RADIX 16 DIVISION, MULTIPLICATION, LOGARITHMIC AND EXPONENTIAL ALGORITHMS BASED ON CONTINUED PRODUCT REPRESENTATIONS

BY

MILOŠ DRAGUTIN ERCEGOVAC

Elec. Engr., University of Belgrade, 1965

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1972

Urbana, Illinois

ACKNOWLEDGMENT

I wish to express my sincerest gratitude to my advisor, Professor James E. Robertson of the Department of Computer Science of the University of Illinois for his highly valued guidance, suggestions and support. I thank also National Science Foundation and the Department of Computer Science of the University of Illinois for their support.

Thanks are also due to Mr. Kishor S. Trivedi for many helpful discussions. Finally, I would like to thank Mrs. June Wingler for her fine job of typing and Mr. Mark Gobel for excellent drawings.

TABLE OF CONTENTS

1. INTRODUCTION

There is no doubt that the available technological possibilities could justify hardware implementation of a much wider class of functions than is presently done. If the corresponding algorithms have similarity, this is even more true. One effective way to obtain such a class of algorithms is to use, in a convenient way, continued products (CP) or continued sums (CS) during the function evaluation. The use of continued products in the calculation of some elementary functions appears as early as 1959, in Volder's CORDIC technique[5]. The main results of this approach have been recently summarized in the form of a unified algorithm by Walther[6]. Specker[7] derived also a class of algorithms using the concept of continued products. Of particular importance and usefulness are the results obtained by DeLugish[1]. He has defined efficient algorithms for a wide class of functions including division, multiplication, square root, logarithm, exponential, trigonometric and inverse trigonometric functions, with operation times from 1 to 3 multiplication cycle times. These algorithms are specified for the radix 2, using a redundant digit set {-1, 0, 1} in continued products (sums). The main idea is replacement of a required operation or function evaluation by two simple step by step processes using addition/subtraction, shifting and possibly a set of precomputed constants, stored in a read-only memory. One of the processes is normalization, through which the digits of continued products (sums) are generated and another is the related result evaluation. These processes can be carried out in parallel, so for a fast operation at least two arithmetic units, almost the same, are required.

The work done here is based upon the results obtained by DeLugish. It is motivated by the fact that the higher radix implementation offers some speed/hardware trade-offs, worth investigating. In particular, the radix 16 is considered in four algorithms: division, multiplication, logarithm and exponential. The central problem is to find the rules, which are more difficult when the higher radix is used, for selection of the digits of continued products (sums). The rules and the complete algorithms are developed for fractional parts in the conventional range $[1/2, 1)$, of the floating point numbers. The radix 16 merely means "4 bits-at-a-time" and represents, in some sense, the radix of implementation, not of an operand. The exponent arithmetic, being simple, is not considered. The use of a redundant representation [2, 3] effects the selection rules, but not the number of steps to be performed, the probability of zero, "no addition," being too small for the radix 16 approach. In the binary case [1], the redundancy is essential also in decreasing the average number of full steps. As a difference, in the radix 16 case, the number of steps is fixed and corresponds to the number of radix 16 digits, used to represent the fractional part. The digit-by-digit evaluation, employed in the described algorithm, is not a consequence of inherent properties in the continued products (sums) approach, but reflects a realization strategy, which attempts to achieve a reasonably fast implementation for all functions under consideration, retaining at the same time simplicity. Some comparisons between radix 2 and radix 16 approaches are made and a more efficient solution, requiring essentially one "pipelined" arithmetic unit is described.

2. MULTIPLICATIVE NORMALIZATION

By normalization we mean a step by step transformation of a given number $X_0 \in [1/2, 1)$ to one (or to any other number N, in general). For uniformity and simplicity of later described algorithms, the linear convergence is imposed on normalization. Namely, an m-digit normalized number is obtained in (m+1) iterative steps. If the reciprocal of a given number X_0 can be represented in a continued product form as

$$1/X_0 = \prod_{i=0}^{m} M_i$$

then the normalization of X_0 to one can be achieved through a multiplicative iteration:

$$X_{k+1} = X_k \cdot M_k, \qquad k = 0, \ldots, m \tag{2.1}$$

and

$$X_{m+1} = X_0 \cdot \prod_{i=0}^{m} M_i \tilde{=} 1$$

The factors of a continued product representation are of the form

$$M_k = 1 + S_k \cdot 16^{-k} \tag{2.2}$$

where S_k is a one-digit constant, so that an implementation of (2.1) will require only addition and shift operations. The value of S_k is chosen such that the error ϵ_{k+1} after step k becomes

$$|\epsilon_{k+1}| = |1 - X_k \cdot (1 + S_k \cdot 16^{-k})| \leq \frac{|S_k|_{max}}{15} \cdot 16^{-k} \tag{2.3}$$

where $|S_k|_{max}$ is the largest constant in the chosen set $\{S_k\}$. In other words, at every step k, by proper choice of constant S_k, the k-th digit of the partially normalized number X_k will be forced to zero or (radix - 1). Therefore, the final normalized quantity X_{m+1} will differ by at most 16^{-m} from unity. To define the normalization procedure, one must find the rules for selection of the proper value of S_k, given X_k and the set $\{S_k\}$. The set $\{S_k\}$ is determined as follows. To make the selection process simple, it is essential to have a redundant number representation. Therefore, the set $\{S_k\}$ will contain more than r elements, both positive and negative. The maximal value $|S_{kmax}|$, or better, the redundancy ratio is obtained from the practical requirements[3]:

- one of the simplest ways to form the term $S_k(X_k \cdot 16^{-k})$ in the recursion,

$$X_{k+1} = X_k M_k = X_k + S_k X_k 16^{-k} \tag{2.4}$$

is to use a multilevel adder structure, with corresponding selection-complementation networks generating the following sets of multiples:

$\{0, \pm 1, \pm 2\} * (X_k \cdot 16^{-k})$ in level 1;

$\{0, \pm 4, \pm 8\} * (X_k \cdot 16^{-k})$ in level 2.

Therefore, the maximal value should be $|S_{kmax}| = 10$ corresponding to the redundancy ratio $|S_{kmax}|/(r-1) = 2/3$. The set of constants S_k is then

$$\{\overline{10}, \ldots, \overline{1}, 0, 1, \ldots, 10\} \tag{2.5}$$

where the overbar denotes negative values. Now the error (2.3) becomes

$$|\epsilon_{k+1}| = |1 - X_k \cdot M_k| \leq \frac{2}{3} \cdot 16^{-k} \tag{2.6}$$

It is straightforward to show that for $X_0 \in [1/2,1)$, the reciprocal $1/X_0$ can always be represented in a continued product form using constants S_k from the set (2.5). Therefore, the normalization procedure defined by the recursion (2.1) is always possible.

The main problem now is to define a practical selection procedure. The constants S_k are selected in such a way that the error condition (2.6) is satisfied for every step. Since the set of constants S_k has been chosen to be redundant, the range of X_k, for all k, can be partitioned in the overlapping intervals, each corresponding to a particular constant S_k. In the overlaps, at least two constants S_k, differing by 1, are a valid choice. S_k can be determined on the basis of X_k, but to keep selection dependent on the same register positions, i.e., to retain the same "weights" for selection rules, it is convenient to define the scaled remainder as

$$R_k = 16^{k-1}(X_k - 1), \qquad 0 < k \leq m \tag{2.7}$$

The selection process can now be carried on the remainders, obtained recursively from

$$R_{k+1} = 16R_k + S_k + 16^{-k+1}S_kR_k, \qquad 0 < k \leq m \tag{2.8}$$

This recursion follows from (2.1), (2.2) and (2.7).

If the general form of this recursion for radix r is considered, then the following remarks concerning implementation requirements can be made:

i) The number of shifting paths, necessary to generate the last term in the recursion (2.8), is inversely proportional to the radix r;

ii) For the higher radices of the form

$$r = 2^{2p}, \qquad p = 1, 2, 3, \ldots$$

such that multiple formation can be done with a cascade of adders, and with the set of S_k's such that $|S_{kmax}| = \frac{2}{3}(r-1)$, the number of extra levels with respect to the radix 2 is $p - 1$. Clearly, the full carry propogation need be provided only at the last level.

Now, starting from the condition for error (2.6), redefined for scaled remainders as

$$-2/3 < R_k \leq 2/3 \qquad (2.9)$$

the selection rules can be derived. The intervals can be determined by straightforward calculations:

- for every S_k, given bounds on R_{k+1} (2.9), find the interval boundaries as the minimal and maximal value of R_k such that equation (2.8) holds. In addition, the continuity of the range should be preserved by retaining only the overlapping intervals. The numbers representing boundaries between intervals should be simple in the binary sense so that limited precision can be used in implementation of the selection rules.

From the definition of the scaled remainder (2.7) it can be observed that the normalization will be more accurate if the initial step (k=0) is performed on X_0 and then continued using R_k for $k = 1, \ldots, m$. Since $R_1 = X_1 - 1$ this change in the procedure is almost trivial.

The rules of the selection for the initial step can be made very simple, due to the fact that S_0 may be chosen to be either 0 or 1. The rules are:

$$S_0 = 1 \quad \text{if} \quad 1/2 \leq X_0 < 5/8;$$

$$S_0 = 0 \quad \text{if} \quad 5/8 \leq X_0 < 1.$$

The choice of 5/8 as the boundary value is made so that the X_1 will be in a convenient range. From the rules, the range of X_1 is [5/8, 5/4). It is easy to see that even for m = 1, the continued products $\prod_{i=1}^{m}(1+S_i 16^{-i})$ can represent values less than 4/5 or greater than 8/5, hence making normalization successful. Therefore, the restriction of S_0 to the values 0 or 1 is valid.

The precision, necessary to express boundaries between intervals is at most six bits after the radix point. For convenience the bounds of the intervals will be given also to this precision.

For the step k = 1, the lower and upper bounds of intervals containing R_1 and denoted as a and b, are calculated for all possible S_1 and displayed in the following table.

Table 2.1

S_1	$a \leq 64R_1 \leq b$	
10	-26	-23
9	-24	-22
8	-23	-20
7	-21	-18
6	-19	-16
5	-17	-14
4	-14	-11
3	-12	-8
2	-9	-5
1	-6	-2
0	-2	3
-1	2	7
-2	7	12
-3	12	18
-4	18	24
*		
-5	26	32
-6	35	42
-7	46	54
-8	60	69
-9	77	88
-10	100	113

The values below the starred line may not be used since corresponding intervals are not contiguous. Since $-3/8 \leq R_1 < 1/4$, as follows from the initialization rules (k=0), the constants S_1 can be, without problems, restricted to the set $\{\overline{3}, \ldots, 9\}$.

By the same procedure the interval bounds a and b for R_2 are calculated. The correspondence between values of S_2 and allowed intervals is given in Table 2.2.

Table 2.2

S_2	$a \leq 64R_2 \leq b$	
10	-42	-36
9	-37	-33
8	-33	-29
7	-29	-25
6	-25	-21
5	-22	-18
4	-18	-14
3	-14	-10
2	-10	-6
1	-6	-2
0	-2	3
-1	2	6
-2	6	10
-3	10	14
-4	14	18
-5	18	23
-6	23	27
-7	27	31
-8	31	35
-9	35	39
-10	39	42

The intervals in which R_2 may be found are contiguous and S_2 can have all values from the set $\{\overline{10}, \ldots, 10\}$.

For the remaining steps, $k \geq 3$, the simple relationship holds between the value of S_k and the bounds of the corresponding interval:

$$(-2S_k - 1) \leq 32R_k \leq (-2S_k + 1) \qquad (2.10)$$

and

$$S_k \in \{\overline{10}, \ldots, 10\}$$

The result (2.10) indicates, first, that the selection rules for $k > 2$ are invariant and, second, that the selection can be performed by rounding the scaled remainder to one non-sign digit (in radix 16). The selection process itself becomes very simple after the first three steps, due to the following fact. The last term in the remainder recursion, $16^{-k}S_kR_k$, cannot affect the most significant bits of R_{k+1}, used in selecting S_{k+1}, for $k > 2$. At least $k - 3$ most significant digits of R_k remain unchanged except for possible complementation, due to the change of the sign. Therefore, at the k-th step, constants S_k, S_{k+1}, ..., S_{2k-3} are known. For m digit precision, when $k \geq (m+3)/2$ all remaining constants S_k, ..., S_m are actually available and the process of normalization can be simplified. Namely, the basic remainder recursion (2.8) can be replaced by a simple form:

$$R_{k+1} = 16R_k + S_k, \qquad k \geq (m+3)/2 \qquad (2.11)$$

The aforementioned features reveal the amenability of the normalization procedure to higher radix implementations. Once the process gets started, remaining steps are progressively easier.

