Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

8-2017

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: https://repository.rit.edu/theses

Recommended Citation
Glauser, Lincoln, "The Design of a 24-bit Hardware Gaussian Noise Generator via the Box-Muller Method and its Error Analysis" (2017). Thesis. Rochester Institute of Technology. Accessed from

This Master's Project is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@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.

Forward

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 9.42σ, 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

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Gaussian noise, which represents a standard normal distribution, is a natural phenomenon that directly effects 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.

$$f(x;a,b) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & otherwise \end{cases} \tag{2.1}$$

$$F(x;a,b) = \begin{cases} 0 & x < A \\ \frac{x-a}{b-a} & a \leq x < b \\ 1 & x \geq b \end{cases} \tag{2.2}$$

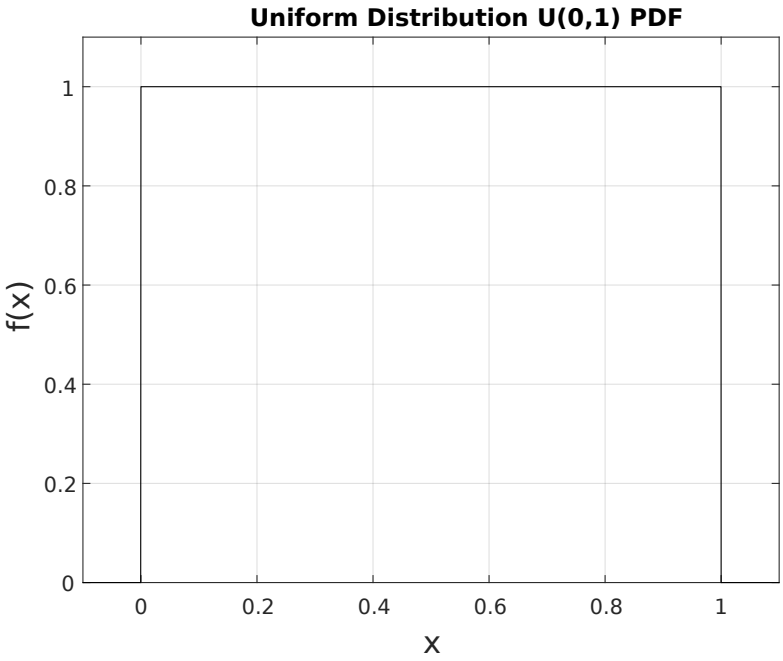

Figure 2.1: Standard Uniform Distribution U(0,1) PDF

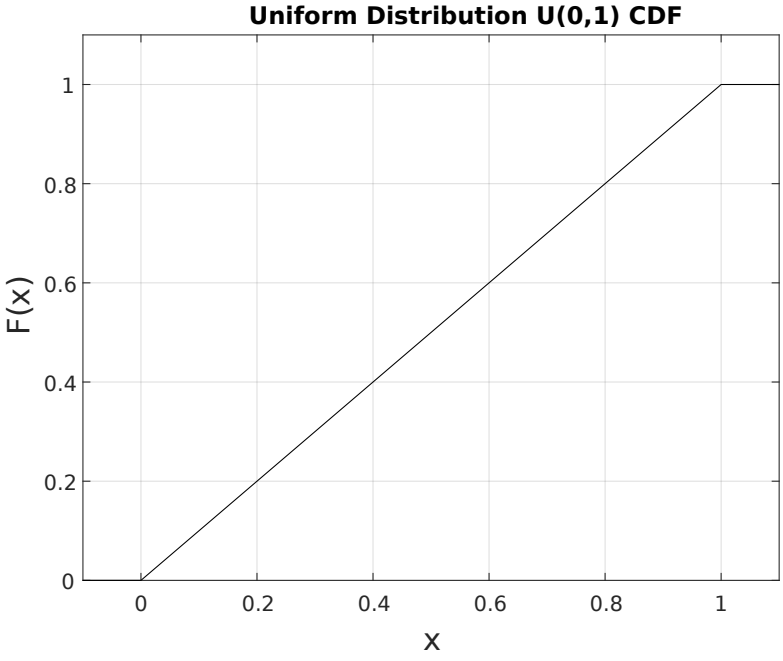

Figure 2.2: Standard Uniform Distribution U(0,1) CDF

2.1.2 Normal Distribution Theory

A normal distribution of continuous random variable X with mean μ and variance σ 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 μ of the value 0 and standard deviation σ 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.

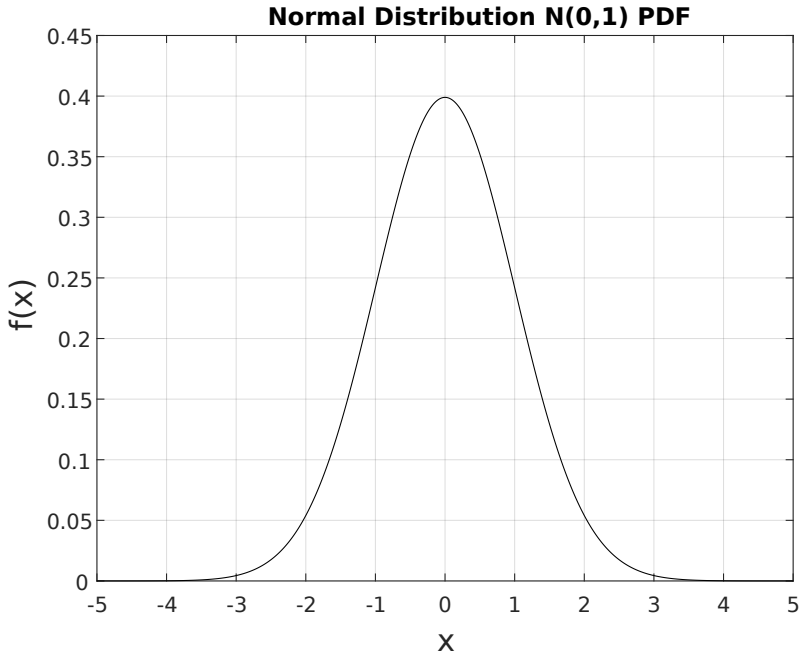

Figure 2.3: Standard Normal Distribution N(0,1) PDF

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

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \qquad -\infty < x < \infty \tag{2.4}$$

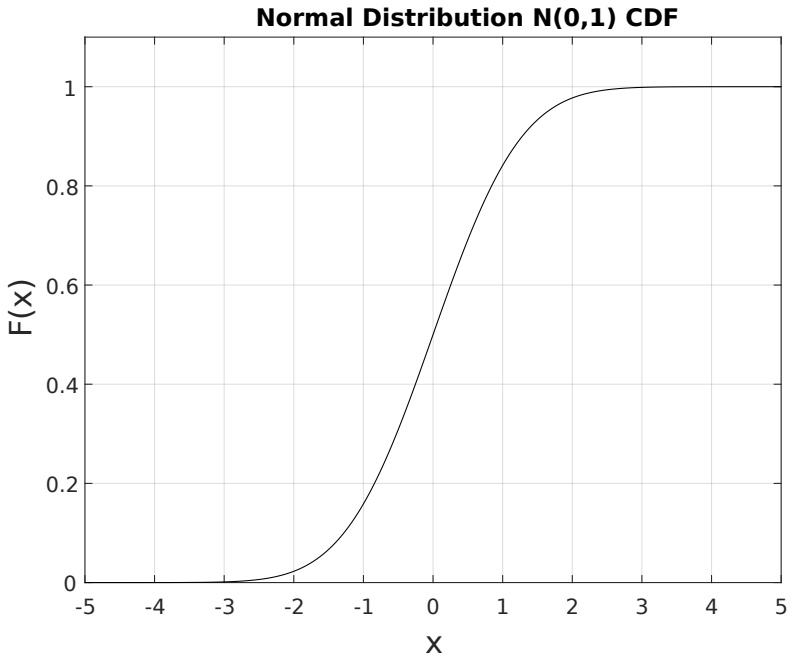

Figure 2.4: Standard Normal Distribution N(0,1) CDF

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

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.

$$R^2 = \frac{\sum(y_{fit} - \bar{y})^2}{\sum(y - \bar{y})^2} \tag{2.6}$$

$$SEE = \sqrt{\frac{\sum(\hat{y} - y)^2}{n - 2}} \tag{2.7}$$

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_l 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,

there is the following form:

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

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 $[x_{min}, x_{max})$.

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:

$$x = x_m + x_l \cdot 2^{-k} \tag{2.9}$$

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.

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

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

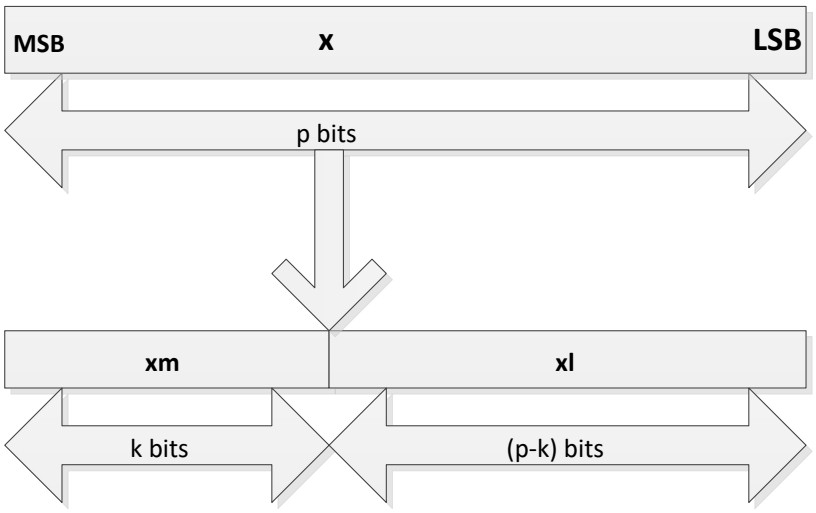

Figure 2.5: Splitting of x for Function Approximation

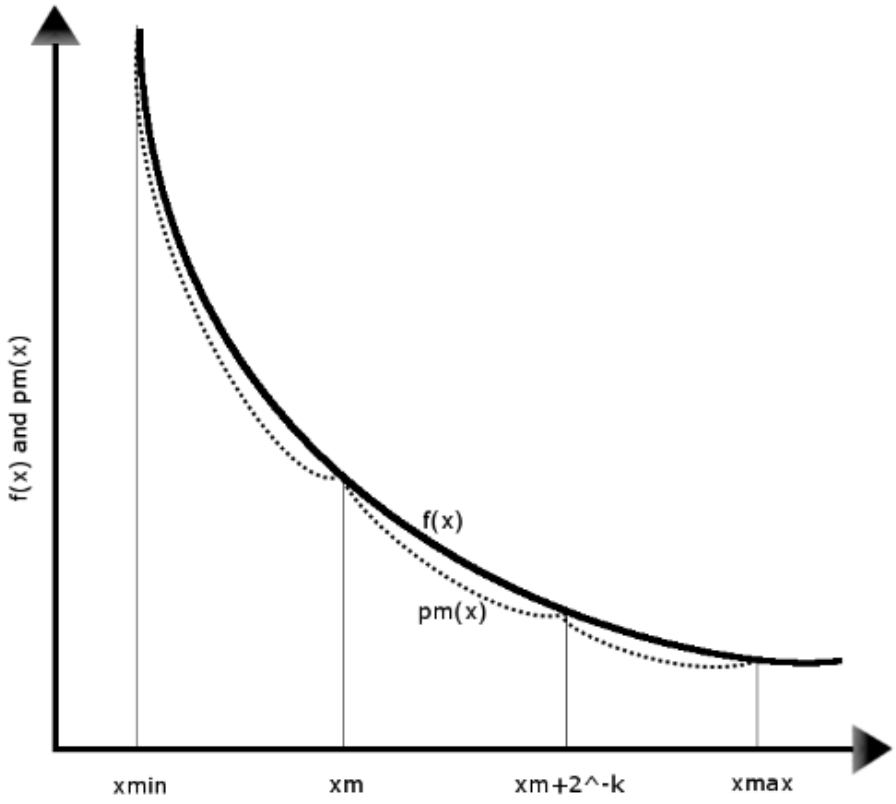

Figure 2.6: Function Approximation on Individual Segments

as the ith Chebyshev node for [-1,1).

$$t_i = cos(\frac{(2 \cdot i + 1)\pi}{2 \cdot n}) \qquad (0 \leq i < n) \tag{2.11}$$

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.

