2007-06

Numerical function generators using LUT cascades

Sasao, Tsutomu

T. Sasao, S. Nagayama and J. T. Butler, "Numerical function generators using LUT cascades," IEEE Transactions on Computers, Vol.56, No.6, June 2007, pp.826-838.

Numerical Function Generators Using LUT Cascades

Tsutomu Sasao, *Fellow*, *IEEE*, Shinobu Nagayama, *Member*, *IEEE*, and
Jon T. Butler, *Fellow*, *IEEE*

Abstract—This paper proposes an architecture and a synthesis method for high-speed computation of fixed-point numerical functions such as trigonometric, logarithmic, sigmoidal, square root, and combinations of these functions. Our architecture is based on the lookup table (LUT) cascade, which results in a significant reduction in circuit complexity compared to traditional approaches. This is suitable for automatic synthesis and we show a synthesis method that converts a Matlab-like specification into an LUT cascade design. Experimental results show the efficiency of our approach as implemented on a field-programmable gate array (FPGA).

Index Terms—LUT cascades, numerical function generators (NFGs), nonuniform segmentation, automatic synthesis, FPGA implementation.

1 Introduction

Numerical functions such as trigonometric, logarithmic, square root, reciprocal, and combinations of these functions are extensively used in computer graphics, digital signal processing, communication systems, robotics, astrophysics, fluid physics, and so forth. To compute elementary functions, iterative algorithms such as the COordinate Rotation DIgital Computer (CORDIC) algorithm [1], [35] have often been used. Although the CORDIC algorithm achieves accuracy with compact hardware, its computation time is proportional to the precision (that is, the number of bits) [34]. For a function composed of elementary functions, the CORDIC algorithm is slower since it has to compute each elementary function sequentially. It is too slow for numerically intensive applications.

Implementation by a single lookup table (LUT) for a numerical function $f(x)$ is simple and fast. For low-precision computations of $f(x)$ (for example, x and $f(x)$ have 8 bits), this implementation is efficient since the size of the LUT is small. For high-precision computations, however, the single LUT implementation is impractical due to the huge table size.

To reduce the memory size, polynomial approximations have been used [4], [6], [13], [14], [19], [30], [31], [32], [33], [36]. These methods approximate the given numerical functions by piecewise polynomials and realize the polynomials with hardware. Linear or quadratic approximations offer fast and relatively high-precision evaluation of numerical functions. However, most existing methods are intended only for standard elementary functions and no systematic method for arbitrary functions exists. This paper considers fast and compact numerical function generators (NFGs) for arbitrary functions and proposes an architecture and a systematic synthesis method for them. Our architecture and synthesis method use linear approximations and are applicable not only to continuous functions but also to noncontinuous functions. The circuit's compactness is due, in large part, to the LUT cascade [12], [25], [26]. Our synthesis method is automated so that fast and compact NFGs can be produced by nonexperts.

Fig. 1 shows our synthesis system. The user specifies only the numerical function $f(x)$, the domain over x, and accuracies for input x and output $f(x)$. Our system can accept an arbitrary numerical function expressed either algebraically (for example, $\sin(\pi x)$) or as a table of input/output values. The user defines the numerical function by using the syntax of Scilab [29], a free Matlab-like numerical software. This is applied directly to our system, along with the domain and accuracy. The user can either use a defined function in Scilab or specify it directly. Note that, by changing the parser of our system, any format can be used for the design entry.

First, our system partitions the given domain over x into segments and generates the begin/end points of all the segments. Next, an error analysis is performed to determine the precision of the various internal computations, as well as the word width of the memory needed to store the various coefficients. The smallest internal precision needed to achieve the (external) accuracy specified by the user is chosen. Finally, the Hardware Description Language (HDL) code is generated, which is then applied to a vendor-supplied tool that produces the field-programmable gate array (FPGA) implementation.

This paper is organized as follows: Section 2 introduces terminology. Section 3 discusses a piecewise linear approximation for numerical functions and proposes a nonuniform

- *T. Sasao is with the Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka-shi, Fukuoka-ken, 820-8502, Japan. E-mail: sasao@cse.kyutech.ac.jp.*
- *S. Nagayama is with the Department of Computer Engineering, Hiroshima City University, Hiroshima-shi, Hiroshima-ken, 731-3194, Japan. E-mail: s_naga@cs.hiroshima-cu.ac.jp.*
- *J.T. Butler is with the Department of Electrical and Computer Engineering, Naval Postgraduate School, Code EC/Bu, Monterey, CA 93943-5121. E-mail: Jon_Butler@msn.com.*

Manuscript received 1 Feb. 2006; revised 23 Oct. 2006; accepted 9 Jan. 2007; published online 20 Feb. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0034-0206.
Digital Object Identifier no. 10.1109/TC.2007.1033.

0018-9340/07/$25.00 © 2007 IEEE Published by the IEEE Computer Society

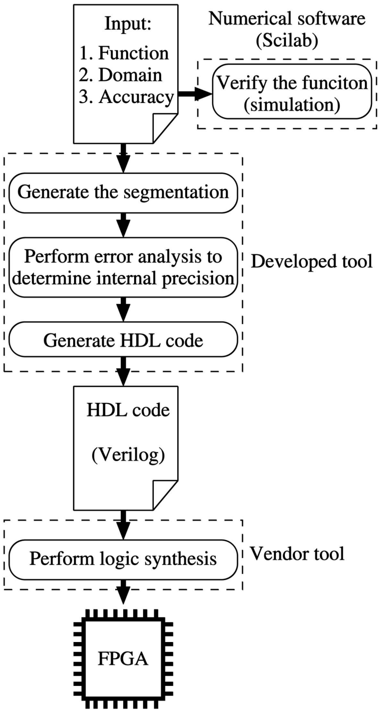


Fig. 1. Synthesis system for NFGs.

segmentation algorithm. Section 4 shows two different architectures for NFGs. Section 5 presents an example NFG design. Section 6 evaluates the performance of our architecture and synthesis system. The error analysis for NFGs is discussed in the Appendix.

2 Preliminaries

2.1 Number Representation and Precision

Definition 1. *The* **binary fixed-point representation** *of a value r has the form*

$$(d_{l-1}\, d_{l-2}\, \ldots\, d_1\, d_0.\, d_{-1}\, d_{-2}\, \ldots\, d_{-m})_2,$$

where $d_i \in \{0,1\}$, l is the number of bits for the integer part, and m is the number of bits for the fractional part of r. This representation is two's complement and, so,

$$r = -2^{l-1} d_{l-1} + \sum_{i=-m}^{l-2} 2^i d_i.$$

Definition 2. Error *is the absolute difference between the exact value and the value produced by the hardware.* **Approximation error** *is the error caused by a function approximation.* **Rounding error** *is the error caused by a binary fixed-point representation. It is the result of truncation or rounding, whichever is applied. However, both operations yield an error that is called rounding error.* **Acceptable error** *is the maximum error that an NFG may assume.* **Acceptable approximation error** *is the maximum approximation error that a function approximation may assume.*

Definition 3. Precision *is the total number of bits for a binary fixed-point representation. Specially, n-***bit precision** *specifies that n bits are used to represent the number, that is, $n = l + m$. An n-***bit precision NFG** *has an n-bit input.*

Definition 4. Accuracy *is the number of bits in the fractional part of a binary fixed-point representation. Specially, m-***bit accuracy** *specifies that m bits are used to represent the fractional part of the number. In this paper, an m-***bit accuracy NFG** *is an NFG with an m-bit fractional part of the input, an m-bit fractional part of the output, and a 2^{-m} acceptable error.*

Definition 5. Truncation *is the process of removing lower order bits from a binary fixed-point number. If r is represented as $r = (d_{l-1}\, d_{l-2}\, \ldots\, d_0.\, d_{-1}\, \ldots\, d_{-m}\, \ldots\, d_{-q})_2$ and is truncated to $r' = (d_{l-1}\, d_{l-2}\, \ldots\, d_0.\, d_{-1}\, \ldots\, d_{-m})_2$, then the resulting rounding error is at most $2^{-m} - 2^{-q}$, where $m \leq q$. Truncation never produces a value larger than r.*

Definition 6. Rounding *is a truncation if $d_{-m-1} = 0$. If $d_{-m-1} = 1$, then 2^{-m} is added to the result of the truncation. That is, if r is represented as*

$$r = (d_{l-1}\, d_{l-2}\, \ldots\, d_0.\, d_{-1}\, \ldots\, d_{-m}\, d_{-m-1}\, \ldots\, d_{-q})_2,$$

then r is rounded to

$$r' = (d_{l-1}\, d_{l-2}\, \ldots\, d_0.\, d_{-1}\, \ldots\, d_{-m})_2 + d_{-m-1} 2^{-m}.$$

The error caused by rounding is at most $2^{-(m+1)}$.

Truncation can cause a larger error than rounding. On the other hand, truncation requires less hardware. In our architecture, we use both truncation and rounding. This is discussed in the Appendix.

