1973-05-08

An analysis of algorithms for hardware evaluation of elementary functions

Franke, Richard H.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/29706

NPS-53FE73051A

NAVAL POSTGRADUATE SCHOOL

1

Monterey, California

An Analysis of Algorithms for Hardware Evaluation of Elementary Functions

by

Richard Franke

Approved for public release; distribution unlimited.

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Abstract: Algorithms for the automatic evaluation of elementary functions were studied. Available algorithms obtained from current literature were analyzed to determine their suitability for hardware implementation, in terms of their accuracy, convergence rate, and hardware requirements. The functions considered were quotient, arctangent, cosine/sine, exponential, power function, logarithm, tangent, square root, and product.

This task was supported by: Naval Electronics Laboratory Center
San Diego, CA 92152

NPS-53FE73051A
8 May 1973

Table of Contents

1.0 Introduction

The purpose of this study was to identify and analyze mathematical algorithms for possible hardware implementation. Emphasis was placed on the evaluation of elementary functions, such as square root, logarithm and exponential, trigonometric functions, and multiply and divide. The study was directed toward algorithms for binary computers, although some references are included which address themselves to radices 10 [56], 16 [18], and -2 [53], [54], [65].

We have not assumed that any fixed degree of accuracy was required. Rather, we have generally concentrated on methods which are flexible enough for the accuracy to be a function of the number of iterations performed. We have generally assumed that extremely low precision is not sufficient, thus ruling out consideration of methods which are essentially table-look-up, with or without interpolation (e.g., [23], [52], [57]). Low precision interpolation, such as Mitchell [43], and variations of it ([11], [27], [39]) have not been studied.

The commitment of a large amount of hardware can sometimes be used to decrease the execution time for evaluation of functions. The use of cellular arrays has been proposed for multiplication [13], division [37], [62], [3], square root [9], [22], [38], and logarithm [15]. While this idea is promising in terms of speed, we have concentrated on algorithms to be implemented with the use of only a moderate amount of parallel processing.

A number of the proposed algorithms are unified in the sense that with variations in certain parameters the same procedure can be used to evaluate any one of several functions. A description of three algorithms of this type will be given in Section 3, before proceeding with a discussion of algorithms for specific functions in Section 4. A short consideration of error analysis is given in Section 2.

2.0 Error Analysis Considerations

The error involved in evaluating a function consists of three parts: (i) Roundoff error is accumulated in doing the necessary arithmetic; (ii) The methods are interative, and convergence is obtained only to within a specified tolerance, thus truncation error occurs; and (iii) If the intermediate results are carried to additional bits of accuracy to reduce roundoff error accumulation, an error is committed by reducing the number of bits in the final result to machine precision.

We will generally assume that we are dealing with fractional numbers which involve N fraction bits. This is compatible with the notion of floating point arithmetic, where the value of the exponent, or characteristic, is taken care of by a separate normalization procedure. This may be performed before or after the algorithm we discuss. More is said about this in discussing the various functions.

Suppose that M operations where roundoff error may occur are performed. If the error committed each time is bounded by ϵ, the total error can not exceed $M\epsilon$. Assuming N fraction bits, we would probably want the roundoff error to be bounded by 2^{-N-1}, or $M\epsilon \leq 2^{-N-1}$. Thus $-\log_2 \epsilon \geq N + 1 + \log_2 M$. Roundoff error is decreased by increasing the precision of the intermediate results; say, use J "guard" bits for a total of N + J fraction bits. Then

$$(1) \qquad \epsilon = K \cdot 2^{-N-J-1} ,$$

$$\text{where} \qquad K = \begin{cases} 1 & , \text{ rounding} \\ 2 & , \text{ chopping} \quad * \end{cases}$$

Thus, (1) in the above, yields $-\log_2 K + N + J + 1 \geq N + 1 + \log_2 M$,

*The term chopping will refer to simply truncating after N + J bits without regard to the N + J + 1st bit. The term truncation error will be defined later.

or

$$(2) \qquad J \geq \begin{cases} \log_2 M & \text{for rounding} \\ \log_2 M + 1 & \text{for chopping .} \end{cases}$$

The truncation error in these algorithms occurs when a function g(x) is to be evaluated at a point α, and actual evaluation point is $\alpha - \gamma$. This error is the result of truncating an infinite process after a finite number of steps. If g is differentiable, which it is in our case, the difference in the function values is $g(\alpha) - g(\alpha - \gamma) = g'(\alpha^*)\gamma$, where α^* is between α and $\alpha - \gamma$. The number γ is related to the convergence criterion used in the algorithm, and in most cases it will be bounded by 2^{-N-1}, the same as the roundoff error bound we will assume.

The choice of bound for truncation error and roundoff error should be undertaken together, since it does not make good sense to choose either so that the truncation or roundoff error bounds differ significantly from each other. That is, it would not be meaningful to do enough iterations in a calculation to make the truncation error as small as 2^{-20} if the roundoff error could be as big as 2^{-16}. Conversely, it is wasteful to hold roundoff error to 2^{-20} if one is trying to obtain accuracy to 2^{-16}.

In line with the above proposed selection of J, the truncation error bound should be made about 2^{-N-1} also. The total error could then be as large as 2^{-N}, in the calculated value. It is then necessary to chop or round this result to N bits, which introduces an additional error of at most 2^{-N} or 2^{-N-1}, respectively. The final value could then be in error by as much as $2 \cdot 2^{-N}$ or $3/2 \cdot 2^{-N}$. These are bounds, and in the usual case the error will be smaller. One should keep in mind that they are sharp bounds, in the sense they may be approached closely in a given case.

It appears that it would certainly be worthwhile to round to N bits after the final iteration, since this procedure gives a significantly smaller total error bound than that for chopping. If the error bound given previously for the results of the iterations (before the final round) are not sufficiently accurate, the bound on this part of the calculation can be reduced to 2^{-N-1} from 2^{-N} by the performance of one additional iteration, using one more guard bit. The total error bound can never be made smaller than 2^{-N-1}, since that is the bound for the error in the final rounding operation.

The error bounds given in Section 4 will be for the N + J bit result, before the final round to N bits, the added error bound for the final rounding being the same for all cases where $J > 0$.

3.0 Unified Algorithms

Algorithms which, with small modifications, can evaluate one of several functions are basically of three types. One is formulated as a coordinate rotation problem, another is formulated as a "pseudo-division/multiplication" process, while a third type is a normalization procedure. We will discuss each of them in this section.

3.1 Coordinate Rotation Methods

The Coordinate Rotation Digital Computer was first discussed by Volder [64], who considered rotations in the usual circular coordinate system. It was indicated that work was also done in hyperbolic and linear coordinate systems, but this was not reported in detail. An earlier report by Volder [63], was not available. Liccardo [31] did a master's thesis on the CORDIC methods, and included the hyperbolic system. He also outlined procedures for multiply and divide. Linhardt and Miller [34] included the details of the hyperbolic system, but not the linear system. Walther [66] presented the unified algorithm and his paper tied together the rotations in the three different coordinate systems. A paper by Perle [47] also appeared, which included a method of obtaining $\sin^{-1}x$. The algorithm for $\sin^{-1}x$ does not fit the pattern, in that it does not generalize to hyperbolic and linear systems. Very recently, Schmid and Bogacki [56] discuss the implementation of the CORDIC algorithm in radix 10.

Basically, the CORDIC method involves taking a sequence of rotations (with radial distortion) of an initially rotated coordinate system. The initial angle of rotation is $z_o = z$. A point $(x_o,y_o) = (x,y)$ is specified by giving its coordinates with respect to the rotated coordinate system. We generate the following sequence of points by the rotations; (x_o,y_o), $(x_1,y_1),\dots,(x_n,y_n)$, where

$$(3)\qquad \begin{aligned} x_{i+1} &= x_i + ms_i\delta_i y_i \\ y_{i+1} &= y_i - \delta_i s_i x_i \end{aligned}$$

Here m is parameter indicating the type of coordinates for the rotations (1, 0, -1 for circular, linear, hyperbolic, respectively), $s_i = \pm 1$ determines the direction of rotation, and the δ_i are specified constants. The angles z_i of the rotated coordinate system, and radii R_i of the radius vectors to the (x_i,y_i) are given by

$$(4)\qquad \begin{aligned} z_{i+1} &= z_i + s_i\alpha_i \\ R_{i+1} &= R_i(1 + m\delta_i^2)^{\frac{1}{2}}, \text{ where} \\ \alpha_i &= m^{-\frac{1}{2}} \tan^{-1}(m^{\frac{1}{2}}\delta_i), \text{ and} \\ R_o &= (x_o^2 + y_o^2)^{\frac{1}{2}} \quad . \end{aligned}$$

z_o is specified, as indicated previously. We see that each (x_i,y_i) is the location of the (radially distorted) point (x_o,y_o), with respect to a coordinate system rotated through angle z_i. Also note that

$$z_n = z_o + \alpha$$

$$R_n = R_o K_m \text{ , where}$$

(5)
$$\alpha = \sum_{i=o}^{n-1} s_i \alpha_i \text{ , and}$$

$$K_m = \prod_{i=o}^{n-1} (1 + m\delta_i^{\,2})^{\frac{1}{2}}$$

are independent of the (x_i,y_i), except insofar as they may influence the s_i.

In terms of the initial values x_o and y_o, it can be shown that

(6)
$$x_n = K_m\{x_o \cos (\alpha m^{\frac{1}{2}}) + y_o m^{\frac{1}{2}} \sin (\alpha m^{\frac{1}{2}})\}$$

$$y_n = K_m\{y_o \cos (\alpha m^{\frac{1}{2}}) - x_o m^{-\frac{1}{2}} \sin(\alpha m^{\frac{1}{2}})\}$$

It is necessary that the initial values of x_o,y_o, and z_o be restricted, both for purposes of representing them in the computer, and to guarantee convergence of the algorithm. We say more about this later. With the appropriate restrictions on x_o,y_o, and z_o, and judicious choice of the s_i, the values of x_n,y_n, and z_n can be forced to approach that values indicated in Table 1. The terms rotation mode (s_i's chosen to force z_n to zero) and vectoring mode (s_i's chosen to force y_n to zero) were coined by Volder and are descriptive. The choice of s_i is as follows. In rotation mode

$$s_i = \begin{cases} 1 & \text{if} \quad z_i < 0 \\ -1 & \text{if} \quad z_i \geq 0. \end{cases}$$

In the vectoring mode

$$s_i = \begin{cases} 1 & \text{if} \quad y_i \geq 0 \\ -1 & \text{if} \quad y_i < 0 \end{cases} .$$

The latter is dependent on x_i being non-negative. This is easily adjusted for in cases where x_o is negative, as we will note in the pertinent sections.

The choice of δ_i is critical to the entire procedure and in order to facilitate computation, we want each δ_i to be a power of two. Thus the transformations (in a binary computer) are accomplished by shifting and adding. In order that the radial distortion constant, K_m, be independent of the input data, it is necessary that the magnitudes of the δ_i be independent of the input data. Thus the same sequence of rotations must always be done, except that the direction of rotation may vary. Walther gives the choice of δ_i given in Table 2. We also list the maximum value of α obtainable with this sequence, as well as the corresponding value of K_m and the number of iterations, N_m, required so that $\alpha_{N_m} \simeq 2^{-N-1}$.

We will see that with appropriate range reduction techniques, discussed for the individual functions in the next section, that the sequences given in Table 2 are adequate.

Because the last rotation of magnitude α_{N_m} must be accomplished, the value of z_{N_m+1} can be as large as α_{N_m}, in the rotation mode, and the value of y_{N_m+1} can be as large as $x_{N_m} \delta_{N_m}$ in the vectoring mode. The truncation error for the CORDIC algorithm is based on α_{N_m}, then.

Rotation mode

m	$\lim\limits_{n\to\infty} x_n$	$\lim\limits_{n\to\infty} y_n$	$\lim\limits_{n\to\infty} z_n$
1	$K_1(x \cos z - y \sin z)$	$K_1(y \cos z + x \sin z)$	0
0	x	$y + x \cdot z$	0
-1	$K_{-1}(x \cosh z + y \sinh z)$	$K_{-1}(y \cosh z + x \sinh z)$	0

Vectoring mode

1	$K_1(x^2 + y^2)^{\frac{1}{2}}$	0	$z + \tan^{-1} y/x$
0	x	0	$z + y/x$
-1	$K_{-1}(x^2 - y^2)^{\frac{1}{2}}$	0	$z + \tanh^{-1} y/x$

Table 1

m	p_i sequence ($\delta_i = \pm 2^{-p_i}$)	Max α	K_m	N_m
1	0,1,2,..., n	~1.74	~1.65	N + 2
0	1,2,3,..., n	1.0	1.0	N + 1
-1	1,2,3,4,4,*..., n	~1.13	~.80	N + 1 + R*

*Repeated values are 4,13,...,k, 3k + 1, R denotes the number of repeated values, e.g., if N = 16, R = 2.

Table 2

The values of K_m and $\alpha_i = m^{-\frac{1}{2}} \tan^{-1}(m^{\frac{1}{2}}\delta_i)$, $i = 1,2,\ldots, N_m$, must be available, and as with most algorithms in this class, it is assumed these will be stored in a read-only-memory. Note that in practice, for the algorithm to operate for all three values of m, values of the α_i for all three m, ($\tan^{-1}2^{-i}$, 2^{-i}, and $\tanh^{-1}2^{-i}$) would need to be stored.

The greatest asset of the CORDIC algorithm is its versatility. It is, in fact, even more versatile than would appear at first glance, since functions related to those of Table 1 can be evaluated by pre - or post - manipulation of the data. For example, $\sqrt{w}$ may be obtained by setting $x = w + \frac{1}{4}$, $y = w - \frac{1}{4}$, and entering the algorithm in the vectoring mode with $m = -1$, followed by a division by K_{-1}.

The main disadvantage of the method is that it requires a fixed number of iterations (rotations), whether needed or not, to avoid changing the constant K_m. DeLugish [16] gives a partial solution to the problem, which we will discuss later. Also, the CORDIC algorithm could be used in part, as we note later, in Section 4.3.3, such that the radial distortion constant is eliminated.

3.1.1 Error Reduction for Certain Functions

For functions which are zero when the argument is zero, special procedures must be used to obtain results which are accurate to N significant digits. Generally the normalization procedures handle this automatically, but we will discuss briefly some functions where this is not the case.

When the sine, tangent, or arctangent of a small argument, in floating point form, is desired, we would like the result to be accurate to as many significant digits as possible. The procedures outlined in Section 4 do not result in this desired accuracy. We will note in Section 4 that all the algorithms for these functions reduce to the CORDIC algorithm, essentially, and Walther [66] has treated this problem. The solution is to scale the argument by an appropriate power of two, beginning the rotations with an appropriately small angle and continuing for the usual number of rotations. This requires the inclusion of a set of distortion constants, depending on the initial angle of rotation, as well as additional values for the α_i. The details of the implementation can be found in Walther. The above should also be done for the hyperbolic system if the hyperbolic sine, tangent, and arctangent are desired. If the hyperbolic system is to be used to calculate the exponential, logorithm, and square root, the increased accuracy there is more apparent than real, since the initial or final transformation negates the increased accuracy.

3.2 Pseudo-Division/Multiplication Methods

Consider the division of y by x using the method of successive subtractions. This can be organized (in binary) as follows. Let y and x be normalized so that $\frac{1}{2} \le x,y < 1$. Then $\frac{1}{2} < y/x < 2$. Let the quotient be represented by the number $Q_{n+1} = \sum_{i=0}^{n} q_i 2^{-i}$, where each q_i is 0 or 1. The q_i are determined by the following loop. Let $D_o = y$, $d_o = x$, $Q_o = 0$.

For i = 0,1,..., n, let

$$q_i = \begin{cases} 0 & \text{if} \quad D_i \ge d_i \text{ , or} \\ 1 & \text{if} \quad D_i < d_i \text{ ,} \end{cases}$$

Then $D_{i+1} = 2(D_i - q_i d_i)$

$d_{i+1} = d_i$

$Q_{i+1} = Q_i + q_i 2^{-i}$.

For multiplication, the procedure is reversed. Let $x = \sum_{i=0}^{n-1} x_i 2^{-i}$. Then the product $xy = p_n$ is generated by $p_o = 0$, $y_o = y$, and for

$$i = 0,1,\ldots, n - 1,$$

$$p_{i+1} = 2(p_i + x_i y_i)$$

$$y_{i+1} = y_i$$

The product is assumed to be accumulated in a register of length 2n+1 bits. At the end of the multiplication the product has been shifted n+1 bits, into the most significant n bits of the register, and chopping (or rounding) to the most significant n bits gives the result. Error occurs only when the least significant half is chopped or rounded off.

A pseudo-division or multiplication is a procedure like one of the above, where the q_i's or x_i's are not necessarily the coefficients in radix 2, and the divisior d_i, or multiplicand y_i, may be modified at each iteration.

Meggitt [40] devised a class of these methods for division/multiplication, logarithm/exponential, tangent/arctangent, and square root. While Meggitt's paper was developed for radix 10, the conversion to any other radix is simple.

Meggitt's algorithms correspond to a restoring division. The test for $D_i \geq d_i$ would be done by computing a tentative value of $\frac{1}{2} D_{i+1}$, $\frac{1}{2} D_{i+1}^{(t)} = D_i - d_i$, then testing $\frac{1}{2} D_{i+1}^{(t)}$ for sign. If $\frac{1}{2} D_{i+1}^{(t)}$ is non-negative, $q_i = 1$, and $D_{i+1} = 2(\frac{1}{2} D_{i+1}^{(t)})$. If $\frac{1}{2} D_{i+1}^{(t)}$ is negative, $q_i = 0$ and $D_{i+1} = 2(\frac{1}{2} D_{i+1}^{(t)} + d_i)$. That is, we must "restore" the value of D_i in the latter case.

It is not necessary to restore the value of D_i, however, since it is possible to compute $\frac{1}{2} D_{i+2}^{(t)}$ directly from $D_{i+1}^{(t)}$. Suppose $q_i = 0$, then we have

$$\tfrac{1}{2} D_{i+2}^{(t)} = D_{i+1} - d_{i+1} = 2\, D_i - d_{i+1}$$

$$= 2(\tfrac{1}{2} D_{i+1}^{(t)} + d_i) - d_{i+1}$$

$$= D_{i+1}^{(t)} + d_i - (d_{i+1} - d_i) \; .$$

Depending on $(d_{i+1} - d_i)$, this expression may or may not be easier to compute than by first restoring D_i. In normal division, $d_{i+1} = d_i$, so in that case it is difinitely easier to use the above, non-restoring division. We then compute the sequence $D_i^{(t)}$ rather than D_i, with

$\tfrac{1}{2} D_{i+1}^{(t)} = D_i^{(t)} - d_i$ if $q_i = 1$, or

$\tfrac{1}{2} D_{i+1}^{(t)} = D_i^{(t)} + d_i$ if $q_i = 0$, and that q_i is 1 or 0 as $D_i^{(t)}$ is non-negative or negative, respectively.

