Function Evaluation by Table Look-up and Addition

Hannes Hassler and Naofumi Takagi

Dept.of Information Engineering
Nagoya University
Nagoya , Japan 464-01

Abstract

We describe a general approach decomposing a function into a sum of functions, each with a smaller input size than the original. Hence we can map such functions with essentially the same precision using small ROM tables and adders. We derive an easy method to compute the worst case error for many elementary functions and an error bound for the rest. Important applications are reciprocals, logarithms, exponentials and others.

Index Terms – reciprocal, logarithm, exponential, table look-up, function decomposition

1 Introduction

There are increasing needs in evaluating elementary functions such as reciprocals, square roots, logarithms, exponentials and trigonometric functions quickly. In evaluating a function $f(X)$, if X has a small number of values, we can preprocess the values of $f(X)$ for all X and store them in a table. Such an approach is limited by the table size available.

However, for elementary functions, we have found that instead of one large table we can use several smaller tables and add their outputs. The input X is split into two or more overlapping subsets X_k's. Each X_k has fewer bits than the original X. The X_k's are the inputs for the smaller tables. The outputs of the tables are added together yielding $f(X)$. In fact this only yields an approximation of $f(X)$, but the error matches the precision bound. The proposed method is efficient for DSP operations in which rather short word length (e.g. 16–24 bits) is requested and in initial approximations for converging methods such as Newton-Raphson to compute elementary functions with larger bits.

To explain this task by a practical example, think of a logarithm table book. Assume it allows the user to look up the logarithm of some number x, where x has 6 decimal digits. Hence it must contain $1,000,000$ entries, a pretty substantial table. Following our approach, we can take the first four digits of x, look it up in a small table; then take the first two and the last two digits and look it up in another small table. The sum of the two table entries provides the logarithm. The table size is largely reduced. Instead of one large table we have only two small tables with $20,000$ entries all together, an improvement by a factor of 50. To illustrate the task by a numerical example consider the computation of $\log_{10} .654321 \approx -.184209$. Our approach suggests to compute $\log_{10} .6543 + \log_{10} .650021 - \log_{10} .65 \approx -.184209$ instead. We can look up the first logarithm in the first table (only the first four digits are needed) and the sum of the other two logarithms in the second table (only the first two and last two digits are needed).

Table look up algorithms are used, for instance, to compute initial solutions for reciprocal and division. DasSarma and Matula [3] give a good overview. Range reduction has already been discussed in other papers [4], though with rather different approaches. We are not aware of any algorithm similar to our approach. The theory behind our method uses so called *Partial Product Arrays* (PPA). This method was introduced by Stefanelli [7] in the early 70's. Stefanelli developed PPA's to perform binary division. Later, others have used PPA's to compute functions like square root, logarithm, exponential and trigonometric functions [1, 2, 5, 6].

A PPA is a symbolic representation of a function by the input bits. If we create the PPA of some function $f(X)$ we get a large number of bit products. Added together at their proper places of significance they generate $f(X)$. It is possible to create the PPA for any of the discussed functions up to some degree of accuracy.

The main question on the proposed method is the error caused by splitting the input bits into several smaller inputs. The model allows us to derive upper bounds on this error for arbitrary splittings. For certain functions, like reciprocal, logarithm and exponential we even gain sharp error bounds and can determine the worst case error exactly.

In Section 2 we define PPA's and make some observations. This allows us, in Section 3, to derive an approximation formula and to compute the worst case error. Section 4 gives an outline of some applications of the method presented. Limitations and open questions are discussed in Section 5.

2 Partial Product Arrays

We evaluate functions that can be developed into converging series $f(X) = \sum_{i=0}^{\infty} a_i X^i$ up to some precision. For our aim, the number of bits of the parameter

X is crucial, because more input bits require larger tables. The facts that the a_i's have limited precision as well and that we talk about infinite series are not so important now. Of course, we approximate infinite series by finite ones. Let us consider parameter X and its powers more closely.

