The Design of a 24-bit Hardware Gaussian Noise Generator via the Box-Muller Method and its Error Analysis

Lincoln Glauser
ldg4627@rit.edu

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Master’s Project is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
The Design of a 24-bit Hardware Gaussian Noise Generator via the Box-Muller Method and its Error Analysis

by

Lincoln Glauser

GRADUATE PAPER

Submitted in partial fulfillment
of the requirements for the degree of
MASTER OF SCIENCE
in Electrical Engineering

Approved by:

Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING

COLLEGE OF ENGINEERING

ROCHESTER INSTITUTE OF TECHNOLOGY

ROCHESTER, NEW YORK

AUGUST, 2017
I would like to dedicate this work to God, for without whom I would never have achieved anything, and to my family and friends for their love, support and prayers during this work.
Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Lincoln Glauser

August, 2017
Acknowledgements

I would like to thank my advisor, Mark A. “EEE” Indovina, for his time, wisdom and
sense of humor that helped me grow to love digital system design and gain a better
appreciation for the field. I would like to thank all the faculty at Rochester Institute of
Technology for preparing me to be a lifelong learner and equipping me to be a productive
asset to any company in my future career with an emphasis on Dr. Dorin Patru and
Professor James Moon.

I would like to thank the “Tight Squad”, my best friends that I met at RIT, Connor
“ConRail” and Matt “GTO”, and the rest of the wonderful classmates that I spent
endless hours in the labs with throughout my collegiate career. I would especially like
to thank Connor for his support in this work by proofreading, providing block diagrams
and providing Python scripts to facilitate the generation of code for the look up tables
and leading zero detectors.
The paper describes a Hardware Gaussian Noise Generator based on the Box-Muller Method as a Graduate Research project undertaken by Lincoln Glauser. Modern systems attempt to transmit data at ever increasing data rates in noisy environments, or store enormous amounts of data in a medium that is known to be unreliable. Designers therefore utilize error correcting codes to improve reliability. To validate operation of these systems, a high quality noise generator is required to introduce errors into the system. Mr. Glauser took the time to understand the mathematics behind Box-Muller Method and produce a 24-bit noise generator with excellent characteristics. I continue to be impressed by Mr. Glauser’s thoroughness and attention to detail, traits that will serve him well in his future career.

Mark A. Indovina

Rochester, NY USA

August 2017
Abstract

In regards to data transmission in communication systems, there is need for robust emulation of communication channels via Gaussian noise generation. Over time, larger sample sizes are desired to reach farther into the tail ends of the distribution and faster sample generation speeds are desired versus the software implementations. This paper proposes a Gaussian noise generator utilizing the Box-Muller method written in Verilog HDL targeting a 65nm ASIC process utilizing Synopsys Design Compiler. The design creates two 24-bit noise samples per clock cycle and each sample is accurate to one unit in the last place. A sample can represent up to $9.42\sigma$, which allows for a sample size of $2 \cdot 10^{20}$. The design generates 800 million samples/s at a clock frequency of 400MHz. After a thorough error analysis, a bit-exact model was created in MATLAB and a thorough probability and statistic analysis was executed on the generated sample sets.
## Contents

Acknowledgements iii
Forward iv
Abstract v
Contents vi
List of Figures viii
List of Listings ix
List of Tables x

1 Introduction 1

2 Background 4
  2.1 Basic Probability & Statistics ................................. 4
    2.1.1 Uniform Distribution Theory ............................ 5
    2.1.2 Normal Distribution Theory ............................. 7
    2.1.3 Goodness-of-fit (GoF) Techniques ...................... 8
      2.1.3.1 Uniform GoF ...................................... 9
      2.1.3.2 Normal GoF ...................................... 9
  2.2 Box-Muller Hardware Implementation ........................... 11
    2.2.1 Elementary Function Implementation .................... 11
      2.2.1.1 Introduction .................................... 11
      2.2.1.2 Chebyshev Series Approximation ................... 12
      2.2.1.3 Box-Muller Elementary Functions ................. 18
    2.2.2 Box-Muller Error Analysis ............................... 24
      2.2.2.1 MiniBit Bit-Width Optimization .................... 25
      2.2.2.2 General Analysis Walk Through .................... 29
    2.2.3 16-bit Error Analysis Walk Through .................... 40

3 LGBMGNG 24-bit Noise Error Analysis 47
4 Hardware Implementation
  4.1 Architecture .................................................. 53
  4.2 Design Choices ................................................ 55

5 Tests and Results
  5.1 Simulation Results ............................................. 60
  5.2 RTL Synthesis Results ......................................... 61
  5.3 Debugging LGBMGNG ............................................ 67
    5.3.1 2s Complement Multiplication ............................ 67
    5.3.2 Overflow .................................................. 67
    5.3.3 Redundancy Assumptions ................................ 68
    5.3.4 Rounding Schemes ........................................ 68
  5.4 Test Vector Probability and Statistic Analysis ............ 69

6 Conclusion
  6.1 Future Work ................................................... 72
    6.1.1 Pipeline Extension ...................................... 72
    6.1.2 Architectural Changes ................................... 73
    6.1.3 Approximation Changes .................................. 73
  6.2 Conclusions ................................................... 73

References .......................................................... 75

I Source Code
  I.1 LGBMGNG Design in Verilog HDL ............................. 80
  I.2 32-bit Tausworthe URNG ...................................... 102
  I.3 Coefficient Look up Table ................................... 104
  I.4 Leading Zero Detector ....................................... 107
  I.5 LGBMGNG Test Vector Testbench ............................ 109
List of Figures

2.1 Standard Uniform Distribution U(0,1) PDF ............................ 6
2.2 Standard Uniform Distribution U(0,1) CDF ............................ 6
2.3 Standard Normal Distribution N(0,1) PDF ............................ 7
2.4 Standard Normal Distribution N(0,1) CDF ............................ 8
2.5 Splitting of x for Function Approximation ............................ 14
2.6 Function Approximation on Individual Segments ...................... 14
2.7 Lee et al.’s 16-bit Noise Block Diagram [1] ............................ 42

4.1 LGBMGNG 24-bit Block Diagram ................................. 54
4.2 Degree One Approximation of SQRT Hardware Implementation .... 55
4.3 Degree Two Approximation of COS Hardware Implementation .... 56
4.4 Degree Three Approximation of LN Hardware Implementation .... 56
4.5 LGBMGNG Pipeline .................................................. 58

5.1 LN and SQRT Data Path Simulation Waveform ...................... 62
5.2 SIN/COS Data Path Simulation Waveform ......................... 63
5.3 1 Million Test Vectors Synthesis Simulation Waveform ............ 64
5.4 20 Million Test Vectors RTL Simulation Waveform .............. 64
# List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Box-Muller Method Pseudo Code</td>
<td>24</td>
</tr>
<tr>
<td>2.2</td>
<td>16-bit Box-Muller Method Pseudo Code</td>
<td>44</td>
</tr>
<tr>
<td>3.1</td>
<td>LGBMGNG, 24-bit Box-Muller Method Pseudo Code</td>
<td>50</td>
</tr>
<tr>
<td>I.1</td>
<td>LGBMGNG, 24-bit Box-Muller Implementation Source Code</td>
<td>102</td>
</tr>
<tr>
<td>I.2</td>
<td>TAUS, 32-bit Tausworthe URNG</td>
<td>104</td>
</tr>
<tr>
<td>I.3</td>
<td>Coefficient LUT Example-C2-COS</td>
<td>107</td>
</tr>
<tr>
<td>I.4</td>
<td>Leading Zero Example-47-bit</td>
<td>109</td>
</tr>
<tr>
<td>I.5</td>
<td>LGBMGNG, 24-bit Box-Muller Implementation Testbench</td>
<td>112</td>
</tr>
</tbody>
</table>
List of Tables

4.2 24-bit Operand Sizes ........................................ 59

5.1 The Seed Values for the Four Test Vector Sets ............ 61
5.2 Logic Synthesis Reports for Pre and Post Scan Chain Netlists ........ 64
5.3 Power Usage for Pre and Post Scan Chain Netlists ........ 65
5.4 Post-Scan Chain Cell Area Break-down ....................... 66
5.5 Goodness-of-Fit Metrics for 1 Million Count Test Vector Sets .... 70
5.6 Goodness-of-Fit Metrics for 20 Million Test Vector Sets .... 71
Chapter 1

Introduction

Gaussian noise, which represents a standard normal distribution, is a natural phenomenon that directly affects electronics ranging from the capturing of digital images utilizing sensors to the communication channels of all communicating protocols. The most interest in gaussian noise is in the verification of different low-density parity-check (LDPC) codes [2]. LDPC codes are linear error correcting codes that are designed for transmitting messages over transmission channels that have lots of interference due to noise. Their main function is to overcome noise and therefore, a method for generating noise is desired to provide verification. For this, it is desired to have noise samples at high rates for verification as well as high periodicity to reach the tail ends of the distribution. In addition to LDPC codes are turbo codes, which are also in need of the generation of gaussian noise [3]. Turbo codes are a type of forward error correction (FEC) codes that are used in telecommunications like 3G/4G and satellite communications as well. For gaussian noise generation, there are multiple ways developed where most stem off of uniform random numbers as inputs. There is the Ziggurat method [4], the Inversion method [5, 6], the Wallace method [7], the Box-Muller method [1, 8–10] and those methods implemented with algorithms like CORDIC [11–13]. The Box-Muller method is chosen for this paper due to the error analysis that was executed by Lee et
al. for their 16-bit gaussian noise generator [1]. The Box-Muller transformation can be seen via equations 1.1-1.6, developed by Box and Muller [14], where the uniform numbers, \( u_0 \) and \( u_1 \) are generated using Tausworthe uniform random generators developed by [15] and implemented in this paper following [16]. The implementation in hardware utilizes look up tables where the coefficients were found using Chebyshev series approximations [17]. This paper discusses the error analysis, design and implementation of a 24-bit gaussian noise generator (LGBMGNG). The main contribution of this design is the increase in size of the output noise and the larger periodicity compared to the 16-bit design mentioned above.

\[
e = -2 \cdot ln(u_0) \tag{1.1}
\]

\[
f = \sqrt{e} \tag{1.2}
\]

\[
g_0 = sin(2\pi \cdot u_1) \tag{1.3}
\]

\[
g_1 = cos(2\pi \cdot u_1) \tag{1.4}
\]

\[
x_0 = n_1 = f \cdot g_0 \tag{1.5}
\]

\[
x_1 = n_2 = f \cdot g_1 \tag{1.6}
\]

This paper is organized as follows. Chapter 2 discusses the application of probability and statistics to Gaussian noise generation, a general approach for realizing a Box-Muller generator in hardware, Chebyshev series approximations, a general process for the error analysis of a Box-Muller method and a walk through of that analysis for a 16-bit Gaussian noise generator designed by Lee et al. [1]. Chapter 3 applies the error analysis steps to a 24-bit Gaussian noise generator. Chapter 4 walks through the 24-bit Gaussian noise generator’s implementation in hardware. Chapter 5 discusses the relevant results of the hardware design, while walking through some debugging and analyzing the sample sets.
of noise generated by the 24-bit design. Finally, Chapter 6 describes possible future work and the conclusions of this work.
Chapter 2

Background

2.1 Basic Probability & Statistics

Together, Probability and Statistics provide great benefit to the understanding of different phenomena observed in everyday life. Probability focuses on the likeliness of outcomes happening, randomness and uncertainty. Meanwhile, statistics focuses on gleaning information, making judgments, and making decisions from the presence of data that includes variation and uncertainty. Applications for probability range from games of chance to gambling to machine learning to physics to game theory to computer science to random number generation and most importantly to modeling Gaussian noise. Applications for statistics range from component lifetime to resistor value tolerance to pH levels in soil specimens to motor vehicle emissions to metal corrosion to the stock market to just about any measured quantity and to better understanding Gaussian noise and analyzing its models[18]. For the majority of this paper, probability distributions is the main topic involving probability and statistics. A probability distribution is a term associated with the way the set of numbers in a given sample are aligned in terms of the probability of their occurrence. There are multiple different distributions utilized, but this paper will focus on the two most popular continuous distributions, uniform and nor-
mal, due to their direct use case in modeling Gaussian Noise. See Section 2.1.1 for a brief introduction to uniform distribution theory and see Section 2.1.2 for a brief introduction to normal distribution theory. Gaussian noise can be modeled as a normal distribution of random numbers. Lee et. al utilize the Box-Muller method for generating the normal distribution [1]. For this method of generating a normal distribution, two independent uniformly distributed random numbers are utilized as inputs to the method. The inputs are then transformed into two normally distributed samples.

2.1.1 Uniform Distribution Theory

A uniform distribution of continuous random variable X on interval \([a, b]\) produces the probability distribution function (PDF) found in Equation 2.1 and the cumulative distribution function (CDF) found in Equation 2.2 [18]. For this paper, the uniform distribution will be utilized with a lower bound \((a)\) of the value 0 and a upper bound \((b)\) of the value 1, which is usually regarded as a standard uniform distribution and uses the notation \(U(0,1)\). To visualize these two functions for a standard uniform distribution, see Figures 2.1 and 2.2.

\[
f(x; a, b) = \begin{cases} 
1/\left(b-a\right) & a \leq x \leq b \\
0 & \text{otherwise} 
\end{cases}
\]  

(2.1)

\[
F(x; a, b) = \begin{cases} 
0 & x < A \\
\frac{x-a}{b-a} & a \leq x < b \\
1 & x \geq b 
\end{cases}
\]  

(2.2)
Figure 2.1: Standard Uniform Distribution $U(0,1)$ PDF

Figure 2.2: Standard Uniform Distribution $U(0,1)$ CDF
2.1.2 Normal Distribution Theory

A normal distribution of continuous random variable $X$ with mean $\mu$ and variance $\sigma$ produces the PDF found in Equation 2.3 and the CDF found in Equation 2.5 [18]. Equation 2.5 is derived from Equation 2.3 and the error function found in Equation 2.4 [19]. For this paper, the normal distribution is utilized with a mean $\mu$ of the value 0 and standard deviation $\sigma$ of the value 1, which is usually regarded as a standard normal distribution and uses the notation $N(0,1)$. To visualize these two functions for a standard normal distribution, see Figures 2.3 and 2.4.

\[
f(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad -\infty \leq x \leq \infty \tag{2.3}
\]

\[
erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \quad -\infty < x < \infty \tag{2.4}
\]
2.1 Basic Probability & Statistics

\[ F(x) = \frac{1}{2} \left[ 1 + erf \left( \frac{x - \mu}{\sigma \sqrt{2}} \right) \right] \] (2.5)

2.1.3 Goodness-of-fit (GoF) Techniques

D’Agostino and Stephens describe goodness-of-fit techniques as a series of methodologies for associating a set of data to a specific distribution for its population \[20\]. These techniques can be utilized on just about any set of data, but they are all specifically created for unique test cases. Some techniques are made for univariate data while others are for multivariate data; univariate focuses on one random variable and multivariate focuses on multiple random variables. For most tests there is a corresponding hypothesis given that the sample data will follow a specific distribution and the tests will either prove or reject the hypothesis. The goodness-of-fit techniques for this paper are applied for testing the quality of sample sets for fitting either the standard uniform distribution or the standard normal distribution. General tests for the standard uniform distribution
that are utilized in this paper can be found in Section 2.1.3.1 with a focus on the well known TestU01 software library [21]. General goodness-of-fit techniques and tests for the standard normal distribution that are utilized in this paper can be found in Section 2.1.3.2 with a focus on the popular Chi-Squared and Anderson-Darling tests.

2.1.3.1 Uniform GoF

There are multiple goodness-of-fit techniques developed for standard uniform distribution hypothesis testing. To test the Tausworthe uniform random number generator (TAUS) and disapprove of a traditional linear feedback shift register generators, Lee et al. utilize the well known Diehard tests [1]. The algorithm Lee et al. found for TAUS was developed by Pierre L’Ecuyer who then later co-authored a software suite named TestU01 that is written in ANSI C. TestU01 incorporates the well known Diehard tests and also builds upon it with multiple tests that are proved to be more effective in guaranteeing the sample set is of a standard uniform distribution [21]. For this paper, the TestU01’s battery of tests called SmallCrush are utilized to approve of generated sample sets that pertain to a standard uniform distribution.

2.1.3.2 Normal GoF

There are multiple goodness-of-fit techniques developed for normal or Gaussian distribution hypothesis testing. The Chi-Squared test is a very general test; D’Agostino and Stephens highly recommend the Anderson-Darling test as a very effective omnibus test and they also highly recommend detailed graphical analysis involving the corresponding normal probability plot[18][20]. There are multiple other tests similar to the Kolmogorov-Smirnov test, but D’Agostino and Stephens prove they are only of historical curiosity and are less powerful when compared to their recommended ones with an emphasis on the Anderson-Darling test [20].
Chi-Squared  The Chi-Squared test is a very popular and general test developed by Karl Pearson in 1900 [20]. The test applies to data that is of both univariate or multivariate and both discrete or continuous. In general, the test emphasizes the difference between observed cell counts and the expected values under the specific hypothesized distribution to analyze the goodness of fit for the sample data. The Chi-Squared test is utilized in this paper for the standard normal distribution hypothesis testing by using the chi2gof function in MATLAB [22].

Anderson-Darling  The Anderson-Darling test is a test that is computed utilizing a formula based off of the empirical distribution function (EDF) statistic. The EDF statistic is a measurement of the difference between \( F_n(x) \) and \( F(x) \) where \( F_n(x) \) is a measurement of the proportion of observations for values less than or equal to \( x \) for a step function and \( F(x) \) is the specific probability for that corresponding observation of values less than or equal to \( x \) [20]. The Anderson-Darling test is utilized in this paper for the standard normal distribution hypothesis testing by using the adtest function in MATLAB [23].

Normal Probability Plot Analysis  Graphical and numerical analysis of a normal probability plot both visually and numerically display features of the data, where the numerical analysis quantifies the association of the data to the particular distribution [20]. A probability plot will be a straight line if the corresponding data fits the distribution, which is very easy to visualize and standard linear regression analyses can be executed to obtain numerical metrics for the sample data [20]. The Coefficient of Determination \( R^2 \) and Standard Error of the Estimate (Mean squared error or SEE or \( S \)) are utilized in this paper for numerically analyzing the normal probability plot. Both analyses use \( y_{fit} \), an approximated line fit to the \( y \) and \( x \) data presented, where the \( y \) values are the sorted array of the random variable samples and the \( x \) values are the
constant values associated with the normal distribution. Definitions of both analyses are found in Equations 2.6 and 2.7.

\[ R^2 = \frac{\sum (y_{\text{fit}} - \bar{y})^2}{\sum (y - \bar{y})^2} \]  
\[ \text{SEE} = \sqrt{\frac{\sum (\hat{y} - y)^2}{n - 2}} \]  

2.2 Box-Muller Hardware Implementation

2.2.1 Elementary Function Implementation

2.2.1.1 Introduction

Elementary transcendental functions are those functions that are not algebraic. There are four elementary functions utilized in the Box-Muller method, natural log (LN), square root (SQRT), sine and cosine (SIN/COS). As [17] explains, hardware implementations of elementary functions may obtain the speed improvements desired, but the results are less accurate. Schulte et al. then propose executing function approximation by Chebyshev series approximation and performing transformations on the coefficients to provide exactly rounded results for reciprocal, square root, \( e^x \), and \( \log_2(x) \) [17]. The coefficient transformation method for exactly rounded results is only guaranteed for lower precisions, (eg. 24-bit fractional portion) and does not work for either sine or cosine. Therefore, this paper utilizes the Chebyshev series approximation that they used and rounds the transcendental functions to nearest (round-to-nearest), which will introduce an error of up to one unit in the last place (ulp). The lack of exact rounding hardware is mostly due to the Table Maker’s Dilemma [24] that Schulte et al. and Lee et al. reference, where the problem occurs from not being able to determine the required
accuracy for each part to ensure exactly rounded results (0.5 ulp)[17][1]. On the other hand, round-to-nearest (1 ulp) requires only the addition of one bit to the result and will simplify the hardware design.

Function evaluation of hardware is generally split into three steps: range reduction, function approximation and range reconstruction. These steps are utilized because it facilitates the implementation by lowering the complexity to only approximating the function on a smaller input interval. As the titles of the steps suggest, the first is to reduce the range of the input to the specific range the function was approximated to, approximate the reduced input and then reconstruct the output back into the original range. The range reduction and range reconstruction steps are unique to each elementary function and the unique range that those functions are implemented on.

The individual function approximations are implemented by using Chebyshev series approximation. The specific input range and mathematical identities stem from the work done by Walther who provides prescaling mathematical identities for many elementary functions [25]. Section 2.2.1.2 walks through the Chebyshev series approximation, the derivation of the coefficient values for degree one, two and three function approximations, and the transformation of those coefficients to use the corresponding $x_i$ instead of $x$. Section 2.2.1.3 walks through the specific range reduction, function approximation and range reconstruction steps of the elementary functions implemented in the Box-Muller method of generating Gaussian noise.

2.2.1.2 Chebyshev Series Approximation

This section describes the steps found for function approximation in [17] and [26], the coefficient extraction from the approximation and the transformation of the coefficients to the specific input range as seen in [1]. For function approximation via a polynomial,
2.2 Box-Muller Hardware Implementation

there is the following form:

\[ q_{n-1}(x) \approx C_0 + C_1 \cdot x + \cdots + C_{n-1} \cdot x^{n-1} = \sum_{i=0}^{n-1} C_i \cdot x \quad (2.8) \]

where \( q_{n-1}(x) \) is a polynomial of degree \( n-1 \) and \( C_i \) are the coefficients corresponding to each term. The function is approximated then on the input range \([x_{\text{min}}, x_{\text{max}}]\).

To then increase the accuracy of the polynomial approximation while also minimizing the degree of approximation, the input range is split into \( 2^k \) segments, where \( x_m \) represents the segment number, and \( x_l \) represents the value within that segment. The function is then approximated on each new segment or input interval, where the coefficients for that section are indexed by the \( x_m \) value and the \( x \) value is the \( x_l \) value. For example, if \( x \) is on the \([0,1)\) input interval, then there is the following relationship:

\[ x = x_m + x_l \cdot 2^{-k} \quad (2.9) \]

where \( x_m \) is \([0,1)\) and \( x_l \) is \([0,1)\). Figure 2.5 shows the splitting of the \( p \)-bit \( x \) value into the \( k \)-bit \( x_m \) and \((p-k)\)-bit \( x_l \) values for \( 2^k \) segments. Figure 2.6 shows a visual representation of a function, \( f(x) \), approximated on each sub-interval with corresponding approximated polynomials, \( p_m(x) \). The full interval approximation can then be seen in 2.10, where the coefficients, \( C_i \), are indexed off of the corresponding \( x_m \) value.

\[ p_m(x) = C_0(x_m) + C_1(x_m) \cdot x_l + \cdots + C_{n-1}(x_m) \cdot x_l^{n-1} = \sum_{i=0}^{n-1} C_i(x_m) \cdot x_l^i \quad (2.10) \]

The method for obtaining the coefficient values is through a Chebyshev series approximation with input interval \([a,b)\) and degree of approximation \( n-1 \) by the following algorithm.