As mentioned before, for $k \geq 3$ the selection of S_k's can be performed through rounding, i.e., the most significant non-sign (radix 16) digit of R_k represents the correct value of S_k, after rounding. It is, then, natural to specify the selection rules for "irregular" steps $k = 0$, 1, and 2 through a modified rounding procedure rather than using a table look-up or a direct combinational approach.

The following definitions are relevant to the description of the selection rules as well as the algorithms.

Sign and magnitude representation of the constants S_k:

$$S_k = (1-2s_4)\sum_{i=0}^{3} s_i 2^i, \qquad s_i \in \{0,1\} \quad \text{for all } i; \tag{2.12}$$

Two's complement representation of scaled remainders:

$$R_k = -r_0 + \sum_{i=1}^{4m} r_i 2^{-i}, \qquad r_i \in \{0,1\} \quad \text{for all } i; \tag{2.13}$$

(m is the number of radix 16 digits)

Truncated scaled remainder:

$$\hat{R}_k = -r_0 + \sum_{i=1}^{6} r_i 2^{-i} \tag{2.14}$$

Non-sign part of $\hat{R}_k$:

$$T_k = \begin{cases} \sum_{i=1}^{6} r_i 2^{-i} & \text{if } r_0 = 0; \\ \sum_{i=1}^{6} \overline{r}_i 2^{-i} & \text{if } r_0 = 1; \end{cases} \tag{2.15}$$

Step-dependent rounding constant:

$$U_k = \sum_{i=1}^{6} u_i 2^{-i}, \qquad u_i \in \{0,1\} \quad \text{and} \qquad u_i = f_i(\hat{R}_k) \tag{2.16}$$

Then

$$S_k = \lfloor (T_k+U_k).16 \rfloor \quad \text{and}$$

$$\text{Sign}\,(S_k) = \overline{\text{Sign}}\,(R_k) \tag{2.17}$$

where $\lfloor Y \rfloor$ denotes largest integer not larger than Y.

Step k = 0

For the initial step, a modified procedure in selection is applied. The extension of normalization to the negative values of X_0 is easily achieved through the initial step. The selection rules and the starting value of the scaled remainder are specified as follows:

For $X_0 \in [1/2,1)$,

$$\begin{aligned} S_0 &= 1 \quad \text{if} \quad 1/2 \le X_0 < 5/8; \\ S_0 &= 0 \quad \text{if} \quad 5/8 \le X_0 < 1; \\ R_1 &= X_1 - 1 = X_0 + X_0 S_0 - 1 \end{aligned} \tag{2.18}$$

For negative values of X_0, it is a simple approach to determine S_0 by the rules analogous to (2.18) and then generate the negative of X_1 while calculating R_1 as

$$R_1 = -X_1 - 1 = \overline{(X_0 + X_0 S_0)} + 2^{-4m} - 1 \tag{2.19}$$

and proceed with normalization as in the case of positive X_0.

Step k = 1

From the Table 2.1, the interval break points are chosen so that the simple values of U_1 are sufficient to obtain S_1. According to the proposed approach

$$S_1 = \lfloor (T_1 + U_1).16 \rfloor \quad \text{and} \quad \text{Sign}\,(S_1) = \overline{\text{Sign}}\,(R_1)$$

In Table 2.3 the correspondence between intervals and rounding constants (U_1) for each allowed value of S_1 is given.

Table 2.3

$R_1 \leq 0$	S_1	$64(\hat{R}_1+1)$	$64T_1$	$64U_1$
	9	40	23	14
		41	22	
	8	42	21	
		43	20	
	7	44	19	10
		45	18	
	6	46	17	
		47	16	
	5	48	15	6
		49	14	
	4	50	13	
		51	12	
	3	52	11	3
		53	10	
		54	9	
	2	55	8	
		56	7	2
		57	6	
	1	58	5	
		59	4	
		60	3	
		61	2	
	0	62	1	
		63	0	
		0	0	
		1	1	
	-1	2	2	
		3	3	
		4	4	1
		5	5	
		6	6	
	-2	7	7	
		8	8	0
		9	9	
		10	10	
		11	11	
	-3	12	12	
		13	13	
		14	14	
		15	15	

Now, it is a simple task to find relations between U_1 and $\hat{R}_k$ in the form of Boolean equations. The derivation of those equations, listed below, is given in Appendix A.

$$
\begin{aligned}
U_1 &= \sum_{i=1}^{6} u_i 2^{-i} \\
u_1 &= u_2 = 0 \\
u_3 &= r_0 \bar{r}_2 \\
u_4 &= r_0 \bar{r}_4 (\bar{r}_2 + \bar{r}_3) \\
u_5 &= r_0 + \bar{r}_3 \bar{r}_4 \\
u_6 &= \bar{r}_3 r_4
\end{aligned}
\qquad (2.20)
$$

Step k = 2

Using the data from Table 2.2, the interval break points are selected in such a way that the corresponding values of T_2 and U_2 will produce correct values of S_2:

$$S_2 = \lfloor (T_2 + U_2)16 \rfloor \quad \text{and} \quad \text{Sign}\ (S_2) = \overline{\text{Sign}}\ (R_2)$$

The correspondence between $\hat{R}_2$ and U_2 is shown in Table 2.4.

The Boolean equations are derived in Appendix B and listed below:

$$
\begin{aligned}
u_1 &= u_2 = u_3 = u_4 = 0 \\
u_5 &= r_0 + \bar{r}_1 (\bar{r}_2 + \bar{r}_3) + r_6 \\
u_6 &= r_0 (r_1 + r_2 r_3)
\end{aligned}
\qquad (2.21)
$$

Table 2.4

$R_2 \leq 0$	S_2	$64(\hat{R}_2+1)$	$64T_2$	$64U_2$
	10	23	40	3
		26	37	
	9	27	36	
		30	33	
	8	31	32	
		34	29	
	7	35	28	
		38	25	
	6	39	24	
		42	21	
	5	43	20	
		44	19	
		45	18	2
	4	46	17	
		49	14	
	3	50	13	
		53	10	
	2	54	9	
		57	6	
	1	58	5	
		61	2	
	0	62	1	
		63	0	

Table 2.4 Continued

$R_2 \geq 0$	S_2	$64\hat{R}_2$	$64T_2$	$64U_2$
	0	0	0	2
		1	1	
	-1	2	2	
		5	5	
	-2	6	6	
		9	9	
	-3	10	10	
		13	13	
	-4	14	14	
		17	17	
	-5	18	18	
		21	21	
	-6	22	22	
		23	23	
		24	24	1*
		26	26	
	-7	27	27	
		30	30	
	-8	31	31	
		34	34	
	-9	35	35	
		38	38	
	-10	39	39	
		42	42	

* - or $64U_2 = 2$ whenever $t_6 = r_6 = 1$.

Step $k \geq 3$

As mentioned before, the selection procedure for all remaining steps consists of rounding:

$$S_k = \lfloor (T_k+U_k).16 \rfloor, \qquad \text{Sign } (S_k) = \overline{\text{Sign}}\ (R_k)$$

where $$U_k = \sum_{i=1}^{6} u_i 2^{-i} = 1/32, \quad \text{i.e.,}$$

$$u_j = 0, \qquad j \neq 5 \tag{2.22}$$

$$u_5 = 1$$

We now summarize the multiplicative normalization of a given number $X_0 = -x_0 + \sum_{i=1}^{4m} x_i 2^{-i}$ in the form of the following algorithm.

Algorithm N (Multiplicative Normalization): (2.23)

Step N1. [Initialize] $k \leftarrow 0$;

$S_0 \leftarrow 1$ if $1/2 \leq X_0 < 5/8$;

$S_0 \leftarrow 0$ if $5/8 \leq X_0 < 1$;

$R_1 \leftarrow X_0\ (1+S_0) - 1$;

Step N2. [Loop] for $k \leq m$ perform:

$k \leftarrow k + 1$;

$S_k \leftarrow \lfloor (T_k+U_k)16 \rfloor$; Sign $S_k \leftarrow \overline{\text{Sign}}\ R_k$;

if $k < (m+3)/2$ then:

$R_{k+1} \leftarrow 16R_k + S_k + 16^{-k+1} S_k R_k$;

else:

$R_{k+1} \leftarrow 16R_k + S_k$;

where $[Y]$ denotes the largest integer not larger than Y; T_k and U_k are defined according to (2.15) and (2.16) with

$$u_1 = u_2 = 0;$$
$$u_3 = K_1 r_0 \bar{r}_2;$$
$$u_4 = K_1 r_0 \bar{r}_4 (\bar{r}_2 + \bar{r}_3);$$
$$u_5 = K_1 (r_0 + \bar{r}_3 \bar{r}_4) + K_2 [r_0 + \bar{r}_1 (\bar{r}_2 + \bar{r}_3) + r_6] + K$$
$$u_6 = K_1 \bar{r}_3 r_4 + K_2 r_0 (r_1 + r_2 r_3)$$

and K_1, K_2 and K stand for k = 1, k =2 and $k \geq 3$, respectively.

This algorithm will normalize a given number X_0 to one in (m+1) steps with the error bound

$$|X_{m+1} - 1| \leq 2/3 \cdot 16^{-m} \qquad (2.24)$$

and simultaneously generate digits of the continued product representation of $1/X_0$. The method of multiplicative normalization is convergent by definition of the procedure and the existence of such procedure has been shown by construction of the selection rules. In deriving those rules, the aim was to achieve sufficient simplicity, not necessarily optimality. The implementation aspects will be discussed later. From the algorithms considered here, division and (natural) logarithm are based upon the multiplicative normalization and will be described in that order.

3. DIVISION

As stated before, the proposed algorithms apply to floating point numbers with binary radix of the exponent. The radix 16 or, in other words "4 bits at a time" is used to speed up operations on fractional parts. Since, in general, the exponent manipulation presents no problems, we will not be concerned with the exponent arithmetic here.

Let Y_0, $X_0 \in [1/2,1)$ be fractional parts of the given floating point dividend and divisor, respectively. Then by multiplying both the dividend and the divisor with the same sequence of factors M_k

$$Q = \frac{X_0}{X_0} = \frac{Y_0 \prod_{i=0}^{m} M_i}{X_0 \prod_{i=0}^{m} M_i} \tag{3.1}$$

if $X_0 \pi M_i \to 1$, the fractional part Q of the quotient is obtained as $Y_0 \pi M_i$. Defining the factors M_k to be of the form

$$M_k = 1 + S_k \cdot 16^{-k} \tag{3.2}$$

and $$S_k \in \{\overline{10}, \ldots, 10\}$$

one can determine constants S_k through the multiplicative normalization (Algorithm N). To form a desired quotient Q, let $Q_0 = Y_0$ and define recursively the partial result as

$$Q_{k+1} = Q_k \cdot M_k = Q_k(1+S_k \cdot 16^{-k}), \quad 0 \leq k \leq m \tag{3.3}$$

Then $Q = Q_{m+1}$ has m correct digits, the error bound in the normalized divisor being $|X_{m+1} - 1| \leq 2/3 \cdot 16^{-m}$.

In presenting the algorithm for division, as well as for the other operations, it is assumed that the normalization and the result evaluation are carried out in two similar arithmetic units. Later, when discussing implementation aspects, it will be shown how the proposed algorithms can be realized with essentially one arithmetic unit with a tolerable decrease in performance.

Algorithm D (Division)

(AU1: Normalization) (AU2: Result Evaluation)

Step D1. [Initialize] $k \leftarrow 0$;

Step N1 (Algorithm N); $Q_0 \leftarrow Y_0$;

Step D2. [Loop] for $k \leq m$ perform:

Step N2; $Q_{k+1} \leftarrow Q_k + Q_k S_k 16^{-k}$;

An example of the division is given in Figure 3-1. The "predictability" feature, described before (2.11), is apparent at step $k = 7$: the first five digits of R_8, when recoded, are the next five constants $S_8, \ldots, S_{12}$.