X is a nonnegative fixed point number and consists of a string of bits, $X = [b_1 \cdots b_n]$. (For certain cases we will assume that the radix point is such that $X = .b_1 \cdots b_n$) The series can be written as:

$$f([b_1 \cdots b_n]) = \sum_{i=0}^{\infty} a_i [b_1 \cdots b_n]^i \qquad (1)$$

In the subsequent discussion we assume that the a_i's are all nonnegative. We call such functions a_i-*nonnegative functions*. General functions which have both negative and positive coefficients, can be separated into positive and negative parts: $f(X) = f_1(X) - f_2(X)$. Each f_i is an a_i-nonnegative function. Note that the functions $1/(1-X)$, $-\log(1-X)$ and 2^X are a_i-nonnegative. (We later define the first two functions only for $0 \leq X < 1/2$.) We compute the function $f(X)$ in a bitwise and symbolic way. All powers of X are expressed in bit notation.

Example 1

Consider a symbolic two-bit number $X = [.b_1 b_2] \hat{=} b_1 2^{-1} + b_2 2^{-2}$. Squaring X yields $X^2 = b_1 2^{-2} + b_1 b_2 2^{-2} + b_2 2^{-4}$. (Note that we write b_i instead of b_i^2 because we are dealing with single bits.) The cube of X is then, after simplification, $X^3 = b_1 b_2 2^{-2} + b_1 2^{-3} + b_1 b_2 2^{-5} + b_2 2^{-6}$. Multiplying by some constant a_i spreads these bit products further to different places of significance; for instance $[.101] \times b_1 2^{-2}$ becomes $b_1 2^{-3} + b_1 2^{-5}$. Consider a small example for a symbolic representation of a function. Take $f_{ex}(X) = X + X^2 + X^3$. $f_{ex}(X)$ is then $b_1 b_2 2^{-1} + b_1 2^{-1} + b_1 2^{-2} + b_2 2^{-2} + b_1 2^{-3} + b_2 2^{-4} + b_1 b_2 2^{-5} + b_2 2^{-6}$.

We get symbolic bit products weighted by some power of two. This simple example serves as an illustration for a definition of a positive PPA.

Definition 1 *A positive PPA of an a_i-nonnegative function $f([b_1 \cdots b_n])$ is the representation of the function as a sum of nonnegative bit products, i.e. $f([b_1 \cdots b_n]) = \sum_{j=1}^{S} B_j 2^{r_j}$, where each nonnegative bit product $B_j = b_{j1} \times \cdots \times b_{jk}$ and the bits of each product are a subset of the input bits; $\{b_{j1}, \cdots, b_{jk}\} \subseteq \{b_1, \cdots, b_n\}$.*

Each a_i-nonnegative function can be developed into a positive PPA. Our definition of PPA's is a very basic one. Compare, for instance, with Schwarz [6]. He uses more general bit products, allowing signed products and logic complements of single bits. His goal is to derive a hardware design from a PPA. Our simplification rules are very simple, there is only one:

$$B2^r + B2^r \rightarrow B2^{r+1} \qquad (2)$$

Generally the symbolic representation of $f(X)$ contains all possible bit products out of X, all of which are nonnegative. The PPA of $f([1 \cdots 1])$ (all bits 1) contains, for all possible bit products, a weighted *1* accordingly (i.e. none of them is made zero). We observe $0 \leq f(X) \leq f([1 \cdots 1])$.

Let us consider for instance the development of the reciprocal function into its PPA. We compute $1/(1-X)$ and use the equivalence $1/(1-X) = \sum_{i=0}^{\infty} X^i$ to compute all bit products. For this case it makes sense to assume $0 \leq X < 1/2$, where $X = [.0 b_2 b_3 \cdots b_n] \hat{=} b_2 2^{-2} + b_3 2^{-3} + \cdots + b_n 2^{-n}$ and we can proceed as in Example 1. We can calculate a finite sum $\sum_{i=0}^{L} X^i$ such that the error matches the precision bound 2^{-n}. We get $1/(1-X) = 1 + b_2 2^{-2} + b_3 2^{-3} \cdots$, see equation (3).