2.2 Binary Decision Diagram (BDD)

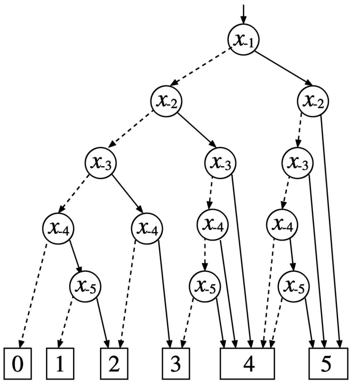
Definition 7. *A binary decision diagram (BDD) [2] is a rooted directed acyclic graph representing a logic function $\{0,1\}^n \rightarrow \{0,1\}$. The BDD is obtained by repeatedly applying the Shannon expansion to the logic function. It consists of two terminal nodes representing function values 0 and 1, respectively, and nonterminal nodes representing input variables. Each nonterminal node has two outgoing edges, namely, 0-edge and 1-edge, that correspond to the values of input variables.*

Definition 8. *A* **multiterminal BDD (MTBDD)** *[5] is an extension of the BDD and represents an integer function $\{0,1\}^n \rightarrow Z$, where Z is a set of integers. The MTBDD has multiple terminal nodes representing integer values.*

Example 1. Fig. 2a shows an integer function of five variables. It represents a segment index function, which will be explained in Section 4.2. Fig. 2b shows an MTBDD for the integer function. In Fig. 2b, dashed lines and solid lines denote 0-edges and 1-edges, respectively. In the MTBDD, terminal nodes represent function values. Thus, to evaluate the function, we traverse the MTBDD from the root node to a terminal node according to the input values and obtain the function value (an integer) at a terminal node. This MTBDD will be used for the design of the segment index encoder, as discussed in Section 4.2.

x_{-1}	x_{-2}	x_{-3}	x_{-4}	x_{-5}	f
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	1
0	0	0	1	1	2
0	0	1	0	0	2
0	0	1	0	1	2
0	0	1	1	0	3
		⋮			⋮
0	1	0	0	0	3
0	1	0	0	1	4
		⋮			⋮
1	0	0	1	0	4
1	0	0	1	1	5
		⋮			⋮
1	1	1	1	1	5

(a)



(b)

Fig. 2. MTBDD for an integer function. (a) Function table. (b) MTBDD.

3 Piecewise Linear Approximation Based on Nonuniform Segmentation

3.1 Segmentation Problem

A straightforward method to realize a given function $f(x)$ is to use a single memory in which the address is the binary representation of the value of x and the content of that address is the corresponding value of $f(x)$. However, with this simple approach, the memory size can be very large. For example, if x and $f(x)$ are represented by 24-bit binary numbers, then $2^{24} \times 24 = 48$ Mbytes of memory is needed. In addition, memory is not used efficiently since adjacent locations contain nearly the same value of $f(x)$.

A more efficient realization takes advantage of the last observation. Divide a domain over x into segments and realize $f(x)$ in each segment as the linear function $c_1 x + c_0$. In this way, the number of memory locations is the number of segments. Although it is now necessary to store two numbers, c_1 and c_0, the total memory needed is typically much lower.

Many existing methods [4], [6], [13], [14], [19], [30], [31], [32], [33], [36] use *uniform segmentation*, which divides a domain over x into segments with the same size. In such a segmentation, the segment size is determined by the least significant bits of x. In this case, the most significant bits can be used to specify a segment number, which is used to find a memory location that stores c_1 and c_0. The number of uniform segments is determined by the smallest segment size needed to achieve the specified approximation error. One then must choose the size of all segments to be the same as this smallest size. This can yield too many segments, depending on the function. Since the memory size depends on the number of segments, one seeks a segmentation with the fewest segments.

A more sophisticated approach is to choose all segment sizes to be as large as possible while maintaining the specified approximation accuracy. In this case, the segments are likely to have different widths. Cantoni [3] has proposed an algorithm to approximate a given function by using *nonuniform segments*. This introduces a problem of designing an additional circuit that maps values of x to a segment number. Potentially, this is a complex circuit.

To simplify the additional circuit, Lee et al. [16] have proposed a *special nonuniform segmentation* called the *hierarchical segmentation scheme*. The hierarchical segmentation partitions a domain into nonuniform segments by using one of four segmentation types, {P2S(US), P2SL(US), P2SR(US), US(US)}. Since these nonuniform segmentations can be realized by simple circuits, the hierarchical segmentation method results in fewer segments, as well as faster and more compact NFGs, than produced by existing uniform segmentation methods. However, in their method, the designer has to choose the segmentation type and segment widths for the given function by trial and error.

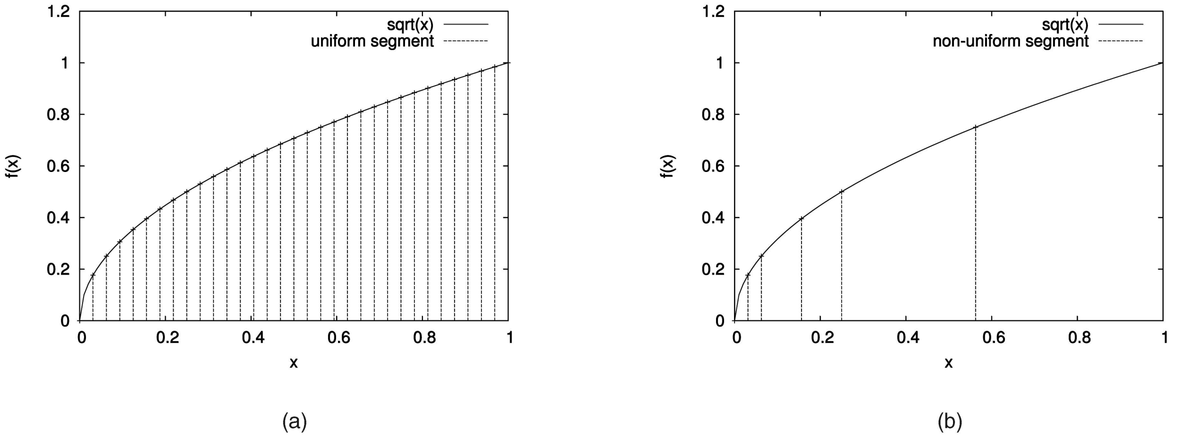
We seek a method in which the optimum segmentation is generated automatically, with no input from the user. It was shown in [26] that an LUT cascade can compactly realize *any nonuniform segmentation* in which the memory size depends on the number of segments. As a result, for fast and compact NFG design, we can use any nonuniform segmentation algorithm, such as Cantoni's algorithm [3], and can design the nonuniform segmentation circuit for the given function.

Example 2. Fig. 3 shows uniform and nonuniform segmentations of $\sqrt{x}$, where x has a 5-bit accuracy and the acceptable approximation error is 2^{-7}. The number of uniform segments is 32, whereas the number of nonuniform segments is 6.

Note that the segmentation shown in Fig. 3b is identical to the segmentation represented by the table and MTBDD shown in Fig. 2.

3.2 Nonuniform Segmentation Algorithm

The algorithms proposed by Cantoni [3] find the piecewise linear functions with the minimum approximation error for *the given number of segments*. However, we want to find the fewest segments to approximate a given function for *a given approximation error*. Thus, we use another algorithm. Fig. 4 presents a heuristic nonuniform segmentation algorithm based on the Douglas-Peucker algorithm, which has been used in rendering curves for graphics displays [9]. Since this is a dedicated algorithm for piecewise linear approximation, this cannot be directly extended to higher order polynomial approximations. However, since this algorithm is simple and robust, it can be applied to a wide range of functions (even noncontinuous functions) and it is fast. The inputs of the segmentation algorithm are a numerical function $f(x)$, a domain $[a, b]$ over x, an accuracy m_{in} of x, and an acceptable approximation error ε_a. Note that ε_a has to be smaller than $2^{-(m_{out}+1)}$ to produce an m_{out}-bit accuracy NFG because the total rounding error is larger than $2^{-(m_{out}+1)}$ (see the Appendix). In our system, ε_a is set to $2^{-(m_{out}+2)}$ by default, but the user can choose any $\varepsilon_a < 2^{-(m_{out}+1)}$. This algorithm begins by forming one segment over the whole domain $[a, b]$. This is an initial piecewise approximation by a linear function whose end points are $(a, f(a))$ and $(b, f(b))$. If the current segment fails to provide an acceptable approximation error, then it is partitioned into two segments joined at a point $(p, f(p))$



(a)

(b)

Fig. 3. (a) Uniform and (b) nonuniform segmentations of $\sqrt{x}$.

```
Input:    Numerical function $f(x)$, domain $[a,b]$ over $x$, accuracy $m_{in}$ of $x$, and acceptable approximation error $\varepsilon_a$.
Output:   Segments $[s_0,e_0],[s_1,e_1],\ldots,[s_{t-1},e_{t-1}]$, and correction values $v_0,v_1,\ldots,v_{t-1}$

Process:  Recursively compute segments. Set the current segment to $[a,b]$.
   1.     Approximate $f(x)$ in the current segment $[s,e]$ by the linear function $g(x)=c_1x+c_0$, where $c_1=\frac{f(e)-f(s)}{e-s}$ and
          $c_0=f(s)-c_1s$.
   2.     Find a value $p_{max}$ of the variable $x$ that maximizes $f(x)-g(x)$ in $[s,e]$, and let $max_{fg}=f(p_{max})-g(p_{max})$,
          $(max_{fg}\geq 0)$, where $p_{max}$ has $m_{in}$-bit accuracy.
   3.     Similarly, find a $p_{min}$ that minimizes $f(x)-g(x)$, and let $min_{fg}=f(p_{min})-g(p_{min})$, $(min_{fg}\leq 0)$, where $p_{min}$
          has $m_{in}$-bit accuracy.
   4.     Let $p=p_{max}$ if $|max_{fg}|>|min_{fg}|$, and let $p=p_{min}$ otherwise.
   5.     Let $error=|max_{fg}-min_{fg}|/2$, and $v=(max_{fg}+min_{fg})/2$.
   6.     If $error\leq\varepsilon_a$, then declare $[s,e]$ to be a complete segment. If all segments are complete, stop.
   7.     For any segment $[s,e]$ that is not complete, partition $[s,e]$ into two segments $[s,p]$ and $[p,e]$, declare each as the
          current segment, and go to 1.
```

Fig. 4. Nonuniform segmentation algorithm for the domain.