Sarkar and Krishnamurthy [55] modified Meggitt's algorithms to correspond to a non-restoring pseudo-division/multiplication. This results in a faster algorithm. It would appear this is more advantageous in radix 10 than in radix 2. The above mentioned paper incorporated a possibility of restoration or non-restoration, depending on which appeared to be more advantageous, i.e., which left the smaller dividend.

We list in Tables 3 and 4 the initialization and iteration equations for Meggitt's algorithms in radix 2. In cases where it is assumed a number is expressed in a variable radix ($\log(1 + 2^{-i})$ or $\tan^{-1} 2^{-i}$), this must be first obtained by a modified division. The procedure given by Meggitt is as follows, where D is to be recoded in one of the above radices.

$$D_o = D$$

For $i = 0,1,\ldots, n - 1$

$$d_i = 2^i \log(1 + 2^{-i}) \text{ , [or : } 2^i \tan^{-1} 2^{-i}]$$

$$q_i = \begin{cases} 1 & \text{if} \quad D_i \geq d_i \\ 0 & \text{if} \quad D_i < d_i \end{cases}$$

$$D_{i+1} = 2(D_i - q_i d_i)$$

At the end of the loop, we have

$$D = \sum_{i=0}^{n-1} q_i \log(1 + 2^{-i}) \text{ , [or: } \sum_{i=0}^{n-1} q_i \tan^{-1} 2^{-i}] \text{ ,}$$

with remainder $D_n/2^n$.

The advantage of the pseudo-division/multiplication processes is that they look very much like multiplication and division, and could be implemented with little additional hardware. The routines are inherently accurate and insensitive to roundoff error, and accuracy comparable to other methods can be obtained with only one guard bit. The use of one double length register is necessary, although with some reformulation this might be replaced by one having only a sufficient number of guard bits, about $\log_2 n+1$.

The evaluation of some functions require a modified division to recode an argument, followed by a psuedo-multiplication, or a pseudo-division, followed by a modified multiplication, and as outlined, some of these must be done serially, not in parallel.

	y/x	log(1+y/x)	$\tan^{-1} y/x$	$\sqrt{y}/x$
Z_o	y	y	y	y
x_o	x	x	x	x
Pseudo-dividend Z_{i+1}	$2(Z_i - q_i x_i)$	$2(Z_i - q_i x_i)$	$2(Z_i - q_i x_i)$	$2(Z_i - q_i x_i)$
Pseudo-divisor x_{i+1}	x_i	$x_i - q_i 2^{-i} x_i$	$x_i + q_i 2^{-2i} Z_i$	$x_i + x q_i 2^{-i+1} - x 2^{-i-1}$
Pseudo-quotient	$\sum_{i=o}^{n-1} q_i 2^{-i}$	$\sum_{i=o}^{n-1} q_i \log(1+2^{-i})$	$\sum_{i=o}^{n-1} q_i \tan^{-1} 2^{-i}$	$\sum_{i=o}^{n-1} q_i 2^{-i}$

Note: $q_i = 0$ or 1 as $Z_i \geq x_i$ or $Z_i < x_i$, respectively.

Table 3: Meggitt Pseudo-division

		$xp+y$	$x(e^P-1)+y$	Tan p	yp^2
Pseudo-Multiplier	p	$\sum_{i=0}^{n-1} q_i 2^{-i}$	$\sum_{i=0}^{n-1} q_i \log(1+2^{-i})$	$\sum_{i=0}^{n-1} q_i \tan^{-1}2^{-i}$	$\sum_{i=0}^{n-1} q_i 2^{-i}$
	Z_o		y		
	Z_n	y		y	
	x_o		x		x
	x_n	x		Q(arbitrary)	
	Z_{i+1}		$2(Z_i+q_i x_i)$		$2(Z_i+q_i x_i)$
Pseudo-product	Z_{i-1}	$1/2\ (Z_i+q_i x_i)$		$1/2\ (Z_i+q_i x_i)$	
	x_{i+1}		$x_i+q_i 2^{-i} x_i$		$x_i-xq_i 2^{-i+1}+x2^{-i-1}$
Pseudo-Multiplicand	x_{i-1}	x_i		$x_i-2^{-2i} q_i Z_i$	

Table 4: Meggitt Pseudo-Multiplication

3.3 Normalization Methods

Consider a function of two variables which is to be subjected to a series of transformations under which the function value is to be invariant. The first variable will be transformed to a specified value while the second is altered in such a way that its value is forced to approach the desired function value.

For example, consider the function $y + \log x$. Let $x_o = x$, $y_o = y$, and define the sequence (x_i, v_i) by $x_{i+1} = a_i x_i$, $y_{i+1} = y_i - \log a_i$, where the a_i are positive constants.

Then we have

$$y + \log x = y_o + \log x_o = y_1 + \log x_1 = \ldots = y_n + \log x_n.$$

If the a_i are chosen in such a way that

$\lim_{n\to\infty} x_n = 1$, we see that $\lim_{n\to\infty} y_n = y + \log x$.

Thus, as x_n is "normalized" to one, y_n is forced to the desired function value.

The above is but one example of a function evaluated by a normalization method. Others amenable to the same approach are given in Table 5, along with appropriate transformations for the variables.

Function	(x_o, y_o)	(x_{i+1}, y_{i+1})	$\lim_{n\to\infty}(x_n, y_n)$
$y + \log x$	(x,y)	$(a_i x_i, y_i - \log a_i)$	$(1, y + \log x)$
y/x	(x,y)	$(a_i x_i, a_i y_i)$	$(1, y/x)$
$w/x^{\frac{1}{2}}$	(x,y)	$(a_i^2 x_i, a_i y_i)$	$(1, y/x^{\frac{1}{2}})$
ye^x	(x,y)	$(x_i - \log a_i, a_i y_i)$	$(0, ye^x)$

Table 5: Normalization Methods

A number of authors have addressed themselves to the implementation of the normalization methods, including (in historical order), Specker [61], Perle [46], DeLugish [16], and Chen [5]. The principal matter to be decided is how the a_i should be chosen. Clearly a_i is to be of the form $1 + s_i 2^{-p}i$, where p_i is an integer, and $s_i = \pm 1$ or 0. Ideally, the a_i should be chosen so that x_n is forced to within a specified tolerance of its limit value for as small an n as possible. We shall discuss the various strategies for choosing the a_i as we consider the various functions.

Chen suggests the use of a termination algorithm which decreases the number of iterations required. Essentially the idea is to use a one term Taylor series expansion when the most significant half of the number has been computed. This is generally done at the expense of a half precision multiply (i.e., the most significant half of the multiplier is represented by all zero bits). It does have an additional advantage in that it halves the round-off error accumulation since only about one-half as many iterations are required. We will discuss the individual termination algorithms in the next section.

The advantage of the normalization methods is their simplicity (depending on how one decides on a_i), and their potential speed. Of course they are not as versatile as the CORDIC algorithm and no normalization methods have been devised for trigonometric functions.

4.0 Algorithms for Specific Functions

4.1 Quotient

There are several kinds of division algorithms. We have previously mentioned the CORDIC, successive subtraction, and normalization algorithms. Another normalization method using multiplication and based on Newton's method has been proposed. This procedure converges quadratically, and thus is best suited when high precision is required. Flynn [20] discusses a number of iterative techniques for division. Another method is based on a reciprocal generator. Huber [24] did a thesis on binary division algorithms which discusses most of the above methods.

Suppose that we desire the quotient X/Y , where $Y = 2^{\alpha}\cdot y$, $X = 2^{\alpha}\cdot x$, with α,β integers and $\frac{1}{2} \le x,y < 1$, that is X and Y are expressed in normalized form. In some algorithms, we will assume $y < x$, relaxing the requirement that both x and y be in $[\frac{1}{2},1)$. This leads to the quotient being in $[\frac{1}{2},1)$, hence no normalizing shift would be required for the quotient. In any case, we concern ourselves with the generation of the quotient y/x.

4.1.1 CORDIC Algorithm

Inspection of Tables 1 and 2 reveal that N + 1 iterations are required. Because of the sequence of α's for this case we must have $y < x$; this can be accomplished by right shifting y one place, if necessary, or always. To avoid a possible left shift to normalize the quotient, the right shift of y should be done only if necessary.

After $N_o = N + 1$ iterations we obtain

$$y_{N_o+1} = y_o - x_o \sum_{i=0}^{N_o} s_i\alpha_i,$$

then $|y_{N_o+1}| = |y_o - x_o \sum_{i=0}^{N_o} s_i\alpha_i| \le x_N\alpha_N = x_o\alpha_{N_o}$.

Since $z_{N_o+1} = -\sum_{i=0}^{N_o} s_i\alpha_i$, the above yields, with $z_o = 0$,

$|{}^{y_o}/{}_{x_o} - z_{N_o+1}| \le \alpha_{N_o} = 2^{-N-1}$. Thus the truncation error is bounded by 2^{-N-1}.

If a left shift were necessary to normalize the quotient, the error would become as large as 2^{-N+1}, thus we see why it is important to right shift Y only if necessary.

4.1.2 Restoring and Non-restoring Division

It would again be advantageous to have $y < x$ and y/x in $[\frac{1}{2},1)$ to avoid a normalization shift at the end of the operation. Also, one would then avoid generating $N + 1$ quotient bits when only N are necessary, since the first bit would always be 1.

If N quotient bits are generated, there will be as many as N subtractions of the divisor from the dividend. Assuming left shifts of the dividend rather than right shifts of the divisor, no roundoff error will occur during these operations. The truncation error will be determined by the size of the remainder divided by the divisor, and will be bounded by $K \cdot 2^{-N-1}$, where

$$K = \begin{cases} 1 & \text{if a final round is performed} \\ 2 & \text{if chopping is used} \end{cases}$$

Thus the error is at most one bit.

If $N + 1$ quotient bits are generated with a possible right shift required to normalize the quotient, the error bound is the same.

Nandi and Krishnamurthy [44] proposed a procedure for a non-restoring type of division for radix β. It obtained the quotient in redundant form, and was especially designed for divisors with leading coefficient 1 or $\beta - 1$. While the procedure could be adapted for radix 2, there is a conflict in the decision process, which differs depending on whether the leading coefficient of the divisor is 1 or $\beta - 1$. These digits coincide in radix 2, of course.

The decision process for the quotient bits is somewhat complicated, as well.

The advantage of redundant representation (using ± 1 and 0 as coefficients, instead of just 1 and 0) of function values is that more zero bits can be "built in". Every number has a minimal representation, i.e., a representation with a minimal number of non-zero bits. Metze has investigated this idea for several functions, one of which is division [41]. Basically the idea is an extension of non-restoring division, and contains S-R-T division [50] as a special case. The decision as to the value of the quotient bit is based on the value of the previous remainder. The decision is somewhat complicated, with the comparison constant a function of the divisor.

The following procedure is that given by Metze for minimally represented quotients. The comparison constant, K, is obtained from Table 6. We first assume that $\frac{1}{2} \leq x < 1$ and $0 < y < x$. We just as well assume that $\frac{1}{2} \leq y/x < 1$. Let $Q = y/x = \sum_{i=0}^{n} q_i 2^{-i}$, $Q_{-1} = 0$, $2R_{-1} = x$. For $i = 0,1,\ldots, N$, determine q_i by

$$q_i = \begin{cases} 1 & \text{if} \quad 2R_{i-1} \geq K \\ -1 & \text{if} \quad 2R_{i-1} < K \\ 0 & \text{otherwise} \end{cases}$$

Then $2R_i = 2(2R_{i-1} - q_i\ x)$

$$Q_i = Q_{i-1} + q_i 2^{-i}$$

The comparison constants given in Table 6 are not particularly convenient, but it is the smallest number of different ones possible. Another, more convenient set is given in Table 7.

For S-R-T division the comparison constant is $K = \frac{1}{2}$ for all divisors, and a minimally represented quotient is obtained for divisors in the range [3/5, 3/4].

Range of divisor	K
[1/2, 39/64)	13/32
[39/64, 3/4)	1/2
[3/4, 15/16)	5/8
[15/16, 1)	3/4

Table 6

Range of divisor	K
[1/2, 9/16)	3/8
[9/16, 5/8)	7/16
[5/8, 3/4)	1/2
[3/4, 15/16)	5/8
[15/16, 1)	3/4

Table 7

A minimal recoding can be obtained by requiring that non-zero bits be separated by at least one zero, thus at most $[\frac{N+1}{2}]$ bits can be non-zero (see [41], first paragraph). Because of the left shift of the remainder, R_k, no roundoff error is introduced. It is easy to prove that $y = x - Q_k + 2^{-k}R_k$ thus for $k = N$, we have truncation error $y/x - Q_N = \frac{2^{-N}R_N}{x}$. We show by induction that $|R_k| < x$. $R_{-1} = x/2$, and assume that $|R_{k-1}| < x$. Then if $q_k = 0$, $R_k = 2R_{k-1}$, and $|R_k| = |2R_{k-1}| < K$, by the selection rule for q_k, and the recursion formula. Tables 6 and 7 both show that $K < x$, thus $|R_k| < x$. If $q_k = \pm 1$, we have $R_k = 2R_{k-1} - \text{sign}\ (R_{k-1})x$, or $|R_k| = |2|R_{k-1}| - x| < x$, since by induction $|R_{k-1}| < x$. Thus $|R_k| < x$ for all k, and in particular, $|R_N| < x$, thus the truncation error $\frac{2^{-N}R_N}{x} < 2^{-N}$, or less than one in the least significant bit.

Note that no shift for normalization is required. In order to be assured that $y < x$, however, it may be necessary to right shift y initially. If no guard bit is provided, this could cause an error of 2^{-N-1} in y, and since $2^{-N-1}/x < 2^{-N}$, we see that the error could be as large as 2^{-N+1}, or the last two bits could be in error. The error in y can be eliminated with one gaurd bit.

Metze makes no mention of the average number of non-zero quotient bits. The probability of a zero occuring in S-R-T division is about .63, and Metze's algorithm must do better than that. We noted previously we must have about one-half of the digits equal to zero, although it is unlikely that in the average case, half of the remaining digits will also be zero. The probable number of non-zero bits is likely between N/4 and N/3.

4.1.3 Normalization Methods

As we noted earlier, the principal matter to be decided here is a strategy for the choice of a_i(see Table 5). We assume that $a_i = 1 + s_i 2^{-p_i}$, where $s_i = \pm 1$ or 0, and p_i is an integer. The iteration procedure is $x_o = x$, $y_o = y$, and for $i = 0,1,2,\ldots, n-1$

$$x_{i+1} = x_i a_i$$

$$y_{i+1} = y_i a_i \ .$$

Further we assume that $\frac{1}{2} \le x < 1$. We discuss several strategies for choice of a_i.

The first is very simple, and reminiscent of the CORDIC algorithm.

$$a_i = \begin{cases} 1 + 2^{-i} & \text{if} \quad x_i < 1 \\ 1 - 2^{-i} & \text{if} \quad x_i \ge 1 \end{cases}$$

It is easily seen that $1 - 2^{-i} \le x_{i+1} < 1 + 2^{-i}$, thus after n iterations

$$|y_{n+1} - y_o/x_o| = |y_{n+1} - y_{n+1}/x_{n+1}| = |\frac{x_{n+1}-1}{x_{n+1}} \cdot y_{n+1}| \le 2^{-n} \frac{y_{n+1}}{x_{n+1}} = 2^{-n} y/x \ .$$

Thus, the relative error is 2^{-n}, and if x and y are normalized as usual, $\frac{1}{2} < y/x < 2$, hence the truncation error is less than one in the $(n-1)^{st}$ significant bit. Roundoff error would accumulate up to $\log_2 n + 1$ bits, thus if $n = N + 1$ iterations are used, $\log_2(N + 1) + 1$ guard bits would give an error no greater than one in the last bit.

Another similar strategy is

$$a_i = \begin{cases} 1 + 2^{-i} & \text{if} \quad x_i(1 + 2^{-i}) < 1 \\ 1 & \qquad \text{otherwise} \end{cases}$$

This division procedure is the analogue of the Specker [61] and Perle [46] algorithms. It has the seeming advantage of skipping some iterations (if $a_i = 1$), and the real advantage of keeping $x_i < 1$. However, note that the decision requires calculation of a tentative value of x_{i+1}, hence no saving is obtained there. The roundoff error would be decreased in the average case, but not the bound. Truncation error is the same, although one knows its sign and could compensate after the last iteration.

DeLugish [16] gives a more sophisticated strategy for choice of a_i. DeLugish normally requires an initialization procedure. In this case it is

$$a_o = \begin{cases} 2 & \text{if} \quad 1/2 \le x_o < 3/4 \\ 1 & \text{if} \quad 3/4 \le x_o < 1 \end{cases} ,$$

and $x_1 = a_o x_o$, $y_1 = a_o y_o$.

Now, as one would likely do in practice in the previous examples, DeLugish does not actually compute the sequence of x_i's. Rather, for ease in testing, we compute the sequence $u_i = 2^i(x_i - 1)$, and then $u_{i+1} = 2u_i + s_i + s_i u_i 2^{-i}$. Here $p_i = i+1$, and s_i is determined by

$$s_i = \begin{cases} 1 & \text{if} \quad u_i < -3/8 \\ -1 & \text{if} \quad u_i \ge 3/8 \\ 0 & \text{otherwise} \end{cases}$$

Delugish shows that $|x_i - 1| \le 3/8 \cdot 2^{-i+1}$ for $i \ge 2$, thus $|x_{N-1} - 1| \le 3/8 \cdot 2^{-N}$, so truncation error is no more than 3/4 that derived for the earlier examples. No study was made of roundoff error, but the bound depends on the number of iterations with $s_i \neq 0$. DeLugish shows the probable number of non-zero s_i's is about N/3. The maximum number is not discussed. However, it is easy to show that not more than two successive s_i can be non-zero. For $i \geq 2$, $|x_i - 1| \leq 3/8 \cdot 2^{-i+1}$, thus $|u_i| \leq 2^i \cdot 3/8 \cdot 2^{-i+1} = 3/4$. Now, suppose $u_i > 3/8$ and thus $s_i = -1$. Then $u_{i+1} = 2u_i + s_i + s_i u_i 2^{-i} \leq 2 \cdot 3/4 - 1 = 1/2$. Also we have $u_{i+1} \geq (2 - 1/4) \cdot 3/8 - 1 = -11/32 > -3/8$. Then, suppose $3/8 < u_{i+1} \leq 1/2$, hence $s_{i+1} = -1$, since otherwise $s_{i+1} = 0$. Then $u_{i+2} = 2u_{i+1} + s_{i+1}u_{i+1}2^{-i-1} \leq (2 - 1/8) \cdot 1/2 - 1 = -1/16$, and as before $u_{i+2} > -3/8$. Thus $s_{i+2} = 0$, as was to be proved. If u_i is negative, the argument is symmetric and the result follows. Thus we have that $\log_2 \frac{2N}{3} + 1$ guard bits are sufficient, with chopping.

