

Application of LUT Cascades to Numerical Function Generators

T. Sasao

J. T. Butler

M. D. Riedel

Department of Computer Science
and Electronics
Kyushu Institute of Technology
Iizuka, 820-8502 JAPAN
sasao@kyutech.ac.jp

Department of Electrical
and Computer Engineering
Naval Postgraduate School
Monterey, CA U.S.A. 93921-5212
Jon_Butler@msn.com

Department of Electrical
Engineering
California Institute of Technology
Pasadena, CA U.S.A. 91125
riedel@caltech.edu

June 11, 2004

Abstract— The availability of large, inexpensive memory has made it possible to realize numerical functions, such as the reciprocal, square root, and trigonometric functions, using a look-up table. This is much faster than by software. However, a naive look-up method requires unreasonably large memory. In this paper, we show the use of a look-up table (LUT) cascade to realize a piecewise linear approximation to the given function. Our approach yields memory of reasonable size and significant accuracy.

1 Introduction

CORDIC (COordinate Rotation DIgital Computer) [1, 15] algorithms are often used to compute trigonometric functions, like \sin and \cos , as well as other functions. They have been implemented in many calculators [8]. CORDIC achieves accuracy with relatively little hardware by *iteratively* computing successively more accurate bits using a shift and add technique. This is slow compared to a table lookup process in which an argument x is encoded as an n -bit number that is used as an address for $f(x)$ in memory. The computation time, in this case, is small and equal to one memory access. However, a naive table lookup can involve huge tables. For example, if x is represented as a 16 bit word and the results are realized by an 8-bit word, there would be $8 \times 2^{16} = 2^{19}$ bits total, a large number. In addition, there is much

redundancy of stored values, as higher order bits of the stored values are the same for nearby addresses. This has motivated the search for methods to achieve the high speed of table lookup with memories of reasonable size.

Hassler and Takagi [4] studied the problem of reducing the large size of a single lookup table by using two or more smaller lookup tables. Their approach applies to functions that can be represented as a converging series and uses the Partial Product Array (PPA), formed by multiplying together the various bits of the input variable x .

Stine and Schulte [13, 14] propose a technique that is based on the Taylor series expansion of a differentiable function. The first two terms of the expansion are realized and added using smaller lookup tables than needed in the naive method. Schulte and Swartzlander [12] consider algorithms for a family of variable precision arithmetic function generators that produce an upper and lower bound on the result, in effect carrying along the range over which the function is accurate. These algorithms have been simulated in behavioral level VHDL.

Lee, Luk, Villasenor, and Cheung [6, 7] have proposed a non-uniform segmentation method for use in computing trigonometric and logarithmic functions by table lookup. Their algorithm places closely-spaced points in regions where the change in function value is greatest. However, they used an ad hoc circuit to generate the non-uniform segmentation, and

the segments were not optimized to the given function.

Rather than an ad hoc choice for this circuit, we propose a circuit, called a segment index encoder, that is specifically designed for the function. Toward this end, we propose an algorithm that derives a near-optimal segmentation intended to minimize the approximation error. Then, we show how to design a LUT cascade [5, 9, 10, 11] to implement the segment index encoder. The advantage of our approach is that approximations are more accurate over a wider class of functions.

To illustrate this, we analyze a wider class of functions, extending to sigmoid and entropy functions. Our approach can be applied to elementary functions (including trigonometric functions, transcendental functions, and the power function), and to non-elementary functions (including the normal distribution and elliptic integral function). We do not require a converging series for the realized function, as in [4], nor do we require that the function be differentiable, as in [13, 14].

2 The Problem

In our designs, we assume that a single variable x is given as a 16 bit binary number, and we seek an 8 bit binary value that represents the function $f(x)$. We could represent $f(x)$ in a single memory, where x is applied as an address, and the memory contents represents a binary value for $f(x)$. Instead, we choose to find a piecewise linear approximation to $f(x)$, where each segment is represented as $xc_1 + c_0$. In this case, we require a segment index encoder that converts the 16 bit representation of x into an q bit code that is the segment index. This is then applied to a much smaller memory that produces binary numbers for c_1 and c_0 . This scheme requires a multiplier that computes c_1x and an adder that adds c_0 to c_1x to form $f(x)$.

