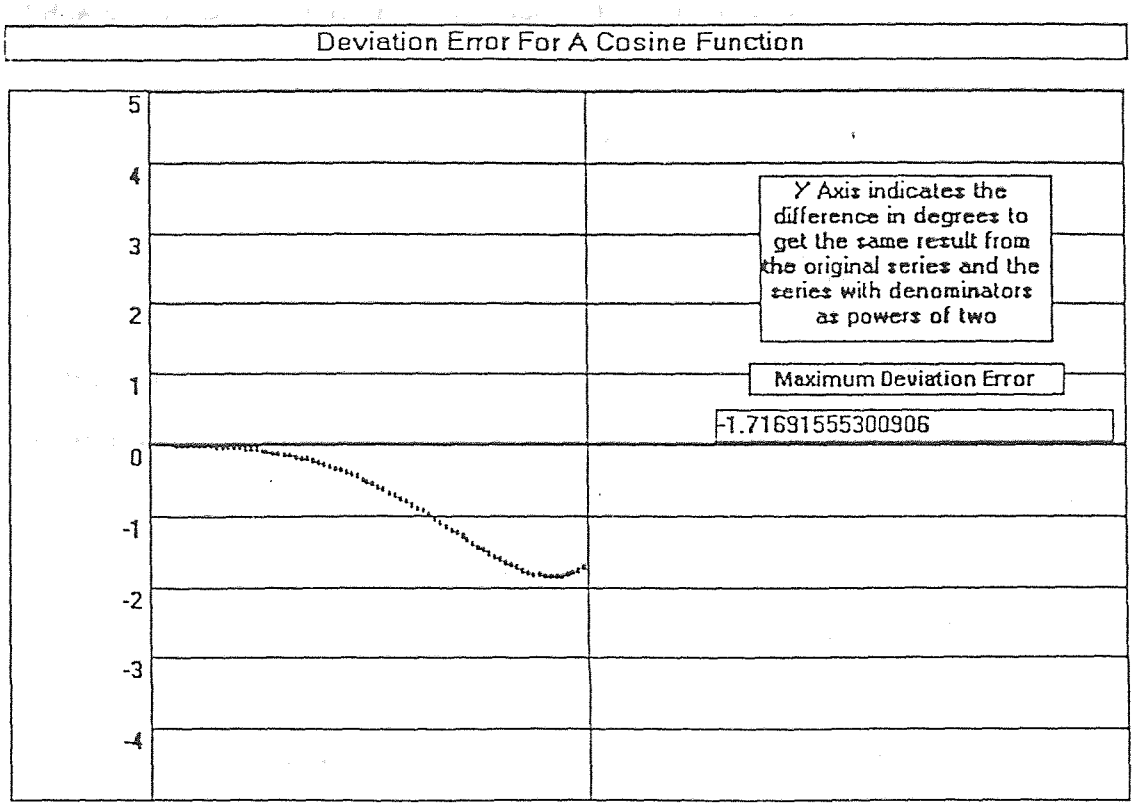

Figure 9 Input Deviation Error Curve For Modified Cosine Series $1-X^2/2 + X^4/16 - X^6/128$

series, an input of 60° + error should be given. This error in terms of degrees, is shown by the graph. Error curves and deviation error curves for different possible series are discussed in following section. The maximum error at 90 degrees is 0.029661147.

4.2 Error Analysis

4.2.1 Error Analysis

The error between the function and the series approximation is defined as $| \cos(x) - f_{cos}(x) |$. Appendix C shows several error curves for different series at different precision values. The curves are plotted for both sine and cosine series.

A close study of these figures indicates several important characteristics of the sine and cosine series implementation :

- Keeping input values between 0 and 30 degrees result in minimal error;
- Input values at or near 90 degrees result in maximal error;
- Changing denominators in the power series result in different error characteristics;
- Changing precision value keeps the characteristic of the error curve approximately the same but the net error changes;
- For cosine series $1 - x^2/2 + x^4/16 - x^6/128$, the maximal error region is at or near 70 degree instead of 90 degrees;
- For sine series $x - x^3/8 + x^5/32 - x^7/16384$, the error curve takes a steep increase, which indicates this series may not converge at all.

4.2.2 Input Deviation Error

The error between the function and the series approximation is defined as $| x - series^{-1}(x) |$. Input deviation error is an indication of how much the input

value is deviated from the correct value to obtain the same results for each function. Input deviation error curves for cosine series at 16 bit precision is shown in figure 9. The X axis indicates the input value in degrees while Y axis indicates the deviation error in degrees.

A close study of the curves(Appendix C) indicates the following important characteristics:

- Keeping input values between 0 and 30 degrees result in minimal input deviation error;
- Input values at or near 90 degrees result in maximal input deviation error;
- Changing denominators in power series result in different error characteristics;
- Changing precision value keeps the characteristic of the error curve approximately the same but the net error changes;
- For cosine series $1 - x^2/2 + x^4/16 - x^6/128$, the maximal error region is at or near 70 degrees instead of 90 degrees;
- For cosine series $1 - x^2/2 + x^4/16 - x^6/512$, the error curve takes a steep increase after around 60 degrees, which indicates this series may not be feasible to use for inputs greater than 60 degrees.

4.2.3 Optimal Series

An optimal series in our case is a series with the best error and input deviation error characteristics. Programs were written which were capable of running heuristic calculations on every possible powers of two, for each term in the series. The following results were obtained:

- Some series may have the smallest error in a small region of input, that is it is optimized for that region of input, but further observation shows that error in other regions are increased;

- Changing the computation precision does not have a major effect on the optimal series denominators;
- The optimal series that can be found under the constraints of available resources and time are,

  $f_{cos}(x) = 1 - x^2/2 + x^4/16 - x^6/128$, maximum normal error of 0.02958

  $f_{sin}(x) = x - x^3/8 + x^5/1024 - x^7/16384$, maximum normal error of 0.09395

CHAPTER 5

HARDWARE IMPLEMENTATION

The general cosine and sine functions in their series forms are respectively,

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^n \times \frac{x^{2n}}{2n!} + \ldots$$
$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + (-1)^n \times \frac{x^{(2n+1)}}{(2n+1)!} + \ldots \quad (5.1)$$

As derived in the chapter 4, the series is truncated and the factorial denominators are approximated by exact powers of two to simplify the hardware implementations. The approximate equations for sine and cosine in equation 5.1 are

$$f_{\cos}(x) = 1 - \frac{x^2}{2^1} + \frac{x^4}{2^4} - \frac{x^6}{2^7}$$
$$f_{\sin}(x) = x - \frac{x^3}{2^3} + \frac{x^5}{2^{10}} - \frac{x^7}{2^{14}} \quad (5.2)$$

An exploded block diagram of the implementation of the cosine series using on-line arithmetic is shown in figure 10. A similar implementation for the sine function using equation 5.2 can be realized without making major changes to the design for the cosine function. The implementation utilizes on-line units capable of doing multiplication, addition (3-operands) and shifts.

Table 4 Truth Table for Y = -X

x1	x0	y1	y0
0	0	0	0
0	1	1	1
1	1	0	1

- The truth table indicates that

$$y0 = x0 \text{ and } y1 = x1 \oplus x0 \quad (5.3)$$

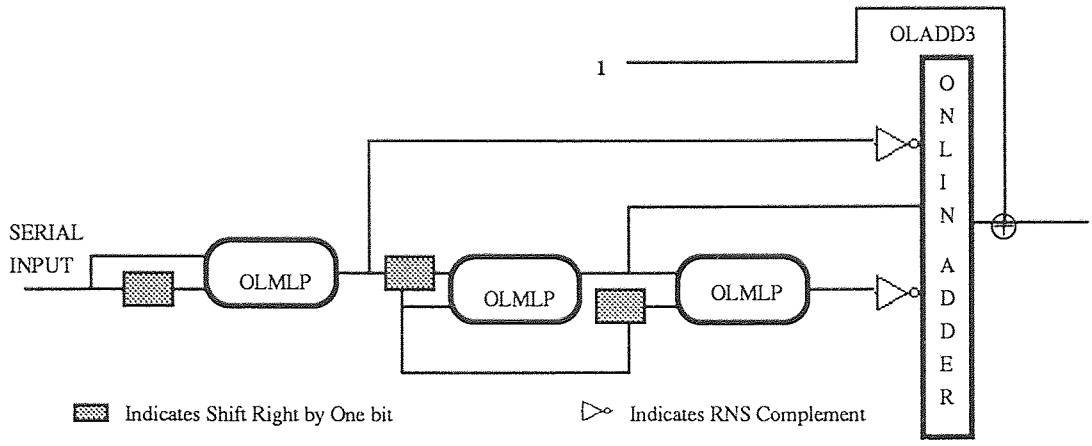

Figure 10 Block Diagram of the On-Line Implementation of the Cosine Function

Each OLMLP block in Figure 10 indicates an on-line multiplier which is capable of multiplying two operands in an on-line fashion. The multiplier blocks are used to form powers of X. Shifters at various input points of the blocks are used for divisions. Subtraction is done by complementing the result bits at the output of the on-line units. Since the number system used is a redundant number system, complementing the result bits is based on the function shown in table 4.

The OLADD3 block is an on-line adder capable of adding three input operands simultaneously. A constant number (1.0) is added directly using a "preappend" method instead of conventional addition.

5.1 Design Considerations

- Inputs are given in redundant number system with working set as {-1,0,1}
- Output is received also as a number in redundant representation.
- Multiplication algorithm works only for inputs less than 1.0 so the inputs are scaled down, by using shifters at their input points.

- A "1" is added at the end by just appending it in front of the result (result will always be within the range [-1,1]).
- The input operand X can be given at each step j as indicated in equation 5.4.

$$X_j = X_{j-1} + x_{j+\delta}\, r^{-\delta-j}$$

$$X_0 = \sum_{i=1}^{\delta} x_i r^{-i}, \tag{5.4}$$

where δ is the on-line delay factor.

Described below is the detailed design of each of the on-line units beginning with OLMLP, the on-line multiplication unit.

5.2 OLMLP

The OLMLP unit is described best by its recurrence equation (equation 5.5), which iteratively calculates the product of the operands X and Y (Appendix B).

$$w_j = 2(w_{j-1} - d_{j-1}) + (X_j y_j + Y_{j-1} x_j). \tag{5.5}$$

The various hardware techniques and components required to implement the above recurrence equation are discussed briefly in the following sections.

5.2.1 Data Conversion

Working with redundant number system operands internally in an accumulator complicates its structure and operations. Hence, we work with conventional binary number system instead of RNS(redundant number system) internally in an accumulator. This requires converting the input operands in RNS to the conventional(2's comp.) numbers (i.e. mapping numbers from (-1,0,1) to (0,1)).