The method suggested by Chen [5] used a different approach. Here we must use some sort of procedure for counting left zeros in $1 - x_i$. We write $1 - x_i = 2^{-p_i} + v_i$, where $0 \leq v_i < 2^{-p_i}$. Then p_i is one plus the number of left zeros in $1 - x_i$. With $a_i = 1 + 2^{-p_i}$ we have

$$x_{i+1} = (1 - 2^{-p_i} - v_i)(1 + 2^{-p_i}) = 1 - v_i(1 + 2^{-p_i}) - 2^{-2p_i} .$$

Thus we see that the transformation has eliminated the leading non-zero bit of $1 - x_i$, and leaves the succeeding bits largely unchanged. Thus, in no more than $N + 1$ iterations, $1 - x_i < 2^{-N-1}$. On the average, one expects only about $N/2$ iterations would be necessary. The error bound here is the same as for the first two ideas presented.

However, Chen suggests a termination algorithm, to be applied when $p_i > N/2$. It is simply a Taylor series expansion about the then current point. In this case, if we have computed $(x_o,y_o),(x_1,y_1),\ldots,(x_n,y_n)$, with $p_n > N/2$, then $y/x \approx y_n + y_n(1 - x_n)$. This requires a multiply, however note that since $p_n > N/2$, $1 - x_n$ is a half precision number (i.e., the most significant $N/2$ bits are zero), thus costing half a usual multiply time.

The truncation error bound is 2^{-N}. Chen notes that since the error is of a definite sign, we could halve the truncation error by computing

$$y/x \approx y_n + y_n (1 - x_n + 2^{-N-1}) .$$

The roundoff error is halved, of course, since at most N/2 iterations need to be performed. Hence $\log_2 N$ guard bits are sufficient with chopping. The expected number of iterations is N/4.

4.1.4 Quardratically Converging Normalization Methods

The methods of Section 2.1.3 converge linearly, that is, the number of iterations required depends linearly on the number of bits of accuracy required. Quadratically convergent normalization methods are based on the Newton method for solving $1/z - 1 = 0$ with initial guess $z_o = x_o = x$, where x is the divisor. The calculations can be arranged as $y_o = y$, $x_o = x$, then for $i = 1,2,\ldots, n$,

$$x_i = x_{i-1}(2 - x_{i-1})$$

$$y_i = y_{i-1}(2 - x_{i-1}) \quad .$$

The error $1 - x_i = E_i = (1 - x_o)^{2^i}$. Thus convergence is critically tied to the accuracy of the "initial guess". We must have $|1 - x_o| < 1$ for convergence.

We see that if $0 < x_o < 2$, convergence is assured, hence $x = x_o$ is adequate for convergence, with x in $[1/2,1)$.

We pause to consider the number of iterations required. If we want $E_n < 2^{-N}$, then n must be taken so that $(1 - x_o)^{2^n} < 2^{-N}$, or

$$2^n > \frac{N}{-\log_2|1-x_o|} \quad .$$

For x_o in $[1/2,1),(1 - x_o) \le 1/2$, thus no more than $\log_2 N$ iterations would be required.

We could improve this a little by some initial adjustment of x and y. Thus we might take $x_o = ax$, $y_o = ay$, where a may or may not be a function of x. To avoid an actual multiplication here, we could take

$$a = \begin{cases} 2 & \text{if} \quad 1/2 \le x < 2/3 \\ 1 & \text{if} \quad 2/3 \le x < 1 \ . \end{cases}$$

Then $|1 - x_o| \le 1/3$, and $\log_2 \frac{N}{\log_2 3} \simeq \log_2 \frac{N}{1.58}$ iterations would then be required. For N = 20, five iterations would be required by the previous procedure, and four iterations plus the initialization for the latter procedure. This example is biased somewhat, however. Note that if N is a power of two, no advantage is had by first initializing as above. For N a power of 2, one would need to initialize so that $|1 - ax| \le 2^{-2^k}$ to save k iterations.

In general the roundoff error doubles at each iteration. Thus for roundoff error to be less than 2^{-N-1} requires that J guard bits be used, where $2^n(2^{-N-J}) < 2^{-N-1}$, or $J = n + 1$ with chopping.

Krishnamurthy [29] has considered solving $1/z - 1/K = 0$ instead of $1/z - 1 = 0$. For a simple recursion for the x_i's, this necessitates $K = 2^{-r}$ for some integer r, although K = 2/3 is possible. He also considers using lower precision multiplies for early iterations. A table of initial transformations is given for computing a 48 bit quotient in 4 iterations (plus the initial transformation).

The idea of a table for the initial transformation is used in the IBM System 360 Model 91 [1]. With that machine 56 bit accuracy is desired, and a table of $2^7 = 128$ entries is used and gives $(1 - x_o) < 2^{-7}$. Thus, after the initial transformation, only three iterations would be required. However, in order to speed up the early iterates, low precision multiplications are used with truncated intermediate results, and four iterations lead to 64 bit accuracy.

Ferrari [19] proposed a method he calls the optimized geometric series (OGS) method. If the two multiplications are to be done serially, his method is faster, since convergence is superquadratic. If the multiplications can be done simultaneously, his method has no advantage.

Shaham and Riesel [58] suggest a modification of the Newton iteration. Suppose that $1 - x_i = 2^{-k} + v_i$, with $0 \le v_i < 2^{-k}$. Then normally one would have $1 - x_{i+1} = 2^{-2k} + v_{i+1}$, with $0 \le v_{i+1} < 2^{-2k}$. They suggest using as a multiplier, $2 - x_i + p2^{-2k}$, instead of $2 - x_i$, where p is a table-look-up value depending on the first few bits of v_i. They show, for example, that a 56 bit result can be obtained in one less iteration than with the classical method, starting with an initial result accurate to six bits.

Ling [33] has given a version of this algorithm which was purportedly designed especially for 32 bit accuracy, and yields the quotient in 3 multiplication times. Ling's method is to first transform the divisor, $x = .1\delta_2\delta_3...\delta_8 + 2^{-8}(.\delta_9...\delta_n)$ into a form $x = .d_1...d_7 \pm 2^{-8}(.d_9..d_n)$, i.e., with the eighth bit equal to zero. We see that if $\delta_8 = 0$, the plus sign is taken, and $d_i = \delta_i$. If $\delta_8 = 1$, the minus sign is taken, and $.d_1..d_7 = .1\delta_2...\delta_8 + 2^{-8}$, $.d_9...d_n = (1 - .\delta_9...\delta_n)$. If $\delta_2 = \delta_3 = ... = \delta_8 = 1$, there is an apparent difficulty here, which washes out later on. Ling implements the above via logical circuits rather than by actual addition and complementation. The initial transformation is then $y/x = \frac{y(2k)}{x(2k)}$, where

$k = 1/2 \cdot (.d_1..d_7)^{-1}$ is obtained by table-look-up. We then have $x(2k) = 1 \pm 2^{-7}k(.d_9...d_n)$, the ± having been determined previously. Ling indicates that the denominator is within 2^{-8} of one, thus within 2^{-32} of one after two Newton iterations. This is not the case however, as can be seen by considering $x = .1000000011...1$, whence $k = 1$, and $2xk = 1 + 2^{-7}(.111...1)$. Thus Ling's algorithm is good for at least 28 bits, but one cannot be sure of more than 28 bits.

4.1.5 Other Methods

Dean [10] proposes a reciprocal generator which forms the reciprocal of a number in a controlled register as the number is entered serially into a shift register. Formation of the quotient bits is done by means of logical circuits. He gives complete information for four, five, and six bit numbers. The logic would seem to become quite complicated for high precision numbers. However, if, as Dean says, the size of connectors negates the possiblity of components becoming much smaller, but rather their complexity increasing to take up (some of) the available space, such a device might be feasible where speed is the main consideration. However, I am assuming the logical circuits could be implemented so as to be very fast.

4.2 Arctangent

Several authors have directed their attention to calculation of the arctangent. While it is not readily apparent when glancing through the literature, it is in fact true that only one algorithm has been developed, so far as this author has been able to determine. The CORDIC method was the first to appear, followed by Meggitt [40] and Sarkar and Krishnamurthy [55], Specker [61], and DeLugish [16]. The algorithms differ mainly from the CORDIC algorithm in that not all of the rotations are performed.

Meggitt keeps $z_i \geq 0$, Sarkar and Krishnamurthy minimize $|z_i|$, Specker's method coincides with the CORDIC algorithm, and DeLugish uses a decision process similar to that for division.

We assume that $\tan^{-1} y/x$ is to be computed, where $x \neq 0$. We will generally assume that a first quadrant angle is to be computed, hence an initial transformation using the identity $\tan^{-1}(-y/x) = -\tan^{-1} y/x$ may be necessary. For the CORDIC algorithm, we must have $x > 0$, so the sign is attached to whichever of x or y is appropriate. If the result of the calculation must be in the correct quadrant, additional care must be taken with that step, and furthermore, an additional rotation through π radians may be necessary. In the CORDIC algorithm this can be accomplished with no post-manipulation of the data.

4.2.1 CORDIC Algorithm

As Table 1 indicates, $\tan^{-1} y/x$ is obtained by the algorithm with $m = 1$ in the vectoring mode. Since we have discussed the general procedure in some detail in Section 3.1, we need only to discuss the truncation error bound for this particular calculation. As we indicated, the magnitude of y_{N_1+1} can be as large as $x_{N_1} \alpha_{N_1}$ in vectoring mode. Now $x_{N_1} \approx K_1(x_o^2 + y_o^2)^{1/2}$, so $|y_{N_1+1}| \leq K_1(x_o^2 + y_o^2)^{1/2} 2^{-N-1}$. Consideration of equation (6) and some algebraic manipulation yields $\tan \alpha = y_o/x_o - \dfrac{y_{N_1+1}}{x_o K_1 \cos\alpha}$. Thus we have

$$\alpha = \tan^{-1}[y_o/x_o - \gamma] = \tan^{-1} y_o/x_o - \frac{\gamma}{1 + r^2} \quad ,$$

where r is between y_o/x_o and $y_o/x_o - \gamma$. Thus the truncation error is

$$- \frac{\gamma}{1 + r^2} \approx - \frac{\gamma}{1 + y_o^2/x_o^2} \quad .$$

Then $\left|\frac{\gamma}{1+y_o^2/x_o^2}\right| \leq \frac{x_o^2 \cdot K_1(x_o^2+y_o^2)^{1/2}2^{-N-1}}{x_oK_1 \cos \alpha(x_o^2+y_o^2)} = \frac{x_o 2^{-N-1}}{(x_o^2+y_o^2)^{1/2}\cos \alpha}$.

But $\frac{x_o}{(x_o^2+y_o^2)^{1/2}} \approx \cos \alpha$, hence the truncation error is no more than (approximately) 2^{-N-1}. Roundoff error is likewise bounded, hence total error is no more than 2^{-N}.

We will consider the problem of initializing for both principal values of the arctangent and for values in the correct quadrant, the quadrant determined by the signs of y and x.

For principal value we let

$$x_o = x \text{ sign } x$$
$$y_o = y \text{ sign } x$$
$$z_o = 0 \quad .$$

For the correct quadrant to be obtained, we take

$$x_o = x \text{ sign } x$$
$$y_o = y \text{ sign } x$$
$$z_o = \begin{cases} \pi & \text{if} \quad \text{sign } x < 0 \\ 0 & \text{if} \quad \text{sign } x > 0 \; . \end{cases}$$

One can also accomplish the initialization by always doing an initial rotation of $\pi/2$ radians, clockwise if $y > 0$, and counterclockwise if $y < 0$. This will also force $x_o > 0$. Then

$$x_o = y \text{ sign } y$$
$$y_o = - x \text{ sign } y$$
$$z_o = \pi/2 \text{ sign } y \quad .$$

4.2.2 Modified CORDIC Algorithm.

We will discuss the various modifications of the CORDIC algorithm in a unified manner. Basically, all of the modifications can be put in the same form as equations (3) - (5), except that the decision for s_i is different. This will change the radial distortion constant, but it does not enter significantly in this case, anyway.

Meggitt takes

$$s_i = \begin{cases} \text{sign } y_i & \text{if } |y_i| \geq \delta_i x_i \\ 0 & \text{otherwise} \end{cases} .$$

Sarkar and Krishnamurthy take

$$s_i = \begin{cases} \text{sign } y_i & \text{if } |y_i - \text{sign } y_i \cdot \delta_i x_i| < |y_i| \\ 0 & \text{otherwise} \end{cases} .$$

DeLugish considers $\tan^{-1} x$, $0 \leq x < 1$, and takes

$$s_i = \begin{cases} +1 & \text{if } y_i < -3/8\ 2^{-i} \\ -1 & \text{if } y_i \geq 3/8\ 2^{-i} \\ 0 & \text{otherwise.} \end{cases}$$

with the initialization procedure

$$x_{-1} = 2^{\beta}(x - 1)\ ,\ y_{-1} = 2^{\beta}(x + 1)\ ,$$

with β chosen so that x_{-1} is in $[-1,0)$.

Then y_{-1} is in $[1,2)$. We set $z_o = \pi/4$,

$$x_o = x_{-1} t_o,\ y_o = y_{-1} t_o\ , \quad \text{where}$$

$$t_o = \begin{cases} 3/4 & \text{if} \quad 0 \le x < 1/4 \\ 5/8 & \text{if} \quad 1/4 \le x < 1/2 \\ 1/2 & \text{if} \quad 1/2 \le x < 1 \end{cases}$$

Note that the sign of the s_i is different because $x_i < 0$ for his initialization.

The error bounds for all of the above are the same as for the CORDIC algorithm, and the roundoff error will probably be smaller in practice because the average number rotations is less than N+2. For the DeLugish method the average number of rotations (number of non zero s_i) is about N/3.

4.3 Cosine/Sine

As with the arctangent algorithms, all of the algorithms for cosine/sine reduce to the CORDIC algorithm, or a variation of it. We consider that the angle, θ, could be any angle. This is then reduced to the range $[0,2\pi)$. Let the reduced angle be denoted by Z. Then we can find z in $[0,\pi/2)$ so that $Z = \pi/2 \cdot Q + z$, where Q = 0,1,2, or 3. We compute sin z and cos z, and then

$$\sin Z = \sin \theta = \begin{cases} \sin z & \text{if} \quad Q = 0 \\ \cos z & \text{if} \quad Q = 1 \\ -\sin z & \text{if} \quad Q = 2 \\ -\cos z & \text{if} \quad Q = 3 \end{cases}$$

and

$$\cos Z = \cos \theta = \begin{cases} \cos z & \text{if} \quad Q = 0 \\ -\sin z & \text{if} \quad Q = 1 \\ -\cos z & \text{if} \quad Q = 2 \\ \sin z & \text{if} \quad Q = 3 \end{cases}$$

The proper values can be obtained either by pre - or post - manipulation of the data.

4.3.1 CORDIC Algorithm

By Table 1 we see that if one enters the algorithm with $m = 1$, $x = K_1^{-1}$, $y = 0$, and z in the rotation mode, one obtains cos z and sin z for $|z| \leq \pi/2$. From equation (6)

$$x_{N_1+1} = K_1(K_1^{-1} \cos \alpha) = \cos \alpha$$

$$y_{N_1+1} = K_1(-K_1^{-1} \sin \alpha) = - \sin \alpha \ .$$

Now $|z + \alpha| = |z_{N_1+1}| \leq \alpha_{N_1}$, thus

$$x_{N_1+1} = \cos (z - \gamma) = \cos z + \sin z^* \cdot \gamma,$$

$$y_{N_1+1} = \sin (z - \gamma) = \sin z - \cos \bar{z} \cdot \gamma,$$

where z^* and $\bar{z}$ are each within $|\gamma|$ of z, and $|\gamma| \leq \alpha_{N_1} = \tan^{-1} 2^{-N-1} \approx 2^{-N-1}$. Then, since sine and cosine are bounded by one, the truncation error is bounded by 2^{-N-1}. If roundoff error is also bounded by 2^{-N-1}, total error is bounded by 2^{-N}.

4.3.2 DeLugish Modification

Because the distortion constant K_1 depends on the rotations performed, ϵ_{R_1} the usual DeLugish [16] modification will not work. Recall that the distortion in the step

$$x_{i+1} = x_i + s_i \delta_i y_i$$

$$y_{i+1} = y_i - s_i \delta_i x_i$$

is $(1 + \delta_i^2)^{1/2}$. A correction to remove the distortion would require division of each variable by $(1 + \delta_i^{2})^{1/2}$. This is not feasible, of course. But $(1 - \delta_i^{2})^{-1/2} = 1 - 1/2\ \delta_i^{2} + 3/8\ \delta_i^{4} \ldots$, and if δ_i is small enough for the third term to be less than 2^{-N-1}, we have $(1 + \delta_i^{2})^{-1/2} = 1 - 1/2\ \delta_i^{2}$, rounded to N digit accuracy, and such a correction could be made with a shift and subtract, since δ_i is a power of two. Eventually, $(1 + \delta_i^{2})^{-1/2} = 1$, rounded to N bit accuracy, and then no correction is required. This idea is used by DeLugish to introduce redundancy into the cosine/sine calculation.

We consider $0 \le z < \pi/2$. The initialization is

$$z_o = \begin{cases} z & \text{if} \quad z \le \pi/4 \\ \pi/2 - z & \text{if} \quad z > \pi/4 \end{cases}$$

then

$$(x_1, y_1) = \begin{cases} (1/K^* ,\ (1/K^*) \tan \pi/8) & \text{if } z \le \pi/4 \\ ((1/K^*) \tan \pi/8,\ 1/K^*) & \text{if } z > \pi/4 \end{cases}$$

and $\quad z_1 = z_o - \pi/4,$

where $\quad K^* = (1 + \tan \pi/8)^{1/2} \prod_{j=0}^{N^*} (1 + 2^{-j+1})^{1/2}$.