1. Chebyshev nodes are created on the interval \([-1,1)\) via 2.11, where \( t_i \) is classified
2.2 Box-Muller Hardware Implementation

Figure 2.5: Splitting of x for Function Approximation

\[ t_i = \cos\left( \frac{(2 \cdot i + 1) \pi}{2 \cdot n} \right) \quad (0 \leq i < n) \] (2.11)
2. Then the Chebyshev nodes, $t_i$, are transformed from the [-1,1) input interval to [a,b) through 2.12, where a and b are the values that are unique depending on the number of segments and the input range of the original function.

$$x_i = \frac{t_i(b-a) + (b+a)}{2} \quad (0 \leq i < n) \quad (2.12)$$

3. The individual interval Lagrange polynomial $p_m(x)$ is formed by interpolating the Chebyshev nodes in 2.13, where the $L_i$ values are created via 2.14 and the $y_i$ values are created via 2.15.

$$p_m(x) = y_0 \cdot L_0(x) + y_1 \cdot L_1(x) + \cdots + y_{n-1} \cdot L_{n-1}(x) = \sum_{i=0}^{n-1} y_i \cdot L_i(x) \quad (2.13)$$

$$L_i(x) = \frac{(x - x_0) \cdots (x - x_{i-1}) \cdot (x - x_{i+1}) \cdots (x - x_{n-1})}{(x_i - x_0) \cdots (x_i - x_{i-1}) \cdot (x_i - x_{i+1}) \cdots (x_i - x_{n-1})} = \prod_{k=0, k \neq i}^{n-1} \frac{x - x_k}{x_i - x_k} \quad (2.14)$$

$$y_i = f(x_i) \quad (2.15)$$

4. The overall $p_m(x)$ is created by extracting the coefficients $C_i$ from the individual segment approximations. This paper works with polynomial approximations for degree one, two and three. The coefficient extraction for each degree of approximation can be found via the following equations.

(a) Degree One Approximation:

$$C_1 = \frac{f(x_0)}{(x_0 - x_1)} + \frac{f(x_1)}{(x_1 - x_0)} \quad (2.16)$$
\[
C_0 = \frac{-f(x_0) \cdot x_1}{(x_0 - x_1)} + \frac{-f(x_1) \cdot x_0}{(x_1 - x_0)}
\] (2.17)

\[
p_m(x) = C_0 + C_1 \cdot x
\] (2.18)

(b) Degree Two Approximation:

\[
C_2 = \frac{f(x_0)}{(x_0 - x_1)(x_0 - x_2)} + \frac{f(x_1)}{(x_1 - x_0)(x_1 - x_2)} + \frac{f(x_2)}{(x_2 - x_0)(x_2 - x_1)}
\] (2.19)

\[
C_1 = \frac{f(x_0) \cdot (-x_1 - x_2)}{(x_0 - x_1)(x_0 - x_2)} + \frac{f(x_1) \cdot (-x_0 - x_2)}{(x_1 - x_0)(x_1 - x_2)} + \frac{f(x_2) \cdot (-x_0 - x_1)}{(x_2 - x_0)(x_2 - x_1)}
\] (2.20)

\[
C_0 = \frac{f(x_0) \cdot x_1 \cdot x_2}{(x_0 - x_1)(x_0 - x_2)} + \frac{f(x_1) \cdot x_0 \cdot x_2}{(x_1 - x_0)(x_1 - x_2)} + \frac{f(x_2) \cdot x_0 \cdot x_1}{(x_2 - x_0)(x_2 - x_1)}
\] (2.21)

\[
p_m(x) = C_0 + C_1 \cdot x + C_2 \cdot x^2
\] (2.22)

(c) Degree Three Approximation:

\[
C_3 = \frac{f(x_0)}{(x_0 - x_1)(x_0 - x_2)(x_0 - x_3)} + \frac{f(x_1)}{(x_1 - x_0)(x_1 - x_2)(x_1 - x_3)} + \frac{f(x_2)}{(x_2 - x_0)(x_2 - x_1)(x_2 - x_3)} + \frac{f(x_3)}{(x_3 - x_0)(x_3 - x_1)(x_3 - x_2)}
\] (2.23)
2.2 Box-Muller Hardware Implementation

\[ C_2 = \frac{f(x_0) \cdot (-x_1 - x_2 - x_3)}{(x_0 - x_1) (x_0 - x_2) (x_0 - x_3)} + \frac{f(x_1) \cdot (-x_0 - x_2 - x_3)}{(x_1 - x_0) (x_1 - x_2) (x_1 - x_3)} + (2.24) \]
\[ = \frac{f(x_2) \cdot (-x_0 - x_1 - x_3)}{(x_2 - x_0) (x_2 - x_1) (x_2 - x_3)} + \frac{f(x_3) \cdot (-x_0 - x_1 - x_2)}{(x_3 - x_0) (x_3 - x_1) (x_3 - x_2)} \]

\[ C_1 = \frac{f(x_0) \cdot (x_1 \cdot (x_2 + x_3) + x_2 \cdot x_3)}{(x_0 - x_1) (x_0 - x_2) (x_0 - x_3)} + \frac{f(x_1) \cdot (x_0 \cdot (x_2 + x_3) + x_2 \cdot x_3)}{(x_1 - x_0) (x_1 - x_2) (x_1 - x_3)} + \]
\[ = \frac{f(x_2) \cdot (x_0 \cdot (x_1 + x_3) + x_1 \cdot x_3)}{(x_2 - x_0) (x_2 - x_1) (x_2 - x_3)} + \frac{f(x_3) \cdot (x_0 \cdot (x_1 + x_2) + x_1 \cdot x_2)}{(x_3 - x_0) (x_3 - x_1) (x_3 - x_2)} \]

\[ C_0 = \frac{f(x_0) \cdot (-x_1 \cdot x_2 \cdot x_3)}{(x_0 - x_1) (x_0 - x_2) (x_0 - x_3)} + \frac{f(x_1) \cdot (-x_0 \cdot x_2 \cdot x_3)}{(x_1 - x_0) (x_1 - x_2) (x_1 - x_3)} + \]
\[ = \frac{f(x_2) \cdot (-x_0 \cdot x_1 \cdot x_3)}{(x_2 - x_0) (x_2 - x_1) (x_2 - x_3)} + \frac{f(x_3) \cdot (-x_0 \cdot x_1 \cdot x_2)}{(x_3 - x_0) (x_3 - x_1) (x_3 - x_2)} \]

\[ p_m(x) = C_0 + C_1 \cdot x + C_2 \cdot x^2 + C_3 \cdot x^3 \quad (2.27) \]

5. The coefficients \( C_i(x_m) \) have values for the input \( x \) and then are transformed for having values for the input \( x_i \). This transformation process is dependent on the input interval to the function approximation and will be discussed in Section 2.2.1.3.

6. The coefficients \( C_i(x_m) \) then are rounded to the specific precision desired due to the error analysis and as per 2.10, \( p_m(x) \) is assembled.

The maximum error between the original function and its Chebyshev series approximation for the interval \([a,b] \) according to [17] is in the form of 2.28, where \( \xi \) is the point on the interval \([a,b] \) where the \( n \)th derivative of the original function, \( f(x) \), has
its maximum value.

\[
E_n(x) = \left(\frac{b - a}{4}\right)^n \cdot \frac{2 \cdot |f^n(\xi)|}{n!} \quad a \leq \xi < b
\] (2.28)

Throughout this paper, the input interval \([a, b)\) is split into \(2^k\) equal segments. Therefore, Equation 2.28 is utilized to determine a value of \(k\). A simple method is to cycle through the first three degrees of approximation, where \(k\) is increased until the degree of approximation reaches the error requirement of \(E_{\text{approx}}\) being less than 0.5 ulp of \(y\).

### 2.2.1.3 Box-Muller Elementary Functions

The four elementary functions within the Box-Muller method are sine (SIN), cosine (COS), square root (SQRT) and natural logarithm (LN). The function approximation of these elementary functions is executed utilizing uniform segments and Chebyshev series approximation found in Section 2.2.1.2. For the approximation, Walther provides prescaling mathematical identities for all four elementary functions, where sine can be found in 2.29, cosine in 2.30, square root in 2.31 and natural log in 2.32.

\[
sin\left((Q + D) \pi 2\right) = \begin{cases} 
sin(D \frac{\pi}{2}) & \text{if } Q \mod 4 = 0 \\
\cos(D \frac{\pi}{2}) & \text{if } Q \mod 4 = 1 \\
-sin(D \frac{\pi}{2}) & \text{if } Q \mod 4 = 2 \\
-cos(D \frac{\pi}{2}) & \text{if } Q \mod 4 = 3 
\end{cases}
\] (2.29)

\[
cos\left((Q + D) \pi 2\right) = \begin{cases} 
\cos(D \frac{\pi}{2}) & \text{if } Q \mod 4 = 0 \\
-sin(D \frac{\pi}{2}) & \text{if } Q \mod 4 = 1 \\
-cos(D \frac{\pi}{2}) & \text{if } Q \mod 4 = 2 \\
sin(D \frac{\pi}{2}) & \text{if } Q \mod 4 = 3 
\end{cases}
\] (2.30)
2.2 Box-Muller Hardware Implementation

\[
\sqrt{M_x \cdot 2^{E_x}} = \begin{cases} 
2^{E_x} \sqrt{M_x} & \text{if } E_x \mod 1 = 0 \\
2^{E_x+1} \sqrt{\frac{M_x}{2}} & \text{if } E_x \mod 1 = 1 
\end{cases} \tag{2.31}
\]

\[ln(M_x \cdot 2^{E_x}) = ln(M_x) + E_x \cdot ln(2) \tag{2.32}\]

In 2.29 and 2.30, \(Q\) corresponds to the particular quadrant the function is in and \(D\) is \([0,1)\), which corresponds to the input value within quadrant \(Q\). For the Box-Muller method the \(Q\) can be designed to be the 2 most significant bits of \(u1\) and the rest of \(u1\) can be scaled to \([0,1)\) to work as the \(D\) value. For both identities, only \(cos(x \cdot \frac{\pi}{2})\) needs to be approximated, where \(x\) is either \(D\) (approximating \(cos(D \frac{\pi}{2})\)) or \(1-D\) (approximating \(sin(D \frac{\pi}{2})\)) to calculate both the sine and cosine values for every quadrant. Therefore, for the sine and cosine approximation, \(cos(x \cdot \frac{\pi}{2})\) is approximated, where \(x\) is \([0,1)\). For the Chebyshev series approximation in Section 2.2.1.2, \(x\) will range from \([0,1)\) as per 2.33, where \(x_l\) is of the form found in 2.34. The coefficients for the approximation are then found depending on the degree of approximation utilizing the method in Section 2.2.1.2 and the coefficients are transformed to be a function of \(x_l\) by the equations starting at 2.35 and 2.37, respectively, for approximations of degree one and two, for \(x\) on \([0,1)\).

\[x = x_m + x_l \cdot 2^{-k} \tag{2.33} \]

\[x_l = (x - x_m) \cdot 2^k \tag{2.34} \]

**Degree One Coefficient Transformations from \(x\) to \(x_l\) for \([0,1)\) as per 2.34:**

\[\bar{C}_1 = C_1 \cdot 2^{-k} \tag{2.35} \]

\[\bar{C}_0 = C_1 \cdot x_m + C_0 \tag{2.36} \]
Degree Two Coefficient Transformations from $x$ to $x_l$ for $x \in [0,1)$ as per 2.34:

$$\tilde{C}_2 = C_2 \cdot 2^{-2k} \quad (2.37)$$

$$\tilde{C}_1 = (2 \cdot C_2 \cdot x_m + C_1) \cdot 2^{-k} \quad (2.38)$$

$$\tilde{C}_0 = C_2 \cdot x_m^2 + C_1 \cdot x_m + C_0 \quad (2.39)$$

In Equation 2.32, $M_x$ represents the Mantissa and $E_x$ represents the Exponent for floating point representation; where the sign bit is not taken into consideration since $LN$ is always positive. The function to approximate is $\ln(M_x)$, since $\ln(2)$ is a constant and $E_x$ is an integer. Since $M_x$ is in floating point representation, it will range on $[1,2)$ where only the fractional section is included in $x_m$ and $x_l$ values as seen by 2.40 and $x_l$ is in the form found in 2.41. The coefficients are then found via Section 2.2.1.2 depending on the degree of approximation and the coefficients are transformed to be a function of $x_l$ by the equations starting at 2.42, 2.44 and 2.47 for approximations of degree one, two and three respectively for $x$ on $[1,2)$.

$$x = 1 + x_m + x_l \cdot 2^{-k} \quad (2.40)$$

$$x_l = (x - 1 - x_m) \cdot 2^k \quad (2.41)$$

Degree One Coefficient Transformations from $x$ to $x_l$ for $x \in [1,2)$ as per 2.41:

$$\tilde{C}_1 = C_1 \cdot 2^{-k} \quad (2.42)$$

$$\tilde{C}_0 = C_1 \cdot (x_m + 1) + C_0 \quad (2.43)$$
2.2 Box-Muller Hardware Implementation

Degree Two Coefficient Transformations from $x$ to $x_l$ for $x \in (1,2)$ as per 2.41:

$$\bar{C}_2 = C_2 \cdot 2^{-2^k}$$ \hspace{1cm} (2.44)

$$\bar{C}_1 = (C_2 \cdot (2 + 2 \cdot x_m) + C_1) \cdot 2^{-k}$$ \hspace{1cm} (2.45)

$$\bar{C}_0 = C_2 \cdot (x_m^2 + 2 \cdot x_m + 1) + C_1 \cdot (x_m + 1) + C_0$$ \hspace{1cm} (2.46)

Degree Three Coefficient Transformations from $x$ to $x_l$ for $x \in (1,2)$ as per 2.41:

$$\bar{C}_3 = C_3 \cdot 2^{-3^k}$$ \hspace{1cm} (2.47)

$$\bar{C}_2 = (3 \cdot C_3 \cdot x_m + 3 \cdot C_3 + C_2) \cdot 2^{-2^k}$$ \hspace{1cm} (2.48)

$$\bar{C}_1 = \left(3 \cdot C_3 \cdot x_m^2 + 2 \cdot (3 \cdot C_3 + C_2) \cdot x_m + 3 \cdot C_3 + 2 \cdot C_2 + C_1\right) \cdot 2^{-k}$$ \hspace{1cm} (2.49)

$$\bar{C}_0 = C_3 \cdot x_m^3 + (3 \cdot C_3 + 2 \cdot C_2) \cdot x_m^2 + (3 \cdot C_3 + 2 \cdot C_2 + C_1) \cdot x_m + C_3 + C_2 + C_1 + C_0$$ \hspace{1cm} (2.50)

Equation 2.31 for SQRT follows the same form as LN, where $M_x$ and $E_x$ are floating point values and there is no need to store the sign bit. The function to approximate is however now distributed into two different input intervals due to $M_x$ and $M_x/2$ being the two inputs. Therefore, $M_x$ is always scaled to $[2,4)$ and if $E_x$ is odd, the input interval becomes $[1,2)$ due to $M_x/2$. Accordingly, the function $\sqrt{x}$ needs to be approximated for two different intervals, $[1,2)$ and $[2,4)$, where each interval will have its own coefficients. The process for the interval of $[1,2)$ will be exactly the same as LN, but the $[2,4)$ case will need to utilize unique equations for transforming the coefficients to $x_l$. For $[2,4)$,
the way input $x$ is split into $x_m$ and $x_l$ can be seen in 2.51, where the $x_l$ is of form 2.52 and the equations for transforming the coefficients to $x_l$ can be found starting at 2.53 for a degree one approximation.

$$x = 2 \cdot \left(1 + x_m + x_l \cdot 2^{-k}\right) = 2 + 2 \cdot x_m + x_l \cdot 2^{-k+1}$$

(2.51)

$$x_l = \left(\frac{x}{2} - 1 - x_m\right) \cdot 2^k$$

(2.52)

**Degree One Coefficient Transformations from $x$ to $x_l$ for $x [2,4)$ as per 2.52:**

$$\bar{C}_1 = C_1 \cdot 2^{-k+1}$$

(2.53)

$$\bar{C}_0 = C_1 \cdot (2 + 2 \cdot x_m) + C_0$$

(2.54)

The range reduction and range reconstruction steps are performed following the methods determined by Lee et al. along with the hardware approximation method for the elementary functions as described above. There are two inputs to the Box-Muller, $u_0$ and $u_1$, where $u_0$ is the input to LN and $u_1$ is the input to COS/SIN and both inputs are interpreted as $[0,1)$. For COS/SIN, range reduction involves extracting the $Q$ and $D$ values from $u_1$, where $Q$ is the 2 most significant bits corresponding to the quadrant, and $D$ is the rest of $u_1$ scaled to $[0,1)$. For LN, range reduction involves fixed point to floating point conversion, where the resulting $M_x$ is $[1,2)$. For SQRT, range reduction involves fixed point to floating point conversion, except the resulting $M_x$ is $[2,4)$ and then depending on whether $E_x$ is even or odd, the $M_x$ value is either kept at $[2,4)$ (even $E_x$) or shifted to $[1,2)$ (odd $E_x$). At this stage, all three equations obtain their corresponding $x_m$ and $x_l$ values depending on the input range and specific segment count for the approximation. After the approximation is evaluated with $x_l$ as the input
value and \( x_m \) as the index for the corresponding coefficient values for the segment specific approximation, range reconstruction is to be executed. For COS/SIN, depending on the quadrant \( Q \), the positive or negative output values of either \( \cos(D \pi/2) \) or \( \cos((1 - D) \pi/2) \) are selected as per 2.29 and 2.30. For LN, as per 2.32, \( E \cdot \ln(2) \) is evaluated and joined with the approximation result to obtain the overall result. For SQRT, as per 2.31, \( 2^{E/2} \) (\( E \) is even) or \( 2^{(E+1)/2} \) (\( E \) is odd) are evaluated and multiplied to the result of the approximation to obtain the overall result. The combination of range reduction, polynomial approximation and range reconstruction for each function within the Box-Muller method can be further understood by viewing the pseudo code found in Listing 2.1.

```
%%Box-Muller Method Pseudo Code
%%--------------- Generate u0 and u1 ---------------
u0 = U(0,1); % TAUS generated
u1 = U(0,1); % TAUS generated

%%------------- Evaluate e = -2ln(u0) --------------
exp_e = LeadingZeroDetector(u0) + 1;
x_e = u0 << exp_e;

% Approximate ln(x_e) where x_e = [1,2)
% Degree-(n-1) piecewise polynomial - 2^{k_e} segments
% x_e is [1,2) --> x_e = 1 + x_m * x_l * 2^{-k_e}
% example is Degree-1
y_e = C1_e(x_m_e_index)*x_l_e + C0_e(x_m_e_index);

%%--------------- Evaluate f = sqrt (e) ------------
ln2 = ln(2);
e' = exp_e*ln2;
e = (e'-y_e)<<1;

%%----------------- Evaluate f = sqrt (e) ------------
exp_f = Offset-LeadingZeroDetector(e); % Note: Offset=IBe-2;
x_f' = e >> exp_f;
x_f = if(exp_f[0], x_f'>>1, x_f');
```
2.2 Box-Muller Hardware Implementation

% Approximate sqrt (x_f) where x_f = [1,4)
30  % x_f is [1,2) --> x_f = 1 + x_m_f + x_l_f * 2^-k_f
31  % or
32  % x_f is [2,4) --> x_f = 2 + 2*x_m_f + x_l_f * 2^(-k_f+1)
33  % Degree-(n-1) piecewise polynomial - 2^k_f segments
34  % example is Degree-1
35  y_f = C1_f(x_m_f_index)*x_l_f + C0_f(x_m_f_index);
36
37  % Range Reconstruction
38  exp_f' = if(exp_f[0], exp_f+1>>1, exp>>1);
39  f = y_f << exp_f';
40
41  % %------------- Evaluate g0=sin(2*pi*u1) -------------
42  % %------------- g1=cos(2*pi*u1) -------------
43  % Range Reduction
44  quadrant = u1[MSB:MSB-1];
45  x_g_a = u1[MSB-2:0];
46  x_g_b = (1-2^-(MSB-1))-u1[MSB-2:0];
47
48  % Approximate cos(x_g_a*pi/2) and cos(x_g_b*pi/2)
49  % where x_g_a, x_g_b = [0,1-2^-(MSB-1)]
50  % Degree-(n-1) piecewise polynomial - 2^k_g segments
51  % x_g is [0,1) --> x_g = x_m_g + x_l_g * 2^-k_g
52  % example is Degree-2
53  y_g_a = C2_g(x_m_g_a_index)*x_l_g_a^2 + C1_g(x_m_g_a_index)*x_l_g_a + C0_g(x_m_g_a_index);
54  y_g_b = C2_g(x_m_g_b_index)*x_l_g_b^2 + C1_g(x_m_g_b_index)*x_l_g_b + C0_g(x_m_g_b_index);
55
56  % Range Reconstruction
57  switch(quadrant)
58      case 0: g0 = y_g_b;  g1 = y_g_a;  % [0, pi/2)
59      case 1: g0 = y_g_a;  g1 = -y_g_b; % [pi/2, pi)
60      case 2: g0 = -y_g_b; g1 = -y_g_a; % [pi, 3*pi/2)
61      case 3: g0 = -y_g_a; g1 = y_g_b;  % [3*pi/2, 2*pi)
62  end
63
64  x0 = f*g0; x1 = f*g1;

---

Listing 2.1: Box-Muller Method Pseudo Code

2.2.2 Box-Muller Error Analysis

One of the main contributions of this paper is the execution of the analysis that is based off of the work of Lee et al. [1]. This section walks through the error analysis in a general process that is specific to the Box-Muller method of generating Gaussian noise and the architecture that is associated with using 32-bit Tausworthe uniform random
number generators. This section explains the MiniBit bit-width optimization approach developed by Lee et al. and walks through the general error analysis process [27].