The conversion algorithm is based on Ercegovac [8] and is described briefly in equations 5.6 and 5.7.

m_k = Input bit at time k (redundant form)

$n[k]$ = Output bit (conventional form) at time k

$$n[k] = \begin{cases} P[k-1] + 2^{-k} & \text{if } m_k = 1 \\ P[k-1] & \text{if } m_k = 0 \\ Q[k-1] + 2^{-k} & \text{if } m_k = -1 \end{cases}$$

$$P[k-1] = -n_0 + \sum_{i=1}^{k-1} n_i . 2^{-i} \quad = \quad n[k-1]$$

where, (5.6)

$$Q[k-1] = P[k-1] - 2^{-(k-1)}$$

- **Conversion Algorithm**

(1) Initialization:

$$P[1] = \begin{cases} +1/2 \ \ (0.1) \ \text{if } m > 0 \ \ (m_1 = +1) \\ -1/2 \ \ (1.1) \ \text{if } m < 0 \ \ (m_1 = -1) \end{cases}$$

$$Q[1] = \begin{cases} +0 \ \ (0.0) \ \text{if } m > 0 \ \ (m_1 = +1) \\ -1 \ \ (1.0) \ \text{if } m < 0 \ \ (m_1 = -1) \end{cases}$$

(2) Recurrence:

For k > 1,

$$P[k+1] = \begin{cases} P[k] + 2^{-(k+1)} & m_{k+1} = 1 \\ P[k] & m_{k+1} = 0 \\ Q[k] + 2^{-(k+1)} & m_{k+1} = -1 \end{cases}$$

$$Q[k+1] = \begin{cases} P[k] & m_{k+1} = 1 \\ Q[k] + 2^{-(k+1)} & m_{k+1} = 0 \\ Q[k] & m_{k+1} = -1 \end{cases} \quad (5.7)$$

Adding $2^{-(k+1)}$ indicates a simple operation of appending appropriate bit to P[k] or Q[k].

The above algorithm is based on the well known conditional sum algorithm for fast addition. As indicated by the recurrence equations, we do not

require any kind of carry propagate addition at any stage. The implementation of the algorithm is shown in figure 11.

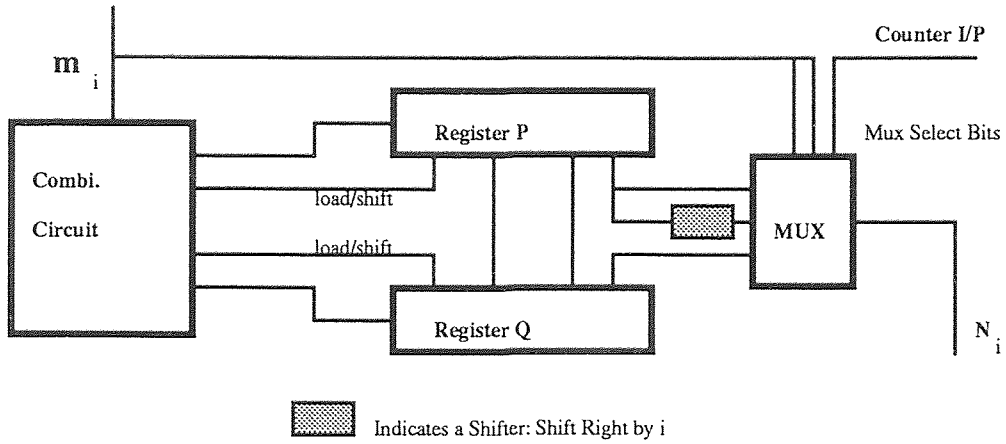

Figure 11 Implementation of the Conversion Algorithm

The implementation requires two registers to hold P[k] and Q[k] per bit slice. A combinational circuit is used to select the inputs of the two registers and a multiplexer decides which output should be given (Figure 11).

A counter input to the multiplexer decides whether the output is from the buffers or from the input itself depending on the count. That is, at the i th clock cycle, the i th bit-slice selects the input directly without taking from the buffers. This represents the appending of the $2^{-(k+1)}$ to the P[k] or Q[k] terms in the recurrence equation (equation 5.7), while other bit slices (1 to i-1) select their outputs from their respective buffer registers depending on the input bit (1,0,-1).

5.2.2 Single Bit Multiplier

Multiplications such as $X_j y_j$ and $Y_{j-1} x_j$ require single bit multiplication between its operands. Such multiplication can be achieved by the following function given in equation 5.8.

for: $Z_j = X_j y_j$ where, $X_j = x^0 x^1 x^2 \ldots\ldots x^j$

$$Z_j = \begin{cases} X_j & \text{if } y_j = 1 \\ 0 & \text{if } y_j = 0 \\ -X_j & \text{if } y_j = -1 \end{cases}$$

hence,

$$z^j = \begin{cases} x^j & \text{if } y_j = 01 \\ 0 & \text{if } y_j = 00 \\ \overline{x^j} & \text{if } y_j = 11 \end{cases} \qquad (5.8)$$

5.2.3 Addition

Addition of the variables in the recurrence equation is done by using the carry save adder technique[9]. Since there are multiple input terms to the adder, we implement the adder as bi-level carry save adders. Actual addition in the design of bit slice is realized by full adders implemented as shown in figure 12.

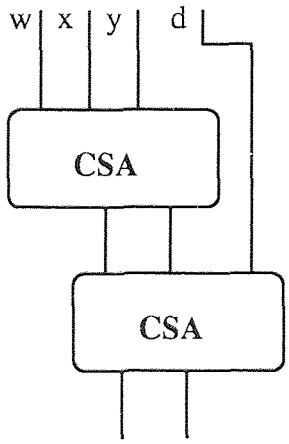

Figure 12 Addition by Carry Save Addition Technique

5.2.4 Result Digit Selection

The result digit selection is done by using a selection function S which is given in equation (5.9), $s_j = S(w_j)$. The selection function S is done by rounding.

$$S(w_j) = \begin{cases} \text{sign}(w_j)\lfloor |(w_j)|+1/2 \rfloor & \text{if } |(w_j)| \le 1 \\ \text{sign}(w_j)\lfloor |(w_j)| \rfloor & \text{otherwise} \end{cases} \quad (5.9)$$

This implies that whenever the intermediate sum W_j is less than 1, then the binary value 0.1 is added to it and a floor rounding is done. If it is greater than 1, then a simple floor rounding is done. This implies the following bitwise input/output relationship between W_j and S_j shown in table 5.

Table 5 indicates that,

$$S_{j-2} = \begin{cases} 1 \quad (01) & \text{when } \overline{w_{-1}}.(w_0 + w_1) \\ -1 \ (11) & \text{when } \overline{w_{-1}}.w_0 \\ 0 \quad (00) & \text{when } s_1 s_0 = (w_2 + w_3 + \ldots\ldots\ldots + w_j) \cdot (w_1 w_0 w_{-1}) + (\overline{w_1}\,\overline{w_0}\,\overline{w_{-1}}) \end{cases} \quad (5.10)$$

Table 5 Truth Table for Selection Function

W_i						s_{i-2}	
w_{-1}	w_0	w_1	w_2	w_3	$w_4 \ldots$	s_0	s_1
0	1	0	*	*	*	0	1
0	0	1	*	*	*	0	1
0	1	1	*	*	*	0	1
0	0	0	*	*	*	0	0
1	0	0	*	*	*	not possible	
1	1	0	*	*	*	1	1
1	1	1	*	*	*	1	1
1	0	1	*	*	*	not possible	
1	1	1	1	*	*	0	0
1	1	1	0	1	*	0	0
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

* indicates the don't care condition.

The selection is independent of the precision of the input w.

Instead of generating logic for all 1,0 and -1 we generate only 1 and -1 and assume 0 as the default output.

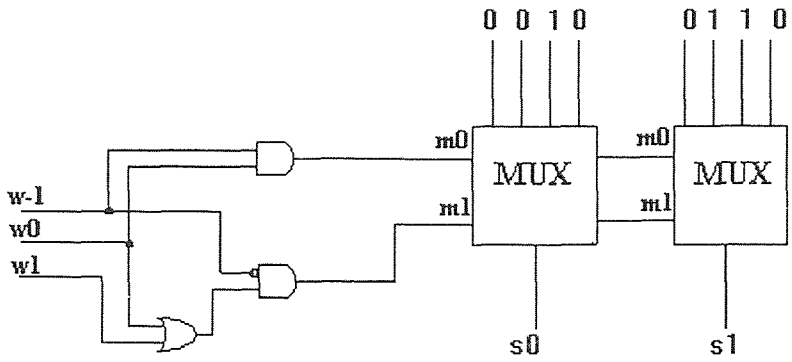

Figure 13 Implementation of Select Function

Figure 13 shows implementation of $S(w_j)$ (equation 5.10) using only w_{-1},w_0, w_1 bit of the input w. The logic used is that if S is not 1 or -1, then for all valid inputs it has to be 0. Hence, only the first three bits of the input are used to select for 1 or -1, otherwise, it is 0 by default.

5.2.5 Pipelining Implementation

To minimize the number of steps of the computation, we use a two stage pipelined scheme which is implemented by using the modified equations(5.11):

$$\begin{aligned} &(1)\ \ w'_j = 2(w'_{j-1} - d_{j-1}) + (X_j y_j + Y_{j-1} x_j) \\ &(2)\ \ d_j = \text{SELECT}(2(w'_{j-1} - d_{j-1})) \end{aligned} \qquad (5.11)$$

Step 1 is done in stage 1 while step 2 is done in stage 2 in the pipeline.