In Figure 3-2, the basic hardware configuration, consisting of two arithmetic units, is shown. The control part is not described. The only difference between the two arithmetic units is that AU1 has the additional network TU and the five-bit register S, used in the selection process.

Then $Q = Q_{m+1}$ has m correct digits, the error bound in the normalized divisor being $|X_{m+1} - 1| \leq 2/3 \cdot 16^{-m}$.

In presenting the algorithm for division, as well as for the other operations, it is assumed that the normalization and the result evaluation are carried out in two similar arithmetic units. Later, when discussing implementation aspects, it will be shown how the proposed algorithms can be realized with essentially one arithmetic unit with a tolerable decrease in performance.

Algorithm D (Division)

(AU1: Normalization) (AU2: Result Evaluation)

Step D1. [Initialize] $k \leftarrow 0$;

Step N1 (Algorithm N); $Q_0 \leftarrow Y_0$;

Step D2. [Loop] for $k \leq m$ perform:

Step N2; $Q_{k+1} \leftarrow Q_k + Q_k S_k 16^{-k}$;

An example of the division is given in Figure 3-1. The "predictability" feature, described before (2.11), is apparent at step $k = 7$: the first five digits of R_8, when recoded, are the next five constants $S_8, \ldots, S_{12}$.

In Figure 3-2, the basic hardware configuration, consisting of two arithmetic units, is shown. The control part is not described. The only difference between the two arithmetic units is that AU1 has the additional network TU and the five-bit register S, used in the selection process.

XO= 0.70999997854232 YO= 0.59314718055994 YO/XO= 0.83541858941702

K	SK	RK+1 (IN HEXADECIMAL)	XK+1 (IN DECIMAL)	QK+1 (IN DECIMAL)
0	0	-.4A3D7100000238	0.70999997854232	0.59314718055994
1	7	.547AD8FFFFCD00	1.02062496915459	0.85264907205491
2	-5	.2D472C2FFCDFF0	1.00069088772578	0.83599576986634
3	-3	-.2C151284C1F79F	0.99995795982950	0.83538346827708
4	3	.3EA693C067A1B1	1.00000373427224	0.83542170909747
5	-4	-.1597BE93D4E68E	0.99999991956073	0.83541852221656
6	1	-.597BEA96CA521D	0.99999997916537	0.83541857201138
7	6	.6841547A735EA7	1.00000000151711	0.83541859068444
8	-7	-.7BEAB88666AA85	0.99999999988730	0.83541858932287
9	8	.41547795B601EB	1.00000000000371	0.83541858942012
10	-4	.154779584FC992	1.00000000000009	0.83541858941708
11	-1	.54779584FC83D8	1.00000000000002	0.83541858941704
12	-5	.47795B4FC8231A	1.00000000000000	0.83541858941702

Figure 3-1

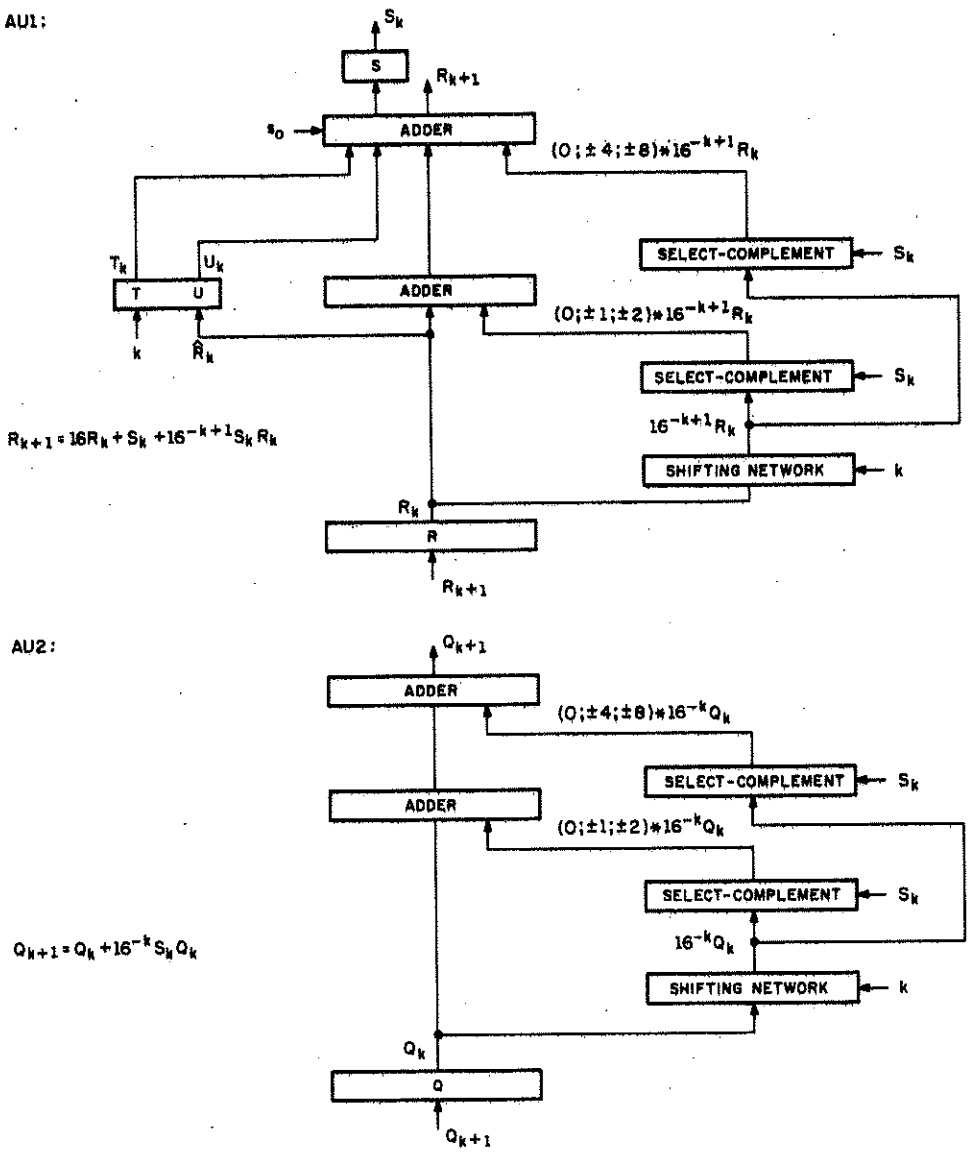
AU1:
S_k
S
R_{k+1}
s_o
ADDER
$(0;\pm 4;\pm 8)*16^{-k+1}R_k$
SELECT-COMPLEMENT
S_k
T_k
U_k
T
U
ADDER
$(0;\pm 1;\pm 2)*16^{-k+1}R_k$
k
$\hat{R}_k$
SELECT-COMPLEMENT
S_k
$16^{-k+1}R_k$
$R_{k+1} = 16R_k + S_k + 16^{-k+1}S_k R_k$
SHIFTING NETWORK
k
R_k
R
R_{k+1}
AU2:
Q_{k+1}
ADDER
$(0;\pm 4;\pm 8)*16^{-k}Q_k$
SELECT-COMPLEMENT
S_k
ADDER
$(0;\pm 1;\pm 2)*16^{-k}Q_k$
SELECT-COMPLEMENT
S_k
$16^{-k}Q_k$
$Q_{k+1} = Q_k + 16^{-k}S_k Q_k$
SHIFTING NETWORK
k
Q_k
Q
Q_{k+1}

Figure 3-2

4. NATURAL LOGARITHM

Let $X = X_0 \cdot 2^{E_x}$ be a given floating point number with fractional part $X_0 \in [1/2,1)$ and exponent E_x. In formulation of the algorithm in radix 16, we follow the same approach as for the radix 2 case, given in [1]. To obtain $\ell n\, X$, the problem is split in two parts. Namely,

$$\ell n\, X = \ell n\, X_0 + E_x\, \ell n\, 2 \tag{4.1}$$

The algorithm for calculation of the first term $\ell n\, X_0$ is derived from the identity:

$$X_0 = X_0 \prod_{i=0}^{m} M_i \Big/ \prod_{i=0}^{m} M_i \tag{4.2}$$

where multipliers are $M_k = 1 + S_k \cdot 16^{-k}$ and constants $S_k \in \{\overline{10}, \ldots, 0, \ldots, 10\}$, as defined before for the multiplicative normalization algorithm. Now,

$$\ell n\, X_0 = \ell n\, \left(X_0 \prod_{i=0}^{m} M_i\right) - \ell n\, \left(\prod_{i=0}^{m} M_i\right) \tag{4.3}$$

From the normalization algorithm N (2.26) we know that the error in normalization is bounded by

$$|X_{m+1} - 1| \leq 2/3 \cdot 16^{-m}$$

Therefore, $\ell n\, \left(X_0 \prod_{i=0}^{m} M_i\right) = 0$ for m digits precision and $\ell n\, X_0$ is reduced to

$$\ell n\, X_0 = -\ell n\, \left(\prod_{i=0}^{m} M_i\right) = \sum_{i=0}^{m} \left[-\ell n(1+S_i 16^{-i}\right] \tag{4.4}$$

To obtain $\ell n\, X_0$, one needs to perform the summation of m + 1 sets of precomputed constants of the form

$$- \ell n\, (1 + S_k 16^{-k})$$

stored in a fast read-only memory (ROM).

The stored constants are retrieved from the ROM using S_k's, obtained in normalizing X_0. It is clear that this procedure is equally well applicable to any logarithm function--just the set of stored constants need to be precomputed in the corresponding base. The summation (4.4) is performed recursively:

$$L_{k+1} = L_k + [- \ell n\, (1 + S_k \cdot 16^{-k})] \qquad \text{for } k = 0, \ldots, m$$

$$\text{where} \qquad L_0 = 0 \tag{4.5}$$

Then $L_{m+1} = \ell n\, X_0$. It should be noted that this result may not have an accuracy of m digits because the stored constant cannot be exactly represented with m digits and during m + 1 additions, errors will accumulate. If the result $\ell n\, X_0$ is to be correct to m digits (4m bits), i.e., with error less than $1/2 \cdot 2^{-4m}$, then the precision of the second arithmetic unit should be extended by

$$\Delta\, m = \lceil \ell n\, (m+1) / \ell n\, 2 \rceil$$

where $\lceil X \rceil$ is the smallest integer not smaller than X. For m = 12, this amounts to an extension of 4 bits. If the algorithm is performed in radix 2, to retain the same error bound, the extension will be 6 bits.

The calculation of the second term $E_x\, \ell n\, 2$ can be performed using conventional multiplication since E_x is always of short precision compared

to X_0. The constant $\ell n\, 2$ can be stored in a ROM. As an alternative, it might be convenient to implement $E_x \ell n\, 2$ using a multiplication algorithm based on continued sums, as described in Chapter 6. Assuming that the length of the exponent E_x is 8 bits, such a solution will increase the total time by 3 basic cycles but reduce the hardware requirements by an extra low precision multiplier. Namely, after $\ell n\, X_0$ has been computed in the second arithmetic unit (of Figure 4.2), this result is taken as the first partial product. Then the first arithmetic unit is initialized to E_x, the step counter set to zero and the multiplication $E_x \ell n\, 2$ is performed using recursion (6.2):

$$L_{k+1} = L_k + (\ell n\, 2)S_k \cdot 16^{-k} \qquad \text{for } k = 0,\ 1,\ 2$$

where

$$L_0 = L_{m+1} = \ell n\, X_0$$

and S_0, S_1, S_2 are obtained through the additive normalization algorithm. Therefore, the last partial product P_3 will represent the final result $\ell n\, X$. This approach has assimilated the extra add step, indicated in (4.1).

For higher radices the set of constants S_k is enlarged and consequently the capacity of the ROM must be increased. As in the radix 2 case [1], there is no need to store all of the constants $[-\ell n\,(1 + S_k \cdot 16^{-k})]$. The possible reduction can be shown using the power series expansion for the logarithm.