### 2.2.2.1 MiniBit Bit-Width Optimization

This section describes the MiniBit bit-width optimization approach developed in [27] and used by [1], which is the main foundation for the error analysis of the Box-Muller method in hardware because it is a fixed-point design. From a high-level, the approach abstracts the error created by the quantization of signals over multiple math operations into a function of error at the output. In this function, the fractional bit sizes of the signals are used as the function variables. This abstraction turns the error analysis into an optimization problem, where the number of fractional bits for all the signals can be minimized while meeting the error requirement. The quantization of the signals to finite precision can be done in multiple ways (eg. round to zero, round away from zero, round towards infinity, rounds toward negative infinity...), but the main two ways in hardware designs are truncation and round-to-nearest. Truncation obtains a maximum error of 1 unit in the last place (1 ulp) and round-to-nearest obtains a maximum error of half a unit in the last place (0.5 ulp). Truncation splices the result up to the fractional bit count and omits any of the bit values that were to the right of the final fractional bit determined, which requires no additional hardware. Round-to-nearest executes the same as truncation, but adds the one bit value that is to the right of the last fractional bit to the result, which is done utilizing an adder. The values obtained by quantizing the result using truncation are essentially obtaining new magnitudes that are towards negative infinity. On the other hand, values obtained by quantizing the result round-to-nearest are essentially obtaining new magnitudes that are rounded towards positive infinity. Round-to-nearest is selected for the duration of this paper to minimize error while also not adding large complexity to the hardware design.
For the MiniBit approach, the quantized version of a signal \( x \) is shown as \( \tilde{x} \) and can be seen in 2.55, where \( E_{\tilde{x}} \) represents the corresponding error due to quantization and \( FB_{\tilde{x}} \) represents the number of fractional bits by which the signal \( x \) is quantized. Now, 2.56 shows the walk through of the approach involving the addition of two quantized numbers to create a quantized number. In addition to this, 2.57 then shows the approach for multiplication. For approximation utilizing polynomials, the process can be split into individual steps involving either multiplication or addition and these equations are the building blocks to the overall approximation error analysis. The polynomials for approximation are evaluated utilizing Horner’s rule, which facilitates the abstraction into a series of additions and multiplications for the overall approximation. The application of the MiniBit approach for degree one, two and three polynomials, respectively, can be found starting at 2.58, 2.62 and 2.66. The only addition is the \( E_{\text{approx}} \) term in each corresponding \( E_y \) which represents the error created by the Chebyshev series approximation before any quantization is executed.

\[
\begin{align*}
\tilde{x} &= x + 2^{-FB_{\tilde{x}}-1} \\
E_{\tilde{x}} &= 2^{-FB_{\tilde{x}}-1}
\end{align*}
\] (2.55)

\[
\begin{align*}
\tilde{z} &= \tilde{x} \pm \tilde{y} = x \pm y = E_{\tilde{x}} \pm E_{\tilde{y}} + 2^{-FB_{\tilde{z}}-1} \\
E_{\tilde{z}} &= E_{\tilde{x}} + E_{\tilde{y}} + 2^{-FB_{\tilde{z}}-1}
\end{align*}
\] (2.56)

\[
\begin{align*}
\tilde{z} &= \tilde{x} \cdot \tilde{y} = x \cdot y + x \cdot E_{\tilde{y}} + y \cdot E_{\tilde{x}} + E_{\tilde{x}} \cdot E_{\tilde{y}} + 2^{-FB_{\tilde{z}}-1} \\
E_{\tilde{z}} &= \max(x) \cdot E_{\tilde{y}} + \max(y) \cdot E_{\tilde{x}} + E_{\tilde{x}} \cdot E_{\tilde{y}} + 2^{-FB_{\tilde{z}}-1}
\end{align*}
\] (2.57)

Degree One Approximation MiniBit Approach:

\[
\tilde{y} = \tilde{C}_1 \cdot x_l + C_0
\] (2.58)
\[ \tilde{D}_0 = \tilde{C}_1 \cdot \tilde{x}_I \]
\[ \tilde{y} = \tilde{D}_0 + \tilde{C}_0 \]

\[ E_{C_1} = 2^{-FB_{C_1}} \quad E_{C_0} = 2^{-FB_{C_0}} \]  
\[ E_{D_0} = \max(\tilde{x}_I) \cdot E_{C_1} + \max(\tilde{C}_1) \cdot E_{x_I} + 2^{-FB_{D_0}} \]
\[ E_y = E_{D_0} + E_{C_0} + 2^{-FB_y} \]  

\[ \tilde{D}_2 = \tilde{C}_2 \cdot \tilde{x}_I \]  
\[ \tilde{D}_1 = \tilde{D}_2 + \tilde{C}_1 \]  
\[ \tilde{D}_0 = \tilde{D}_1 \cdot \tilde{x}_I \]  
\[ \tilde{y} = \tilde{D}_0 + \tilde{C}_0 \]

\[ E_{C_2} = 2^{-FB_{C_2}} \quad E_{C_1} = 2^{-FB_{C_1}} \quad E_{C_0} = 2^{-FB_{C_0}} \]

\[ E_{D_2} = \max(\tilde{x}_I) \cdot E_{C_2} + \max(\tilde{C}_2) \cdot E_{x_I} + 2^{-FB_{D_2}} \]  
\[ E_{D_1} = E_{D_2} + E_{C_1} + 2^{-FB_{D_1}} \]
\[ E_{D_0} = \max(\tilde{x}_I) \cdot E_{D_1} + \max(\tilde{D}_1) \cdot E_{x_I} + 2^{-FB_{D_0}} \]  
\[ \max(\tilde{D}_1) = \max(\tilde{C}_2) \cdot \max(\tilde{x}_I) + \max(\tilde{C}_1) \]
\[ E_y = E_{D_0} + E_{C_0} + 2^{-FB_y} + E_{approx} \]
Degree Three Approximation MiniBit Approach:

\[ y = ((C_3 \cdot x_l + C_2) \cdot x_l + C_1) \cdot x_l + C_0 \]  \hspace{1cm} (2.66)

\[ \tilde{D}_4 = \tilde{C}_3 \cdot \tilde{x}_l \]
\[ \tilde{D}_3 = \tilde{D}_4 + \tilde{C}_2 \]
\[ \tilde{D}_2 = \tilde{D}_3 \cdot \tilde{x}_l \]
\[ \tilde{D}_1 = \tilde{D}_2 + \tilde{C}_1 \]
\[ \tilde{D}_0 = \tilde{D}_1 \cdot \tilde{x}_l \]
\[ \tilde{y} = \tilde{D}_0 + \tilde{C}_0 \]

\[ E_{C_3} = 2^{-FBC_3} \quad E_{C_2} = 2^{-FBC_2} \quad E_{C_1} = 2^{-FBC_1} \quad E_{C_0} = 2^{-FBC_0} \]  \hspace{1cm} (2.68)

\[ E_{D_4} = \max(\tilde{x}_l) \cdot E_{C_3} + \max(\tilde{C}_3) \cdot E_{x_l} + 2^{-FBD_4-1} \]
\[ E_{D_3} = E_{D_4} + E_{C_2} + 2^{-FBD_3-1} \]
\[ E_{D_2} = \max(\tilde{x}_l) \cdot E_{D_3} + \max(\tilde{D}_3) \cdot E_{x_l} + 2^{-FBD_2-1} \]
\[ \max(\tilde{D}_3) = \max(\tilde{C}_3) \cdot \max(\tilde{x}_l) + \max(\tilde{C}_2) \]
\[ E_{D_1} = E_{D_2} + E_{C_1} + 2^{-FBD_1-1} \]
\[ E_{D_0} = \max(\tilde{x}_l) \cdot E_{D_1} + \max(\tilde{D}_1) \cdot E_{x_l} + 2^{-FBD_0-1} \]
\[ \max(\tilde{D}_1) = \max(\tilde{D}_3) \cdot \max(\tilde{x}_l) + \max(\tilde{C}_1) \]
\[ E_y = E_{D_0} + E_{C_0} + 2^{-FBy-1} + E_{approx} \]

Once the corresponding values that are unique to each analysis (e.g. \( max(C_i), \ E_{x_l}, \ E_{approx}, \ldots \)) are found, the problem becomes an optimization problem, where the fractional bit sizes (\( FB \)) are to be minimized while meeting the error requirement. This paper executes the error analysis with the goal of faithful rounding (1 ulp). Therefore, the error requirement for \( y \) becomes dependent on its fractional bit size as seen in 2.70,
2.2 Box-Muller Hardware Implementation

where the corresponding $E_y$ is found depending on its degree of approximation utilizing either 2.61, 2.65 or 2.69 for degree one, two or three respectively. For the results in this paper, the fmincon MATLAB function is utilized. Fmincon is used with summation of all the fractional bit counts associated in the specific analysis as the function to minimize [28]. In addition to this, it is used with the MiniBit approach equations as the non-linear constraint. It outputs non-integer numbers, therefore the output values are rounded up to the nearest integer.

\[ E_y \leq 2^{-FB_y} \quad (2.70) \]

2.2.2.2 General Analysis Walk Through

This section describes a specific approach for walking through the error analysis of a Box-Muller hardware implementation. Note that this walk through and process is bias due to being derived from the 16-bit analysis that Lee et al. executed and their corresponding architecture and is not sufficient for all Box-Muller hardware implementations [1]. Section 2.2.1.2 provides all the equations for obtaining coefficient values for Chebyshev series approximations. Section 2.2.1.3 provides the elementary function identities, the range reduction and range reconstruction steps, the general pseudo code for the Box-Muller method in Listing 2.1 and the equations to transform the coefficients to the corresponding $x_i$ value for the corresponding $x$ input interval unique to the function being approximated. Finally, Section 2.2.2.1 provides the quantization error constraint equations that are the main focus of the error analysis. The following walk through assumes the architecture is designed utilizing Tausworthe 32-bit random number generators as the inputs to the Box-Muller algorithm. The inputs to this process are a sample size of noise $S$ for periodicity, the output bit size of the noise and how many 32-bit Tausworthe generators are used. Throughout this section, line numbers referenced are
from the Box-Muller method pseudo code given in Listing 2.1.

1. The sample size $S$, determines the corresponding maximum values that the Gaussian noise generator should be designed to produce. This is represented in terms of standard deviations $\sigma$ away from the mean $\mu$. Gaussian noise is represented by a standard normal distribution, a distribution discussed in Section 2.1.2, where $\mu = 0$ and $\sigma = 1$ and is referenced throughout this paper as $N(0, 1)$. Therefore, $S$ leads to a value $X\sigma$ where $X$ represents the number of standard deviations away from the mean. The concept derived from [1] for connecting $S$ to $X\sigma$ is to represent up to $X\sigma$ for a population of $S$, where the probability of the magnitude of a particular sample from the Gaussian noise generator being larger than $X\sigma$ is less than 0.5. This step is solved utilizing the norminv MATLAB function, where for an example of $S = 10^{10}$, it was found that the Gaussian noise generator would need to represent up to $6.47\sigma$ [29].

2. Now that $X\sigma$ is known, the size of $u_0$, which is the input to the LN unit, is to be found. The SIN/COS unit will only produce magnitudes on [0,1), which requires the LN and SQRT units’ joined data path to produce a maximum value of $X\sigma$. In 2.71 the relationship is shown where for the example of $6.47\sigma$, $u_0$ must be less than $8.129 \cdot 10^{-9}$. Since $u_0$ is [0,1), the precision value of $u_0$ is chosen to be less than that value, which leads to requiring 32 bits for $u_0$.

$$X\sigma \geq \sqrt{-2 \cdot \ln(u_0)} \quad (2.71)$$

3. Now that the size of $u_0$ is known, the size of $u_1$ is to be found. For this specific architecture, the remaining bits generated from the Tausworthe generators that aren’t designated for $u_0$ are chosen to be the size of $u_1$. There is no other basis for this decision other than the size of $u_1$ being close to the output bit size with a preference for being larger to minimize the errors propagated through the SIN/COS unit.

4. The bit structure of the noise is determined from the size of $X\sigma$ and the overall
bit size of the output noise. The noise samples are positive and negative and range on \((-X\sigma, X\sigma)\). Therefore, one bit is designated to the sign bit. The count of integer bits \((IB)\) is determined by the \(X\sigma\) value and the rest are designated as fractional bits \((FB)\). For example if \(X = 6.47\) and the output bit size was 16, then 1 bit would correspond to the sign, 3 bits for the integer portion \((IB_{x_0} = IB_{x_1} = 3)\) and 12 bits for the fractional portion \((FB_{x_0} = FB_{x_1} = 12)\).

5. For this step, the type of rounding for the output is chosen. The two choices are either exact rounding (0.5 ulp) or faithful rounding (1 ulp). For this paper faithful rounding is decided upon due to the complexity of exact rounding as discussed in Section 2.2.1.1. Faithful rounding requires 1 ulp and with fractional bit size of output \(FB_{x_0}\), the output error is restricted to less than \(2^{-FB_{x_0}}\).

6. This step executes the error analysis at the output seen in 2.72 to find out the fractional bit sizes of \(f\), \(g_0\) and \(g_1\). The error analysis starts at the end and makes its way back to the beginning because the main goal is for the output’s error to meet the faithful rounding requirement. For the duration of this analysis, \(x_0\) is only considered, since the \(x_1\) data path is identical from an error analysis point of view. \(f\) and \(g_0\) are faithfully rounded, which leads to 2.74. Using 2.74 as the non-linear constraint for fmincon as explained in Section 2.2.2.1, the values for \(FB_f\) and \(FB_{g_0} = FB_{g_1}\) are found where \(FB_f\) is desired to be smaller than the others due to the longer computational chain.

\[
x_0 = f \cdot g_0 \quad \text{and} \quad x_1 = f \cdot g_1 \quad \text{(2.72)}
\]

\[
E_{x_0} \leq 2^{-FB_{x_0}} \quad \text{and} \quad E_{x_1} \leq 2^{-FB_{x_1}} \quad \text{(2.73)}
\]

\[
2^{-FB_{x_0}} \geq max(g_0) \cdot 2^{-FB_f} + max(f) \cdot 2^{-FB_{g_0}} \quad \text{(2.74)}
\]
2.2 Box-Muller Hardware Implementation

7. This step executes the error analysis at the SIN/COS unit output seen in 2.75 to find out the fractional bit sizes of internal signals involved with the specific degree of approximation. See lines 42-63 for the pseudo code involving each of the steps. For simplicity, degree one approximation equations will be utilized. As determined in Section 2.2.1.3, only $\cos(x)$ will be approximated, where $x$ is $[0, \pi/2)$ and there are two dedicated hardware sections for two instances of $\cos(x)$, where the second one implements $\sin(x)$. This is due to the periodicity and the altering of the input to take advantage of the relationship between sine and cosine.

\[
g_0 = \sin(2\pi u_1) \quad \text{and} \quad g_1 = \cos(2\pi u_1) \tag{2.75}\]

7a. The first step within step 7 is to analyze the range reduction and range reconstruction steps to find the approximation output’s fractional bit size, $FB_{y_{ga}} = FB_{y_{gb}}$. The input to this step is $FB_{g_0} = FB_{g_1}$, where the range reduction and reconstruction steps can be seen, respectively, on lines 45-48 and lines 58-53. For range reconstruction there is only the addition of a sign bit and the input $u_1$ is an exact number, therefore no error is propagated through either steps and then $FB_{y_{ga}} = FB_{y_{gb}} = FB_{g_0} = FB_{g_1}$.

7b. This step focuses on finding the internal fractional bit widths within the function approximation step now that $FB_{y_{ga}}$ and $FB_{y_{gb}}$ are known. In regards to the error analysis, $y_{ga}$ and $y_{gb}$ are equivalent, so only $y_{ga}$ will be shown and represented by $y_g$. To start, the degree of approximation and number of segments needs to be found and 2.28 is utilized with the specific input interval, function to approximate and the degree of approximation and $k$ values are cycled to output the lowest $k$ value for each degree of approximation that meets the requirement. Once this $k$ values is known for each degree, a trade-off is made between the degree of approximation (higher degrees mean added multipliers and adders) and the segment counts (larger look up table sizes for the coefficients). For this example, degree one has been chosen for $k_g = 5$ where there are
$2^5 = 32$ segments of approximation. Each segment interval is then approximated where the coefficients are extracted using the degree specific equations starting at 2.16. As derived in Section 2.2.1.3, the input is on [0,1) therefore the coefficients are transformed to using $x_i$ as the input by the equations starting at 2.33. There are then two methods for finding the error due to approximation $E_{approx}$. The first is finding the maximum error value of the Chebyshev error equation in 2.28 for the degree and number of segments for the specific function. This function is run on each segment input range and the largest error value provides the result for $E_{approx}$. The other method is to create the approximated polynomial and cycle through the entire output range at a high precision increment and record the largest error value between the approximation and the actual output of the function for the same input. The transformed coefficients are then analyzed to obtain the maximum values needed for the MiniBit approach in Section 2.2.2.1. The error analysis constraint equations for the MiniBit approach are dependent on the degree of approximation and can be found starting at 2.58, 2.62 and 2.66, respectively, for degrees one, two and three. For the COS/SIN unit $u_1$ is the input and doesn’t propagate in any errors through range reduction due to it being an exact number and therefore, the MiniBit approach equations are simplified where $\max(x_{lg}) = 1$ and $E_{x_{lg}} = 0$. Using the now known values along with the correct MiniBit constraint equations, the optimization results will lead to fractional bit width values for all coefficients $FB_{C_{tg}}$ and internal arithmetic signals $FB_{D_{tg}}$.

7c. Now that the coefficient values are found for the approximation and their corresponding fractional bit sizes are known, a quick analysis is executed for storing them in hardware as a look up table. First, the coefficient arrays are examined to see if there are any values of magnitude greater than or equal to 1, which would result in the need to store integer bits in the table as well. Second, for coefficient values that are significantly smaller than 1, there may be redundant bits that can be omitted when they are stored
in the look up tables and later concatenated to the correct size in hardware. Third, the sign of the coefficients are analyzed to see if the entire array is always negative, which results in the opportunity to omit the storing of the sign bit in the look up table due to its redundancy.

8. This step executes the error analysis at the SQRT unit output seen in 2.76 to find out the fractional bit sizes of internal signals involved with the specific degree of approximation, the range reduction step and the range reconstruction step. See lines 23-40 for the pseudo code involving each of the steps. For simplicity, degree one approximation equations will be utilized. As determined in Section 2.2.1.3, only $\sqrt{x}$ will be approximated, where $x$ is on two different intervals, $[1,2)$ and $[2,4)$ due to the identity getting split up into two different evaluations depending on whether $E_x$ is odd or even.

$$f = \sqrt{e} = \sqrt{-2\ln(u_0)}$$ (2.76)

8a. This step is dedicated to the error analysis involved for the SQRT unit except for the function approximation section. There is the error propagated through the LN unit into the input of the SQRT unit and manipulated in the range reduction section in lines 27 and 28 that adds the most complication. There is also, the shifting of the approximation output error when evaluating $f$ in the range reconstruction steps found in lines 39 and 40. To kick off this part, the range reconstruction steps in lines 39 and 40 are analyzed first to work backwards since $FB_y$ is known and to solve for the approximation output fractional bit width $FB_{yf}$. Shifting of the approximation output $y_f$ essentially shifts the error associated with it as well and is to be taken into consideration. A range analysis is executed on $e$, the output of the LN unit, to find the $IB$ size because it will effect the value of $exp_f$ in the range reduction stage. Once $IB_e$ is found, $Offset = IB_e - 2$ and the minimum and maximum values of $exp_f$ are in the form of 2.77, which stems off of the leading zero detector having a maximum value of
\(IBe + FBe - 1\).

\[
\min(\exp_f) = -FB_e - 1 \quad \text{and} \quad \max(\exp_f) = \text{Offset} = IB_e - 2 \tag{2.77}
\]

Now, the range reconstruction step on line 39 and in 2.78 is analyzed for the maximum values because that will be the worst case scenario in magnifying the error due to the function approximation step \(E_{yf}\). The maximum value from 2.77 is then plugged into 2.78 to find the \(\max(\exp'_f)\) found in 2.79. Stemming from line 40, the relationship between the function approximation error \(E_{yf}\) and \(E_f\) is shown in 2.80. For this analysis \(f\) is faithfully rounded to 1 ulp, which creates the error relationship for the function approximation found in 2.81.

\[
\exp'_f = \begin{cases} 
\exp_f/2, & \exp_f[0] = 0 \\
(\exp_f + 1)/2, & \exp_f[0] = 1
\end{cases} \tag{2.78}
\]

\[
\max(\exp'_f) = \begin{cases} 
(IB_e - 3)/2, & \exp_f[0] = 0 \\
(IB_e - 1)/2, & \exp_f[0] = 1
\end{cases} \tag{2.79}
\]

\[E_f = E_{yf} \cdot 2^{\exp'_f} \tag{2.80}\]

\[E_{yf} \leq 2^{-FB_f - \exp'_f} \tag{2.81}\]

Equation 2.81 is then analyzed taking into consideration the values in 2.79, which leads to the worst case scenario of \(y_f\) needing to be accurate to the values found in 2.82,
which would correspond to the $FB_{y_f}$ value that meets the worst case error.

$$max(E_{y_f}) = \begin{cases} 
2^{-FB_{y_f}-(IB_e-3)/2}, & exp_f[0] = 0 \\
2^{-FB_{y_f}-(IB_e-1)/2}, & exp_f[0] = 1 
\end{cases} \quad (2.82)$$

Now the range reduction steps in lines 27-28 are analyzed to quantify the propagation error from the LN unit. Now $e$ is designed to be rounded faithfully, which leads to $max(E_e) = 2^{-FB_e}$ and (2.83) and (2.84) show the error propagation up to $x_f$. However, $x_{lf}$ is the actual value utilized in the function approximation arithmetic and therefore the value of $E_{x_{lf}}$ is needed. For the arithmetic, $x_{lf}$ is shifted to have its magnitude range from $[0,1)$, which will amplify the error in $E_{x_f}$ depending on the size in bits of $x_{mf}$, which is equal to $k_f$ and leads to (2.85).

$$E'_{x_f} = E_e = 2^{-exp_f} \quad (2.83)$$

$$E_{x_f} = \begin{cases} 
2^{-FB_e-exp_f} & exp_f[0] = 0 \\
2^{-FB_e-exp_f-1} & exp_f[0] = 1 
\end{cases} \quad (2.84)$$

$$E_{x_{lf}} = \begin{cases} 
2^{-FB_e-exp_f+k_f} & exp_f[0] = 0 \\
2^{-FB_e-exp_f-1+k_f} & exp_f[0] = 1 
\end{cases} \quad (2.85)$$

8b. In an actual implementation of this analysis, $FB_e$ and $k_f$ are the only two unknowns of (2.85) at this time. Therefore to find $k_f$, the same process for finding a degree of approximation and number of segments that 7b utilized is executed. Now, it is assumed that from that process a degree n-1 polynomial is chosen and $2^{k_f}$ segments are needed for that approximation. The process is different than 7b due to there being two different approximations, one for $[1,2)$ and one for $[2,4)$, which for the purpose of this
general analysis is assumed to result in having the same degree \( n - 1 \) and \( 2^k f \) segments for both input ranges. For hardware purposes only one chain of approximation arithmetic is implemented and input coefficients get selected from the right range of values using a multiplexer.