$1/(1 - .0 b_2 b_3 \cdots) = q_0 . q_1 q_2 \cdots$

2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	
1		b_2	b_3	b_2	$b_2 b_3$	$\cdots$
+				$b_2 b_3$	$b_2 b_4$	$\cdots$
+				b_4	$b_2 b_3 b_4$	$\cdots$
+					b_5	$\cdots$
						$\ddots$
q_0.	q_1	q_2	q_3	q_4	q_5	$\cdots$

(3)

The simplification rule (2) implies that, finally, there are no two equal bit products with equal weight. Hence the simplified PPA of $f(X)$ can be interpreted as a set. We denote it by $P(f(X))$. (Note that we use X both to refer to the input of function f and to the set of symbolic input bits, $\{b_1, \cdots, b_n\}$.) $P(f(X))$ is then a set of bit products times the appropriate power of two, and can be regarded as a subset of $2^X \times \{1, \cdots, n\}$ (Note that such a set definition covers functions up to a precision of n bits. In theory, of course, $n = \infty$.) For instance, in Example 1 we can derive the set $P(f_{ex}(X)) = \{\langle\{b_1, b_2\}, 1\rangle, \langle\{b_1\}, 1\rangle, \langle\{b_1\}, 2\rangle, \cdots\}$ from $f_{ex}(X)$.

Assume certain input bits are constantly zero. All symbolic bit products of the PPA which contain these zero-bits will disappear, too. The *set of variable bits* becomes smaller, only a subset X_1 of X, the original symbolic input bits is used. It is then obvious that also $P(f(X_1)) \subset P(f(X))$, i.e. $f(X_1)$ generates only a subset of all bit products of $f(X)$. Look at the PPA in equation (3) above. If we use the input subset $X_1 = \{b_2, b_4\} \subset X = \{b_2, b_3, b_4, \cdots\}$ we calculate $f(X_1) = 1/(1 - [.0 b_2 0 b_4 0])$; there remain the products which contain only b_2 and b_4.

We call a function $g(X)$ where $P(g(X)) \subset P(f(X))$ an approximation of $f(X)$. If we unite such approximation sets we get a set describing another (possibly better) approximation. In the following proposition we observe that such unions can be found by addition and subtraction of the function, using various input subsets.

Proposition 1 *Given a function $f(X)$ representable as a PPA, $P(f(X))$, and an approximation $g(X)$ such that $P(g(X)) \subseteq P(f(X))$ and the input subset X_1*

on the common input bits $\{b_1, \cdots, b_n\}$. *The union of the two approximations is a function* $h(X)$ *i.e.* $P(h(X)) := P(g(X)) \cup P(f(X_1))$. *We make the following observervation:*

$$h(X) = g(X) + f(X_1) - g(X_1)$$

and $P(h(X)) \subseteq P(f(X))$.

Example 2

Consider for instance $f_{ex}(X)$ of Example 1. Given is the approximation $g(X) = b_1 b_2 2^{-1} + b_1 2^{-1} + b_2 2^{-2}$; note that $P(g(X)) \subset P(f_{ex}(X))$. We get another approximation by using the input subset $X_1 = \{b_1\}$ on $f_{ex}(X)$. All products which contain b_2 disappear and $f_{ex}(X_1) = b_1 2^{-1} + b_1 2^{-2} + b_1 2^{-3}$. We add $f_{ex}(X_1)$ to $g(X)$ and subtract the products which appear twice, i.e. $g(X_1) = b_1 2^{-1}$. Then $h(X) = b_1 b_2 2^{-1} + b_1 2^{-1} + b_1 2^{-2} + b_2 2^{-2} + b_1 2^{-3}$ and $P(h(X) \subset P(f_{ex}(X))$.

With Proposition 1 we can define recursively the union of several sets $P(f(X_r))$ by function addition and subtraction. Assume that the union of $m-1$ subsets is given by $g(X) = \bigcup_{r=1}^{m-1} P(f(X_r)) = P(\sum_{j=1}^{t} s_j f(G_j))$, where each $s_j = \pm 1$ and G_j are appropriate input subsets. Then with Proposition 1 we get:

$$\bigcup_{r=1}^{m} P(f(X_r)) := g(X) + f(X_m) - g(X_m) \quad (4)$$

3 Subset approximations

3.1 General Approximation

The set considerations of the previous sections allow us to derive an approximation formula and to estimate its error. Formula (4) above provides an iterative generation of approximations for $f(X)$. The basic idea is that we unite smaller sets to approximate the $P(f(X))$. It is not possible to generate all products in this way. However, we get all significant ones, i.e. those within the precision limit.

