# Application of LUT Cascades to Numerical Function Generators

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

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

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

**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

Iterative algorithms have often been used to compute trigonometric functions like $sin(x)$ and $cos(x)$. Such algorithms are appropriate for hand calculators [8], where the input time by a human is much greater than the computation time. For example, the CORDIC (**CO**ordinate **R**otation **DI**gital **C**omputer) [1, 15] algorithm achieves accuracy with relatively little hardware by *iteratively* computing successively more accurate bits using a shift and add technique. This is slow compared to table lookup, 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 are $8 \times 2^{16} = 2^{19}$ 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 behaviorial 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]. Further, we do not require that the function be differentiable, as in [13, 14]; rather, it can be applied to functions that are piecewise differentiable, such as the sawtooth function.

## 2 The Problem

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 $c_1x + c_0$. 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 $c_1x + c_0$. Second, we need a compact realization of the segment index encoder.

Consider the segmentation problem. Fig. 1 shows MATLAB's 'humps' function

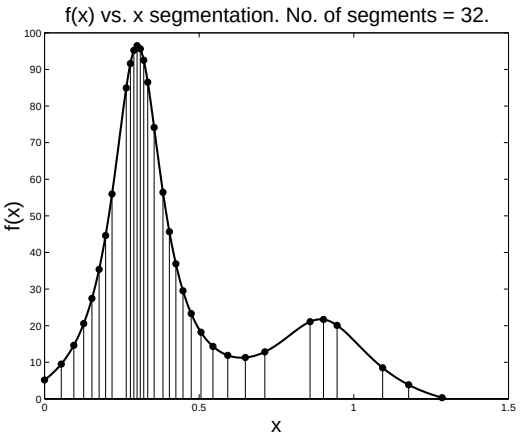


Figure 1: Segmentation of MATLAB's humps function.

$$f(x) = \frac{1}{(x-0.3)^2+0.3} + \frac{1}{(x-0.9)^2+4} - 6, \quad (1)$$

and a piecewise approximation for it.

There are 32 segments. The maximum absolute error over all segments is small, 0.26557, and the error within each segment is approximately uniform. That is, each segment produces an error close to 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 an implementation of a segmentation of the function of Fig. 1, the encoder converts a 16-bit input (value of $x$) into a 5-bit output (segment number), and is potentially a large circuit. In the next section, we present a design method for the encoder using the LUT cascade, such that the resulting circuit is small.

# 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 function $f(x)$ that our circuit approximates, where $x$ is the independent variable. The second row shows the fixed-point numerical representation of $x$ and $f(x)$. To illustrate our approach, we have chosen to represent $x$ in 16 bits and $f(x)$ 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 $\boldsymbol{X}$ and $\boldsymbol{F(X)}$ to denote the *ordered set* of logic variables and logic functions representing fixed-point numbers $x$ and $f(x)$, respectively. That is, $\boldsymbol{F(X)}$ is a multiple-output function on $\boldsymbol{X}$. It is the logic function our proposed circuit implements.

Table 1: Notation

| Type | Ind. Var. | Function | Examples | # Bits |
|---|---|---|---|---|
| real-valued | $x$ | $f(x)$ | $x = \pi/4 = 0.785398$<br>$cos(x) = 0.707107$ | |
| fixed-point | $x$ | $f(x)$ | .1100100100001111<br>.10110101 | 16<br>8 |
| logic | $\boldsymbol{X}$ | $\boldsymbol{F(X)}$ | 1100100100001111<br>10110101 | 16<br>8 |

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 intercept $c_0$ 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. $f(x)$ is represented by 8 bits.

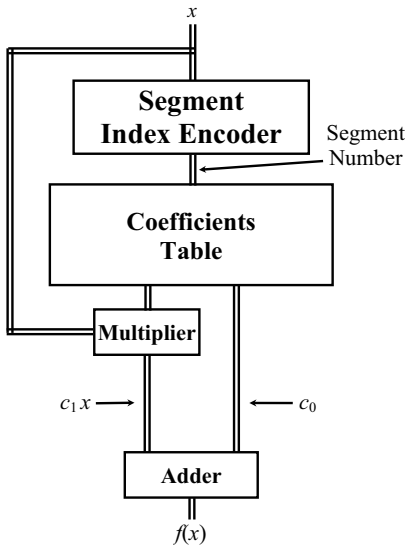


Figure 2: Architecture For the Numerical Function Generator.