8c. Now that \( k f \) has a value, \( FB_e \) is desired to be found to fully complete the values needed for the MiniBit approach. To find \( FB_e \), the MiniBit approach equations found in Section 2.2.2.1 are applied and dominating error is utilized. To walk through this see 2.86, where a degree one approximation is assumed. With this assumption, 2.61 is applied where \( E_{xl_f} \) is found in 2.85. For this constraint, \( max(x_{l_f}) = 1 \), the accuracy of \( y_f \) is found in 2.82, and the approximation error \( E_{approx_f} \) and maximum values of the coefficients \( C_{i_f} \) are found utilizing the same methods used in 7b. At this step, with the knowledge that \( min(exp_f) = -FB_e - 1 \), Lee et al. consider the product \( C_{1_f} \cdot E_{xl_f} \) within 2.86 to be the dominating error within the error constraint equation \([1]\). With this assumption, a straightforward analysis to find \( FB_e \) can be executed at the minimum value of \( exp_f \), which causes the most error in \( E_{xl_f} \). First, the dominating error constraint equations stemming from 2.78, 2.81, 2.85 and 2.86 are created in the form of 2.87 and 2.88, respectively, where \( exp_f \) is even and odd.

\[
E_{y_f} = C_{1_f} \cdot E_{xl_f} + 2^{-FBc_{1_f} - 1} + 2^{-FBc_{1_f} - 1} \cdot E_{xl_f} + 2^{-FBc_{0_f}} + 2^{-FB_{y_f}} + E_{approx_f} \tag{2.86}
\]

\[
2^{-FB_e - exp_f / 2} \geq 2^{-FB_e - exp_f + k_f} \cdot max(C_{1_f}), \quad exp_f[0] = 0 \tag{2.87}
\]

\[
2^{-FB_e - (exp_f + 1) / 2} \geq 2^{-FB_e - exp_f - 1 + k_f} \cdot max(C_{1_f}), \quad exp_f[0] = 1 \tag{2.88}
\]

The even case is considered first, where \( FB_e \) is then odd in value. With the value
of \( \max \left( C_{1j} \right) \) for the \([2,4)\) range, 2.87 is analyzed at the minimum value of \( exp_f \) as per 2.77 and the corresponding \( FBe \) value that is found is rounded up to the nearest odd integer value. The same value analysis is done for 2.88 and the corresponding \( FBe \) value that is found is rounded up to the nearest even integer value. The value that is less in magnitude is then chosen for \( FBe \).

8d. Now that \( FBe \) is known, 2.86 can be optimized where the constraint is evaluated at the tail ends of \( exp_f \) with the maximum and minimum values that are found in 2.77. This optimization problem is executed twice using fmincon at the two different \( exp_f \) values and the specific output set that requires the fractional bit widths to be the largest is chosen due to the need to meet the requirement for both scenarios. From this, the corresponding coefficient fractional bit widths \( FBC_{1j} \) and the fractional bit widths of the internal arithmetic signals \( FBD_{1j} \) are found. The optimization process using fmincon in MATLAB can be found at the end of Section 2.2.2.1.

8e. This section is identical to 7c for the SIN/COS unit, but executed on the two different sets of coefficients for the SQRT unit.

9. This step executes the error analysis at the LN unit output seen in 2.89 to find out the fractional bit sizes of internal signals involved with the specific degree of approximation and the fractional bit sizes of the range reconstruction steps. See lines 9-21 for the pseudo code involving each of the steps. For simplicity, degree one approximation equations will be utilized. As determined in Section 2.2.1.3, only \( ln(x) \) will be approximated where \( x \) is \([0,1)\).

\[
e = -2 \cdot ln(u_0)
\]  

(2.89)

9a. This step will focus on analyzing the range reconstruction found on lines 20-21 and in both 2.90 and 2.91 to obtain \( FBl_{ln2} \), \( FBe \) and \( FB_{y_e} \) after knowing the value of \( FBe \). The foundation principles of the MiniBit approach in Section 2.2.2.1 are now
applied to both 2.90 and 2.91 utilizing the constraint error equations found in 2.56 and 2.57 to create error constraint equations for \( e' \) in the form of 2.92 and \( e \) in the form of 2.93.

\[
e' = \exp_e \cdot \ln 2 \tag{2.90}
\]

\[
e = (e' - y_e) \ll 1; \tag{2.91}
\]

\[
E_{e'} = \max(\exp_e) \cdot 2^{-FB_{\ln 2} - 1} + \max(\ln 2) \cdot E_{\exp_e} + 2^{-FB_{e'} - 1} \tag{2.92}
\]

\[
E_e = 2 \left( E_{y_e} + E_{e'} \right) + 2^{-FB_{e} - 1} \tag{2.93}
\]

From an analysis of the range reduction step in line 9, \( \max(\exp_e) \) is found to equal \( TB_{u_0} \), the total bit size of \( u_0 \). With the maximum value of \( \exp_e \) and the fact that there is no error involved with \( \exp_e \), \( E_{\exp_e} = 0 \), \( E_{e'} \) is now of the form 2.94. Now, \( y_e \) the function approximation output is designed to be faithfully rounded and 2.94 is created by plugging 2.94 into 2.93. Now \( e \) is designed to be faithfully rounded, which leads to the error constraint in 2.96. From 2.96, the fmincon optimization MATLAB approach found at the end of Section 2.2.2.1 is then executed to obtain \( FB_{y_e}, FB_{e'} \) and \( FB_{\ln 2} \).

\[
E_{e'} = TB_{u_0} \cdot 2^{-FB_{\ln 2} - 1} + 2^{-FB_{e'} - 1} \tag{2.94}
\]

\[
E_e = 2^{-FB_{y_e} + 1} + TB_{u_0} \cdot 2^{-FB_{\ln 2}} + 2^{-FB_{e'}} + 2^{-FB_{e} - 1} \tag{2.95}
\]

\[
2^{-FB_{e} - 2} \geq 2^{-FB_{y_e}} + TB_{u_0} \cdot 2^{-FB_{\ln 2} - 1} + 2^{-FB_{e'} - 1} \tag{2.96}
\]
9b. Now that $FB_y$, is known, it is time to determine the degree of approximation $n - 1$, number of segments of approximation $k_e$ and obtain the transformed coefficient values $C_{i_e}$. This process is the same as the process done in 7b and 8b, but the function of approximation is $ln(x)$ and $x \in [1,2)$. Then the maximum values of $C_{i_e}$ are found and the approximation error $E_{approx_e}$ is found utilizing the same strategy found in 7b.

9c. Now that $FB_y$, $E_{approx_e}$ and $max(C_{i_e})$ are known, $E_{x_{l_e}}$ is the last unknown before executing the MiniBit approach. To find $E_{x_{l_e}}$, the range reduction steps in line 9-10 are analyzed and it is seen that any error involved with $u_0$ has the potential to be amplified by $exp_e$. However, $E_{u_0} = 0$ since it is an exact number, which leads to no error propagation through the input and therefore $E_{x_{l_e}} = 0$. With all these values and $max(x_{l_e}) = 1$, the MiniBit approach error constraint equations for a degree $n - 1$ polynomial in Section 2.2.2.1 are utilized along with the fmincon MATLAB function as discussed at the end of that section to solve the optimization problem for the fractional bit widths of the transformed coefficients $FB_{C_{i_e}}$ and the fractional bit widths of the internal arithmetic signals $FB_{D_{i_e}}$.

9d. Now to finish this analysis, the analysis of the coefficients that are stored in the look up tables is executed to find opportunities to optimize the hardware storage. This analysis is identical to 7c for the SIN/COS unit, but executed on the set of coefficients for the LN unit.

Now, the error analysis is complete and the design is ready to move into the hardware implementation phase.

2.2.3 16-bit Error Analysis Walk Through

This section is dedicated to walking through the entire error analysis that Lee et al. executed in [1], where the main novelty is walking through the analysis while applying the general analysis steps in Section 2.2.2.2 that were derived from [1]. All equations
and techniques used in this section are correctly referenced in Section 2.2.2.2. From a high-level, the error analysis is executed utilizing the MiniBit approach for bit width optimization and the function approximations are done utilizing Chebyshev series approximations. The resulting bit-widths for the 16-bit architecture that are solved for in this analysis can be seen visually in the block diagram of the design that Lee et al. created in [1] and their pseudo code for their design can be seen in Listing 2.2. For the duration of this section, the reference to numbered steps (eg. 7b) is referring to the steps in the general analysis within Section 2.2.2.2.

Now to kick off this analysis, the design was required to have 16-bit noise and a sample size $S$ of $10^{15}$ samples. Following step 1, the norminv MATLAB function is utilized with $S = 10^{15}$ and results in needing to represent $8.03\sigma$, which is then rounded to designing for $8.1\sigma$. Step 2 is then followed using 2.71 to find $u_0 \leq 5.66 \cdot 10^{-15}$. From this, the closest bit-wise precision is $2^{-48} = 3.55 \cdot 10^{-15}$, therefore $u_0$ is designed to have the total number of bits, $TB_{u_0}$, be 48. Designing to the lower precision also increases the maximum value to $8.2\sigma$. The design utilizes two 32-bit Tausworthe generators and for step 3, it leads to 16 bits being left over for $u_1$, which is also close to the size of the output noise and doesn’t cause any significant issue. For step 4, 1 bit is designated to the sign bit, 4 bits for the integer portion to represent up to $8.2\sigma$ ($IB_{x_0} = IB_{x_1} = 4$) and the remaining 11 bits are designated for the fractional portion ($FB_{x_0} = FB_{x_1} = 11$) of the fixed point representation of the two 16-bit output noise samples, $x_0$ and $x_1$. For step 5, faithful rounding (1 ulp) is chosen, which results in the output error constrained to less than $2^{-11}$. For step 6, $max(g_0)$ is found to equal 1 due to being the maximum value of sine and cosine ($max(g_0) = 1$). In addition to this, $max(f)$ is found by passing in the minimum value of $u_0$, $2^{-48}$ into $f = \sqrt{-2ln(u_0)}$ and leads to $max(f) = 8.157$. With these values, the output error constraint becomes of the form in 2.97, where fmincon
leads to $FB_f = 13$ and $FB_{g_0} = FB_{g_1} = 15$.

$$2^{-11} \geq 2^{-FB_f} + 8.157 \cdot 2^{-FB_{g_0}} \quad (2.97)$$

\[\text{#16-bit Box-Muller Method Pseudo Code}\]
2.2 Box-Muller Hardware Implementation

--------------- Generate u0 and u1 ---------------

a = taus(); b = taus();
u0 = concat(a,b[31:16]);
u1 = b[15:0];

------------- Evaluate e = -2ln(u0) --------------

# Range Reduction
exp_e = LeadingZeroDetector(u0) + 1;
x_e = u0 << exp_e;

# Approximate -ln(x_e) where x_e = [1,2)
# Degree-2 piecewise polynomial
y_e = ((C2_e[xm_e]*xl_e)+C1_e[xm_e])*xl_e +C0_e[xm_e];

# Range Reconstruction
ln2 = ln(2);
e' = exp_e*ln2;
e = (e'-y_e)<<1;

------------- Evaluate f = sqrt (e) ------------

# Range Reduction
exp_f = 5-Leading ZeroDetector(e);
x_f' = e >> exp_f;
x_f = if(exp_f[0], x_f'>>1, x_f');

# Approximate sqrt (x_f) where x_f = [1,4)
# Degree-1 piecewise polynomial
y_f = C1_f[xm_f]*xl_f+C0_f[xm_f];

# Range Reconstruction
exp_f' = if(exp_f[0], exp_f+1>>1, exp>>1);
f = y_f << exp_f';

------------ Evaluate g0=sin(2*pi*u1) ------------
------------ g1=cos(2*pi*u1) -------------
2.2 Box-Muller Hardware Implementation

```c
switch(quadrant)
    case 0: g0 = y_g_b; g1 = y_g_a; # [0, pi/2)
    case 1: g0 = y_g_a; g1 = -y_g_b; # [pi/2, pi)
    case 2: g0 = -y_g_b; g1 = -y_g_a; # [pi, 3*pi/2)
    case 3: g0 = -y_g_a; g1 = y_g_b; # [3*pi/2, 2*pi)

--------------- Compute x0 and x1 ---------------
x0 = f*g0; x1 = f*g1;
```

Listing 2.2: 16-bit Box-Muller Method Pseudo Code

Now the main error analysis begins for the SIN/COS unit. Step 7a leads to $FB_{y_{bs}} = FB_{y_{gs}} = FB_{y_{g0}} = FB_{y_{g1}} = 15$. In step 7b, for a degree one, two and three approximation, respectively, it is found that 128, 16 and 4 segments are required to meet the error constraint. Lee et al. share their rule of thumb to utilize a degree one approximation if the approximation error constraint is to a precision of less than 20 bits [1]. Therefore, a degree one approximation for $\cos(x \cdot \frac{\pi}{2})$ is then chosen with 128 segments ($n = 2$ and $k_g = 7$) with coefficient values $C_{1_g}$ and $C_{0_g}$ and internal arithmetic signal $D_{0_g}$. From the approximation, $E_{\text{approx}} = 9.41 \cdot 10^{-6}$ and $\text{max}(C_{1_g}) = 0.123$. With these values and knowing $E_{x_{1_g}} = 0$ due to no error propagation on the input, the next part of 7b executes the MiniBit approach for degree one and results in $FB_{C_{1_g}} = 18$, $FB_{C_{0_g}} = 18$, and $FB_{D_{0_g}} = 18$. For 7c, $\text{max}(C_{1_g}) = 0.123$, $\text{max}(C_{0_g}) = 1.000009$, where all $C_{1_g}$ values are negative and all $C_{0_g}$ values are positive. Therefore, for $C_{1_g}$ there are 6 redundant bits and the sign bit doesn’t need to be stored. On the other hand, $C_{0_g}$ has a value over 1 and therefore requires the use of 1 integer bit as well. This results in 12 bits stored for $C_{1_g}$ and 19 bits stored for $C_{0_g}$.

The next analysis is for the SQRT unit and starts with step 8a. A range analysis is executed for $e$ and results in a maximum value of $-2ln(2^{-48}) = 66.5$. This value requires $IB_e = 7$ and therefore $\text{Offset} = 5$. With the value of $\text{Offset}$, $\text{max}(\text{exp}_f) = 5$ and $\text{max}(\text{exp}_{f_1})$ becomes 2 and 3, respectively, for $\text{exp}_f$ being even and odd. $FB_{y_{f}}$ is then found to be 16 bits. Step 8b is now looked at where the degree of approximation and
number of segments are to be found for the approximation of $\sqrt{x}$ for both input ranges of [1,2) and [2,4). For a degree one, two and three approximation, respectively, it is found that 64, 8 and 2 segments are required for both input ranges to meet the error constraint. Therefore, a degree one approximation with 64 segments is chosen ($n = 2$ and $k_f = 6$) with coefficients $C_{1f_{\text{odd}}}$, $C_{0f_{\text{odd}}}$, $C_{1f_{\text{even}}}$ and $C_{0f_{\text{even}}}$ and internal arithmetic signal $D_{0f}$.

From the approximation, $E_{\text{approx}_{f_{\text{odd}}}} = 3.785147 \cdot 10^{-6}$, $E_{\text{approx}_{f_{\text{even}}}} = 5.353006 \cdot 10^{-6}$, $\max(C_{1f_{\text{odd}}}) = 0.00778$ and $\max(C_{1f_{\text{even}}}) = 0.011$. For step 8c, dominating error in the MiniBit approach is utilized to find a $FB_e$ value. The case where $\exp_f$ is even is considered first, which results in an odd $FB_e$ value. The result returns $FB_e \geq 25.98887$, which would lead to $FB_e = 27$ since that is the next odd value. The case where $\exp_f$ is odd and $FB_e$ is even leads to $FB_e \geq 23.98881$, therefore $FB_e = 24$. $FB_e$ is then decided to be 24 bits. For 8d, the two extreme cases of $\exp_f$ are used for the analysis at values of $\min(\exp_f) = -25$ and $\max(\exp_f) = 5$. The analysis is now done using the MiniBit approach equations for degree one and the error propagated through the input, $E_{x_{ij}}$, while taking into consideration the odd values of the constants used (eg. $E_{\text{approx}_{f}}$). This results in two scenarios, where the values for $FB_{C_{1f}}$, $FB_{C_{0f}}$ and $FB_{D_{0f}}$, respectively, are 16, 10 and 10 for the $\min(\exp_f)$ case and 18, 19 and 19 for the $\max(\exp_f)$ case. The latter places more stringent requirements and therefore $FB_{C_{1f}} = 18$, $FB_{C_{0f}} = 19$ and $FB_{D_{0f}} = 19$ values are chosen. For 8e, $C_{1f}$ has 6 redundant bits, $C_{0f}$ requires an integer bit and they both are composed of only positive values, which requires no sign bit to be stored. This leads to the total bits stored for $C_{1f}$ and $C_{0f}$, respectively, to be 12 and 20 for each index.

The next part of the analysis is for the LN unit and will begin with step 9a. From knowing $FB_e = 20$ and $TBu_0 = 48$, the following bit sizes are found, $FB_{ln2} = 32$, $FB_{e'} = 28$ and $FB_{y_e} = 27$. Now that $FB_{y_e}$ is known, 9b is executed where $\ln(x)$ is the function to be approximated on [1,2). For a degree one, two and three approxima-
tion, respectively, it is found that 4096, 256 and 32 segments are required to meet the error constraint. Since the precision is over 20 bits and the segment values are getting larger, a degree two approximation with 256 segments is chosen \((n = 3\) and \(k_e = 8\)) with coefficients \(C_{2e}, C_{1e}, C_{0e}\) and internal arithmetic signals \(D_{2e}, D_{1e}\) and \(D_{0e}\). From the approximation, \(E_{\text{approx}} = 6.1832 \cdot 10^{-10}\), \(\max(C_{2e}) = 7.59969 \cdot 10^{-6}\), \(\max(C_{1e}) = 0.003906\) and \(\max(C_{0e}) = 0.69119\). For 9c, the MiniBit approach equations for degree two are utilized and the values that result from optimization are \(FB_{C_{2e}} = FB_{C_{1e}} = FB_{C_{0e}} = FB_{D_{2e}} = FB_{D_{1e}} = FB_{D_{0e}} = 30\). For 9d, \(C_{2e}\) and \(C_{1e}\), respectively, have 17 and 8 redundant bits and neither of them require integer bits. In addition to this, all \(C_{2e}\) values are negative, therefore there is no need to store the sign bit.
Chapter 3

LGBMGNG 24-bit Noise Error Analysis

This chapter is dedicated to walking through the error analysis for a Gaussian noise generator that creates 24-bit noise samples. The error analysis stems off of the error analysis that Lee et al. went through for 16-bit noise samples as shown in Section 2.2.3. The error analysis applies the general analysis steps in Section 2.2.2.2 that were derived from [1]. All equations and techniques used in this section are correctly referenced in Section 2.2.2.2. From a high-level, the error analysis is executed utilizing the MiniBit approach for bit width optimization and the function approximations are done utilizing Chebyshev series approximations. The resulting bit-widths for the 24-bit architecture that are solved for in this analysis can be seen visually in the block diagram of the design found in Figure 4.1 and the resulting pseudo code for the design can be seen in Listing 3.1. For the duration of this chapter, the reference to numbered steps (eg. 7b) is referring to the steps in the general analysis within Section 2.2.2.2 and the 16-bit design is referring to the design in Section 2.2.3. Throughout this section, line numbers referenced are from the 24-bit Box-Muller method pseudo code given in Listing 3.1.

The LGBMGNG analysis is started by knowing that there is a requirement to have
24-bit noise and there is no specific sample size \( S \) for the noise required. The 16-bit design only provided 16 bits to the SIN/COS unit, which worked fine for 16-bit noise. However, now that 24-bit noise is desired, another Tausworthe generator is brought into the design providing another 32-bits to the inputs. The goal of adding another Tausworthe generator is to include more precision in the \( f \) data path to increase \( X\sigma \). In addition to this, the higher precision for the inputs directly effects the range of precision for the outputs, which increases the quality of the distribution of the output noise values.