Definition 2 *Assume input subsets* $X_1, X_2 \cdots \subset X$. $g_1(f(X), X_1) := f(X_1)$ *and iteratively* $g_{k+1}(f(X), X_1, \cdots, X_{k+1}) := g_k(f(X), X_1, \cdots, X_k) + f(X_{k+1}) - g_k(f(X), X_1 \cap X_{k+1}, \cdots, X_k \cap X_{k+1})$

For easier notation we define

Definition 3 $A_k(f(X)) := g_k(f(X), X_1, \cdots, X_k)$.

Two examples are $A_2(f(X)) = f(X_1) + f(X_2) - f(X_1 \cap X_2)$ and $A_3(f(X)) = f(X_1) + f(X_2) - f(X_1 \cap X_2) + f(X_3) - f(X_1 \cap X_3) - f(X_2 \cap X_3) + f(X_1 \cap X_2 \cap X_3)$. Note that one can map A_2 by two tables, $T_1(X_1)$ and $T_2(X_2)$ and one addition. More generally, we store in the first table $T_1(X_1) := A_1(f(X))$ and in the k-th table $T_k(X_k) := A_k(f(X)) - A_{k-1}(f(X))$. To evaluate all functions for the k-th table we need only the input set X_k and subsets of it, respectively. These considerations boil down to the fact that the mapping of A_k needs k tables and $k-1$ additions. There is, of course, a rounding error, too, due to the limited table output. However, if the output chosen is long enough, this error is small, compared to the approximation error. Each approximation $A_k(f(X))$ corresponds to a subset of $P(f(X))$.

Next we consider the approximation error. Up to now we do not even know which subsets are good or bad. Good approximations will cover all higher significant bit products of $f(X)$. The quality of an approximation can be derived by a simple observation. $P(f(X))$ consists only of positive products and $P(A_k(f(X)) \subseteq P(f(X))$. Hence, we get following proposition:

Proposition 2 *For an approximation* A_k *there is*

$$\max_X (f(X) - A_k(f(X))) = f([1 \cdots 1]) - A_k(f([1 \cdots 1]))$$

Proposition 2 derives a sharp bound on the approximation error. Note that it is restricted to a_i-nonnegative functions. General functions are considered as the difference between two a_i-nonnegative functions, $f(X) = f_1(X) - f_2(X)$. There is a positive error both from f_1 and f_2. The worst case happens when one is large and the other zero, which concludes following proposition:

Proposition 3 *For a function* $f(X) = f_1(X) - f_2(X)$, *each* f_i a_i*-nonnegative, there is*
$|f(X) - A_k(f(X))| \leq \max [f_1([1 \cdots 1]) - A_k(f_1([1 \cdots 1])), f_2([1 \cdots 1]) - A_k(f_2([1 \cdots 1]))]$

Example 3

To see how Proposition 2 works, consider the following 2-table example. Assume we want to approximate the function $f(X) = -\ln(1-X) = \sum_{i=1}^{\infty} X^i/i$ with $0 \leq X < 1/2$. Parameter X consists of 23 bits ($X = [.0b_2 \cdots b_{24}]$). However, we want to use two ROM tables each with only 16-bit input.

From Definition 2 we derive the approximation $A_2(-\ln(1-X)) = -\ln(1-X_1) - \ln(1-X_2) + \ln(1-(X_1 \cap X_2))$. We decide to choose the input subsets $X_1 = [.0b_2 \cdots b_{17}0 \cdots 0]$ and $X_2 = [.0b_2 \cdots b_{10}0 \cdots 0b_{18} \cdots b_{24}]$. Each X_k contains 16 variable bits. Proposition 2 allows us to compute the worst case error as follows:

$-\ln(1-(.5-2^{-24})) + \ln(1-(.5-2^{-17})) + \ln(1-(.5-2^{-10} + 2^{-17} - 2^{-24})) - \ln(1-(.5-2^{-10})) < .98 \times 2^{-25}$

(Of course, we calculate the logarithm with a better precision than 2^{-25}) Hence, we can add the output of table $T_1(X_1) = -\ln(1-X_1)$ to that of $T_2(X_2) = -\ln(1-X_2) + \ln(1-(X_1 \cap X_2))$ and round it to 24 bits to the right of the radix point. Remember that there is a positive error, i.e. $f(X) - A_i(f(X)) \geq 0$. We exploit this fact, too, and map to the first table $T_1(X_1) = -\ln(1-X_1) + .98 \times 2^{-26}$ instead. Thus we can further reduce the final error of the rounded value by a factor of 1/2. Note that we omit in this discussion the so called *Table maker dilemma*, which is the impossibility of doing *exact* rounding on prerounded values [4].