The constants to be stored are of the form

$$\ell n\,(1 + S_k \cdot 16^{-k}) = \ell n\,(1+a) \qquad \text{and}$$

$$S_k \in \{\overline{10},\ \ldots,\ 0,\ \ldots,\ 10\}$$

Then

$$\ell n\,(1+a) = a - \frac{1}{2}a^2 + \text{H.O.T.} \qquad \text{for } -1 < a \leq 1$$

For
$$k \geq k_1 = \frac{2\log_2 10 - 1 + 4m}{8} \approx \frac{5.5 + 4m}{8} \tag{4.6}$$

and m digits accuracy

$$\ell n\,(1 + S_k \cdot 16^{-k}) = S_k \cdot 16^{-k}$$

eliminating the need for storing more constants. Therefore, for radix 16 the necessary capacity of the ROM is at most 10m + 15 words of m digits (4m bits) each, including the constant $\ell n\,2$, used in the initial step as well as in evaluation of the term $E_x\,\ell n\,2$.

The algorithm for natural logarithm is given below:

Algorithm L (Logarithm) (4.7)

(AU1: Normalization) (AU2: Result Evaluation)

Step L1. [Initialize] $k \leftarrow 0$;

Step N1. (Algorithm N); $L_0 \leftarrow 0$;

Step L2. [Loop] for $k \leq m$ perform:

Step N2. if $k < k_1$ then:

$L_{k+1} \leftarrow L_k - \ell n\,(1 + S_k 16^{-k})$;

else:

$L_{k+1} \leftarrow L_k - S_k \cdot 16^{-k}$;

Step L3. [Form $\ell n\,X_0 + E_x\,\ell n\,2$]

Step A1. (Algorithm A); $(L_0 = L_{m+1})$;

- initialize: $R \leftarrow E_x$;

for $k \leq 2$ perform:

Step A2. $L_{k+1} \leftarrow L_k + (\ell n\,2)S_k 16^{-k}$;

where k_1 is given by (4.6), and algorithm A for additive normalization is given in (5.6). In the example, Figure 4-1, only the calculation of the $\ell n\, X_0$ part is shown. The additional requirement for implementation of the logarithm algorithm is a read-only memory (ROM), connected to the result evaluation unit (Figure 4-2).

XO= 0.59314718055994 LN(XO)= -0.52231271422032

K	SK	RK+1 (IN HEXADECIMAL)	XK+1 (IN DECIMAL)	LK+1 (IN DECIMAL)
0	1	.2FB0FCBC7068C0	1.18629436111988	-0.69314718055995
1	-2	.9BADD24E25BA80	1.03800756597990	-0.55961578793542
2	-9	.634B5E96666F18	1.00151511248841	-0.52382668008384
3	-6	.3262252EE08AE5	1.00004804933536	-0.52236076240134
4	-3	.2618E0870FE436	1.00000227076864	-0.52231498498638
5	-2	.618DBC3F3D3540	1.00000036341568	-0.52231307763593
6	-6	.18DB9F5EACBC49	1.00000000578768	-0.52231272000800
7	-2	-.72460A46EB7A2D	0.99999999833710	-0.52231271255742
8	7	-.2460A4A0B6474F	0.99999999996691	-0.52231271418723
9	2	-.460A4A0BAD3639	0.99999999999602	-0.52231271421634
10	4	-.6CA4A0BAE4E622	0.99999999999966	-0.52231271421998
11	6	-.A4A0BAE50A5FBC	1.00000000000000	-0.52231271422032
12	1	.5B5F451AF59F9F	1.00000000000000	-0.52231271422032

Figure 4-1

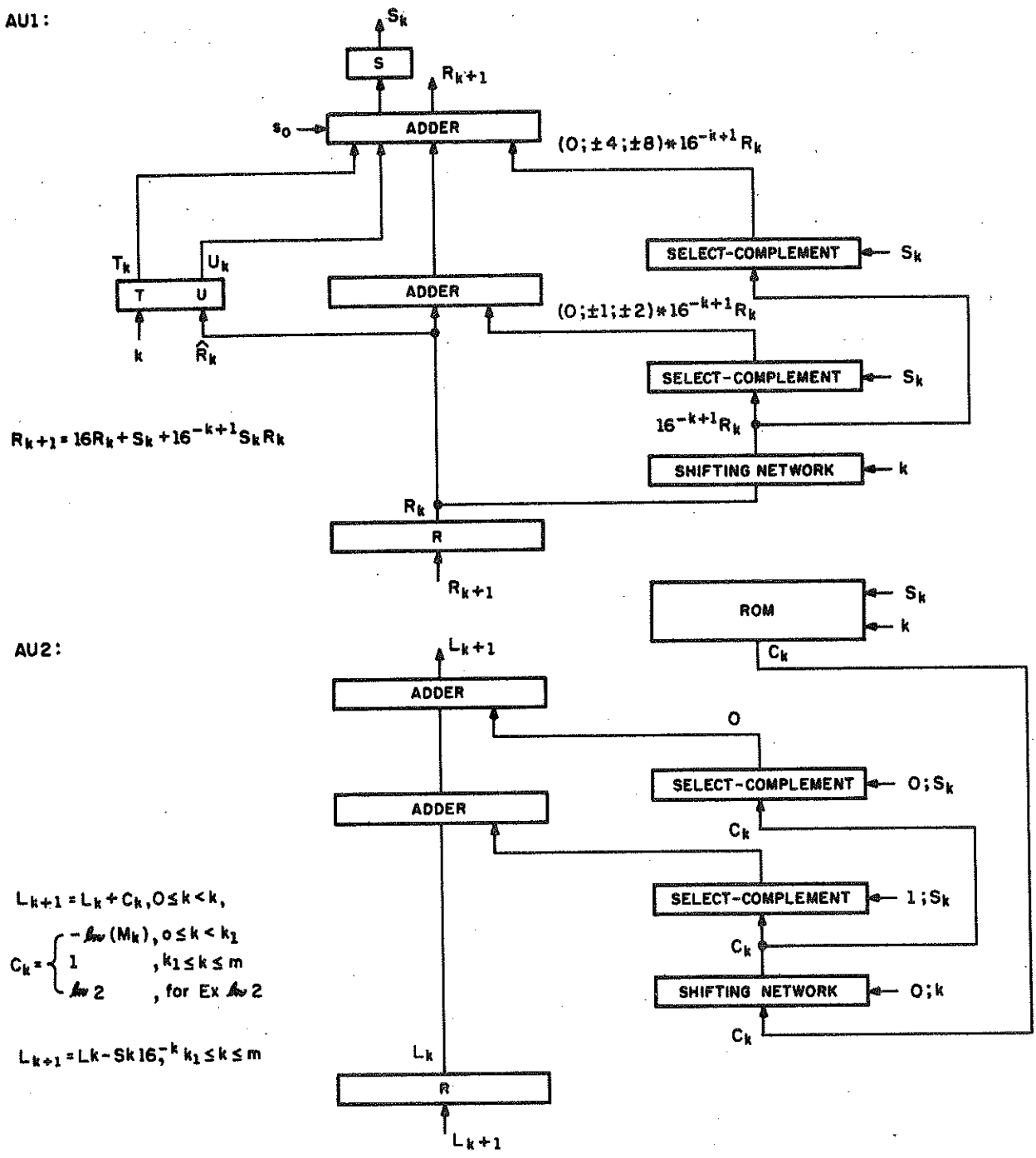
AU1:
S_k
S
R_{k+1}
s_0
ADDER
$(0;\pm4;\pm8)*16^{-k+1}R_k$
T_k
U_k
T
U
k
$\hat{R}_k$
ADDER
SELECT-COMPLEMENT
S_k
$(0;\pm1;\pm2)*16^{-k+1}R_k$
SELECT-COMPLEMENT
S_k
$16^{-k+1}R_k$
SHIFTING NETWORK
k
R_k
R
R_{k+1}
$R_{k+1} = 16R_k + S_k + 16^{-k+1}S_kR_k$
ROM
S_k
k
C_k
AU2:
L_{k+1}
ADDER
0
ADDER
SELECT-COMPLEMENT
$0;S_k$
C_k
SELECT-COMPLEMENT
$1;S_k$
C_k
SHIFTING NETWORK
$0;k$
C_k
L_k
R
L_{k+1}
$L_{k+1} = L_k + C_k, 0 \le k < k,$
$C_k = \begin{cases} -\mathit{fw}(M_k), & 0 \le k < k_1 \\ 1, & k_1 \le k \le m \\ \mathit{fw}\,2, & \text{for Ex } \mathit{fw}\,2 \end{cases}$
$L_{k+1} = Lk - Sk\,16^{-k}\; k_1 \le k \le m$

Figure 4-2

5. ADDITIVE NORMALIZATION

Algorithms for multiplication and the exponential, analogous to those defined for division and the logarithm can be derived on the basis of continued sums. The step by step process, in which a given number $X_0 \in [1/2,1)$ is normalized to zero by proper choice of constants S_k such that

$$X_0 - \sum_{i=0}^{m} S_i \cdot 16^{-i} \cong 0 \qquad (5.1)$$

where $$S_k \in \{\overline{10}, \ldots, 0, \ldots, 10\}$$

is clearly a right directed recoding of X_0. It is termed normalization as an additive counterpart of the previously described multiplicative normalization. The digits x_i from the non-redundant digit set $\{0, \ldots, 15\}$ are replaced, starting from the most significant, with the digits S_k belonging to the redundant digit set $\{\overline{10}, \ldots, 0, \ldots, 10\}$. This recoding is simple and exact. Namely, for every pair of digits $x_k\ x_{k+1}$, if $x_{k+1} \geq 10$, (maximal allowed value for S_{k+1}) then this pair is recoded as $S_k S_{k+1}$, where $S_k = x_k + 1$ and $S_{k+1} = -(15 - x_{k+1})$. Otherwise, $S_k = x_k$.

To define the selection rules, we proceed as before. First, the scaled remainder is defined as

$$R_k = 16^{k-1} X_k \qquad (5.2)$$

where $$X_k = X_0 - \sum_{i=0}^{k-1} S_i 16^{-i}$$

Then,

$$R_{k+1} = 16^k X_{k+1}$$

$$= 16^k (X_k - S_k \cdot 16^{-k})$$

$$= 16R_k - S_k, \qquad 0 \leq k \leq m \tag{5.3}$$

represents the basic recursion. This recursion corresponds exactly to the recursion (2.11), except for the sign of S_k. The scaled remainders are bounded, similarly as before:

$$|R_k| \leq 2/3 \tag{5.4}$$

and the selection rule is the same for all steps. The rule is simple: S_k equals the scaled remainder R_k, rounded to one non-sign digit, the sign being that of R_k. More precisely, the selection rule is:

$$S_k = \lfloor (T_k + U_k) \cdot 16 \rfloor \quad \text{and} \quad \text{Sign } S_k = \text{Sign } R_k \qquad 0 \leq k \leq m \tag{5.5}$$

where:

$$T_k = \sum_{i=1}^{5} r_i 2^{-i} \qquad \text{if} \quad R_k \geq 0;$$

$$T_k = \sum_{i=1}^{5} \bar{r}_i 2^{-i} \qquad \text{if} \quad R_k < 0 \quad \text{and}$$

$$R_k = -r_0 + \sum_{i=1}^{4m} r_i 2^{-i};$$

$$U_k = 1/32.$$

This choice of conventional rounding, i.e., $U_k = 1/32$ will actually restrict the set of constants S_k to $\{\bar{8}, \ldots, 0, \ldots, 8\}$. This choice is

preferred because it is simplest and the restricted set of S_k is sufficient for recoding. The choice of $U_k = 6/256$ would require a full set of constants S_k. Although the selection rule (5.5) holds for the initial step as well, it is more convenient to use the fact that for $|X_0| \geq 1/2$, as is the case, $|S_0|$ can always be taken to be 1. From the definition of the scaled remainder (5.2) it follows that it is preferable to start always with $|S_0| = 1$. Otherwise, for $k = 0$, to prevent loss of accuracy, an extension of one radix-16 digit would be necessary as well as an additional right shift path, needed only in this step. As indicated before, the result of additive normalization is always exact, i.e.,

$$X_0 - \sum_{i=0}^{m} S_i 16^{-i} = 0$$

For reference purposes, the additive normalization is summarized in the form of algorithm A, given below.

Algorithm A (Additive Normalization): (5.6)

Step A1. [Initialize] $k \leftarrow 0$;

$S_0 \leftarrow 1$;

$R_1 \leftarrow X_0 - S_0$;

Step A2. [Loop] for $k \leq m$ perform:

$k \leftarrow k + 1$;

$S_k \leftarrow [(T_k + U_k)16]$; Sign $S_k \leftarrow$ Sign R_k

$R_{k+1} \leftarrow 16R_k - S_k$;

where $U_k = 1/32$.