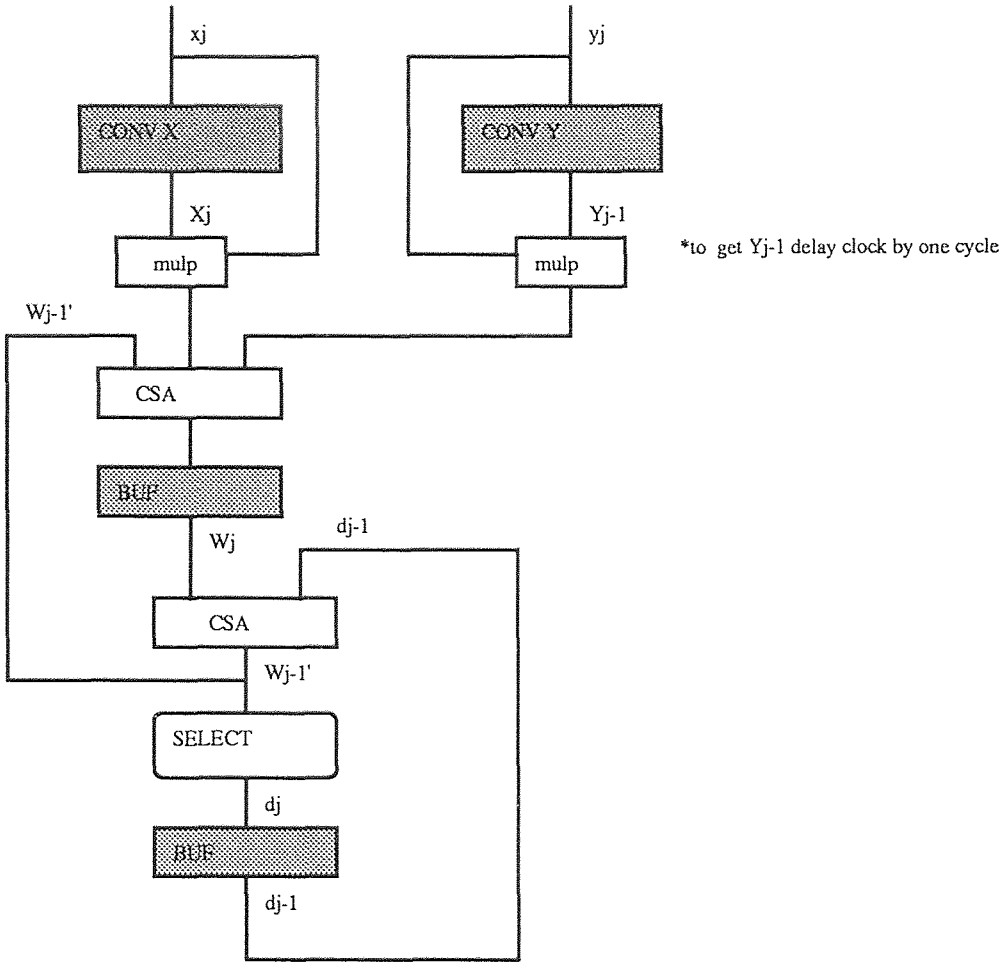

Figure 14 Pipelined Implementation of Multiplication Recurrence

Figure 14 illustrates the pipeline implementation of the multiplication recurrence equation. The select function in equation 5.9 is modified slightly in equation 5.11 to incorporate the two stage pipelining in the design. We have two buffers, one stores the previous value of w'_j while other stores the previous value of dj. The conv. X and conv. Y stages indicate the accumulation and conversion of the input bits from redundant number system to conventional number system. The multiplier stages indicate the single bit multiplier as discussed in equation 5.7. To get Yj-1 at the j th clock cycle, we can put a one-clock cycle delay for conv. Y.

5.2.6 Organization of the On-Line Unit

The on-line unit is organized as a linear array of modules as shown in figure 15.

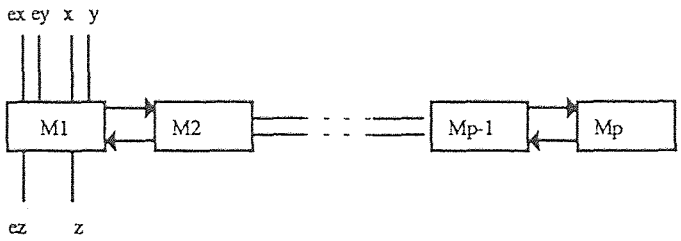

Figure 15 On-Line Linear Array Organization

The first module M1 as shown in figure 15, contains components for generating the result exponent, result digit selection(SELECT) and bit slices for the most significant portion of the recurrence calculation, this includes the sign and integer bit positions and a few fractional bit positions. The other modules M2....Mp are identical in that each consists of bit slices of the fractional bit positions. The number of modules required for a m-bit precision calculation with input of N_{frac} (number of fractional bits) and N_{int} (number of integer bits) is given by equation 5.12.

$$\left\lceil \frac{m}{10} + \frac{\delta + n_{frac}}{2} \right\rceil + n_{int} \qquad (5.12)$$

According to the modular design shown above, the conversion algorithm implementation for our modular pipelined design should also be modular (i.e. bit slices should be able to handle input bit that is coming at each iteration). Hence, our implementation for conversion algorithm should be as shown in figure 16.

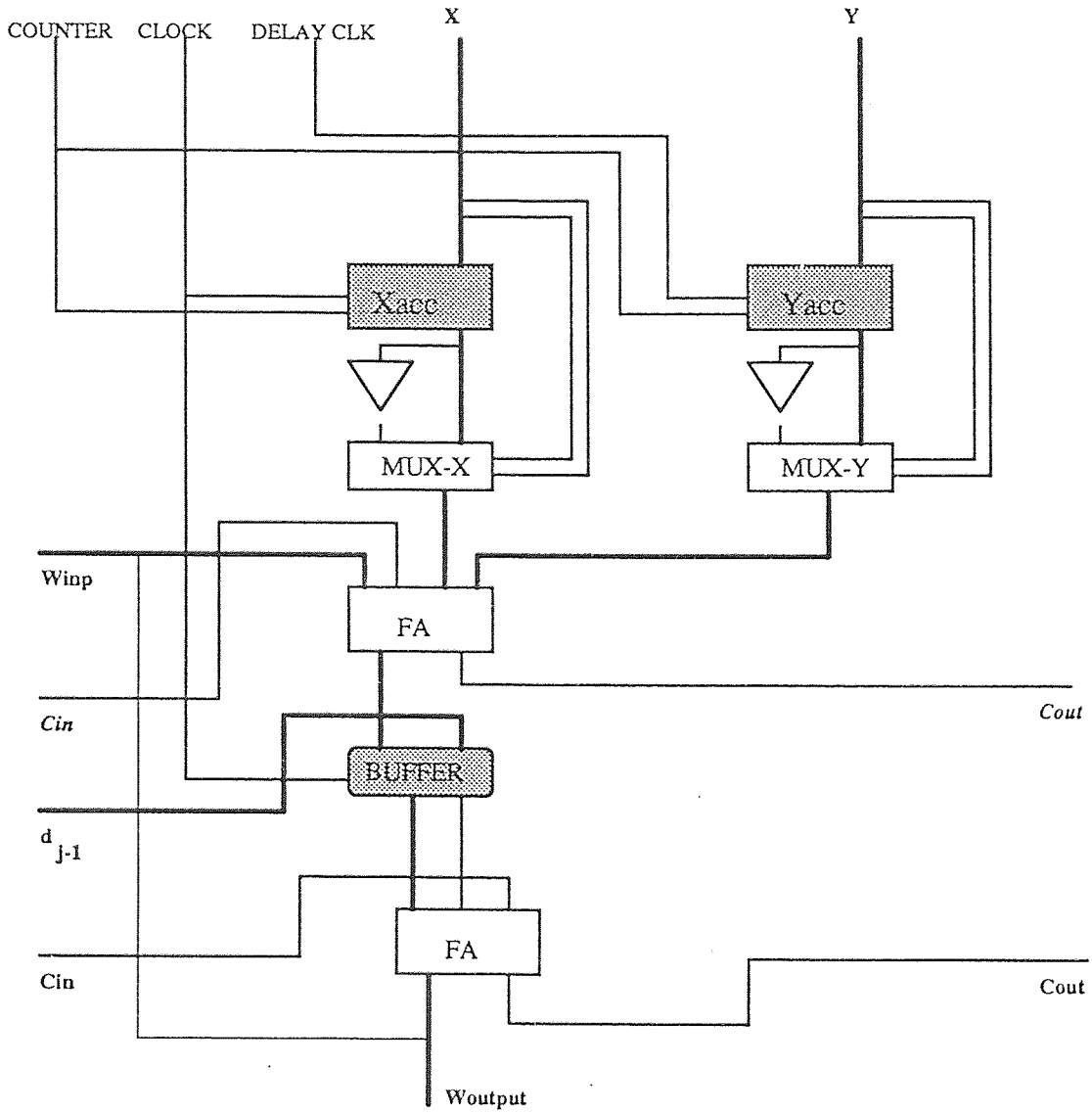

Figure 17 Bit slice Organization of the On-Line Multiply Module

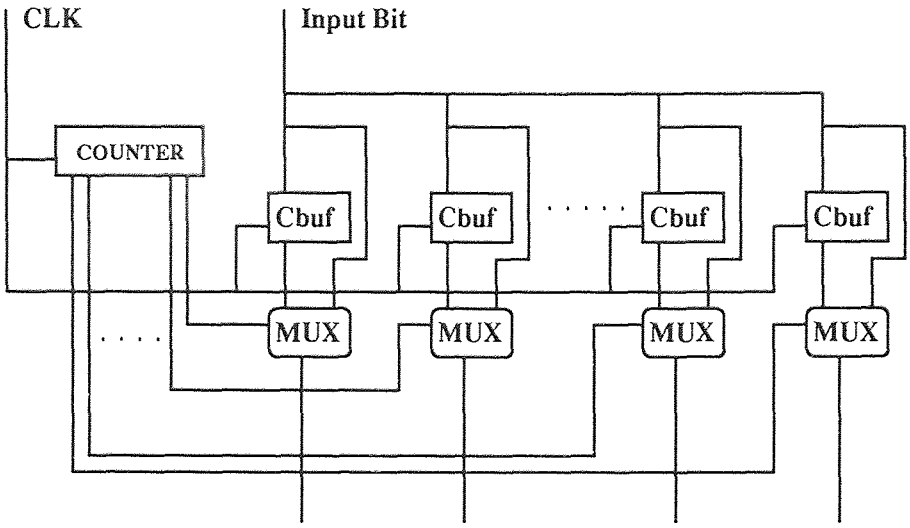

Figure 16 Modular Decomposition of the Conversion Algorithm Implementation

Cbuf is a circuit consisting of two buffers to store P(k) and Q(k) and combinational circuits to form P(k+1) and Q(k+1).

Figure 17 shows a single bit slice organization.

The Xacc and Yacc design goes according to the design of the conversion algorithm as in equations 5.6 & 5.7, and it incorporates the Cbuf along with other necessary logics.

Instead of using two separate buffers as shown in figure 14 we use a common buffer to store w_{j-1} and d_{j-1}.

Notes:

- the clocking circuitry designed latches latches the incoming x_j and y_j bits at the jth clock cycle of the jth bit slice in order to have the correct position of x_j and y_j.
- Also x_j and y_j goes to all bit slices for the single bit(sbit) multiplier, i.e. multiplication of the latched bits in each bit-slice with value of x_j and y_j at time period j.
- inputs x_j and y_j are two bit inputs.