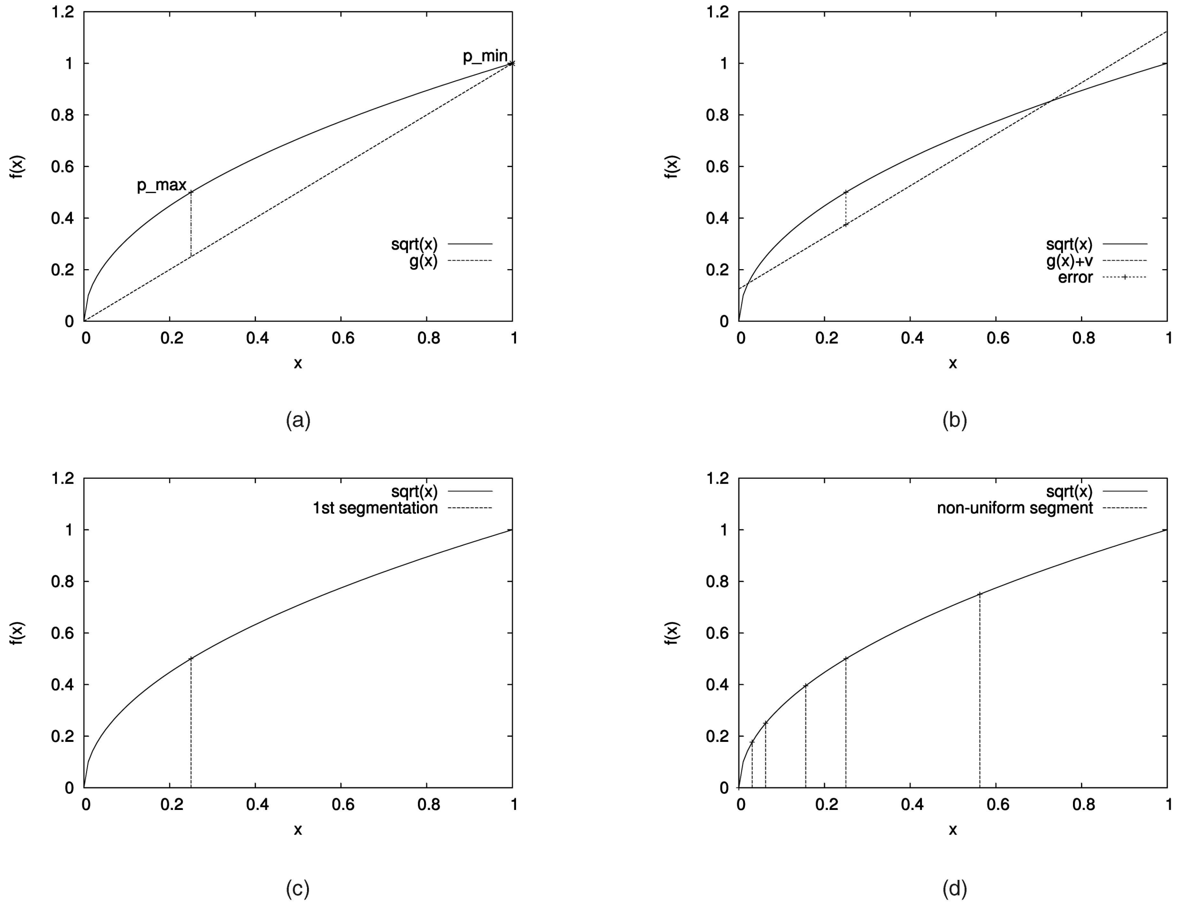
where the maximum error occurs. By iterating this process until the two subsegments approximate $f(x)$ to within the acceptable approximation error, this algorithm finds the nonuniform segmentation with a small number of segments. Note that this algorithm does not always produce the fewest segments because it is a heuristic. In the Douglas-Peucker algorithm, once a point is chosen, it never moves. For example, in the $\sin(\pi x)$ function for $0 \leq x \leq 1$, the point chosen after the two end points is $x = 1/2$. Therefore, if the optimum segmentation requires a segment with $x = 1/2$ internal to the segment, then the optimum segmentation is not achievable. However, the Douglas-Peucker algorithm will place many segments in regions where the function is nonlinear and fewer segments where it is close to linear. Therefore, it is close to optimum. The correction values v_i are used to reduce the approximation errors. In Fig. 4, max_{fg} and min_{fg} denote the maximum positive error and the maximum negative error, respectively. These errors are equalized by a vertical shift of linear function $g(x)$ with v_i. This vertical shift can produce minimax approximations when $f(x)$ is convex or concave on the segment $[s, e]$. Remez's algorithm is usually used to obtain minimax approximations [20]. However, our algorithm can produce minimax approximations without using Remez's algorithm because our algorithm is based on a linear approximation [21]. In Fig. 4, p_{max} and p_{min} can be found by scanning values of x over $[s, e]$. However, it is time consuming. We use a nonlinear programming algorithm [11] to find these values efficiently.

Example 3. Fig. 5 shows the segmentation process when the algorithm in Fig. 4 is applied to $\sqrt{x}$ over the domain [0, 1).

First, we compute a linear function $g(x)$ and find p_{max} and p_{min} (Fig. 5a). In this case, the entire function is approximated as a single line and p_{max} is the point causing the maximum error. In the next iteration of the algorithm, the function is approximated by two segments joined at p_{max}.

Then, we compute the correction value v and the approximation error (Fig. 5b). If the approximation error is larger than the given acceptable approximation error ε_a, then the segment is partitioned at the point p_{max} that causes the maximum error (Fig. 5c). Finally, the nonuniform segmentation is obtained by iterating this step recursively (Fig. 5d).

Example 4. For $\sqrt{-\ln(x)}$, where $2^{-32} \leq x \leq 1$, our algorithm produces 25 segments. It requires a coefficients table with 32 words. On the other hand, the nonuniform segmentation method of Lee et al. [17] produces 59 segments for $\sqrt{-\ln(x)}$. It requires a coefficients table with 64 words. For both approximations, the maximum approximation error is 0.020 and x has a 32-bit accuracy.



(a)

(b)

(c)

(d)

Fig. 5. Process of nonuniform segmentation algorithm. (a) Linear approximation with p_{max} and p_{min}. (b) Vertical shift of $g(x)$ and error. (c) First segmentation. (d) Nonuniform segmentation.

3.3 Computation of Approximated Values

For each segment $[s_i, e_i]$, the numerical function $f(x)$ is approximated by the corresponding linear function $g_i(x)$. That is, the approximated value y of $f(x)$ is computed as

$$y = g_i(x) = c_{1i}x + c_{0i}, \tag{1}$$

where

$$c_{1i} = \frac{f(e_i) - f(s_i)}{e_i - s_i} \quad \text{and} \quad c_{0i} = f(s_i) - c_{1i}s_i + v_i.$$

By substituting c_{0i} into (1) and, simplifying, we have

$$g_i(x) = c_{1i}(x - s_i) + f(s_i) + v_i. \tag{2}$$

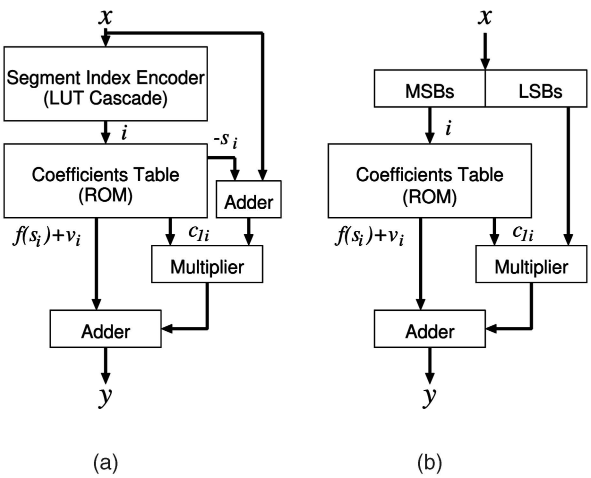
Note that, in this form of the piecewise linear approximation, x is offset by s_i, which is the start of the segment. Comparing (2) with (1) reveals that $(x - s_i)$ will be (often much) smaller than x. As a result, a smaller multiplier is needed to realize (2) than (1). This results in a lower multiplication delay.

When $f(x)$ is convex or concave on the segment $[s_i, e_i]$, this linear function is identical to the minimax approximation to $f(x)$ [21]. For many functions, there are few inflection points and, so, in most segments, the function is entirely concave or convex. Therefore, our piecewise linear approximation yields an accurate representation of the given function with few segments.

4 Architecture for NFGs

4.1 Architectures Based on Nonuniform and Uniform Segmentations

Fig. 6 shows two architectures for the NFGs realizing (2). Fig. 6a shows an architecture based on nonuniform segmentation. It uses five units: the segment index encoder that produces the segment index i for $[s_i, e_i]$, given the value x, the

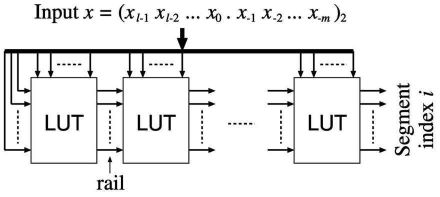


(a) (b)

Fig. 6. Two architectures for NFGs. (a) Architecture for nonuniform segmentation. (b) Architecture for uniform segmentation.