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

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.

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

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

$$y_i = f(x_i) \tag{2.15}$$

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:

$$C_1 = \frac{f(x_0)}{(x_0-x_1)} + \frac{f(x_1)}{(x_1-x_0)} \tag{2.16}$$

$$C_0 = \frac{-f(x_0) \cdot x_1}{(x_0 - x_1)} + \frac{-f(x_1) \cdot x_0}{(x_1 - x_0)} \tag{2.17}$$

$$p_m(x) = C_0 + C_1 \cdot x \tag{2.18}$$

(b) Degree Two Approximation:

$$C_2 = \frac{f(x_0)}{(x_0 - x_1)(x_0 - x_2)} + \frac{f(x_1)}{(x_1 - x_0)(x_1 - x_2)} + \frac{f(x_2)}{(x_2 - x_0)(x_2 - x_1)} \tag{2.19}$$

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

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

$$p_m(x) = C_0 + C_1 \cdot x + C_2 \cdot x^2 \tag{2.22}$$

(c) Degree Three Approximation:

$$C_3 = \frac{f(x_0)}{(x_0 - x_1)(x_0 - x_2)(x_0 - x_3)} + \frac{f(x_1)}{(x_1 - x_0)(x_1 - x_2)(x_1 - x_3)} + \frac{f(x_2)}{(x_2 - x_0)(x_2 - x_1)(x_2 - x_3)} + \frac{f(x_3)}{(x_3 - x_0)(x_3 - x_1)(x_3 - x_2)} \tag{2.23}$$

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

$$C_1 = \frac{f(x_0)\cdot(x_1\cdot(x_2+x_3)+x_2\cdot x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} + \frac{f(x_1)\cdot(x_0\cdot(x_2+x_3)+x_2\cdot x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} + \frac{f(x_2)\cdot(x_0\cdot(x_1+x_3)+x_1\cdot x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)} + \frac{f(x_3)\cdot(x_0\cdot(x_1+x_2)+x_1\cdot x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)} \tag{2.25}$$

$$C_0 = \frac{f(x_0)\cdot(-x_1\cdot x_2\cdot x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} + \frac{f(x_1)\cdot(-x_0\cdot x_2\cdot x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} + \frac{f(x_2)\cdot(-x_0\cdot x_1\cdot x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)} + \frac{f(x_3)\cdot(-x_0\cdot x_1\cdot x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)} \tag{2.26}$$

$$p_m(x) = C_0 + C_1\cdot x + C_2\cdot x^2 + C_3\cdot x^3 \tag{2.27}$$

5. The coefficients $C_i(x_m)$ have values for the input x and then are transformed for having values for the input x_l. 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 ξ is the point on the interval [a,b) where the nth derivative of the original function, $f(x)$, has

its maximum value.

$$E_n(x) = (\frac{b-a}{4})^n \cdot \frac{2 \cdot |f^n(\xi)|}{n!} \qquad a \leq \xi < b \tag{2.28}$$

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_{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.

$$sin\left((Q+D)\frac{\pi}{2}\right) = \left\{\begin{array}{ll} sin(D\frac{\pi}{2}) & if\ Q\ mod\ 4 = 0 \\ cos(D\frac{\pi}{2}) & if\ Q\ mod\ 4 = 1 \\ -sin(D\frac{\pi}{2}) & if\ Q\ mod\ 4 = 2 \\ -cos(D\frac{\pi}{2}) & if\ Q\ mod\ 4 = 3 \end{array}\right\} \tag{2.29}$$

$$cos\left((Q+D)\frac{\pi}{2}\right) = \left\{\begin{array}{ll} cos(D\frac{\pi}{2}) & if\ Q\ mod\ 4 = 0 \\ -sin(D\frac{\pi}{2}) & if\ Q\ mod\ 4 = 1 \\ -cos(D\frac{\pi}{2}) & if\ Q\ mod\ 4 = 2 \\ sin(D\frac{\pi}{2}) & if\ Q\ mod\ 4 = 3 \end{array}\right\} \tag{2.30}$$

$$\sqrt{M_x \cdot 2^{E_x}} = \left\{ \begin{array}{ll} 2^{\frac{E_x}{2}}\sqrt{M_x} & if\ E_x\ mod\ 1 = 0 \\ 2^{\frac{E_x+1}{2}}\sqrt{\frac{M_x}{2}} & if\ E_x\ mod\ 1 = 1 \end{array} \right\} \tag{2.31}$$

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

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 $cos\left(D\frac{\pi}{2}\right)$) or $1-D$ (approximating $sin\left(D\frac{\pi}{2}\right)$) 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 x [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 [0,1) as per 2.34:

$$\bar{C_2} = C_2 \cdot 2^{-2k} \tag{2.37}$$

$$\bar{C_1} = (2 \cdot C_2 \cdot x_m + C_1) \cdot 2^{-k} \tag{2.38}$$

$$\bar{C_0} = C_2 \cdot x_m^2 + C_1 \cdot x_m + C_0 \tag{2.39}$$

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).

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

$$x_l = (x - 1 - x_m) \cdot 2^k \tag{2.41}$$

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

$$\bar{C_1} = C_1 \cdot 2^{-k} \tag{2.42}$$

$$\bar{C_0} = C_1 \cdot (x_m + 1) + C_0 \tag{2.43}$$

Degree Two Coefficient Transformations from x to x_l for x [1,2) as per 2.41:

$$\bar{C_2} = C_2 \cdot 2^{-2k} \tag{2.44}$$

$$\bar{C_1} = (C_2 \cdot (2 + 2 \cdot x_m) + C_1) \cdot 2^{-k} \tag{2.45}$$

$$\bar{C_0} = C_2 \cdot \left(x_m^2 + 2 \cdot x_m + 1\right) + C_1 \cdot (x_m + 1) + C_0 \tag{2.46}$$

Degree Three Coefficient Transformations from x to x_l for x [1,2) as per 2.41:

$$\bar{C_3} = C_3 \cdot 2^{-3k} \tag{2.47}$$

$$\bar{C_2} = (3 \cdot C_3 \cdot x_m + 3 \cdot C_3 + C_2) \cdot 2^{-2k} \tag{2.48}$$

$$\bar{C_1} = \left(3 \cdot C_3 \cdot x_m^2 + 2 \cdot (3 \cdot C_3 + C_2) \cdot x_m + 3 \cdot C_3 + 2 \cdot C_2 + C_1\right) \cdot 2^{-k} \tag{2.49}$$

$$\bar{C_0} = C_3 \cdot x_m^3 + (3 \cdot C_3 + 2 \cdot C_2) \cdot x_m^2 + (3 \cdot C_3 + 2 \cdot C_2 + C_1) \cdot x_m + C_3 + C_2 + C_1 + C_0 \tag{2.50}$$

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.

$$x = 2 \cdot \left(1 + x_m + x_l \cdot 2^{-k}\right) = 2 + 2 \cdot x_m + xl \cdot 2^{-k+1} \tag{2.51}$$

$$x_l = (\frac{x}{2} - 1 - x_m) \cdot 2^k \tag{2.52}$$

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

$$\bar{C_1} = C_1 \cdot 2^{-k+1} \tag{2.53}$$

$$\bar{C_0} = C_1 \cdot (2 + 2 \cdot x_m) + C_0 \tag{2.54}$$

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 $cos(D\frac{\pi}{2})$ or $cos((1-D)\frac{\pi}{2})$ are selected as per 2.29 and 2.30. For LN, as per 2.32, $E_x \cdot ln(2)$ is evaluated and joined with the approximation result to obtain the overall result. For SQRT, as per 2.31, $2^{E_x/2}$ (E_x is even) or $2^{(E_x+1)/2}$ (E_x 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) --------------

% Range Reduction
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 + xm_e + xl_e * 2^-k_e
% example is Degree-1
y_e = C1_e(xm_e_index)*xl_e + C0_e(xm_e_index);

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

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

% Range Reduction
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');

```

```
30 % Approximate sqrt (x_f) where x_f = [1,4)
31 % x_f is [1,2) --> x_f = 1 + xm_f + xl_f * 2^-k_f
32 % or
33 % x_f is [2,4) --> x_f = 2 + 2*xm_f + xl_f * 2^(-k_f+1)
34 % Degree-(n-1) piecewise polynomial - 2^k_f segments
35 % example is Degree-1
36 y_f = C1_f(xm_f_index)*xl_f + C0_f(xm_f_index);
37
38 % Range Reconstruction
39 exp_f' = if(exp_f[0], exp_f+1>>1, exp>>1);
40 f = y_f << exp_f';
41
42 % %------------ Evaluate g0=sin(2*pi*u1) ------------
43 % %------------                 g1=cos(2*pi*u1) ------------
44
45 % Range Reduction
46 quadrant = u1[MSB:MSB-1];
47 x_g_a = u1[MSB-2:0];
48 x_g_b = (1-2^-(MSB-1))-u1[MSB-2:0];
49
50 % Approximate cos(x_g_a*pi/2) and cos(x_g_b*pi/2)
51 % where x_g_a, x_g_b = [0,1-2^-(MSB-1)]
52 % Degree-(n-1) piecewise polynomial - 2^k_g segments
53 % x_g is [0,1) --> x_g = xm_g + xl_g * 2^-k_g
54 % example is Degree-2
55 y_g_a = C2_g(xm_g_a_index)*xl_g_a^2 + C1_g(xm_g_a_index)*xl_g_a + C0_g(xm_g_a_index);
56 y_g_b = C2_g(xm_g_b_index)*xl_g_b^2 + C1_g(xm_g_b_index)*xl_g_b + C0_g(xm_g_b_index);
57
58 % Range Reconstruction
59 switch(quadrant)
60   case 0: g0 =  y_g_b;    g1 =  y_g_a; % [0, pi/2)
61   case 1: g0 =  y_g_a;    g1 = -y_g_b; % [pi/2, pi)
62   case 2: g0 = -y_g_b;    g1 = -y_g_a; % [pi, 3*pi/2)
63   case 3: g0 = -y_g_a;    g1 =  y_g_b; % [3*pi/2, 2*pi)
64
65 %%--------------- Compute x0 and x1 ---------------
66 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 quanitization 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_{approx} term in each corresponding E_y which represents the error created by the Chebyshev series approximation before any quantization is executed.

