View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Calhoun, Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

Faculty and Researcher Publications Faculty and Researcher Publications

2005-08

Programmable numerical function generators: Architectures and synthesis system

Sasao, Tsutomu

þÿT. Sasao, S. Nagayama, and J. T. Butler, Programmable numerical funct
Architectures and synthesis system," FPL2005 ,Tampere, Aug.24-26, 2005, pp.118-123.

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

PROGRAMMABLE NUMERICAL FUNCTION GENERATORS: ARCHITECTURES AND SYNTHESIS METHOD

Tsutomu Sasao[1], *Shinobu Nagayama*[2], *Jon T. Butler*[3]

[1] Dept. of CSE, Kyushu Institute of Technology, Japan
[2] Dept. of CE, Hiroshima City University, Japan
[3] Dept. of ECE, Naval Postgraduate School, U.S.A.

ABSTRACT

This paper presents an architecture and a synthesis method for programmable numerical function generators of trigonometric functions, logarithm functions, square root, reciprocal, etc. Our architecture uses an LUT (Look-Up Table) cascade as the segment index encoder, compactly realizes various numerical functions, and is suitable for automatic synthesis. We have developed a synthesis system that converts MATLAB-like specification into HDL code. We propose and compare three architectures implemented as a FPGA (Field-Programmable Gate Array). Experimental results show the efficiency of our architecture and synthesis system.

1. INTRODUCTION

Numerical functions, such as trigonometric functions, logarithm, square root, reciprocal, etc, are extensively used in computer graphics, digital signal processing, communication systems, robotics, astrophysics, fluid physics, etc. High-level programming languages, such as C and FORTRAN, usually have software libraries for standard numerical functions. However, for high-speed applications, a hardware implementation is needed. Hardware implementation by a single look-up table for a numerical function $f(x)$ is simple and fast. For low-precision computations of $f(x)$, i.e., when x has a small number of bits, this implementation is straightforward. For high-precision computations, however, the single look-up table implementation is impractical due to the huge table size. For such applications, the CORDIC (COordinate Rotation DIgital Computer) algorithm [1, 16] has been used. Although faster than software approaches, it is iterative and therefore slow.

This paper proposes an architecture and a synthesis for NFGs (Numerical Function Generators) using linear approximations. By using the LUT cascade [7, 11], our architecture can realize various numerical functions quickly, and is suitable for the automatic synthesis. Fig. 1 shows the synthesis flow for the NFG. It generates HDL (Hardware Description Language) code from the design specification described by Scilab [14], a MATLAB-like numerical calculation software. The design specification includes a numerical function $f(x)$, a domain $[a, b]$ for the x, and an acceptable error. This system first partitions the given domain for x into segments, and then approximates the given function $f(x)$ by a linear function for each segment. Next, it analyzes the error, and derives the necessary precision for computing units in the NFG. Then, it generates HDL code that maps into an FPGA, using an FPGA vendor tool.

This paper is organized as follows. Section 2 introduces terminology. Section 3 proposes a linear approximation algorithm for numerical functions. Section 4 shows three different architectures for NFGs. Section 5 describes the FPGA implementation method. Section 6 evaluates the performance of our architecture and synthesis system. Due to the page limitation, the error analysis for our NFGs is omitted. It is available in [13]. This paper builds on [12].

2. PRELIMINARIES

Definition 2.1 *The **binary fixed-point representation** of a value r has the form*

$$d_{n_int-1}\, d_{n_int-2}\, \dots\, d_1\, d_0 .\, d_{-1}\, d_{-2}\, \dots\, d_{-n_frac} \quad (1)$$

where $d_i \in \{0,1\}$, n_int is the number of bits for the integer part, and n_frac is the number of bits for the fractional part of r. The representation in (1) is two's complement, and so

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

Definition 2.2 ***Error** is the absolute difference between the original value and the approximated value. **Approximation error** is the error caused by a function approximation, and **rounding error** is the error caused by a binary fixed-point representation. **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 2.3 ***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, $n_int + n_frac = n$. An n-**bit precision NFG** has an n-bit input.*

Definition 2.4 ***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; that is, $n_frac = m$. An m-**bit accuracy NFG** is an NFG with m-bit fractional part of the input, m-bit fractional part of the output, and a 2^{-m} acceptable error.*

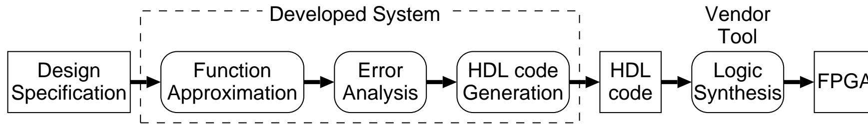


Fig. 1. Synthesis flow for NFGs.