Interval	Index
$s_0 \le x \le e_0$	0
$s_1 < x \le e_1$	1
$\vdots$	$\vdots$
$s_{t-1} < x \le e_{t-1}$	$t-1$

(a)


Input $x = (x_{l-1}\, x_{l-2} \ldots x_0 \,.\, x_{-1}\, x_{-2} \ldots x_{-m})_2$
LUT
LUT
LUT
rail
Segment index i

(b)

Fig. 7. Segment index encoder. (a) Segment index function. (b) LUT cascade.

coefficients table for $-s_i$, c_{1i}, and $f(s_i) + v_i$, the adder for $x + (-s_i)$, the multiplier, and the output adder. Fig. 6b shows an architecture based on uniform segmentation. In uniform segmentation, the segment index i of $[s_i, e_i]$ and the value of $(x - s_i)$ can be obtained from the most significant bits and the least significant bits of x, respectively. Therefore, the architecture in Fig. 6b does not use the segment index encoder.

4.2 Architecture of Segment Index Encoder

Fig. 7a shows a segment index function

$$seg\_func(x) : \{0,1\}^n \rightarrow \{0, 1, \ldots, t-1\},$$

that is, the function realized by the segment index encoder, where x has n-bit precision and t denotes the number of segments. This is used in the case of nonuniform segmentation. It converts a value of x into a segment index, which is then applied to the address inputs of the coefficients memory. Fig. 7b shows the circuit. Each block labeled LUT is a combinational logic circuit. Thus, the complete circuit is combinational logic. For all but the leftmost LUT, part of the input comes from the output of another LUT and the other part of the input comes from x. For the leftmost LUT, all inputs come from x. The outputs of the rightmost LUT are the segment index encoder outputs and they encode the segment index. These outputs and all connections between LUTs are called *rails*. With a fixed number of input lines representing x, the number of rails determines the complexity of this circuit. It was shown in [26] that the LUT cascade in Fig. 7b has reasonable complexity. Specifically, [26] shows the following lemma:

Lemma 1. *Let $seg\_func(x)$ be a segment index function with t segments. Then, there exists an LUT cascade for $seg\_func(x)$ with at most $\lceil \log_2 t \rceil$ rails.*

From Lemma 1, we have the following theorem:

Theorem 1. *Consider a segment index function $seg\_func(x) : \{0,1\}^n \rightarrow \{0, 1, \ldots, t-1\}$, where n is the precision of x and t is the number of segments. Then, $seg\_func(x)$ can be implemented by an LUT cascade, as shown in Fig. 7, where each LUT has $k+2$ inputs and k outputs (rails), the number of LUTs is $\lceil \frac{n-k}{2} \rceil$, the total memory size is $2^{k+1}k(n-k)$ bits, and $k = \lceil \log_2 t \rceil$.*

Proof. By Lemma 1, the number of the rails of the LUT cascade is at most $k = \lceil \log_2 t \rceil$. In addition, we can easily see that the LUT cascade consists of $\lceil \frac{n-k}{2} \rceil$ LUTs, where each LUT has $k+2$ inputs and k outputs. When $(n-k)$ is an even number, the total memory size of the LUT cascade is

$$2^{k+2} \times k \times \frac{n-k}{2} = 2^{k+1}k(n-k) \text{ bits.}$$

When $(n-k)$ is an odd number, the memory size of the rightmost LUT is $2^{k+1} \times k$ bits. Thus, the total memory size of the LUT cascade is

$$2^{k+2} \times k \times \frac{n-k-1}{2} + 2^{k+1} \times k = 2^{k+1}k(n-k) \text{ bits.}$$

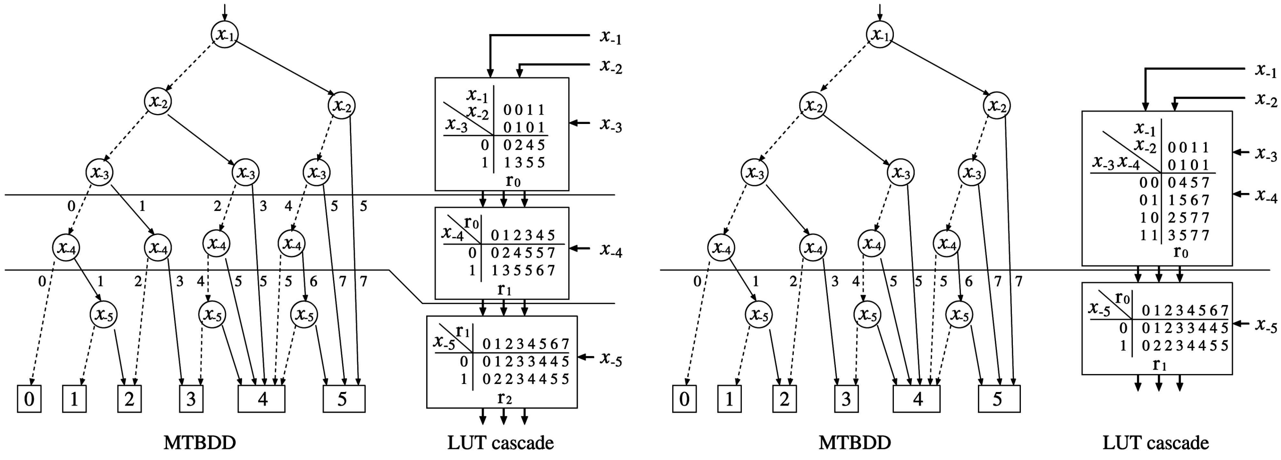
Hence, we have the theorem. □

The above theorem shows that the memory size of the LUT cascade depends only on the precision n and the number of segments t. Thus, an approximation with fewer segments requires a smaller segment index encoder. We can use Theorem 1 to estimate the memory size of the segment index encoder from n and t. This will be shown in Section 4.4.

The LUT cascade is obtained by functional decomposition using an MTBDD for $seg\_func(x)$ [12], [25], [26]. Our synthesis system first finds a nonuniform segmentation, then generates the MTBDD, and, finally, decomposes it to find the LUT cascade.

Example 5. Consider the segment index function in Fig. 2a. Let the binary fixed-point representation of x be $(.\, x_{-1}\, x_{-2}\, x_{-3}\, x_{-4}\, x_{-5})_2$. Fig. 2b is the corresponding MTBDD. By decomposing the MTBDD in Fig. 2, we obtain two different LUT cascades in Fig. 8. Fig. 8 illustrates the relationship between each LUT and decompositions of the MTBDD. In these figures, the "r_i" in each LUT denotes the rails that represent subfunctions in the MTBDD. In the MTBDD, numbers assigned to edges that cut across the horizontal lines represent subfunctions. The LUT cascade in Fig. 8a requires a memory size of $2^3 \times 3 + 2^4 \times 3 + 2^4 \times 3 = 120$ bits and three LUTs. On the other hand, the LUT cascade in Fig. 8b requires a memory size of $2^4 \times 3 + 2^4 \times 3 = 96$ bits and two LUTs. Note that the best realization is the LUT cascade consisting of a single LUT. As shown in Theorem 1, since $k = \lceil \log_2 6 \rceil = 3$, there exists an LUT cascade consisting of a single LUT with $3+2$ inputs and 3 outputs. Its memory size is $2^5 \times 3 = 96$ bits.

As shown in Example 5, the memory size and the number of LUTs in an LUT cascade also depend on the decomposition and the variable order of the MTBDD. To find the best LUT cascade, we use an optimization algorithm for heterogeneous multivalued decision diagrams (MDDs) [22], [23]. In an FPGA implementation of an NFG, each LUT in an LUT cascade is implemented with



(a) (b)

Fig. 8. Example of LUT cascades, where $x = (.\ x_{-1}\ x_{-2}\ x_{-3}\ x_{-4}\ x_{-5})_2$. (a) Decomposition into three LUTs. (b) Decomposition into two LUTs.

a block RAM in an FPGA. In this case, our synthesis system decomposes an MTBDD so that each LUT has the same size as the available block RAM (for example, 4 or 18 Kbits). Note that this automatically produces a pipelined implementation of an LUT cascade since block RAMs in a modern FPGA are synchronous.

To the best of our knowledge, this is the first realization method for *any* general segment index function.

4.3 Size Reduction of Multiplier

The coefficients table in Fig. 6a has 2^k words, where $k = \lceil \log_2 t \rceil$, and t is the number of segments. FPGA technology restricts memory sizes to a power of 2. Therefore, if $t < 2^k$, then we can increase the number of segments up to $t = 2^k$, without increasing the memory size. From Lemma 1, the size of the LUT cascade also depends on the value of k rather than t. Thus, increasing the number of segments to $t = 2^k$ rarely increases the size of the LUT cascade. To reduce the multiplier size, we reduce the maximum value of $(x - s_i)$ (that is, the maximum width of segments $[s_i, e_i]$ that is defined as $(e_i - s_i)$) by dividing the widest segment into two equal-sized segments up to $t = 2^k$. This reduction of the segment width reduces the rounding error as well (see the Appendix).

To reduce the multiplier size further, we use the *scaling method* shown in [15]. We represent c_{1i} as $c_{1i} = c_{1i} \times 2^{-h_i} \times 2^{h_i}$ and compute $c_{1i}(x - s_i)$ by using the multiplier for the right-shifted multiplicand $c_{1i} \times 2^{-h_i}(x - s_i)$ and a left shifter to restore the term. Applying a right shift reduces the number of bits for $c_{1i} \times 2^{-h_i}$ (that is, the multiplier size) by rounding the least significant h_i bits of c_{1i} while increasing the rounding error. In the Appendix, we find the largest h_i for each segment i that preserves the given acceptable error. When the largest h_is are 0 for all of the segments, we do not use the scaling method. For simplicity, Fig. 6a omits the scheme for the scaling method.

Note that these techniques utilizing the variation in segment sizes do not apply to the architecture based on uniform segmentation.