$$\begin{aligned} \tilde{x} &= x + 2^{-FB_{\tilde{x}}-1} \\ E_{\tilde{x}} &= 2^{-FB_{\tilde{x}}-1} \end{aligned} \tag{2.55}$$

$$\begin{aligned} \tilde{z} = \tilde{x} \pm \tilde{y} = x \pm y = E_{\tilde{x}} \pm E_{\tilde{y}} + 2^{-FB_{\tilde{z}}-1} \\ E_{\tilde{z}} = E_{\tilde{x}} + E_{\tilde{y}} + 2^{-FB_{\tilde{z}}-1} \end{aligned} \tag{2.56}$$

$$\begin{aligned} \tilde{z} = \tilde{x} \cdot \tilde{y} = x \cdot y + x \cdot E_{\tilde{y}} + y \cdot E_{\tilde{x}} + E_{\tilde{x}} \cdot E_{\tilde{y}} + 2^{-FB_{\tilde{z}}-1} \\ E_{\tilde{z}} = max(x) \cdot E_{\tilde{y}} + max(y) \cdot E_{\tilde{x}} + E_{\tilde{x}} \cdot E_{\tilde{y}} + 2^{-FB_{\tilde{z}}-1} \end{aligned} \tag{2.57}$$

Degree One Approximation MiniBit Approach:

$$\tilde{y} = \tilde{C_1} \cdot x_l + C_0 \tag{2.58}$$

$$\begin{aligned} \tilde{D_0} &= \tilde{C_1} \cdot \tilde{x_l} \\ \tilde{y} &= \tilde{D_0} + \tilde{C_0} \end{aligned} \tag{2.59}$$

$$E_{C_1} = 2^{-FB_{C_1}} \quad E_{C_0} = 2^{-FB_{C_0}} \tag{2.60}$$

$$\begin{aligned} E_{D_0} &= max(\tilde{x_l}) \cdot E_{C_1} + max(\tilde{C_1}) \cdot E_{x_l} + 2^{-FB_{D_0}-1} \\ E_y &= E_{D_0} + E_{C_0} + 2^{-FB_y-1} \end{aligned} \tag{2.61}$$

Degree Two Approximation MiniBit Approach:

$$y = (C_2 \cdot x_l + C_1) \cdot x_l + C_0 \tag{2.62}$$

$$\begin{aligned} \tilde{D_2} &= \tilde{C_2} \cdot \tilde{x_l} \\ \tilde{D_1} &= \tilde{D_2} + \tilde{C_1} \\ \tilde{D_0} &= \tilde{D_1} \cdot \tilde{x_l} \\ \tilde{y} &= \tilde{D_0} + \tilde{C_0} \end{aligned} \tag{2.63}$$

$$E_{C_2} = 2^{-FB_{C_2}} \quad E_{C_1} = 2^{-FB_{C_1}} \quad E_{C_0} = 2^{-FB_{C_0}} \tag{2.64}$$

$$\begin{aligned} E_{D_2} &= max(\tilde{x_l}) \cdot E_{C_2} + max(\tilde{C_2}) \cdot E_{x_l} + 2^{-FB_{D_2}-1} \\ E_{D_1} &= E_{D_2} + E_{C_1} + 2^{-FB_{D_1}-1} \\ E_{D_0} &= max(\tilde{x_l}) \cdot E_{D_1} + max(\tilde{D_1}) \cdot E_{x_l} + 2^{-FB_{D_0}-1} \\ max(\tilde{D_1}) &= max(\tilde{C_2}) \cdot max(\tilde{x_l}) + max(\tilde{C_1}) \\ E_y &= E_{D_0} + E_{C_0} + 2^{-FB_y-1} + E_{approx} \end{aligned} \tag{2.65}$$

Degree Three Approximation MiniBit Approach:

$$y = ((C_3 \cdot x_l + C_2) \cdot x_l + C_1) \cdot x_l + C_0 \tag{2.66}$$

$$\begin{aligned}
\tilde{D_4} &= \tilde{C_3} \cdot \tilde{x_l} \\
\tilde{D_3} &= \tilde{D_4} + \tilde{C_2} \\
\tilde{D_2} &= \tilde{D_3} \cdot \hat{x_l} \\
\tilde{D_1} &= \tilde{D_2} + \tilde{C_1} \\
\tilde{D_0} &= \tilde{D_1} \cdot \tilde{x_l} \\
\tilde{y} &= \tilde{D_0} + \tilde{C_0}
\end{aligned} \tag{2.67}$$

$$E_{C_3} = 2^{-FB_{C_3}} \quad E_{C_2} = 2^{-FB_{C_2}} \quad E_{C_1} = 2^{-FB_{C_1}} \quad E_{C_0} = 2^{-FB_{C_0}} \tag{2.68}$$

$$\begin{aligned}
E_{D_4} &= max(\tilde{x_l}) \cdot E_{C_3} + max(\tilde{C_3}) \cdot E_{x_l} + 2^{-FB_{D_4}-1} \\
E_{D_3} &= E_{D_4} + E_{C_2} + 2^{-FB_{D_3}-1} \\
E_{D_2} &= max(\tilde{x_l}) \cdot E_{D_3} + max(\tilde{D_3}) \cdot E_{x_l} + 2^{-FB_{D_2}-1} \\
max(\tilde{D_3}) &= max(\tilde{C_3}) \cdot max(\tilde{x_l}) + max(\tilde{C_2}) \\
E_{D_1} &= E_{D_2} + E_{C_1} + 2^{-FB_{D_1}-1} \\
E_{D_0} &= max(\tilde{x_l}) \cdot E_{D_1} + max(\tilde{D_1}) \cdot E_{x_l} + 2^{-FB_{D_0}-1} \\
max(\tilde{D_1}) &= max(\tilde{D_3}) \cdot max(\tilde{x_l}) + max(\tilde{C_1}) \\
E_y &= E_{D_0} + E_{C_0} + 2^{-FB_y-1} + E_{approx}
\end{aligned} \tag{2.69}$$

Once the corresponding values that are unique to each analysis (eg. $max(C_i)$, E_{x_l}, E_{approx}, ...) 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,

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.

$$E_y \leq 2^{-FB_y} \tag{2.70}$$

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_l 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 σ away from the mean μ. 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 6.47σ [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 6.47σ, 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.

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

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.

$$x_0 = f \cdot g_0 \quad and \quad x_1 = f \cdot g_1 \tag{2.72}$$

$$E_{x_0} \leq 2^{-FB_{x_0}} \quad and \quad E_{x_1} \leq 2^{-FB_{x_1}} \tag{2.73}$$

$$2^{-FB_{x_0}} \geq max(g_0) \cdot 2^{-FB_f} + max(f) \cdot 2^{-FB_{g_0}} \tag{2.74}$$

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.

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

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, $FB_{y_{g_a}} = FB_{y_{g_b}}$. 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 $FB_{y_{g_a}} = FB_{y_{g_b}} = FB_{g_0} = FB_{g_1}$.

7b. This step focuses on finding the internal fractional bit widths within the function approximation step now that $FB_{y_{g_a}}$ and $FB_{y_{g_b}}$ are known. In regards to the error analysis, y_{g_a} and y_{g_b} are equivalent, so only y_{g_a} 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_l as the input by the equations starting at 2.33. There are then two methods for finding the error due to approximation E_{approx_g}.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_g}. 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 $max(x_{l_g}) = 1$ and $E_{x_{l_g}} = 0$. 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 $FB_{C_{i_g}}$ and internal arithmetic signals $FB_{D_{i_g}}$.

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.

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

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_f is known and to solve for the approximation output fractional bit width FB_{y_f}. 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

$IBe + FBe - 1$.

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

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_{y_f}. 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_{y_f} 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.

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

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

$$E_f = E_{y_f} \cdot 2^{exp_f^{'}} \tag{2.80}$$

$$E_{y_f} \leq 2^{-FB_f - exp_f^{'}} \tag{2.81}$$

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.

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

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_{l_f} is the actual value utilized in the function approximation arithmetic and therefore the value of $E_{x_{l_f}}$ is needed. For the arithmetic, x_{l_f} 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_{m_f}, which is equal to k_f and leads to 2.85.

$$E'_{x_f} = E_e = 2^{-exp_f} \tag{2.83}$$

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

$$E_{x_{l_f}} = \begin{cases} 2^{-FB_e-exp_f+k_f} & exp_f[0] = 0 \\ 2^{-FB_e-exp_f-1+k_f} & exp_f[0] = 1 \end{cases} \tag{2.85}$$

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 $E_{x_{l_f}}$ 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 $C_{1_f} \cdot E_{x_{l_f}}$ within 2.86 to be the dominating error within the error constraint equation [1]. With this assumption, a straight forward analysis to find FB_e can be executed at the minimum value of exp_f, which causes the most error in $E_{x_{l_f}}$. 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.

$$E_{y_f} = C_{1_f} \cdot E_{x_{l_f}} + 2^{-FBC_{1_f}-1} + 2^{-FBC_{1_f}-1} \cdot E_{x_{l_f}} + 2^{-FBC_{0_f}} + 2^{-FB_{y_f}} + E_{approx_f} \quad (2.86)$$

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

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

The even case is considered first, where FB_e is then odd in value. With the value

of $max\left(C_{1_f}\right)$ 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 FB_e value that is found is rounded up to the nearest even integer value. The value that is less in magnitude is then chosen for FB_e.

8d. Now that FB_e 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 $FB_{C_{i_f}}$ and the fractional bit widths of the internal arithmetic signals $FB_{D_{i_f}}$ 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).

$$e = -2 \cdot ln(u_0) \tag{2.89}$$

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 FB_{ln2}, $FB_{e'}$ and FB_{y_e} after knowing the value of FB_e. 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.

$$e' = exp_e \cdot ln2 \tag{2.90}$$

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

$$E_{e'} = max(exp_e) \cdot 2^{-FB_{ln2}-1} + max(ln2) \cdot E_{exp_e} + 2^{-FB_{e'}-1} \tag{2.92}$$

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

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_{ln2}.

$$E_{e'} = TB_{u_0} \cdot 2^{-FB_{ln2}-1} + 2^{-FB_{e'}-1} \tag{2.94}$$

$$E_e = 2^{-FB_{y_e}+1} + TB_{u_0} \cdot 2^{-FB_{ln2}} + 2^{-FB_{e'}} + 2^{-FB_e-1} \tag{2.95}$$

$$2^{-FB_e-2} \geq 2^{-FB_{y_e}} + TB_{u_0} \cdot 2^{-FB_{ln2}-1} + 2^{-FB_{e'}-1} \tag{2.96}$$

9b. Now that FB_{y_e} 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 is [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}, 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 8.03σ, which is then rounded to designing for 8.1σ. 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 8.2σ. 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 8.2σ ($IB_{x_0} = IB_{x_1} = 4$) 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