5.3 OLADD3(on-line add recurrence)

The OLADD3 is an on-line unit which can add up to three operands simultaneously in an on-line manner. The OLADD3 is required to sum up all the power series terms as shown in figure 18. It is based on the on-line addition algorithm (see Appendix B).

The on-line add algorithm can be implemented similar to the on-line multiply discussed in section 5.2. The differences are that there is an on-line delay δ at the input for the OLADD3 unit, and that there are three operands internally instead of two. The equations for on-line addition implementation are

$$(1) \quad w_j' = 2(w_{j-1}' - s_{j-3}) + (x_j + y_j + z_j)2^{-2}$$

$$(2) \quad s_{j-2} = sel(w_{j-1}' - s_{j-3}) \tag{5.13}$$

Similar to the on-line multiply case, wj' is used in order to facilitate pipelining between the two stages. Step 1 is implemented in pipeline stage 1, while step 2 is implemented in pipeline stage 2. The select function is modified slightly to incorporate the two stage pipelining in the design. Hence as can be seen from the figure 18 we have two buffers, one stores the previous value of wj' while the other stores the previous value of dj. FA and SHR indicates the unit required for adding bitwise three operands and multiplication by 2^{-2}. The FA is implemented as a RNS full adder which can handle inputs in the range of (-1,0,1).

Select is implemented exactly in same manner as equation (5.10). Since no accumulation of the input bits is required in this case, the hardware structure is simpler than what we have discussed for OLMLP unit.

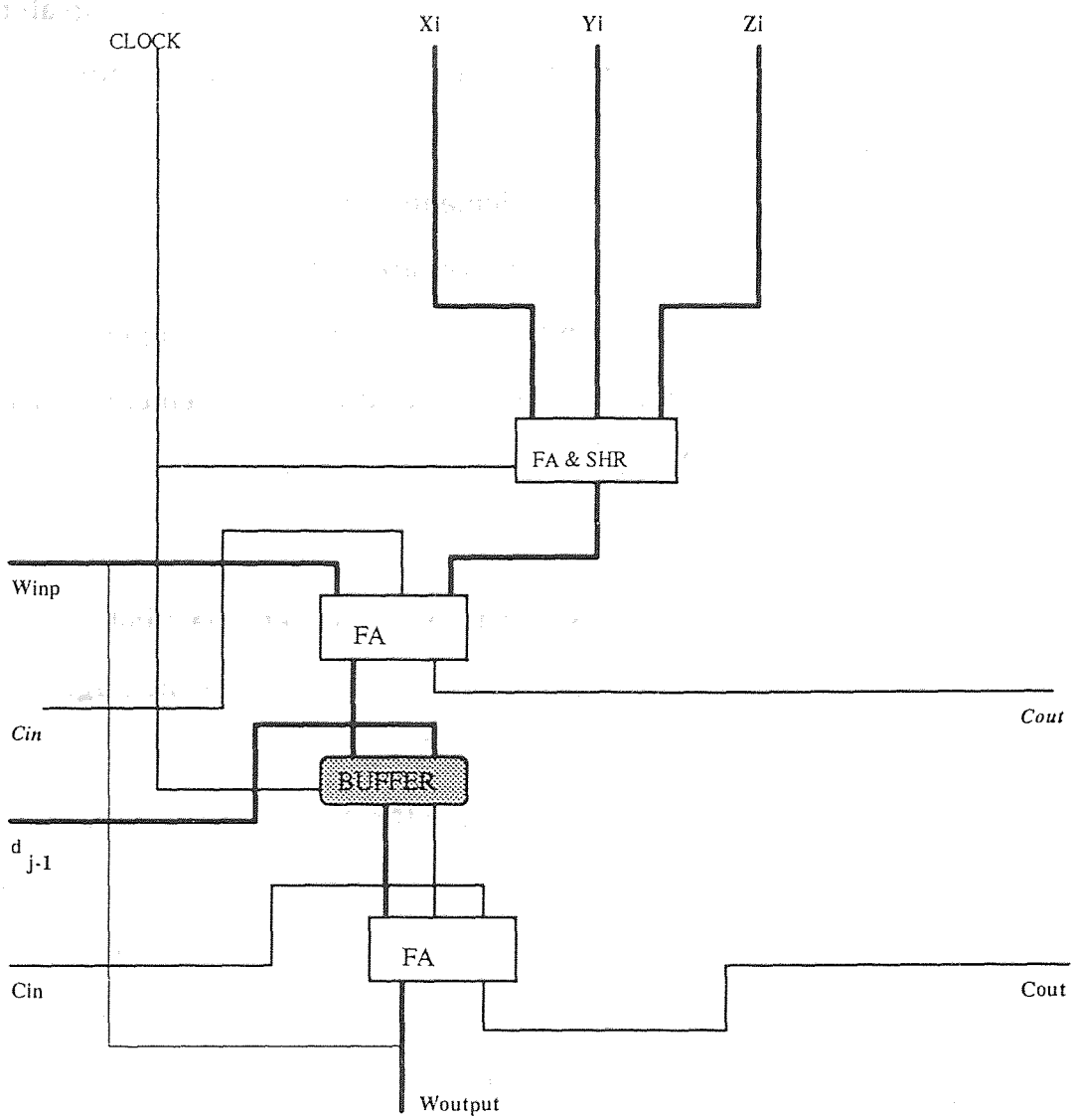

Figure 19 Bit slice Organization of the on-line add module

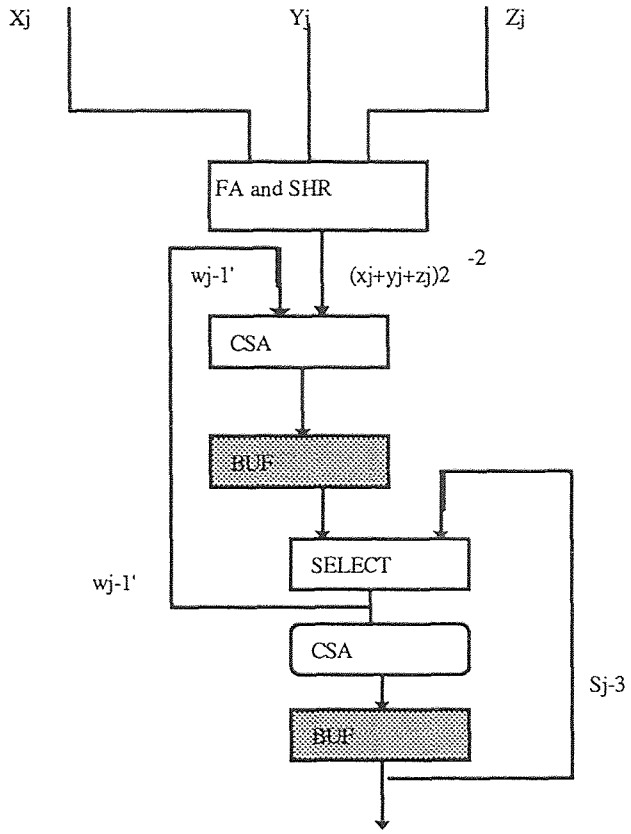

Figure 18 Pipelined Implementation of OLADD3

Since RNS could not be handled properly within the accumulator, it is required that the accumulated bits within the accumulator be converted from RNS to conventional number system.

Instead of using two separate buffers as shown in the figure 19, we use a common buffer to store w_{j-1} and s_{j-3}.

Note that,

- the clocking circuitry designed latches latches the incoming x_j and y_j bits at the jth clock cycle of the jth bit slice in order to have the correct position of x_j and y_j.
- inputs x_j and y_j are two bit inputs.

CHAPTER 6

SOFTWARE SIMULATION

6.1 Overview

The simulation was designed primarily to serve as

1. a tool for experimental testing and understanding of the on-line cosine algorithm;
2. a tool to study the real-time behavior of the algorithm where intermediate results at each step (clock cycle) can be viewed;
3. a performance evaluation tool for on-line arithmetic unit for cosine function evaluation;
4. a calculator which can do binary operations such as add, subtract, multiply, divide (some of which are not supported by conventional calculators) on numbers with precision up to 256 bits. It also serves as a converter which can convert a number in redundant number system (-1, 0, 1) to conventional binary system (0, 1).

6.2 Design of Simulator

The simulator is designed to be highly modular and convenient for interactive usage. The simulator has been written completely in a graphical user interface language - Visual Basic (programming language for windows), which allows the simulator be implemented exactly as the logic design discussed before. The language is different from other languages as being an object oriented language and also an event driven language (Appendix D), which is exactly what is required for simulation of on line algorithm. An event can be generated whenever a bit arrives at the input of the cosine unit in an on-line manner. This

facing 40

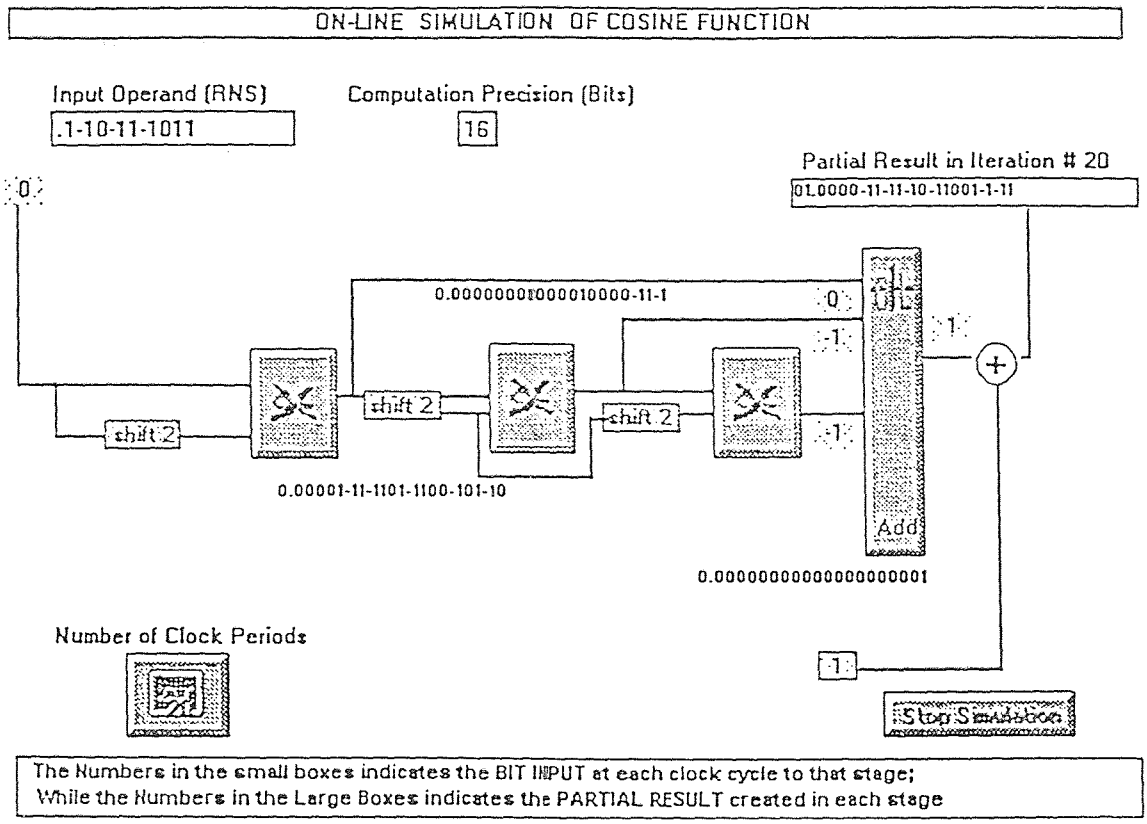

Figure 20 User Interface Screen Of The Simulator

event is used to evaluate the bit and calculate the partial result corresponding to that bit position.