A third is added instead of redesigning one of the first two to limit the complexity of the design. 16-bits are then added to \( u_0 \) to bring it to 64 bits, a popular precision, and the remaining 16-bits to \( u_1 \) making \( TB_{u_0} = 64 \) and \( TB_{u_1} = 32 \). Now that the size of \( u_1 \) is known, step 3 is completed and steps 2 and 1 are worked through backwards to see the sample size \( S \) that the 24-bit design with 3 Tausworthe generators supports. For step 2, 2.71 is utilized backwards where \( 2^{-TB_{u_0}} = 2^{-64} \) is passed in as \( u_0 \), which leads to \( X\sigma = 9.42\sigma \). Now that \( X = 9.42 \), step 1 is performed using norminv and \( S \) is found to be \( 2 \cdot 10^{20} \) for this design, which is \( 2 \cdot 10^{5} \) times larger than the sample size in the 16-bit version. Now step 4 is looked at where 1 bit is designated as the sign bit, 4 bits are designated for the integer portion to represent up to \( 9.42\sigma \) \( (IB_{n_1} = IB_{n_2} = 4) \) and the remaining 19 bits are designated for the fractional portion \( (FB_{n_1} = FB_{n_2} = 19) \). For step 5, faithful rounding (1 ulp) is chosen, which results in the output error constrained to be less than \( 2^{-19} \). For step 6, \( \text{max}(g_0) = \text{max}(g_1) = 1 \) due to 1 being the maximum value for sine and cosine. In addition to this, \( \text{max}(f) \) is found by passing in the minimum value of \( u_0 \), \( 2^{-64} \) into \( f = \sqrt{-2 \cdot ln(u_0)} \), which leads to \( \text{max}(f) = 9.41928 \). With these values, the output error constraint becomes of the form in 3.1 where fmincon leads to \( FB_f = 20 \) and \( FB_{g_0} = FB_{g_1} = 24 \).

\[
2^{-19} \geq 2^{-FB_f} + 9.41928 \cdot 2^{-FB_{g_0}} \tag{3.1}
\]
%LGBMGNG 24-bit Box-Muller Pseudo Code

%----------------- Generate u0 and u1 -----------------
1 a = taus(); b = taus(); c = taus();% all 32-bit-------
2 u0 = concat(a,b);
3 u1 = c;

%----------------- Evaluate e = -2ln(u0) -----------------
4 % Range Reduction
5 exp_e = LeadingZeroDetector(u0) + 1;
6 x_e = u0 << exp_e;
7 % Degree-3 piecewise polynomial - 512 segments
8 C3,C2,C1,C0 = 19,28,38,47 bits.
9 % x_e is [1,2) --> x_e = 1 + xm_e + xl_e * 2^-k_e
10 y_e = (((C3_e(xm_e_index)*xl_e+C2_e(xm_e_index))*xl_e)+C1_e(xm_e_index))*xl_e +
11       C0_e(xm_e_index);
12 % Range Reconstruction
13 ln2 = ln(2);
14 e' = exp_e*ln2;
15 e = (e'-y_e)<<1;
16
%----------------- Evaluate f = sqrt (e) -----------------
17 % Range Reduction
18 exp_f = 5-LeadingZeroDetector(e);
19 x_f' = e >> exp_f;
20 x_f = if(exp_f[0], x_f'>>1, x_f');
21 % Approximate sqrt (x_f) where x_f = [1,4]
22 % x_f is [1,2) --> x_f = 1 + xm_f + xl_f * 2^-k_f
23 % or
24 % x_f is [2,4) --> x_f = 2 + 2*xm_f + xl_f * 2^(-k_f+1)
25 % Degree-1 piecewise polynomial - 1024 segments
26 C1,C0 = 16,26 bits.
27 y_f = C1_f(xm_f_index)*xl_f + C0_f(xm_f_index);
28 % Range Reconstruction
29 exp_f' = if(exp_f[0], exp_f+1>>1, exp>>1);
30 f = y_f << exp_f';
31 %----------------- Evaluate g0=sin(2*pi*u1) -------------
32 %----------------- g1=cos(2*pi*u1) -------------
33
%----------------- Evaluate g0=sin(2*pi*u1) -------------
34 quadrants = u1[31:30];
35 x_g_a = u1[29:0];
36 x_g_b = (1-2^-30)-u1[29:0];
37
% Approximate cos(x_g_a*pi/2) and cos(x_g_b*pi/2)
% where \( x_{g\ a}, x_{g\ b} = [0,1-2^{-30}] \)
% Degree-2 piecewise polynomial - 128 segments
% \( C2,C1,C0 = 14,21,28 \) bits.
% \( x_g \) is \([0,1) \rightarrow x_g = x_m_g + x_l_g * 2^{-k_g} \)
y_{g\ a} = C2_g(x_m_g_a_index)*x_l_g_a^2 + C1_g(x_m_g_a_index)*x_l_g_a + C0_g(x_m_g_a_index);
y_{g\ b} = C2_g(x_m_g_b_index)*x_l_g_b^2 + C1_g(x_m_g_b_index)*x_l_g_b + C0_g(x_m_g_b_index);

% Range Reconstruction
switch(quadrant)
case 0: g0 = y_{g\ b}; g1 = y_{g\ a}; % \([0, \pi/2)\)
case 1: g0 = y_{g\ a}; g1 = -y_{g\ b}; % \([\pi/2, \pi)\)
case 2: g0 = -y_{g\ b}; g1 = -y_{g\ a}; % \([\pi, 3\pi/2)\)
case 3: g0 = -y_{g\ a}; g1 = y_{g\ b}; % \([3\pi/2, 2\pi)\)

%%--------------- Compute n1 and n2 ---------------
n1 = f*g0; n2 = f*g1;

Listing 3.1: LGBMGNG, 24-bit Box-Muller Method Pseudo Code

The main error analysis for the SIN/COS unit is then executed starting with 7a. Due to no error propagation in the range reconstruction steps in lines 60-64, \( FB_{y_{gs}} = FB_{y_{gs}} = FB_{y_{g0}} = FB_{y_{g1}} = 24 \). Step 7b is then executed where for a degree one, two and three approximation, respectively, 4096, 128 and 16 segments are required to meet the error constraint. A degree two approximation for \( \cos(x \cdot \frac{\pi}{2}) \) is then chosen with 128 segments \((n = 3 \text{ and } k_g = 7)\) with coefficient values \( C_{2_g}, C_{1_g} \text{ and } C_{0_g} \) and internal arithmetic signals \( D_{2_g}, D_{1_g} \text{ and } D_{0_g} \). From the approximation, \( E_{approx} = 9.6254 \cdot 10^{-9}, \max(C_{2_g}) = 7.52975 \cdot 2^{-5}, \max(C_{1_g}) = 0.01227 \text{ and } \max(C_{0_g}) = 1.00000000004 \). Then, the MiniBit approach referenced in 7b is executed with the degree two equations and results in \( FB_{C_{2_g}} = 27, FB_{C_{1_g}} = 27, FB_{C_{0_g}} = 27, FB_{D_{2_g}} = 27, FB_{D_{1_g}} = 28 \text{ and } FB_{D_{0_g}} = 28 \). For 7c, \( C_{2_g} \text{ and } C_{1_g}, \) respectively, have 13 and 6 redundant bits and both have all negative values except for the first index value of \( C_{1_g} \). Both are designed to not store the sign bit. Now, \( C_{0_g} \) has a maximum value above 1 and therefore requires an integer bit. This results in \( C_{2_g}, C_{1_g} \text{ and } C_{0_g}, \) respectively, requiring to store 14, 21 and 28 bits. Therefore, \( 128 \cdot (14 + 21 + 28) = 8064 \) bits are stored for each implementation of \( \cos(x \cdot \frac{\pi}{2}) \).
Next, SQRT is analyzed starting with step 8a. The range of $e$ is analyzed for a maximum value, where $\max(e) = -2 \cdot \ln(2^{-64}) = 88.72$. This value requires $IB_e = 7$ and brings about $Offset = 5$. Therefore, $\max(exp_f) = 5$ and $\max(exp'_f)$ becomes 2 and 3, respectively, for $exp_f$ being even and odd. $FB_{y_f}$ is then found to be 23 bits. For step 8b, the degree of approximation and the number of segments are to be found for the approximation of $\sqrt{x}$ for both input ranges of $[1,2)$ and $[2,4)$. For $[1,2)$, a degree one, two and three approximation, respectively, requires 512, 32 and 8 segments to meet the error constraint. On the other hand for $[2,4)$, a degree one, two and three approximation, respectively, requires 1024, 128, 16 segments to meet the error constraint. Typically for 23 bits of precision, a second degree approximation would be selected. However, due to the desire to keep $[1,2)$ and $[2,4)$ at the same level of approximation and degree two of $[2,4)$ requiring 128 segments, the trade-off is decided to not be worth it. Therefore, a degree one approximation of $\sqrt{x}$ is then chosen with 1024 segments ($n = 2$ and $k_f = 10$) with coefficients $C_{1,odd}, C_{0,odd}, C_{1,even}$ and $C_{0,even}$ and internal arithmetic signal $D_0_f$. From the approximation, $E_{approx,odd} = 2.1063 \cdot 10^{-8}$, $E_{approx,even} = 1.4894 \cdot 10^{-8}$, $\max(C_{1,odd}) = 6.90365 \cdot 10^{-4}$, $\max(C_{1,even}) = 4.8816 \cdot 10^{-4}$, $\max(C_{0,odd}) = 1.413868$ and $\max(C_{0,even}) = 1.99951$.

For step 8c, the dominating error within the MiniBit approach error constraint equations is used to find $FB_e$. The case where $exp_f$ is even is considered first, which results in an odd $FB_e$ value. The result returns $FB_e \geq 38.9993$, which would lead to $FB_e = 39$ since that is the next odd value. The case where $exp_f$ is odd and $FB_e$ is even leads to $FB_e \geq 38.9993$, therefore $FB_e = 40$. $FB_e$ is then decided to be 40 bits due to being pessimistic about how close 38.993 is to 39. For 8d, the two extreme cases of $exp_f$ are used for the analysis at values of $\min(exp_f) = -41$ and $\max(exp_f) = 5$. The analysis is now executed using the MiniBit approach equations for degree one in the same manner as 8d in the 16-bit section. This results in two scenarios, where the values for $FB_{c_{1,f}}$,
\(FB_{C_0f}\) and \(FB_{D_0f}\), respectively, are 14, 3 and 3 for the \(\text{min}(exp_f)\) case and 26, 25 and 25 for the \(\text{max}(exp_f)\) case. The latter places more stringent requirements and therefore \(FB_{C_1f} = 26, FB_{C_0f} = 25\) and \(FB_{D_0f} = 25\) are chosen. For 8e, \(C_1f\) has 10 redundant bits, \(C_0f\) requires an integer bit and they both are composed of only positive values, which requires no sign bit to be stored. This results in \(C_1f\) and \(C_0f\), respectively, requiring to store 16 and 26 bits and there are two look up tables for each for the two input ranges, [1,2) and [2,4). Therefore, \(1024 \cdot 2 \cdot (16 + 26) = 86016\) bits are stored for the implementation of \(\sqrt{x}\).

The final section of the analysis is for the LN unit, which begins with step 9a. From knowing \(FB_e = 40\) and \(TBu_0 = 64\), the following bit sizes are found, \(FB_{ln2} = 49\), \(FB_{e'} = 43\) and \(FB_{ye} = 43\). Now that \(FB_{ye}\) is known, 9b is executed where \(\ln(x)\) is the function to be approximated on [1,2). For a degree three and four approximation, respectively, it is found that 512 and 128 segments are required to meet the error constraint. Due to degree two being out of scope and degree three already adding plenty of hardware, a degree three approximation with 512 segments is chosen \((n = 4\) and \(k_e = 9\)\) with coefficients \(C_3_e, C_2_e, C_1_e, C_0_e\) and internal arithmetic signals \(D_4_e, D_3_e, D_2_e, D_1_e\) and \(D_0_e\). From the approximation, \(E_{\text{approx}} = 2.8422 \cdot 10^{-14}\), \(\max(C_3_e) = 2.47627 \cdot 10^{-9}\), \(\max(C_2_e) = 1.9073441 \cdot 10^{-6}\), \(\max(C_1_e) = 0.001953125\) and \(\max(C_0_e) = 0.69217\). For 9c, the MiniBit approach equations for degree three are utilized and the values that result from optimization are \(FB_{C_3e} = FB_{C_2e} = FB_{C_1e} = FB_{C_0e} = FB_{D_4e} = FB_{D_3e} = 47\) and \(FB_{D_2e} = FB_{D_1e} = FB_{D_0e} = 48\). For 9d, \(C_3_e, C_2_e\) and \(C_1_e\), respectively, have 28, 19 and 9 redundant bits. For all four, no integer bits are required and \(C_2_e\) has values that are all negative, therefore there is no need to store the sign bit. This results in \(C_3_e, C_2_e, C_1_e, C_0_e\), respectively, requiring to store 19, 28, 38 and 47 bits. Therefore, \(512 \cdot (19 + 28 + 38 + 47) = 67584\) bits are stored for the implementation of \(\ln(x)\).
Chapter 4

Hardware Implementation

This chapter describes the hardware implementation of LGBMGNG, a 24-bit Gaussian noise generator utilizing the Box-Muller method. Chapter 3 walks through the entire error analysis for LGBMGNG and all of the fractional bit widths of the main internal signals are derived there.

4.1 Architecture

The high level architecture of the LGBMGNG is seen in Figure 4.1. Three 32-bit Tausworthe uniform random number generators are implemented, where each has a unique seed passed into LGBMGNG. Tausworthe generators are chosen due to the algorithm only utilizing bit-wise and, xor and shift operations, which require small area to implement. There are two leading zero detectors, a 64-bit one in the LN unit and a 47-bit one in the SQRT unit. The detectors were designed utilizing Verilog’s casez statement, which is implemented very well by synthesis tools and requires minimal area. The approximations of the elementary functions requires coefficient values to be used in an efficient manner. Therefore, look up tables were implemented using Verilog’s case statement, which is also implemented very well by synthesis tools. In addition to this, the
error analysis in Section 3 took into account redundancy for the look up tables. This reduced the size of the operands stored in the look up tables, which saved area.

![Figure 4.1: LGBMGNG 24-bit Block Diagram](image-url)
4.2 Design Choices

To minimize the size of the internal signals of LGBMGNG, a range analysis was executed in MATLAB for the maximum values passed through the data paths involved in the design. The maximum values lead to determining the number of integer bits required for the signals. The results from this analysis along with the values for the size of the fractional portions derived in the error analysis can be found in Table 4.2.

The SQRT unit required degree one approximation, which is implemented using a simple multiply add circuit as seen in Figure 4.2 where the coefficient values from the look up tables are determined by the segment of approximation the input is in. Similarly, the SIN/COS unit is approximated by the circuit in Figure 4.3 where there are now two multipliers, two adders and another look up table due to requiring a degree two approximation. Also, the LN unit follows this form as seen in Figure 4.4, but is a degree three approximation and therefore requires three multipliers, three adders and four look up tables.

![Figure 4.2: Degree One Approximation of SQRT Hardware Implementation](image-url)
The implementation of LGBMGNG leads to a pipeline that is of the form found in Figure 4.5. LGBMGNG has a latency of 14 clock cycles before the first output noise.
samples are valid. This pipeline implementation was decided upon to keep the design simple and leave optimization to future work. The design designates a clock cycle to each major operation within the pseudo code. The range reduction steps are consolidated into one clock cycle for each unit and the range reconstruction steps are consolidated into one clock cycle for each unit as well for the simplicity of this design. For each approximation, the error analysis split the polynomial into a series of multiply and add operations. Again for simplicity, each multiply or add is designed to be single cycle. After the 14 clock cycles, the design will produce two samples of noise per clock signal due to the implementation of a pipeline.
Figure 4.5: LGBMGNG Pipeline
### Signals

<table>
<thead>
<tr>
<th>Signals</th>
<th>Total bits (TB)</th>
<th>Integer bits (IB)</th>
<th>Fractional bits (FB)</th>
<th>Sign bit (SB)</th>
<th>Redundant bits (RB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a,b,c</td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u₀</td>
<td>64</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u₁</td>
<td>32</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>expₑ</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xₑ</td>
<td>64</td>
<td>1</td>
<td>63</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cₑᵢ</td>
<td>47</td>
<td>0</td>
<td>47</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>Cₑ₁</td>
<td>48</td>
<td>0</td>
<td>47</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>Cₑ₀</td>
<td>48</td>
<td>0</td>
<td>47</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Dₑ₁</td>
<td>47</td>
<td>0</td>
<td>47</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dₑ₀</td>
<td>48</td>
<td>0</td>
<td>47</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>yₑ</td>
<td>43</td>
<td>0</td>
<td>43</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ln2</td>
<td>49</td>
<td>0</td>
<td>49</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e⁺</td>
<td>49</td>
<td>6</td>
<td>43</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>expᶠ</td>
<td>7</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>xᶠ</td>
<td>47</td>
<td>2</td>
<td>46</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xᶠ</td>
<td>48</td>
<td>2</td>
<td>46</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cₑᶠ₁</td>
<td>26</td>
<td>0</td>
<td>26</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Cₑᶠ₀</td>
<td>26</td>
<td>1</td>
<td>25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dₑᶠ₁</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>yᶠ</td>
<td>24</td>
<td>1</td>
<td>23</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>expᶠ</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>24</td>
<td>4</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Q</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xₐ, xₐᵇ</td>
<td>30</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cₑ₂ᵢ</td>
<td>28</td>
<td>0</td>
<td>27</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>Cₑ₁ᵢ</td>
<td>28</td>
<td>0</td>
<td>27</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Cₑ₀ᵢ</td>
<td>29</td>
<td>1</td>
<td>27</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Dₑ₂ᵢ</td>
<td>28</td>
<td>0</td>
<td>27</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Dₑ₁ᵢ</td>
<td>29</td>
<td>0</td>
<td>28</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Dₑ₀ᵢ</td>
<td>29</td>
<td>0</td>
<td>28</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>yₐ, yₐᵇ</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>g₀, g₁</td>
<td>25</td>
<td>0</td>
<td>24</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>n₁, n₂</td>
<td>24</td>
<td>4</td>
<td>19</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.2: 24-bit Operand Sizes
Chapter 5

Tests and Results

This chapter goes into detail about the tests and results for the hardware implementation of LGBMGNG, the use of the bit-exact model in MATLAB to provide test vector verification, the quality of the test vector sample sets and it walks through some bugs encountered in the design phase.

5.1 Simulation Results

A bit-exact model of the LGBMGNG was created in MATLAB utilizing the bit-widths in Table 4.2. The model was tested against the Box-Muller method using elementary functions in MATLAB to ensure the model met the error requirement of faithful round, $E \leq 2^{-19}$. This model was used to create test vectors for the verification of LGBMGNG. Four sets of test vectors were created. For each vector set, unique seeds were passed to the Tausworthe generators and they can be found in Table 5.1. Two of the sets were for 1 million samples for $n_1$ and $n_2$ and the other two were for 20 million samples. Seed 1 was run in the model and results in $n_1 = 3.5299$ and $n_2 = 0.4469$. Those values in hex for 24-bit signed outputs, leads to $n_1 = 0xC3D37$ and $n_2 = 0x039349$. Seed 1 was then simulated using the gate-level netlist, where the data path through the
5.2 RTL Synthesis Results

<table>
<thead>
<tr>
<th>Seed</th>
<th>Seed_a</th>
<th>Seed_b</th>
<th>Seed_c</th>
<th>Sample Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seed 1</td>
<td>7654321</td>
<td>87654321</td>
<td>987654321</td>
<td>1 Million</td>
</tr>
<tr>
<td>Seed 2</td>
<td>741732741</td>
<td>900903737</td>
<td>655563292</td>
<td>20 Million</td>
</tr>
<tr>
<td>Seed 3</td>
<td>1107711288</td>
<td>490903959</td>
<td>338177554</td>
<td>20 Million</td>
</tr>
<tr>
<td>Seed 4</td>
<td>886334683</td>
<td>2126303890</td>
<td>161170592</td>
<td>1 Million</td>
</tr>
</tbody>
</table>

Table 5.1: The Seed Values for the Four Test Vector Sets

LN and SQRT units is shown in Figure 5.1 and the data path through the SIN/COS unit is shown in Figure 5.2. These waveforms follow the pipeline infrastructure seen in 4.5, where the first sample set of $n_1$ and $n_2$, respectively, is seen to have the values 0x1C3D37 and 0x039349 as the model provided for verification. Seed 1 for 1 million samples was verified by the test vectors for the gate-level simulation and the console output can be seen in Figure 5.3. In addition to this, Seed 2 was run for 20 million samples and was verified by the test vectors in RTL simulation and the console output can be seen in Figure 5.4.

5.2 RTL Synthesis Results

The LGBMGNG hardware design targeted the TMSC 65nm ASIC process where it was synthesized using Synopsys Design Compiler at a clock frequency of $400MHz$. Two netlists are created: the first is the pre-scan netlist, which has the entire design, but there is no hardware designated for design for test (DFT). Full-scan test insertion was utilized for DFT and resulted in the post-scan netlist. Both netlists were compared in Table 5.2, where the scan insertion increased the number of gates used by 3,000. The overall design is roughly 116,000 gates and requires 166,866.48 $\mu m^2$ of area. Both netlists were then compared in Table 5.3 in regards to power usage where the dynamic power drastically increases.

The cell area for the post-scan netlist was then broken down to gain a better understanding of the Box-Muller hardware implementation and can be seen in Table 5.4.
### Figure 5.1: LN and SQRT Data Path Simulation Waveform
Figure 5.2: SIN/COS Data Path Simulation Waveform
5.2 RTL Synthesis Results

![Image 1]{image1}

**Figure 5.3: 1 Million Test Vectors Synthesis Simulation Waveform**

![Image 2]{image2}

**Figure 5.4: 20 Million Test Vectors RTL Simulation Waveform**

<table>
<thead>
<tr>
<th>Area</th>
<th>Pre-Scan Netlist</th>
<th>Post-Scan Netlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinational Area ($\mu$m²)</td>
<td>146373.12</td>
<td>145705.68</td>
</tr>
<tr>
<td>Buffer and Inverter Area ($\mu$m²)</td>
<td>6338.16</td>
<td>5604.12</td>
</tr>
<tr>
<td>Non-Combinational Area ($\mu$m²)</td>
<td>16164.72</td>
<td>21160.80</td>
</tr>
<tr>
<td>Total Cell Area ($\mu$m²)</td>
<td>162537.84</td>
<td>166866.48</td>
</tr>
<tr>
<td>Total Number of Gates</td>
<td>112874</td>
<td>115880</td>
</tr>
</tbody>
</table>

Table 5.2: Logic Synthesis Reports for Pre and Post Scan Chain Netlists
<table>
<thead>
<tr>
<th></th>
<th>Pre-Scan Netlist</th>
<th>Post-Scan Netlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Internal Power (mW)</td>
<td>20.6966</td>
<td>37.1</td>
</tr>
<tr>
<td>Net Switching Power (mW)</td>
<td>5.0397</td>
<td>11.5</td>
</tr>
<tr>
<td>Total Dynamic Power (mW)</td>
<td>25.7363</td>
<td>38.6</td>
</tr>
<tr>
<td>Cell Leakage Power (µW)</td>
<td>9.7345</td>
<td>10.05</td>
</tr>
</tbody>
</table>

Table 5.3: Power Usage for Pre and Post Scan Chain Netlists

In Table 5.4, MULT refers to the multipliers, ADD refers to the adders, LUTs refers to the look up tables in general, TAUS refers to the Tausworthe URNGs, LZD64 refers to the 64-bit leading zero detector implemented in the LN unit, LZD47 refers to the 47-bit leading zero detector implemented in the SQRT unit, LUT_COS refers to all the LUTs involving the cosine approximation, LUT_LN refers to all the LUTs involving the natural log approximation and LUT_SQRT refers to the LUTs involving the approximation of square root. From the table, multipliers account for half of the entire design’s area, while the look up tables only account for a quarter of the design’s area. A lower degree of approximation removes a multiplier and an adder from the circuit and removes two stages of the pipeline, while increasing the look up table size. Therefore, it can be seen that for this design, a lower degree of approximation for one of the functions would have been feasible. The largest multiplier is 15.6% of the design and is located within the LN unit where the approximation arithmetic is extremely large due to the bit-size of $u_0$ and the internal signal bit-widths to meet the more stringent error constraints. Compared to the 16-bit design, another TAUS unit was utilized and it is seen that only 1% of area was added to the design, but the $u_0$ size has added large area due to the multiplier sizes. The leading zero detectors were implemented well and use minimal area. The SIN/COS unit used significantly less area for the LUTs due to smaller error precision needed through the data path and no error propagation.
### 5.2 RTL Synthesis Results

<table>
<thead>
<tr>
<th></th>
<th>MULT</th>
<th>ADD</th>
<th>LUTs</th>
<th>TAUS</th>
<th>LZD64</th>
<th>LZD47</th>
<th>LUT_COS</th>
<th>LUT_LN</th>
<th>LUT_SQRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Cell Area ($\mu m^2$)</td>
<td>86902.92</td>
<td>6853.32</td>
<td>45104.76</td>
<td>4904.64</td>
<td>361.8</td>
<td>239.76</td>
<td>5695.2</td>
<td>17820.36</td>
<td>21587</td>
</tr>
<tr>
<td>Percent Total (%)</td>
<td>52</td>
<td>4.1</td>
<td>27</td>
<td>3</td>
<td>0.2</td>
<td>0.1</td>
<td>3.4</td>
<td>10.7</td>
<td>12.9</td>
</tr>
<tr>
<td>Count in Design</td>
<td>11</td>
<td>22</td>
<td>14</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Largest Cell Area ($\mu m^2$)</td>
<td>26053.2</td>
<td>628.92</td>
<td>6499.8</td>
<td>1635.12</td>
<td>N/A</td>
<td>N/A</td>
<td>1136.88</td>
<td>5707.08</td>
<td>6499.8</td>
</tr>
<tr>
<td>Largest Percent Total (%)</td>
<td>15.6</td>
<td>0.4</td>
<td>3.9</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>0.7</td>
<td>3.4</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Table 5.4: Post-Scan Chain Cell Area Break-down
5.3 Debugging LGBMGNG

This section walks briefly through the bugs found in the design of LGBMGNG.