3. LINEAR APPROXIMATION ALGORITHM

For functions that are approximately linear, such as $\sin(x)$ $(0 \leq x \leq \pi/2)$, the linear approximation method yields small approximation error with relatively few segments. Indeed, in such cases uniformly wide segments yield good approximations. Uniform segments have been used in previous studies [2, 5, 15] to simplify the circuits. However, for some kinds of numerical functions such as $\sqrt{-\ln(x)}$, uniform segmentation requires too many segments. To approximate such functions using fewer segments, a partitioning method of the domain into *non-uniform segments* is proposed [8]. Unfortunately, their segmentation is fixed; it is not optimized for the given function. We improve on this by adapting the segmentation so that relatively few segments are needed. This reduces the memory required.

3.1. Segmentation Algorithm

Fig. 2 presents the segmentation algorithm, where the inputs are a numerical function $f(x)$, a domain $[a, b]$ for x, and an acceptable approximation error c. This algorithm begins by forming one segment over the whole domain $[a, b]$. This is an initial piecewise approximation by a linear function whose endpoints are $(a, f(a))$ and $(b, f(b))$. If the current segment fails to provide the acceptable approximation error, it is partitioned into two segments joined at a point p where the maximum error occurs. This process iterates until the two subsegments approximate $f(x)$ to within the acceptable approximation error. The correction values v_i are used to reduce the approximation errors. In Fig. 2, max_{fg} and min_{fg} denote the maximum positive error and the maximum negative error, respectively. These errors are equalized by vertical shift of linear function $g(x)$ with v_i. In Fig. 2, 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 [6] to find these values efficiently.

The algorithm is based on the Douglas-Peucker algorithm [4] that is used in rendering curves for graphics displays.

3.2. Computation of Approximated Values

A segment $[s_i, e_i]$ is denoted by seg_i; thus, the segments generated by the segmentation algorithm are denoted by seg_0, $seg_1, \ldots, seg_{t-1}$. For each seg_i, the numerical function $f(x)$ is approximated by the corresponding linear function $g_i(x)$. Therefore, the approximated value y of $f(x)$ is computed as follows:

$$y = g_i(x) = c_{1i}x + c_{0i}, \tag{2}$$

where $g_i(x)$ is the linear function for the segment seg_i,

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

By substituting c_{0i} into Equation (2), and simplifying it, we have

$$g_i(x) = c_{1i}(x - s_i) + f(s_i) + v_i. \tag{3}$$

Let $h = e_i - s_i$ and $h \to 0$. Then, we have $c_{1i} = f'(s_i)$. By substituting this equation into Equation (3), we have $g_i(x) = f'(s_i)(x - s_i) + f(s_i) + v_i$. This is the *first-order Taylor expansion* around $x = s_i$ for $f(x)$ with the correction value v_i. Our algorithm can approximate $f(x)$ with any acceptable approximation error using sufficiently many segments.

4. ARCHITECTURE FOR NFGS

4.1. Overview

Although Equation (2) and Equation (3) represent the same values, the architectures for the NFGs realizing them are different. Fig. 3 (a) shows the architecture for Equation (2); it uses four units: the segment index encoder that computes the index i for segment seg_i including the value x; the coefficients table for c_{1i} and c_{0i}; the multiplier; and the adder. On the other hand, Fig. 3 (b) shows the architecture for Equation (3); it uses five units: the four units used in Fig. 3(a), where $-s_i$ is stored in the coefficients table, and an additional adder for computation of $x + (-s_i)$. In Equation (3), when $s_i =$ (the most significant $(n - k)$ bits of $x) \times 2^k$, the index i of the segment seg_i is equal to the most significant $(n - k)$ bits, and $(x - s_i)$ is equal to the least significant k bits of x, where x has the n-bit precision. Therefore, in this case, the linear approximations are realized using only three units as shown in Fig. 3 (c): the coefficients table for c_{1i} and $f(s_i) + v_i$; the multiplier; and the adder. Note that this architecture realizes a uniform segmentation.