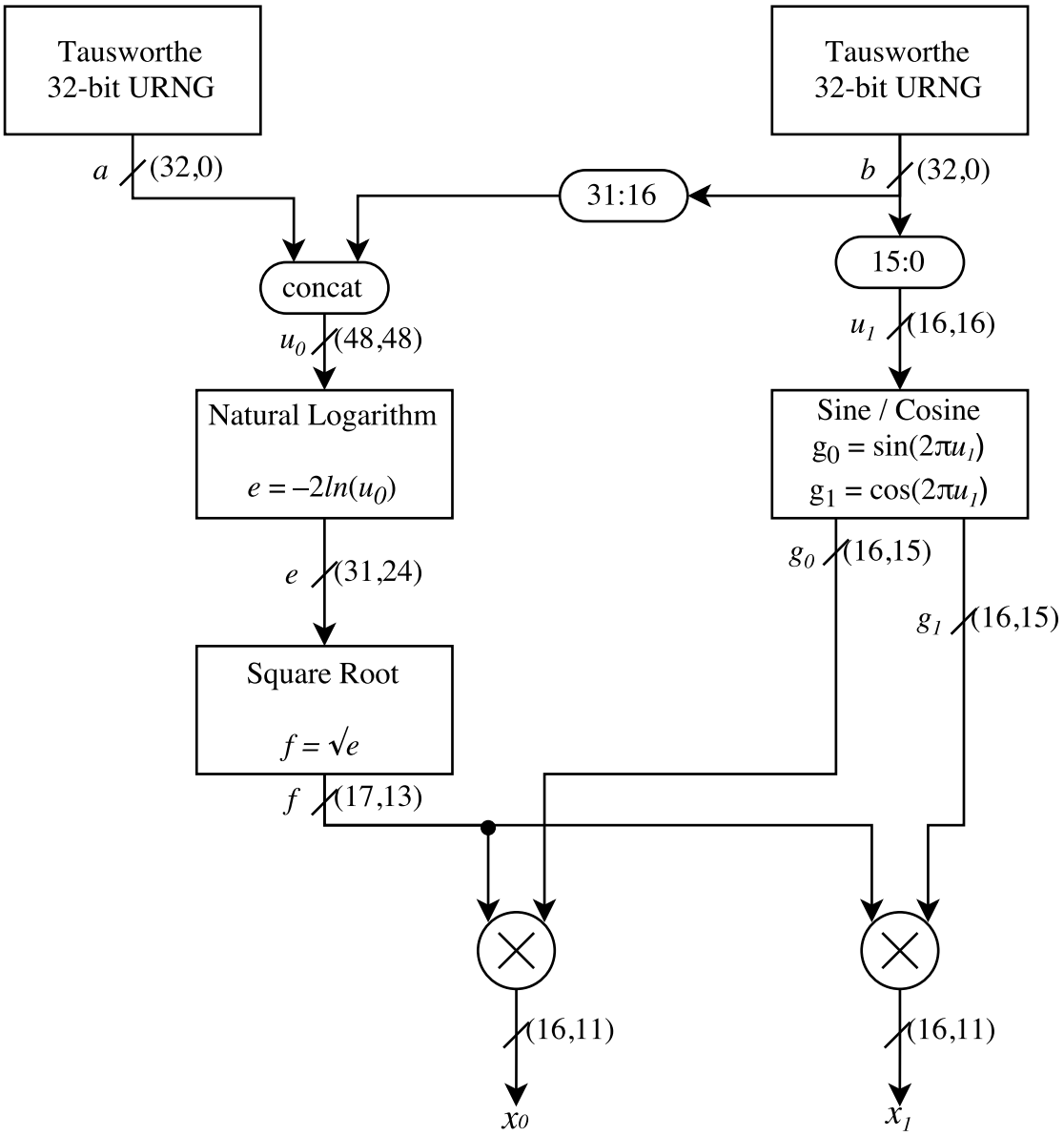

Figure 2.7: Lee *et al.*'s 16-bit Noise Block Diagram [1]

leads to $FB_f = 13$ and $FB_{g_0} = FB_{g_1} = 15$.