There are two important parts to this. First, we need a segmentation of $f(x)$ that minimizes the error caused by representing a general function as a linear function $xc_1 + c_0$. Second, we need a compact realization of the segment index encoder.

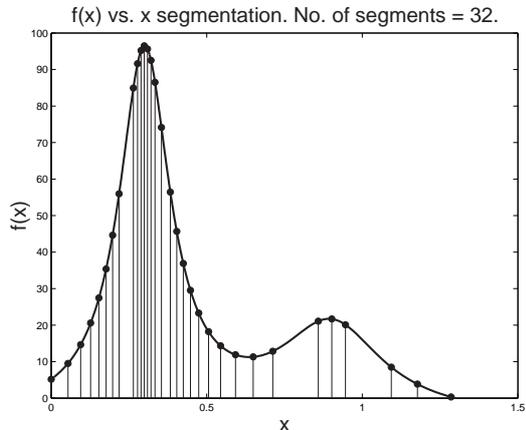


Figure 1: Segmentation of MATLAB's humps function.

Consider the segmentation problem. Fig. 1 shows MATLAB's 'humps' function $f(x) = \frac{1}{(x-0.3)^2+0.3} + \frac{1}{(x-0.9)^2+4} - 6$ and a piecewise approximation for it. There are 32 segments. The maximum absolute error over all segments is 0.26557, and the error within each segment is approximately uniform; each segment produces an error close or the same as 0.26557. Notice that small widths are needed around the left hump and to a lesser extent around the smaller right hump. These small segments produce nearly the same error as the large segments in the approximately linear portion of the curve on the right. The problem of generating near-optimum segmentations of functions is discussed in Section 4.

The second problem of designing the segment index encoder is complicated by the fact that different functions require different segmentations. In our running example, this encoder has 16 inputs and 5 outputs, and is potentially a large circuit. In the next section, we present a design method for the encoder using the LUT cascade.

3 Architecture for Numerical Function Generator

3.1 Overview

Table 1 shows the notation used in this paper. The first row shows the real-valued single-variable func-

tion $f(x)$ that our circuit approximates, where x is the independent variable. The second row shows the fixed-point numerical representation of x and $f(x)$. For the purposes of this paper, we choose x to be represented as a 16 bit fixed-point number and $f(x)$ to be represented similarly in 8 bits. That is, we use $f(x)$ and x to denote a real-valued function and its independent variable, *as well as* their fixed-point representations. Context will determine which meaning is intended. We use \mathbf{X} and $\mathbf{F}(\mathbf{X})$ to denote the *ordered set* of logic variables and logic functions representing fixed-point numbers x and $f(x)$, respectively. That is, $\mathbf{F}(\mathbf{X})$ is a multiple-output function on \mathbf{X} . It is the logic function our proposed circuit implements.

Table 1: Notation

Type	Ind. Var.	Function	Examples	# Bits
real-valued	x	$f(x)$	$x = \pi/4$ $\cos(x) = 0.0$	
fixed-point	x	$f(x)$.1000000000001001 .00000000	16 8
logic	\mathbf{X}	$\mathbf{F}(\mathbf{X})$	1000000000001001 00000000	16 8

As shown in Table 1, the number of bits, n , we use to represent the input variable, x , is 16. Another parameter, m , represents the range of values for $f(x)$. For example, if $0.0 \leq f(x) \leq 4.0$, we would choose $m = 2$. If $n = 6$, there are four bits of accuracy after the binary point, as can be seen by the two extreme values: $00.0000_2 = 0_{10}$ to $11.1111 = 4 - \frac{1}{16}$.