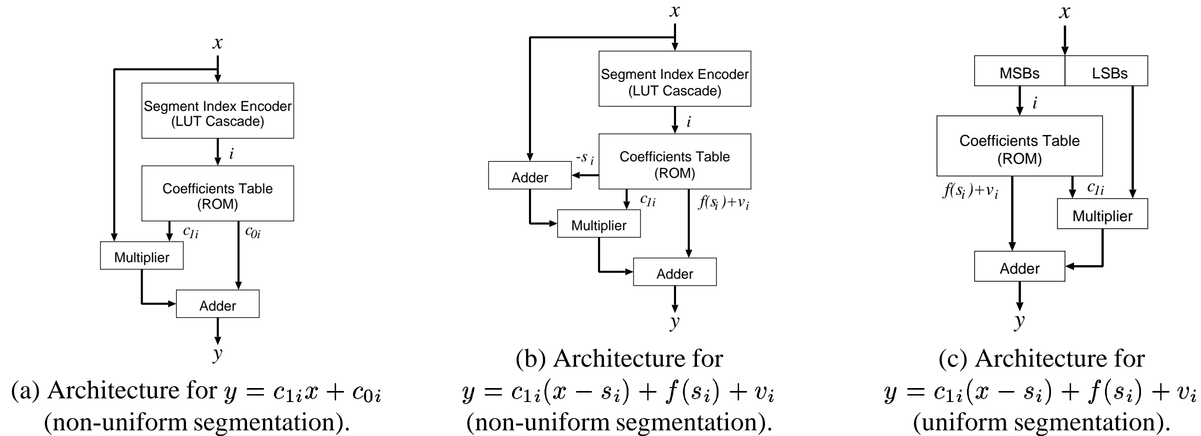
We use the architecture shown in Fig. 3 (b) to produce fast and compact NFGs. In Section 6, we will compare the performances of three different architectures.

4.2. Segment Index Encoder

A *segment index encoder* converts an input value x into a segment index i for seg_i. It realizes the segment index function $seg_func(x) : B^n \to \{0, 1, \ldots, t-1\}$ shown in Fig. 4 (a), where x has n-bit precision, $B = \{0, 1\}$, and t denotes the number of segments. In [8], to simplify the segment index encoder, the values of s_i and e_i are restricted. That is, the restrictive non-uniform segmentation is used for the segment index encoder. Such segmentation increases the number of segments and is unsuitable for automatic segmentation. Our synthesis system uses the *LUT cascade* [7, 11, 12]

Input:	Numerical function $f(x)$, Domain $[a,b]$ for x, Acceptable approximation error c.
Output:	Segments $[s_0,e_0],[s_1,e_1],\ldots,[s_{t-1},e_{t-1}]$, Correction values $v_0,v_1,\ldots,v_{t-1}$
Process:	This is recursive procedure. Initial segment is set to $[a,b]$.
1.	For given segment $[s,e]$, compute a line connecting two points $(s,f(s))$ and $(e,f(e))$, represented by linear function $g(x)=c_1x+c_0$, where $c_1=\frac{f(e)-f(s)}{e-s}$, $c_0=f(s)-c_1s$.
2.	Find a value p_{max} of the variable x that maximizes $f(x)-g(x)$ in $[s,e]$, and let $max_{fg}=f(p_{max})-g(p_{max})$, where $max_{fg}\geq 0$.
3.	Similarly, find a p_{min} that minimizes $f(x)-g(x)$, and let $min_{fg}=f(p_{min})-g(p_{min})$, where $min_{fg}\leq 0$.
4.	Let $p=p_{max}$ if $\lvert max_{fg}\rvert>\lvert min_{fg}\rvert$, and let $p=p_{min}$ otherwise.
5.	Let $error=\lvert max_{fg}-min_{fg}\rvert/2$, and $v=(max_{fg}+min_{fg})/2$.
6.	If $error\leq c$, then declare $[s,e]$ to be a completed segment. If all segments are completed, stop.
7.	For any segment $[s,e]$ that is not completed, partition $[s,e]$ into two segments $[s,p]$ and $[p,e]$, and iterate the same process for each new segment recursively.