The simulator uses as input the same number system as is actually required by the design. The use of redundant number representations of the operands is necessary as in on-line arithmetic. Results are also in redundant number. The simulator uses modules for on-line add (OLADD3) and on-line multiply(OLMLP) which have the same capabilities as discussed in Chapter 5. All hardware logic designs discussed previously have been implemented by equivalent software modules. Algorithms for OLADD3 and OLMLP are discussed in Appendix B. The on-line delay of OLADD3 is taken as $\delta = 1$.

A functional block diagram of the on-line simulator is shown in figure 20. This is an actual print out of the user interface screen of the simulator. The block diagram shows the OLMLP and OLADD3 units along with the required shifters to implement the cosine function. A "1" is added to the final result in each step using the preappend method discussed in section 5.2. A button on the left hand corner is used to display the current clock period(iteration number), it can also be used to run the simulation manually. The button on the right hand corner is used to start or stop the simulation. Intermediate boxes show the partial result generated by each stage in that clock period.

The input part prompts the user to enter the input bit stream (either in regular binary or in redundant number system). Even though the input section displays the complete bit stream, the simulator engine only takes input bitwise depending on the clock period. The advantage of the user giving the whole input is, once a complete input has been given the whole simulation procedure can be automated instead of the user entering each bit at each clock period. The small boxes indicate the bit inputs to the corresponding stage at that particular

Input Precison: 16
Input x: 0.1-11-101-1011
Input y: 0.-1110-1101101
Output: 000.1-1-1100001-1100
Input z:
Return
Clear

i	xi	yi	(xi + yi)2^-2	Wi	Si-2	2(Wi - Si-2)
0	0	0	0000000000000000	0.0000000000000000	0	0.0000000000000000
1	1	-1	0000000000000000	0.0000000000000000	0	0.0000000000000000
2	-1	1	0000000000000000	0.0000000000000000	0	0.0000000000000000
3	1	1	1000000000000000	0.1000000000000000	1	-1.0000000000000000
4	-1	0	-0100000000000000	-1.0100000000000000	-1	-0.1000000000000000
5	0	-1	-0100000000000000	-0.1100000000000000	-1	0.1000000000000000
6	1	1	1000000000000000	1.0000000000000000	1	0.0000000000000000
7	-1	0	-0100000000000000	-0.0100000000000000	0	-0.1000000000000000
8	0	1	0100000000000000	-0.0100000000000000	0	-0.1000000000000000
9	1	1	1000000000000000	0.0000000000000000	0	0.0000000000000000
10	1	0	0100000000000000	0.0100000000000000	0	0.1000000000000000
11	0	1	0100000000000000	0.1100000000000000	1	-0.1000000000000000
12	0	0	0000000000000000	-0.1000000000000000	-1	1.0000000000000000
13	0	0	0000000000000000	1.0000000000000000	1	0.0000000000000000
14	0	0	0000000000000000	0.0000000000000000	0	0.0000000000000000
15	0	0	0000000000000000	0.0000000000000000	0	0.0000000000000000

Figure 21 Exploded View Of The OLADD3 Unit

Input Precison: 16
Input x: .1-101-11001-111
Input y: .-1011-1100-1-111
Output: 0.-1110100-1-1100000
Input z:
Return
Clear

i	xi	yi	XByi+Yi-1xi	Wi	di	2(Wi - di)
1	1	-1	-0.1000000000000000	-0.1000000000000000	-1	1.0000000000000000
2	-1	0	0.1000000000000000	1.1000000000000000	1	1.0000000000000000
3	0	1	0.0100000000000000	1.0100000000000000	1	0.1000000000000000
4	1	1	-0.0001000000000000	0.0111000000000000	0	0.1110000000000000
5	-1	-1	0.0000100000000000	0.1110100000000000	1	-0.0011000000000000
6	1	1	-0.0000110000000000	-0.0011110000000000	0	-0.0111100000000000
7	0	0	0.0000000000000000	-0.0111100000000000	0	-0.1111000000000000
8	0	0	0.0000000000000000	-0.1111000000000000	-1	0.0010000000000000
9	1	-1	-0.1010000010000000	-0.1000000010000000	-1	0.1111111100000000
10	-1	-1	0.0000100001000000	1.0000011101000000	1	0.0000111010000000
11	1	1	-0.0000100001100000	0.0000011000100000	0	0.0000110001000000
12	1	1	-0.0000100000110000	0.0000010000010000	0	0.0000100000100000
13	0	0	0.0000000000000000	0.0000100000100000	0	0.0001000001000000
14	0	0	0.0000000000000000	0.0001000001000000	0	0.0010000010000000
15	0	0	0.0000000000000000	0.0010000010000000	0	0.0100000100000000
16	0	0				

Figure 22 Exploded View Of The OLMLP Unit

clock period. Also global parameters such as precision can be entered by the user (if not entered the default computation precision is 16 bits).

The centralized control mechanism used in this simulator is a function which facilitates input in an on-line fashion. It finds the input bit from the given bit stream and does the necessary conversions so that it can be interpreted by Visual Basic properly. The simulator engine then keeps the current clock period and simultaneously broadcasts the clock signal to each unit (OLMLP0, OLMLP1, OLMLP2, OLADD3) so that they can take the necessary action in response to that. One more important aspect that the simulator takes care of is that the OLMLP algorithm can handle only numbers which are less than 1.0. The simulator eliminates this problem by placing the shifters at the appropriate positions in the data path so that the input to any on-line unit has computation value less than 1.0. Hence instead of shifting the operand after calculating different powers of it, we just shift it intermediately along the data path as shown in figure 18.

The execution part of the simulator is interfaced to a set of on-line arithmetic procedures. These procedures are reentrant and the design is such that it is relatively easy to add new modules specifying new on-line algorithms. The interface between the execution part of the simulator and the algorithm procedures is a set of parameters which are used and modified by the algorithms (one such example is the current clock period in which the simulator is executing). This way the control structure of the simulator is hidden and modifying an algorithm or implementing a new algorithm is relatively easy.

In the evaluation of the cosine function, computations are performed in an automated manner. An event is generated by using a timer control which in turn generates interrupt and is used to start the computation using the input bit. The evaluation of the input operand continues until the specified precision output is

obtained. After that a user can override the automatic simulation and gets output of higher precision than specified in the input area by clicking the time control button.

The following objectives were achieved by developing a simulator for on-line evaluation of the cosine function:

1. an insight into the algorithm on the implementation and operational details;
2. automate computations without having to do complex calculations manually;
3. performance evaluation by running it virtually as a real time application using a timer control which generates timer interrupts at every specified interval;
4. provide user with a very easy to understand and comprehensive simulation interface, so that it can help them in understanding the working of on-line mechanism.

CHAPTER 7

CONCLUSION

An alternative to the conventional CORDIC method for evaluating trigonometric functions was studied. On-line algorithms have been developed to evaluate the sine and cosine functions. Error analysis and heuristics are carried out to arrive at a minimal error algorithm based on the series expansion of the sine and cosine function. A logical design based on the algorithm is presented where the unit is designed as a set of basic modules. A detailed bit slice design of each module is also presented. A simulator was designed as an experimental tool for synthesis of the on-line algorithms, and a tool for performance evaluation.

The main advantages achieved by this approach are 1) the use of redundant number system, which allows carry free addition to replace time consuming carry propagate additions; 2) the use of on-line arithmetic to reduce the communication bandwidth and maximize the overlap between successive operations; 3) eliminate ROM table lookup method used by other trigonometric algorithms[10].

The error analysis indicates errors for the optimal sine and cosine series as $f_{cos}(x) = 1 - x^2/2 + x^4/16 - x^6/128$ maximum normal error of $2.958 * 10^{-2}$ and $f_{sin}(x) = x - x^3/8 + x^5/1024 - x^7/16384$ maximum normal error of $9.395 * 10^{-2}$.

7.1 Future Research

The on-line algorithms presented can be further optimized in terms of their hardware implementation. The restriction of requiring the input within the range [-1,1] on the OLMLP algorithm can be eliminated. This can facilitate in reducing the number of shifters used in the hardware design. A more

sophisticated simulator can be developed where the user can use the available modules to generate new on-line algorithms without having to make major changes at the code-level. The series approximations can be improved to reduce the error using advanced heuristic techniques.

APPENDIX A

CONVERSION ALGORITHM

We now consider an algorithm to convert redundant number representation into conventional representation. The algorithm is performed concurrently by each accumulators as discussed in section 5.2. It can be divided into bit slice form, so that each bit slice in the logic implementation can utilize it inherently. This algorithm has a step delay approximately equivalent to the delay of a carry-save adder and its implementation is relatively simple. It can be applied to any on-line arithmetic algorithms generating redundantly represented results in a digit-by-digit manner from the most significant digit to the least significant digit. The important characteristics of this algorithm are:

- It performs the conversion on the fly, as the digits of the result are obtained in a serial fashion from most significant to least significant;
- it uses two conditional forms of the current result, similar to the conditional sum technique;
- it has a delay which is compatible with one step of the fast division algorithm. This delay is approximately of a carry-save adder.