Here $N^* = [\frac{N-6}{4}] + 1$, the number of iterations to be performed before shift and subtract corrections can be made for the radial distortion. The iteration equations are

$$x_{i+1} = (x_i - s_i y_i 2^{-i-1}) T_i$$

$$y_{i+1} = (y_i + s_i\ x_i 2^{-i-1}) T_i$$

$$z_{i+1} = z_i - s_i \tan^{-1} 2^{-i-1}$$

where s_i is determined by

$$s_i = \begin{cases} -1 & \text{if} \quad z_i < 0 \\ 1 & \text{if} \quad z_i \geq 0 \end{cases} ,$$

for $i = 1,2,.., N^*$,

$$s_i = \begin{cases} -1 & \text{if} \quad z_i < -\ 3/8\ 2^{-i} \\ 1 & \text{if} \quad z_i > 3/8\ 2^{-i} \\ 0 & \text{otherwise} \end{cases}$$

for $i = N^* + 1,\ldots, N$

and

$$T_i = \begin{cases} 1 & \text{for} \quad i = 1,2,\ldots,N^* \\ 1 - s_i^2\ 2^{-(2i+3)} & \text{for } i = N^*+1,\ldots,\ N^{**} \\ 1 & \text{for} \quad i = N^{**}+1,\ldots,\ N\ . \end{cases}$$

Here $N^{**} = [\frac{N-3}{2}] + 1$, the point at which $(1 + \delta_i^2)^{-1/2} = 1$, to N bit accuracy.

The above was as given by DeLugish. DeLugish, however, did not consider the effects of roundoff error, and his examples were done with $N = 40$, and at least 13 guard bits. For purposes of roundoff error control, the values of N^* and N^{**} should be determined on the basis of $N + J$ bit accuracy, rather than N bit accuracy. Thus, one should take N^* and N^{**} to be $[\frac{N+J-6}{4}] + 1$ and $[\frac{N+J-3}{2}] + 1$, respectively.

The truncation error is similar to the CORDIC algorithm. The roundoff error bound is complicated by the fact that about $\frac{N+J}{4}$ iterations are performed which may require corrections T_i, with $T_i \neq 1$. It is conceivable that nearly all rotations (all s_i) will be non-zero, thus the roundoff error bound will be about 25% larger. However, the avarage number of s_i equal to zero should be about $\frac{3N}{8}$, so on the average, roundoff error will be smaller.

4.3.3 An Alternate Procedure

We have seen in the previous section that the radial distortion constant complicates the usage of no rotation at any step. If one were to use a scheme such as the DeLugish scheme for computing the tangent (See Section 4.7.2), the sine could then be computed from $\sin z = \frac{\tan z}{(1 + \tan^2 z)^{1/2}}$. This type of algorithm was implemented in the Hewlett-Packard HP-35 calculator [6]. In addition to the fact that the calculations are radix 10, the goal there was compactness of the program, rather than speed. The above identity is certainly not likely to achieve high speed.

4.4 Exponential

The calculation of the exponential function has been treated by several authors. Normalization is the principal technique, although the other unified algorithms also can be used.

We will generally allow the argument to be any number, X. The range will be reduced by the transformation $X = Q \log 2 + x$,* where Q is an integer, and $0 \le x < \log 2$. Then $e^X = e^{Q \log 2 + x} = 2^Q e^x$. Note that $1 \le e^x < 2$.

4.4.1 CORDIC Algorithm

Table 1 yields the information that entering the algorithm with m = - 1 in the rotation mode, taking $x = 1/K_{-1}$, $y = 0$, and z = argument, the result of the calculation will be $x_{N_{-1}+1} \approx \cosh z$, $y_{N_{-1}+1} \approx \sinh z$. Then $e^z = \cosh z + \sinh z \approx x_{N_{-1}+1} + y_{N_{-1}+1}$.

*We will use $\log u$ to denote $\log_e u = \ln u$, while any other base will be specified by subscript.

The truncation errors are given by

$$\begin{aligned} x_{N_{-1}+1} - \cosh z &= \cosh \alpha - \cosh z \\ &= \cosh (z_o - z_{N_{-1}+1}) - \cosh z \\ &= - (\sinh z^*)\, z_{N_{-1}+1} , \end{aligned}$$

and similarly,

$y_{N_{-1}+1} - \sinh z = - (\cosh \bar{z})\, z_{N_{-1}+1}$, where z^* and $\bar{z}$ are between z and $z - z_{N_{-1}+1}$. Then, the truncation error for e^z is the sum,

$- z_{N_{-1}+1}(\sinh \bar{z} + \cosh z^*) \simeq - e^z z_{N_{-1}+1}$. Since $|z_{N_{-1}+1}| \leq 2^{-N-1}$ and $0 \leq z < \log 2$, the bound is $2 \cdot 2^{-N-1} = 2^{-N}$. Roundoff error in each is 2^{-N-1} , hence the total roundoff error could be as large as 2^{-N}, giving a total error of no more than 2^{-N+1}. However recall that a right shift is necessary to normalize the result, thus the error is then bounded by 2^{-N}.

4.4.2 Pseudo-Division/Multiplication Methods

From Table 4 we see it is first necessary to recode the exponent in in the variable radix $\log (1 + 2^{-i})$ This is accomplished by the modified divider discussed in Section 3.2. Let us first investigate the error in the process of recoding the exponent. Without rounding error, and chopping after N bits it is easily seen that the truncation error in the process is bounded by $\log (1 + 2^{-N}) \approx 2^{-N}$. The roundoff error is introduced solely through the divisors, and is left shifted in the new dividend at each iteration. If the error in divisor d_i is ϵ_i, we have error

$- 2^N q_o \epsilon_o - 2^{N-1} q_1 \epsilon_1 - \ldots - 2 q_N \epsilon_N$ in D_{N+1}. Thus the error accumulation in remainder is the above, divided by 2^{N+1}. Assuming correct rounding of the divisors the roundoff error accumulation is bounded by

$$2^{-N-1}(2^{N} + \ldots + 2)2^{-N-1} \leq = 2^{-N-1}.$$

Thus if no guard bits are used, roundoff error is less than 2^{-N-1}. Thus the total error in this procedure could be as large as $2^{-N} + 2^{-N-1}$. If we take y = x = 1 in the procedure, we obtain $e^{p-\gamma}$, where $|\gamma| \leq 3/2 \cdot 2^{-N}$. Thus the error here is $e^{p} - e^{p-\gamma} = e^{p^{*}}\gamma$, where p^{*} is near p. Thus the error is bounded by $e^{\log 2}|\gamma| \leq 2 \cdot 3/2\ 2^{-N} = 3 \cdot 2^{-N}$.

Now in the pseudo-multiplication, roundoff can occur only in the calculation of the pseudo-multiplicand, x_i. The error propogated to the pseudo-product z_{N+1}, then is $2^{N-1}q_o\epsilon_o + 2^{N}q_1\epsilon_1 + \ldots + 2q_N\epsilon_N$, where ϵ_i is the error in x_i. $\epsilon_o = 0$, and $|\epsilon_i| \leq 2^{-N}$, so the total error is bounded by $(2^{N} + 2^{N-1} + \ldots + 2)2^{-N} < 2$. Because the accumulation is in a double length register (the least significant part being N + 1 bits), we see that the actual error is no more than $2 \cdot 2^{-N-1} = 2^{-N}$.

Thus the total error in the procedure is bounded by $3 \cdot 2^{-N} + 2^{-N} = 2^{-N+2}$. Recall that a right shift is necessary for normalization which reduces the error to 2^{-N+1}.

We note that the above was accomplished without guard bits, and one double length register. The inclusion of one guard bit would reduce this error to 2^{-N}. However it would be necessary to include an additional quotient bit in the modified division to recode the exponent, since this was the largest source of error.

4.4.3 Normalization Methods

The basic algorithm has been described previously, the choice of a_i being the issue to be decided. Specker [61] and later Perle [46] use the criterion

$$a_i = \begin{cases} 1 + 2^{-i-1} & \text{if} \quad x_i - \log(1 + 2^{-i-1}) \geq 0 \\ 1 & \text{otherwise.} \end{cases}$$

Then $x_i \geq 0$ for all i, and it is easy to show that $x_i \leq 2^{-i}$, thus after N iterations $x_{N+1} \leq 2^{-N-1}$. Then we have $y_{N+1} e^{x_{N+1}} = y\, e^x$, and taking $y_{N+1} \approx e^x$ gives an error of $y_{N+1}(e^{x_{N+1}} - 1) \approx y_{N+1} \cdot x_{N+1} \leq e^x \cdot 2^{-N-1}$. Since $0 \leq x < \log 2$, the truncation error is seen to be bounded by 2^{-N}. Roundoff error is bounded by 2^{-N-1}, assuming a sufficient number of guard bits, and since the result must be right shifted to normalize, the total error is bounded by $2^{-N-1} + 2^{-N-2} < 2^{-N}$.

DeLugish [16] initializes by taking

$$x_1 = x - \log e_o$$

$$y_1 = e_o \quad ,$$

where

$$e_o = \begin{cases} e^{1/2} & \text{if} \quad x \geq 1/2 \;{}^* \\ e^{1/4} & \text{if} \quad 1/4 \leq x < 1/2 \\ 1 & \text{if} \quad x < 1/4 \end{cases}$$

Then $a_i = 1 + s_i 2^{-i-1}$ is chosen by

$$s_i = \begin{cases} -1 & \text{if} \quad x_i < -\,3/8 \cdot 2^{-i} \\ 1 & \text{if} \quad x_i \geq 3/8 \cdot 2^{-i} \\ 0 & \text{otherwise} \end{cases}$$

It is shown that $|x_i| \leq 3/8 \cdot 2^{-i+1}$, hence $|x_{N+1}| \leq 3/8 \cdot 2^{-N}$, so truncation error is slightly smaller than in the previous case. The number of non-zero

* DeLugish actually allows $|x| < \log 2$, and thus two more possible initilization values are given by him.

s_i is about N/3. One can again show that (for moderate i) that no more than two successive s_i can be non-zero, thus the maximum number of non-zero s_i is about 2N/3, decreasing the number of guard bits required. Total error is bounded by $3/4 \cdot 2^{-N-1} + 2^{-N-2} = 5/4 \cdot 2^{-N-1} < 2^{-N}$.

Chen [5] lets p_i be one plus the number of leading zeros in x_i, i.e., $x_i = 2^{-p_i} + v_i$, $0 \leq v_i < 2^{-p_i}$. Then $\log(a_i) = \log(1 + 2^{-p_i}) = 2^{-p_i} + O(2^{-2p_i})$, and the leading one bit in x_i is eliminated. At most N + 1 iterations will be required for $x_i < 2^{-N-1}$. The termination algorithm, to be applied when $p_n > N/2$ is $e^x \simeq y_n + x_n y_n$. Again, as with division, truncation error can be halved by taking $e^x \simeq y_n + y_n(x_n + 2^{-N-2})$. The total error bound is again less than 2^{-N}.

4.5 Power Function

The power function, x^y, has not received much attention in the literature. This function is usually evaluated by the identity $x^y = e^{y \log x}$, which requires evaluating both a logarithm and an exponential, with a multiplication necessary also.

A direct method was proposed by Krishnamurthy [28]. He assumed the numbers were in radix r, but we specialize this to r = 2. We first write $y = I + f$, where I is an integer and $0 \leq f < 1$. x^I can be evaluated by multiplications and a division, possibly. Let us write $f = \sum_{i=1}^{n} q_i 2^{-i}$. Then

$$x^f = x^{\sum_{i=1}^{n} q_i 2^{-1}} = \prod_{i=1}^{n} (x^{2^{-i}})^{q_i}$$

If we let $z_1 = x^{1/2}$, $z_{i+1} = z_i^{1/2}$, for i = 1,2,...,N - 1, we have $x^f = \prod_{i=1}^{N} z_i^{q_i}$. We can determine x^f with the following loop.

Let $p_o = 1$, $z_o = x$, then

for $i = 1,2,\ldots,N$

$$z_i = z_{i-1}^{1/2} ,$$

$$p_i = p_{i-1} \cdot z_i^{q_i} = \begin{cases} p_{i-1} & \text{if} \quad q_i = 0 \\ p_{i-1} z_i & \text{if} \quad q_i = 1 \end{cases}$$

Then $x^f \simeq p_N$.

This method would seem to be a rather time consuming procedure, since N square roots and up to N multiplications are necessary for only the fractional part of the exponent. It seems unlikely that it would compare favorably with the conventional method, unless a square root operation were available, and a logarithm/exponential routine was not.

Roundoff error would be able to come in both during the square rooting operation and during formation of the product, and would thus require more guard bits than usual.

4.6 Logarithm

Since the logarithm is the inverse of the exponential, it seems reasonable that the inverse of methods used for the exponential could be used for the algorithm. This is essentially true. Most routines are for any base logarithm, depending on the stored constants, but one due to Philo [48] and Dean [15] is for base 2 logarithms.

We will consider the logarithm of X, where X is any positive number. We then write $X = 2^\alpha \cdot x$, where $1/2 \leq x < 1$, and then $\log X = \alpha \log 2 + \log x$. It is then necessary to discuss the computation of $\log x$.

4.6.1 CORDIC Algorithm

Here we will make use of the identity $\log w = 2 \tanh^{-1} \frac{w-1}{w+1}$. If $1/2 \leq w < 1$, then $-1/3 \leq \frac{w-1}{w+1} < 0$, a region for which the CORDIC algorithm converges. We enter the algorithm with $m = -1$ in the vectoring mode, with $z = 0$, $x = w + 1$ and $y = w - 1$. The value of $z_{N_{-1}+1}$ must then be left shifted to obtain the result.

Truncation error is found by consideration of equation (6), by a procedure similar to that for the inverse tangent, and we obtain the bound $\delta_{N_{-1}}$. Roundoff error is of similar size, however, these bounds are for the hyperbolic tangent, which must be left shifted to obtain the logarithm thus the error bound is 2^{-N+1}.

4.6.2 Pseudo-Division/Mulitplication Methods

This is a pseudo-division process, and can be carried out in one pass, as is seen from Table 3. Normally, $\log z$ is required, and this can be obtained by setting $y = 1 - z$, $x = z$. This gives $\log(1 + y/x) = \log(1 + \frac{1-z}{z}) = -\log z$. This particular transformation keeps x and y small and non-negative, as is desirable and necessary.

Using no guard bits, we find that the error in z_{N+1} is $-2^{N+1}q_o\epsilon_o - 2^N q_1\epsilon_1 - \ldots - 2q_N\epsilon_N$. Since $\epsilon_o = 0$, and $|\epsilon_i| \leq 2^{-N}$, we have the error bounded by $(2^N + 2^{N-1} + .. + 2)2^{-N} = 2$. The pseudo-remainder is $z_{N+1}/2^{N+1}$, hence the error is bounded by 2^{-N}. Now the accumulation of the psuedo-dividend requires the use of guard bits. If it is done in the double length register, as suggested by Meggitt, roundoff error will be insignificant, and the total error will be less than 2^{-N}.

4.6.3 Normalization Methods

Specker [61], Perle [46], DeLugish [16], and Chen [5] have suggested these algorithms, and the selection of a_i is the same as for division, since in each case it is desired to force x_i to 1. The error bounds are about the same. If $|1 - x_n| \leq 2^{-N-1}$, then the error is $\log|1 - x_n| \approx 2^{-N-1}$, as before. Roundoff error again can accumulate up to 2^{-N-1} for a total error of 2^{-N}.

Chen's termination algorithm is $y + \log x \approx y_n - (1 - x_n)$, or to reduce the truncation error, $y + \log x \approx y_n - (1 - x_n + 2^{-N-2})$. Again n is taken so that $p_n > N/2$. Comments pertaining to errors, made in Section 4.1.3, apply here also.

4.6.4 Other Methods

Philo and Dean suggested the following method for computing $\log_2 x$. It is the inverse of the procedure described in Section 4.5 for computing x^y. It was discussed by Dean in terms of implementation using cellular arrays.

We want to compute digits q_i such that $x = 2^{\sum_{i=1}^{N} q_i 2^{-i}}$. We assume that $1 \leq x < 2$ so that $0 \leq \log_2 x < 1$. Then $x = 2^{1/2 \cdot q_1} \cdot 2^{1/4 \cdot q_2} \ldots \cdot$ $\cdot\ 2^{1/2^n \cdot q_n}$. If $x^2 \geq 2$, then it is apparent that $q_1 = 1$. If $x^2 < 2$, then $q_1 = 0$. The following recursion can be seen to generate the q_i.

$$x_o = x, \quad \text{for } i = 1,2,\ldots, N,$$

$$q_i = \begin{cases} 1 & \text{if} \quad {x_{i-1}}^2 \geq 2 \\ 0 & \text{if} \quad {x_{i-1}}^2 < 2 \end{cases}$$

and

$$x_i = \begin{cases} 1/2 \cdot x_{i-1}^2 & \text{if} \quad q_i = 1 \\ x_{i-1}^2 & \text{if} \quad q_i = 0 \end{cases}$$

The truncation error of the above procedure is 2^{-N} if enough guard bits are carried in the calculation of the x_i. The error will generally more than double at each iteration, although a possible right shift will tend to keep it about the same magnitude. For example, if the error at the $i-1^{th}$ stage is ϵ_{i-1}, then

$$\epsilon_i = \begin{cases} 1/2 \cdot (2x_{i-1}\epsilon_{i-1} + \epsilon_{i-1}^2) & \text{if} \quad q_i = 1 \\ 2x_{i-1}\epsilon_{i-1} + \epsilon_{i-1}^2 & \text{if} \quad q_i = 0 \ . \end{cases}$$

Considering that $q_i = 1$ or 0 as $x_{i-1} \geq 2^{1/2}$ or $x_{i-1} < 2^{1/2}$, we have

$$\epsilon_i \leq \begin{cases} 1/2 \cdot (2 \cdot 2 \cdot \epsilon_{i-1} + \epsilon_{i-1}^2) = 2\epsilon_{i-1} + 1/2\epsilon_{i-1}^2 \approx 2\epsilon_{i-1} \\ 2 \cdot 2^{1/2}\epsilon_{i-1} + \epsilon_{i-1}^2 \quad = 2^{3/2}\epsilon_{i-1} + \epsilon_{i-1}^2 \approx 2^{3/2}\epsilon_{i-1} \end{cases}$$

This does not include the error due to chopping after the squaring operation, which would be bounded by 2^{-N-J}, using $J < N$ guard bits. ϵ_o is zero, so we have $|\epsilon_1| \leq 2^{3/2} \cdot 0 + \epsilon_{R_1} = 2^{-N-J}$. Then $|\epsilon_2| \leq 2^{3/2}(2^{-N-J}) + 2^{-N-J} = (2^{3/2} + 1)2^{-N-J}$. Continuing in this fashion, or by induction, one obtains

$$|\epsilon_N| \leq [(2^{3/2})^{N-1} + (2^{3/2})^{N-2} + \ldots + 1]2^{-N-J}, \qquad \text{or}$$

$|\epsilon_N| \leq (2^{3/2})^N \cdot 2^{-N-J} = 2^{N/2 - J}$. Thus the use of $N/2$ guard bits will prevent buildup of error larger than 2^{-N}.