Fig. 2. Segmentation algorithm for the domain.



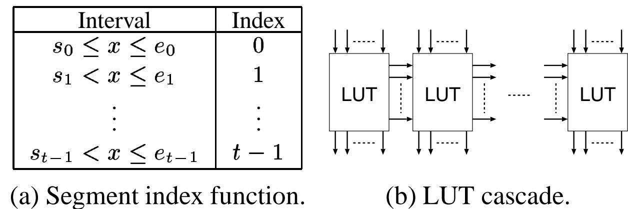
(a) Architecture for $y=c_{1i}x+c_{0i}$ (non-uniform segmentation).

(b) Architecture for $y=c_{1i}(x-s_i)+f(s_i)+v_i$ (non-uniform segmentation).

(c) Architecture for $y=c_{1i}(x-s_i)+f(s_i)+v_i$ (uniform segmentation).

Fig. 3. Three architectures for NFGs.

shown in Fig. 4 (b) to realize any $seg_func(x)$. It can be designed by functional decomposition using BDDs (Binary Decision Diagrams) representing $seg_func(x)$. That is, our synthesis system uses the nonrestrictive segmentation. This is suitable for automatic synthesis. In LUT cascades, the interconnecting lines between adjacent LUTs are called *rails*. The size of an LUT cascade depends on the number of rails. Thus, to produce a compact LUT cascade, a small number of rails is sought. The next theorem shows that the segment index functions are realized by compact LUT cascades.



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

(a) Segment index function. (b) LUT cascade.

Fig. 4. Segment index encoder.

Theorem 4.1 *[12] 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.*

Our synthesis system uses heterogeneous MDDs (Multi-valued Decision Diagrams) [10] to find compact LUT cascades. Since the LUT cascade is suitable for pipeline processing, it offers a fast and compact circuit. Experimental results will show that LUT cascades have sizes comparable for the segment index encoder using uniform segmentation for certain functions, like trigonometric functions, and much smaller sizes for other functions, like $\sqrt{x}$.

5. IMPLEMENTATION WITH FPGAS

Modern FPGAs consist of logic elements, synchronous memory blocks, multipliers (DSP units), etc. Our synthesis system efficiently generates NFGs using these components. Each unit for the NFG shown in Fig. 3 (b) is implemented by the following components in an FPGA: 1) Segment index encoder (LUT cascade) and coefficients table (ROM): by synchronous memory blocks; 2) Multiplier: by DSP units; and 3) Adder: by logic elements. Our synthesis system derives the optimum bit-width for each component by automatic error analysis [13].

Table 1. Number of pipeline stages for NFGs.

Name of units	Pipeline stages
1. LUT cascade	n_cas
2. Coefficients table	1
3. Adder for $x + (-s_i)$	1
4. Multiplier for $c_{1i}(x - s_i)$	1
5. Shifter (optional)	0 or 1
6. Two's complementer (optional)	0 or 1
7. Adder for $c_{1i}(x - s_i) + c_{0i}$	1
Total pipeline stages	$n_cas + 4$
	$\sim n_cas + 6$

n_cas: Number of LUTs for LUT cascade.

5.1. Size Reduction of Multiplier

Although modern FPGAs have dedicated multipliers, large multipliers are slow. In our architecture, the multiplier often has the longest delay time among all the units. Thus, to generate a fast NFG, reducing the size of the multiplier is important. Since the size of multiplier depends on the number of bits for c_{1i}, we reduce the number of bits for c_{1i}.

First, we consider the case where the absolute value of c_{1i} is large. When $|c_{1i}|$ is large, many bits are required to represent c_{1i} in binary fixed-point. To reduce the number of bits for such c_{1i}, we use the *scaling method* shown in [8]. When $|c_{1i}|$ is large, we represent c_{1i} as $c_{1i} = c_{1i} \times 2^{-l_i} \times 2^{l_i}$. Instead of the original value of c_{1i}, we store the values of $c_{1i} \times 2^{-l_i}$ and l_i in the coefficients table. In this case, the product $c_{1i}(x - s_i)$ is computed using the multiplier for $c_{1i} \times 2^{-l_i} \times (x - s_i)$ and the shifter for an l_i-bit shift to the left. The increase of l_i reduces the number of bits to represent the value of $c_{1i} \times 2^{-l_i}$, while increasing the rounding error. Our synthesis system finds the optimum value of l_i for each segment seg_i within the acceptable error [13]. When the optimum values of l_i are 0 for all the segments seg_i, no shifter is implemented, that is, $c_{1i}(x - s_i)$ is directly implemented with the multiplier.