Fig. 2 shows the architecture used to implement the function. The independent variable x labels the 16 binary inputs that drive the Segment Index Encoder. The Encoder, in turn, produces the segment number in which this value of x is located. The segment number is applied to the Coefficients Table, which produces the slope c_1 and the constant c_0 for the linear approximation $c_1x + c_0$ to $f(x)$ in this interval. A multiplier is needed to compute c_1x and an adder is needed to compute the sum in $c_1x + c_0$. The logic variables from the adder, labelled by $f(x)$, form the approximation to the function.

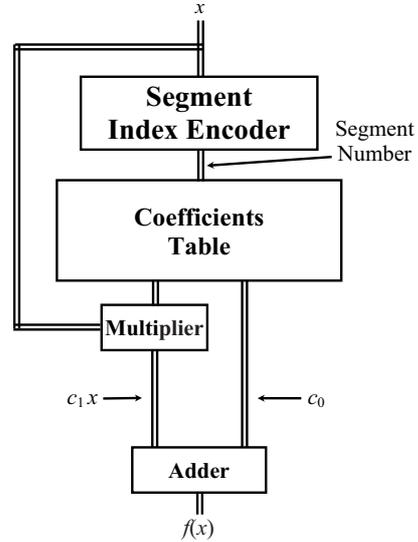


Figure 2: Architecture For the Numerical Function Generator.

3.2 Segment Index Encoder

The segment index encoder realizes the *segment index function* $g(x) : [0, 1 - 2^{-16}] \rightarrow \{0, 1, 2, \dots, p-1\}$ shown in Table 2. It assumes $0 \leq x < 1.0$ ($m = 0$). Suppose that x is represented in 16 bits, and we want to approximate $f(x)$ using p segments. The segment index encoder, therefore, has 16 inputs and $q = \lceil \log_2 p \rceil$ outputs. The success of this approach depends on finding a compact circuit for the segment index encoder.

Table 2: Segmentation Index Function

Input Range	Segment #
$0 \leq x < s_0$	0
$s_0 \leq x < s_1$	1
$s_1 \leq x < s_2$	2
$s_{p-1} \leq x < 1 - 2^{-16}$	$p - 1$

We propose the use of a LUT cascade [5, 9, 11] to realize the segment index encoder, as shown in Fig. 3. This maps \mathbf{X} to \mathbf{S} , where $\mathbf{S} = (s_{q-1}, s_{q-2}, \dots, s_0)$ represents the segment

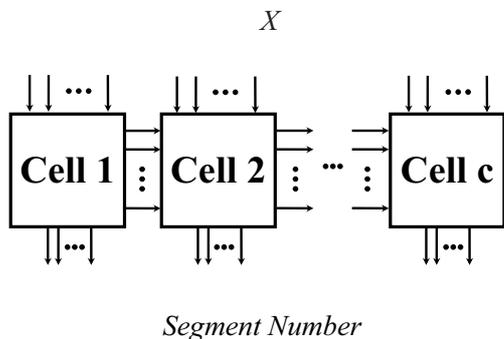


Figure 3: LUT Cascade Realization of the Segment Index Encoder.

number $s_{q-1}2^{q-1} + s_{q-2}2^{q-2} + \dots + s_02^0$. The LUT cascade realizes the segment index function shown in Table 2. This function is monotone increasing. That is, as we scan x in ascending order of values, the segment number never decreases. This property results in a LUT cascade with reasonable size, as we show in Lemma 1. We measure *size* by the number of bits of memory needed to store the cell's function over all cells in the LUT cascade. The size, in turn, is dependent of the number of *rails* or interconnecting lines between cells. This number can be determined from the decomposition chart of the function. This chart partitions the variables into two subsets. One subset corresponds to the variables on the input side of a set R of rails and the other subset corresponds to the variables on the output side of R .

Lemma 1: Let $(\mathbf{X}_{\text{high}}, \mathbf{X}_{\text{low}})$ be an ordered partition of \mathbf{X} into two parts, where $\mathbf{X}_{\text{high}} = (x_{n-1}, x_{n-2}, \dots, x_{n-k})$ represents the most significant k bits of x (x_{high}), and $\mathbf{X}_{\text{low}} = (x_{n-k-1}, x_{n-k-2}, \dots, x_0)$ represents the least significant $n - k$ bits of x (x_{low}). Consider the decomposition chart of $g(\mathbf{X})$ (representing a monotone increasing numerical function $g(x)$), where values of \mathbf{X}_{low} label the columns, values of \mathbf{X}_{high} label the rows, and entries are values of the p -valued segmentation function, s . Its column multiplicity is at most p .