6. MULTIPLICATION

The algorithm for multiplication, given here, is based on a conventional procedure applied to the radix 16 case. Let the multiplicand and the multiplier be floating point numbers, satisfying the usual requirements, i.e.,

$$Y = Y_0 2^{E_Y} \qquad Y_0 \in [1/2, 1)$$

$$X = X_0 2^{E_X} \qquad X_0 \in [1/2, 1)$$

Again, only the multiplication of fractional parts is described, omitting straightforward exponent arithmetic as well as postnormalization of the result.

Consider

$$P = Y_0 X_0$$

$$= Y_0 \left[X_0 - \sum_{i=0}^{m} Z_i + \sum_{i=0}^{m} Z_i \right]$$

where m is the number of radix 16 digits in the fractional parts. If terms $Z_k = S_k \cdot 16^{-k}$ are properly chosen, then

$$X_0 - \sum_{i=0}^{m} S_i \cdot 16^{-i} = 0$$

and

$$P = Y_0 \sum_{i=0}^{m} S_i \cdot 16^{-i} \tag{6.1}$$

where the constants S_k, as before, are from the set $\{\overline{10}, \ldots, 0, \ldots, 10\}$. Applying additive normalization on X_0, the constants S_k can be obtained, as described by Algorithm A (5.6).

The summation of the partial products is performed simultaneously in the second arithmetic unit, using the following recursion:

$$P_{k+1} = P_k + Y_0 \cdot S_k \cdot 16^{-k}, \qquad 0 \leq k \leq m \tag{6.2}$$

where $P_0 = 0$.

Since the normalization of X_0 is exact, the only error in multiplication comes from the single precision result representation.

The algorithm for multiplication, compatible with other proposed algorithms, is given below.

Algorithm M (Multiplication): (6.3)

(AU1: Normalization) (AU2: Result Evaluation)

Step M1. [Initialize] $k \leftarrow 0$;

Step A1 (Algorithm A); $P_0 \leftarrow 0$;

Step M2. [Loop] for $k \leq m$ perform:

Step A2; $P_{k+1} \leftarrow P_k + Y_0 S_k 16^{-k}$;

An example is shown in Figure 6-1. For implementation, shown in Figure 6-2, an extra register to hold the multiplicand is needed.

XO= 0.70999997854232 YO= 0.59314718055994 YO*XO= 0.42113448547000

K	SK	RK+1 (IN HEXADECIMAL)	XK+1 (IN DECIMAL)	PK+1 (IN DECIMAL)
0	1	-.4A3D7100000238	-0.29000002145768	0.59314718055994
1	5	.5C28EFFFFFDC80	0.02249997854232	0.40778868663496
2	-6	-.3D710000023800	-0.00093752145768	0.42169057367933
3	4	.28EFFFFFDC8000	0.00003904104232	0.42111132838582
4	-3	-.71000002380000	-0.00000673532487	0.42113848050895
5	7	-.10000023800000	-0.00000005960465	0.42113452082433
6	1	-.23800000000000	-0.00000000000001	0.42113448547000
7	0	-.23800000000000	-0.00000000000001	0.42113448547000
8	0	-.23800000000000	-0.00000000000001	0.42113448547000
9	0	-.23800000000000	-0.00000000000001	0.42113448547000
10	0	-.23800000000000	-0.00000000000001	0.42113448547000
11	0	-.23800000000000	-0.00000000000001	0.42113448547000
12	2	-.38000000000000	-0.00000000000000	0.42113448547000

Figure 6-1

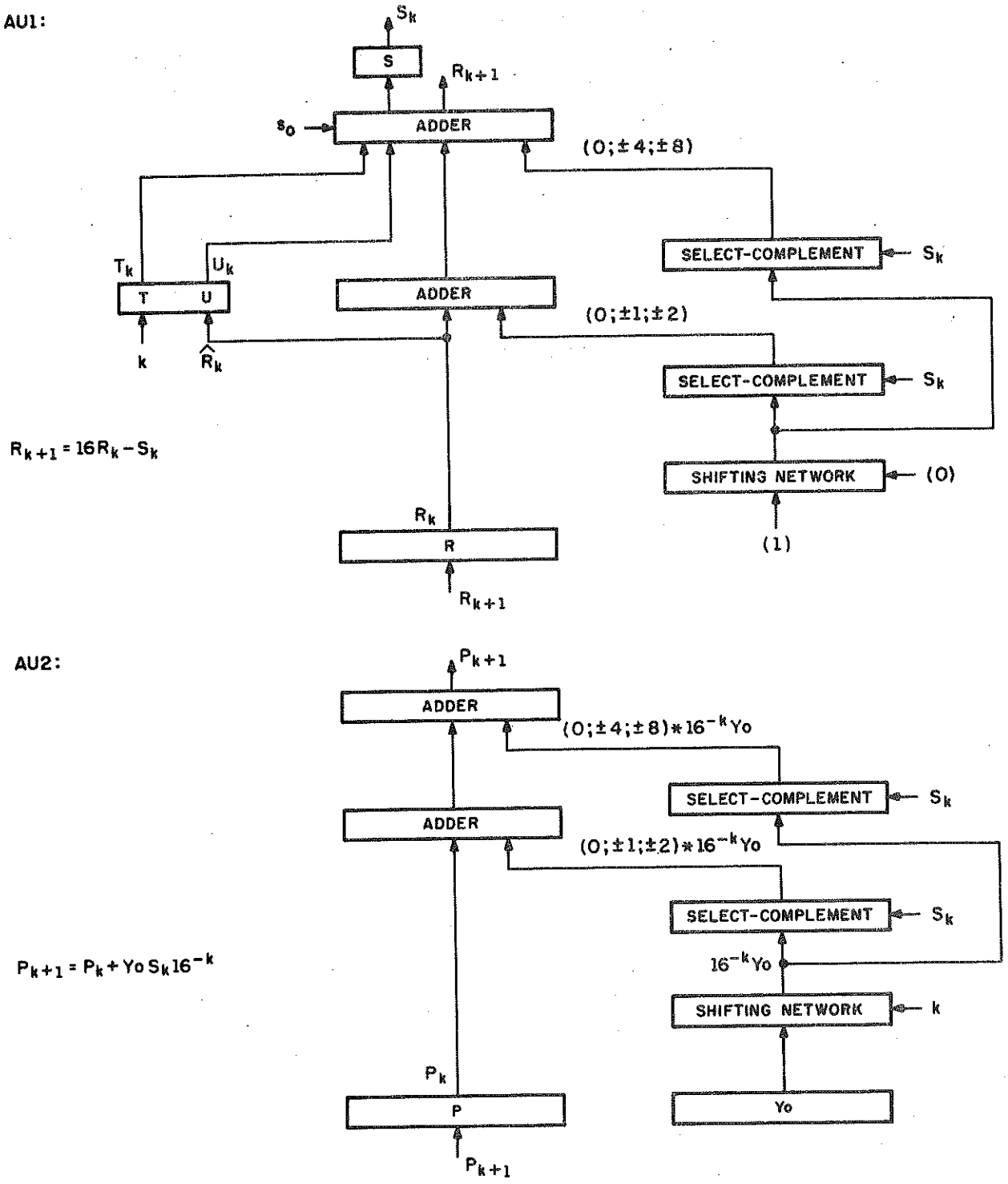
AU1:
S_k
S
R_{k+1}
s_o
ADDER
(0;±4;±8)
SELECT-COMPLEMENT
S_k
T_k
U_k
T
U
ADDER
(0;±1;±2)
k
$\hat{R}_k$
SELECT-COMPLEMENT
S_k
$R_{k+1} = 16R_k - S_k$
SHIFTING NETWORK
(0)
(1)
R_k
R
R_{k+1}
AU2:
P_{k+1}
ADDER
$(0;\pm4;\pm8)*16^{-k}Yo$
SELECT-COMPLEMENT
S_k
ADDER
$(0;\pm1;\pm2)*16^{-k}Yo$
SELECT-COMPLEMENT
S_k
$P_{k+1} = P_k + Yo\,S_k\,16^{-k}$
$16^{-k}Yo$
SHIFTING NETWORK
k
P_k
P
Yo
P_{k+1}

Figure 6-2

7. EXPONENTIAL

The following manipulation, as defined for radix 2 case in [1], applies without change to the radix 16, producing a convenient form of the exponential e^X. In the identity

$$e^X = e^{X \cdot \log_2 e \cdot \ln 2} \tag{7.1}$$

let

$$X \log_2 e = I + F$$

where I and F denote integer and fractional part, respectively.

Now

$$\begin{aligned} e^X &= e^{I \ln 2} e^{F \ln 2} \\ &= 2^I e^{F \ln 2} \end{aligned} \tag{7.2}$$

and defining X_0 as

$$X_0 = F \ln 2 \tag{7.3}$$

we obtain the result in the form

$$Y = Y_0 2^{E_Y} = e^X = 2^I e^{X_0} \tag{7.4}$$

Therefore the problem of finding e^X is substantially reduced to the problem of finding e^{X_0}, the factor 2^I being easily incorporated into the exponent part of the result E_Y. The exponent X is any number such that Y is in the

machine range. Therefore, $|F| < 1$, $|X| < \ell n\, 2$ yielding $e^{X_0} \in (1/2, 2)$. It is a simple detail of an actual design to obtain F bounded between -1 and 0, giving $e^{X_0} \in (1/2, 1]$, as usual. In the following discussion, we assume that

$$-1 < F \leq 0 \quad \text{or}$$

$$-\ell n\, 2 < X_0 \leq 0 \tag{7.5}$$

The described approach requires two extra multiplications before the main algorithm can begin. Namely, one multiplication is necessary to determine the terms I and F and another to obtain X_0.

The algorithm to evaluate e^{X_0}, described here, is similar to the other algorithms, both in derivation and in structure.

We start with the identity

$$e^{X_0} = e^{X_0 - \ell n \left(\prod_{i=0}^{m} M_i \right) + \ell n \left(\prod_{i=0}^{m} M_i \right)} \tag{7.6}$$

where $$M_k = 1 + S_k \cdot 16^{-k} \qquad 0 \leq k \leq m$$

and $$S_k \in \{\overline{10}, \ldots, 0, \ldots, 10\}$$

Once again, if constants S_k are selected properly, then

$$X_0 - \ell n \left[\prod_{i=0}^{m} (1 + S_i 16^{-i}) \right] \tilde{=} 0 \tag{7.7}$$

and the result is obtained as

$$e^{X_0} \tilde{=} e^{\ell n \left(\prod_{i=0}^{m} M_i \right)} = \prod_{i=0}^{m} (1 + S_i 16^{-i}) \tag{7.8}$$

i.e., in the form of a continued product. To define the selection procedure, we note first that

$$X_k = X_0 - \sum_{i=0}^{k-1} \ell n\,(1 + S_i 16^{-i}), \qquad 0 < k \leq m \tag{7.9}$$

Then

$$X_{k+1} = X_k - \ell n\,(1 + S_k 16^{-k}), \qquad 0 \leq k \leq m \tag{7.10}$$

and the scaled remainders, upon which the selection is performed, can be defined as

$$R_k = 16^{k-1} X_k, \qquad 0 < k \leq m \tag{7.11}$$

The basic recursion is

$$R_{k+1} = 16R_k - 16^k\, \ell n\,(1 + S_k 16^{-k}), \quad 0 < k \leq m \tag{7.12}$$

This recursion shows that again precomputed constants of the form $\ell n\,(1 + S_k 16^{-k})$ are necessary. Since these constants are the same ones used in the logarithm evaluation, all remarks about storage requirements and simplification apply here:

$$\text{- for } \quad k \geq k_1 \approx \frac{5.5 + 4m}{8}, \tag{4.6}$$

the logarithmic constants can be replaced with $S_k \cdot 16^{-k}$ reducing the basic recursion (7.12) to:

$$R_{k+1} = 16R_k - S_k, \qquad \text{for } k \geq k_1 \tag{7.13}$$

The last expression shows that for $k \geq k_1$ the selection process will be identical to one defined by the additive normalization (5.6). It would be, therefore, natural to try to find selection rules such that the similarity with additive normalization can also be satisfied for $k < k_1$.