Next, we consider the case where the range of c_{1i} includes negative values. In this case, our synthesis system stores the absolute value of c_{1i} and the sign bit for c_{1i} separately in the coefficients table, and first uses the unsigned multiplier to compute $|c_{1i}|(x-s_i)$, and then a two's complementer to produce the signed value with the sign bit. When c_{1i} is positive for all segments seg_i, no two's complementer is implemented. That is, $c_{1i}(x - s_i)$ is directly implemented with an unsigned multiplier.

For simplicity, Fig. 3 omits the schemes for the scaling method and the two's complementer.

5.2. Pipeline Processing

To implement a high-throughput NFG, our synthesis system inserts pipeline registers between all the units in the architecture. Since all the units operate in parallel, and each unit has a short delay time, our NFGs achieves high throughput. Table 1 shows the units and the number of pipeline stages for them. Our NFGs may have from $n_cas + 4$ to $n_cas + 6$ pipeline stages, where n_cas is the number of LUTs for the LUT cascade.

Table 3. Numbers of segments for non-uniform and uniform segmentations.

Acceptable approximation error : 2^{-17}			
Function $f(x)$	Domain	Number of segments	
		Non-uniform	Uniform
$\sin(\pi x)$	$[0, 1/2]$	127	257
$\cos(\pi x)$	$[0, 1/2]$	127	257
$\tan(\pi x)$	$[0, 1/4]$	112	257
$1/x$	$[1/8, 1]$	702	3585
$1/\sqrt{x}$	$[1/32, 1]$	620	7937
$\sqrt{x}$	$[0, 1]$	231	32769
$\sqrt{-\ln(x)}$	$(0, 1]$	584	32768

6. EXPERIMENTAL RESULTS

6.1. Computation Time for Segmentation Algorithm

Table 2 shows the CPU time for the segmentation algorithm applied to 12 of the 14 functions used in [12] with various acceptable approximation errors. In this table, the Sigmoid and the Gaussian are defined as follows:

$$\text{Sigmoid} = \frac{1}{1 + e^{-4x}} \quad , \quad \text{Gaussian} = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

The segmentation algorithm is recursive, and computation time depends on the number of segments. Smaller acceptable approximation error requires more segments and longer computation time. However, Table 2 shows that for all the functions in the table, the CPU times were smaller than 2 seconds when the acceptable approximation error was 2^{-25}. These results show that our segmentation algorithm generates non-uniform segments quickly.

6.2. Comparison of Three Architectures

This section compares the three architectures for NFGs shown in Fig. 3. Let Arc_A, Arc_B, and Arc_C denote the architectures shown in Fig. 3 (a), (b), and (c), respectively. To compare these architectures for various functions, we implemented 16-bit precision NFGs on the same FPGA (Altera Stratix EP1S10F484C5), using an the acceptable approximation error of 2^{-17} for each function.

Table 3 compares the numbers of segments for the non-uniform and the uniform segmentations. It shows that, for all the functions, the number of non-uniform segments is less than the half that of uniform segments. Since Arc_A and Arc_B use non-uniform segmentation, they implement various numerical functions with small coefficients tables. On the other hand, Arc_C uses uniform segmentation. Thus, although Arc_C implements functions, such as trigonometric functions, with a relatively small coefficients table, it requires a large coefficients table for other functions. In this experiment, there were not enough memory blocks in the FPGA (EP1S10F484C5) to implement the non-trigonometric functions using Arc_C.

Tables 4 and 5 compare the amount of hardware and performances for the three architectures. These tables show that, for trigonometric functions, Arc_C implements the shortest latency and most compact NFGs among the three, since Arc_C requires no segment index encoder. Therefore, when the number of uniform segments is relatively small, Arc_C is

Table 2. CPU time [msec] for the segmentation algorithm.