Radix- r Conversion

m_k = Input bit at time k (redundant form)

$n[k]$ = Output bit (conventional form) at time k

$$n[k] = \begin{cases} P[k-1] + m_k r^{-k} \ if \ m_k \geq 0 \\ Q[k-1] + (r - |p_k|) r^{-k} \ if \ m_k < -1 \end{cases}$$

$$P[k-1] = -n_0 + \sum_{i=1}^{k-1} n_i . 2^{-i} \quad = \quad n[k-1]$$

where,

$$Q[k-1] = P[k-1] - r^{-(k-1)} \tag{5.6}$$

(1) Initialization:

$$P[1] = \begin{cases} +m_1 r^{-1} \quad (0.m_1) & \text{if } m > 0 \\ -|m_1| r^{-1} \quad (1.(r-|m_1|)) & \text{if } m < 0 \end{cases}$$

$$Q[1] = \begin{cases} +(m_1 - 1) r^{-1} \quad (0.(m_1 - 1)) & \text{if } m > 0 \\ -(|m_1|+1) r^{-1} \quad (1.(r-1-|m_1|)) & \text{if } m < 0 \end{cases}$$

(2) Recurrence:

For k > 1,

$$P[k+1] = \begin{cases} P[k] + m_{k+1} r^{-(k+1)} & m_{k+1} \geq 0 \\ Q[k] + (r - |m_{k+1}|) r^{-(k+1)} & m_{k+1} < 0 \end{cases}$$

$$Q[k+1] = \begin{cases} P[k] + (m_{k+1} - 1) r^{-(k+1)} & m_{k+1} > 0 \\ Q[k] + (r - 1 - |m_{k+1}|) r^{-(k+1)} & m_{k+1} = -1 \end{cases} \tag{5.7}$$

Adding $2^{-(k+1)}$ indicates a simple operation of appending appropriate bit to P[k] or Q[k].

The implementation of the algorithm requires two resisters to hold P[k] and Q[k]. These registers can be shifted one bit left with insertion in the least significant bit depending on the value of m_k. They also require parallel loading of P[k] with Q[k] and vice versa. The implementation related to our application has been described in details in chapter 5.

Example

r = 2

M = 0.1-101-110-1

Table 6 Conversion Procedure for Radix r = 2.

k	m_k	P[k]	Q[k]	n[k]
0	0			
1	1	0.1	0.0	0.1
2	-1	0.01	0.00	0.01
3	0	0.010	0.001	0.010
4	1	0.0101	0.0100	0.0101
5	-1	0.01001	0.01000	0.01001
6	1	0.010011	0.010010	0.010011
7	0	0.0100110	0.0100101	0.0100110
8	-1	0.01001011	0.01001010	0.01001011

The converted fraction is n = 0.01001011

Another algorithm for converting the redundant representation to the conventional representation is also available which is simpler in logic but requires more complex hardware implementation. This algorithm separates the redundant number into two different numbers A[j] and B[j] according to table 7.

Table 7 Substitution Table.

j	A[j]	B[j]
1	1	0
-1	0	1
0	0	0

We get the converted number n[j] by just subtracting A[j] - B[j]. Due to large carry propagate delays and hardware complexities, we do not implement this algorithm.

APPENDIX B

ON-LINE ALGORITHMS

We present here some of the On-line algorithms that were frequently used in the on-line cosine function implementation. All of these algorithms are presented as actual functions that were used in developing the simulator. They require certain global parameters which are either algorithm generated or are given by the user as global parameters.

- xj: bit input of operand X at iteration "xitr"
- yj: bit input of operand Y at iteration "yitr"
- xitr: iteration number for operand X
- yitr: iteration number for operand Y
- wj: intermediate sum at step j
- dj: result bit at step j
- X[j]: the accumulated operand X at step j
- Y[j]: the accumulated operand Y at step j

They also use certain global parameters like exp() which indicates the exponent part of the two inputs and the result.

- Shiftrt function shifts the passed operand right by the number of bits specified;
- sSelect function implements the rounding select function.

On-Line ADD

```
Function OLADD3( xj as integer, yj as integer, xitr as integer, yitr as integer, wj
as                    double, dj as integer)  as integer
dim lvi, lvj, lvk, lvdiff as integer
dim lwj, ldj as double
```

```
if exp(0) > exp(1) then
  lvdiff = (exp(0) - exp(1))
 shiftrt (yj, lvdiff)
  exp(2) = exp(0)
  xitr = xitr + lvdiff
else if exp(1) > exp(0) then
  lvdiff = (exp(1) - exp(0))
  shiftrt (xj, lvdiff)
  exp(2) = exp(1)
   yitr = yitr + lvdiff
endif
lvj = xitr
lvk = yitr
lwj = 2*(wj - dj) + (2^-1)*( xj + yj )
wj = lwj
ldj = sSelect( wj)
OLADD3 = ldj   ' Return the result bit value
End Function
```

On-Line Multiply

```
Function OLMLP ( xj as integer, yj as integer, xitr as integer, yitr as integer, wj as
                                double, dj as integer)  as integer
dim lvi, lvj, lvk, lvdiff as integer
dim lwj, ldj as double
  exp(2) = exp(1) + exp(0)
For lvi = 0 to xitr -1
  lvx = lvx + X[lvi]*2^(-lvi)
```

```
Next lvi
For lvj = 0 to yitr
  lvy = lvy + Y[lvj]*2^(-lvj)
Next lvj
lwj = 2*(wj - dj) + ( xj * lvx + yj * lvy )
wj = lwj
ldj = sSelect( wj)
OLMLP = ldj   ' Return the result bit value
End Function
```

On-Line Squaring

```
Function OLSQR( xj as integer, xitr as integer, wj as double, dj as integer)  as
integer
dim lvi, lvj, lvk, lvdiff as integer
dim lwj, ldj as double
exp (2) =  2*exp(0)
For lvi = 0 to xitr -1
  lvx = lvx + X[lvi]*2^(-lvi)
Next lvi
lwj = 2*(wj - dj) + ( xj * ( lvx + xj*2^(-j) )
wj = lwj
ldj = sSelect( wj)
OLSQR = ldj   ' Return the result bit value
End Function
```

APPENDIX C

HEURISTICS & ERROR ANALYSIS GRAPHS

We present here the figures of the error and the input deviation error curves that were obtained during the error analysis are discussed in chapter 3. All of these figures are presented as actual printouts of the interface screen of the program that was written to graph these errors.

- Figure 23 shows the error curve for modified sine series $f_{sin}(x) = X - X^3/2^3 + X^5/2^{10} - X^7/2^{14}$ with computation precision as 16 bits.
- Figure 24 shows the error curve for modified cosine series $f_{cos}(x) = 1 - X^2/2^1 + X^4/2^4 - X^6/2^9$ with precision as 16 bits.
- Figure 25 shows the error curve for modified sine series $f_{sin}(x) = X - X^3/2^3 + X^5/2^5 - X^7/2^{14}$ with computation precision as 16 bits.
- Figure 26 shows the error curve for modified cosine series $f_{cos}(x) = 1 - X^2/2^1 + X^4/2^4 - X^6/2^7$ with precision as 8 bits.
- Figure 27 shows the input deviation error curve for the series $f_{cos}(x) = 1 - X^2/2^1 + X^4/2^4 - X^6/2^9$ with precision as 16 bits.
- Figure 28 gives the input deviation error curve for series $f_{cos}(x) = 1 - X^2/2^1 + X^4/2^4 - X^6/2^7$ with precision as 8 bits.

As indicated in the figures 23-26, the X-Axis gives the increase in input values in radian, where each tick indicates an increase in input by 0.1 radian. Each tick on the Y-Axis inidicates an increase in error by 0.05 units. Figures 27-28 show that each tick on Y-Axis indicates the increase in input deviation error by 1.0 degree.

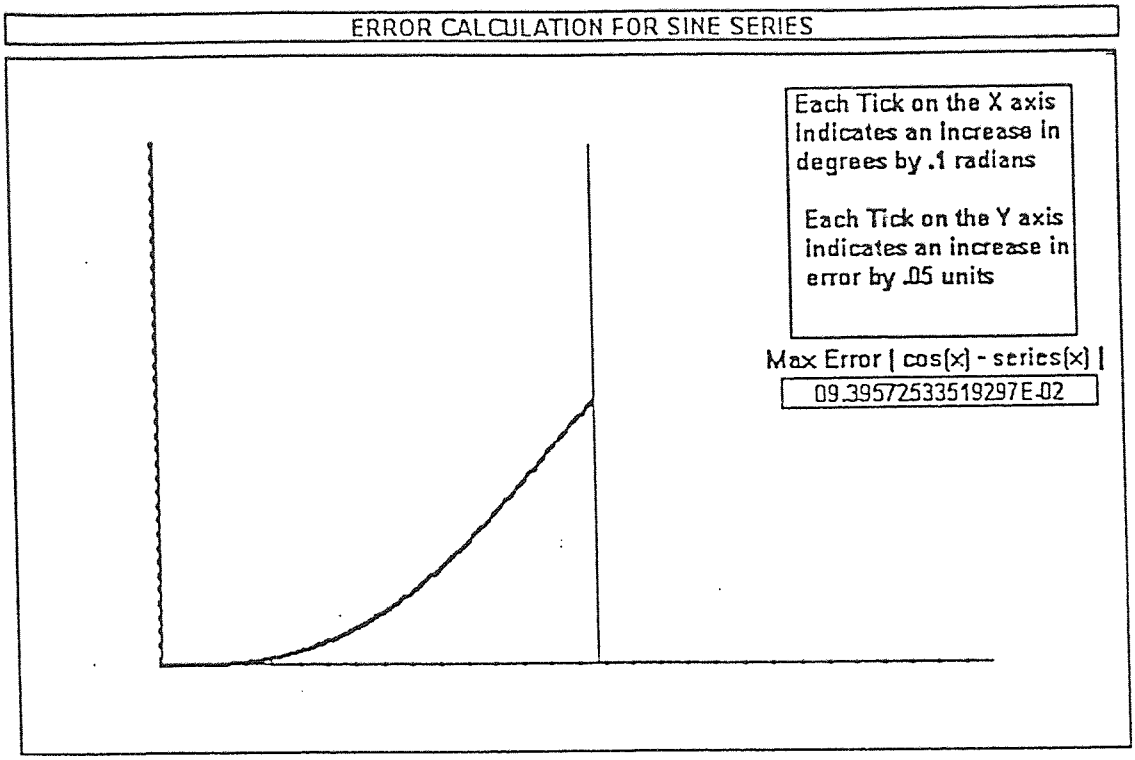

Figure 23 The Error Curve for $f_{sin}(x) = X - X^3/2^3 + X^5/2^{10} - X^7/2^{14}$; (m = 16).