We now consider rules for $k < k_1$ in reverse order. First we recall that the selection rules are determined by choosing appropriate boundaries between intervals in R_k corresponding to particular S_k. Next we assume that the five bit precision is sufficient, i.e., the boundaries between intervals can be represented as L/32, L being an integer. To find an interval in R_k corresponding to S_k, the bounds are determined as

$$R_k \in ((\underline{R}_{k+1} \cdot 16^{-1} + 16^{k-1} \ell n(1+S_k 16^{-k}),\ \overline{R}_{k+1} \cdot 16^{-1} + 16^{k-1} \ell n(1+S_k 16^{-k})) \qquad (7.14)$$

where $\underline{R}_{k+1}$ and $\overline{R}_{k+1}$ are minimal and maximal allowed values of R_{k+1}, respectively. From the power series expansion, we have

$$\ell n\ (1 + S_k 16^{-k}) = S_k 16^{-k} + \epsilon_k \qquad (7.15)$$

where

$$\epsilon_k = -\frac{1}{2}\ \frac{S_k^2}{16^{2k}} + \frac{1}{3}\ \frac{S_k^3}{16^{3k}} - \frac{1}{4}\ \frac{S_k^4}{16^{4k}} + \ldots\ , \qquad (7.16)$$

the condition of expansion clearly being satisfied.

Let

$$e_k = 16^k \cdot \epsilon_k \qquad (7.17)$$

then, to select S_k, R_k must be in the interval

$$R_k \in (\frac{1}{16}\ (\underline{R}_{k+1} + S_k + e_k),\ \ \frac{1}{16}\ (\overline{R}_{k+1} + S_k + e_k)) \qquad (7.18)$$

or, using the assumption of 5 bit precision

$$R_k \in (\frac{a_k}{32}\ ,\ \frac{b_k}{32}) \qquad (7.19)$$

where

$$a_k = \lceil 2(\underline{R}_{k+1} + S_k + e_k) \rceil$$

and for all S_k (7.20)

$$b_k = \lfloor 2(\overline{R}_{k+1} + S_k + e_k) \rfloor$$

where $\lceil x \rceil$ denotes the smallest integer not smaller than x, and $\lfloor x \rfloor$ denotes the largest integer not larger than x. Since $\underline{R}_k$ and $\overline{R}_k$ have asymptotic limits - 2/3 and 2/3, respectively, it follows that if $|e_k| < 1/6$, then a_k and b_k can be determined as

$$a_k = \lceil 2(\underline{R}_{k+1} + S_k) \rceil$$

and for all S_k (7.21)

$$b_k = \lfloor 2(\overline{R}_{k+1} + S_k) \rfloor$$

Clearly, $|e_k| < 1/6$ is a sufficient but not a necessary condition.

If the last expressions for a_k and b_k are valid then for selection of S_k the rounding of R_k to the most significant non-sign (radix 16) digit suffices, i.e., the selection process becomes the same as in additive normalization. Of course, once S_k is obtained, the next remainder is calculated using all terms in the basic recursion (7.13). By calculating e_k it turns out that for $k \geq 3$, $|e_k| < 1/6$ and hence we have simple selection rules as before.

For k = 2, it can be shown that it is possible to find intervals in R_2 for all S_2 except $S_2 = \overline{10}$, such that if $R_2 \in [(2S_2-1)/32, (2S_2+1)/32)$ then S_2 is the correct constant. Therefore, if the range of R_2 is restricted so that $S_2 = \overline{10}$ is excluded, again rounding can be used as a selection rule.

It has been found that this restriction in the possible range of R_2 does neither affect the selection process for k = 1 and k = 0 nor the representation of e^{X_0}.

For k = 1 intervals are determined, as before, using (7.14) and the results are given in Table 7.1.

Table 7.1

S_1	a_1	$\leq 32R_1 \leq$	b_1
10	15		$32\bar{R}_2$
9	14		15
8	12		14
7	11		12
6	9		11
5	8		10
4	6		8
3	5		6
2	3		5
1	1		3
0	-1		1
-1	-3		-1
-2	-5 (-11/2)		-3
-3	$32R_2$		-11/2

For all other values of S_1, i.e., for $\{\bar{10}, \ldots, \bar{4}\}$ intervals are not contiguous and hence those constants may not be used. In fact, if the possible range of R_2 is not restricted, $S_1 = \bar{4}$ can be included in the set of allowed constants in step 1. The actual selection rules for k = 1 can be specified as in multiplicative normalization, i.e., using conventions described by expressions (2.12 - 18) one can determine an additive constant U_1 and through modified rounding obtain S_1. Another choice, which is given here, is to restrict the range of R_1 so that conventional rounding applies. This restriction should not affect the possibility of representation of the required result. From Table 7.1, if $-.171 < R_1 < .212$ then $S_1 \epsilon \{\bar{2}, \bar{1}, 0, 1, 2, 3\}$

can be selected applying rounding to one non-sign digit precision and no special rules, differing from those for $k > 1$, are necessary.

To obtain R_1 in the desired range, the following initialization (step $k = 0$) can be devised. Since $X_0 \in (-\ell n\, 2,\ 0]$ by assumption and $R_1 = X_1 = X_0 - \ell n\, M_0$ the rules are:

Table 7.2

X_0	M_0	$\ell n\, M_0$	R_1
$[-1/8,\ 0]$	1	0	$[-1/8,\ 0]$
$[-3/8,\ -1/8)$	$e^{-1/4}$	$-1/4$	$[-1/8,\ 1/8)$
$(-\ell n\, 2,\ -3/8)$	$e^{-17/32}$	$-17/32$	$[-.162,\ .157)$

Since $e^{X_0} \in (1/2,\ 1]$, it can be easily shown that such a choice for the initialization as well as the restricted sets of constants in steps 1 and 2, i.e., $S_1 \in \{\overline{2},\ \ldots,\ 3\}$ and $S_2 \in \{\overline{9},\ \ldots,\ 10\}$ can give the correct continued product representation of the result.

A summary of the procedure for evaluation of e^X follows:

Preparatory Operations P (7.22)

Step P1. $N \leftarrow X \log_2 e$;

Step P2. if $X > 0$ then:

$I \leftarrow [N] + 1$;

else:

$I \leftarrow [N]$;

Step P3. $F \leftarrow N - I$;

$X_0 \leftarrow F\, \ell n\, 2$;

where [N] denotes the integer part of N; after preparatory operations, X_0 will be in the range $(-\ell n\, 2,\ 0]$, with corrected integer part I.

Algorithm E (Exponential) (7.23)

(AU1: Normalization) (AU2: Result Evaluation)

Step E1. [Initialize] $k \leftarrow 0$;

$R_1 \leftarrow X_0 - \ell n\, M_0$; $E_1 \leftarrow M_0$;

Step E2. [Loop] for $k \leq m$ perform:

$k \leftarrow k + 1$;

if $k < k_1$ then: $E_{k+1} \leftarrow E_k + E_k S_k 17^{-k}$;

$S_k \leftarrow \lfloor (T_k + U_k)16 \rfloor$;

Sign $S_k \leftarrow$ Sign R_k;

$R_{k+1} \leftarrow 16R_k - 16^k\, \ell n\, (1 + S_k 16^{-k})$;

else:

Step A2. (Algorithm A);

where k_1 is defined in (4.6), and $U_k = 1/32$. An example is shown in Figure 7-1. The read-only memory, for this algorithm, communicates with the normalization unit, Figure 7-2. The remaining configuration is the same as before.

XO= 0.59314718C55994 EXP(XO)= 1.80967483607191

X*LD(E)= 0.85573049591110 F= 0.85573049591110 I= 0

K	SK	RK+1 (IN HEXADECIMAL)	XK+1 (IN DECIMAL)	EK+1 (IN DECIMAL)
0	0	-.1999999999B1A	-0.10000000000001	1.00000000000000
1	2	.895836AB62DDD0	0.03353139262452	0.87500000000000
2	-9	-.42E959DBBEE2D0	-0.00102098888214	0.90576171875000
3	4	-.2E15886298CAC0	-0.00004394923432	0.90487718582153
4	3	.1EABF9DF7F6840	0.00000182818064	0.90483576383122
5	-2	-.15404208CC2690	-0.00000007916617	0.90483748966847
6	1	-.54042000C268B0	-0.00000001956153	0.90483743573596
7	5	-.4041FF44268AF0	-0.00000000093507	0.90483741888204
8	4	-.41FF44268AF000*16^{-1}	-0.00000000000375	0.90483741803935
9	0	-.41FF44268AF000	-0.00000000000375	0.90483741803935
10	4	-.1FF44268AF0000	-0.00000000000011	0.90483741803606
11	2	.8BD97510000000*16^{-2}	0.00000000000000	0.90483741803595
12	0	.8BD97510000000*16^{-1}	0.00000000000000	0.90483741803595

EXP(XO)= 1.80967483607191

Figure 7-1

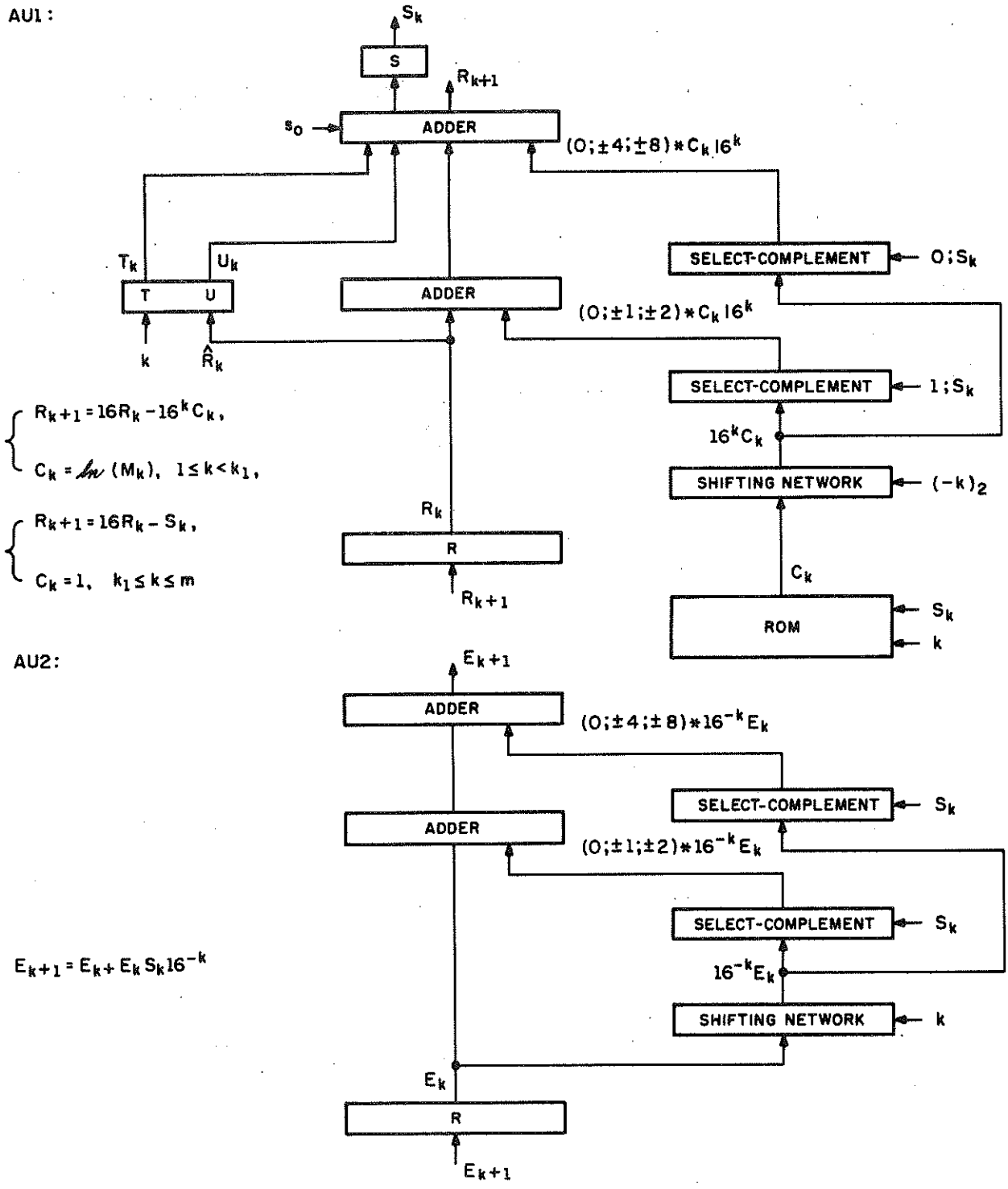
AU1:
S_k
S
R_{k+1}
s_0
ADDER
$(0;\pm4;\pm8)*C_k16^k$
T_k
U_k
T
U
k
$\hat{R}_k$
ADDER
$(0;\pm1;\pm2)*C_k16^k$
SELECT-COMPLEMENT
$0;S_k$
SELECT-COMPLEMENT
$1;S_k$
16^kC_k
SHIFTING NETWORK
$(-k)_2$
C_k
ROM
S_k
k
$R_{k+1}=16R_k-16^kC_k,$
$C_k=h(M_k),\ 1\le k<k_1,$
$R_{k+1}=16R_k-S_k,$
$C_k=1,\ k_1\le k\le m$
R_k
R
R_{k+1}
AU2:
E_{k+1}
ADDER
$(0;\pm4;\pm8)*16^{-k}E_k$
SELECT-COMPLEMENT
S_k
ADDER
$(0;\pm1;\pm2)*16^{-k}E_k$
SELECT-COMPLEMENT
S_k
$16^{-k}E_k$
SHIFTING NETWORK
k
$E_{k+1}=E_k+E_kS_k16^{-k}$
E_k
R
E_{k+1}

Figure 7-2

8. IMPLEMENTATION

One of the basic features, relevant for an efficient realization of the previously described algorithms, is that the original operation is replaced by the two much simpler processes of limited dependency. Through one process, the normalization, a sequence of constants S_k, the digits of the continued products (sums) are generated. Another process, the result evaluation, produces the final result using constants S_k. Both processes are defined recursively, requiring only simple hardware operations: addition, shift and multiple formation.