## 3.2 Segment Index Encoder

The segment index encoder realizes the *segment index function* $g(x) : [0, 1-2^{16}] \rightarrow \{0, 1, 2, \ldots, p-1\}$ shown in Table 2. It assumes $0 \leq x < 1.0$. 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 $\boldsymbol{X}$ to $\boldsymbol{S}$, where $\boldsymbol{S}$= $(s_{q-1}, s_{q-2}, \ldots, s_0)$ represents the segment number $s_{q-1}2^{q-1} + s_{q-2}2^{q-2} + \ldots + s_0 2^0$. The LUT

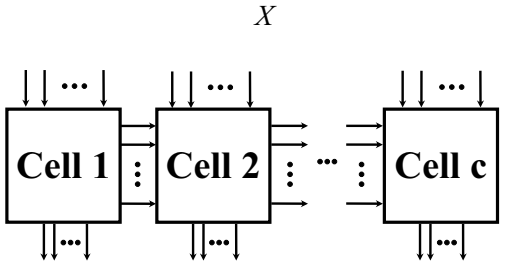


Figure 3: LUT Cascade Realization of the Segment Index Encoder.

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 ($\boldsymbol{X}_{\text{high}}$,$\boldsymbol{X}_{\text{low}}$) be an ordered partition of $\boldsymbol{X}$ into two parts, where $\boldsymbol{X}_{\text{high}} = (x_{n-1}, x_{n-2}, \ldots, x_{n-k})$ represents the most significant $k$ bits of $x$ ($x_{\text{high}}$), and $\boldsymbol{X}_{\text{low}} = (x_{n-k-1}, x_{n-k-2}, \ldots, x_0)$ represents the least significant $n-k$ bits of $x$ ($x_{\text{low}}$). Consider the decomposition chart of $\boldsymbol{g(X)}$ (representing a monotone increasing numerical function $g(x)$), where values of$\boldsymbol{X}_{\text{low}}$ label the columns, values of $\boldsymbol{X}_{\text{high}}$ 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 $x_{\text{low}}$ and $x_{\text{high}}$, 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 $\boldsymbol{X}_{\text{low}}$ and $\boldsymbol{X}_{\text{high}}$. 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 $\lceil \log_2 p \rceil$.

As shown in Fig. 3, the outputs of each cell in the LUT cascade are partitioned into two parts, those that drive the next cell and those that are part of the segment number. For some cells, there may be *no* outputs that are part of the segment number. Indeed, our experience is that leftmost cells tend *not* to produce segment number bits, and most such outputs come only the right cells. In the example we describe in Section 5, *all* segment number bits come from the single rightmost cell.

# 4 Segmentation Algorithm

Our approach to segmentation 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 4. 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 $b)$ by restricting the largest value of $x$ to be $b - 2^{\alpha}$, where $\alpha$ is the contribution of the least significant bit.

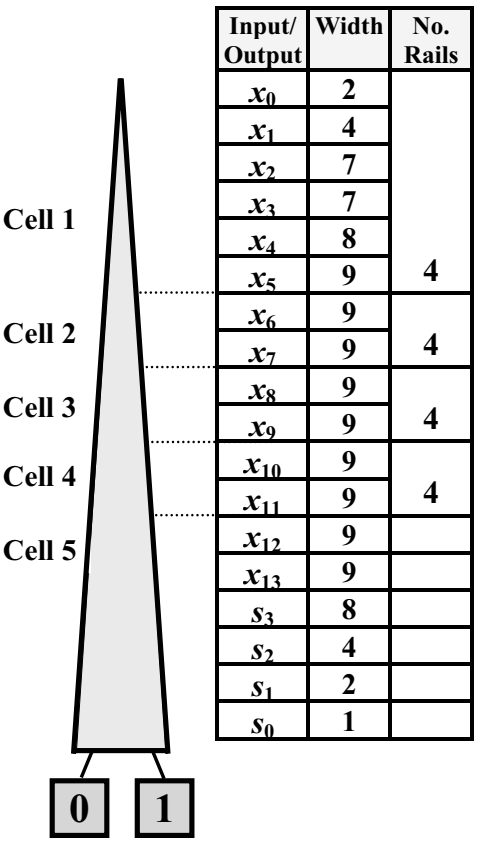


Figure 4: BDD for the Segment Index Encoder for the $cos(x)$ Function.