Because repeated squaring is required, this method is probably not too attractive, even though it is a simple calculation.

Nicoud and Dessoulavy [45] suggested a method which requires repeated multiplication. Suppose log x is desired, where $1 \le x < 2$. Now, for an appropriate number $u = 1 + 2^{-k}$, we consider the sequence $1, u, \ldots, u^n$, until $u^n \ge x$. Then we have $\log x \approx \log u^n = n \log u$. The truncation error is

$$\begin{aligned} |\log u^n - \log x| &\le \log u^n - \log u^{n-1} \\ &= \log u \approx 2^{-k}, \end{aligned}$$

hence we would probably take $k = N + 1$. The error introduced by forming the sequence $1, u, \ldots, u^n$ is bounded by $n\ 2^{-N-J}$. Unfortunately, for $x \approx 2$, we would require

$$n \approx \frac{\log 2}{\log(1+2^{-N-1})} \approx 3 \cdot 2^{N-1} \quad ,$$

thus the number of operations required for high precision is excessive. About N + 1 guard bits would be required.

4.7 Tangent

Several authors have discussed generation of the tangent function, including Meggitt [40] and DeLugish [16]. Of course, the tangent can always be generated as the quotient of the sine and cosine, say from the CORDIC algorithm. Both Meggitt and DeLugish use this idea, with modifications.

We assume that the angle is reduced as in the sine/cosine routines. Thus we have a first quadrant angle. This is further reduced by the identity $\tan z = \cot(\pi/2 - z)$ if $z > \pi/4$.

Meggitt's initial modified division is equivalent to finding which rotations need to be done to drive the angle to zero, and doing only those which do not "over-shoot" zero. The arbitrary value, Q, should be approximately of full register length, near one. The accumulation in the double length register renders roundoff error to be insignificant. Error in recoding the angle is about the same as was discussed in the case of the exponential function, and one guard bit and N + 1 iterations (n = N + 2) are required to bring the truncation error bound down to the desired level.

DeLugish use the same initilization as for sine/cosine, and the same decision process as is used in the latter stages of that procedure. Of course, the radial distortion correction is not necessary for any of the iterations.

The errors in calculation of the sine and cosine were found to be bounded by 2^{-N}, and these bounds hold for the modified procedures here. Then

$$\frac{K \sin z - \epsilon_1}{K \cos z - \epsilon_2} \approx (\tan z - \epsilon_1/K)(1 + \frac{\epsilon_2}{K \cos z}$$

$$\approx \tan z - \epsilon_1/K + \frac{\epsilon_2 \tan z}{\cos z} .$$

The approximate error is then bounded by $|\epsilon_1| + \frac{|\epsilon_2|}{\cos z} \leq (1 + \sqrt{2})2^{-N}$. However, separate consideration of the truncation and roundoff errors in $K \sin z$ and $K \cos z$ will yield a better bound, about $(1 + 1/\sqrt{2})2^{-N}$, an error confined to the last two bits of the result. For angles greater than $\pi/4$, the reciprocal is taken, and the error becomes arbitrarily large. This is natural, however, since the value of the tangent also becomes arbitrarily large.

4.8 Square Root

Computation of square roots has received a great deal of attention. Lenaerts [30] was one of the first to discuss the possibility of a build-in square root operation; his suggestion was for a particular type of existing computer, and was an adaption of the usual hand method for extracting square roots. Cowgill [8] gave a non-restoring version of the usual hand method. Kostopoulos [26] and others have also suggested adaptions of the basic hand calculation. Metze [42] gave an algorithm which generates the square root in a minimally represented redundant form. The pseudo-multiplication of Meggitt [40] and Sarkar and Kirshnamurthy [55] will calculate square roots, as will the CORDIC algorithm. Chen [5] and DeLugish [16] give normalization techniques for the calculation. DeLugish gives two algorithms for the square root. One is an additive normalization procedure which is actually a modified hand method. We will discuss iterative techniques briefly.

We assume that $X = 2^{2\alpha} . x$, where $1/4 \le x < 1$, then $X^{1/2} = 2^{\alpha} . \sqrt{x}$, and $1/2 \le \sqrt{x} < 1$. We concern ourselves with the computation of $\sqrt{x}$.

4.8.1 CORDIC Algorithm

Suppose $\sqrt{w}$, $1/4 \le w < 1$ is to be computed. Then, setting $x = w + 1/4$, $y = w - 1/4$, and entering the algorithm with $m = -1$ in the vectoring mode will yield the value $K_{-1}\sqrt{x^2-y^2} = K_{-1} \sqrt{w}$. Unfortunately, a multiplication by K_{-1}^{-1} is then necessary to obtain $\sqrt{w}$, taking an additional multiplication and increasing roundoff error. An initial formation of wK_{-1}^{-2} could be done, also, but would require that K_{-1}^{-2} be stored, as well as K_{-1}^{-1}.

The truncation error for the calculation of $K_{-1} \sqrt{w}$ is a lengthy calculation, and is approximately $-\dfrac{K_{-1} (w^2-w+3/16)}{(w+1/4) \cosh \alpha} \delta_{N_{-1}}$, where $\alpha \simeq \tanh^{-1} \dfrac{w - 1/4}{w + 1/4}$.
Hence the truncation error is bounded by $K_{-1} \cdot 2^{-N-2}$. Roundoff errors are the same as for other applications of the CORDIC algorithm, so the error in $K_{-1} \sqrt{w}$ is bounded by $2^{-N-1} + K_{-1} 2^{-N-2}$. The division (or multiplication) would introduce additional error.

4.8.2 "Hand" Methods

The usual hand method for extracting square roots is a matter of trial and error. Let us outline and review the procedure for radix two. Suppose we have a number x, $1/4 \le x < 1$, and we have computed digits $q_1, q_2, \ldots, q_{i-1}$ so that $R_{i-1} = \sum_{j=1}^{i-1} q_j 2^{-j}$ is as large as possible, and still satisfies $x - R_{i-1}^2 \ge 0$. Then we take q_i to be zero or one, depending on whether $x - (R_{i-1} + 2^{-i})^2$ is negative or non-negative. If we let $E_i = x - R_i^2$, then

$$E_i = x - (R_{i-1} + q_i 2^{i})^2 = E_{i-1} - q_i 2^{-i+1}(R_{i-1} + q_i 2^{-i-1}).$$

Thus, we see that a trial value of E_i can be computed by subtracting $2^{-i+1}(R_{i-1} + 2^{-i-1})$ from E_{i-1}. If the result is negative, $q_i = 0$ and $E_i = E_{i-1}$. If the result is non-negative, $q_i = 1$, and E_i has been computed.

For machine implementation it is more convenient to deal with $M_i = 2^i E_i$, rather than E_i. The calculations can then be arranged as follows. Since $1/4 \le x < 1$, $q_i = 1$; thus we can initialize

$$M_1 = 2(x - 1/4)$$

$$R_1 = 1/2$$

For $i = 2, 3, \ldots, N$

$$T_i = 2M_{i-1} - (2R_{i-1} + 2^{-i}).$$

Then, if $T_i \ge 0$

$$q_i = 1$$

$$M_i = T_i$$

$$R_i = R_{i-1} + 2^{-i}\text{ , or}$$

if $T_i < 0$

$$q_i = 0$$

$$M_i = 2M_{i-1}$$

$$R_i = R_{i-1}$$

This is seen to be a restoring type of process, since a trial value is computed, and then may be discarded.

It is easily seen that the truncation error is 2^{-N}, since we have computed the first N bits of the square root, and the remainder, $\sum_{i=N+1}^{\infty} q_i 2^{-i} \le 2^{-N}$. As we have outlined the procedure, no roundoff error occurs since no numbers are right shifted off the register, and all other numbers are represented exactly with N fraction bits. We should note that M_i can be large enough to require some bits to the left of the binary point, however. This might be best handled by scaling and inclusion of an appropriate number of guard bits.

The above procedure can easily be converted to a non-restoring square root routine. The version we give is not generalized the same way as the non-restoring division routine, however.

Let $M_1 = 2(x-1/4)$

$R_1 = 1/2$

For $i = 2,3,\ldots,N$

$$q_i = \begin{cases} 1 & \text{if} \quad M_{i-1} \ge 0 \\ -1 & \text{if} \quad M_{i-1} < 0 \end{cases},$$

then $R_i = R_{i-1} + q_i 2^{-1}$

$$M_i = 2M_{i-1} - q_i(2R_{i-1} + q_i 2^{-i}).$$

The error bounds are the same as for the restoring square root procedure.

Metze has given a non-restoring algorithm which yields the value in a minimally represented redundant form. As with the Metze division algorithm [41] there is overlap in the regions where a digit may be chosen as zero or non-zero. We outline the algorithm as given, and then discuss the overlap and a possible compromise. Let

$$R_{-1} = x$$

$$S_{-1} = 0$$

Here R_i will correspond to the partial remainder $x-S_i^2$, where S_i is the approximation to the square root after i steps.

For $i = 0,1,\ldots,N$

$$K_i^{\pm 0} = 2^{-i}(\pm 5/3 \cdot S_{i-1} + 25/36 \cdot 2^{-i})$$

$$K_i^{\pm 1} = 2^{-i}(\pm 4/3 \cdot S_{i-1} + 4/9 \cdot 2^{-i})$$

Determine q_i by

$$q_i = \begin{cases} 0 & \text{if } K_i^{-0} < R_{i-1} < K_i^{+0} \\ 1 & \text{if } K_i^{+1} < R_{i-1} \\ -1 & \text{if } K_i^{-1} > R_{i-1} \end{cases}$$

Then

$$R_i = R_{i-1} - q_i 2^{-1}(2S_{i-1} + q_i 2^{-1})$$

$$S_i = S_{i-1} + q_i 2^{-i}$$

Since $K_i^{-0} < K_i^{-1} < K_i^{+1} < K_i^{+0}$, it is not well defined in the above how q_i is to be chosen if R_{i-1} is in one of the overlapping regions. It can be chosen either way one prefers; the representation will still be minimal. Since the above values of $K_i^{\pm 0}$ and $K_i^{\pm 1}$ are not particularly easy to compute, a reasonable suggestion would seem to be to compromise at some point in the regions which is easier to compute. Such a value is $K_i^{\pm} = 2^{-i}(\pm 3/2 \cdot S_{i-1} + 2^{-i-1})$. Actually, in the above there appears to be a slight difficulty with the iteration i = 0 since then $K_o^{\pm 0} = 25/36$, and $K_o^{\pm 1} = 4/9$. However, since $R_{-1} > 0$, the negative values are not required. The same difficulty appears in our suggestion, and is resolved the same way.

With our suggested criterion for determining the q_i and the replacement of R_i by $2^i R_i$, an algorithm for computing minimally represented square roots is as follows. We remove the apparent difficulty on the first iteration by initializing differently.

Let

$$S_o = \begin{cases} 1 & \text{if} \quad x \geq 1/2 \\ 0 & \text{if} \quad x < 1/2 \end{cases}$$

and $\quad M_o = x - S_o^2 = x - S_o$

For i = 1,2,...,N

$$K_i^{\pm} = \pm\, 3/2 \cdot S_{i-1} + 2^{-i-1},$$

then determine q_i by

$$q_i = \begin{cases} 1 & \text{if} \quad K_i^{+} < 2M_{i-1} \\ -1 & \text{if} \quad K_i^{-} > 2M_{i-1} \\ 0 & \text{otherwise} \quad . \end{cases}$$

Then $\quad M_i = 2M_{i-1} - q_i(2S_{i-1} + q_i 2^{-i})$

$$S_i = S_{i-1} + q_i 2^{-i}$$

The same comments pertaining to roundoff error made for previous methods applies here. Metze shows that the remainder R_N is bounded by $K_N^{\pm 0}$, thus we have

$$2^{-N}(-5/3 \cdot S_{N-1} + 25/36 \cdot 2^{-N}) < x - S_N^2 < 2^{-N}(+5/3 \cdot S_{N-1} + 25/36 \cdot 2^{-N}).$$

Now, $S_{N-1} \approx S_N$, and $2^{-N} << S_N$, so $x - S_N^2$ is bounded by about $2^{-N} \cdot 5/3 \cdot S_N < 2^{-N+1} S_N$. The latter can be shown to be a rigorous bound for moderate values of N. We are interested in $\sqrt{x} - S_N$, not $x - S_N^2$, but recalling that $S_N \cong \sqrt{x}$, we have

$$|\sqrt{x} - S_N| = \left|\frac{x - S_N^2}{\sqrt{x} + S_N}\right| \approx \frac{|x - S_N^2|}{2S_N} \leq \frac{2^{-N+1} S_N}{2S_N} = 2^{-N} .$$

Thus the error is no greater than one in the last bit.

DeLugish gives a similar procedure where the comparison constant is not a function of the partial result. Let $U_o = x_o/4$, then the initialization step is

$$U_1 = 1/2(x_o - r_o^2)$$

$$R_1 = r_o$$

where r_o is given in Table 8. The comparison constant c^1 is also given in the table.

interval for x	r_o	c^1
[1/2, 5/16)	1/2	3/8
[5/16, 3/8)	9/16	7/16
[3/8, 9/16)	5/8	1/2
[9/16, 7/8)	3/4	5/8
[7/8, 1)	1	3/4

Table 8

For $i = 1,2,\ldots,N-1$,

$$s_i = \begin{cases} -1 & \text{if } U_i < -\frac{c^1}{4} \\ 1 & \text{if } U_i \geq \frac{c^1}{4} \\ 0 & \text{otherwise} \end{cases}$$

and $U_{i+1} = 2U_i - 1/2 \cdot s_i(R_i + s_i\, 2^{-i-2})$ $\quad R_{i+1} = R_i + s_i 2^{-i-1}$.

DeLugish shows that $|\sqrt{x} - R_i| \leq 2^{-i}$, hence $|\sqrt{x} - R_N| \leq 2^{-N}$, so truncation error is bounded by 2^{-N}. Because of the way DeLugish initializes, 2 guard bits are required to avoid error in the initial calculation of U_o and in the latter stages of the U_i sequence. This is the same problem referenced earlier when discussing the magnitude of the partial remainders and a possible scaling with guard bits to contain the right shift.

4.8.3 Pseudo-Division/Multiplication Methods

Meggitt's method can be used to compute square roots, as Table 3 shows. The only complication is that the recursion for x_{i+1} has three terms, but this is common to most methods. Roundoff error will occur in the calculation of x_{i+1}, but as before, the effect is less than 1/2 in the Nth digit, or less than 2^{-N-1}, with no guard bits required.

4.8.4 Normalization Methods

Normalization methods are for the calculation of $y/\sqrt{x}$, and $\sqrt{x}$ is obtained by setting $y = x$. The procedures are similar to those for division, except that x_i is multiplied by a_i^2 , requiring a three term sum, as in the hand methods. We assume $1/4 \leq x < 1$.

DeLugish initializes by $x_o = x_1$, $R_o = y$, then

$$U_1 = r_o x_o - 1$$

$$R_1 = R_o \sqrt{r_o} \quad , \quad \text{where}$$

$$r_o = \begin{cases} 4 & \text{if} \quad 1/4 \le x_o < 1/2 \\ 1 & \text{if} \quad 1/2 \le x_o < 1 \end{cases}$$

For $i = 1,2,\ldots,N$

$$s_i = \begin{cases} -1 & \text{if} \quad U_i \ge 3/8 \\ 1 & \text{if} \quad U_i < 3/8 \\ 0 & \text{otherwise} \end{cases} \quad ,$$

and then

$$U_{i+1} = 2U_i + s_i + 2^{-i}(2U_i s_i + 1/4 \cdot s_i^2) + U_i s_i^2 \cdot 2^{-2i-1}$$

$$R_{i+1} = R_i \; (1 + s_i 2^{-i-1})$$

In the above, we note that $a_i = 1 + s_i 2^{-i-1}$ and $U_{i+1} = 2^i(x_{i+1} - 1)$, where $x_{i+1} = x_i(1 + s_i 2^{-i-1})^2$.

DeLugish gives the error bound $|x_{N+1} - 1| \le 3/4 \cdot 2^{-N}$, and this gives a truncation error bound of about $3/8 \cdot 2^{-N} < 2^{-N-1}$. Roundoff error bounds are the same as most other DeLugish algorithms. The average number of non-zero s_i is about $N/3$. It is not known what the maximum number of non-zero s_i is, but in any case $\log_2 N + 1$ guard bits are sufficient.

The Chen algorithm again counts left zeros in $1 - x_i = 2^{-p_i+1} + V_i$, $0 \le V_i < 2^{-p_i+1}$, and p_i is two plus the number of left zeros in $1 - x_i$. Then $\alpha_i = 1 + 2^{-p_i}$, and $1 - x_{i+1} = 1 - \alpha_i x_i = 1 - (1 + 2^{-p_i})^2(1 - 2^{-p_i+1} + V_i)$ $= V_i + 0(2^{-2p_i + 2})$, thus, again the leading one-bit is eliminated.

The termination algorithm, again applied when $p_n > \frac{N}{2}$, is $y/\sqrt{x} \approx y_n + 1/2 \cdot y_n(1 - x_n)$. For purposes of reducing truncation error, the termination algorithm $y/\sqrt{x} \simeq y_n + 1/2\ y_n(1 - x_n + 2^{-N-2})$ may be used. Truncation error is then bounded by 2^{-N-2}. The same comments about roundoff error made in other discussions of Chen's normalization algorithms apply here.

4.8.5 Iterative Procedures

The classical iteration for the solution of $Z^2 - x = 0$ is the Newton iteration, $Z_{i+1} = 1/2 \cdot (Z_i - x/Z_i)$, with Z_o given as an initial guess. Convergence is quadratic. Many papers have been written concerning optimal starting values and other aspects of the iteration. As written above, a division is required.

Ramamoorthy, Goodman, and Kim [49] have made a study of iterative methods which use only multiplication, except for a possible final division. The iteration

$$R_{i+1} = \frac{R_i}{2}(3 - xR_i^2)$$

converges to $x^{-1/2}$ for appropriate R_o.