5.3.1 2s Complement Multiplication

For the first set of 1 million test vectors, roughly 15,000 failed. Using the model to view the correct results for the internal signals, it was seen that the 2s complement multiplication result wasn’t matching up. Upon further analysis, it was learned that it was a sign extension problem. For multiplication, an $X$ bit number multiplied by a $Y$ bit number results in a $X + Y$ bit number. To fix this error, the sign extension for the $X$ bit operand and the $Y$ bit operand needed to be to $X + Y$ bits before multiplying them.

5.3.2 Overflow

In the SIN/COS unit, $y_{ga}$ and $y_{gb}$ are calculated. During an analysis of multiple test vectors failing against the model, it was seen that either $g_0$ or $g_1$ was becoming either 0 when it should have been a normal number or all ones when it should have been 0. First, the analysis led to the realization that in the cases where the result was at its maximum value to store dependent on its bit size, there was the chance that the addition of round-to-nearest would overflow the result and it would look like 0. Then, the analysis also led to the cases where the approximation and rounded coefficients had the opportunity to reach a negative result. This case is extremely rare and only possible due to the approximation values since $\cos(x \cdot \frac{\pi}{2})$ on $[0, \pi/2)$ can never achieve a negative number. An intermediate wire for the addition before removing the integer bit and sign bit was implemented along with an intermediate adder to execute the round-to-nearest addition. For the final result, if the the integer bit was set in the intermediate addition
wire, a decision was made depending on the sign bit. If the sign bit was set, that means the addition brought about a negative result and therefore, the output was set to 0, the lowest value it is able to represent by design. However, if the sign bit was not set, then the addition overflowed and the output was set to all ones, the highest value it is able to represent by design. On the other hand, if the integer bit was not set, the result from the round to nearest wire was analyzed. If the integer bit of that addition was set, another overflow occurred. Therefore, the output result is grabbed from the original addition without round-to-nearest to avoid rounding out of the range that can be represented. And last but not least, if the integer bit of the round-to-nearest wire is not set, then there was no overflow and the output of the round-to-nearest wire is used as the actual output value.

5.3.3 Redundancy Assumptions

For this bug, the sign extended $C_{1g}$ was outputting a single value that was negative, when the model was showing it to be 0. In the design at this point, $C_{1g}$ was assumed to be all negative values and therefore, a negative sign bit was automatically concatenated to the value outputted from the LUT. Therefore, a wrong assumption was made and that case had to be updated.

5.3.4 Rounding Schemes

For this bug, it was more of a model bug than hardware. 7 out of 20 million test vectors were failing and every single internal variable for all 7 seven test vectors were the exact value as the model except one of the output noise samples would be off by 1 ulp. The count of failures was extremely low and there wasn’t anything else to go off of since everything was equal. After some time exploring values in both the Model and the Verilog, the value of the output noise before rounding to nearest was exactly the
halfway point between the precision to round to. For example if rounding to an integer, the value was $-11.5$. Then after further analysis and research, it was learned that round-to-nearest (implemented in hardware), is round to positive infinity in the halfway case. In addition to this, it was noticed that in the model, the halfway case was round away from zero. Therefore, for the example of $-11.5$, the model would have resulted in $-12$ and the hardware would have resulted in $-11$ resulting in 1 ulp error. The halfway case decision is not important for an error analysis because both types of rounding at the halfway case would lead to 0.5 ulp. Therefore, since round towards positive infinity is the default for round-to-nearest in hardware, the design was kept simple and the model was updated to round towards positive infinity as well.

### 5.4 Test Vector Probability and Statistic Analysis

The test vector sets given by the seed values and sample count in Table 5.1, were analyzed utilizing basic probability and statistics and goodness-of-fit metrics discussed in Section 2.1.3.2. The results of the analyses done in MATLAB are found in Tables 5.3 and 5.4, respectively, for 1 million sample sets and 20 million sample sets. For these tables, Med. refers to median, Std. refers to standard deviation, Min. refers to minimum, Max. refers to maximum, Pe_mean refers to percent error of the mean versus $N(0,1)$, Pe_med. refers to percent error of the median versus $N(0,1)$, Pe_std. refers to percent error of the standard deviation versus $N(0,1)$, Chi p refers to Chi-squared goodness of fit p-value, A-D p refers to the Anderson-Darling goodness of fit p-value, $R^2$ refers to the coefficient of determination and SEE refers to the standard error of the estimate.

From analysis of Tables 5.5 and 5.6, all four test vector sets represent $N(0,1)$ well. The smaller sample sets of 1 million show less magnitude in the minimum and maximum categories, which is as expected since those are the tail ends of the distribution. Seed 3 shows a median value of exactly 0, which is impressive. For both Chi p and A-D p,
### 5.4 Test Vector Probability and Statistic Analysis

<table>
<thead>
<tr>
<th></th>
<th>Seed 1</th>
<th></th>
<th>Seed 4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N1</td>
<td>N2</td>
<td>N1</td>
<td>N2</td>
</tr>
<tr>
<td>Mean</td>
<td>-0.000962</td>
<td>0.000125</td>
<td>0.000480</td>
<td>0.001550</td>
</tr>
<tr>
<td>Med.</td>
<td>-0.001003</td>
<td>-0.000721</td>
<td>-0.000176</td>
<td>0.001717</td>
</tr>
<tr>
<td>Std.</td>
<td>1.000454</td>
<td>0.999101</td>
<td>0.998968</td>
<td>0.999692</td>
</tr>
<tr>
<td>Min.</td>
<td>-4.885721</td>
<td>-4.635759</td>
<td>-4.644117</td>
<td>-4.685152</td>
</tr>
<tr>
<td>Max.</td>
<td>4.731121</td>
<td>4.808264</td>
<td>4.953455</td>
<td>4.929619</td>
</tr>
<tr>
<td>Pe_mean</td>
<td>0.096233</td>
<td>0.012474</td>
<td>0.048030</td>
<td>0.154974</td>
</tr>
<tr>
<td>Pe_med.</td>
<td>0.100327</td>
<td>0.072098</td>
<td>0.017643</td>
<td>0.171661</td>
</tr>
<tr>
<td>Pe_std.</td>
<td>0.045448</td>
<td>0.089864</td>
<td>0.103159</td>
<td>0.030834</td>
</tr>
<tr>
<td>Chi p</td>
<td>0.658276</td>
<td>0.533465</td>
<td>0.539513</td>
<td>0.320134</td>
</tr>
<tr>
<td>A-D p</td>
<td>0.866158</td>
<td>0.543795</td>
<td>0.543483</td>
<td>0.905227</td>
</tr>
<tr>
<td>$R^2$</td>
<td>0.9999977466</td>
<td>0.9999965110</td>
<td>0.9999967027</td>
<td>0.9999965679</td>
</tr>
<tr>
<td>SEE</td>
<td>0.001502</td>
<td>0.001866</td>
<td>0.001814</td>
<td>0.001852</td>
</tr>
</tbody>
</table>

Table 5.5: Goodness-of-Fit Metrics for 1 Million Count Test Vector Sets

A value greater than 0.05 proves the distribution is of $N(0,1)$, where Seed 2 shows a tremendous value of 0.956603 for A-D $p$. 
### Table 5.6: Goodness-of-Fit Metrics for 20 Million Test Vector Sets

<table>
<thead>
<tr>
<th></th>
<th>Seed 2</th>
<th>Seed 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N1</td>
<td>N2</td>
</tr>
<tr>
<td>Mean</td>
<td>0.000644</td>
<td>0.000405</td>
</tr>
<tr>
<td>Med.</td>
<td>0.000633</td>
<td>0.000418</td>
</tr>
<tr>
<td>Std.</td>
<td>0.999892</td>
<td>0.999789</td>
</tr>
<tr>
<td>Min.</td>
<td>-5.379921</td>
<td>-5.642990</td>
</tr>
<tr>
<td>Max.</td>
<td>5.397518</td>
<td>5.335030</td>
</tr>
<tr>
<td>Pe_mean.</td>
<td>0.064446</td>
<td>0.040544</td>
</tr>
<tr>
<td>Pe_med.</td>
<td>0.063324</td>
<td>0.041771</td>
</tr>
<tr>
<td>Pe_std.</td>
<td>0.010758</td>
<td>0.021119</td>
</tr>
<tr>
<td>Chi p</td>
<td>0.409535</td>
<td>0.650388</td>
</tr>
<tr>
<td>A-D p</td>
<td>0.956603</td>
<td>0.663481</td>
</tr>
<tr>
<td>$R^2$</td>
<td>0.9999998758</td>
<td>0.9999998549</td>
</tr>
<tr>
<td>SEE</td>
<td>0.000352</td>
<td>0.000381</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusion

This chapter discusses conclusions from this work, while also indicating possible future work.

6.1 Future Work

6.1.1 Pipeline Extension

LGBMGNG was designed with simplicity in mind for the pipeline. This means that there are multiple ways to improve the hardware design. One of these ways would be to multi-cycle path the large multipliers by designating a separate always block, registering the inputs the output and declaring it a multi-cycle path. Another would be to register the outputs of the look up tables. It is believed with the extension of the pipeline, the area will drop drastically due to the increase in required time for the large multiplier instances resulting in the synthesis selecting much less aggressive implementations.
6.1.2 Architectural Changes

The error analysis process could be applied for 32-bit noise or higher. The sample size could be increased even more by generating more bits via Tausworthe random number generators. A single, larger Tausworthe generator could be used to minimize the number of seeds needed. Another method for exact rounding could be sought after for sine and cosine to turn the entire design into being exactly rounded. The central limit theorem could be applied to the samples due to the nature of the theorem bringing about any set of samples closer to a normal distribution. Also, other papers utilize it on the output samples of the Box-Muller method. [9, 30].

6.1.3 Approximation Changes

The trade-off between coefficient look up tables and length of the data path arithmetic could be better understood by implementing this design with different combinations (eg. degree two approximation for SQRT). The MATLAB fixed-point designer and fixed-point optimization tools could be explored for applying to this work [31, 32]. A software program could be developed that automates the entire error analysis when passed a sample size and output bit size.

6.2 Conclusions

This paper walked through the process of designing a 24-bit Gaussian noise generator based on the Box-Muller method where the core of the design stemmed from a thorough error analysis. The analysis and design drew insights from a 16-bit design, where the increase in 8 bits of noise led to a significant increase in internal signal sizes and degrees of approximation for the elementary functions. The error analysis was abstracted to a general case that can be applied for a range of sample sizes and output bit widths.
Multipliers accounted for half of the design’s area, while the coefficient look up tables only accounted for around a quarter of the design’s area. The design was written in Verilog HDL, targeted for a 65nm ASIC process and synthesized at 400 MHz where it generates 800 million samples of Gaussian noise per second and maintains periodicity up to a sample size of $2 \cdot 10^{20}$ due to achieving a maximum value of $9.42\sigma$. 
References


References


Appendix I

Source Code

I.1 LGBMGNG Design in Verilog HDL

```verilog
// Author: Lincoln Glauser
// August 2017
module LGBMGNG (  
    reset,  
    clk,  
    seed_a,  
    seed_b,  
    seed_c,  
    n1,  
    n2,  
    valid,  
    scan_in0,  
    scan_en,  
    test_mode,  
    scan_out0
);  

/*  
* LG’s Box Muller Gaussian Noise Generator (LGBMGNG)  
* generates three independent uniform 32-bit random numbers [U(0,1)]  
* and then produces two samples of 24-bit gaussian noise [N(0,1)].  
*/  

input  
    reset,  // system reset  
    clk;  // system clock
```
I.1 LGBMGNG Design in Verilog HDL

```verilog
input [31:0] // Seeds for TAUS
seed_a,
seed_b,
seed_c;

output reg signed [23:0]
n1, // 24 bit N(0,1)
n2; // 24 bit N(0,1)

output reg valid;

input
scan_in0, // test scan mode data input
scan_en, // test scan mode enable
test_mode; // test mode select

output
scan_out0; // test scan mode data output

//---------------------------------//--

// Note:
// TB: Total Bits
// FB: Fractional Bits
// RB: Redundant Bits
// SB: Sign Bit

// TAUSWORTHE
wire [31:0]
a,
b,
c;

// Actual Inputs

// UQ64
wire [63:0] u0;
parameter TBu0 = 64;

// UQ32
wire [31:0] u1;

//---------------------------------------------------------------
// LN UNIT /// LN UNIT /// LN UNIT /// LN UNIT /// LN UNIT ///
//---------------------------------------------------------------

// LN UNIT -- e = -2*ln(u0)
// Range Reduction
parameter TBexp_e = 7;
wire [6:0] u0_LZD_cnt;
wire [TBexp_e-1:0] exp_e;
```
// UQ1.63
parameter FBx_e = 63;
wire [63:0] x_e;

// Degree 3 Function Approximation. ln(x), x is [1,2)
parameter k_ln = 9; // 512 Segments
parameter FBxl_e = FBx_e-k_ln;

wire [k_ln-1:0] xm_e;
wire [FBxl_e-1:0] xl_e;

// C3_LN UQ47
parameter FBC3e = 47;
parameter RBC3e = 28;
wire [FBC3e-RBC3e-1:0] raw_val_C3e; // Values stored in LUT

// UQ47 D4e
// D4e = xl_e * C3e;
parameter FBD4e = 47;
reg [k_ln-1:0] sel_D4e;
reg [FBxl_e-1:0] xl_e_D4e;
wire [2*FBxl_e-1:0] D4e_next_raw;
wire [FBD4e-1:0] D4e_next;
reg [FBD4e-1:0] D4e;

// C2_LN Q47 $$ Always Negative $$
parameter FBC2e = 47;
parameter RBC2e = 19;
wire [FBC2e-RBC2e-1:0] raw_val_C2e; // Values stored in LUT
wire signed [FBC2e:0] val_C2e;

// Q47 D3e
// D3e = D4e + C2e;
parameter FBD3e = 47;
reg [k_ln-1:0] sel_D3e;
reg [FBxl_e-1:0] xl_e_D3e;
wire signed [FBD3e-1:0] D3e_next;
reg signed [FBD3e-1:0] D3e;

// Q48 D2e
// D2e = D3e * xl_e;
parameter FBD2e = 48;
reg [k_ln-1:0] sel_D2e;
reg signed [FBxl_e:0] xl_e_D2e;
wire signed [2*(FBxl_e+1)-1:0] D2e_next_raw;
wire signed [FBD2e-1:0] D2e_next;
reg signed [FBD2e-1:0] D2e;

// UQ48 D1e
// D1e = D2e + C1e;
parameter FBD1e = 48;
reg [k_ln-1:0] sel_D1e;
reg [FBxl_e-1:0] xl_e_D1e;
wire signed [FBD1e-1:0] D1e_next;
reg [FBD1e-1:0] D1e;

// C1_LN Q47
parameter FBC1e = 47;
parameter RBC1e = 9;
wire [FBC1e-RBC1e-1:0] raw_val_C1e; // Values stored in LUT
wire signed [FBD1e-1:0] val_C1e; // Prep for signed arithmetic since D2 is negative.

// UQ48 D0e
// D0e = xl_e * D1e;
parameter FBDOe = 48;
reg [k_ln-1:0] sel_D0e;
reg [FBxl_e-1:0] xl_e_D0e;
wire [125:0] D0e_next_raw;
wire [FBD0e-1:0] D0e_next;
reg [FBD0e-1:0] D0e;

// CO_LN UQ47
parameter FBC0e = 47;
parameter RBC0e = 0;
wire [FBC0e-RBC0e-1:0] raw_val_C0e; // Values stored in LUT

// UQ43 ye
// ye = D0e + C0e;
parameter FBye = 43;
reg [k_ln-1:0] sel_ye;
wire [FBD0e-1:0] ye_next_raw;
wire [FBye-1:0] ye_next;
reg [FBye-1:0] ye;

// Range Reconstruction
// UQ6.43 = UQ7.0 * UQ29
// e_p = exp_e * LN2;
// exp_e pipeline bc val needed for e_p_raw
reg [TBexp_e-1:0] exp_e_D4e;
reg [TBexp_e-1:0] exp_e_D3e;
reg [TBexp_e-1:0] exp_e_D2e;
reg [TBexp_e-1:0] exp_e_D1e;
reg [TBexp_e-1:0] exp_e_D0e;
reg [TBexp_e-1:0] exp_e_ye;
reg [TBexp_e-1:0] exp_e_e;
wire [54:0] e_p_raw; // TB = 49 + 6;
wire [48:0] e_p;

// UQ7.40 = (UQ6.43-UQ43)*(2*2^43)
// e = (e_p-y_e)*2;
wire [48:0] e_offset;
wire [49:0] e_next_raw;
wire [46:0] e_next;
I.1 LGBMGNG Design in Verilog HDL

```
reg [46:0] e;

///////////////////////////////////////////////
// SQRT UNIT /// SQRT UNIT /// SQRT UNIT /// SQRT UNIT/
///////////////////////////////////////////////

// SQRT Unit -- f = sqrt(e) = sqrt(-2ln(u0))

// Range Reduction

// Q6.0 expf = 5-LZD(e);
parameter TBexp_f = 7; // 1SB 6IB
wire [TBexp_f-2:0] e_LZD_cnt;
wire signed [TBexp_f-1:0] exp_f;

// x_f_p = e >> exp_f;
// UQ2.45 = UQ7.40 >> exp_f;
parameter FBx_f_p = 45;
parameter TBx_f_p = 47;
wire [TBx_f_p-1:0] x_f_p;

// x_f = exp_f[0] ? x_f_p/2 : x_f_p;
// UQ2.46
parameter FBx_f = 46;
parameter TBx_f = 48;
wire [TBx_f-1:0] x_f;

// Degree 1 Function Approximation. sqrt(x), x is [2,4)
parameter k_f = 10; // 1024 Segments
parameter FBxl_f = FBx_f-k_f;
wire [k_f-1:0] xm_f;
wire [FBxl_f-1:0] xl_f;

// C1_SQRT UQ26
parameter FBC1f = 26;
parameter RBC1f = 10;
wire [FBC1f-RBC1f-1:0] val_C1f;
wire [FBC1f-RBC1f-2:0] raw_val_C1f_1_2; // Values stored in LUT Note: 1_2 has one more RB
wire [FBC1f-RBC1f-1:0] raw_val_C1f_2_4; // Values stored in LUT

// C0_SQRT UQ25 Dof
// Dof = xl_f * C1f;
parameter FBDof = 25;
reg [k_f-1:0] sel_Dof;
reg [FBxl_f-1:0] xl_f_Dof;
reg signed [TBexp_f-1:0] exp_f_Dof;
wire [2*FBxl_f-1:0] Dof_next_raw;
wire [FBDof-1:0] Dof_next;
reg [FBDof-1:0] Dof;
```

parameter FBC0f = 25;
parameter RBC0f = 0;
wire [FBC0f:0] val_C0f; // Values stored in LUT
wire [FBC0f:0] raw_val_C0f_1_2; // Values stored in LUT
wire [FBC0f:0] raw_val_C0f_2_4; // Values stored in LUT

// UQ1.23 yf
// yf = D0f + C0f;
parameter FByf = 43;
reg [k_f-1:0] sel_yf;
reg signed [TBexp_f-1:0] exp_f_yf;
wire [FBD0f:0] yf_next_raw;
wire [FByf-1:0] yf_next;
reg [FByf-1:0] yf;

// Range Reconstruction
// exp_f_p = exp_f[0] ? exp_f+1>>1 : exp_f>>1;
parameter TBexp_f_p = 6; // 1SB 5IB
reg signed [TBexp_f-1:0] exp_f_f;
wire signed [TBexp_f_p-1:0] exp_f_p;

// f UQ4.20
// f = y_f << exp_f_p;
parameter FBf = 20;
parameter TBf = 24;
wire [46:0] f_next_raw;
wire [TBf-1:0] f_next;
reg [TBf-1:0] f;
wire signed [TBf:0] f_g;

////////////////////////////////////////////////////////
// SIN/COS UNIT ////// SIN/COS UNIT ////// SIN/COS UNIT //////
////////////////////////////////////////////////////////

// SIN/COS Unit -- g0 = sin(2*pi*u1) & g1 = cos(2*pi*u1)
wire [1:0] Q;

// xg_a/xg_b UQ30
parameter FBx_g = 30;
wire [FBx_g-1:0] xg_a;
wire [FBx_g-1:0] xg_b;

// Degree 2 Function Approximation. cos(x*pi/2), x is [0,1)
parameter k_g = 7; // 128 Segments
parameter FBxl_g = FBx_g-k_g;
wire [k_g-1:0] xm_g_a;
wire [FBxl_g-1:0] xl_g_a;
wire [k_g-1:0] xm_g_b;
wire [FBxl_g-1:0] xl_g_b;

// C2_COS Q27
parameter FBC2g = 27;
parameter RBC2g = 13;

wire [FBC2g-RBC2g-1:0] raw_val_C2g_a; // Values stored in LUT
wire signed [FBC2g:0] val_C2g_a;
wire [FBC2g-RBC2g-1:0] raw_val_C2g_b; // Values stored in LUT
wire signed [FBC2g:0] val_C2g_b;

// Q27 D2g
// Q27 = Q23 * Q27
// D2g = xl_g * C2g;
parameter FBD2g = 27;
reg [k_g-1:0] sel_D2g_a;
reg [FBxl_g:0] xl_g_D2g_a;
wire signed [2*(FBC2g+1)-1:0] D2g_next_raw_a;
wire signed [FBD2g:0] D2g_next_a;
reg signed [FBD2g:0] D2g_a;

reg [k_g-1:0] sel_D2g_b;
reg [FBxl_g:0] xl_g_D2g_b;
wire signed [2*(FBC2g+1)-1:0] D2g_next_raw_b;
wire signed [FBD2g:0] D2g_next_b;
reg signed [FBD2g:0] D2g_b;

// C1_COS Q27
parameter FBC1g = 27;
parameter RBC1g = 6;

wire [FBC1g-RBC1g-1:0] raw_val_C1g_a; // Values stored in LUT
wire signed [FBC1g:0] val_C1g_a;
wire [FBC1g-RBC1g-1:0] raw_val_C1g_b; // Values stored in LUT
wire signed [FBC1g:0] val_C1g_b;

// D1g Q28
// Q28 = Q27 + Q27;
// D1g = D2g + C1g;
parameter FBD1g = 28;
reg [k_g-1:0] sel_D1g_a;
reg [FBxl_g:0] xl_g_D1g_a;
wire signed [FBD1g:0] D1g_next_a;
reg signed [FBD1g:0] D1g_a;
reg [k_g-1:0] sel_D1g_b;
reg signed [FBxl_g:0] xl_g_D1g_b;
wire signed [FBD1g:0] D1g_next_b;
reg signed [FBD1g:0] D1g_b;