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

```
1 #16-bit Box-Muller Method Pseudo Code
```

```
2  --------------- Generate u0 and u1 ---------------
3
4  a = taus(); b = taus();
5  u0 = concat(a,b[31:16]);
6  u1 = b[15:0];
7
8  ------------- Evaluate e = -2ln(u0) --------------
9
10 # Range Reduction
11 exp_e = LeadingZeroDetector(u0) + 1;
12 x_e = u0 << exp_e;
13
14 # Approximate -ln(x_e) where x_e = [1,2)
15 # Degree-2 piecewise polynomial
16 y_e = ((C2_e[xm_e]*xl_e)+C1_e[xm_e])*xl_e
17         +C0_e[xm_e];
18
19 # Range Reconstruction
20 ln2 = ln(2);
21 e' = exp_e*ln2;
22 e = (e'-y_e)<<1;
23
24 --------------- Evaluate f = sqrt (e) ------------
25
26 # Range Reduction
27 exp_f = 5-Leading ZeroDetector(e);
28 x_f' = e >> exp_f;
29 x_f = if(exp_f[0], x_f'>>1, x_f');
30
31 # Approximate sqrt (x_f) where x_f = [1,4)
32 # Degree-1 piecewise polynomial
33 y_f = C1_f[xm_f]*xl_f+C0_f[xm_f];
34
35 # Range Reconstruction
36 exp_f' = if(exp_f[0], exp_f+1>>1, exp>>1);
37 f = y_f << exp_f';
38
39 ------------ Evaluate g0=sin(2*pi*u1) ------------
40 ------------          g1=cos(2*pi*u1) ------------
41
42 # Range Reduction
43 quadrant = u1[15:14];
44 x_g_a = u1[13:0];
45 x_g_b = (1-2^-14)-u1[13:0];
46
47 # Approximate cos(x_g_a*pi/2) and cos(x_g_b*pi/2)
48 # where x_g_a, x_g_b = [0,1-2^-14]
49 # Degree-1 piecewise polynomial
50 y_g_a = C1_g[xm_g_a]*xl_g_a+C0_g[xm_g_a];
51 y_g_b = C1_g[xm_g_b]*xl_g_b+C0_g[xm_g_b];
52
53 # 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 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 $FB_{y_{g_a}} = FB_{y_{g_b}} = FB_{g_0} = FB_{g_1} = 15$. 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, $E_{approx_g} = 9.41 \cdot 10^{-6}$ and $max(C_{1_g}) = 0.123$. With these values and knowing $E_{x_{l_g}} = 0$ 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, $max(C_{1_g}) = 0.123$, $max(C_{0_g}) = 1.000009$, 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 $Offset = 5$. With the value of $Offset$, $max(exp_f) = 5$ and $max(exp'_f)$ becomes 2 and 3, respectively, for exp_f being even and odd. FB_{y_f} is then found to be 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 $C_{1_{f_{odd}}}$, $C_{0_{f_{odd}}}$, $C_{1_{feven}}$ and $C_{0_{feven}}$ and internal arithmetic signal D_{0_f}. From the approximation, $E_{approx_{f_{odd}}} = 3.785147 \cdot 10^{-6}$, $E_{approx_{feven}} = 5.353006 \cdot 10^{-6}$, $max(C_{1_{f_{odd}}}) = 0.00778$ and $max(C_{1_{feven}}) = 0.011$. 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, $E_{x_{l_f}}$, while taking into consideration the odd values of the constants used (eg. E_{approx_f}). This results in two scenarios, where the values for $FB_{c_{1_f}}$, $FB_{C_{0_f}}$ and $FB_{D_{0_f}}$, 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 $FB_{C_{1_f}} = 18$, $FB_{C_{0_f}} = 19$ and $FB_{D_{0_f}} = 19$ values are chosen. For 8e, C_{1_f} has 6 redundant bits, C_{0_f} 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_{1_f} and C_{0_f}, 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_{2_e}, C_{1_e}, C_{0_e} and internal arithmetic signals D_{2_e}, D_{1_e} and D_{0_e}. From the approximation, $E_{approx_e} = 6.1832 \cdot 10^{-10}$, $max(C_{2_e}) = 7.59969 \cdot 10^{-6}$, $max(C_{1_e}) = 0.003906$ and $max(C_{0_e}) = 0.69119$. For 9c, the MiniBit approach equations for degree two are utilized and the values that result from optimization are $FB_{C_{2_e}} = FB_{C_{1_e}} = FB_{C_{0_e}} = FB_{D_{2_e}} = FB_{D_{1_e}} = FB_{D_{0_e}} = 30$. For 9d, C_{2_e} and C_{1_e}, respectively, have 17 and 8 redundant bits and neither of them require integer bits. In addition to this, all C_{2_e} 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 9.42σ ($IB_{n_1} = IB_{n_2} = 4$) 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, $max(g_0) = max(g_1) = 1$ due to 1 being the maximum value for sine and cosine. In addition to this, $max(f)$ is found by passing in the minimum value of u_0, 2^{-64} into $f = \sqrt{-2 \cdot ln(u_0)}$, which leads to $max(f) = 9.41928$. 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 ---------------
a = taus(); b = taus(); c = taus();% all 32-bit-------
u0 = concat(a,b);
u1 = c;

%%------------- 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-3 piecewise polynomial - 512 segments
% C3,C2,C1,C0 = 19,28,38,47 bits.
% x_e is [1,2) --> x_e = 1 + xm_e + xl_e * 2^-k_e
y_e = (((C3_e(xm_e_index)*xl_e+C2_e(xm_e_index))*xl_e)+C1_e(xm_e_index))*xl_e +
↪  C0_e(xm_e_index);

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

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

% Range Reduction
exp_f = 5-LeadingZeroDetector(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)
% x_f is [1,2) --> x_f = 1 + xm_f + xl_f * 2^-k_f
% or
% x_f is [2,4) --> x_f = 2 + 2*xm_f + xl_f * 2^(-k_f+1)
% Degree-1 piecewise polynomial - 1024 segments
% C1,C0 = 16,26 bits.
y_f = C1_f(xm_f_index)*xl_f + C0_f(xm_f_index);

% 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) ------------

% Range Reduction
quadrant = u1[31:30];
x_g_a = u1[29:0];
x_g_b = (1-2^-30)-u1[29:0];