(Proof) Assume, without loss of generality, that both the columns and rows are labelled in ascending order of the value of x_{low} and x_{high} , respectively. Because $g(x)$ is a monotone increasing function, in scanning left-to-right and then top-to-bottom, the values of $g(x)$ will never decrease. An increase causes two columns to be distinct. Conversely, if no increase occurs anywhere across two adjacent columns, they are identical.

In a monotone increasing p -valued output function, there are $p - 1$ dividing lines among 2^n output values. Dividing lines among values divide columns in the decomposition chart. Thus, there can be at most p distinct columns. ■

The significance of Lemma 1 is that a column multiplicity of p implies that there are at most $\lceil \log_2 p \rceil$ lines between the block associated with \mathbf{X}_{low} and \mathbf{X}_{high} . A low value, as suggested in Lemma 1, implies the individual cells have a small number of rails (interconnecting lines). As a result, the individual cells are reasonably simple. A formal statement of this is

Theorem 1: If the segment index function $g(x)$ maps to at most p segments, then there exists a LUT cascade realizing $g(x)$, where the number of rails is at most $\lceil \log_2 p \rceil$.

Fig. 3 shows that the outputs for the segment index encoder extend from the bottom of each cell. In general, any cell can produce an output. However, our experience is that most outputs come only the right cells. In the example we describe, all outputs come from the rightmost cell.

4 Segmentation Algorithm

Our approach is based on the Douglas-Peucker [3] polyline simplification algorithm. This algorithm finds a piecewise linear approximation to a function $f(x)$ recursively. First, it approximates $f(x)$ as a single straight-line segment connecting the end points. Then, it finds a point P on the curve for $f(x)$ that is farthest from the straight-line segment on a line

perpendicular to the segment. It then creates two straight-line segments joined at P connecting to the end points. It proceeds in this manner, stopping when the maximum distance from the straight-line segment is below a given threshold. The Douglas-Peucker algorithm is used in rendering curves for graphics displays. For our purposes, however, we seek a piecewise linear approximation that minimizes the approximation error. That is, if $f_p(x)$ is the piecewise linear approximation to $f(x)$, where p is the number of segments, we seek to minimize $|f(x) - f_p(x)|$. Thus, we have modified the Douglas-Peucker algorithm by replacing the perpendicular distance criteria with a minimum error criteria.

We have applied the modified Douglas-Peucker algorithm to the functions in Table 3. This shows common numeric functions, including transcendental functions, the entropy function, the sigmoid function, and the Gaussian function. The interval of x values is shown using the $[a, b)$ notation, where $a \leq b$. Here, $[a$ means the interval *includes* the smallest value a , and $b)$ means the interval *excludes* the largest value b . In the binary number representation of x , we enforce this by deducting 2^{m-n} from the largest value of x . That is, we deduct an amount equivalent to value of the least significant bit.

5 Example Design

In this section, we discuss in detail the design of the function generator for one function, $\cos(x)$. Then, we summarize key features of the designs for all functions listed in Table 3.

For the $\cos(x)$ function, the input \mathbf{X} has 16 variables and represents x to a precision of $2^{-16} \simeq 1.5 \times 10^{-5}$. Assume there are 16 segments. The LUT cascade must produce 4 outputs. Using the Douglas-Peucker algorithm, we determined an optimal segmentation as shown in Table 4

We sketch briefly the BDD design process described in [9]. Fig. 4 shows the BDD as a triangle. For each variable, there is an associated width that is shown next to the BDD. In this BDD, let y_1 be the top variable, y_2 the next variable, etc.. The **width** of a BDD at level k is the number of edges from variables

Table 3: The Realized Functions.