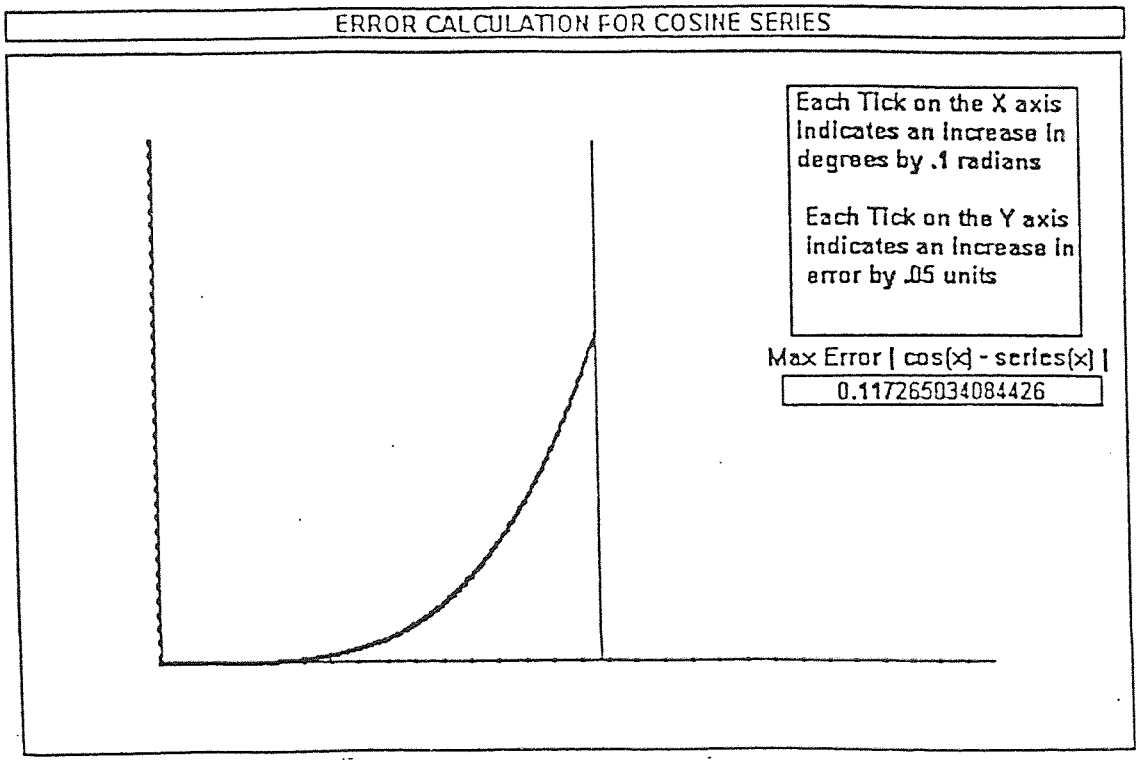

Figure 24 The Error Curve for $f_{cos}(x) = 1 - X^2/2^1 + X^4/2^4 - X^6/2^9$; (m =16).

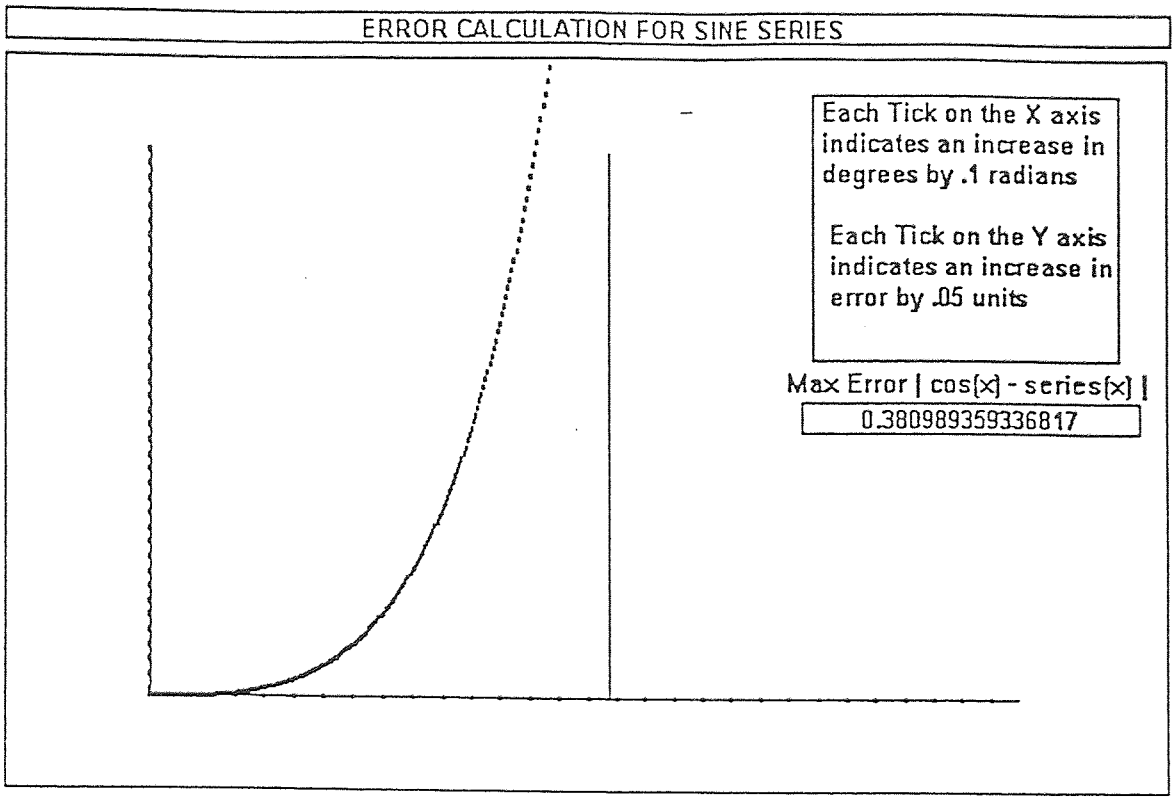

Figure 25 The Error Curve for $f_{sin}(x) = X - X^3/2^3 + X^5/2^5 - X^7/2^{14}$; (m =16).

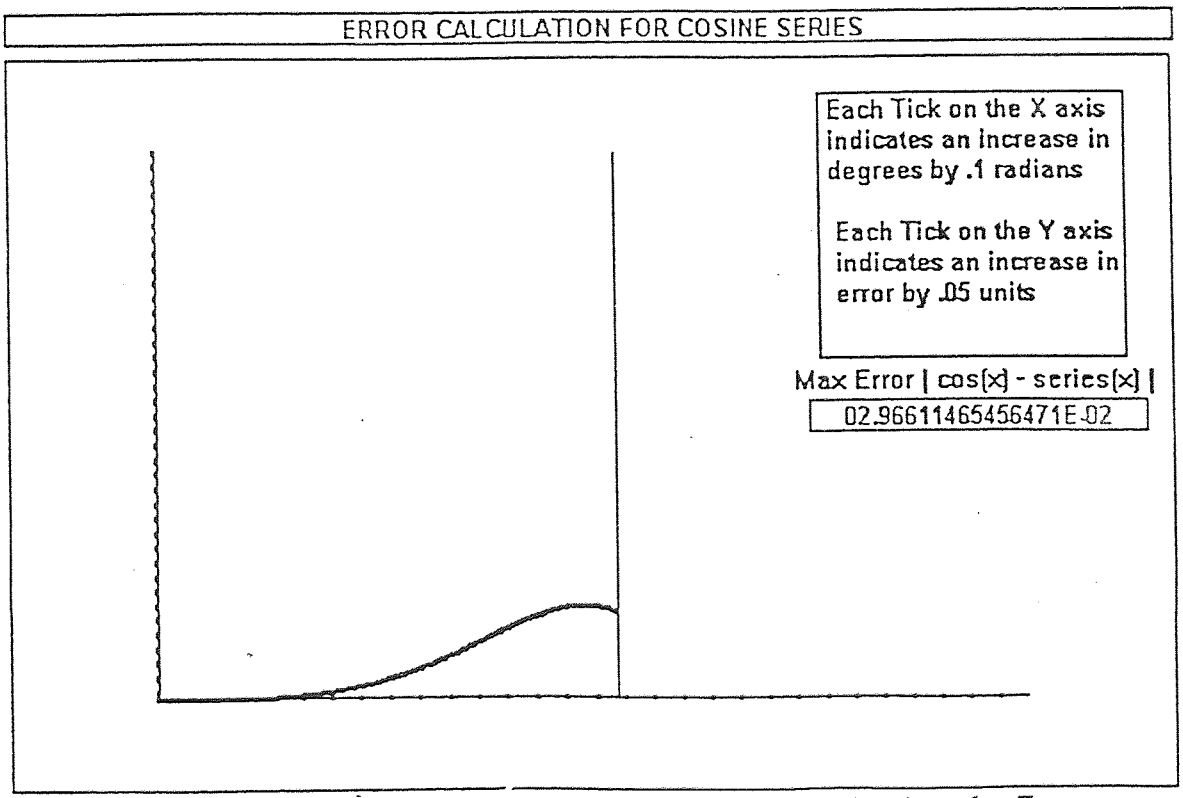

Figure 26 The Error Curve for $f_{cos}(x) = 1 - X^2/2^1 + X^4/2^4 - X^6/2^7$; (m =8).

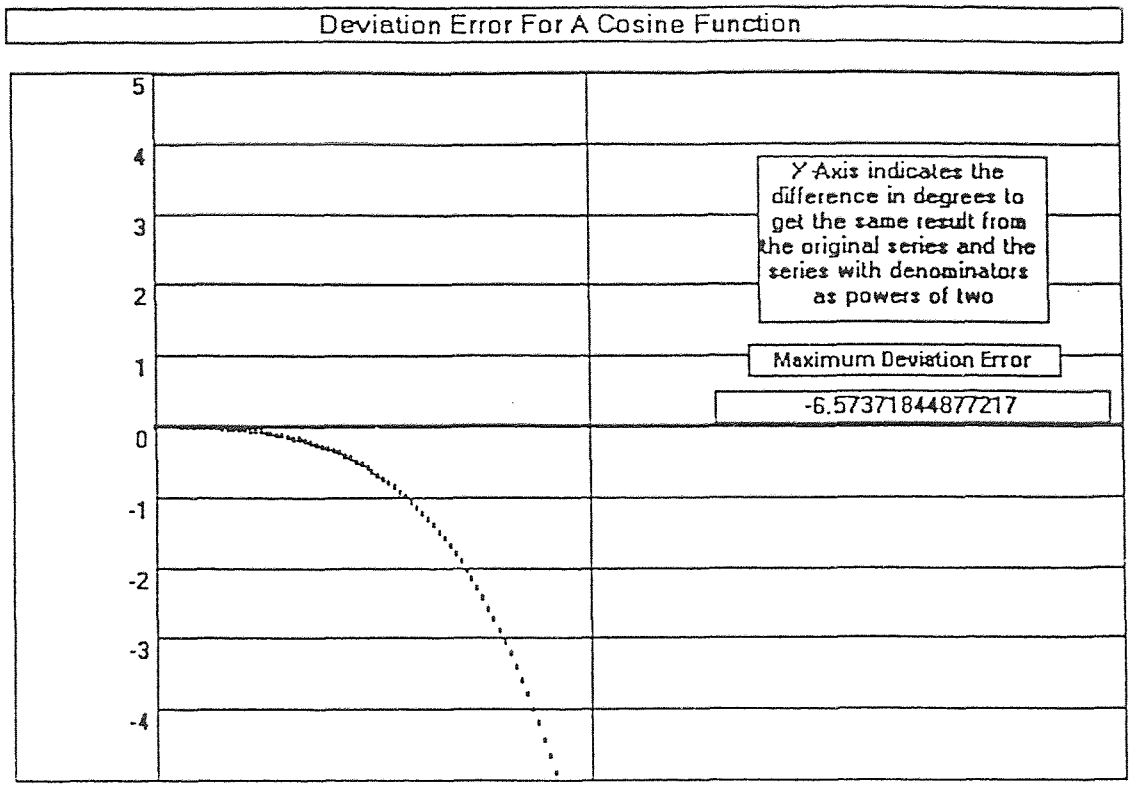

Figure 27 Deviation Error Curve for $f_{cos}(x) = 1 - X^2/2^1 + X^4/2^4 - X^6/2^9$.