% Approximate cos(x_g_a*pi/2) and cos(x_g_b*pi/2)
```

```
52 % where x_g_a, x_g_b = [0,1-2^-30]
53 % Degree-2 piecewise polynomial - 128 segments
54 % C2,C1,C0 = 14,21,28 bits.
55 % x_g is [0,1) --> x_g = xm_g + xl_g * 2^-k_g
56 y_g_a = C2_g(xm_g_a_index)*xl_g_a^2 + C1_g(xm_g_a_index)*xl_g_a + C0_g(xm_g_a_index);
57 y_g_b = C2_g(xm_g_b_index)*xl_g_b^2 + C1_g(xm_g_b_index)*xl_g_b + C0_g(xm_g_b_index);
58
59 % Range Reconstruction
60 switch(quadrant)
61   case 0: g0 =  y_g_b;    g1 =  y_g_a; % [0, pi/2)
62   case 1: g0 =  y_g_a;    g1 = -y_g_b; % [pi/2, pi)
63   case 2: g0 = -y_g_b;    g1 = -y_g_a; % [pi, 3*pi/2)
64   case 3: g0 = -y_g_a;    g1 =  y_g_b; % [3*pi/2, 2*pi)
65
66
67 %%--------------- Compute n1 and n2 ---------------
68 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, $FB_{y_{g_a}} = FB_{y_{g_b}} = FB_{g_0} = FB_{g_1} = 24$. 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 ($n = 3$ and $k_g = 7$) with coefficient values C_{2_g}, C_{1_g} and C_{0_g} and internal arithmetic signals D_{2_g}, D_{1_g} and D_{0_g}. From the approximation, $E_{approx_g} = 9.6254 \cdot 10^{-9}$, $max(C_{2_g}) = 7.52975 \cdot 2^{-5}$, $max(C_{1_g}) = 0.01227$ and $max(C_{0_g}) = 1.00000000004$. Then, the MiniBit approach referenced in 7b is executed with the degree two equations and results in $FB_{C_{2_g}} = 27$, $FB_{C_{1_g}} = 27$, $FB_{C_{0_g}} = 27$, $FB_{D_{2_g}} = 27$, $FB_{D_{1_g}} = 28$ and $FB_{D_{0_g}} = 28$. For 7c, $C_{2_g}$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} 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 $C_{1_{f_{odd}}}$, $C_{0_{f_{odd}}}$, $C_{1_{feven}}$ and $C_{0_{feven}}$ and internal arithmetic signal D_{0_f}. From the approximation, $E_{approx_{f_{odd}}} = 2.1063 \cdot 10^{-8}$, $E_{approx_{feven}} = 1.4894 \cdot 10^{-8}$, $max(C_{1_{f_{odd}}}) = 6.90365 \cdot 10^{-4}$, $max(C_{1_{feven}}) = 4.8816 \cdot 10^{-4}$, $max(C_{0_{f_{odd}}}) = 1.413868$ and $max(C_{0_{feven}}) = 1.99951$.

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_{C0_f} and FB_{D0_f}, respectively, are 14, 3 and 3 for the $min(exp_f)$ case and 26, 25 and 25 for the $max(exp_f)$ case. The latter places more stringent requirements and therefore $FB_{C1_f} = 26$, $FB_{C0_f} = 25$ and $FB_{D0_f} = 25$ are chosen. For 8e, C_{1_f} has 10 redundant bits, C_{0_f} 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_{1_f} and C_{0_f}, 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 $FB_{y_e} = 43$. 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 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, $E_{approx_e} = 2.8422 \cdot 10^{-14}$, $max(C_{3_e}) = 2.47627 \cdot 10^{-9}$, $max(C_{2_e}) = 1.9073441 \cdot 10^{-6}$, $max(C_{1_e}) = 0.001953125$ 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 $FB_{C3_e} = FB_{C2_e} = FB_{C1_e} = FB_{C0_e} = FB_{D4_e} = FB_{D3_e} = 47$ and $FB_{D2_e} = FB_{D1_e} = FB_{D0_e} = 48$. 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.

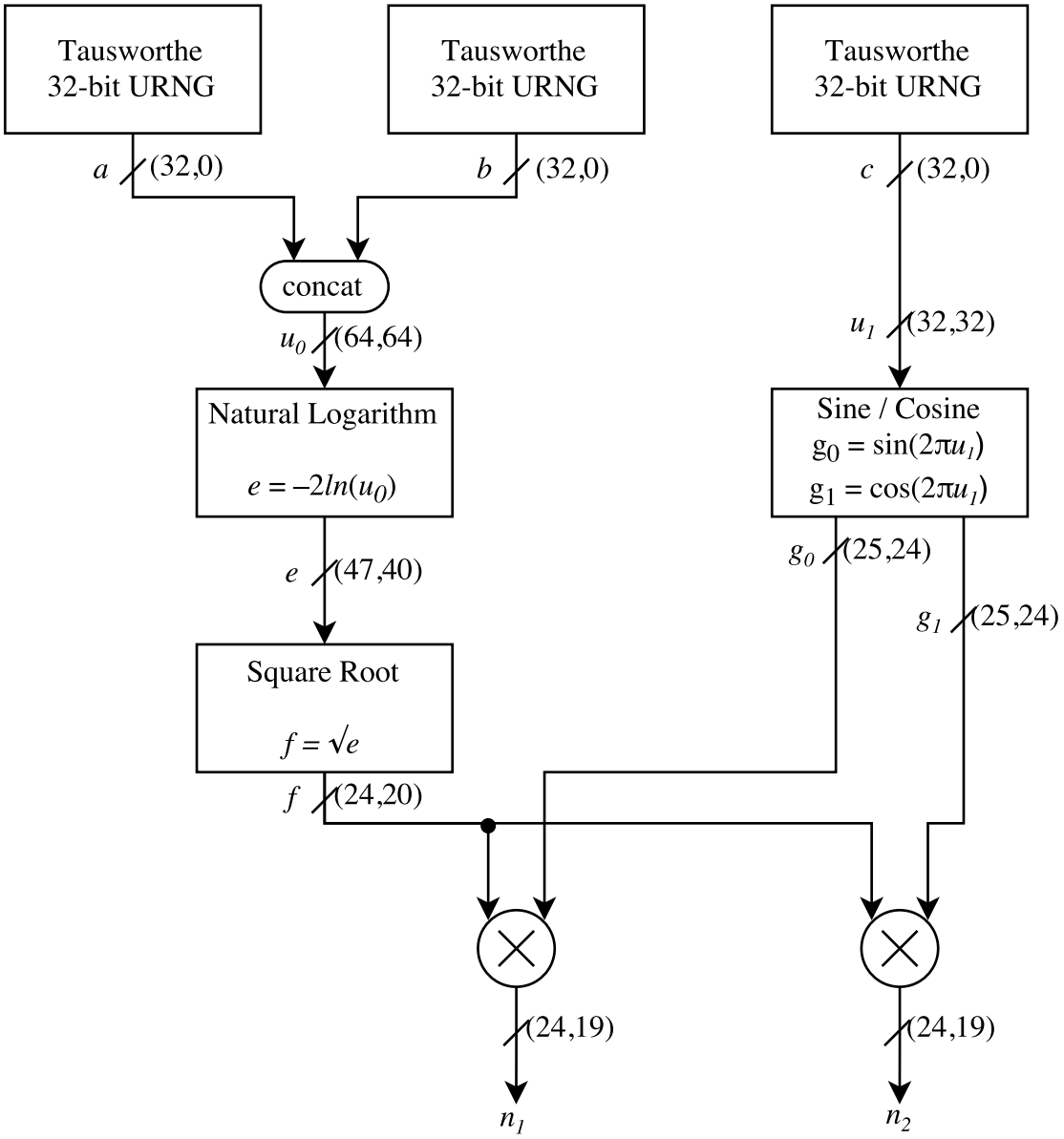

Figure 4.1: LGBMGNG 24-bit Block Diagram

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.

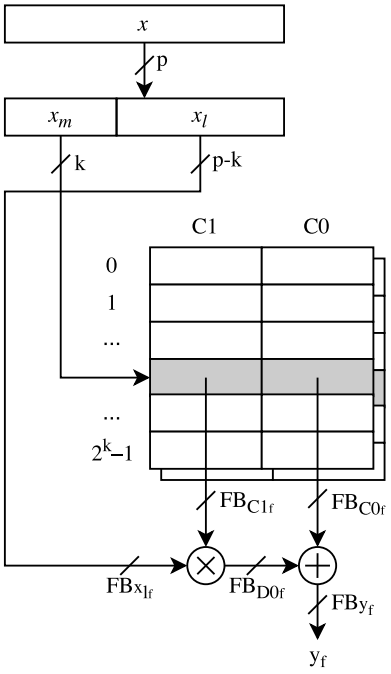

Figure 4.2: Degree One Approximation of SQRT Hardware Implementation

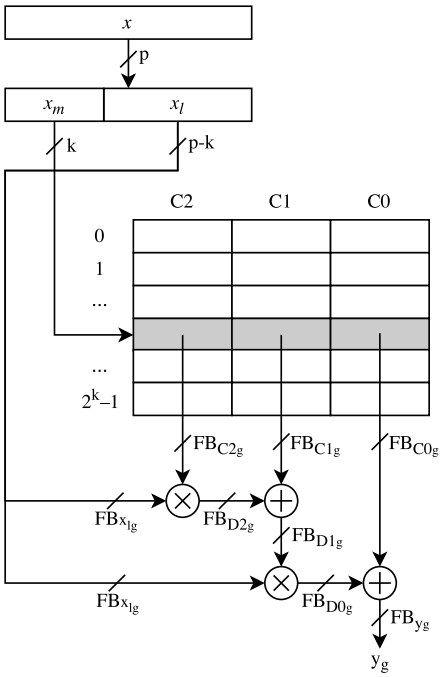

Figure 4.3: Degree Two Approximation of COS Hardware Implementation

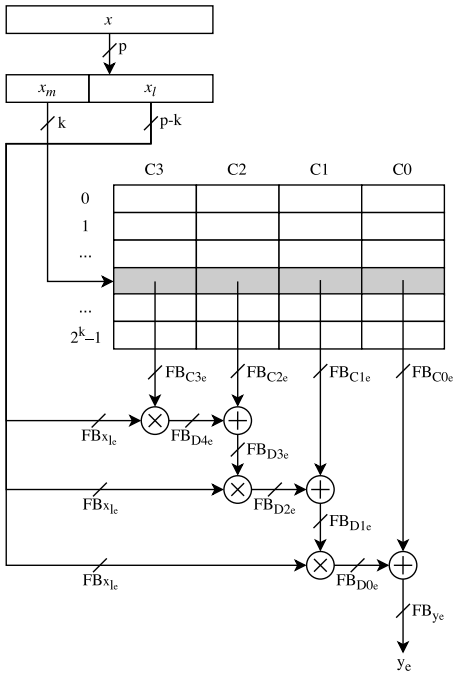

Figure 4.4: Degree Three Approximation of LN Hardware Implementation

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.

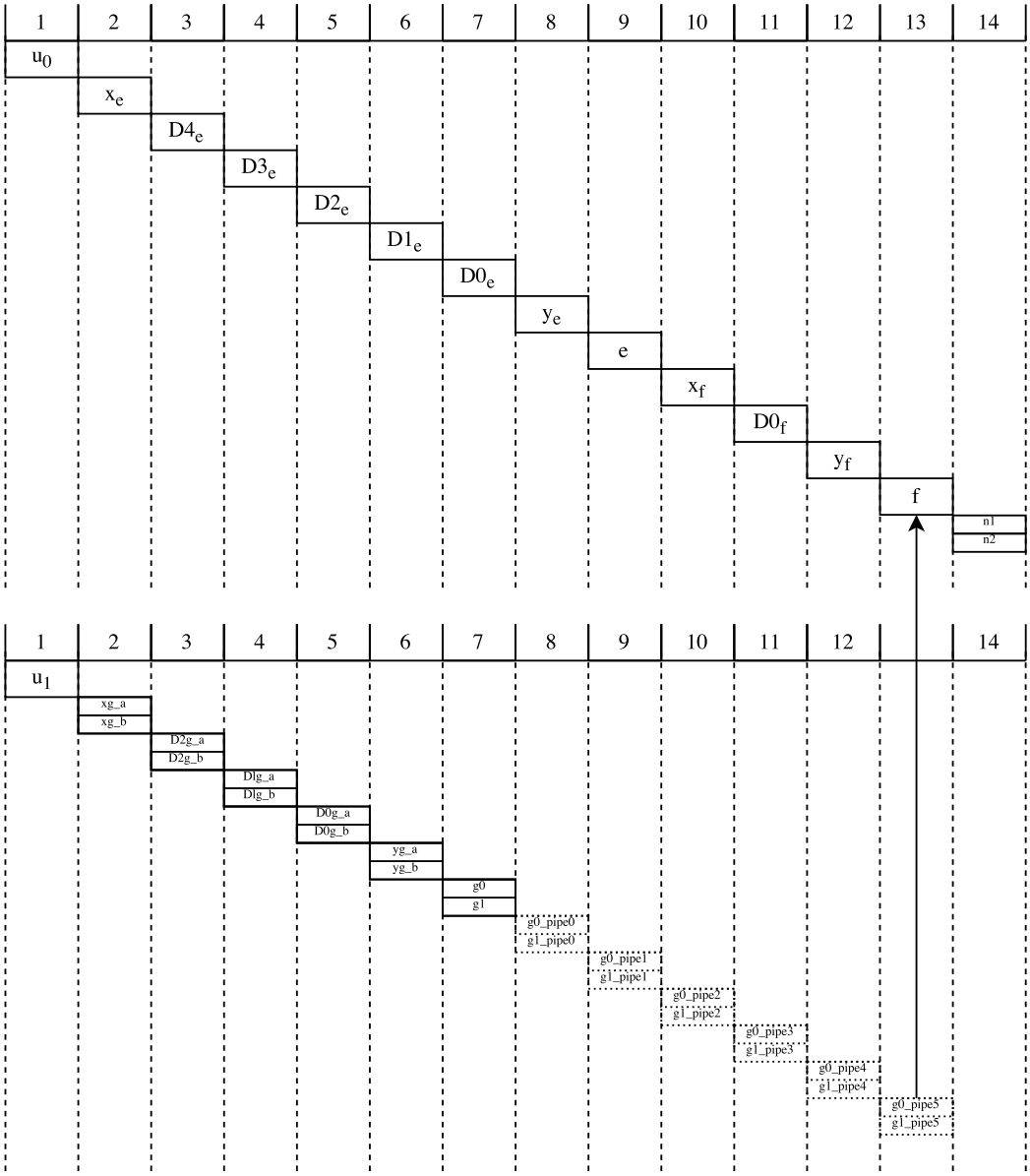

Figure 4.5: LGBMGNG Pipeline

Signals	Total bits (TB)	Integer bits (IB)	Fractional bits (FB)	Sign bit (SB)	Redundant bits (RB)
a,b,c	32	32	0	0	
u_0	64	0	64	0	
u_1	32	0	32	0	
exp_e	7	7	0	0	
x_e	64	1	63	0	
C_{3_e}	47	0	47	0	28
C_{2_e}	48	0	47	1	19
C_{1_e}	48	0	47	1	9
C_{0_e}	47	0	47	0	0
D_{4_e}	47	0	47	0	
D_{3_e}	48	0	47	1	
D_{2_e}	49	0	48	1	
D_{1_e}	48	0	48	0	
D_{0_e}	48	0	48	0	
y_e	43	0	43	0	
$ln2$	49	0	49	0	
e'	49	6	43	0	
e	47	7	40	0	
exp_f	7	6	0	1	
x'_f	47	2	46	0	
x_f	48	2	46	0	
C_{1_f}	26	0	26	0	10
C_{0_f}	26	1	25	0	0
D_{0_f}	25	0	25	0	
y_f	24	1	23	0	
exp'_f	6	5	0	1	
f	24	4	20	0	
Q	2	2	0	0	
x_{g_a},x_{g_b}	30	0	30	0	
C_{2_g}	28	0	27	1	13
C_{1_g}	28	0	27	1	6
C_{0_g}	29	1	27	1	0
D_{2_g}	28	0	27	1	
D_{1_g}	29	0	28	1	
D_{0_g}	29	0	28	1	
y_{g_a},y_{g_b}	24	0	24	0	
g_0,g_1	25	0	24	1	
n_1,n_2	24	4	19	1	

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 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 $n_1 = 0x1C3D37$ and $n_2 = 0x039349$. Seed 1 was then simulated using the gate-level netlist, where the data path through the

	Seed_a	Seed_b	Seed_c	Sample Count
Seed 1	7654321	87654321	987654321	1 Million
Seed 2	741732741	900903737	655563292	20 Million
Seed 3	1107711288	490903959	338177554	20 Million
Seed 4	886334683	2126303890	161170592	1 Million

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 μm^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.

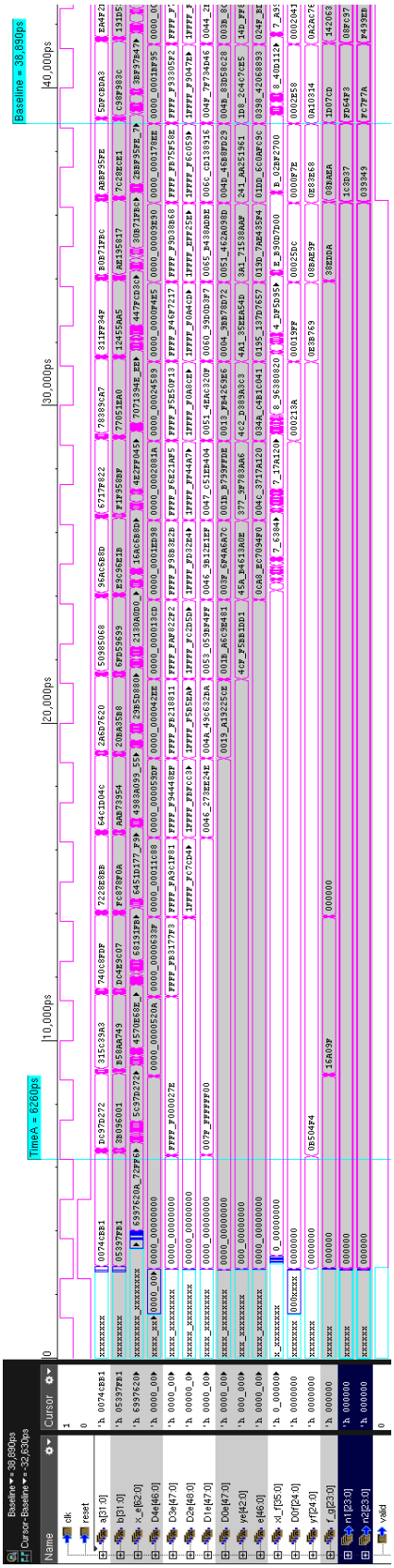

Figure 5.1: LN and SQRT Data Path Simulation Waveform

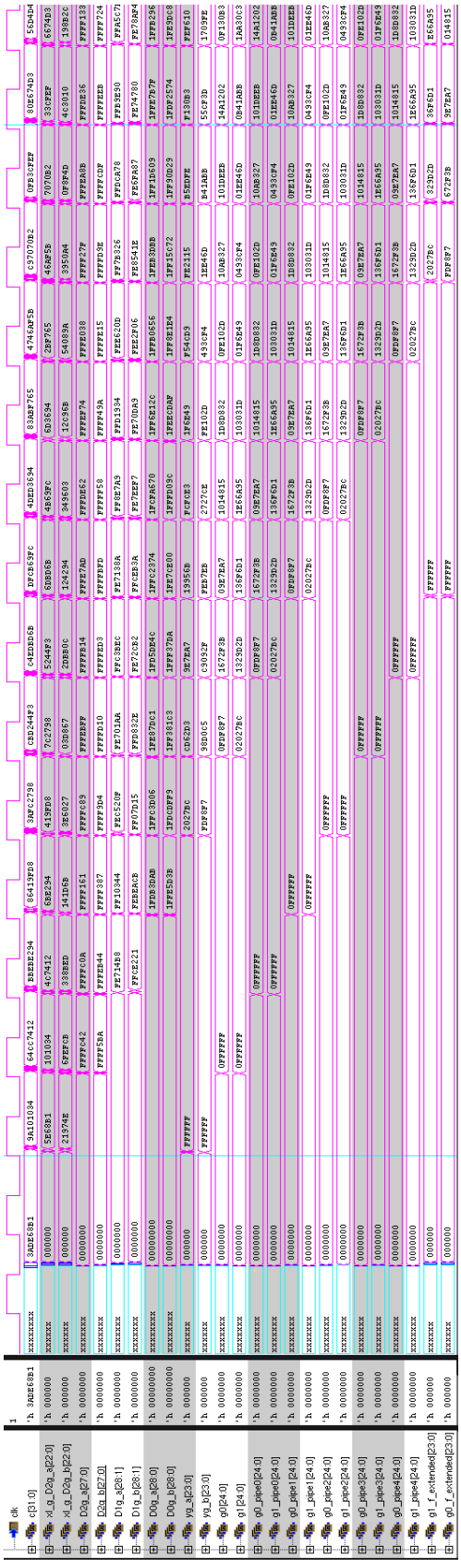

Figure 5.2: SIN/COS Data Path Simulation Waveform