Example 6. Consider a 16-bit-precision NFG for $\sqrt{-\ln(x)}$ for $0 < x \leq 1$ and its FPGA implementation with the Altera Stratix EP1S20F484C5. By using the scaling method, c_{1i} and h_i have 16 and 3 bits, respectively, and they produce an NFG with 241 logic elements (LEs), two multipliers (digital signal processors (DSPs)), a memory size of 168,960 bits, an operating frequency of 208 MHz, and a delay of 38 ns. On the other hand, without the scaling method, c_{1i} has 22 bits and it produces an NFG with fewer LEs (153 LEs) but more DSPs (eight DSPs), a larger memory size (172,032 bits), a lower operating frequency (130 MHz), and a longer delay (54 ns).

4.4 Memory Size Estimate for Each Architecture

Our synthesis system estimates the memory size for each of the two architectures in Fig. 6 and generates the HDL code that implements the architecture with the smaller memory size. That is, our synthesis system chooses an architecture based on the *estimate* of the memory size. The estimate is discussed below.

4.4.1 Memory Size Estimate for Nonuniform Segmentation

The coefficients table in Fig. 6a has 2^k words, where $k = \lceil \log_2 t \rceil$, and t is the number of nonuniform segments obtained by the algorithm in Fig. 4. We assume that three coefficients, $-s_i$, c_{1i}, and $f(s_i) + v_i$, have m_{out} bits, respectively, where m_{out} is the output accuracy of NFG given by the specification. Then, the memory size of the coefficients table is estimated as

$$2^k \times 3m_{out} \text{ bits.} \tag{3}$$

We assume that when we generate an LUT cascade with the minimum memory size, each LUT in the LUT cascade has $k + 2$ inputs and k outputs (the number 2 of $k + 2$ is based on the results in [28]). Then, from Theorem 1, the memory size of the LUT cascade (segment index encoder) is estimated as

$$2^{k+1} \times k(n_{in} - k) \text{ bits,} \tag{4}$$

where n_{in} is the input precision of NFG given by the specification.

The total memory size of an NFG based on nonuniform segmentation is estimated as (3) + (4):

$$2^k \times \{3m_{out} + 2k(n_{in} - k)\} \text{ bits.}$$

4.4.2 Memory Size Estimate for Uniform Segmentation

First, we estimate the number of uniform segments by using the nonuniform segmentation results obtained by the algorithm in Fig. 4. Let $[a, b]$ and $[s_{min}, e_{min}]$ be the given domain and the narrowest segment produced by the algorithm, respectively. Then, the number of uniform segments t_u is estimated as

$$t_u = \left\lceil \frac{b - a + 2^{-m_{in}}}{2^w} \right\rceil,$$

where

$$w = \lfloor \log_2(e_{min} - s_{min} + 2^{-m_{in}}) \rfloor$$

and m_{in} is the input accuracy of the NFG given by the specification. Thus, the coefficients table in Fig. 6b has 2^{k_u} words, where $k_u = \lceil \log_2 t_u \rceil$. We assume that the coefficients have the same number of bits as in the nonuniform case. Then, the memory size of the coefficients table is estimated as

$$2^{k_u} \times 2m_{out} = 2^{k_u+1} m_{out} \text{ bits.}$$

This is the total memory size estimate of the NFG based on uniform segmentation. Note that, when $k_u = n_{in}$, the total memory size is estimated as a single LUT of size $2^{n_{in}} n_{out}$, where n_{out} is the output precision.

5 Example Design

In this section, we show in detail the design flow of a 5-bit precision (5-bit accuracy) NFG based on nonuniform segmentation for $\sqrt{x}$ for $0 \leq x < 1$. For simplicity, we do not use the multiplier reduction techniques shown in Section 4.3.

First, we partition the given domain $0 \leq x < 1$ into nonuniform segments by using the algorithm shown in Fig. 4. This produces six segments, as shown in Fig. 3b. Note that the acceptable approximation error is set to $2^{-(5+2)} = 2^{-7}$, as discussed in Section 3.2.

Then, we analyze the errors of the NFG and compute the number of bits needed to store coefficients by the method shown in the Appendix. Applying this analysis yields the following required number of bits: s_i, 4 bits; c_{1i}, 9 bits (3 integer bits and 6 fractional bits); and c_{0i}, 7 bits (7 fractional bits).

Finally, we design the NFG shown in Fig. 6a by using these required numbers of bits. From the segments produced by the algorithm in Fig. 4, we generate the MTBDD, as shown in Fig. 2. By decomposing the MTBDD as shown in Fig. 8b, we obtain the LUT cascade. Moreover, from the segments and correction values, we generate the linear functions and then the memory data for the coefficients table. Since the number of segments is six, the memory size of the coefficients table is

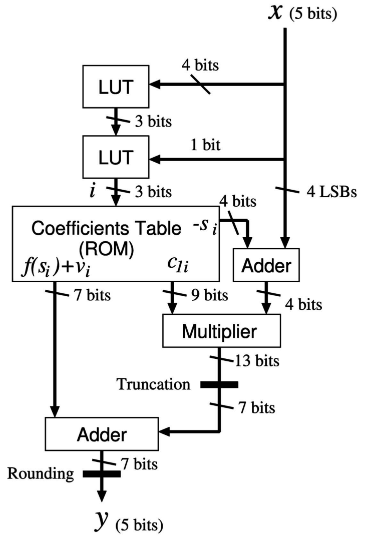


Fig. 9. Five-bit-precision NFG for $\sqrt{x}$.

$$2^{\lceil \log_2 6 \rceil} \times (4 + 9 + 7) = 2^3 \times 20 = 160 \text{ bits}$$

and the memory size of the LUT cascade is

$$2^4 \times 3 + 2^4 \times 3 = 96 \text{ bits.}$$

Thus, the total memory size of the NFG is 256 bits. Fig. 9 shows the generated NFG.

Our synthesis system automates this design flow.

6 Experimental Results

6.1 Comparison of the Number of Uniform and Nonuniform Segments

Table 1 compares uniform and nonuniform segmentations for two acceptable approximation errors: 2^{-17} and 2^{-25}. Eleven functions are shown. The last one, Matlab's "humps" function, is a quotient of polynomials:

$$\text{humps} = \frac{0.0004x + 0.0002}{x^4 - 1.96x^3 + 1.348x^2 - 0.378x + 0.0373}.$$

Table 1 shows that, for all functions, nonuniform segmentation yields fewer segments than uniform segmentation. Especially for logarithmic, square root, reciprocal, and their compound functions, the number of segments needed in nonuniform segmentation is only a few percent of the number of segments needed in uniform segmentation. However, for the $\sin(\pi x)$ and e^x functions, the additional segments needed in a uniform segmentation is not so large—only about twice that needed for a nonuniform segmentation. This suggests that uniform segmentation is a better choice for these functions. Many existing polynomial approximation methods are based on uniform segmentation. For the $\sin(\pi x)$ and e^x functions, the methods based on uniform segmentation require relatively few segments and, therefore, a small memory size. However, for logarithmic, square root, reciprocal, and their compound functions on the domains listed in Table 1, the uniform

TABLE 1
Numbers of Uniform and Nonuniform Segments

Function $f(x)$	Domain	Acceptable approx. error: 2^{-17} (x has 15-bit accuracy)					Acceptable approx. error: 2^{-25} (x has 23-bit accuracy)				
		No. of segments		Ratio	Time [msec]		No. of segments		Ratio	Time [msec]	
		Uniform	Non	[%]	Uniform	Non	Uniform	Non	[%]	Uniform	Non
e^x	$[0,1)$	256	128	50	10	10	4,096	2,048	50	150	60
$1/x$	$[1/32,1]$	31,745	1,721	5	460	70	507,905	28,010	6	7,570	1,040
$1/\sqrt{x}$	$[1/32,1]$	7,937	620	8	130	30	126,977	9,946	8	1,930	310
$\sqrt{x}$	$[0,1]$	32,769	231	1	460	10	8,388,609	3,941	0.05	123,030	120
$\ln(x)$	$[1/256,1]$	32,641	726	2	520	40	522,241	11,761	2	8,410	480
$x\ln(x)$	$(0,1)$	8,192	282	3	140	10	2,097,152	4,535	0.2	32,960	130
$\sqrt{-\ln(x)}$	$(0,1]$	32,768	584	2	620	40	8,388,608	12,089	0.1	161,400	1,710
$\sin(\pi x)$	$[0,1/2]$	257	127	49	10	10	4,097	2,027	49	140	70
$\arcsin(x)$	$[0,1]$	32,769	260	0.8	610	20	8,388,609	4,415	0.05	158,340	170
$\tan(\pi x)$	$[0,31/64]$	15,873	1,328	8	410	100	507,905	20,770	4	11,280	1,340
humps	$[0,1)$	4,096	910	22	270	110	65,536	14,735	22	3,570	1,270

Uniform: Uniform segmentation. Non: Non-uniform segmentation. Ratio: Non / Uniform × 100.
Time: CPU time for our segmentation algorithm conducted on the following environment.
System: Sun Blade 2500 (Silver), CPU: UltraSPARC-IIIi 1.6GHz, memory : 6GB, OS: Solaris 9, C compiler : gcc -O2.