Two other problems are demonstrated by the above example. The first is, how did we decide which subsets

# ROM's	ROM size
1	$2^{23} \times 1 \times l$
2	$2^{16} \times 2 \times l$
3	$(2^{15} \times 2 + 2^{11} \times 1) \times l$
5	$(2^{14} \times 4 + 2^{8} \times 1) \times l$
6	$2^{13} \times 6 \times l$
20	$2^{10} \times 20 \times l$

Table 1: *The table shows number and size of ROM's to map* $\ln(1-X)$ *with an error less than* 2^{-25}. *In the cases of 3 and 5 ROM tables one of the tables is smaller in size. To compute* $\ln(Y)$ *we assume* $Y = [.1w_2 \cdots w_{24}]$. *The transformed parameter is then* $X = [.0b_2 \cdots b_{24}]$. *The number of output bits* l *should be slightly larger than 24. The 20-table case is near the limit of testing and input size. (It would require 19 additions, too.) The according subsets can be found in Appendix 2.*

to choose. The second is as follows. It is very easy to compute the worst case error, once the ROM size, the number of parameter bits and bit subsets have been determined. However, it seems difficult to gain general results for a variable number of bits. We deal with both problems in the next section.

3.2 2-Table Approximation

There is naturally a trade-off among the needed storage, number of adders and approximation error. Table 1 shows the relation between the number of subsets (=number of ROM's) and magnitude of subsets (=ROM input size) with the approximation error less than 2^{-25}.

Figure 1 in Appendix 2 depicts an informal description of the algorithm. The subsets according to Table 1 can also be found in Appendix 2. In this paper we assume that the 1-Table approach is impossible because of the table size. The next fastest approach is the 2-Table case. It needs only one addition after the look up phase. Hence we present some general results for the important 2-Table approach.

Definition 4 *Assume a nonoverlapping partition of the input* X *into three blocks* x, y *and* z. *Namely* $x = [.b_1 \cdots b_r]$, $y = [.0 \cdots 0 b_{r+1} \cdots b_s]$ *and* $z = [.0 \cdots 0\ b_{s+1} \cdots b_n]$ *where* $X = [.b_1 \cdots b_r\ b_{r+1} \cdots b_s\ b_{s+1} \cdots b_n]$. *Let* $X_1 = [.b_1 \cdots b_s]$ *and* $X_2 = [b_1 \cdots b_r\ 0 \cdots 0 b_{s+1} \cdots b_n]$. *We call the 2-table approximation using the above* X_1 *and* X_2 *for indexing 3-block method.*

The blocks x, y and z represent also different sections of significance of X, i.e. $x < 1$, $y < 2^{-r}$ and $z < 2^{-s}$. This significance is used to formulate the following theorem.

Theorem 4 *The approximation of a function* $f(X) = \sum_{i=0}^{\infty} a_i X^i$ *with* A_2 *by the 3-block method has an approximation error* ϵ *bound by:*

Function	ROM size
$1/(1-X)$	$(2^{17} + 2^{16}) \times l$
$\mathrm{ld}\,(1-X)$	$(2^{17} + 2^{15}) \times l$
2^X	$(2^{17} + 2^{15}) \times l$

Table 2: *Shown are the ROM-sizes for several elementary functions mapped by two tables. For all cases we assume parameters to the 24th bit right of the radix and map the functions with an approximation error less than* 2^{-25}. l *should be chosen slightly larger than 24 to keep the rounding error within this range. The according subsets have been generated by the heuristic given in Appendix 2.*

(a) *For* $0 \le X < 1/2$ *and* $0 \le a_i \le 1$ *the error is* $\epsilon < 2^4 yz$

(b) *For* $0 \le X$, $1/4 \ge y$ *and* $0 \le a_i \le 1/i!$ *the error is* $\epsilon < e^X(yz + 4y^3)$

The proof can be found in Appendix 1. Theorem 4 provides a good rule of thumb for the 3-block method. It guarantees for many functions that the bit range with n bits can be divided into three (almost equal) ranges. Hence, the storage is reduced from 2^n to about $2^{2n/3}$. However, once the function and the subsets are known, Proposition 2 and 3 provide more accurate bounds.