Another iteration, involving two variables is

$$R_{i+1} = R_i(2 - B_iR_i)$$

$$B_{i+1} = 1/2(B_i + xR_{i+1}) \quad .$$

Here $\{R_i\}$ converges to $x^{-1/2}$ and $\{B_i\}$ to $x^{1/2}$.

It is interesting to note that if $B_i = N$ for all i in the first equation, that this is the iteration for the reciprocal of N, thus if this type of division were employed, the square root could be implemented easily.

4.9 Product

Unlike many functions, the product is easily implemented in a straightforward fashion. A modification of the usual successive shift and add method, additive normalization, is suggested by DeLugish [16]. Ling [32] has given a method based on logical circuits to form the product of 8 bit segments of numbers. Chen [4] gives another, somewhat similar algorithm, based on $AB = (\frac{A+B}{2})^2 - (\frac{A-B}{2})^2$. Logan [36] has a similar proposal using "squaring" chips. And of course, the CORDIC algorithm will yield a product.

In general we assume that we wish to compute yx, where $1/2 \le x < 1$ and y may be any number.

4.9.1 CORDIC Algorithm

Entering the CORDIC algorithm with $m = 0$, $y = 0$, in the rotation mode, yields $y_{N_o+1} \approx xz$, where in this instance $|z| \le 1$. We have $y_{N_o+1} = x(-\sum_{i=0}^{N_o} \alpha_i)$, and $|z + \sum_{i=0}^{N_o} \alpha_i| = |z_{N_o+1}| \le \alpha_{N_o} = 2^{-N-1}$.

Thus, the truncation error is no more than $2^{-N-1}|x|$, and if $|x| \le 1$, the bound is 2^{-N-1}. The roundoff error is also bounded by 2^{-N-1} for a total error of no more than 2^{-N}, or one in the last bit.

4.9.2 "Hand" Methods

The usual shift and add method was outlined in Section 3.2. There is no truncation error, and N guard bits are required for no roundoff error (i.e., a double length register), but $\log_2 N + 1$ guard bits renders roundoff error less than 2^{-N-1}, with chopping.

DeLugish uses the following additive normalization. Let $U_o = x$, $P_o = 0$.

Initialize $$U_1 = 2(U_o - m_o)$$

$$P_1 = P_o + ym_o \quad ,$$

where

$$m_o = \begin{cases} 1/2 & \text{if } 1/2 \le x < 3/4 \\ 1 & \text{if } 3/4 \le x < 1 \end{cases}$$

For $i = 1,2,\ldots,N$

$$s_i = \begin{cases} -1 & \text{if } U_i < -3/8 \\ 1 & \text{if } U_i \geq 3/8 \\ 0 & \text{otherwise} \end{cases},$$

then $$U_{i+1} = 2U_i - s_i$$

$$P_{i+1} = P_i + ys_i 2^{-i-1} .$$

Letting $m_i = s_i 2^{-i-1}$ for $i \geq 1$, we have $U_i = 2^i(x - \sum_{j=0}^{i-1} m_j)$, and

$$P_i = y \sum_{j=0}^{i-1} m_j .$$

Thus the truncation error is

$$yx - P_{N+1} = y(x - \sum_{j=0}^{N} m_j).$$

DeLugish shows that $|x - \sum_{j=0}^{N} m_j| \leq 3/8 \cdot 2^{-N}$,

thus the truncation error is bounded by $|y| . 3/8 \cdot 2^{-N} < 2^{-N-1}$ if $|y| \leq 1$. Roundoff error again depends on the number of non-zero s_i, which average about $N/3$ again. By an argument similar to that for division, it can be shown that no more than two successive s_i can be non-zero, thus at most about $2N/3$ of the s_i can be non-zero. Hence, about $\log_2 2N/3 + 1$ guard bits are sufficient to bound roundoff error by 2^{-N-1}. Total error is no more than one bit.

4.9.3 Other Methods

Several methods have been derived for obtaining the product by decomposing it into a sum of several other terms which are determined by means of logical circuits. Ling gave a method whereby the product of two 8 bit numbers is obtained in three additions. For longer word lengths, the idea is applied to segments of the numbers in parallel, and then the product found by summing. Time is about 1 + 2k addition times, where the word length is $8 \, . \, 2^{k-1}$, $k \geq 1$.

Basically, the idea is to form

$$\frac{x+y}{2} = I = 2^n i \quad , \quad 1/2 \leq i < 1$$

$$\frac{x-y}{2} = J = 2^n j \quad , \quad 1/2 \leq j < 1 \quad .$$

Then, with $S(z) = z - \frac{z^2}{2}$, S(i) and S(j) are formed by means of logical circuits. Then it is easily verified that

$$xy = 2 \, \{2^n I - 2^m J - 2^{2n} S(i) + 2^{2m} S(j)\}.$$

As many as 26 terms involving seven variables occur in the definition of the bits of S(z).

Chen uses the idea that $xy = (\frac{x+y}{2})^2 - (\frac{x-y}{2})^2$, and a decomposition of the square of a number into the sum of three quantities (for 8 bits; two quantities for 4 or 6 bits), $Z^2 = R + S + T$.

The expressions for R, S, and T are decided upon by considering the "squaring parallelogram" that results when forming the square with pencil and paper. Symmetry can be exploited, and the decomposition given by Chen results in each bit involving no more than 14 terms and six variables.

Logan [36] has a similar idea using squaring chips [35], except that he forms $2xy = (x+y)^2 - x^2 - y^2$. This would appear to be more costly in terms of both time and hardware than using $xy = (\frac{x+y}{2})^2 - (\frac{x-y}{2})^2$.

The above procedures form a double length product, thus there is no error involved.

5.0 Conclusion

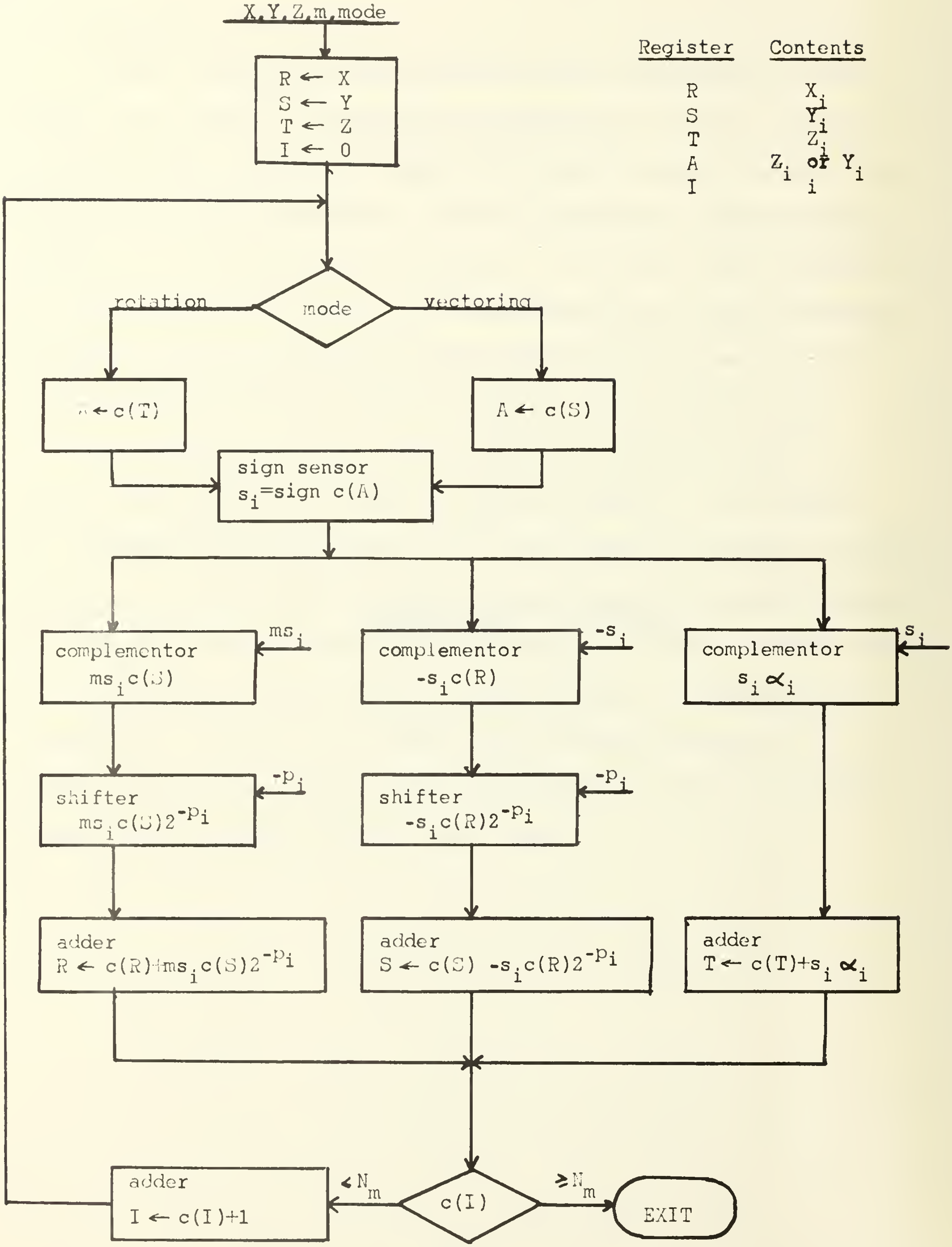
Several conclusions can be drawn from this study, depending on the point of view taken. If simplicity and generality is the main objective, then the CORDIC algorithm of Walther [66] should be heavily favored. If speed is of principal importance, then the normalization method proposed by Chen [5] is excellent for its purpose. We note that Chen's algorithm is the subject of a U. S. patent. For implementation with more conventional hardware, and a good compromise between simplicity and speed, the set of normalization algorithms proposed by DeLugish must be highly rated.

The special purpose algorithms usually (although not always) are faster than those which are part of a unified algorithm. The minimal representation algorithms by Metze [41], [42] should be excellent for their purpose. The decomposition schemes for performing multiplication, proposed by Ling [32], Chen [4], and Logan [36], appear to be capable of high speeds.

As was noted earlier, the only method available for computation of trigonometric functions is the CORDIC algorithm and variations of it. It appears that this is one area where some additional study is necessary to develop faster and more efficient algorithms.

Appendix A: Block Diagrams

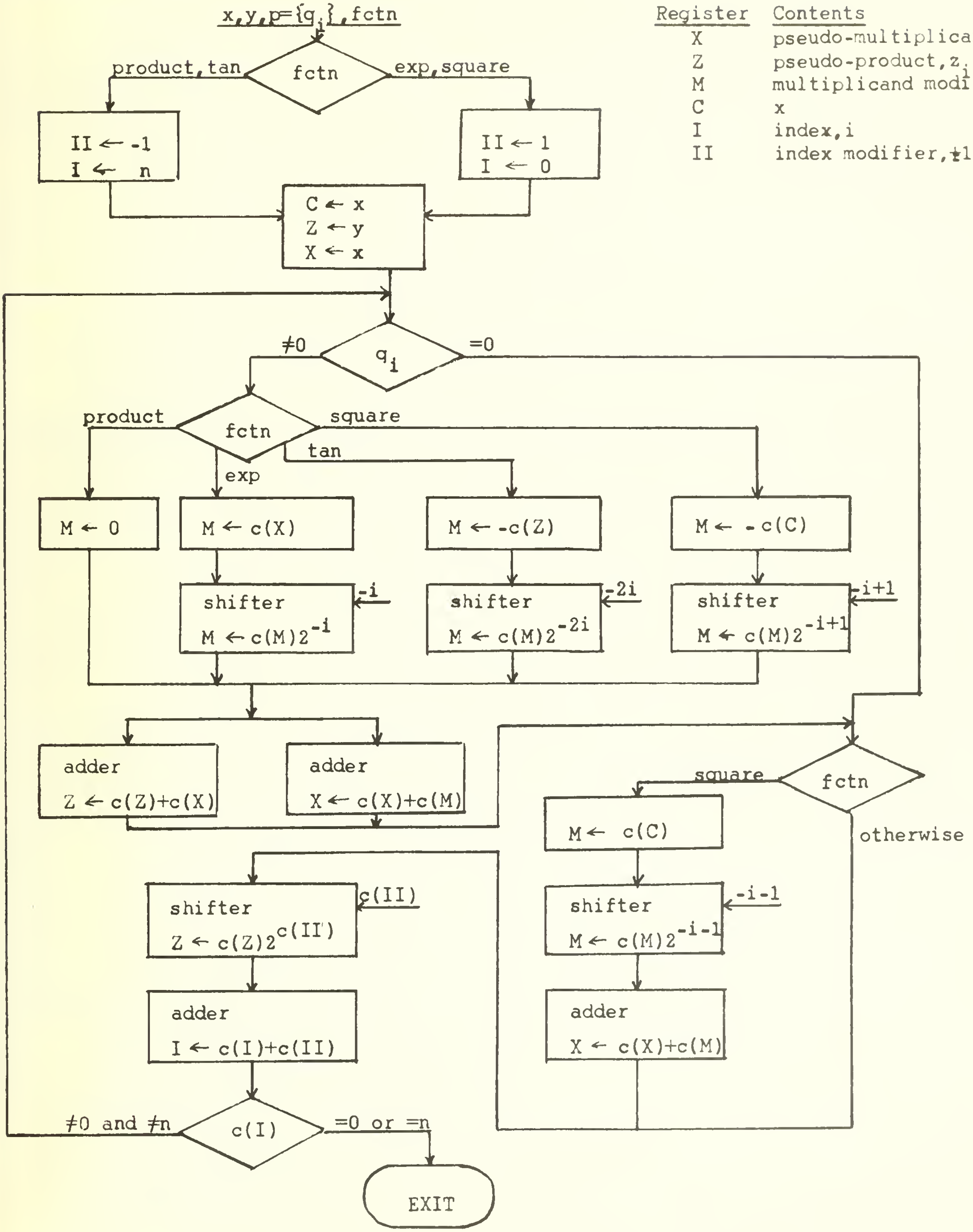
BLOCK DIAGRAM - CORDIC ALGORITHM



Register	Contents
R	X_i
S	Y_i
T	Z_i
A	Z_i or Y_i
I	i

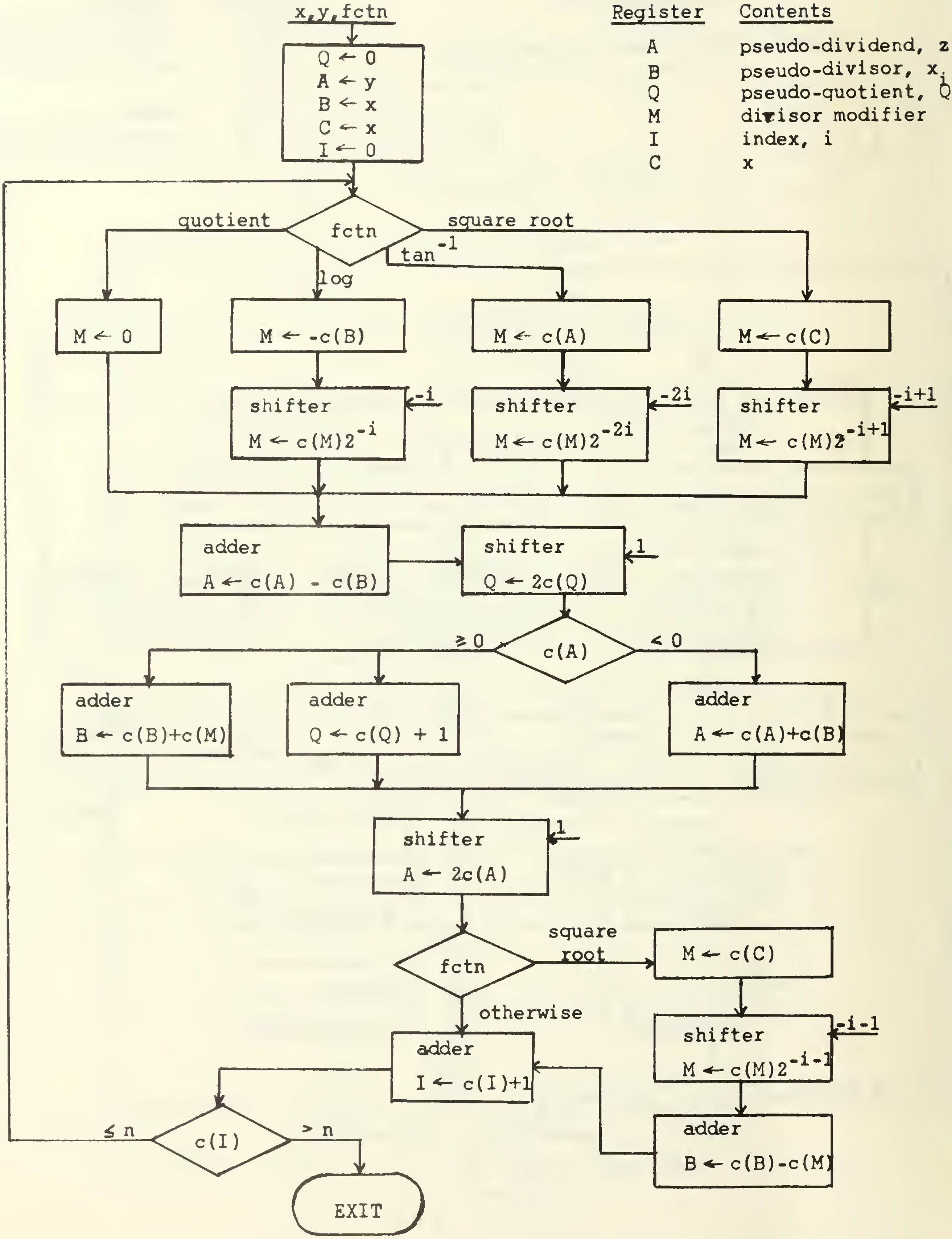
BLOCK DIAGRAM - PSEUDO-MULTIPLICATION

Register	Contents
X	pseudo-multiplicand, x_i
Z	pseudo-product, z_i
M	multiplicand modifier
C	x
I	index, i
II	index modifier, ±1