```
Progress:     1000000

** "Success" **



Count =     1000000
seed_a =    7654321
seed_b =   87654321
seed_c =  987654321

Simulation complete via $finish(1) at time 2500038750 PS + 0
./src/LGBMGNG_test.v:171         $finish;
```

Figure 5.3: 1 Million Test Vectors Synthesis Simulation Waveform

```
Progress:    19600000
Progress:    19800000
Progress:    20000000

** "Success" **



Count =    20000000
seed_a =  741732741
seed_b =  900903737
seed_c =  655563292

Simulation complete via $finish(1) at time 50000038750 PS + 0
./src/LGBMGNG_test.v:171         $finish;
ncsim> exit
```

Figure 5.4: 20 Million Test Vectors RTL Simulation Waveform

	Pre-Scan Netlist	Post-Scan Netlist
Combinational Area (μm^2)	146373.12	145705.68
Buffer and Inverter Area (μm^2)	6338.16	5604.12
Non-Combinational Area (μm^2)	16164.72	21160.80
Total Cell Area (μm^2)	162537.84	166866.48
Total Number of Gates	112874	115880

Table 5.2: Logic Synthesis Reports for Pre and Post Scan Chain Netlists

	Pre-Scan Netlist	Post-Scan Netlist
Cell Internal Power (mW)	20.6966	37.1
Net Switching Power (mW)	5.0397	11.5
Total Dynamic Power (mW)	25.7363	38.6
Cell Leakage Power (μW)	9.7345	10.05

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.

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

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_{g_a} and y_{g_b} 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 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 $C_{1_g}$was outputting a single value that was negative, when the model was showing it to be 0. In the design at this point, C_{1_g} 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,

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

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.

	Seed 2		Seed 3	
	N1	N2	N1	N2
Mean	0.000644	0.000405	0.000053	0.000154
Med.	0.000633	0.000418	0.000000	0.000467
Std.	0.999892	0.999789	1.000023	0.999913
Min.	-5.379921	-5.642990	-5.905350	-5.153084
Max.	5.397518	5.335030	5.261393	5.042143
Pe_mean.	0.064446	0.040544	0.005254	0.015429
Pe_med.	0.063324	0.041771	0.000000	0.046730
Pe_std.	0.010758	0.021119	0.002292	0.008712
Chi p	0.409535	0.650388	0.342973	0.437817
A-D p	0.956603	0.663481	0.387935	0.116103
R^2	0.9999998758	0.9999998549	0.9999998175	0.9999997822
SEE	0.000352	0.000381	0.000427	0.000467

Table 5.6: Goodness-of-Fit Metrics for 20 Million Test Vector Sets

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 $400MHz$ 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 9.42σ.

References

[1] D. U. Lee, J. D. Villasenor, W. Luk, and P. H. W. Leong, "A hardware gaussian noise generator using the box-muller method and its error analysis," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 659–671, June 2006.

[2] R. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, January 1962.

[3] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *Communications, 1993. ICC '93 Geneva. Technical Program, Conference Record, IEEE International Conference on*, vol. 2, May 1993, pp. 1064–1070 vol.2.

[4] W. Hörmann and J. Leydold, "Continuous random variate generation by fast numerical inversion," *ACM Trans. Model. Comput. Simul.*, vol. 13, no. 4, pp. 347–362, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/945511.945517

[5] R. Gutierrez, V. Torres, and J. Valls, "Hardware architecture of a gaussian noise generator based on the inversion method," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 59, no. 8, pp. 501–505, Aug 2012.

[6] R. C. C. Cheung, D. U. Lee, W. Luk, and J. D. Villasenor, "Hardware generation of arbitrary random number distributions from uniform distributions via the inver-

sion method," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 8, pp. 952–962, Aug 2007.

[7] D.-U. Lee, W. Luk, J. D. Villasenor, G. Zhang, and P. H. W. Leong, "A hardware gaussian noise generator using the wallace method," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 8, pp. 911–920, Aug 2005.

[8] J. S. Malik, J. N. Malik, A. Hemani, and N. D. Gohar, "An efficient hardware implementation of high quality awgn generator using box-muller method," in *2011 11th International Symposium on Communications Information Technologies (ISCIT)*, Oct 2011, pp. 449–454.

[9] Q. Lu, J. Fan, C. W. Sham, and F. C. M. Lau, "A high throughput gaussian noise generator," in *2014 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, Nov 2014, pp. 117–120.

[10] I. Paraskevakos and V. Paliouras, "A flexible high-throughput hardware architecture for a gaussian noise generator," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2011, pp. 1673–1676.

[11] B. G. Sileshi, C. Ferrer, and J. Oliver, "Accelerating hardware gaussian random number generation using ziggurat and cordic algorithms," in *IEEE SENSORS 2014 Proceedings*, Nov 2014, pp. 2122–2125.

[12] J. S. Malik, A. Hemani, and N. D. Gohar, "Unifying cordic and box-muller algorithms: An accurate and efficient gaussian random number generator," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, June 2013, pp. 277–280.

[13] Y. Wang and Z. Bie, "A novel hardware gaussian noise generator using box-muller

and cordic," in *2014 Sixth International Conference on Wireless Communications and Signal Processing (WCSP)*, Oct 2014, pp. 1–6.

[14] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *Ann. Math. Statist.*, vol. 29, no. 2, pp. 610–611, 06 1958. [Online]. Available: http://dx.doi.org/10.1214/aoms/1177706645

[15] R. C. Tausworthe, "Random numbers generated by linear recurrence modulo two," *Mathematics of Computation*, vol. 19, no. 90, pp. 201–209, 1965. [Online]. Available: http://www.jstor.org/stable/2003345

[16] P. L'Ecuyer, "Maximally equidistributed combined tausworthe generators," *Math. Comput.*, vol. 65, no. 213, pp. 203–213, Jan. 1996. [Online]. Available: http://dx.doi.org/10.1090/S0025-5718-96-00696-5

[17] M. J. Schulte and E. E. Swartzlander, "Hardware designs for exactly rounded elementary functions," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 964–973, Aug 1994.

[18] J. Devore, *Probability and Statistics for Engineering and the Sciences*. Cengage Learning, 2011. [Online]. Available: https://books.google.com/books?id=3qoP7dlO4BUC

[19] L. Andrews, *Special Functions of Mathematics for Engineers*, ser. Oxford science publications. SPIE Optical Engineering Press, 1992. [Online]. Available: https://books.google.com/books?id=2CAqsF-RebgC

[20] R. B. D'Agostino and M. A. Stephens, Eds., *Goodness-of-fit Techniques*. New York, NY, USA: Marcel Dekker, Inc., 1986. [Online]. Available: http://dl.acm.org/citation.cfm?id=19293

[21] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, no. 4, pp. 22:1–22:40, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1268776.1268777

[22] [Online]. Available: https://www.mathworks.com/help/stats/chi2gof.html

[23] [Online]. Available: https://www.mathworks.com/help/stats/adtest.html

[24] V. Lefevre, J. M. Muller, and A. Tisserand, "Toward correctly rounded transcendentals," *IEEE Transactions on Computers*, vol. 47, no. 11, pp. 1235–1243, Nov 1998.

[25] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*, ser. AFIPS '71 (Spring). New York, NY, USA: ACM, 1971, pp. 379–385. [Online]. Available: http://doi.acm.org/10.1145/1478786.1478840

[26] M. Schulte and E. Swartzlander, "Exact rounding of certain elementary functions," in *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, Jun 1993, pp. 138–145.

[27] D.-U. Lee, A. A. Gaffar, O. Mencer, and W. Luk, "Minibit: bit-width optimization via affine arithmetic," in *Proceedings. 42nd Design Automation Conference, 2005.*, June 2005, pp. 837–840.

[28] [Online]. Available: https://www.mathworks.com/help/optim/ug/fmincon.html

[29] [Online]. Available: https://www.mathworks.com/help/stats/norminv.html

[30] A. A. Kalatchikov and G. G. Streltsov, "Fpga implementation of gaussian noise generator," in *2004 International Siberian Workshop on Electron Devices and Materials*, July 2004, pp. 100–.

[31] [Online]. Available: https://www.mathworks.com/products/fixed-point-designer.html

[32] [Online]. Available: https://www.mathworks.com/help/fixedpoint/examples/fixed-point-optimizations-using-specified-minimum-and-maximum-values.html

Appendix I

Source Code

I.1 LGBMGNG Design in Verilog HDL

```
// 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 (LGMBGNG)
 *  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
```

```
27
28  input [31:0]                    // Seeds for TAUS
29      seed_a,
30      seed_b,
31      seed_c;
32
33  output reg signed [23:0]
34      n1,                         // 24 bit N(0,1)
35      n2;                         // 24 bit N(0,1)
36
37  output reg valid;
38
39  input
40      scan_in0,                   // test scan mode data input
41      scan_en,                    // test scan mode enable
42      test_mode;                  // test mode select
43
44  output
45      scan_out0;                  // test scan mode data output
46
47  //------------------------------------------------------------------------//
48
49  // Note:
50  // TB: Total Bits
51  // FB: Fractional Bits
52  // RB: Redundant Bits
53  // SB: Sign Bit
54
55  // TAUSWORTHE
56  wire [31:0]
57      a,
58      b,
59      c;
60
61  // Actual Inputs
62
63  // UQ64
64  wire [63:0] u0;
65  parameter   TBu0 = 64;
66
67  // UQ32
68  wire [31:0] u1;
69
70  ////////////////////////////////////////////////////////////////
71  // LN UNIT //// LN UNIT //// LN UNIT //// LN UNIT //// LN UNIT ///
72  ////////////////////////////////////////////////////////////////
73
74  // LN UNIT -- e = -2*ln(u0)
75  // Range Reduction
76  parameter TBexp_e = 7;
77  wire [6:0]  u0_LZD_cnt;
78  wire [TBexp_e-1:0]  exp_e;
```

```
79
80  // UQ1.63
81  parameter FBx_e = 63;
82  wire [63:0] x_e;
83
84  // Degree 3 Function Approximation. ln(x), x is [1,2)
85  parameter   k_ln = 9;       // 512 Segments
86  parameter FBxl_e = FBx_e-k_ln;
87
88  wire [k_ln-1:0] xm_e;
89  wire [FBxl_e-1:0] xl_e;
90
91  // C3_LN UQ47
92  parameter FBC3e = 47;
93  parameter RBC3e = 28;
94  wire [FBC3e-RBC3e-1:0]  raw_val_C3e; // Values stored in LUT
95
96  // UQ47 D4e
97  // D4e = xl_e * C3e;
98  parameter FBD4e = 47;
99  reg [k_ln-1:0] sel_D4e;
100 reg [FBxl_e-1:0] xl_e_D4e;
101 wire [2*FBxl_e-1:0] D4e_next_raw;
102 wire [FBD4e-1:0] D4e_next;
103 reg [FBD4e-1:0]  D4e;
104
105 // C2_LN Q47 $$ Always Negative $$
106 parameter FBC2e = 47;
107 parameter RBC2e = 19;
108 wire [FBC2e-RBC2e-1:0]  raw_val_C2e;            // Values stored in LUT
109 wire signed [FBC2e:0]   val_C2e;
110
111 // Q47 D3e
112 // D3e = D4e + C2e;
113 parameter FBD3e = 47;
114 reg [k_ln-1:0] sel_D3e;
115 reg [FBxl_e-1:0] xl_e_D3e;
116 wire signed [FBD3e:0] D3e_next;
117 reg signed [FBD3e:0]  D3e;
118
119 // Q48 D2e
120 // D2e = D3e * xl_e;
121 parameter FBD2e = 48;
122 reg [k_ln-1:0] sel_D2e;
123 reg signed [FBxl_e:0] xl_e_D2e;
124 wire signed [2*(FBxl_e+1)-1:0] D2e_next_raw;
125 wire signed [FBD2e:0] D2e_next;
126 reg signed [FBD2e:0]  D2e;
127
128 // UQ48 D1e
129 // D1e = D2e + C1e;
130 parameter FBD1e = 48;
```

```
131 reg [k_ln-1:0] sel_D1e;
132 reg [FBxl_e-1:0] xl_e_D1e;
133 wire signed [FBD1e:0] D1e_next;
134 reg [FBD1e-1:0] D1e;
135
136 // C1_LN Q47
137 parameter FBC1e = 47;
138 parameter RBC1e = 9;
139 wire [FBC1e-RBC1e-1:0]  raw_val_C1e; // Values stored in LUT
140 wire signed [FBD1e:0] val_C1e; // Prep for signed arithmetic since D2 is negative.
141
142 // UQ48 D0e
143 // D0e = xl_e * D1e;
144 parameter FBD0e = 48;
145 reg [k_ln-1:0] sel_D0e;
146 reg [FBxl_e-1:0] xl_e_D0e;
147 wire [125:0] D0e_next_raw;
148 wire [FBD0e-1:0] D0e_next;
149 reg [FBD0e-1:0]  D0e;
150
151 // C0_LN UQ47
152 parameter FBC0e = 47;
153 parameter RBC0e = 0;
154 wire [FBC0e-RBC0e-1:0]  raw_val_C0e; // Values stored in LUT
155
156 // UQ43 ye
157 // ye = D0e + C0e;
158 parameter FBye = 43;
159 reg [k_ln-1:0] sel_ye;
160 wire [FBD0e-1:0] ye_next_raw;
161 wire [FBye-1:0] ye_next;
162 reg [FBye-1:0] ye;
163
164 // Range Reconstruction
165 // UQ6.43 = UQ7.0 * UQ29
166 // e_p = exp_e * LN2;
167     // exp_e pipeline bc val needed for e_p_raw
168 reg [TBexp_e-1:0] exp_e_D4e;
169 reg [TBexp_e-1:0] exp_e_D3e;
170 reg [TBexp_e-1:0] exp_e_D2e;
171 reg [TBexp_e-1:0] exp_e_D1e;
172 reg [TBexp_e-1:0] exp_e_D0e;
173 reg [TBexp_e-1:0] exp_e_ye;
174 reg [TBexp_e-1:0] exp_e_e;
175 wire [54:0] e_p_raw; // TB = 49 + 6;
176 wire [48:0] e_p;
177
178 // UQ7.40 = (UQ6.43-UQ43)*(2*2^43)
179 // e = (e_p-y_e)*2;
180 wire [48:0] e_offset;
181 wire [49:0] e_next_raw;
182 wire [46:0] e_next;
```