4 Applications

4.1 Reciprocal

One of the most obvious applications is reciprocation. Fast reciprocation can be used to do fast division. Many algorithms have been proposed to compute reciprocals [3].

Given a number $Y = .1w_2 \cdots w_n$, our task is to compute $1/Y$. We transform Y to $X = 1 - Y$ and $X = [.0v_2 \cdots v_n] < 1/2$. We can compute $1/Y = 1/(1-X)$ with the approach described in Section 3, using $1/(1-X) = \sum_{i=0}^{\infty} X^i$. A 2-Table approach as discussed in Section 3.2 is possible with an error smaller than the bound given in Theorem 4 (a). See also Table 2 for the 24 bit case. The heuristic of Figure 1 can be used to generate subsets for a larger number of tables as well.

4.2 Logarithm

We already presented subset combinations to compute the natural logarithm of some number Y'. For base 2 logarithms we just multiply by $\ln 2$ (also increasing the error by this factor).

We can always scale Y' by shifting to $Y = [.1u_2 \cdots u_n]$. To compute $\mathrm{ld}\,Y$ we transform $X = 1 - Y = [0.0v_2 \cdots v_n]$. Then $\mathrm{ld}\,(1-X) = -1/\ln 2 \sum_{i=1}^{\infty} X^i/i$. The outputs of the according tables add up to the logarithm. Finally we get $\mathrm{ld}\,(Y') = \mathrm{ld}\,(1-X)$+number of shifts.

Since the function is a_i-nonpositive we always get a negative error. We assume an approach which gener-

ates the worst case error ϵ_w less than 2^{-n}. Subtracting $\epsilon_w/2$ allows virtually exact rounding.

4.3 Exponentiation

Consider an approach to compute $2^B = \sum_{i=0}^{\infty} (B \ln 2)^i / i!$. If B has an integer part, it has to be removed and added later to the exponent or used to make appropriate shifts. Assume B is $B = [I.b_1 \cdots b_n]$. The integer part I is removed. To compute $2^{[.b_1 \cdots b_n]}$ one can find according subsets with the algorithm of Figure 1, and generate the appropriate ROM tables.

5 Limitations and Open Questions

The proposed method exploits properties of fundamental functions, which may not be found in general functions. Especially important is the convergence rate. A function which converges quickly has in the significant bit range relatively small bit products, which are easier to contain in small tables.

The problem of deciding if given input subsets are optimal under certain restrictions is not trivial and quite important, too. Rote of TU Graz has pointed out a connection with Monge matrices to check subsets for optimality. This will be a topic of further investigation.

Another question concerns the minimum input size of the tables. For $0 \leq X = [.b_1 \cdots b_n] < 1$ there is a trivial lower limit for the number of input bits r. If we use only tables with r input bits we cannot contain bit products which consist of more than r bits. Of all these unmappable bit products the most significant is $b_1 b_2 \cdots b_{r+1}$. It is created by X^{r+1} and causes an error $E = a_{r+1} 2^{-(r+2)(r+1)/2}$. To make sure that this error is below the significant range it is required that $E < 2^{-n}$ which we transform to $(r+2)(r+1)/2 > n - \mathrm{ld}\, a_{r+1}$. For instance the function $1/(1-X)$ would have a lower limit of roughly $\sqrt{2n}$ for the input bits of the according tables. The actual minimum may be harder to find. However, an approach which uses tables with smallest input size possible does not look promising. A large number of tables has to be used causing many additions, which wipe out the speed advantage of look up's.

6 Conclusion

An approximation algorithm has been presented which allows us to decompose a function $f(X)$ into a sum of functions, each with smaller input. The general idea is to compute functions by several parallel look up's in small ROM-tables and additions. Hence the overall memory space required is much smaller than that of a direct 1-Table approach. The additional costs are one or a few additions. We have shown how to compute bounds for the worst case error for arbitrary and overlapping input subsets $X_k \subset X$. We can derive the error bound for an arbitrary number of tables and for arbitrary functions which can be developed into a converging series. Included is also a fast heuristic algorithm which generates input subsets. Several examples have been presented with a precision of 24 bits. In these examples we gain a memory improvement by a factor of between 64 and 400. For the general n-bit precision and for general functions (which can be developed into series) we also show a general memory improvement from 2^n to about $2^{2n/3}$ for the 2-Table approach. We consider the presented method useful for fast hardware implementations of functions like reciprocal, logarithm, exponentiation, division, square root, trigonometric functions and others.