// Q28 D0g
// Q28 = Q23 * Q28
// D0g = xl_g * D1g;
parameter FBD0g = 28;

reg [k_g-1:0] sel_D0g_a;
reg signed [FBxl_g:0] xl_g_D0g_a;
wire signed [2*(FBD1g+1)-1:0] D0g_next_raw_a;
wire signed [FBD0g:0] D0g_next_a;
reg signed [FBD0g:0] D0g_a;

reg [k_g-1:0] sel_D0g_b;
reg signed [FBxl_g:0] xl_g_D0g_b;
wire signed [2*(FBD1g+1)-1:0] D0g_next_raw_b;
wire signed [FBD0g:0] D0g_next_b;
reg signed [FBD0g:0] D0g_b;

// C0_COS Q1.27
parameter FBC0g = 27;
parameter RBC0g = 0;
wire [FBC0g-RBC0g:0] raw_val_C0g_a; // Values stored in LUT
wire signed [FBC0g+1:0] val_C0g_a;
wire [FBC0g-RBC0g:0] raw_val_C0g_b; // Values stored in LUT
wire signed [FBC0g+1:0] val_C0g_b;

// yg UQ24
// UQ24 = Q28 + Q1.27;
// yg = D0g + C0g;
parameter FByg = 24;

reg [k_g-1:0] sel_yg_a;
wire signed [FBDOg+1:0] yg_next_raw_a;
wire [FByg+1:0] yg_next_raw_rtn_a;
wire [FByg+1:0] yg_next_a;
reg [FByg-1:0] yg_a;

reg [k_g-1:0] sel_yg_b;
wire signed [FBDOg+1:0] yg_next_raw_b;
wire [FByg+1:0] yg_next_raw_rtn_b;
wire [FByg-1:0] yg_next_b;
reg [FByg-1:0] yg_b;

// Range Reconstruction
// switch(Q)
// case 0: g0 = y_g_b; g1 = y_g_a;
// case 1: g0 = y_g_a; g1 = -y_g_b;
// case 2: g0 = -y_g_b; g1 = y_g_a;
// case 3: g0 = -y_g_a; g1 = y_g_b;

// need to pipeline Q val
reg [1:0] Q_D2g;
reg [1:0] Q_D1g;
reg [1:0] Q_D0g;
reg [1:0] Q yg;
reg [1:0] Q g;

// g Q24
parameter FBg = 24;
wire signed [FBg:0] g0_next;
wire signed [FBg:0] g1_next;
reg signed [FBg:0] g0;
reg signed [FBg:0] g1;

// Noise Creation

// Pipeline g0,g1
reg signed [FBg:0] g0_pipe0;
reg signed [FBg:0] g1_pipe0;
reg signed [FBg:0] g0_pipe1;
reg signed [FBg:0] g1_pipe1;
reg signed [FBg:0] g0_pipe2;
reg signed [FBg:0] g1_pipe2;
reg signed [FBg:0] g0_pipe3;
reg signed [FBg:0] g1_pipe3;
reg signed [FBg:0] g0_pipe4;
reg signed [FBg:0] g1_pipe4;
reg signed [FBg:0] g0_f;
reg signed [FBg:0] g1_f;

wire signed [57:0] f_g_extended;
wire signed [57:0] g0_f_extended;
wire signed [57:0] g1_f_extended;
wire signed [57:0] n1_next_raw;
wire signed [57:0] n2_next_raw;
wire signed [23:0] n1_next;
wire signed [23:0] n2_next;

// valid bit pipeline
reg valid_pipe;
reg valid_pipe0;
reg valid_pipe1;
reg valid_pipe2;
reg valid_pipe3;
reg valid_pipe4;
reg valid_pipe5;
reg valid_pipe6;
reg valid_pipe7;
reg valid_pipe8;
reg valid_pipe9;
reg valid_pipe_last;

//-----------------------------------------/

TAUS TAUS_a {
    .reset(reset),
    .clk(clk),
    .seed(seed_a),
    .u0(a),
    .scan_in0(scan_in0),
    .scan_en(scan_en),
    .test_mode(test_mode),
    .scan_out0(scan_out0)
};

TAUS TAUS_b {
    .reset(reset),
    .clk(clk),
    .seed(seed_b),
    .u0(b),
    .scan_in0(scan_in0),
    .scan_en(scan_en),
    .test_mode(test_mode),
    .scan_out0(scan_out0)
};

TAUS TAUS_c {
    .reset(reset),
    .clk(clk),
    .seed(seed_c),
    .u0(c),
    .scan_in0(scan_in0),
    .scan_en(scan_en),
    .test_mode(test_mode),
    .scan_out0(scan_out0)
};

LZD_u0 LZD_LN {
    .reset(reset),
    .clk(clk),
    .val(u0),
    .LZ_count(u0_LZD_cnt),
    .scan_in0(scan_in0),
    .scan_en(scan_en),
    .test_mode(test_mode),
    .scan_out0(scan_out0)
};

C3_LN C3_LN {
    .reset(reset),
    .clk(clk),
I.1 LGBMGNG Design in Verilog HDL

```verilog
. sel(sel_D4e),
. val(raw_val_C3e),
. scan_in0(scan_in0),
. scan_en(scan_en),
. test_mode(test_mode),
. scan_out0(scan_out0)
);

C2_LN C2_LN (  
. reset(reset),
. clk(clk),
. sel(sel_D3e),
. val(raw_val_C2e),
. scan_in0(scan_in0),
. scan_en(scan_en),
. test_mode(test_mode),
. scan_out0(scan_out0)
);

C1_LN C1_LN (  
. reset(reset),
. clk(clk),
. sel(sel_D1e),
. val(raw_val_C1e),
. scan_in0(scan_in0),
. scan_en(scan_en),
. test_mode(test_mode),
. scan_out0(scan_out0)
);

C0_LN C0_LN (  
. reset(reset),
. clk(clk),
. sel(sel_ye),
. val(raw_val_C0e),
. scan_in0(scan_in0),
. scan_en(scan_en),
. test_mode(test_mode),
. scan_out0(scan_out0)
);

LZD_e LZD_SQRT (  
. reset(reset),
. clk(clk),
. val(e),
. LZ_count(e_LZD_cnt),
. scan_in0(scan_in0),
. scan_en(scan_en),
. test_mode(test_mode),
. scan_out0(scan_out0)
);

C1_SQRT_1_2 C1_SQRT_1_2 (  
. reset(reset),
. clk(clk),
. sel(sel_D0f),
. val(raw_val_C1f_1_2),
```
I.1 LGBMNG Design in Verilog HDL

```verilog
C1_SQRT_2_4 C1_SQRT_2_4 (reset, clk, sel, val, .scan_in0, .scan_en, .test_mode, .scan_out0);
C0_SQRT_1_2 C0_SQRT_1_2 (reset, clk, sel, val, .scan_in0, .scan_en, .test_mode, .scan_out0);
C0_SQRT_2_4 C0_SQRT_2_4 (reset, clk, sel, val, .scan_in0, .scan_en, .test_mode, .scan_out0);
C2_COS C2_COS_a (reset, clk, sel, val, .scan_in0, .scan_en, .test_mode, .scan_out0);
C2_COS C2_COS_b (reset, clk, sel, val, .scan_in0, .scan_en, .test_mode, .scan_out0);
```
.test_mode(test_mode),
.scan_out0(scan_out0)
);

C1_COS C1_COS_a (
.reset(reset),
.clk(clk),
.sel(sel_D1g_a),
.val(raw_val_C1g_a),
.scan_in0(scan_in0),
.scan_en(scan_en),
.test_mode(test_mode),
.scan_out0(scan_out0)
);

C1_COS C1_COS_b (
.reset(reset),
.clk(clk),
.sel(sel_D1g_b),
.val(raw_val_C1g_b),
.scan_in0(scan_in0),
.scan_en(scan_en),
.test_mode(test_mode),
.scan_out0(scan_out0)
);

C0_COS C0_COS_a (
.reset(reset),
.clk(clk),
.sel(sel_yg_a),
.val(raw_val_C0g_a),
.scan_in0(scan_in0),
.scan_en(scan_en),
.test_mode(test_mode),
.scan_out0(scan_out0)
);

C0_COS C0_COS_b (
.reset(reset),
.clk(clk),
.sel(sel_yg_b),
.val(raw_val_C0g_b),
.scan_in0(scan_in0),
.scan_en(scan_en),
.test_mode(test_mode),
.scan_out0(scan_out0)
);

// Actual Inputs
assign u0 = {a,b};
assign u1 = c;

// LN Range Reduction
I.1 LGBMGNG Design in Verilog HDL

```verbatim
assign exp_e = u0_LZD_cnt + 1;
assign x_e = u0 << (exp_e-1);

// LN Degree Three Approximation
assign xm_e = x_e[FBx_e-1:FBx_e-k_ln];
assign xl_e = x_e[FBx_e-k_ln-1:0];

// D4e = xl_e * C3e;
// UQ47 = UQFBx_l_e * UQ47;
assign D4e_next_raw = xl_e_D4e * {raw_val_C3e,((FBx1_e - FBC3e){1'b0})};
assign D4e_next = D4e_next_raw[2*FBx1_e-1:2*FBx1_e-FBD4e] +
                D4e_next_raw[2*FBx1_e-FBD4e-1]; // w/ Round to nearest

// D3e = D4e + C2e;
// Q47 = UQ47 + Q47;
assign val_C2e = {{(RBC2e+1){1'b0}},raw_val_C2e}; // Executes sign extension.
                // C2 stored in 2s comp w/o SB
assign D3e_next = D4e + val_C2e;

// D2e = xl_e * D3e;
// Q48 = QFBx1_e * Q47
assign D2e_next_raw = {{55{1'b0}},xl_e_D2e} * {{55{D3e[47]}},D3e,((FBx1_e -
                FBD3e){1'b0})};
assign D2e_next =
                D2e_next_raw[2*(FBx1_e+1)-1],D2e_next_raw[2*FBx1_e-1:2*FBx1_e-FBD2e] +
                D2e_next_raw[2*FBx1_e-FBD2e-1]; // w/ Round to nearest plus SB stuff

// D1e = D2e + C1e;
// Q48 = Q48 + Q47;
assign val_C1e = {{1'b0},raw_val_C1e,1'b0};
assign D1e_next = D2e + val_C1e;

// D0e = xl_e * D1e;
// UQ48 = UQFBx_e * UQ48;
assign D0e_next_raw = {xl_e_D0e,{k_ln{1'b0}}} * {D1e,((FBx_e - FBD1e){1'b0})};
assign D0e_next = D0e_next_raw[2*FBx_e-1:2*FBx_e-FBD0e] +
                D0e_next_raw[2*FBx_e-FBD0e-1]; // w/ Round to nearest

// ye = D0e + C0e;
// UQ43 = UQ48 + UQ47;
assign ye_next_raw = D0e + {raw_val_C0e,1'b0};

// Range Reconstruction
// ln2 UQ49
`define LN2 49'h162E42FEFA39F // 390207173010335 -- 49 FB
assign exp_e = exp_e * LN2;
assign e_p_raw = exp_e_e * `LN2;
// e_p UQ6.43 UQ7.0 * UQ49
assign e_p = e_p_raw[54:6] + e_p_raw[5]; // 49FB -> 43FB Q6.43 w/
                // Round to nearest
```
// e = (e_p-ye)*2;
// e UQ7.40 = UQ6.43 - UQ43
assign e_offset = (e_p-ye);    // Q6.43
assign e_next_raw = {e_offset,1'b0}; // Q7.43
assign e_next = e_next_raw[49:3] + e_next_raw[2];    // Q7.40 w/ Round to nearest LP TODO: can optimize slightly (skip mid steps)

///////////////////////////////////////////////////////////
// SQRT UNIT /// SQRT UNIT /// SQRT UNIT /// SQRT UNIT/
///////////////////////////////////////////////////////////

// Range Reduction
// expf = 5-LZD(e);
assign exp_f = 5 - e_LZD_cnt;

// x_f_p = e >> exp_f;
// UQ2.45 = UQ7.40 >> exp_f;
assign x_f_p = e << e_LZD_cnt;

// x_f = exp_f[0] ? x_f_p/2 : x_f_p;
// UQ2.46
assign x_f = exp_f[0] ? x_f_p : {x_f_p,1'b0};    // opposite nature due to addition of one FB vs x_f_p

// SQRT Degree One Approximation
assign xm_f = exp_f[0] ? x_f[FBx_f-1:FBx_f-k_f] : x_f[FBx_f-k_f-1:0];
assign xl_f = exp_f[0] ? x_f[FBx_f-k_f-1:0] : x_f[FBx_f-k_f:1];

// D0f = xl_f * C1f;
// UQ25 = UQFBxl_f * UQ26;
assign val_C1f = exp_f_D0f[0] ? {1'b0,raw_val_C1f_1_2}: raw_val_C1f_2_4;
assign D0f_next_raw = xl_f_D0f * {val_C1f,{(FBxl_f - FBC1f){1'b0}}};
assign D0f_next = D0f_next_raw[2*FBxl_f-1:2*FBxl_f-FBD0f] +
    D0f_next_raw[2*FBxl_f-FBD0f-1]; // w/ Round to nearest

// yf = D0f + C0f;
// UQ1.23 = UQ25 + UQ1.25;
assign val_C0f = exp_f_yf[0] ? raw_val_C0f_1_2 : raw_val_C0f_2_4;
assign yf_next_raw = D0f + val_C0f;
assign yf_next = yf_next_raw[25:2] + yf_next_raw[1]; // w/ Round to nearest

// Range Reconstruction
// exp_f_p = exp_f[0] ? exp_f+1>>1 : exp_f>>1;
// Q5.0 from Q6.0
assign exp_f_p_raw = exp_f[0] ? exp_f + 1 : exp_f;
assign exp_f_p = exp_f_p_raw[6:1];

// f UQ4.20
// f = y_f << exp_f_p;
assign f_next_raw = yf << (20 + exp_f_p);
assign f_next = f_next_raw[46:23] + f_next_raw[22];
I.1 LGBMGNG Design in Verilog HDL

743  
744  /////////////////////////////////////////////////////////////////////////////
745  // SIN/COS UNIT /// SIN/COS UNIT /// SIN/COS UNIT ///
746  /////////////////////////////////////////////////////////////////////////////
747  // Range Reduction
748  // UQ2.0
749  assign Q = u1[31:30];
750  // xg UQ30
751  assign xg_a = u1[FBx_g-1:0];
752  assign xg_b = ~xg_a;
753  // Degree Two Function Approximation
754  assign xm_g_a = xg_a[FBx_g-1:FBx_g-k_g];
755  assign xl_g_a = xg_a[FBx_g-k_g-1:0];
756  assign xm_g_b = xg_b[FBx_g-1:FBx_g-k_g];
757  assign xl_g_b = xg_b[FBx_g-k_g-1:0];
758  // D2g Q27
759  // Q27 = Q23 * Q27
760  // D2g = xl_g * C2g;
761  assign val_C2g_a = {{(RBC2g+1){1'b1}},raw_val_C2g_a};
762  assign val_C2g_b = {{(RBC2g+1){1'b1}},raw_val_C2g_b};
763  assign D2g_next_raw_a = {{28{1'b0}},xl_g_D2g_a,4'b0000} * {{28{1'b1}},val_C2g_a};
764  assign D2g_next_raw_b = {{28{1'b0}},xl_g_D2g_b,4'b0000} * {{28{1'b1}},val_C2g_b};
765  // D1g Q28
766  // Q28 = Q27 + Q27;
767  // D1g = D2g + C1g;
768  // All negative except 0 index
769  assign val_C1g_a = sel_D1g_a == 7'b0000000 ? 28'b0000000 : {{(RBC1g+1){1'b1}},raw_val_C1g_a};
770  assign val_C1g_b = sel_D1g_b == 7'b0000000 ? 28'b0000000 : {{(RBC1g+1){1'b1}},raw_val_C1g_b};
771  assign D1g_next_a = {(D2g_a + val_C1g_a),1'b0};
772  assign D1g_next_b = {(D2g_b + val_C1g_b),1'b0};
773  // D0g Q28
774  // Q28 = Q23 * Q28
775  // D0g = xl_g * D1g;
I.1 LGBMGNG Design in Verilog HDL

assign D0g_next_raw_a = {{29{1'b0}},{xl_g_D0g_a,5'b00000}} * {{29{D1g_a[FB1d1g]}},D1g_a}; // 2s complement multiplication requires full end range sign extension

assign D0g_next_raw_b = {{29{1'b0}},{xl_g_D0g_b,5'b00000}} * {{29{D1g_b[FB1d1g]}},D1g_b}; // 2s complement multiplication requires full end range sign extension

assign D0g_next_a = D0g_next_raw_a[2*FB1d1g:2*FB1d1g-28] + D0g_next_raw_a[2*FB1d1g-29]; // w/ round to nearest
assign D0g_next_b = D0g_next_raw_b[2*FB1d1g:2*FB1d1g-28] + D0g_next_raw_b[2*FB1d1g-29]; // w/ round to nearest

// yg UQ24
// UQ24 = Q28 + Q1.27;
// yg = D0g + C0g;

assign val_C0g_a = {{(RBC0g+1){1'b0}},raw_val_C0g_a};
assign val_C0g_b = {{(RBC0g+1){1'b0}},raw_val_C0g_b};

// add sign extend to D0g
assign yg_next_raw_a = {D0g_a[FB1d0g],D0g_a} + {val_C0g_a,1'b0};
assign yg_next_raw_b = {D0g_b[FB1d0g],D0g_b} + {val_C0g_b,1'b0};

assign yg_next_raw_rtn_a = yg_next_raw_a[27:4] + yg_next_raw_a[3];
assign yg_next_raw_rtn_b = yg_next_raw_b[27:4] + yg_next_raw_b[3];

// [28]=1 shouldn't happen, but very rarely it does due to the approx nature. It is 1 in 2 cases, the addition result is greater than 1 (can't store) or the addition is negative (can't store)
// in case one (greater than 1), we force the highest representable value, 1-2^-FB, and in case two we force the lowest
// representable value, 0. Also, there is the case when round to nearest extends above representable range (1-2^-FB) and therefore forced to grabbing original result before rtn.

// Range Reconstruction
// switch(Q)

// case 0: g0 = yg_b; gl = yg_a;
// case 1: g0 = yg_a; gl = -yg_b;
// case 2: g0 = -yg_b; gl = -yg_a;
// case 3: g0 = -yg_a; gl = yg_b;

assign g0_next = ~Q_g[1] ? (~Q_g[0] ? {1'b0,yg_b} : {1'b0,yg_a}) : (-Q_g[0] ? {-1'b0,yg_b} + 1 : {-1'b0,yg_b} + 1);
assign gl_next = ~Q_g[1] ? (~Q_g[0] ? {1'b0,yg_a} : {-1'b0,yg_b}) : (-Q_g[0] ? {-1'b0,yg_a} + 1 : {1'b0,yg_b});
// Noise Creation
// n1 = f_g * g0;
// n2 = f_g * g1;
// Q4.19 = Q4.20 * Q24;
assign f_g = {1'b0,f};

assign f_g_extended = {{29{1'b0}},f_g,4'b0000};
assign g0_f_extended = {{33{g0_f[24]}},g0_f};
assign g1_f_extended = {{33{g1_f[24]}},g1_f};

assign n1_next_raw = f_g_extended * g0_f_extended;
assign n2_next_raw = f_g_extended * g1_f_extended;

assign n1_next = {n1_next_raw[57],n1_next_raw[51:29]} + n1_next_raw[28];
assign n2_next = {n2_next_raw[57],n2_next_raw[51:29]} + n2_next_raw[28];

always @(posedge reset or posedge clk)
begin
  if (reset)
  begin
    // D4e
    sel_D4e <= 0;
    xl_e_D4e <= 0;
    D4e <= 0;
    exp_e_D4e <= 0;

    // D3e
    sel_D3e <= 0;
    xl_e_D3e <= 0;
    D3e <= 0;
    exp_e_D3e <= 0;

    // D2e
    sel_D2e <= 0;
    xl_e_D2e <= 0;
    D2e <= 0;
    exp_e_D2e <= 0;

    // D1e
    sel_D1e <= 0;
    xl_e_D1e <= 0;
    D1e <= 0;
    exp_e_D1e <= 0;

    // D0e
    sel_D0e <= 0;
    xl_e_D0e <= 0;
    D0e <= 0;
    exp_e_D0e <= 0;

    // ye
sel_ye <= 0;
ye <= 0;
exp_e_ye <= 0;

// e
e <= 0;
exp_e_e <= 0;

// D0f
sel_D0f <= 0;
xl_f_D0f <= 0;
exp_f_D0f <= 0;
D0f <= 0;

// yf
sel_yf <= 0;
exp_f_yf <= 0;
yf <= 0;

// f
exp_f_f <= 0;
f <= 0;

// D2g_a
sel_D2g_a <= 0;
xl_g_D2g_a <= 0;
D2g_a <= 0;

// D2g_b
sel_D2g_b <= 0;
xl_g_D2g_b <= 0;
D2g_b <= 0;

// D1g_a
sel_D1g_a <= 0;
xl_g_D1g_a <= 0;
D1g_a <= 0;

// D1g_b
sel_D1g_b <= 0;
xl_g_D1g_b <= 0;
D1g_b <= 0;

// D0g_a
sel_D0g_a <= 0;
xl_g_D0g_a <= 0;
D0g_a <= 0;

// D0g_b
sel_D0g_b <= 0;
xl_g_D0g_b <= 0;
D0g_b <= 0;
// yg_a
sel_yg_a <= 0;
yg_a <= 0;

// yg_b
sel_yg_b <= 0;
yg_b <= 0;

// g
Q_D2g <= 0;
Q_D1g <= 0;
Q_D0g <= 0;
Q_yg <= 0;
Q_g <= 0;

// g
g0 <= 0;
g1 <= 0;
g0_pipe0 <= 0;
g1_pipe0 <= 0;
g0_pipe1 <= 0;
g1_pipe1 <= 0;
g0_pipe2 <= 0;
g1_pipe2 <= 0;
g0_pipe3 <= 0;
g1_pipe3 <= 0;
g0_pipe4 <= 0;
g1_pipe4 <= 0;
g0_f <= 0;
g1_f <= 0;

// Valid Bit
valid_pipe <= 0;
valid_pipe0 <= 0;
valid_pipe1 <= 0;
valid_pipe2 <= 0;
valid_pipe3 <= 0;
valid_pipe4 <= 0;
valid_pipe5 <= 0;
valid_pipe6 <= 0;
valid_pipe7 <= 0;
valid_pipe8 <= 0;
valid_pipe9 <= 0;
valid_pipe_last <= 0;
valid <= 0;

// LGBMGNG
n1 <= 0;
n2 <= 0;
end
else
begin

// D4e
sel_D4e <= xm_e;
xl_e_D4e <= xl_e;
D4e <= D4e_next;
exp_e_D4e <= exp_e;

// D3e
sel_D3e <= sel_D4e;
xl_e_D3e <= xl_e_D4e;
D3e <= D3e_next;
exp_e_D3e <= exp_e_D4e;

// D2e
sel_D2e <= sel_D3e;
xl_e_D2e <= {1'b0, xl_e_D3e};  // unsigned to signed conversion for
                                  // the signed arithmetic
D2e <= D2e_next;
exp_e_D2e <= exp_e_D3e;