TABLE 2
Memory Sizes (Bits) for Uniform and Nonuniform Segmentations

Function $f(x)$	16-bit precision (15-bit accuracy)					24-bit precision (23-bit accuracy)				
	Estimated mem. size		Actual mem. size		Ratio	Estimated mem. size		Actual mem. size		Ratio
	Uni.	Non	Uni.	Non	[%]	Uni.	Non	Uni.	Non	[%]
e^x	7,680	20,096	7,680	19,584	255	188,416	681,984	172,032	657,408	382
$1/x$	688,128	317,440	688,128	321,536	47	24,117,248	11,108,352	25,165,824	10,911,744	43
$1/\sqrt{x}$	245,760	168,960	253,952	164,864	65	6,029,312	5,718,016	5,767,168	5,521,408	96
$\sqrt{x}$	1,048,576	44,288	1,048,576	43,264	4	402,653,184	1,462,272	402,653,184	1,417,216	0.4
$\ln(x)$	622,592	168,960	622,592	167,936	27	24,117,248	5,718,016	22,544,384	5,586,944	25
$x\ln(x)$	245,760	78,336	204,800	75,264	37	96,468,992	2,695,168	69,206,016	2,588,672	4
$\sqrt{-\ln(x)}$	557,056	168,960	557,056	168,960	30	209,715,200	5,718,016	209,715,200	5,668,864	3
$\sin(\pi x)$	15,360	20,096	14,336	19,200	134	376,832	681,984	327,680	653,312	199
$\arcsin(x)$	1,048,576	87,552	1,048,576	84,992	8	402,653,184	2,908,160	402,653,184	2,818,048	0.7
$\tan(\pi x)$	327,680	227,328	327,680	225,280	69	24,117,248	9,142,272	23,592,960	8,847,360	38
humps	122,880	148,480	110,592	141,312	128	3,014,656	5,259,264	2,555,604	5,029,888	197

Uni.: Uniform segmentation. Non: Non-uniform segmentation.

method requires an excessive number of segments. For standard elementary functions such as $1/x$, $\ln(x)$, and $\sqrt{x}$, range reduction [20] is often used to translate the given domain into the domain suitable for uniform segmentation and to reduce the number of segments. Thus, the uniform method requires range reduction for these functions even if the given domain is a small range, as shown in Table 1. On the other hand, the nonuniform method approximates a wide range of functions with fewer segments without using range reduction. That is, our method requires no range reduction when the given domain is not large. This is useful for nonexperts who are unfamiliar with range reduction and for functions for which there is no known range reduction technique, as in the case of compound functions. In addition, Table 1 shows that the CPU time to compute the segmentation is strongly dependent on the number of segments. Therefore, a smaller acceptable approximation error requires more segments and longer CPU time. However, for all of the functions in Table 1, the CPU times needed to compute nonuniform segmentations are shorter than 2 sec when the acceptable approximation error is 2^{-25}.

These results show that, for various functions, our segmentation algorithm generates fewer nonuniform segments quickly and it is useful for automatic synthesis.

6.2 Memory Sizes for Two Architectures

Table 2 compares the memory sizes for the two architectures shown in Fig. 6. These correspond to nonuniform and uniform segmentations. In Table 2, the values under the column "Estimated mem. size" are obtained by the estimates derived in Section 4.4. As shown in Section 4.2, the memory size of the LUT cascade depends on the decomposition and variable order of the MTBDD. In this experiment, we used the optimum decomposition and variable order that minimize the memory size [22], [23].

Table 2 shows that, for logarithmic, square root, reciprocal, and their compound functions, the memory size for nonuniform segmentation is smaller than that for the uniform segmentation. On the other hand, for $\sin(\pi x)$ and e^x functions, the memory size for uniform segmentation is smaller than that for the nonuniform segmentation. As shown in Table 1, for $\sin(\pi x)$ and e^x functions, the difference between the numbers of uniform and nonuniform segments is not so large. Thus, for such functions, the architecture based on uniform segmentation requires a smaller memory size than that based on nonuniform segmentation because it needs no segment index encoder. In addition, Table 2 shows that the estimated memory sizes are accurate. Designing the two architectures for each function can take minutes of CPU time. To reduce the design time, our synthesis system first estimates the memory sizes, then selects the better architecture, and, finally, designs only the better one.

As mentioned in the previous section, for $1/x$, $\ln(x)$, and $\sqrt{x}$, the number of uniform segments can be reduced by using range reduction, thus reducing the memory size for

TABLE 3
FPGA Implementation of NFGs Based on Two Architectures

Precision, Accuracy:	16-bit precision, 15-bit accuracy
FPGA device:	Altera Stratix (EP1S20F484C5)
Logic synthesis tool:	Altera QuartusII 5.0 (speed optimization, timing requirement of 200MHz)

Function	Uniform segmentation					Non-uniform segmentation				
$f(x)$	#LEs	#DSPs	Freq. [MHz]	#stages	Delay [nsec]	#LEs	#DSPs	Freq. [MHz]	#stages	Delay [nsec]
e^x	89	2	193	3	16	155	2	196	8	41
$1/x$	122	0	–	1	–	259	2	207	8	39
$1/\sqrt{x}$	105	2	177	3	17	183	2	228	8	35
$\sqrt{x}$	194	0	–	1	–	298	2	212	9	42
$\ln(x)$	112	0	–	1	–	220	2	234	8	34
$x\ln(x)$	90	1	183	3	16	195	2	224	8	36
$\sqrt{-\ln(x)}$	102	0	–	1	–	241	2	208	8	38
$\sin(\pi x)$	85	2	192	3	16	149	2	179	8	45
$\arcsin(x)$	194	0	–	1	–	217	2	224	9	40
$\tan(\pi x)$	54	0	183	1	5	222	2	207	7	34
humps	77	2	181	3	17	116	2	174	7	40

–: NFGs cannot be mapped into the FPGA due to the excessive memory size.
#LEs: Number of logic elements. #DSPs: Number of 9-bit × 9-bit DSP units.
Freq. : Operating frequency. #stages : Number of pipeline stages.

TABLE 4
Comparison with Existing Methods

FPGA device:	Xilinx Virtex-II (XC2V4000-6)
Logic synthesis tool:	Xilinx ISE 8.2i (default option)

Function	Domain	In prec.		Out prec.		Our method				Method in [15]			
$f(x)$		Int	Frac	Int	Frac	Delay	Slices	RAMs	Mult.	Delay	Slices	RAMs	Mult.
$\sqrt{-\ln(x)}$	(0,1]	1	32	3	5	85	159	8	2	105			
$\sin(2\pi x)$	[0,1/4]	0	16	1	8	16	33	1	1	105	1,864	2	4
$\cos(2\pi x)$	[0,1/4]	0	16	1	8	17	34	1	1	105			
										Method in [16]			
$x\ln(x)$	(0,1)	0	16	0	16	59	98	6	1	85	483	1	4
humps	[0,1)	0	16	0	16	18	39	14	1	61	234	1	3

In prec.: Input precision. Out prec.: Output precision. Int: Number of integer bits.
Frac: Number of fractional bits. Delay: Total delay time [nsec]. Slices: Number of slices.
RAMs: Number of block RAMs. Mult.: Number of 18×18-bit multipliers.

uniform segmentation. Even in such cases (reduced domains), our memory size estimates are accurate and, so, our synthesis system accommodates range reduction.

6.3 FPGA Implementation

We implemented 16-bit-precision NFGs by using the Altera Stratix EP1S20F484C5 FPGA. Note that both of our architectures contain only combinational logic. For application-specific integrated circuit (ASIC) implementations, such a design yields the smallest delay. It represents one extreme in a range of NFG designs. However, in an FPGA implementation, this form does not always yield the smallest delay due to the interconnection delay. The pipelined design often achieves higher performance. In addition, modern FPGAs have *synchronous* block RAMs and this requires pipelined implementations. Thus, we adopt another extreme design, that is, a fully pipelined design. In this design, all elements, LUTs, adders, multipliers, and memory are buffered. Note that the delay of a small unit such as an adder is much smaller than a larger combinational logic circuit. Table 3 compares the FPGA implementation results of two architectures shown in Fig. 6. In this table, the "Delay" columns show the total delay time of each NFG from the input to the output in nanoseconds. These results correspond to the delays of combinational NFGs.

The NFGs based on uniform segmentation require fewer pipeline stages and have shorter delays than the nonuniform segmentation because they have no segment index encoder. However, for logarithmic, square root, and reciprocal functions, the NFGs based on uniform segmentation are not so easily implemented in an FPGA due to the excessive memory size. Table 3 shows that they cannot be mapped into the FPGA due to the insufficient memory blocks. Note that NFGs that have only one pipelined stage in Table 3 are realized with a single LUT due to the excessively many segments. On the other hand, for various functions, the NFGs based on nonuniform segmentation achieve a high operating frequency.

To show the efficiency of our automatic synthesis system, we compare our NFGs with the ones reported in [15] and [16]. NFGs in [15] are based on nonuniform segmentation and linear approximation and they are manually designed. Moreover, NFGs in [16] are not only also based on nonuniform segmentation, but also on *quadratic* approximation. We generated the NFGs with the same precision as in [15] and [16]. Note that, in [15], only one result for three functions is reported. Thus, the detailed result for each function is unavailable. Table 4 compares FPGA implementation results for the NFGs. In our NFGs, the segment index encoders are realized only by RAMs. Thus, our NFGs require more block RAMs than the NFGs in [15] and [16]. However, our NFGs require fewer slices and achieve a shorter delay time.

In [16], the 24-bit-precision NFGs for $x\ln(x)$ and the humps functions are implemented with the FPGA (XC2V4000) in Table 4. On the other hand, our 24-bit NFGs for those functions require a larger FPGA (Virtex-II Pro, XC2VP70) since our NFGs are based on linear approximation and require more block RAMs. Although NFGs based on linear approximation require a larger memory size than NFGs based on quadratic approximation, they are faster. This is because linear approximation requires only one multiplier [24].

7 Conclusion and Comments

We have proposed an architecture and a synthesis method for programmable NFGs for trigonometric functions, logarithm functions, square root, reciprocal, and so forth. Our architecture uses the LUT cascade to compactly realize various numerical functions and is suitable for automatic synthesis. To the best of our knowledge, this is the first architecture appropriate for *any* nonuniform segmentation. Experimental results show that our synthesis system 1) efficiently implements NFGs for wide range of numerical functions, 2) generates more suitable NFGs for the given functions between two architectures by using an accurate estimate of memory sizes, and 3) generates the NFGs with comparable performance to manually designed ones.