Acknowledgement

The first author has performed this work at the Department of Information Science of Kyoto University. He is Monbusho scholarship student. The authors thank Mr. Masayuki Ito(†), Prof. Franco Preparata(‡), Doz. Guenter Rote(◁), Ms. Abigail Schweber(⋆), Mr. Kazuyoshi Takagi(†) and Prof. Shuzo Yajima(†) for helpful discussions. † Kyoto University, ‡ Brown University, ◁ TU Graz, ⋆ Harvard University

Appendix 1

Proof of Theorem.4 (a) Assume first $a_i = 1$. Then we have to consider function $1/(1-X) = \sum_{i=0}^{\infty} X^i$. From Definition 2 and Proposition 2 we get:

$\epsilon \geq 1/(1-X) - A_2(1/(1-X)) =$

$$\frac{1}{1-x-y-z} - \frac{1}{1-x-y} - \frac{1}{1-x-z} + \frac{1}{1-x} \quad (5)$$

We set $x' = 1-x$ and $y' = x' - y$. Then (5) becomes:

$$\frac{1}{y'-z} - \frac{1}{y'} - \frac{1}{x'-z} + \frac{1}{x'} = \frac{z[x'(x'-z) - y'(y'-z)]}{x'y'(x'-z)(y'-z)}$$

Substituting for x'^2 and y'^2 we get

$$\frac{z(2x'y - y^2 - yz)}{x'y'(x'-z)(y'-z)} < \frac{2zy}{y'(x'-z)(y'-z)} < 2^4 yz$$

Because of $x < 1/2$ we have $x' > 1/2$. See above requirement that x, y and z represent different bit ranges and $y > z$. Hence also $y' > 1/2$ and also $x' - z > y' - z > 1/2$. So we can further simplify our estimate to the final bound.

Let us consider now the case for more general a_i. The error is:

$\epsilon = f(X) - A_2(f(X)) = \sum_{i=0}^{\infty} a_i(X^i - (x+y)^i - (x+z)^i + x^i$

Since there is $0 \leq a_i < 1$ and because all sum terms are a_i-nonnegative we simplify the estimate to

$\epsilon < \sum_{i=0}^{\infty} (X^i - (x+y)^i - (x+z)^i + x^i) = 1/(1-X) - A_2(1/(1-X)) < 2^4 yz$

Proof of (b). The error is $\epsilon = f(x+y+z) - A_2(f(x+y+z))$ and we set $a_i = 1/i!$. Then $\epsilon = e^{x+y+z} - e^{x+y} - e^{x+z} + e^x$. We write it as:

$$e^x[(e^y - 1)(e^z - 1)] \quad (6)$$

The infinite series formula for $e^y = \sum_{i=0}^{\infty} y^i/i!$. We approximate these products by the first two terms and give an upper bound for the rest, yielding

$$(e^y - 1) < y + y^2/(1-y)$$

Similarly for z. Substituting $e^y - 1$ and $e^z - 1$ by their upper bounds in equation (6), replacing all higher order terms z by y, reduce y^4 to $y^3/4$ and replace $(1-y)$ by $3/4$, respectively, yields the upper bound. Then we get the upper bound

$$\epsilon < e^x(yz + 4y^3)$$

□

Appendix 2

Figure 1 describes the heuristic we used to find input subsets. In the heuristic we use integers to describe subsets and partial products, X_k and Z. The union of two such sets is simply bitwise OR. A partial product $b_2b_3b_7$ corresponds to $Z = 1000110$. In principal all possible partial products could be counted through, but the heuristic is speeded up tremendously by skipping partial products which are beyond significance in step (G3). ROMSIZE is the number of input bits for the ROM. Each subset has maximum ROM-size elements. To test the size of a subset we count the number of 1-bits of the according integer. The variable *Bound* is the error bound which must finally be fulfilled by the subsets. To compute $error(f, subsets)$ the algorithm uses Proposition 2.

The main idea of the heuristic is to generate the most significant products of the PPA of $\sum_{i=0}^{\infty} X^i$. The found subsets can also be used for functions with sum coefficients $0 \leq a_i < 1$. The heuristic is easy to program, very fast and needs (for most cases) much less memory than programming the full PPA.