## 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 implemented.

For the $cos(x)$ function, the input $\boldsymbol{X}$ has 16 variables and represents $x$ to a precision of $2^{-16} \simeq 1.5 \times 10^{-5}$. Using the Douglas-Peucker algorithm, we determined a 9-element segmentation as shown in Table 3

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 labelled $y_k$ down variables lower in the BDD, where edges incident to the same lower variable counted as 1. An order top-to-bottom that produced small widths is $x_0$, $x_1$, ... $x_{13}$, $s_3$, $s_2$, $s_1$, and $s_0$. Note that only the end points of Table 3 need be used and, for these points, the two most significant bits are always 0. Therefore, only 14 bits ($x_0$, $x_1$, ... and $x_{13}$) of $X$ are used.

Note that the width never exceeds 9. Thus, from Theorem 3.2, any partition yields a LUT cascade with at most 4 rails. The third column of Fig. 4 shows a

Table 3: Segmentation for the $cos(x)$ Function.

| Segment Begin Point in Decimal | Segment Begin Point in Binary | Segment End Point in Decimal | Segment End Point in Binary | Segment Number |
|---|---|---|---|---|
| 0.000000 | 0.000 0000 0000 0000 | 0.053314 | 0.000 0110 1101 0011 | 0000 |
| 0.053345 | 0.000 0110 1101 0100 | 0.107300 | 0.000 1101 1011 1100 | 0001 |
| 0.107330 | 0.000 1101 1011 1101 | 0.162994 | 0.001 0100 1101 1101 | 0010 |
| 0.163025 | 0.001 0100 1101 1110 | 0.219696 | 0.001 1100 0001 1111 | 0011 |
| 0.219727 | 0.001 1100 0010 0000 | 0.277740 | 0.010 0011 1000 1101 | 0100 |
| 0.277771 | 0.010 0011 1000 1110 | 0.307800 | 0.010 0111 0110 0110 | 0101 |
| 0.307831 | 0.010 0111 0110 0111 | 0.339386 | 0.010 1011 0111 0001 | 0110 |
| 0.339417 | 0.010 1011 0111 0010 | 0.406799 | 0.011 0100 0001 0010 | 0111 |
| 0.406830 | 0.011 0100 0001 0011 | 0.500000 | 0.100 0000 0000 0000 | 1000 |

repeated partitioning that yields four instances of a set of 4 rails. These separate 5 cells in the cascade that are used to realize the given function. Each 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 on the basis of the number of bits of memory required. 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 5 cells each with a memory of $4 \times 2^6 = 256$ bits for a total of $5 \times 256 = 1280$ bits. The coefficients memory has a 4 bit address and stores 9 segment coefficient pairs. The two coefficients, $c_1$ and $c_0$, are each represented in 10 bits, for a total of $9 \times (10 + 10) = 180$ bits. Totalling the LUT cascade and coefficients memory yields $1280 + 180 = 1460$ 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. The memory reduction is significant, slightly more than 1/300 of the memory size for the naive method!

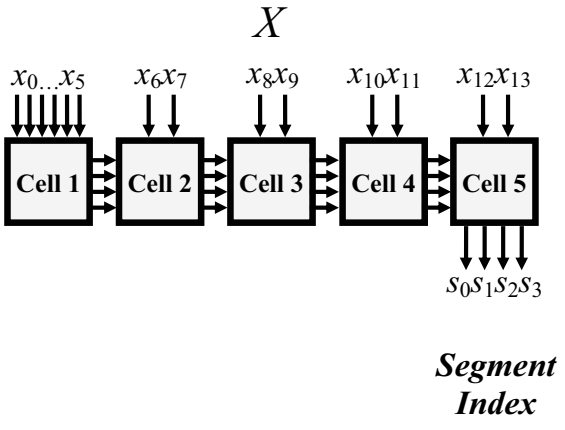


Figure 5: Segment Index Encoder for the $cos(x)$ Function Realized by an LUT Cascade.

## 5.2 Summary of Memory Requirements For Numerical Functions

Table 4 summarizes the results of the design process just described. This shows that the sizes across the various functions are small. They range from less than 100 bits to approximately 4000 bits. In forming the LUT cascades, we did not minimize the memory. For some functions, the minimum memory corresponded to a cascade with 9 cells.

In Table 4, the functions listed all have an input

value for $x$ of 16 bits. However, over the range of values specified in Table 4, the most significant bit is constant, and, in the case of the $tan(\pi x)$ function, the most significant two bits are constant. Thus, in comparing with the naive method, one must consider a memory of size $2^{15}$ or $2^{14}$, as appropriate.

There is some correlation between the total memory size, as shown in the rightmost column, and the number of segments, as shown in the fourth column from the left. Enlarging the domain increases the number of segments and thus the memory size. Besides the domain size, the memory size is dependent on the function realized. For example, the Gaussian distribution (last line of Table 4) has surprisingly low memory requirements.