Function $f(x)$	Input x	Output $f(x)$
2^x	$[0, 1)$	$[1, 2)$
$1/x$	$[1, 2)$	$[\frac{1}{2}, 1)$
\sqrt{x}	$[0, 2)$	$[0, \sqrt{2})$
$1/\sqrt{x}$	$[1, 2)$	$[1, \frac{1}{\sqrt{2}}]$
$\log_2(x)$	$[1, 2)$	$[0, 1)$
$\ln x$	$[1, 2)$	$[0, \ln 2)$
$\sin(\pi x)$	$[0, \frac{1}{2})$	$(0, 1)$
$\cos(\pi x)$	$(0, \frac{1}{2})$	$(0, 1)$
$\tan(\pi x)$	$[0, \frac{1}{4})$	$[0, 1]$
$\sqrt{-\ln(x)}$	$[\frac{1}{256}, \frac{1}{4})$	$[0, 2]$
$\tan^2(\pi x) + 1$	$[0, \frac{1}{4})$	$[1, 2]$
$-(x \log_2 x + (1-x) \log_2(1-x))$	$(0, 1)$	$(0, 1]$
$\frac{1}{1+e^{-x}}$	$[0, 1)$	$[\frac{1}{2}, \frac{1}{1+e^{-1}}]$
$\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$	$[0, \sqrt{2}]$	$[\frac{1}{\sqrt{2\pi}}, \frac{1}{\sqrt{2\pi}e}]$

labelled y_k down variables lower in the BDD, where edges incident to the same lower variable counted as 1. The order top-to-bottom is $x_0, x_1, \dots, x_{15}, s_3, s_2, s_1$, and s_0 .

Note that the width never exceeds 16. Thus, any partition yields a LUT cascade with at most 4 rails. The third column shows how 6 cells are can be used to realize the given function. Each cell has 6 inputs and 4 outputs. The resulting circuit is shown in Fig. 5.

5.1 Memory Size Needed For the $\cos(\pi x)$ Function

We can compare this realization with the naive method by computing the number of bits stored. That is, with the naive method, there is a large memory with 2^{16} locations, each producing 8 bits, for a total of $2^{19} = 524,588$ bits. With the LUT cascade realization, there are 6 cells each with a memory of $4 \times 2^6 = 256$ bits for a total of $6 \times 256 = 1536$ bits. The coefficients memory has a 4 bit address

Table 4: Optimal Segmentation for the $\cos(x)$ Function.

Segment in Decimal	Begin Point in Binary	Segment Number
0.00000000	0.0000000000000000	0000
0.08398545	0.0001010110000000	0001
0.16816267	0.0010101100001110	0010
0.25272335	0.010000001011001	0011
0.33795513	0.010101101000010	0100
0.42395397	0.011011001000100	0101
0.51110320	0.100000101101011	0110
0.59969064	0.100110011000010	0111
0.69009958	0.101100001010101	1000
0.77935814	0.110001111000010	1001
0.87139694	0.110111110001001	1010
0.96679140	0.111101110111111	1011
1.06678779	1.000100010001100	1100
1.16918097	1.001010110100111	1101
1.28058636	1.010001111101010	1110
1.57070045	1.100100100001100	1111

and stores 15 segment coefficient pairs. The two coefficients are represented in 10 and 9 bits, for a total of $15 \times (10 + 9) = 285$ bits. Totalling the LUT cascade and coefficients memory yields total of $1536 + 285 = 1821$ bits. It should be noted that the approximation method we propose requires a LUT cascade, a multiplier and an adder that is not present in the naive realization, and this contributes delay. However, a larger memory is likely to be slower than the much smaller memory required in the approximation approach. That is, the approximation method requires a memory that is slightly more than 1/300 of the memory size for the naive method!

5.2 Summary of Memory Requirements For Numerical Functions

Table 5 shows the results of the design process described for the $\cos(x)$ function.

