PROGRAMMABLE NUMERICAL FUNCTION GENERATORS: ARCHITECTURES AND
SYNTHESIS METHOD

Tsutomu Sasao¹, Shinobu Nagayama², Jon T. Butler³

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 an 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, theCORDIC (COordinate Rotation Digital Computer) algorithm [1, 16] has been used. Although faster than software approaches, this 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 for NFGs (Numerical Function Generators) using linear approximations. By using the LUT cascade [7, 11], our architecture is sim-
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.
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. 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, \( \max f_g \) and \( \min f_g \) 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, \( \max f_g \) and \( \min f_g \) 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 \( \text{seg}_i \); thus, the segments generated by the segmentation algorithm are denoted by \( \text{seg}_0, \text{seg}_1, \ldots, \text{seg}_{n-1} \). For each \( \text{seg}_i \), 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:

\[
y = g_i(x) = c_{1i}x + c_{0i},
\]

where \( g_i(x) \) is the linear function for the segment \( \text{seg}_i \),

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

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 \( \text{seg}_i \) 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 \( s_i = \{ (\text{the most significant } (n - k) \text{ bits of } x) \times 2^k \} \), the index \( i \) of the segment \( \text{seg}_i \) 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 \( \text{seg}_i \). It realizes the segment index function \( \text{seg} - \text{func}(x) : B^m \to \{0, 1, \ldots, t - 1\} \) 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 $g(x) = c_1 x + c_0$, where $c_1 = \frac{e - f(e)}{s - f(s)}$, $c_0 = f(s) - c_1 s$.
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 $|max_{fg}| > |min_{fg}|$, and let $p = p_{min}$ otherwise.
5. Let $error = |max_{fg} - min_{fg}|/2$, and $v = (max_{fg} + min_{fg})/2$.
6. If $error \leq c$, 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.

![Segmentation algorithm](image)

Fig. 3. Three architectures for NFGs.

shown in Fig. 4 (b) to realize any $\text{seg}_f \text{unc}(x)$. It can be designed by functional decomposition using BDDs (Binary Decision Diagrams) representing $\text{seg}_f \text{unc}(x)$. 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.

**Theorem 4.1** [12] Let $\text{seg}_f \text{unc}(x)$ be a segment index function with $t$ segments. Then, there exists an LUT cascade for $\text{seg}_f \text{unc}(x)$ 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.

<table>
<thead>
<tr>
<th>Name of units</th>
<th>Pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. LUT cascade</td>
<td>( n_{cas} )</td>
</tr>
<tr>
<td>2. Coefficients table</td>
<td>1</td>
</tr>
<tr>
<td>3. Adder for ( P )</td>
<td></td>
</tr>
<tr>
<td>4. Multiplier for ( f_1 )</td>
<td></td>
</tr>
<tr>
<td>5. Shifter (optional)</td>
<td>0 or 1</td>
</tr>
<tr>
<td>6. Two's completer (optional)</td>
<td>0 or 1</td>
</tr>
<tr>
<td>7. Adder for ( c_1(x - s_i) + c_{2i} )</td>
<td>1</td>
</tr>
<tr>
<td>Total pipeline stages</td>
<td>( n_{cas} + 4 )</td>
</tr>
</tbody>
</table>

\( n_{cas} \): 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 \( c_1 \), we reduce the number of bits for \( c_1 \).

First, we consider the case where the absolute value of \( c_1 \) is large. When \(|c_1|\) is large, many bits are required to represent \( c_1 \) in binary fixed-point. To reduce the number of bits for such \( c_1 \), we use the scaling method shown in [8].

When \(|c_1|\) is large, we represent \( c_1 \) as \( c_1 = c_1 \times 2^{-l_1} \times 2^{l_2} \).