Function $f(x)$	Domain	AAE $= 2^{-9}$ #seg.	AAE $= 2^{-9}$ CPU time	AAE $= 2^{-17}$ #seg.	AAE $= 2^{-17}$ CPU time	AAE $= 2^{-25}$ #seg.	AAE $= 2^{-25}$ CPU time
2^x	$[0, 1]$	8	0.1	128	0.1	2048	80
$1/x$	$[1/8, 1]$	39	0.1	702	30	11218	280
$1/\sqrt{x}$	$[1/32, 1]$	31	0.1	620	20	9946	300
$\sqrt{x}$	$[0, 1]$	12	0.1	231	10	3941	110
$\sqrt{-\ln(x)}$	$(0, 1]$	23	0.1	584	40	12089	1840
$\log_2(x)$	$[1, 2)$	8	0.1	128	10	2048	70
$\ln(x)$	$[1, 2)$	6	0.1	89	0.1	1437	30
$\sin(\pi x)$	$[0, 1/2]$	8	0.1	127	10	2027	50
$\cos(\pi x)$	$[0, 1/2]$	8	0.1	127	10	2027	50
$\tan(\pi x)$	$[0, 1/4]$	7	0.1	112	10	1787	50
Sigmoid	$[0, 1]$	8	0.1	127	10	2020	60
Gaussian	$[0, 1/2]$	2	0.1	32	10	512	10

AAE: Acceptable Approximation Error. #seg.: Number of segments.
Experiment environment
CPU: Pentium4 Xeon 2.8GHz memory : 4GB
OS: Redhat (Linux 7.3) C compiler : gcc -O2

Table 4. Amount of hardware for three architectures.

Precision, Accuracy: 16-bit precision, 15-bit accuracy
FPGA device: Altera Stratix (EP1S10F484C5)
Logic synthesis tool: Altera QuartusII 4.1 (default option)

Function $f(x)$	Arc_A #LEs	Arc_A Memory	Arc_A #DSPs	Arc_B #LEs	Arc_B Memory	Arc_B #DSPs	Arc_C #LEs	Arc_C Memory	Arc_C #DSPs
$\sin(\pi x)$	106	19355	8	107	20061	2	82	14848	2
$\cos(\pi x)$	136	19543	8	116	20169	2	67	15417	2
$\tan(\pi x)$	106	19355	8	116	20039	2	83	29696	2
$1/x$	153	172102	8	172	172119	2	112	278594	2
$1/\sqrt{x}$	182	159826	8	183	160861	2	145	557119	2
$\sqrt{x}$	191	43610	2	175	44359	2	195	1048576	0
$\sqrt{-\ln(x)}$	226	164944	8	230	164957	2	206	1114112	0

The domains of functions $f(x)$ are the same as Table 3.
#LEs: Number of logic elements. Memory : Memory size [bit].
#DSPs: Number of 9-bit $\times$ 9-bit DSP units.

smaller and faster than Arc_A and Arc_B. However, Arc_C cannot implement the square root or reciprocal functions using the FPGA due to the excessive size of the coefficients tables. In Table 5, for $\sqrt{x}$ and $\sqrt{-\ln(x)}$, Arc_C used large single look-up tables. From these results, we can see that Arc_C is suitable only for trigonometric functions, and is unsuitable for square root, reciprocal, etc.

Arc_B implements various functions with fewer DSP units than Arc_A, because Arc_B requires a smaller multiplier than Arc_A. Note that the FPGA synthesis system uses more DSP units for a multiplier with more bits. Thus, Arc_B offers a fast and compact implementation. In Arc_A, for all the functions except for $\sqrt{x}$, the multiplier has the longest delay time among all units. On the other hand, in Arc_B, for all the functions, the multiplier is not the slowest unit, and the coefficients table or the adder has the longest delay time among all units. For $\sqrt{x}$, Arc_A that has a smaller coefficients table was faster than Arc_B. From these results, we can conclude: 1) To implement a fast NFG with an FPGA, the size reduction of multiplier size is important. 2) Arc_B is the most efficient for various numerical functions among three architectures.

6.3. Comparison with an Existing Method

To show the efficiency of our automatic synthesis system, we compare our NFGs with ones reported in [8]. NFGs in [8] are also based on non-uniform segmentation, while they are designed by hand. We generated the NFGs with the same precision as [8]. Table 6 shows that our NFGs have comparable performances to [8]. Our system generated 24-bit precision NFGs with the operating frequency of more than 125 MHz for some functions in [12]. Due to the page limitation, the results are omitted.