GenerateSet(f,X,ROMSIZE,N,Bound)

(S0) $significance(Z)$:=sum of all non zero bit positions; (e.g. $significance(1100110) = 6+5+2+1$)

(G1) $Precision := Inputbits$ (=length of X);

(G2) For $k = 1$ To N $X_k := \{\}$;

(G3) For $Z = 0$ To $2^{Inputbits} - 1$
{If $significance(Z) \leq Precision$ { Seek first subset X_k with $hamming(Z \cup X_k, 0) \leq ROMSIZE$; $X_k := X_k \cup Z$}
Else Increment Z at lowest nonzero postion (e.g. 110100 → 111000) }

(G4) If $error(f, subsets) > Bound$
{ Precision:=Precision+1; Goto (G2)}

Figure 1: *Greedy algorithm to generate N input sets, subsets from X, according to function f(X).*

Below are the subsets which have been found by the heuristic algorithm of Figure 1 according to Table 1. All of them assume a parameter with 24 bit precision $X = [.0b_2 \cdots b_{24}]$. and have been generated to guarantee an error less than 2^{-25}. In the notation below we denote bits included in the subset by **1** and missing bits by **0**.

```
RB=ROM input bits

1 Table
11111111 11111111 1111111 (23RB)

2 Tables, ld(Error)= -25.025
11111111 11111111 0000000 (16RB)
11111111 10000000 1111111 (16RB)

3 Tables, ld(Error)= -25.145
11111111 11111110 0000000 (15RB)
11111111 11100001 1110000 (15RB)
11111110 00000000 0001111 (11RB)

5 Tables ld(Error)= -25.636
11111111 11111100 0000000 (14RB)
11111111 11111010 0000000 (14RB)
11111111 11110001 1000000 (14RB)
11111111 11000000 0111100 (14RB)
11111100 00000000 0000011 (8RB)

6 Tables ld(Error)= -25.087
11111111 11111000 0000000 (13RB)
11111111 11110100 0000000 (13RB)
11111111 11001110 0000000 (13RB)
11111111 10110011 0000000 (13RB)
11111111 11000001 1100000 (13RB)
11111111 00000000 0011111 (13RB)

20 Tables ld(Error)= -25.159
11111111 11000000 0000000 (10RB)
11111111 10100000 0000000 (10RB)
11111111 01100000 0000000 (10RB)
11111100 11110000 0000000 (10RB)
11111111 10010000 0000000 (10RB)
11110011 01111000 0000000 (10RB)
11111111 10001000 0000000 (10RB)
11101100 01111100 0000000 (10RB)
11111111 10000100 0000000 (10RB)
11111000 01110110 0000000 (10RB)
11111111 10000010 0000000 (10RB)
11111110 00001011 0000000 (10RB)
11111101 11000001 0000000 (10RB)
11111100 00110001 1000000 (10RB)
11111111 10000000 1000000 (10RB)
11111110 01000000 1100000 (10RB)
11111001 10100000 0110000 (10RB)
11111110 01000000 0011000 (10RB)
11111101 10000000 0001100 (10RB)
11111110 00000000 0000111 (10RB)
```

References

[1] D.M. Mandelbaum, *A systematic method for division with high average bit skipping*, IEEE Trans. Comput. v.39, p.127–130, Jan. (1990)

[2] D.M. Mandelbaum, *Some results on a SRT type division scheme* IEEE Trans. Comput. v.42, p.102–106, Jan. (1993)

[3] D. DasSarma, D.W. Matula, *Measuring the Accuracy of ROM Reciprocal Tables*, IEEE Trans. Comput., v.43, p.932–940, Aug. (1994)

[4] M. Schulte, E. Swartzlander, *Exact Rounding of Certain Elementary Functions*, Arith 11, Proceedings, p.138–145 (1993)

[5] E.M. Schwarz, M.J. Flynn, *Hardware Starting Approximations For The Sqaure Root Operation*, Arith 11, Proceedings, p.103–111 (1993)

[6] E.M. Schwarz, *High Radix Algorithms For High Order Arithmetic Operations*, Dissertation, Univ. Stanford California, Dec.(1992)

[7] R. Stefanelli, *A suggestion for a high-speed parallel binary divider*, IEEE Trans. Comput. v.42(1), p.42–45 Jan. (1972)