A realization, providing one separate arithmetic unit for each process, clearly offers the fastest solution and the simplest control requirements. Since both units are identical, as far as the main configuration is considered, a replication using an advanced technology should make this cost acceptable. If the speed is not of primary importance, one arithmetic unit can be used in both processes, performed in series. For simplicity, the processes should alternately use the arithmetic unit so that the current value of S_k need only be available. A "pipelining" of processes through one adder and shifting network, described at the end of this chapter, can achieve only 15%-25% slower operation than the double arithmetic unit realization, using essentially one arithmetic unit.

As mentioned in the Introduction, this investigation of the use of radix 16 in implementation of the described algorithms, has been motivated

by a possible speed improvement over the radix 2 approach and by some trade-offs in hardware requirements. In the following comparisons of the radix 2 [1] and radix 16 solutions, a two arithmetic unit realization is assumed in both cases. Furthermore, no actual design being done in either approach, given comparisons are approximate in nature and restricted only to the size-dominant parts. Control is assumed to be synchronous, each recursive step being performed in one basic cycle. The main parts of the arithmetic units, used in comparisons, are as follows.

a) The adder structure with the multiple formation networks, used in the radix 16 case, is estimated to be twice as complex as the corresponding part in the radix 2 case. Namely, in the former case, two adders and two (1 out 2) select and complement networks are required, while the later case requires one adder with one select-complement network. The speed of addition in the radix 16 case will be only slightly decreased if both adders are unified into one three-input adder. If the add time of a two-input adder (the radix 2 case) is t_{a2}, we estimate that $t_{a16} < 1.2t_{a2}$, for sufficiently large m.

b) The shifting network, required to shift right/left k-digits, for $0 \leq k \leq m - 1$, is simpler for a higher radix. We assume that the fast shifting network is realized using a "barrel switch" technique [8]. Namely, shifting is performed in two or more levels so that the combination of level shifts corresponds to the required total shift. This technique, besides being fast, ensures low loading requirements and the shifting can be done using same paths both ways: shift count is represented in two's complement, a negative number specifying left shift. Implemented in integrated

circuits technology easily with its regular and simple structure, the barrel switch as a standard block can be used also in some other operations, e.g., shifting, normalization, etc. Because of an additional level, the shifting network in the radix 2 case is estimated to require 30%-50% more hardware than radix 16 for $m = 48$ to 64 bits. For example, if $m = 48$, then level 1 may provide displacements of 0, 16 and 32 positions, level 2 provides then displacements of 0, 4, 8 and 12 positions, and in the radix 2 case, an additional level 3 would be necessary with the displacements 0, 1, 2 and 3 positions. Speedwise, then, $t_{sh2} > 1.3t_{sh16}$, where t_{sh} denotes shifting delay.

c) The selection procedure in the radix 2 case requires implementation of a simple 4 bit comparison. For the radix 16 approach, the required precision for selection is 7 bits and the five Boolean equations (2.23), costing less than 40 literals, are to be implemented. As described before, the selection is performed using rounding, so the additional inputs to the 7 most significant positions of the adder should be provided as well as the 5 bit register S to store the current value of the constant S_k. Even with those requirements, the selection hardware size is small compared with the rest of the unit. In the radix 2 case, this is even more true, so the selection hardware requirements are neglected in both cases.

d) For m bits precision, the number of precomputed logarithmic constants, stored in the read-only memory (ROM) is about m, in the radix 2 case, and about 3m, in the radix 16 case.

e) The control part, which includes also the step counter (two bits shorter in the radix 16 case) is not considered as being highly dependent on a particular realization technique.

From the above considerations, the hardware requirement ratio for the radix 2 and the radix 16 is approximately 2:3. The basic cycle can be taken to be the same, since the add time is dominant over control, selection and shifting time. The ROM capacity requirement ratio is about 1:3 in favor of the binary case.

Let the performance of an implementation in the radix r be $P_r = \log_2 r/Tr$, where Tr is the total delay necessary to evaluate $\log_2 r$ bits of the result, as defined in [4]. Tr is equivalent to the basic cycle. In the radix 2 case, the probability of $S_k = 0$ ($p_0 = 2/3$) is utilized by providing an adder bypass and reducing the number of full basic cycles to m/3 on the average, where m is number of bits. Then, it can be taken that the radix 16 basic cycle is $T_{16} \cong 3T_2$, since the number of basic cycles in the radix 16 case is always the same, the probability of $S_k = 0$ being too low. Then, the ratio of performances is $P_{16}/P_2 \cong 4/3$ on the average. If the efficiency of the implementation is defined as the ratio between performance and cost per bit, then, with all previous assumptions, $E_{16}/E_2 \cong 1$ without considering ROM requirements. If the ROM capacity requirement is taken into account then the radix 2 approach will offer more efficient design, but the radix 16 case will maintain better performance with shorter execution time. The selection procedure for radix 16 has been shown to be sufficiently simple.

Even with the available efficient technological solutions, the use of two arithmetic units may be objectionable. Since both the process of normalization and the process of result evaluation have addition as the basic operation, a "pipelined" use of the same adder would be possible,

provided proper latching of the operands and the results is made. One way to achieve this is shown in Figure 8-1. The adder with multiple formation networks is split into two equal parts AS" and AS' by breaking the carry path and inserting a one-bit carry register C. Outputs from the left half SN" of the shifting network are to be saved in a latch L. The selection is carried out in the block S on the basis of adder outputs and returns the value of S_k. The initial operands are in register B, for normalization, and in register A, for result evaluation. Each register contains two separately controlled halves, B", B' and A", A'. The outputs from AS" and AS' are connected, under a separate control, to the inputs of A" and A' registers, respectively. One separate path a from A" to SN" must be provided. The operation of this scheme is described for the division algorithm, with the help of Figure 8-2, with the initial control details omitted.

The normalization process requires realization of the recursion $R_{k+1} = 16R_k + S_k + 16^{-k+1}S_kR_k$ while the result is evaluated as $Q_{k+1} = Q_k + 16^{-k}S_kQ_k$. Since the operand $16R_k$ in the first equation corresponds to the operand Q_k in the second equation, additional path b from B to AS must be provided as well as one 4-bit register not shown in the scheme, to save the most significant digit of the left half of R_k.

The operation begins with the divisor X_0 in the B register and the dividend in the A register. Corresponding to the scheme, the superscripts ' and " denote the right and the left half of each result. The basic cycle now contains two periods, each period terminated with the clock pulse. The time of the period corresponds to approximately one half of the full length addition time. The registers are assumed to be of master-slave type. In the first period, R_1' is obtained and then, simultaneously, Q_0' from A' is transferred to B_1', R_1' from AS' to A' and the generated carry bit is saved