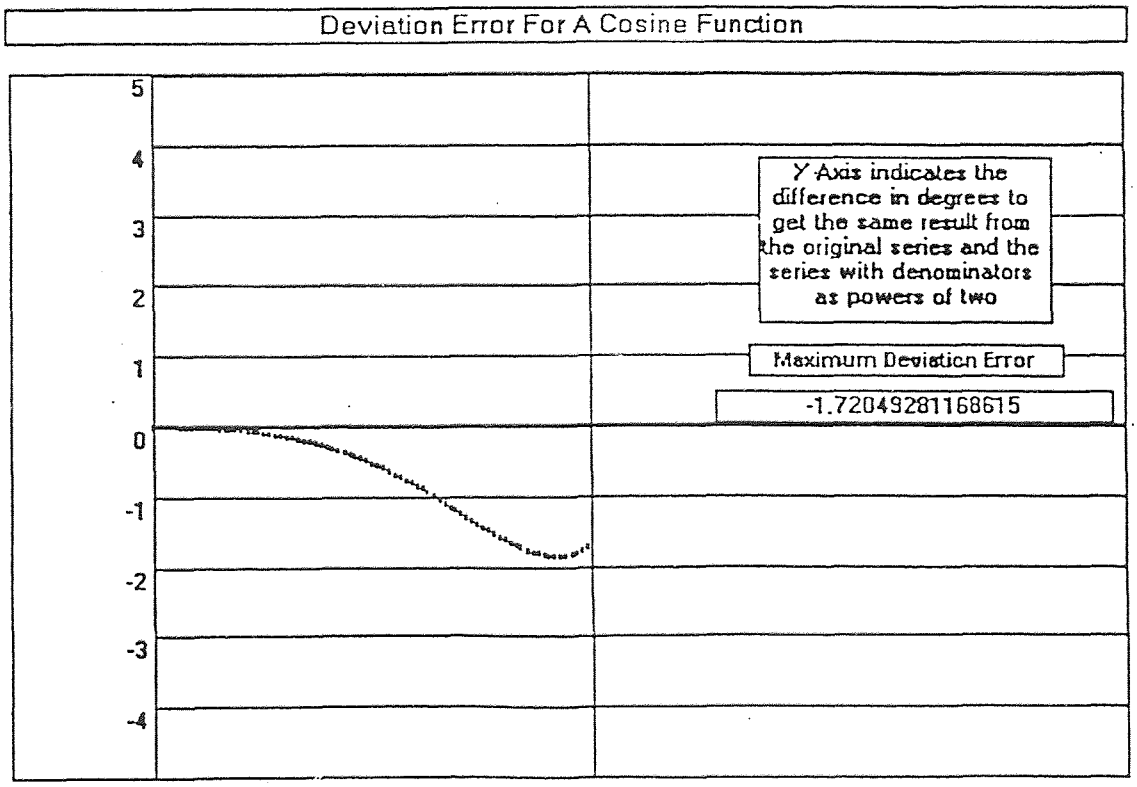

Figure 28 Deviation Error Curve for $f_{cos}(x) = 1 - X^2/2^1 + X^4/2^4 - X^6/2^7$.

APPENDIX D

Introduction to Visual Basic

Visual Basic is a programming language from Microsoft , a graphical user interface (GUI) revolution. Applications for Microsoft Windows or OS/2 Presentation Manager can be easily and efficiently developed in Visual Basic. It is a development system especially geared toward creating graphical applications. It includes graphical design tools and a simplified, high-level language. It emphasizes feedback and debugging tools that quickly take you from an idea to a running application.

Visual Basic is centered around two types of objects: you create windows, called forms, and on those forms you draw objects, called controls. Then you program how forms and controls respond to user actions. The applications you produce are fast and can include all of the most common features an user expect in a GUI environment.

Visual Basic works under the Windows operating system environment. The Windows operating environment differs from DOS in at least two important ways:

- Applications share screen space. A Visual Basic application runs in a group of one or more windows and rarely takes over the whole screen.
- Applications share computing time. An application cannot run continually, or if it does, it has to be able to run in the background.

The event-driven approach used by Visual Basic enables you to share computing time and other resources(such as Clipboard). An event-driven application consists of objects that wait for a particular event to happen. (An

event is an action recognized by a Visual Basic object. Objects include form and controls.)

The Visual Basic code does not work in the linear fashion of a DOS program - starting at the top, proceeding toward the bottom, and finally stopping. Instead, in event-driven programming, you use code that remains idle until called upon to respond to specific user-caused or system-caused events.

For example, you might program a command button to respond to a mouse click. When the command button recognizes that the event has occurred, it invokes the code you wrote for that event.

While the application is waiting for an event, it remains in the environment. In the meantime, the user can run other applications, resize windows, or customize system settings such as color. But the code is always present, ready to be activated when the user returns to the application.

Features Supported:

- A full set of the objects needed to create Windows applications, including: command buttons, option buttons, check boxes, list boxes, combo boxes, text boxes, scroll bars, frames, file and directory selection boxes, and menu bars.
- Multiple windows in an application.
- Highly flexible response to mouse and keyboard events at run time, including automated drag-and-drop support.
- Ability to show and hide any number of items at run time.
- Direct access to the environment's Clipboard and to the printer.
- Direct system calls to Windows functions.
- Communication with other Windows applications through dynamic data exchange (DDE), and extensibility via dynamic-link-libraries (DLL). For example, a user can call dynamic-link-libraries (including Windows functions) for within Visual Basic code.

- Graphics statements.
- A powerful math and string handling library.
- Easy-to-use string variables.
- Both fixed arrays and dynamic arrays. (The latter help to simplify memory-management problems.)
- Random-access and sequential file support.
- Sophisticated run-time error handling.

Visual Basic also makes development easier by providing a set of powerful debugging commands that help isolate and correct errors in code. Visual Basic operates as an incremental compiler, instantly translating code statements into "runnable" form as soon as they are typed. GUI environments generally make computing easier and more fun for the user. But these environments create more complexities for developers, who now must think visually, write code that responds to events, and anticipate how users will interact with applications.

REFERENCES

1. Hwang, K. "On-Line Algorithms for Divison and Multiplication." *IEEE Trans. Comput.*, (1977 Jul):681-687.

2. Owens, R.M., and M.J. Irwin. "Designing Pipeline Architectures using On-Line Algorithms." *Proc. of the 6th Annual Symposium on Computer Architecture,* (1979 Apr):12-19.

3. Hwang, K. Advanced Computer Architecture with Parallel Programming. Preliminary Edition.,(1993).

4. Avizienis, A.,"Signed-Digit Number Representations for Fast Parallel Arithmetic." *IRE Trans. Electron. Comput.*,(1961):389-400.

5. Ercegovac, M.D., "On-Line Arithmetic: An Overview." *Proc. SPIE Conf. Real Time Signal Proc.*(1984):667-680.

6. Volder, J.E. "The CORDIC Trigonometric Computing Technique." *IRE Trans. Elect. Comp.*, (1959):330-334.

7. Golub, G.H., and C.F. Van Loan. Matrix Computations. Baltimore, MD: The John Hopkins University Press (1983).

8. Haviland, G.L., and A.A. Tuszyunski. "A CORDIC Arithmetic Processor Chip." *IEEE Trans. Comp.*, (1980):68-78.

9. Waser, S., and M.J. Flynn. Introduction to Arithmetic for Digital Systems Designer. NY: Holt, Rinehart & Winston, (1982).

10. Fowkes, R.E. "Hardware Efficient Algorithms for Trigonometric Functions." *IEEE Trans. Comput.*, Vol. 42 No.2(1993 Feb).

11. Steer, D.G., and S.R. Penstone. "Digital Hardware for Sine-Cosine Function." *IEEE Trans. Comp.*, (1977):1283-1286.

12. Brackert Jr., "A High Speed Recursive Digital Filter Using On-Line Arithmetic." *IEEE ISCAS*,(1989).

13. Tullsen D.M. and M.D. Ercegovac "Design and VLSI Implementation of an On-Line Algorithm." *Proc. SPIE Conf. Real-Time Signal Process.*, (1986).

REFERENCES
(Continued)

14. Ercegovac, M.D., "A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer." *IEEE Trans. Comput.* (1977):667-680.

15. Gorji-Sinaka, A., and M.D. Ercegovac., "Design of a Digit Slice On-Line Arithmetic Unit." *IEEE Symp on Comput. Arith.*, (1981):72-80.

16. Ercegovac, M.D., and T. Lang. "On-The-Fly Conversion of Redundant into Conventional Representation." *IEEE Trans. Comput.*, (1987):895-897.

17. Atkins, D.E. "Introduction to the Role of Redundancy in Computer Arithmetic." *Computer*. Vol. 8, No. 6(1975):84-96.

18. Ercegovac, M.D. "An On-Line Square Rooting Algorithm." *Proc. of the 4th Symposium on Computer Arithmetic*, (1978 Oct):183-189.

19. Mano, M.M., Computer System Architecture, Second Edition., (1982).

20. Hwang, K., Computer Arithmetic Priciples, Architecture and Design. NY: Wiley (1979).

21. Microsoft, Visual Basic Manual 2.0 Edition. Microsoft Press (1992).

22. Irwin, M.J., "Reconfigurable Pipeline Systems." *Proc.1978 Annual Conf. of the ACM*, (1979 Apr):86-92.