Since our NFGs are based on linear approximation, the practical precision for an FPGA implementation is up to 24 bits because of the memory size. Therefore, our method is useful to generate fast NFGs for a wide range of functions in up to 24-bit precision.

Appendix A

We analyze the error for our NFGs and show a method to obtain the appropriate bit sizes for the units in our architectures. Our synthesis system applies the error analysis method proposed in [8]. We show only the error analysis for the NFG using nonuniform segmentation in Fig. 6a. The analysis for the NFG using uniform segmentation in Fig. 6b is similar.

A.1 Error of NFG

In our NFG, there are two kinds of errors: approximation error and rounding error. The approximation error, ε_a, was discussed in Section 3. Thus, this section focuses on rounding error. Since truncation and rounding occur only in the fractional bits, we ignore the integer bits in this analysis. We assume that there is at least one fractional bit.

Errors in coefficients. The coefficients, c_{1i} and $f(s_i)+v_i$, used in the linear approximation $g_i(x)=c_{1i}(x-s_i)+f(s_i)+v_i$ are rounded to u_1 and u_0 bits, respectively. That is, due to rounding, the coefficients c_{1i} and $f(s_i)+v_i$ become

$$\begin{aligned} c_{1i}+\alpha_1 \quad & (|\alpha_1|\leq 2^{-(u_1+1)}) \\ f(s_i)+v_i+\alpha_0 \quad & (|\alpha_0|\leq 2^{-(u_0+1)}), \end{aligned}$$

where α_1 and α_0 are rounding errors of c_{1i} and $f(s_i)+v_i$, respectively. In this case, we need no hardware for rounding coefficients because they are precomputed before storing in the coefficients table. Since rounding yields a more accurate result than truncation, we choose rounding.

Errors in multiplication. When input x has m_{in} bits, our synthesis system ensures that s_i also has m_{in} bits. Therefore, the result of the subtraction operation $(x-s_i)$ also has m_{in} bits and has no rounding error. As shown in the previous paragraph, the value of c_{1i} is stored as $c_{1i}+\alpha_1$ in the coefficients table. Thus, the original product $c_{1i}(x-s_i)$ is changed to

$$(c_{1i}+\alpha_1)(x-s_i)=c_{1i}(x-s_i)+\alpha_1(x-s_i). \qquad \text{(i)}$$

The value of (i) has (u_1+m_{in}) bits and it is truncated to u_0 bits in order to match the addition with $f(s_i)+v_i$. Note that $u_0\leq u_1+m_{in}$. We choose truncation because it does not require additional hardware, whereas rounding requires half adders at the multiplier output. Following the method shown in [30], the (u_0+1)th fractional bit $d_{-(u_0+1)}$ in (i) is set to 1 after truncation. The rounding error α_2 for truncation from (u_1+m_{in}) to u_0 bits is

$$0\leq\alpha_2\leq 2^{-(u_0+1)}-2^{-(u_1+m_{in})}.$$

Thus, the linear part $c_{1i}(x-s_i)$ of the approximation has the value

$$c_{1i}(x-s_i)+\alpha_1(x-s_i)-\alpha_2.$$

This is added to the constant coefficient $f(s_i)+v_i$ to form the complete approximation.

Errors in addition. From the previous paragraphs, the original sum $c_{1i}(x-s_i)+f(s_i)+v_i$ is changed to

$$c_{1i}(x-s_i)+f(s_i)+v_i+\alpha_0+\alpha_1(x-s_i)-\alpha_2. \qquad \text{(ii)}$$

This value has u_0 bits and this is rounded to m_{out} bits (the output accuracy given in the specification), where $m_{out}\leq u_0$. Since $d_{-(u_0+1)}$ in (i) is set to 1, $d_{-(u_0+1)}$ in (ii) is also 1. Note that $d_{-(u_0+1)}$ is not implemented in hardware [30]. The error α_3 for this rounding is

$$|\alpha_3|\leq 2^{-(m_{out}+1)}-2^{-(u_0+1)}.$$

Thus, the value of the approximation $g_i(x)$ is

$$c_{1i}(x-s_i)+f(s_i)+v_i+\alpha_0+\alpha_1(x-s_i)-\alpha_2+\alpha_3.$$

It follows that

$$\alpha_0+\alpha_1(x-s_i)-\alpha_2+\alpha_3 \qquad \text{(iii)}$$

is the total rounding error for an NFG implemented with nonuniform segmentation. The absolute value of (iii) is maximum when the rounding errors are

$$\begin{aligned} \alpha_0 &= -2^{-(u_0+1)}, \qquad \alpha_1=-2^{-(u_1+1)}, \\ \alpha_2 &= 2^{-(u_0+1)}-2^{-(u_1+m_{in})}, \quad \text{and} \\ \alpha_3 &= -(2^{-(m_{out}+1)}-2^{-(u_0+1)}). \end{aligned}$$

Since the smallest value of x in the ith segment is the segment's starting point s_i, the value of $(x-s_i)$ is positive. Therefore, the maximum total rounding error ε_r is

$$\begin{aligned} \varepsilon_r = & 2^{-(u_0+1)}+2^{-(u_1+1)}max\_seg-2^{-(u_1+m_{in})} \\ & +2^{-(m_{out}+1)}, \end{aligned}$$

where

$$max\_seg=\max_{0\leq i\leq t-1}(e_i-s_i)$$

and t is the number of segments.

A.2 Calculation of Bit Sizes for Units

For any unit in Fig. 6a, the number of bits in the integer part is

$$\lceil \log_2(max\_value + 1)\rceil + 1,$$

where max_value is an integer that denotes the maximum absolute value of output of each unit. That is, $max\_value = \max(|max\_val|, |min\_val|)$, where max_val and min_val are the maximum and minimum values, respectively, of the unit's output.

On the other hand, the number of bits for the fractional part is calculated using the result of the error analysis. From the error analysis, an NFG with an acceptable error ε is achieved when

$$\varepsilon_a + \varepsilon_r < \varepsilon, \tag{iv}$$

where ε_a and ε_r are the approximation error and the rounding error, respectively. To generate fast and compact NFGs, we find the minimum u_0 and u_1 that satisfy (iv) by using nonlinear programming [11], where the objective and constraint are

$$\begin{aligned}\text{Minimize}: &\quad u_0 + u_1\\ \text{Constraint}: &\quad \varepsilon_a + \varepsilon_r < \varepsilon.\end{aligned}$$

Calculation of shift bit h_i. From (i), the maximum error in the multiplication is

$$2^{-(u_1+1)} max\_seg,$$

where max_seg is the maximum value of $(e_i - s_i)$ over all segments i. There is the largest positive integer h_i that satisfies the following relation for each segment i:

$$\begin{aligned}2^{-(u_1+1)}2^{h_i}(e_i - s_i) &\leq 2^{-(u_1+1)} max\_seg\\ 2^{-(u_1-h_i+1)}(e_i - s_i) &\leq 2^{-(u_1+1)} max\_seg.\end{aligned} \tag{v}$$

From (v), for each segment, coefficient c_{1i} can be rounded to $(u_1 - h_i)$ bits within an acceptable error. Note that h_i, as shown in (vi), can be used in the multiplier scaling method in [15], as described in Section 4.3. From (v), we have

$$h_i \leq \log_2\left(\frac{max\_seg}{e_i - s_i}\right).$$

Furthermore, we choose

$$h_i = \left\lfloor \log_2\left(\frac{max\_seg}{e_i - s_i}\right)\right\rfloor. \tag{vi}$$

Acknowledgments

This research is partly supported by the Grant-in-Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS), funds from the Ministry of Education, Culture, Sports, Science, and Technology (MEXT) via the Kitakyushu innovative cluster project, and a contract with the US National Security Agency. The authors appreciate Dr. Marc D. Riedel for the work in [26]. The comments of the three reviewers were useful in improving this paper. This paper is an extended version of [27].

References

[1] R. Andraka, "A Survey of CORDIC Algorithms for FPGA-Based Computers," *Proc. Sixth ACM/SIGDA Int'l Symp. Field Programmable Gate Array (FPGA '98)*, pp. 191-200, Feb. 1998.

[2] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677-691, Aug. 1986.

[3] A. Cantoni, "Optimal Curve Fitting with Piecewise Linear Functions," *IEEE Trans. Computers*, vol. 20, no. 1, pp. 59-67, Jan. 1971.

[4] J. Cao, B.W.Y. Wei, and J. Cheng, "High-Performance Architectures for Elementary Function Generation," *Proc. 15th IEEE Symp. Computer Arithmetic (ARITH '01)*, pp. 136-144, June 2001.

[5] E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping," *Proc. 30th ACM/IEEE Design Automation Conf.*, pp. 54-60, June 1993.

[6] F. de Dinechin and A. Tisserand, "Multipartite Table Methods," *IEEE Trans. Computers*, vol. 54, no. 3, pp. 319-330, Mar. 2005.

[7] N. Doi, T. Horiyama, M. Nakanishi, and S. Kimura, "Minimization of Fractional Wordlength on Fixed-Point Conversion for High-Level Synthesis," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC '04)*, pp. 80-85, 2004.

[8] N. Doi, T. Horiyama, M. Nakanishi, and S. Kimura, "An Optimization Method in Floating-Point to Fixed-Point Conversion Using Positive and Negative Error Analysis and Sharing of Operations," *Proc. 12th Workshop Synthesis and System Integration of Mixed Information Technologies (SASIMI '04)*, pp. 466-471, Oct. 2004.