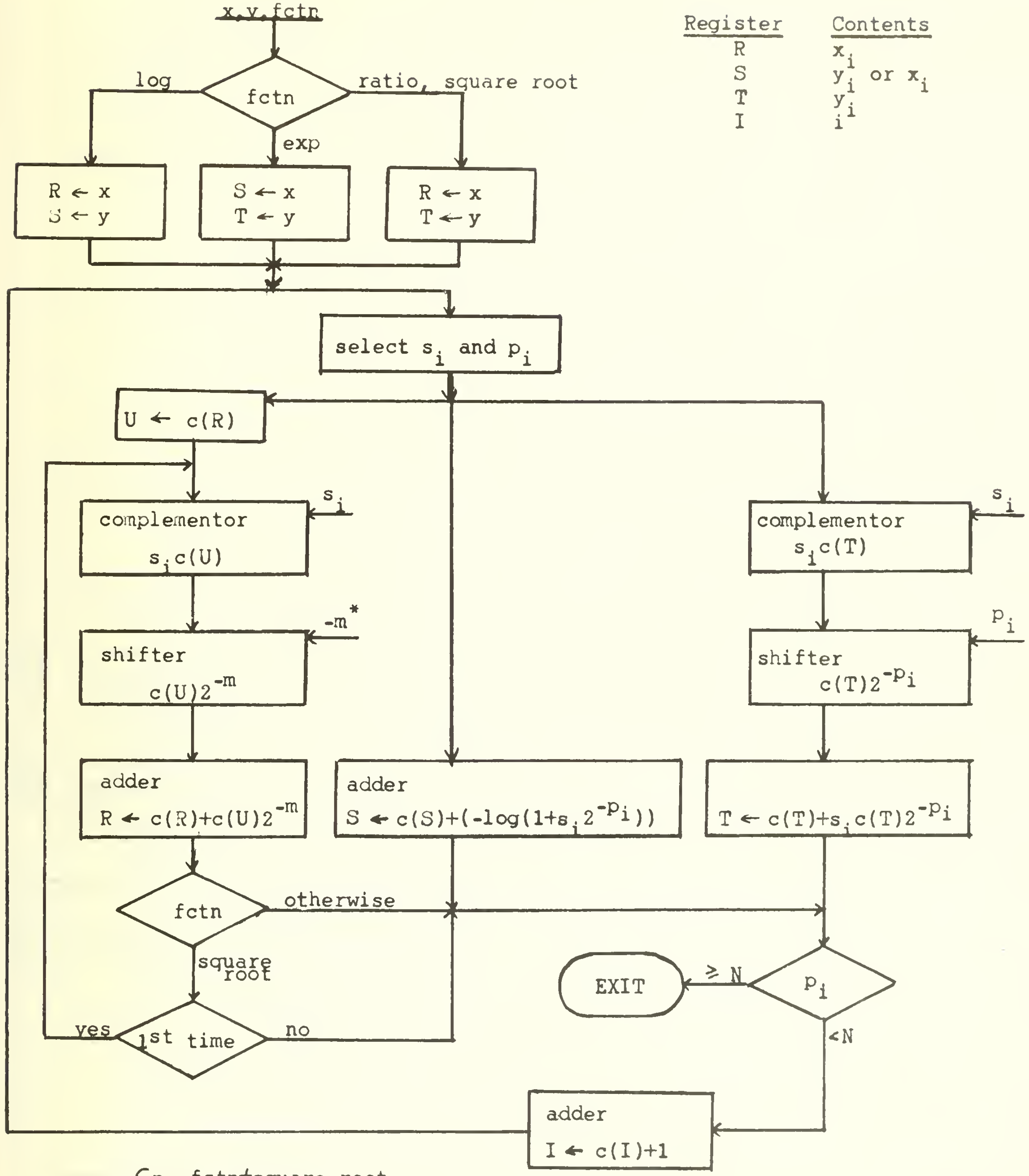


BLOCK DIAGRAM - PSEUDO-DIVISION

Register	Contents
A	pseudo-dividend, z_i
B	pseudo-divisor, x_i
Q	pseudo-quotient, Q_i
M	divisor modifier
I	index, i
C	x



BLOCK DIAGRAM - NORMALIZATION METHODS

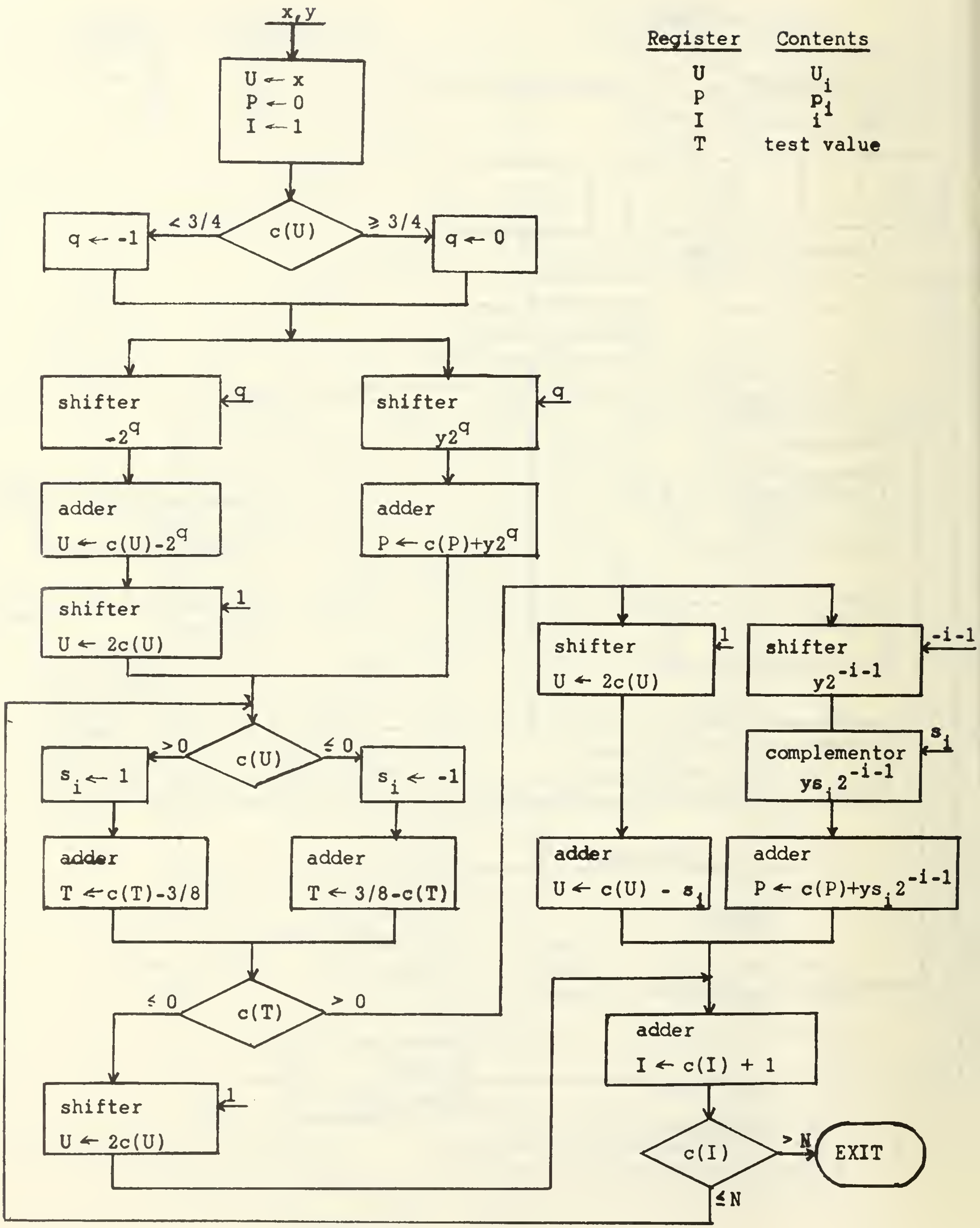


Register	Contents
R	x_i
S	y_i or x_i
T	y_i
I	i

$$* \quad m = \begin{cases} p_i, \text{fctn} \neq \text{square root} \\ p_i - 1, \text{fctn} = \text{square root, 1st time} \\ p_i + 1, \text{fctn} = \text{square root, 2nd time} \end{cases}$$

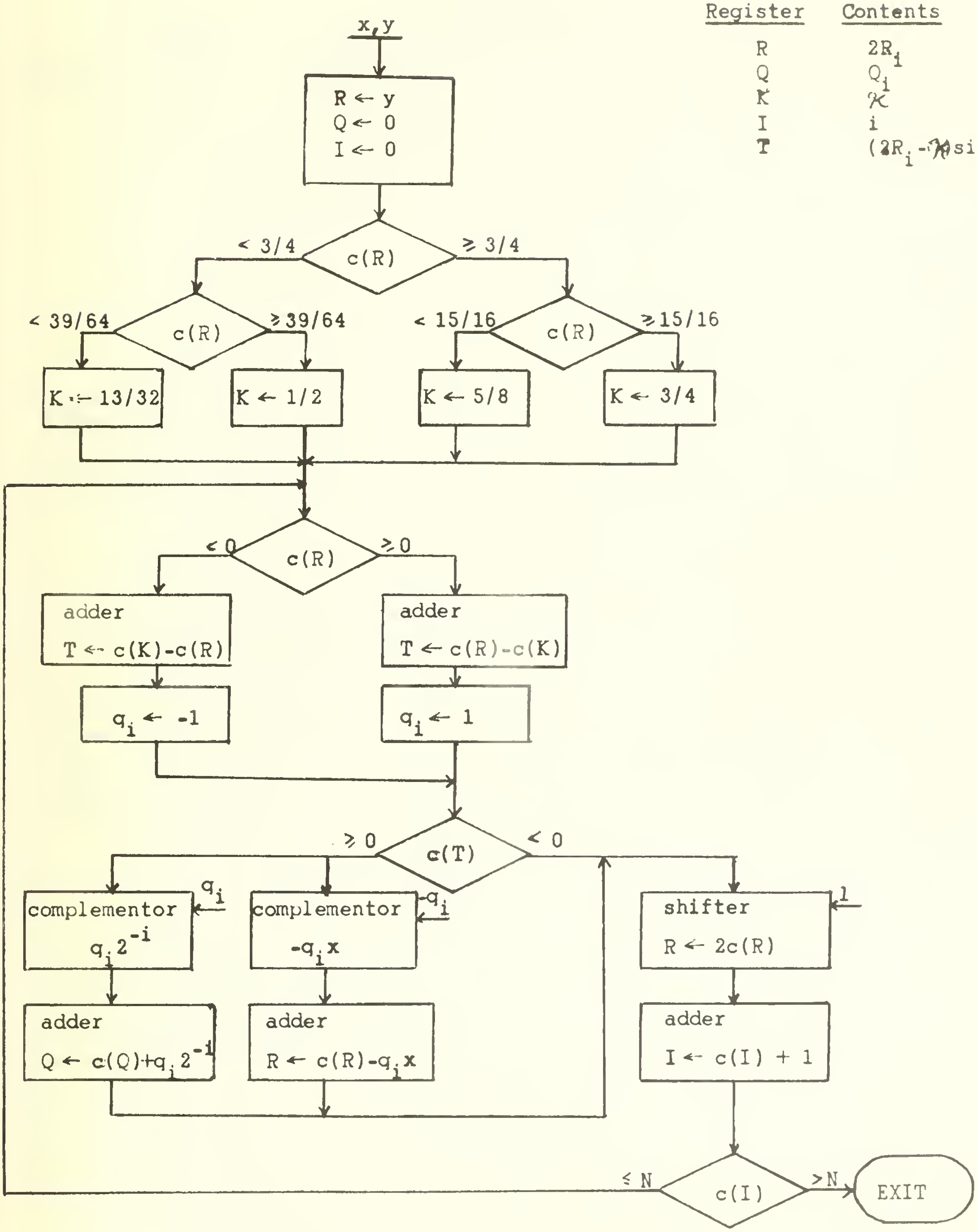
BLOCK DIAGRAM - DE LUGISH MULTIPLICATION

Register	Contents
U	U_i
P	p_i
I	i
T	test value



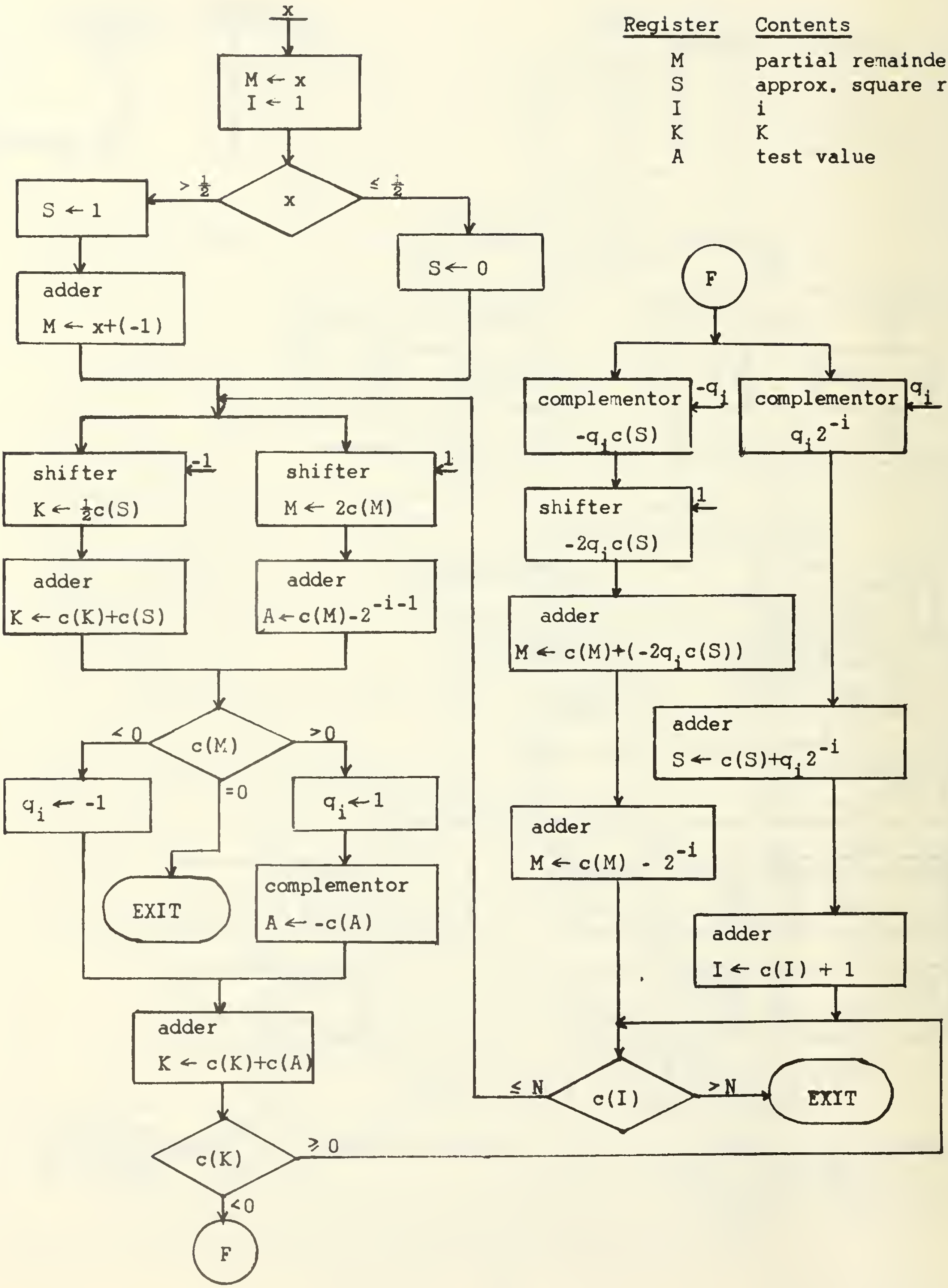
BLOCK DIAGRAM - METZE DIVISION

Register	Contents
R	$2R_i$
Q	Q_i
K	$\mathcal{K}$
I	i
T	$(2R_i - \mathcal{K})\text{sign}(R_i)$