Instead of the original value of \( c_1 \), we store the values of \( c_1 \times 2^{-l_1} \) and \( l_2 \) in the coefficients table. In this case, the product \( c_1(x - s_i) \) is computed using the multiplier for \( c_1 \times 2^{-l_1} \times (x - s_i) \) and the shifter for an \( l_1 \)-bit shift to the left. The increase of \( l_1 \) reduces the number of bits to represent the value of \( c_1 \times 2^{-l_1} \), while increasing the rounding error. Our synthesis system finds the optimum value of \( l_1 \) for each segment \( \text{seg}_i \) within the acceptable error [13].

Next, we consider the case where the range of \( c_1 \) includes negative values. In this case, our synthesis system stores the absolute value of \( c_1 \), and the sign bit for \( c_1 \) separately in the coefficients table, and first uses the unsigned multiplier to compute \(|c_1| \times (x - s_i)\), and then a two’s complementer to produce the signed value with the sign bit. When \( c_1 \) is positive for all segments \( \text{seg}_i \), no two’s complementer is implemented. That is, \( c_1 \times (x - s_i) \) 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.

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:

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

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 \( 2^{-20} \). 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 \( \text{Arc}_A \), \( \text{Arc}_B \), and \( \text{Arc}_C \) 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 \( \text{Arc}_A \) and \( \text{Arc}_B \) use non-uniform segmentation, they implement various numerical functions with small coefficients tables. On the other hand, \( \text{Arc}_C \) uses uniform segmentation. Thus, although \( \text{Arc}_C \) 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 \( \text{Arc}_C \).

Tables 4 and 5 compare the amount of hardware and performances for the three architectures. These tables show that, for trigonometric functions, \( \text{Arc}_C \) implements the shortest latency and most compact NFGs among the three, since \( \text{Arc}_C \) requires no segment index encoder. Therefore, when the number of uniform segments is relatively small, \( \text{Arc}_C \) is
C is suitable only for trigonometric functions, and is unsuitable for square root, reciprocal, etc. Arc B requires a smaller multiplier than Arc A, because Arc B requires a smaller multiplier than Arc B. 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 \( \sqrt{\frac{x}{E}} \), 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 \( \sqrt{\frac{x}{E}} \), 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 the 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.

<table>
<thead>
<tr>
<th>Function $f(x)$</th>
<th>Arc_A</th>
<th>Arc_B</th>
<th>Arc_C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sin(\pi x)$</td>
<td>124</td>
<td>185</td>
<td>188</td>
</tr>
<tr>
<td>$\cos(\pi x)$</td>
<td>126</td>
<td>187</td>
<td>184</td>
</tr>
<tr>
<td>$\tan(\pi x)$</td>
<td>125</td>
<td>190</td>
<td>183</td>
</tr>
<tr>
<td>$1/x$</td>
<td>125</td>
<td>179</td>
<td>183</td>
</tr>
<tr>
<td>$1/\sqrt{x}$</td>
<td>124</td>
<td>178</td>
<td>188</td>
</tr>
<tr>
<td>$\sqrt{x}$</td>
<td>182</td>
<td>179</td>
<td>183</td>
</tr>
<tr>
<td>$\sqrt{-\ln(x)}$</td>
<td>125</td>
<td>176</td>
<td>188</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Function $f(x)$</th>
<th>Domain</th>
<th>In prec.</th>
<th>Out prec.</th>
<th>Our method</th>
<th>Method in [8]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sqrt{-\ln(x)}$</td>
<td>(0, 1)</td>
<td>1 32 3 5</td>
<td>123 20 163</td>
<td>133 14 105</td>
<td></td>
</tr>
<tr>
<td>$\sin(2\pi x)$</td>
<td>[0, 1/4]</td>
<td>0 16 1 8</td>
<td>153 10 63</td>
<td>133 14 105</td>
<td></td>
</tr>
<tr>
<td>$\cos(2\pi x)$</td>
<td>[0, 1/4]</td>
<td>0 16 1 8</td>
<td>164 11 67</td>
<td>133 14 105</td>
<td></td>
</tr>
</tbody>
</table>

In prec.: Precision of input. Out prec.: Precision of output. Int: $n_{\text{int}}$. Frac: $n_{\text{frac}}$.

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