# 6 Summary and Conclusions

We have shown a design method for circuits that computes elementary and non-elementary functions quickly and accurately. 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 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.

The previous approach [6, 7] used an ad hoc circuit to generate segmentation. So, such a method is only useful for a limited class of functions. However, our approach uses an LUT cascade, a universal circuit that generates optimized segmentation for wider classes of functions.

Extensions of this work include the use of: 1. a scaling factor (shifter) for functions with a large dynamic range, 2. a higher-order approximations ($\sum_{i=0}^{m} c_i x^i$) to reduce the approximation error in the segment, and 3. improved segmentation algorithm.

## 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 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.

Table 4: Comparison of Sizes of Memory For Various Functions, Where $x$ and $f(x)$ are Realized in 16 and 8 Bits.

| **Function** | **Interval** | | **#** | **# Bits** | | **# Cell Inputs** | | | | | | **LUT** | **Coef** | **Total** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $f(x)$ | $x$ | $f(x)$ | **Seg** | $c_1$ | $c_0$ | **# Cell Outputs** | | | | | | **Mem** | **Mem** | **Mem** |
| $2^x$ | [0,1] | [1,2] | 7 | 10 | 10 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 576 | 140 | 716 |
| $1/x$ | [1,2) | ($\frac{1}{2}$,1] | 8 | 10 | 10 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 576 | 160 | 736 |
| $\sqrt{x}$ | [$\frac{1}{32}$,2) | [$\frac{1}{\sqrt{32}}$, $\sqrt{2}$) | 18 | 16 | 15 | 7<br>5 | 7<br>5 | 7<br>5 | 7<br>5 | 6<br>5 | | 2900 | 558 | 3458 |
| $1/\sqrt{x}$ | [1,2) | ($\frac{1}{\sqrt{2}}$, 1] | 4 | 10 | 10 | 5<br>2 | 5<br>2 | 5<br>2 | 5<br>2 | 3<br>2 | | 268 | 80 | 348 |
| $\log_2(x)$ | [1,2) | [0, 1) | 5 | 10 | 10 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 576 | 100 | 676 |
| $\ln x$ | [1,2) | [0,ln 2) | 7 | 10 | 10 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 576 | 140 | 716 |
| $sin(\pi x)$ | [0, $\frac{1}{2}$] | [0, 1] | 9 | 10 | 10 | 6<br>4 | 6<br>4 | 6<br>4 | 6<br>4 | 6<br>4 | | 1280 | 180 | 1460 |
| $cos(\pi x)$ | [0, $\frac{1}{2}$] | [0, 1] | 9 | 10 | 10 | 6<br>4 | 6<br>4 | 6<br>4 | 6<br>4 | 6<br>4 | | 1280 | 180 | 1460 |
| $tan(\pi x)$ | [0, $\frac{1}{4}$] | [0, 1] | 8 | 10 | 10 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | | 480 | 160 | 640 |
| $\sqrt{-\ln x}$ | [ $\frac{1}{32}$,1) | (0, $\sqrt{5 \ln 2}$] | 26 | 16 | 14 | 7<br>5 | 7<br>5 | 7<br>5 | 7<br>5 | 7<br>5 | | 3200 | 806 | 4006 |
| $tan^2(\pi x)+1$ | [0, $\frac{1}{4}$] | [1,2] | 16 | 11 | 10 | 6<br>4 | 6<br>4 | 6<br>4 | 6<br>4 | 5<br>4 | | 1152 | 336 | 1488 |
| $x \log_2 x - (1-x) \log_2(1-x)$ | [$\frac{1}{256}$, $\frac{255}{256}$] | (0,1) | 28 | 12 | 12 | 7<br>5 | 7<br>5 | 7<br>5 | 7<br>5 | 7<br>5 | | 3200 | 192 | 3392 |
| $\frac{1}{1+e^{-4x}}$ | [0,1] | [$\frac{1}{2}$, $\frac{1}{1+e^{-4}}$] | 8 | 10 | 10 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | 5<br>3 | | 480 | 160 | 640 |
| $\frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$ | [0, $\frac{1}{2}$] | [$\frac{1}{\sqrt{2\pi}}$, $\frac{1}{\sqrt{2\pi} e^{\frac{1}{8}}}$] | 2 | 10 | 10 | 5<br>1 | 5<br>1 | 4<br>1 | | | | 28 | 40 | 68 |


[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.