7. CONCLUSION AND COMMENTS

We have proposed an architecture and a synthesis method for programmable NFGs for trigonometric functions, logarithm functions, square root, reciprocal, etc. Our architecture using an LUT cascade compactly realizes various numerical functions, and is suitable for automatic synthesis. Experimental results show that: 1) Our architecture efficiently implements NFGs for wide range of numerical functions; and 2) Our synthesis system generates the NFGs with comparable performance to those designed by hand.

Currently, we are working for the NFGs using the quadratic approximation algorithm to reduce the memory size.

Table 5. Comparison of performances for three architectures.

Precision, Accuracy: 16-bit precision, 15-bit accuracy
FPGA device: Altera Stratix (EP1S10F484C5)
Logic synthesis tool: Altera QuartusII 4.1 (default option)

Function $f(x)$	Arc_A			Arc_B			Arc_C		
	Freq.	#stages	Latency	Freq.	#stages	Latency	Freq.	#stages	Latency
$\sin(\pi x)$	124	7	56	185	8	43	188	3	16
$\cos(\pi x)$	126	8	64	187	9	48	184	4	22
$\tan(\pi x)$	125	7	56	190	9	47	183	3	16
$1/x$	125	8	64	179	9	50	–	4	–
$1/\sqrt{x}$	124	9	73	178	10	56	–	4	–
$\sqrt{x}$	182	8	44	179	9	50	–	1	–
$\sqrt{-\ln(x)}$	125	9	72	176	10	57	–	1	–

The domains of functions $f(x)$ are the same as Table 3.
"–" shows that the function could not be implemented.
Freq.: Operating frequency [MHz]. #stages: Number of pipeline stages.
Latency: [nsec].

Table 6. Performance comparison with existing method.

FPGA device: Xilinx Virtex-II (XC2V4000-6)
Logic synthesis tool: Xilinx ISE 6.3 (default option)

Function $f(x)$	Domain	In prec.		Out prec.		Our method			Method in [8]		
		Int	Frac	Int	Frac	Freq.	#stages	Latency	Freq.	#stages	Latency
$\sqrt{-\ln(x)}$	$(0,1]$	1	32	3	5	123	20	163	133	14	105
$\sin(2\pi x)$	$[0,1/4]$	0	16	1	8	153	10	65	133	14	105
$\cos(2\pi x)$	$[0,1/4]$	0	16	1	8	164	11	67	133	14	105

In prec.: Precision of input. Out prec.: Precision of output.
Int: n_int Frac: n_frac

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 Ministry of Education, Culture, Sports, Science, and Technology (MEXT) via Kitakyushu innovative cluster project, and NSA Contract RM A-54. We appreciate Dr. Marc D. Riedel for the work of [12].

8. 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 Array (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] 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. the 12th workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI'04)*, Kanazawa, Japan, pp. 466–471, Oct. 2004.

[4] 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.

[5] 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.

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

[7] 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 (ICCD'01)*, Austin, TX, pp. 388–393, Sept. 23–26, 2001.

[8] D.-U. Lee, W. 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.

[9] J.-M. Muller, *Elementary function: Algorithms and implementation*, Birkhauser Boston, Inc., Secaucus, NJ, 1997.

[10] S. Nagayama and T. Sasao, "Compact representations of logic functions using heterogeneous MDDs," *IEICE Trans. on fundamentals*, Vol. E86-A, No. 12, pp. 3168–3175, Dec. 2003.

[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] T. Sasao, J. T. Butler, and M. D. Riedel, "Application of LUT cascades to numerical function generators," *Proc. the 12th workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI'04)*, Kanazawa, Japan, pp. 422–429, Oct. 2004.

[13] T. Sasao, S. Nagayama, and J. T. Butler, "Error analysis for programmable numerical function generators," http://www.lsi-cad.com/Error-NFG/.

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

[15] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Trans. on Comp.*, Vol. 48, No. 8, pp. 842–847, Aug. 1999.

[16] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Comput.*, Vol. EC-820, No. 3, pp. 330–334, Sept. 1959.