BLOCK DIAGRAM - METZE SQUARE ROOT

Register	Contents
M	partial remainder
S	approx. square root
I	i
K	K
A	test value



Appendix B: CORDIC Simulation Program

```
C     PROGRAM                          C O R D I C
C
C     THIS PROGRAM SIMULATES THE GENERALIZED CORDIC ALGORITHM OF WALTHER
C
C     ROM VALUES ARE COMPUTED WITH ROUNDING TO NPB + NPG BITS ACCURACY
C
C     IN THE ITERATIONS, CHOPPING IS USED AFTER MULTIPLICATIONS.  THIS
C     CORRESPONDS TO A RIGHT SHIFT OFF THE END
C
C
C     INPUTS ARE AS FOLLOWS
C
C     M                   IS M
C     MODE                IS 1 FOR ROTATION, 2 FOR VECTORING
C     NPB                 IS THE NUMBER OF PRECISION BITS OF THE COMPUTER
C     NGB                 IS THE NUMBER OF GUARD BITS WITH WHICH THE
C                         CALCULATIONS ARE PERFORMED  (NOTE THAT WITH 32 BIT
C                         ARITHMETIC THE VALUE OF NPB+NGB MUST BE NO MORE THAN
C                         31)
C     IO                  OUTPUT IS GIVEN ON THE ITERATION WHEN I/IO IS AN
C                         INTEGER
C     X                   THE INPUT VALUE OF X (SHOULD BE LESS THAN ONE)
C     Y                   THE INPUT VALUE OF Y (SHOULD BE LESS THAN ONE)
C     Z                   THE INPUT VALUE OF Z (IS IN DEGREES IF M = 1, AND
C                         SHOULD BE  180 OR LESS IN MAGNITUDE.   IF M IS -1
C                         OR 0   Z SHOULD BE LESS THAN ONE IN MAGNITUDE)
C
      DIMENSION ROM(2,50),NXI(32),NYI(32),NZI(32)
      INTEGER ROM,SFP,SFG,TSF,ZI,XI,YI,XTEMP,DELI,M,MODE,NPB,D1,SZ,K
     1 ,REPEAT,XOR,YOR,ZOR,HSFG
      REAL*8 D,DATANH,RK,X,Y,Z,PI,ANG,XC,YC,ZC
      DATANH(X) = .5D0*DLOG((1.D0 + X)/(1.D0 - X))
      PI = DATAN(1.D0)*4.D0
  100 READ 1,M,MODE,NPB,NGB,IO
      IF(NPB.LE.0)STOP
      REPEAT = 4
      SFP = 2**NPB
      SFG = 2**NGB
      HSFG = SFG/2
      TSF = SFP * SFG
C
C       INITIALIZE ROM,COMPUTE SCALE FACTOR
C
      RK = 1.D0
      D1 = 1
      IF(M.LE.0)D1 = 2
      I = 0
      J = 0
  120 I = I + 1
      J = J + 1
      D = 1.D0/D1
      ROM(2,I) = TSF/D1
      IF(M)140,150,160
  140 ROM(1,I) = DATANH(D)*TSF     +.5D0
      GO TO 180
  150 ROM(1,I) = TSF/D1
      GO TO 180
  160 ROM(1,I) = DATAN(D)*TSF + .5D0
  180 RK = RK*(1.D0 + M*D**2)
  190 IF(M)195,200,200
  195 IF(J.NE.REPEAT)GO TO 200
      I = I + 1
      REPEAT = 3*REPEAT + 1
      ROM(1,I) = ROM(1,I-1)
      ROM(2,I) = ROM(2,I-1)
      RK = RK*(1.D0 + M*D**2)
  200 IF(D1.GT.SFP)GO TO 201
      D1 = 2*D1
      GO TO 120
  201 NMAX = I
      IF(NGB.LE.0)NMAX = NMAX - 1
      K = DSQRT(RK)*TSF + .5D0
      READ 2,X,Y,Z
      PRINT 5,M,MODE,NPB,NGB,X,Y,Z

       SCALE INPUTS TO INTEGER (NPB+NGB BITS) AND DO FIRST 90 DEGREE
      ROTATION

      IF(M.GT.0)GO TO 205
      ZI = Z*SFP + DSIGN(.5D0,Z)
      XI = X*SFP/K*TSF + DSIGN(.5D0,X)
      YI = Y*SFP/K*TSF + DSIGN(.5D0,Y)
      GO TO 250
  205 ANG = Z/18.D1*PI
      GO TO (210,220),MODE
  210 SZ = DSIGN(1.D0,Z)
      GO TO 230
  220 SZ = -DSIGN(1.D0,Y)
  230 IF(SZ.EQ.0)SZ = 1
      ZI = (Z - SZ*90)/1.8D2*PI*SFP + DSIGN(.5D0,Z - SZ*90)
      XI = -SZ*Y*SFP/K*TSF +    DSIGN(.5D0,-SZ*Y)
      YI = SZ*X*SFP/K*TSF + DSIGN(.5D0,SZ*X)
  250 ZI = ZI*SFG
      XI = XI*SFG
      YI = YI*SFG

      NOW START THE ROTATIONS

      DO 400 I=1,NMAX
      J = I - 1
      IF(MOD(J,10).NE.0)GO TO 280
      CALL BINARY(XI,NXI,32)
      CALL BINARY(YI,NYI,32)
      CALL BINARY(ZI,NZI,32)
      PRINT 3,J,NXI,NYI,NZI,XI,YI,ZI
  280 CONTINUE
      GO TO (300,310),MODE
  300 SZ = ISIGN(1,ZI)
      GO TO 320
  310 SZ = -ISIGN(1,XI)*ISIGN(1,YI)
  320 IF(SZ.EQ.0)SZ = 1
      DELI = -SZ*ROM(2,I)
      XTEMP = XI
      IDLCSF = TSF/DELI
      XI = XI + M*YI/IDLCSF
      YI = YI - XTEMP/IDLCSF
      ZI = ZI - SZ*ROM(1,I)
  400 CONTINUE
      CALL BINARY(XI,NXI,32)
      CALL BINARY(YI,NYI,32)
      CALL BINARY(ZI,NZI,32)
      PRINT 3,NMAX,NXI,NYI,NZI,XI,YI,ZI

      SUMMARY OF RESULTS

      LPTG = MODE + 2*(M + 1)
      GO TO (410,420,430,440,450,460),LPTG
  410 XC = X*DCOSH(Z) + Y*DSINH(Z)
      YC = Y*DCOSH(Z) + X*DSINH(Z)
      ZC = 0.D0
      GO TO 470
  420 XC = DSQRT(X**2 - Y**2)
      YC = 0.D0
      ZC = Z + DATANH(Y/X)
      GO TO 470
  430 XC = X
      YC = Y + X*Z
      ZC = 0.D0
      GO TO 470
  440 XC = X
      YC = 0.D0
      ZC = Z + Y/X
      GO TO 470
  450 XC = X*DCOS(ANG) - Y*DSIN(ANG)
```

```
      YC = Y*DCOS(ANG) + X*DSIN(ANG)
      ZC = 0.D0
      GO TO 470
  460 XC = DSQRT(X**2 + Y**2)
      YC = 0.D0
      ZC = ANG + DATAN2(Y,X)
  470 XI = (XI + ISIGN(HSFG,XI))/SFG
      YI = (YI + ISIGN(HSFG,YI))/SFG
      ZI = (ZI + ISIGN(HSFG,ZI))/SFG
      XCR = XC*SFF + DSIGN(.5D0,XC)
      YCR = YC*SFF + DSIGN(.5D0,YC)
      ZCR = ZC*SFF + DSIGN(.5D0,ZC)
      PRINT 4,SFP,XI,XCR,YI,YCR,ZI,ZCR
      GO TO 100
    1 FORMAT(10I5)
    2 FORMAT(8D10.0)
    3 FORMAT(I3,3(1X,32A1),3I10)
    4 FORMAT(//'0SUMMARY OF RESULTS                        COMPUTED   CORRECT'/
     1 '0DENOMINATOR',I12,3X,'X',2I11/27X,'Y',2I11/27X,'Z',2I11//
     2 16(' *  ** * '),'  RF')
    5 FORMAT(///'CM,MODE,NPB,NGB',4I5/'0X,Y,Z',1P3D15.6/'0 I',
     1 27X,'XI',31X,'YI',31X,'ZI',10X,'XI',8X,'YI',8X,'ZI'//)
      END
```

```
      SUBROUTINE BINARY(NI,NC,ND)
C
C     THIS ROUTINE CONVERTS THE NUMBER  NI  TO SIGNED BINARY (BCD)
C     THE CHARACTERS ARE STORED IN THE ARRAY NI
C     THE LEAST SIGNIFICANT DIGIT IS STORED IN NI(ND)
C     IT IS ASSUMED THAT NI IS NO BIGGER THAN 2**(ND-2) - 1 IN MAGNITUDE
C
      DATA ISP,ISM,ISB,IS1,IS0 /1H+,1H-,1H ,1H1,1H0/
      DIMENSION NC(1)
      DO 100 I=1,ND
  100 NC(I) = ISB
      IS = ISP
      IF(NI.LT.0)IS = ISM
      N = IABS(NI)
      DO 200 I=1,ND
      M = N/2
      K = ND - I + 1
      NC(K) = IS0
      IF(N.NE.2*M)NC(K) = IS1
      IF(M.NE.0)GO TO 200
      IF(K.GT.1)NC(K-1) = IS
      RETURN
  200 N = M
      RETURN
      END
```

Bibliography

1. S. Anderson, J. Earle, R. Goldschmidt, and D. Powers, "IBM System 360 Model 91: Floating Point Execution Unit", IBM J. Res. and Dev. 11(1967) 34-53.

2. D. Cantor, G. Estrin, and R. Turn, "Logarithmic and Exponential Function Evaluation in a Variable Structure Digital Computer", IRE Trans. on Electronic Computers EC-11(1962)155-164.

3. Maurus Cappa and V. Carl Hamacher, "An Augmented Iterative Array for High-Speed Binary Division", IEEE Trans. on Computers C-22(1973)172-175.

4. Tien Chi Chen, "A Binary Multiplication Scheme Based on Squaring", IEEE Trans. on Computers C-20(1971)678-680.

5. ________, "Automatic Computation of Exponentials, Logarithms, Ratios, and Square Roots", IBM J. Res. and Dev. 16(1972)380-388.

6. David S. Cochran, "Algorithms and Accuracy in the HP-35", H. P. Journal, June 1972; 10-11.

7. M. Combet, H. Van Zonneveld, and L. Verbeek, "Computation of the Base Two Logarithm of Binary Numbers", IEEE Trans. on Electronic Computers EC-14(1965)863-867.

8. D. Cowgill, "Logic equations for a built-in square root method", IEEE Trans. on Electronic Computers EC-13(1964)156-157.

9. K. J. Dean, "Cellular logical array for extracting square roots", Electron. Lett. 4(1966)314-315.

10. ________, "A precision code converter for the reciprocals of binary numbers", The Computer Bulletin, June 1968; 55-58.

11. ________, "Binary logarithms", Electronic Engineering, 40(1968), p. 560-562.

12. ________, "Logical circuits for use in iterative arrays", Electron. Lett. 4(1968)81-82.

13. ________, "A design for a full multiplier, Proc. IEEE 115(1968)1592-1594.

14. ________, "Some applications of cellular logic arithmetic arrays" Radio and Electronic Engineer 37(1969)225-227.

15. ________, "A fresh approach to logarithmic computation", Electronic Engineering 41(1969)488-490.

16. Bruce G. DeLugish, "A class of algorithms for automatic evaluation of certain elementary functions in a binary computer", Ph.D Thesis, University of Illinois, 1970.

17. R. C. Devires and M. H. Chao, "Fully iterative array for extracting square-roots", Electron. Lett. 6(1970)255-256.

18. Miloš D. Ercegovac, "Radix 16 evaluation of some elementary functions", Report Number UIUCDCS-R-72-540, Dept. of Computer Science, University of Illinois, 1972.

19. Domenico Ferrari, "A division method using a fast multiplier", IEEE Trans. on Computers EC-16(1967)224-226.

20. Michael J. Flynn, "On division by functional iteration", IEEE Trans. on Computers C-19(1970)702-706.

21. Munehiro Goto and Teruo Fukumura, "Application of a generalized minimal representation of binary numbers to square rooting", Elect. Co. J 50(1967) 122-128.

22. H. H. Guild, "Cellular logical array for non-restoring square-root extraction", Electron. Lett. 6(1970)66-67.

23. Albert Hemel, "Square root extraction with read only memories", Computer Design, April 1972; 100-102, 104.

24. Andrew Edward Huber, "Division algorithms for binary computers", M.S. Thesis, San Diego State College, 1971.

25. A. A. Kamal and M. A. N. Ghannum, "High speed multiplication systems", IEEE Trans. on Computers C-21(1972)1017-1021.

26. George K. Kostopoulos, "Computing the square root of binary numbers", Computer Design, August 1971; 53-57.

27. E. V. Krishnamurthy, "On computer multiplication and division using binary logarithms", IEEE Trans. on Electronic Computers EC-12(1963)319-320.

28. ________, "A simple algorithm for evaluating positive values of the function x^y", IEEE Trans. on Electronic Computers EC-13(1964)55.

29. ________, "An optimal iterative scheme for high speed division", IEEE Trans. on Computers C-19(1970)227-231.

30. E. H. Lenaerts, "Automatic square rooting", Electronic Engineering, 23(1955)287-289.

31. Michael A. Liccardo, "An interconnect processor with emphasis on CORDIC mode operation", M.S. Thesis, Department of Electrical Engineering, University of California at Berkeley, 1968.

32. H. Ling, "High speed computer multiplication using a multiple-bit decoding algorithm", IEEE Trans. on Computers C-19(1970)706-709.

33. ________, "High speed division for binary computers", AFIPS Conference Proceedings 38(1971)373-378.

34. R. J. Linhardt and H. S. Miller, "Digit-by-digit transcendental function evaluation", RCA Review 30(1969)209-247.

35. J. Robert Logan, "A design technique for digital squaring networks", Computer Design, February 1970; 84,86,88.

36. ________, "A square-summing high-speed multiplier", Computer Design, June 1971; 67-70.

37. J. C. Majithia, "Non-restoring binary division using a cellular array", Electron. Lett. 6(1970)303-304.

38. J. C. Majithia and R. Kitai, "A cellular array for the nonrestoring extraction of square roots", IEEE Trans. on Computers C-20(1971)1617-1618.

39. D. Marino, "New algorithms for the approximate evaluation in hardware of binary logarithms and elementary functions", IEEE Trans. on Computers C-21(1972)1416-1421.

40. J. E. Meggitt, "Pseudo-division and pseudo-multiplication processes", IBM Journal, April 1962; 210-226.

41. Gernot Metze, "A class of binary divisions yielding minimally represented quotients", IRE Trans. on Electronic Computers EC-11(1962)761-764.

42. ________, "Minimal square rooting", IEEE Trans. on Electronic Computers EC-14(1965)181-185.

43. John N. Mitchell, Jr., "Computer multiplication and division using binary logarithms", IRE Trans. on Electronic Computers EC-11(1962)512-517.

44. Salil K. Nandi and E. V. Krishnamurthy, "A simple technique for digital division", Com. of ACM 10(1967)299-301.

45. J. D. Nicoud and R. Dessoulavy, "Logarithmic converter", Elect. Lett. 7(1971)230-231.

46. Michael D. Perle, "The dual logarithm algorithms", Computer Design, December 1969; 88,90.

47. ________, "CORDIC technique reduces trigonometric function look-up", Computer Design, June 1971; 72,74,76,78.

48. P.W. Philo, "An algorithm to evaluate the logarithm of a number to base 2, in binary form", The Radio and Elect. Engr. 38(1969)49-50.

49. C. V. Ramamoorthy, James R. Goodman, and K. H. Kim, "Some properties of iterative square-rooting methods using high-speed multiplication", IEEE Trans. on Computers, C-21(1972)837-847.

50. J. E. Robertson, "A new class of digital division methods", IRE Trans. on Electronic Computers EC-7(1958)218-222.

51. ________, "The correspondence between methods of digital division and multiplication recoding procedures", IEEE Trans. on Computers C-19(1970) 692-701.

52. C. Ross, "Trig. function generator", MOS Brief 10, National Semiconductor Corp, Santa Clara, CA., January 1970.

53. DV Sankar, S. Chakrabarti, E. V. Krishnamurthy, "Deterministic division algorithm in a negative base", IEEE Trans. on Computers, C-22(1973)125-128.

54. ________, "Arithmetic algorithms in a negative base", IEEE Trans. on Computers C-22(1973)120-125.

55. B. P. Sarkar and E. V. Kishnamurthy, "Economic Pseudo-division processes for obtaining square root, logarithm, and arctan", IEEE Transactions on Computers C-20(1971)1589-1593.

56. Hermann Schmid and Anthony Bogacki, "Use decimal CORDIC for generation of many transcendental functions", EDN, February 20, 1973; 64-73.

57. Hermann Schmid and David Busch, "Generate functions from discrete data", Electronic Design 20(1970)42-47.

58. Z. Shaham and Z. Riesel, "A note on division algorithms based on multiplication", IEEE Trans. on Computers C-21(1972)513-514.

59. Barry J. Shepherd, "Right shift for low cost multiply and divide", IEEE Trans. on Computers C-20(1971)1586-1589.

60. Shankar Singh and Ronald Waxman, "Multiple operand addition and multiplication", IEEE Trans on Computers C-22(1973)113-120.

61. W. H. Specker, "A class of algorithms for ln x, exp x, sin x, cos x, $\tan^{-1}x$ and $\cot^{-1}x$", IEEE Trans. on Electronic Computers EC-14(1965)85-86.

62. Renato Steffanelli, "A suggestion for a high-speed parallel binary divider", IEEE Trans. on Computers C-21(1972)42-55.

63. Jack E. Volder, "Binary computation algorithms for coordinate rotation and function generation", CONVAIR Report IAR-1,148, Aeroelectronics Group, June 1956.

64. Jack. E. Volder, "The CORDIC trigonometric computing technique", IRE Trans. on Electronic Computers EC-8(1959)330-334.

65. L. B. Wadel, "Negative base number systems", IRE Trans. on Electronic Computers EC-6(1957)123

66. J. S. Walther, "A unified theory for elementary functions", AFIPS Conference Proceedings 38(1971)379-385.

67. ________, "A floating point processor for elementary functions", Manuscript.

Distribution List

Defense Documentation Center 12
Cameron Station
Alexandria, Virginia 22314

Library 2
Naval Postgraduate School
Monterey, California 93940

Naval Electronics Laboratory Center 2
Research Library, Code 6700
San Diego, California 92152

NELC 8
Mr. W. J. Dejka, Code 4300
San Diego, California 92152

NELC 1
Mr. S. Snyder, Code 4000
San Diego, California 92152

NELC 1
Mr. H. T. Mortimer, Code 0220
San Diego, California 92152

NELC 1
Mr. M. Lamendola, Code 5200
San Diego, California 92152

NELC 1
Mr. W. Loper, Code 5200
San Diego, California 92152

NELC 1
Dr. R. Kolb, Code 3300
San Diego, California 92152

NELC 1
Mr. E. E. McCown, Code 3100
San Diego, California 92152

NELC 1
Mr. H. F. Wong, Code 3200
San Diego, California 92152

NELC 1
Dr. P. C. Fletcher, Code 2000
San Diego, California 92152

NELC 1
Mr. John Wasilewski, Code 4300
San Diego, California 92152

Dean J. M. Wozencraft, Code 023 2
Dean of Research
Naval Postgraduate School
Monterey, California 93940

Professor W. M. Woods, Code 53Wo 1
Chairman, Department of Mathematics
Naval Postgraduate School
Monterey, California 93940

Professor Richard Franke, Code 53Fe 5
Department of Mathematics
Naval Postgraduate School
Monterey, California 93940

Professor Craig Comstock, Code 53Zk 1
Department of Mathematics
Naval Postgraduate School
Monterey, California 93940

Professor R. E. Barnhill 1
University of Utah
Salt Lake City, Utah 84112

Steve Walther 1
Hewlett-Packard Laboratories
Electronics Research Laboratory
1501 Page Mill Road
Palo Alto, California 94304

Dr. Richard Lau 1
Office of Naval Research
Pasadena, California 91100

Dr. Leila Bram 1
Director, Mathematics Program
Office of Naval Research
Arlington, Virginia 22217

Professor V. Michael Powers, Code 52Pw 1
Department of Electrical Engineering
Naval Postgraduate School
Monterey, California 93940

Professor Gary Kildall, Code 53Kd 1
Department of Mathematics
Naval Postgraduate School
Monterey, California 93940

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Naval Postgraduate School Monterey, California	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP

3. REPORT TITLE

An Analysis of Algorithms for Hardware Evaluation of Elementary Functions

4. DESCRIPTIVE NOTES *(Type of report and, inclusive dates)*

Technical Report, 1973

5. AUTHOR(S) *(First name, middle initial, last name)*

Franke, Richard

6. REPORT DATE 8 May 1973	7a. TOTAL NO. OF PAGES 84	7b. NO. OF REFS 67

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S) NPS-53FE73051A
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Navy Electronics Laboratory Center San Diego, CA 92152

13. ABSTRACT

Algorithms for the automatic evaluation of elementary functions were studied. Available algorithms obtained from current literature were analyzed to determine their suitability for hardware implementation, in terms of their accuracy, convergence rate, and hardware requirements. The functions considered were quotient, arctangent, cosine/sine, exponential, power function, logarithm, tangent, square root, and product.

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6811

UNCLASSIFIED
Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Elementary Functions						
Hardware Evaluation						
Algorithms						

[redacted]