// D1e
sel_D1e <= sel_D2e;
xl_e_D1e[FBxl_e-1:0] <= xl_e_D2e;  // signed to unsigned conversion
                                 // (xl_e_D2e Always pos)
D1e <= D1e_next[FBD1e-1:0];  // D1e_next was signed for
                              // arithmetic. but D1e always pos. ?todo;
exp_e_D1e <= exp_e_D2e;

// D0e
sel_D0e <= sel_D1e;
xl_e_D0e <= xl_e_D1e;
D0e <= D0e_next;
exp_e_D0e <= exp_e_D1e;

// ye
sel_ye <= sel_D0e;
ye <= ye_next;
exp_e_ye <= exp_e_D0e;

// e
e <= e_next;
exp_e_e <= exp_e_ye;

// D0f
sel_D0f <= xm_f;
xl_f_D0f <= xl_f;
exp_f_D0f <= exp_f;
D0f <= D0f_next;

// yf
sel_yf <= sel_D0f;
exp_f_yf <= exp_f_D0f;

yf <= yf_next;

// f
exp_f_f <= exp_f_yf;
f <= f_next;

// D2g_a
sel_D2g_a <= xm_g_a;
xl_g_D2g_a <= {1'b0,xl_g_a}; // unsigned to signed (add 0 SB)
D2g_a <= D2g_next_a;

// D2g_b
sel_D2g_b <= xm_g_b;
xl_g_D2g_b <= {1'b0,xl_g_b}; // unsigned to signed (add 0 SB)
D2g_b <= D2g_next_b;

// D1g_a
sel_D1g_a <= sel_D2g_a;
xl_g_D1g_a <= xl_g_D2g_a;
D1g_a <= D1g_next_a;

// D1g_b
sel_D1g_b <= sel_D2g_b;
xl_g_D1g_b <= xl_g_D2g_b;
D1g_b <= D1g_next_b;

// D0g_a
sel_D0g_a <= sel_D1g_a;
xl_g_D0g_a <= xl_g_D1g_a;
D0g_a <= D0g_next_a;

// D0g_b
sel_D0g_b <= sel_D1g_b;
xl_g_D0g_b <= xl_g_D1g_b;
D0g_b <= D0g_next_b;

// yg_a
sel_yg_a <= sel_D0g_a;
yg_a <= yg_next_a;

// yg_b
sel_yg_b <= sel_D0g_b;
yg_b <= yg_next_b;

// Q
Q_D2g <= Q;
Q_D1g <= Q_D2g;
Q_D0g <= Q_D1g;
Q_yg <= Q_D0g;
Q_g <= Q_yg;

// g
module TAUS (}

endmodule // LGBMGNG

Listing I.1: LGBMGNG, 24-bit Box-Muller Implementation Source Code

I.2 32-bit Tausworthe URNG
I.2 32-bit Tausworthe URNG

```verilog
reset,
clk,
seed,
u0,
scan_in0,
scan_en,
test_mode,
scan_out0);

input
reset, // system reset
clk; // system clock

input [31:0]
seed; // seed

output [31:0]
u0; // 32 bit U(0,1)

input
scan_in0, // test scan mode data input
scan_en, // test scan mode enable
test_mode; // test mode select

output
scan_out0; // test scan mode data output

reg [31:0]
s0,
s1,
s2;

wire [31:0]
b0,
b1,
b2,
s0_next,
s1_next,
s2_next;

assign b0 = ((s0 << 13) ^ s0) >> 19;
assign s0_next = ((s0 & 32'hFFFFFFFE) << 12) ^ b0;
assign b1 = ((s1 << 2) ^ s1) >> 25;
assign s1_next = ((s1 & 32'hFFFFFFFF) << 4) ^ b1;
assign b2 = ((s2 << 3) ^ s2) >> 11;
assign s2_next = ((s2 & 32'hFFFFFFFF) << 17) ^ b2;
```

I.3 Coefficient Look up Table

I.3 Coefficient Look up Table

Listing I.2: TAUS, 32-bit Tausworthe URNG

```verilog
assign u0 = s0 ^ s1 ^ s2;
always @(posedge reset or posedge clk)
begin
  if (reset)
    begin
      s0 <= seed;
      s1 <= seed;
      s2 <= seed;
    end
  else
    begin
      s0 <= s0_next;
      s1 <= s1_next;
      s2 <= s2_next;
    end
endmodule // TAUS
```

```verilog
module C2_COS (  
  input[6:0] sel,  
  output reg[13:0] val,  

  // DFT ports  
  input reset,  
  input clk,  
  input scan_en,  
  input scan_in0,  
  input test_mode,  
  output scan_out0  
);  
always @(sel) begin
  case(sel)
  0 : val[13:0] = 14'h1886;
  1 : val[13:0] = 14'h1887;
  2 : val[13:0] = 14'h188a;
  3 : val[13:0] = 14'h188f;
  4 : val[13:0] = 14'h1895;
  5 : val[13:0] = 14'h189d;
  default: val[13:0] = 14'h0000;
endcase
endmodule
```
<table>
<thead>
<tr>
<th>Index</th>
<th>Value (hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td><code>val[13:0] = 14'h18a6;</code></td>
</tr>
<tr>
<td>7</td>
<td><code>val[13:0] = 14'h18b0;</code></td>
</tr>
<tr>
<td>8</td>
<td><code>val[13:0] = 14'h18bc;</code></td>
</tr>
<tr>
<td>9</td>
<td><code>val[13:0] = 14'h18ca;</code></td>
</tr>
<tr>
<td>10</td>
<td><code>val[13:0] = 14'h18d9;</code></td>
</tr>
<tr>
<td>11</td>
<td><code>val[13:0] = 14'h18ea;</code></td>
</tr>
<tr>
<td>12</td>
<td><code>val[13:0] = 14'h18fc;</code></td>
</tr>
<tr>
<td>13</td>
<td><code>val[13:0] = 14'h1910;</code></td>
</tr>
<tr>
<td>14</td>
<td><code>val[13:0] = 14'h1925;</code></td>
</tr>
<tr>
<td>15</td>
<td><code>val[13:0] = 14'h193c;</code></td>
</tr>
<tr>
<td>16</td>
<td><code>val[13:0] = 14'h1954;</code></td>
</tr>
<tr>
<td>17</td>
<td><code>val[13:0] = 14'h196e;</code></td>
</tr>
<tr>
<td>18</td>
<td><code>val[13:0] = 14'h1989;</code></td>
</tr>
<tr>
<td>19</td>
<td><code>val[13:0] = 14'h19a6;</code></td>
</tr>
<tr>
<td>20</td>
<td><code>val[13:0] = 14'h19c4;</code></td>
</tr>
<tr>
<td>21</td>
<td><code>val[13:0] = 14'h19e3;</code></td>
</tr>
<tr>
<td>22</td>
<td><code>val[13:0] = 14'h1a04;</code></td>
</tr>
<tr>
<td>23</td>
<td><code>val[13:0] = 14'h1a27;</code></td>
</tr>
<tr>
<td>24</td>
<td><code>val[13:0] = 14'h1a4b;</code></td>
</tr>
<tr>
<td>25</td>
<td><code>val[13:0] = 14'h1a70;</code></td>
</tr>
<tr>
<td>26</td>
<td><code>val[13:0] = 14'h1a97;</code></td>
</tr>
<tr>
<td>27</td>
<td><code>val[13:0] = 14'h1ac0;</code></td>
</tr>
<tr>
<td>28</td>
<td><code>val[13:0] = 14'h1ae9;</code></td>
</tr>
<tr>
<td>29</td>
<td><code>val[13:0] = 14'h1b15;</code></td>
</tr>
<tr>
<td>30</td>
<td><code>val[13:0] = 14'h1b41;</code></td>
</tr>
<tr>
<td>31</td>
<td><code>val[13:0] = 14'h1b6f;</code></td>
</tr>
<tr>
<td>32</td>
<td><code>val[13:0] = 14'h1b9f;</code></td>
</tr>
<tr>
<td>33</td>
<td><code>val[13:0] = 14'h1bd0;</code></td>
</tr>
<tr>
<td>34</td>
<td><code>val[13:0] = 14'h1c02;</code></td>
</tr>
<tr>
<td>35</td>
<td><code>val[13:0] = 14'h1c36;</code></td>
</tr>
<tr>
<td>36</td>
<td><code>val[13:0] = 14'h1c6b;</code></td>
</tr>
<tr>
<td>37</td>
<td><code>val[13:0] = 14'h1ca1;</code></td>
</tr>
<tr>
<td>38</td>
<td><code>val[13:0] = 14'h1cd9;</code></td>
</tr>
<tr>
<td>39</td>
<td><code>val[13:0] = 14'h1d12;</code></td>
</tr>
<tr>
<td>40</td>
<td><code>val[13:0] = 14'h1d4c;</code></td>
</tr>
<tr>
<td>41</td>
<td><code>val[13:0] = 14'h1d88;</code></td>
</tr>
<tr>
<td>42</td>
<td><code>val[13:0] = 14'h1dc5;</code></td>
</tr>
<tr>
<td>43</td>
<td><code>val[13:0] = 14'h1e04;</code></td>
</tr>
<tr>
<td>44</td>
<td><code>val[13:0] = 14'h1e43;</code></td>
</tr>
<tr>
<td>45</td>
<td><code>val[13:0] = 14'h1e85;</code></td>
</tr>
<tr>
<td>46</td>
<td><code>val[13:0] = 14'h1ec7;</code></td>
</tr>
<tr>
<td>47</td>
<td><code>val[13:0] = 14'h1f0a;</code></td>
</tr>
<tr>
<td>48</td>
<td><code>val[13:0] = 14'h1f4f;</code></td>
</tr>
<tr>
<td>49</td>
<td><code>val[13:0] = 14'h1f96;</code></td>
</tr>
<tr>
<td>50</td>
<td><code>val[13:0] = 14'h1fdd;</code></td>
</tr>
<tr>
<td>51</td>
<td><code>val[13:0] = 14'h2026;</code></td>
</tr>
<tr>
<td>52</td>
<td><code>val[13:0] = 14'h2070;</code></td>
</tr>
<tr>
<td>53</td>
<td><code>val[13:0] = 14'h20bb;</code></td>
</tr>
<tr>
<td>54</td>
<td><code>val[13:0] = 14'h2107;</code></td>
</tr>
<tr>
<td>55</td>
<td><code>val[13:0] = 14'h2154;</code></td>
</tr>
<tr>
<td>56</td>
<td><code>val[13:0] = 14'h21a3;</code></td>
</tr>
<tr>
<td>57</td>
<td><code>val[13:0] = 14'h21f3;</code></td>
</tr>
</tbody>
</table>
I.3 Coefficient Look up Table

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>val[13:0] = 14'h2244;</td>
</tr>
<tr>
<td>59</td>
<td>val[13:0] = 14'h2296;</td>
</tr>
<tr>
<td>60</td>
<td>val[13:0] = 14'h22e9;</td>
</tr>
<tr>
<td>61</td>
<td>val[13:0] = 14'h233e;</td>
</tr>
<tr>
<td>62</td>
<td>val[13:0] = 14'h2393;</td>
</tr>
<tr>
<td>63</td>
<td>val[13:0] = 14'h23ea;</td>
</tr>
<tr>
<td>64</td>
<td>val[13:0] = 14'h2442;</td>
</tr>
<tr>
<td>65</td>
<td>val[13:0] = 14'h249a;</td>
</tr>
<tr>
<td>66</td>
<td>val[13:0] = 14'h24f4;</td>
</tr>
<tr>
<td>67</td>
<td>val[13:0] = 14'h254f;</td>
</tr>
<tr>
<td>68</td>
<td>val[13:0] = 14'h25ab;</td>
</tr>
<tr>
<td>69</td>
<td>val[13:0] = 14'h2608;</td>
</tr>
<tr>
<td>70</td>
<td>val[13:0] = 14'h2666;</td>
</tr>
<tr>
<td>71</td>
<td>val[13:0] = 14'h26c5;</td>
</tr>
<tr>
<td>72</td>
<td>val[13:0] = 14'h2725;</td>
</tr>
<tr>
<td>73</td>
<td>val[13:0] = 14'h2785;</td>
</tr>
<tr>
<td>74</td>
<td>val[13:0] = 14'h27e7;</td>
</tr>
<tr>
<td>75</td>
<td>val[13:0] = 14'h284a;</td>
</tr>
<tr>
<td>76</td>
<td>val[13:0] = 14'h28ae;</td>
</tr>
<tr>
<td>77</td>
<td>val[13:0] = 14'h2912;</td>
</tr>
<tr>
<td>78</td>
<td>val[13:0] = 14'h2977;</td>
</tr>
<tr>
<td>79</td>
<td>val[13:0] = 14'h29de;</td>
</tr>
<tr>
<td>80</td>
<td>val[13:0] = 14'h2a45;</td>
</tr>
<tr>
<td>81</td>
<td>val[13:0] = 14'h2aad;</td>
</tr>
<tr>
<td>82</td>
<td>val[13:0] = 14'h2b16;</td>
</tr>
<tr>
<td>83</td>
<td>val[13:0] = 14'h2b7f;</td>
</tr>
<tr>
<td>84</td>
<td>val[13:0] = 14'h2bea;</td>
</tr>
<tr>
<td>85</td>
<td>val[13:0] = 14'h2c55;</td>
</tr>
<tr>
<td>86</td>
<td>val[13:0] = 14'h2cc1;</td>
</tr>
<tr>
<td>87</td>
<td>val[13:0] = 14'h2d2d;</td>
</tr>
<tr>
<td>88</td>
<td>val[13:0] = 14'h2d9b;</td>
</tr>
<tr>
<td>89</td>
<td>val[13:0] = 14'h2e09;</td>
</tr>
<tr>
<td>90</td>
<td>val[13:0] = 14'h2e78;</td>
</tr>
<tr>
<td>91</td>
<td>val[13:0] = 14'h2ee7;</td>
</tr>
<tr>
<td>92</td>
<td>val[13:0] = 14'h2f57;</td>
</tr>
<tr>
<td>93</td>
<td>val[13:0] = 14'h2fc8;</td>
</tr>
<tr>
<td>94</td>
<td>val[13:0] = 14'h3039;</td>
</tr>
<tr>
<td>95</td>
<td>val[13:0] = 14'h30ab;</td>
</tr>
<tr>
<td>96</td>
<td>val[13:0] = 14'h311e;</td>
</tr>
<tr>
<td>97</td>
<td>val[13:0] = 14'h3191;</td>
</tr>
<tr>
<td>98</td>
<td>val[13:0] = 14'h3205;</td>
</tr>
<tr>
<td>99</td>
<td>val[13:0] = 14'h3279;</td>
</tr>
<tr>
<td>100</td>
<td>val[13:0] = 14'h32ee;</td>
</tr>
<tr>
<td>101</td>
<td>val[13:0] = 14'h3363;</td>
</tr>
<tr>
<td>102</td>
<td>val[13:0] = 14'h33d9;</td>
</tr>
<tr>
<td>103</td>
<td>val[13:0] = 14'h344f;</td>
</tr>
<tr>
<td>104</td>
<td>val[13:0] = 14'h34c6;</td>
</tr>
<tr>
<td>105</td>
<td>val[13:0] = 14'h353d;</td>
</tr>
<tr>
<td>106</td>
<td>val[13:0] = 14'h35b4;</td>
</tr>
<tr>
<td>107</td>
<td>val[13:0] = 14'h362c;</td>
</tr>
<tr>
<td>108</td>
<td>val[13:0] = 14'h36a5;</td>
</tr>
<tr>
<td>109</td>
<td>val[13:0] = 14'h371d;</td>
</tr>
</tbody>
</table>
I.4 Leading Zero Detector

Listing I.3: Coefficient LUT Example-C2-COS

I.4 Leading Zero Detector

// 47-bit Leading Zero Detector
module LZD_e (  
    reset,  
    clk,  
    val,  
    LZ_count,  
    scan_in0,  
    scan_en,  
    test_mode,  
    scan_out0  
);  

input  
    reset,  // system reset  
    clk;  // system clock  

input [46:0] val;
I.4 Leading Zero Detector

output reg [5:0] LZ_count;

input
scan_in0, // test scan mode data input
scan_en, // test scan mode enable
test_mode; // test mode select

output
scan_out0; // test scan mode data output

always @(val)
begin
  casez (val)
  47'b1?????????????????????????????????????????????: LZ_count = 0;
  47'b01???????????????????????????????????????????: LZ_count = 1;
  47'b001????????????????????????????????????????: LZ_count = 2;
  47'b0001???????????????????????????????????????: LZ_count = 3;
  47'b00001?????????????????????????????????????: LZ_count = 4;
  47'b000001????????????????????????????????????: LZ_count = 5;
  47'b0000001?????????????????????????????????: LZ_count = 6;
  47'b00000001????????????????????????????????: LZ_count = 7;
  47'b000000001?????????????????????????????: LZ_count = 8;
  47'b0000000001????????????????????????????: LZ_count = 9;
  47'b00000000001?????????????????????????: LZ_count = 10;
  47'b000000000001????????????????????????: LZ_count = 11;
  47'b0000000000001?????????????????????: LZ_count = 12;
  47'b00000000000001???????????????????: LZ_count = 13;
  47'b000000000000001?????????????????: LZ_count = 14;
  47'b0000000000000001???????????????: LZ_count = 15;
  47'b00000000000000001?????????????: LZ_count = 16;
  47'b000000000000000001???????????: LZ_count = 17;
  47'b0000000000000000001?????????: LZ_count = 18;
  47'b00000000000000000001?????????: LZ_count = 19;
  47'b000000000000000000001?????????: LZ_count = 20;
  47'b0000000000000000000001?????????: LZ_count = 21;
  47'b00000000000000000000001?????????: LZ_count = 22;
  47'b000000000000000000000001?????????: LZ_count = 23;
  47'b0000000000000000000000001?????????: LZ_count = 24;
  47'b00000000000000000000000001?????????: LZ_count = 25;
  47'b000000000000000000000000001?????????: LZ_count = 26;
  47'b0000000000000000000000000001?????????: LZ_count = 27;
  47'b00000000000000000000000000001?????????: LZ_count = 28;
  47'b000000000000000000000000000001?????????: LZ_count = 29;
  47'b0000000000000000000000000000001?????????: LZ_count = 30;
  47'b00000000000000000000000000000001?????????: LZ_count = 31;
  47'b000000000000000000000000000000001?????????: LZ_count = 32;
  47'b0000000000000000000000000000000001?????????: LZ_count = 33;
  47'b00000000000000000000000000000000001?????????: LZ_count = 34;
  47'b000000000000000000000000000000000001?????????: LZ_count = 35;
  47'b0000000000000000000000000000000000001?????????: LZ_count = 36;
  47'b00000000000000000000000000000000000001?????????: LZ_count = 37;
  47'b000000000000000000000000000000000000001?????????: LZ_count = 38;
I.5 LGBMGNG Test Vector Testbench

Listing I.4: Leading Zero Example-47-bit

```vhdl
module test;

  wire scan_out0;

  reg clk, reset;
  reg scan_in0, scan_en, test_mode;

  reg [31:0] seed_a,
             seed_b,
             seed_c;

  wire [23:0] n1,
              n2;

  wire valid;

  integer error_cnt;
  integer index;
  integer count;

  parameter pcount = 20000000;  // Need to change for different TV sizes manually

  reg [31:0] cfg [0:3];
  reg [31:0] memory_N1 [0:pcount-1];
  reg [31:0] memory_N2 [0:pcount-1];

endmodule
```

I.5 LGBMGNG Test Vector Testbench
reg [24:0] error_index [1:pcount]; // 2^25 -> Max ~20 million
reg [24:0] error_index_index;

LGBMGNG top(
  .reset(reset),
  .clk(clk),
  .seed_a (seed_a),
  .seed_b (seed_b),
  .seed_c (seed_c),
  .n1 (n1),
  .n2 (n2),
  .valid (valid),
  .scan_in0(scan_in0),
  .scan_en(scan_en),
  .test_mode(test_mode),
  .scan_out0(scan_out0)
);

initial begin
  $timeformat(-9,2,"ns", 16);
  `ifdef SDFSCAN
  $sdf_annotate("sdf/LGBMGNG_tsmc065_scan.sdf", test.top);
  `endif
  clk = 1'b0;
  reset = 1'b0;
  scan_in0 = 1'b0;
  scan_en = 1'b0;
  test_mode = 1'b0;

  // Uncomment option 1 for debug and option 2 for test_vectors
  // Comment the option task call that is not being used.
  // Option 1 is for singular a,b,c value debug
  // For Debug set the a,b,c values as the seeds and view first sample set
  seed_a = 1120679337;
  seed_b = 3622075866;
  seed_c = 3221225471;

  // debug; // Comment for TV and uncomment for Debug
  // Option 2 for Test Vector Test
  test_vector; // Comment for debug and uncomment for Test Vector support
end

always begin
  #1.25 clk = ~clk ; // 400MHz
end

task test_vector ;
  begin
```verbatim
$readmemh("TV_config.txt",cfg);
$readmemh("TV_in_N1.txt",memory_N1);
$readmemh("TV_in_N2.txt",memory_N2);

count = cfg[0];
seed_a = cfg[1];
seed_b = cfg[2];
seed_c = cfg[3];

error_cnt = 0;

$display("Count = %d
seed_a = %d
seed_b = %d
seed_c = %d",
        count,seed_a,seed_b,seed_c);

@negedge clk;
reset = 1'b1;
@negedge clk;
reset = 1'b0;
@negedge clk;

@posedge valid;
index = 0;
@posedge clk;
repeat (count)
  begin
    @posedge clk;
    if(memory_N1[index][23:0] != n1)
      begin
        $display("Error: Mismatch @ index = %d
mdl N1: %d
N1: %d",
                index,memory_N1[index][23:0],n1);
        error_cnt = error_cnt + 1;
      end
    if(memory_N2[index][23:0] != n2)
      begin
        $display("Error: Mismatch @ index = %d
mdl N2: %d
N2: %d",
                index,memory_N2[index][23:0],n2);
        error_cnt = error_cnt + 1;
      end
    index = index + 1;
    if (index%(pcount/100) == 0)
      begin
        $display("Progress: %d",index);
      end
  end
if(error_cnt == 0)
  begin
    $display("** Success **");
  end
else begin
  $display("** TEST FAILED **%d errors found\n",error_cnt);
end
 $display("\nCount = %d
seed_a = %d
seed_b = %d
seed_c = %d",
        count,seed_a,seed_b,seed_c);
$finish;
```
end
endtask
task debug ;
begin
@($negedge clk);
reset = 1'b1;
@($negedge clk);
reset = 1'b0;
@($negedge clk);
repeat (30)
@($posedge clk)
begin
$display("N1: %d\nN2: %d\n",n1,n2);
end
$finish;
end
dmodule

Listing I.5: LGBMNG, 24-bit Box-Muller Implementation Testbench