[9] 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.

[10] H. Hassler and N. Takagi, "Function Evaluation by Table Look-Up and Addition," *Proc. 12th IEEE Symp. Computer Arithmetic (ARITH '95)*, pp. 10-16, July 1995.

[11] T. Ibaraki and M. Fukushima, *FORTRAN 77 Optimization Programming*, 1991 (in Japanese).

[12] Y. Iguchi, T. Sasao, and M. Matsuura, "Realization of Multiple-Output Functions by Reconfigurable Cascades," *Proc. Int'l Conf. Computer Design: VLSI in Computers and Processors (ICCD '01)*, pp. 388-393, Sept. 2001.

[13] V.K. Jain, S.A. Wadekar, and L. Lin, "A Universal Nonlinear Component and Its Application to WSI," *IEEE Trans. Components, Hybrids, and Manufacturing Technology*, vol. 16, no. 7, pp. 656-664, Nov. 1993.

[14] V.K. Jain and L. Lin, "High-Speed Double Precision Computation of Nonlinear Functions," *Proc. 12th IEEE Symp. Computer Arithmetic (ARITH '95)*, pp. 107-114, July 1995.

[15] D.-U. Lee, W. Luk, J. Villasenor, and P.Y.K. Cheung, "Non-Uniform Segmentation for Hardware Function Evaluation," *Proc. 11th Int'l Conf. Field Programmable Logic and Applications (FPL '03)*, pp. 796-807, Sept. 2003.

[16] D.-U. Lee, W. Luk, J. Villasenor, and P.Y.K. Cheung, "Hierarchical Segmentation Schemes for Function Evaluation," *Proc. IEEE Conf. Field-Programmable Technology (FPT '03)*, pp. 92-99, Dec. 2003.

[17] D.-U. Lee, W. Luk, J. Villasenor, and P.Y.K. Cheung, "A Gaussian Noise Generator for Hardware-Based Simulations," *IEEE Trans. Computers*, vol. 53, no. 12, pp. 1523-1534, Dec. 2004.

[18] D.-U. Lee, A.A. Gaffar, O. Mencer, and W. Luk, "Optimizing Hardware Function Evaluation," *IEEE Trans. Computers*, vol. 54, no. 12, pp. 1520-1531, Dec. 2005.

[19] D.M. Lewis, "Interleaved Memory Function Interpolators with Application to an Accurate LNS Arithmetic Unit," *IEEE Trans. Computesr*, vol. 43, no. 8, pp. 974-982, Aug. 1994.

[20] J.-M. Muller, *Elementary Function: Algorithms and Implementation.* Birkhauser Boston, 1997.

[21] J.-M. Muller, "Partially Rounded Small-Order Approximations for Architecture, Hardware-Oriented, Table-Based Methods," *Proc. 16th IEEE Symp. Computer Arithmetic (ARITH '03)*, pp. 114-121, June 2003.

[22] S. Nagayama and T. Sasao, "Compact Representations of Logic Functions Using Heterogeneous MDDs," *IEICE Trans. Fundamentals*, vol. E86-A, no. 12, pp. 3168-3175, Dec. 2003.

[23] S. Nagayama, T. Sasao, "On the Optimization of Heterogeneous MDDs," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 11, pp. 1645-1659, Nov. 2005.

[24] S. Nagayama, T. Sasao, and J.T. Butler, "Programmable Numerical Function Generators Based on Quadratic Approximation: Architecture and Synthesis Method," *Proc. Asia and South Pacific Design Automation Conf. (ASPDAC '06),* pp. 378-383, 2006.

[25] T. Sasao and M. Matsuura, "A Method to Decompose Multiple-Output Logic Functions," *Proc. 41st Design Automation Conf. (DAC '04),* pp. 428-433, June 2004.

[26] T. Sasao, J.T. Butler, and M.D. Riedel, "Application of LUT Cascades to Numerical Function Generators," *Proc. 12th Workshop Synthesis and System Integration of Mixed Information Technologies (SASIMI '04),* pp. 422-429, Oct. 2004.

[27] T. Sasao, S. Nagayama, and J.T. Butler, "Programmable Numerical Function Generators: Architectures and Synthesis Method," *Proc. Int'l Conf. Field-Programmable Logic and Applications (FPL '05),* pp. 118-123, Aug. 2005.

[28] T. Sasao, "Design Methods for Multiple-Valued Input Address Generators," *Proc. 36th IEEE Int'l Symp. Multiple-Valued Logic (ISMVL '06),* May 2006.

[29] Scilab 3.0, INRIA-ENPC France, http://scilabsoft.inria.fr/, 2004.

[30] M.J. Schulte and J.E. Stine, "Symmetric Bipartite Tables for Accurate Function Approximation," *Proc. 13th IEEE Symp. Computer Arithmetic (ARITH '97),* vol. 48, no. 9, pp. 175-183, 1997.

[31] M.J. Schulte and J.E. Stine, "Approximating Elementary Functions with Symmetric Bipartite Tables," *IEEE Trans. Computers,* vol. 48, no. 8, pp. 842-847, Aug. 1999.

[32] M.J. Schulte and E.E. Swartzlander Jr., "Hardware Designs for Exactly Rounded Elementary Functions," *IEEE Trans. Computers,* vol. 43, no. 8, pp. 964-973, Aug. 1994.

[33] J.E. Stine and M.J. Schulte, "The Symmetric Table Addition Method for Accurate Function Approximation," *J. VLSI Signal Processing,* vol. 21, no. 2, pp. 167-177, June 1999.

[34] N. Takagi, T. Asada, and S. Yajima, "Redundant CORDIC Methods with a Constant Scale Factor for Sine and Cosine Computation," *IEEE Trans. Computers,* vol. 40, no. 9, pp. 989-995, Sept. 1991.

[35] J.E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electronic Computers,* vol. 8, no. 3, pp. 330-334, Sept. 1959.

[36] W. Wong and E. Goto, "Fast Evaluation of the Elementary Functions in Single Precision," *IEEE Trans. Computers,* vol. 44, no. 3, pp. 453-457, Mar. 1995.

Tsutomu Sasao (S'72-M'77-SM'90-F'94) received the BE, ME, and PhD degrees in electronics engineering from Osaka University, Osaka, Japan, in 1972, 1974, and 1977, respectively. He has held faculty/research positions at Osaka University, Japan, the IBM T.J. Watson Research Center, Yorktown Heights, New York, and the Naval Postgraduate School, Monterey, California. He is now a professor in the Department of Computer Science and Electronics at the Kyushu Institute of Technology, Iizuka, Japan. He has served as the program chairman for the IEEE International Symposium on Multiple-Valued Logic (ISMVL) many times. In addition, he was the symposium chairman of the 28th ISMVL held in Fukuoka, Japan, in 1998. He has also served as an associate editor of the *IEEE Transactions on Computers*. His research areas include logic design and switching theory, representations of logic functions, and multiple-valued logic. He has published more than nine books on logic design, including *Logic Synthesis and Optimization*, *Representation of Discrete Functions*, *Switching Theory for Logic Synthesis*, and *Logic Synthesis and Verification* (Kluwer Academic, 1993, 1996, 1999, and 2001, respectively). He received the National Institute of Water and Atmospheric Research (NIWA) Memorial Award in 1979, the Distinctive Contribution Awards from the IEEE Computer Society Technical Committee on Multiple-Valued Logic (MVL-TC) for papers presented at ISMVL in 1986, 1996, 2003, and 2004, and the Takeda Techno-Entrepreneurship Award in 2001. He is a fellow of the IEEE and a member of the IEEE Computer Society.

Shinobu Nagayama (S'02-M'04) received the BS and ME degrees from Meiji University, Kanagawa, Japan, in 2000 and 2002, respectively, and the PhD degree in computer science from the Kyushu Institute of Technology, Iizuka, Japan, in 2004. He is now a research associate at Hiroshima City University, Hiroshima, Japan. He received the Outstanding Contribution Paper Award from the IEEE Computer Society Technical Committee on Multiple-Valued Logic (MVL-TC) in 2005 for a paper presented at the 34th IEEE International Symposium on Multiple-Valued Logic (ISMVL '04) and the Excellent Paper Award from the Information Processing Society of Japan (IPS) in 2006. His research interests include numerical function generators, decision diagrams, software synthesis, and embedded systems. He is a member of the IEEE and the IEEE Computer Society.

Jon T. Butler (S'67-M'67-SM'82-F'89) received the BEE and MEngr degrees from Rensselaer Polytechnic Institute, Troy, New York, in 1966 and 1967, respectively, and the PhD degree from Ohio State University, Columbus, in 1973. Since 1987, he has been a professor at the Naval Postgraduate School, Monterey, California. From 1974 to 1987, he was with Northwestern University, Evanston, Illinois. During that time, he served two periods of leave at the Naval Postgraduate School: first as a National Research Council senior postdoctoral associate (1980-1981) and second as the NAVALEX chair professor (1985-1987). He served one period of leave as a foreign visiting professor at the Kyushu Institute of Technology, Iizuka, Japan. His research interests include logic optimization and multiple-valued logic. He has served on the editorial boards of the *IEEE Transactions on Computers*, *Computer*, and IEEE Computer Society Press. He served as the editor-in-chief of *Computer* and the IEEE Computer Society Press. He received the Award of Excellence, the Outstanding Contributed Paper Award, and a Distinctive Contributed Paper Award for papers presented at the International Symposium on Multiple-Valued Logic (ISMVL). He received the Distinguished Service Award, two Meritorious Awards, and nine Certificates of Appreciation for service to the IEEE Computer Society. He is a fellow of the IEEE and a member of the IEEE Computer Society.