The tables shows that the sizes across the various functions are small. They range from approximately

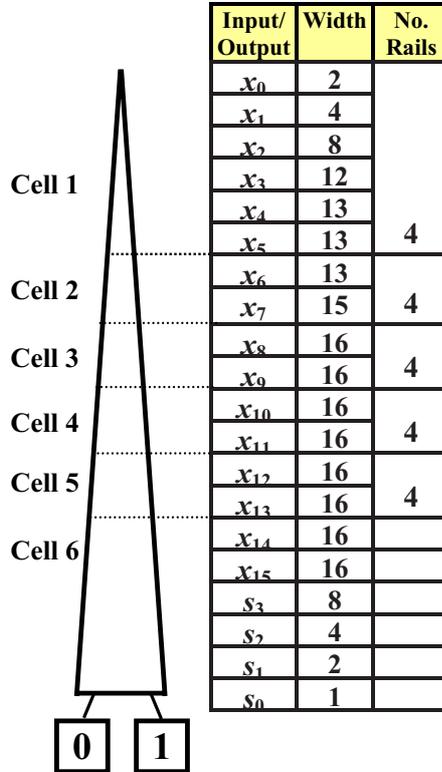


Figure 4: BDD for the Segment Index Encoder for the $\cos(x)$ Function.

1000 bits to approximately 4000 bits. Thus, our example of the $\cos(\pi x)$ function is representative. In forming the LUT cascades, we did not minimize the memory. For some functions, the minimum memory corresponded to a cascade with 9 cells.

6 Summary and Conclusions

We have shown a design method for circuits that computes elementary and non-elementary functions quickly and with accuracy. It is based on the piecewise linear approximation of the function. The effectiveness of this approach lies in two contributions, 1. an approximation algorithm of high accuracy and 2. the use of a LUT cascade in a compact realization

Table 5: Comparison of Sizes of Memory For Various Functions

Function $f(x)$	# Seg.	# Bits For				# Cell Inputs # Cell Outputs						LUT Mem.	Coef. Mem.	Total Mem.
		x	$f(x)$	c_1	c_0									
2^x	11	16	8	9	9	6	6	6	6	6	6	1536	198	1734
$1/x$	10	16	8	10	9	6	6	6	6	6	6	1472	85	1557
\sqrt{x}	5	16	8	9	8	6	6	6	5	5		896	85	981
$1/\sqrt{x}$	6	16	8	13	9	6	6	6	5	5		896	132	1028
$\log_2(x)$	9	16	8	11	10	6	6	6	6	6		1152	189	1341
$\ln x$	11	16	8	11	10	6	6	6	6	6	6	1536	231	1767
$\sin(\pi x)$	14	16	8	10	9	6	6	6	6	6	6	1536	266	1802
$\cos(\pi x)$	15	16	8	10	9	6	6	6	6	6	6	1536	285	1821
$\tan(\pi x)$	10	16	8	10	8	6	6	6	6	6	6	1536	180	1716
$\sqrt{-\ln x}$	26	16	8	12	10	7	7	7	7	7	6	3520	572	4092
$\tan^2(\pi x) + 1$	19	16	8	12	11	6	6	6	6	6	6	1920	437	2357
$x \log_2 x - (1-x) \log_2(1-x)$	34	16	7	13	12	8	8	8	8	8		7680	850	8530
$\frac{1}{1+e^{-4x}}$	12	16	8	8	8	6	6	6	6	6	6	1536	192	1726
$\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$	6	16	8	6	8	6	6	6	6	5		896	84	980

of the segment index encoder. The latter converts a binary representation of x into a binary representation of the segment number. Each segment number is an address to a reasonably small memory, which provides the coefficients of the corresponding segment.

Extensions of this work include the use of 1. a scaling factor (shifter) for functions with a large dynamic range, 2. a second order coefficient (c_2x^2) to reduce the approximation error in the segment, and 3. a design that incorporates a set of numeric functions.

ACKNOWLEDGEMENTS

This research is partly supported by a Grant-in-Aid for Scientific Research from the Japan Society for the Promotion of Science (JSPS) and funds from MEXT via the Kitakyushu Innovative Cluster Project.