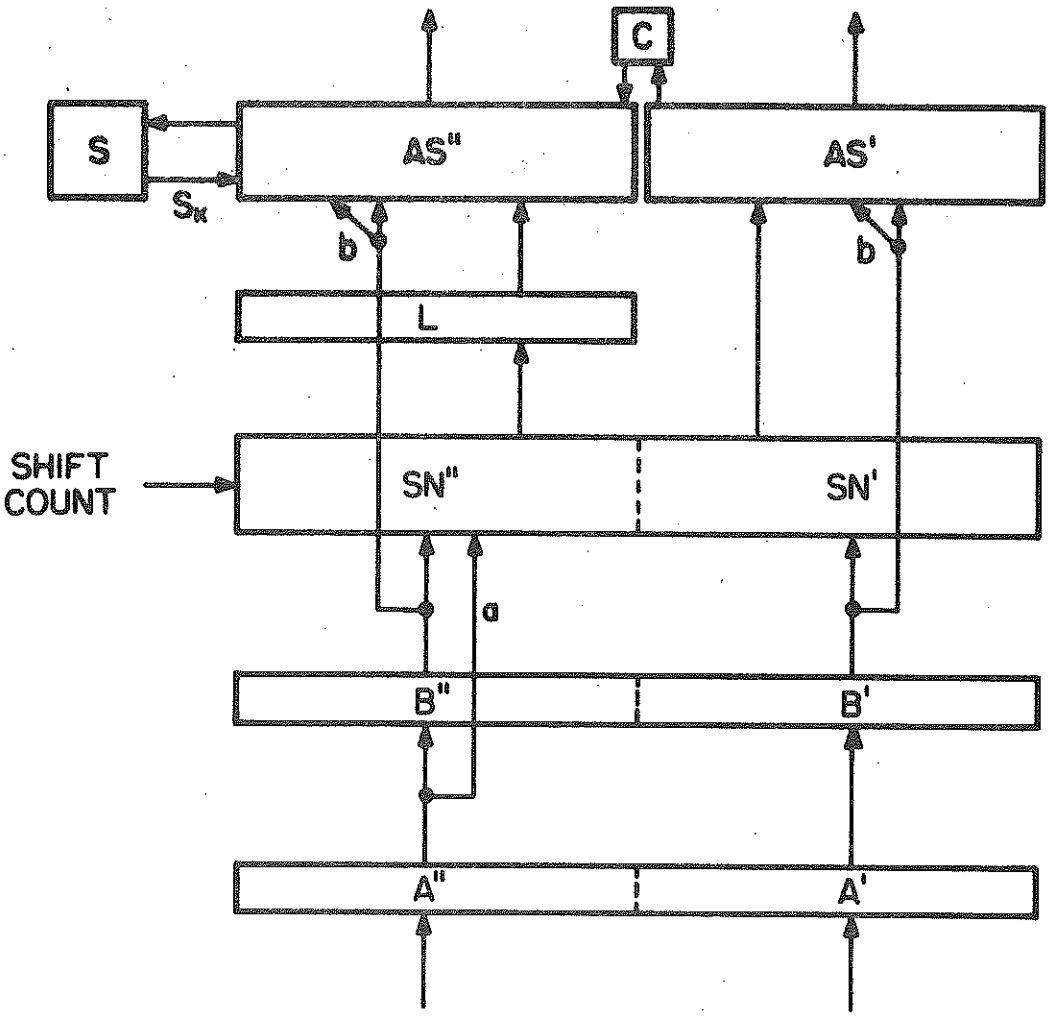

Figure 8-1

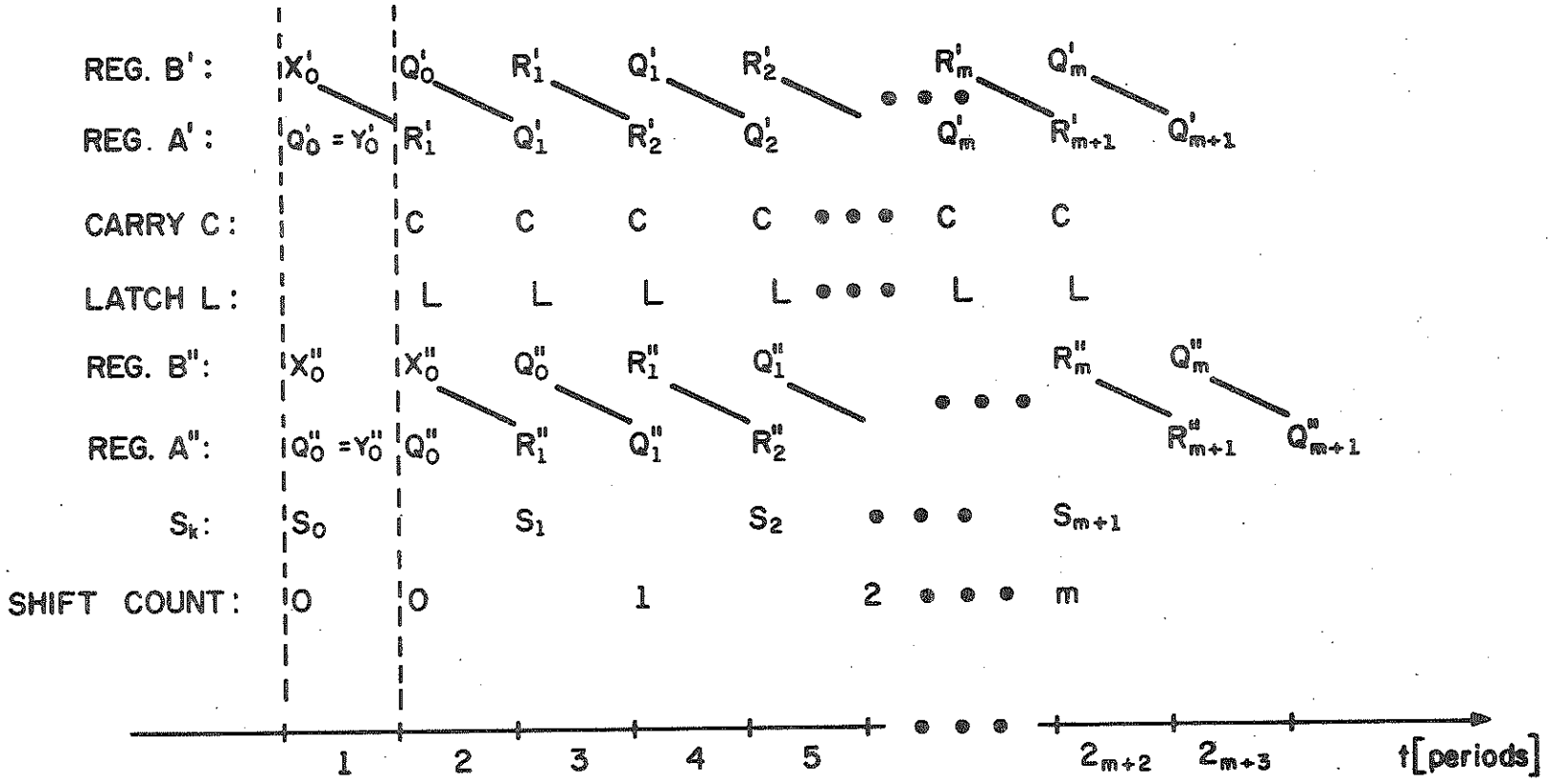
REG. B':
REG. A':
CARRY C:
LATCH L:
REG. B":
REG. A":
S_k:
SHIFT COUNT:
X'_0
Q'_0
R'_1
Q'_1
R'_2
R'_m
Q'_m
$Q'_0 = Y'_0$
Q'_2
R'_{m+1}
Q'_{m+1}
C
L
X''_0
Q''_0
R''_1
Q''_1
R''_2
R''_m
Q''_m
$Q''_0 = Y''_0$
R''_{m+1}
Q''_{m+1}
S_0
S_1
S_2
S_{m+1}
0
1
2
m
1
2
3
4
5
2_{m+2}
2_{m+3}
t[periods]

Figure 8-2

in C. During this period, operation on the left half is inhibited. In the second period, the right half evaluates Q_1' while the left half finishes calculation of R_1 by obtaining R_1''. The latch L and the path a provide for correct sharing of the shifting network. Once R_1'' is obtained, S_1 can be determined, the register transfer, now on both halves, is done and the process repeats. Shift count is changed when Q_k'' is obtained, matching requirements of both recursions. Path b is selected whenever R_k is being calculated. After $2m + 3$ periods, where m is the number of radix 16 digits, the result $Q = Q_{m+1}$ is obtained in the A register. It is reasonable to estimate that the period will take 0.55-0.6 of the basic cycle so that the total time of operation in the pipelined mode will be increased by 15% to 25%. Since the major blocks, adder with multiple formation networks and shifting network, are reduced from two to one, and since the new data paths, latches and the extra control are still significantly less complex than the major blocks, the solution might be optimal.

Without going into detailed description, we mention that a pipelined implementation is also possible in radix 2. With the assumption that $(t_{shift} + t_{select}) << t_{add}$, the total number of full cycles will be about M/3 on the average, where M is the number of bits, if bypassing of the adder is performed whenever $S_k = 0$. In a pipelined version, analogous to the one previously described for radix 16, after S_k is determined and before the next period starts, the full scaled remainder is in the register A. If the next S_k is zero, the transfers between registers are inhibited and only one-bit left shift on the register A is performed, making selection of the next S_k possible. When S_k becomes non-zero, the normal operation is resumed. Therefore, the average number of full cycles can be preserved in a pipelined version.

9. CONCLUSIONS

A radix 16 approach for implementation of the algorithms based on the continued products (sums), as proposed by DeLugish [1] in the binary case, has been studied. Those algorithms, in general, offer simplicity in mechanization and uniformity in hardware requirements for a wide class of elementary functions. The use of a higher radix makes the execution of the algorithms faster, but additional complexity, both in the selection procedures and hardware, must be considered. For the radix 16 case, it has been found that the selection rules remain relatively simple. Namely, the starting difficulties disappear after the first three steps, making the rules very simple. Furthermore, after performing the initial steps, an increasing number of constants S_k is simultaneously available at each successive step. This property can be utilized to simplify normalization or to define a variable radix method, provided fast and cheap multi-input adder arrays are available. Such a method would restore fast convergence of the algorithms, which is, in some sense, lost by specifying algorithms in a step by step mode using a fixed radix. Hardware requirements for implementation of algorithms based on continued products (sums) are, even for the radix 2, greater than those of conventional arithmetic units but still not prohibitive. A fast variable shift network, not commonly found in conventional arithmetic units, is an essential part for the proposed algorithms, but can be used advantageously in many other operations, like floating point normalization, conversion between floating point and fixed point number representations, shifting, etc.

Only division, multiplication, logarithm and exponential have been presently considered. Whether square root, trigonometric and inverse trigonometric functions can be easily included in the radix 16 approach, remains to be decided by finding corresponding selection rules. It is believed that this is possible.

Since the basic hardware is used for implementation of many algorithms, even a design with two arithmetic units would be acceptable. The outlined "pipeline" solution makes the entire approach more attractive, since the most complex parts, like the adder structure with the multiple formation network and shifting network, are shared by both processes. This solution illustrates also a possible general approach in defining arithmetic algorithms: a difficult operation is decomposed into a set of simple processes with such interdependencies that the overlapping of their execution is feasible. In this particular case, the normalization and the result evaluation are two such processes.

LIST OF REFERENCES

[1] B. G. DeLugish, "A class of algorithms for automatic evaluation of certain elementary functions in a binary computer," Report No. 399, Department of Computer Science, University of Illinois, Urbana, June 1970.

[2] J. E. Robertson, "A new class of digital division methods," IRE Transactions on Electronic Computers, vol. EC-7, pp. 218-222, September, 1958.

[3] _____, Lecture Notes for Computer Science Courses 394 (Fall 1970) and 482 (Spring 1971), Department of Computer Science, University of Illinois, Urbana.

[4] D. E. Atkins, "A study of methods for selection of quotient digits during digital division," Report No. 397, Department of Computer Science, University of Illinois, Urbana, June 1970.

[5] J. E. Volder, "The CORDIC trigonometric computing technique," IEEE Transactions on Electronic Computers, vol. EC-8, No. 5, pp. 330-334, September 1959.

[6] J. S. Walther, "A unified algorithm for elementary functions," AFIPS Conf. Proc., vol. 38, pp. 379-385, Spring Joint Computer Conference 1971.

[7] W. H. Specker, "A class of algorithms for LnX, ExpX, SinX, CosX, $Tan^{-1}x$, and $Cot^{-1}X$," IEEE Transactions on Electronic Computers, vol. EC-14, No. 1, pp. 85-86, February 1965.

[8] R. L. Davis, "The ILLIAC IV processing element," IEEE Transactions on Computers, vol. C-18, No. 9, pp. 800-816, September 1969.

APPENDIX

A. Derivation of equations $u_i = f(\hat{R}_1)$

We defined $U_1 = \sum_{i=1}^{6} u_i 2^{-i}$. From the Table 2.3 it can be observed that

$$u_i = f_i(r_j) \qquad \begin{matrix} i = 1, \ldots, 6 \\ j = 0, \ldots, 4 \end{matrix}$$

Using minterm notation we obtain:

- for $r_0 = 1$:

"don't care" minterms are $m_0, m_1, \ldots, m_9$

$$\begin{aligned} u_1 &= u_2 = 0 \\ u_3 &= m_{10} + m_{11} = \bar{r}_2 \\ u_4 &= m_{10} + m_{12} = (\bar{r}_2 + \bar{r}_3)\bar{r}_4 \\ u_5 &= r_0 \\ u_6 &= m_{13} = \bar{r}_3 r_4 \end{aligned} \qquad \text{(A1.1)}$$

- for $r_0 = 0$:

"don't care" minterms are $m_4, m_5, \ldots, m_{15}$.

$$\begin{aligned} u_1 &= u_2 = u_3 = u_4 = 0 \\ u_5 &= m_0 = \bar{r}_3 \bar{r}_4 \\ u_6 &= m_1 = \bar{r}_3 r_4 \end{aligned} \qquad \text{(A1.2)}$$

Combining (A1.1) and (A1.2), equations for step k = 1 are:

$$\begin{aligned} u_1 &= u_2 = 0 \\ u_3 &= r_0 \cdot \bar{r}_2 \\ u_4 &= r_0 \cdot \bar{r}_4 (\bar{r}_2 + \bar{r}_3) \\ u_5 &= r_0 + \bar{r}_3 \bar{r}_4 \\ u_6 &= \bar{r}_3 r_4 \end{aligned} \qquad \text{(A1.3)}$$

B. Derivation of equations $u_i = f(\hat{R}_2)$.

- for $r_0 = 1$:

"don't care" minterm are $m_0, m_1, \ldots, m_4$

$$\begin{aligned} u_1 &= u_2 = u_3 = u_4 = 0 \\ u_5 &= r_0 \\ u_6 &= m_5 + m_6 + m_7 + m_8 + m_9 + m_{10} = \bar{r}_1 + \bar{r}_2 (\bar{r}_3 + \bar{r}_4) \end{aligned} \qquad \text{(A1.4)}$$

- for $r_0 = 0$:

"don't care" minterms are $m_{11}, m_{12}, \ldots, m_{15}$

$$\begin{aligned} u_1 &= u_2 = u_3 = u_4 = 0 \\ u_5 &= m_0 + m_1 + m_2 + m_3 + m_4 + m_5 = \bar{r}_1 (\bar{r}_2 + \bar{r}_3) \\ u_6 &= m_6 + m_7 + m_8 + m_9 + m_{10} = r_1 + r_2 r_3 \end{aligned} \qquad \text{(A1.5)}$$

but u_5 and u_6 can be given also as:

$$\begin{aligned} u_5 &= \bar{r}_1 (\bar{r}_2 + \bar{r}_3) + r_6 \\ u_6 &= 0 \end{aligned}$$

according to remarks given after Table 2.4.

Therefore, for step k = 2

$$\begin{aligned} u_1 &= u_2 = u_3 = u_4 = 0 \\ u_5 &= r_0 + \bar{r}_1 (\bar{r}_2 + \bar{r}_3) + r_6 \\ u_6 &= r_0 (r_1 + r_2 r_3) \end{aligned} \qquad \text{(A1.6)}$$