References

- [1] R. Andrata, "A survey of CORDIC algorithms for FPGA based computers," *Proc. of the 1998 ACM/SIGDA Sixth*

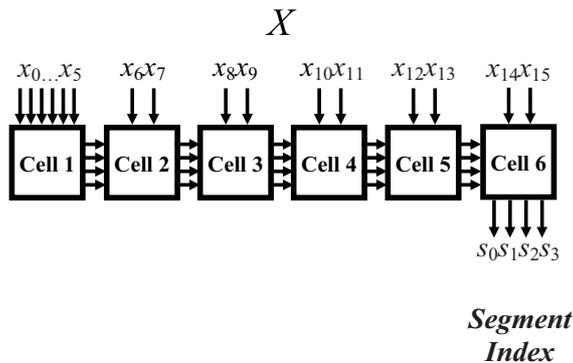


Figure 5: Segment Index Encoder for the $\cos(x)$ Function Realized by an LUT Cascade.

Inter. Symp. on Field Programmable Gate Arrays (FPGA '98), pp. 191-200, Monterey, CA, Feb. 1998.

- [2] J. Cao, B. W. Y. Wei, and J. Cheng, "High-performance architectures for elementary function generation," *Proc. of the 15th IEEE Symp. on Computer Arithmetic (ARITH'01)*, Vail, Co, pp. 136-144, June 2001.
- [3] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a line or its caricature," *The Canadian Cartographer*, Vol. 10, No. 2, pp. 112-122, 1973.
- [4] H. Hassler and N. Takagi, "Function evaluation by table look-up and addition," *Proc. of the 12th IEEE Symp. on Computer Arithmetic (ARITH'95)*, Bath, England, pp. 10-16, July 1995.
- [5] Y. Iguchi, T. Sasao, and M. Matsuura, "Realization of multiple-output functions by reconfigurable cascades," *International Conference on Computer Design: VLSI in Computers and Processors (ICCD01)*, Austin, TX, pp. 388-393, Sept. 23-26, 2001.
- [6] D.U. Lee, Wayne, Luk, J. Villasenor, and P. Y. K. Cheung, "Non-uniform segmentation for hardware function evaluation," *Proc. Inter. Conf. on Field Programmable Logic and Applications*, pp. 796-807, Lisbon, Portugal, Sept. 2003
- [7] D.U. Lee, Wayne, Luk, J. Villasenor, and P. Y. K. Cheung, "A hardware Gaussian noise generator for channel code evaluation," *Proc. of the 11th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'03)*, Napa, CA, pp. 69-78, April 2003.
- [8] S. Muroga, *VLSI system design: when and how to use very-large-scale integrated circuits*, John Wiley & Sons, New York, 1982.
- [9] T. Sasao, M. Matsuura, and Y. Iguchi, "A cascade realization of multiple-output function for reconfigurable hardware," *Inter. Workshop on Logic Synthesis (IWLS01)*, Lake Tahoe, CA, pp. 225-230, June 12-15, 2001.
- [10] T.Sasao, M. Kusano, and M. Matsuura, "Optimization methods in look-up table rings," *International Workshop on Logic and Synthesis (IWLS-2004)*, Temecula, CA, pp.431-437, June 2-4, 2004.
- [11] T. Sasao and M. Matsuura, "A method to decompose multiple-output logic functions," *41st Design Automation Conference*, San Diego, CA, pp.428-433, June 2-6, 2004.
- [12] M. J. Schulte and E. E. Swartzlander, "A family of variable-precision interval arithmetic processors," *IEEE Trans. on Comp.*, Vol. 49, No. 5, pp. 387-397, May 2000.
- [13] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Trans. on Computers.*, Vol. 48, No. 8, pp. 842-847, Aug. 1999.
- [14] J. E. Stine and M. J. Schulte, "The symmetric table addition method for accurate function approximation," *Jour. of VLSI Signal Processing*, Vol. 21, No. 2, pp. 167-177, June, 1999.
- [15] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Comput.*, Vol. EC-8, No. 3, pp. 330-334, Sept. 1959.