```
183 reg [46:0] e;
184
185 ////////////////////////////////////////////////////////
186 // SQRT UNIT //// SQRT UNIT //// SQRT UNIT //// SQRT UNIT/
187 ////////////////////////////////////////////////////////
188
189 // SQRT Unit -- f = sqrt(e) = sqrt(-2ln(u0))
190
191 // Range Reduction
192
193 // Q6.0 expf = 5-LZD(e);
194 parameter TBexp_f = 7; // 1SB 6IB
195 wire [TBexp_f-2:0]          e_LZD_cnt;
196 wire signed [TBexp_f-1:0]   exp_f;
197
198 // x_f_p = e >> exp_f;
199 // UQ2.45 = UQ7.40 >> exp_f;
200 parameter FBx_f_p = 45;
201 parameter TBx_f_p = 47;
202 wire [TBx_f_p-1:0] x_f_p;
203
204 // x_f = exp_f[0] ? x_f_p/2 : x_f_p;
205 // UQ2.46
206 parameter FBx_f = 46;
207 parameter TBx_f = 48;
208 wire [TBx_f-1:0] x_f;
209
210 // Degree 1 Function Approximation. sqrt(x), x is [2,4)
211 parameter   k_f = 10;       // 1024 Segments
212 parameter   FBxl_f = FBx_f-k_f;
213 wire [k_f-1:0] xm_f;
214 wire [FBxl_f-1:0] xl_f;
215
216 // C1_SQRT UQ26
217 parameter FBC1f = 26;
218 parameter RBC1f = 10;
219 wire [FBC1f-RBC1f-1:0]  val_C1f;
220 wire [FBC1f-RBC1f-2:0]  raw_val_C1f_1_2; // Values stored in LUT Note: 1_2 has one
    ↪  more RB
221 wire [FBC1f-RBC1f-1:0]  raw_val_C1f_2_4; // Values stored in LUT
222
223 // UQ25 D0f
224 // D0f = xl_f * C1f;
225 parameter FBD0f = 25;
226 reg [k_f-1:0] sel_D0f;
227 reg [FBxl_f-1:0] xl_f_D0f;
228 reg signed [TBexp_f-1:0]   exp_f_D0f;
229 wire [2*FBxl_f-1:0] D0f_next_raw;
230 wire [FBD0f-1:0] D0f_next;
231 reg [FBD0f-1:0]  D0f;
232
233 // C0_SQRT UQ1.25
```

```
234 parameter FBC0f = 25;
235 parameter RBC0f = 0;
236 wire [FBC0f:0]  val_C0f; // Values stored in LUT
237 wire [FBC0f:0]  raw_val_C0f_1_2; // Values stored in LUT
238 wire [FBC0f:0]  raw_val_C0f_2_4; // Values stored in LUT
239
240 // UQ1.23 yf
241 // yf = D0f + C0f;
242 parameter FByf = 43;
243 reg [k_f-1:0] sel_yf;
244 reg signed [TBexp_f-1:0]   exp_f_yf;
245 wire [FBD0f:0] yf_next_raw;
246 wire [FByf-1:0] yf_next;
247 reg [FByf-1:0] yf;
248
249 // Range Reconstruction
250 // exp_f_p = exp_f[0] ? exp_f+1>>1 : exp_f>>1;
251 parameter TBexp_f_p = 6; // 1SB 5IB
252 reg signed [TBexp_f-1:0]   exp_f_f;
253 wire signed [TBexp_f-1:0] exp_f_p_raw;
254 wire signed [TBexp_f_p-1:0] exp_f_p;
255
256 // f UQ4.20
257 // f = y_f << exp_f_p;
258 parameter FBf = 20;
259 parameter TBf = 24;
260 wire [46:0] f_next_raw;
261 wire [TBf-1:0] f_next;
262 reg [TBf-1:0] f;
263 wire signed [TBf:0] f_g;
264
265 ////////////////////////////////////////////////////////
266 // SIN/COS UNIT //// SIN/COS UNIT //// SIN/COS UNIT ////
267 ////////////////////////////////////////////////////////
268
269 // SIN/COS Unit -- g0 = sin(2*pi*u1) & g1 = cos(2*pi*u1)
270
271 // Range Reduction
272
273 // UQ2.0
274 // Q = u1[31:30]
275 wire [1:0] Q;
276
277 // xg_a/xg_b UQ30
278 parameter FBx_g = 30;
279 wire [FBx_g-1:0] xg_a;
280 wire [FBx_g-1:0] xg_b;
281
282 // Degree 2 Function Approximation. cos(x*pi/2), x is [0,1)
283 parameter   k_g = 7;        // 128 Segments
284 parameter FBxl_g = FBx_g-k_g;
285
```

```
286 wire [k_g-1:0] xm_g_a;
287 wire [FBxl_g-1:0] xl_g_a;
288
289 wire [k_g-1:0] xm_g_b;
290 wire [FBxl_g-1:0] xl_g_b;
291
292 // C2_COS Q27
293 parameter FBC2g = 27;
294 parameter RBC2g = 13;
295
296 wire [FBC2g-RBC2g-1:0]  raw_val_C2g_a;  // Values stored in LUT
297 wire signed [FBC2g:0]   val_C2g_a;
298
299 wire [FBC2g-RBC2g-1:0]  raw_val_C2g_b;  // Values stored in LUT
300 wire signed [FBC2g:0]   val_C2g_b;
301
302 // Q27 D2g
303 // Q27 = Q23 * Q27
304 // D2g = xl_g * C2g;
305 parameter FBD2g = 27;
306
307 reg [k_g-1:0] sel_D2g_a;
308 reg signed [FBxl_g:0] xl_g_D2g_a;
309 wire signed [2*(FBC2g+1)-1:0] D2g_next_raw_a;
310 wire signed [FBD2g:0] D2g_next_a;
311 reg signed [FBD2g:0]  D2g_a;
312
313 reg [k_g-1:0] sel_D2g_b;
314 reg signed [FBxl_g:0] xl_g_D2g_b;
315 wire signed [2*(FBC2g+1)-1:0] D2g_next_raw_b;
316 wire signed [FBD2g:0] D2g_next_b;
317 reg signed [FBD2g:0]  D2g_b;
318
319 // C1_COS Q27
320 parameter FBC1g = 27;
321 parameter RBC1g = 6;
322
323 wire [FBC1g-RBC1g-1:0]  raw_val_C1g_a;  // Values stored in LUT
324 wire signed [FBC1g:0]   val_C1g_a;
325
326 wire [FBC1g-RBC1g-1:0]  raw_val_C1g_b;  // Values stored in LUT
327 wire signed [FBC1g:0]   val_C1g_b;
328
329 // D1g Q28
330 // Q28 = Q27 + Q27;
331 // D1g = D2g + C1g;
332 parameter FBD1g = 28;
333
334 reg [k_g-1:0] sel_D1g_a;
335 reg signed [FBxl_g:0] xl_g_D1g_a;
336 wire signed [FBD1g:0] D1g_next_a;
337 reg signed [FBD1g:0]  D1g_a;
```

```
338
339 reg [k_g-1:0] sel_D1g_b;
340 reg signed [FBxl_g:0] xl_g_D1g_b;
341 wire signed [FBD1g:0] D1g_next_b;
342 reg signed [FBD1g:0]  D1g_b;
343
344 // Q28 D0g
345 // Q28 = Q23 * Q28
346 // D0g = xl_g * D1g;
347 parameter FBD0g = 28;
348
349 reg [k_g-1:0] sel_D0g_a;
350 reg signed [FBxl_g:0] xl_g_D0g_a;
351 wire signed [2*(FBD1g+1)-1:0] D0g_next_raw_a;
352 wire signed [FBD0g:0] D0g_next_a;
353 reg signed [FBD0g:0]  D0g_a;
354
355 reg [k_g-1:0] sel_D0g_b;
356 reg signed [FBxl_g:0] xl_g_D0g_b;
357 wire signed [2*(FBD1g+1)-1:0] D0g_next_raw_b;
358 wire signed [FBD0g:0] D0g_next_b;
359 reg signed [FBD0g:0]  D0g_b;
360
361 // C0_COS Q1.27
362 parameter FBC0g = 27;
363 parameter RBC0g = 0;
364
365 wire [FBC0g-RBC0g:0]  raw_val_C0g_a;  // Values stored in LUT
366 wire signed [FBC0g+1:0]   val_C0g_a;
367
368 wire [FBC0g-RBC0g:0]  raw_val_C0g_b;  // Values stored in LUT
369 wire signed [FBC0g+1:0]   val_C0g_b;
370
371 // yg UQ24
372 // UQ24 = Q28 + Q1.27;
373 // yg = D0g + C0g;
374 parameter FByg = 24;
375
376 reg [k_g-1:0] sel_yg_a;
377 wire signed [FBD0g+1:0] yg_next_raw_a;
378 wire [FByg+1:0] yg_next_raw_rtn_a;
379 wire [FByg-1:0] yg_next_a;
380 reg [FByg-1:0]  yg_a;
381
382 reg [k_g-1:0] sel_yg_b;
383 wire signed [FBD0g+1:0] yg_next_raw_b;
384 wire [FByg+1:0] yg_next_raw_rtn_b;
385 wire [FByg-1:0] yg_next_b;
386 reg [FByg-1:0]  yg_b;
387
388 // Range Reconstruction
389 // switch(Q)
```

```
390 //   case 0: g0 = y_g_b;     g1 = y_g_a;
391 //   case 1: g0 = y_g_a;     g1 = -y_g_b;
392 //   case 2: g0 = -y_g_b;    g1 -y_g_a;
393 //   case 3: g0 = -y_g_a;    g1 = y_g_b;
394
395 // need to pipeline Q val
396 reg [1:0] Q_D2g;
397 reg [1:0] Q_D1g;
398 reg [1:0] Q_D0g;
399 reg [1:0] Q_yg;
400 reg [1:0] Q_g;
401
402 // g Q24
403 parameter FBg = 24;
404 wire signed [FBg:0] g0_next;
405 wire signed [FBg:0] g1_next;
406 reg signed [FBg:0] g0;
407 reg signed [FBg:0] g1;
408
409 // Noise Creation
410
411 // Pipeline g0,g1
412 reg signed [FBg:0] g0_pipe0;
413 reg signed [FBg:0] g1_pipe0;
414 reg signed [FBg:0] g0_pipe1;
415 reg signed [FBg:0] g1_pipe1;
416 reg signed [FBg:0] g0_pipe2;
417 reg signed [FBg:0] g1_pipe2;
418 reg signed [FBg:0] g0_pipe3;
419 reg signed [FBg:0] g1_pipe3;
420 reg signed [FBg:0] g0_pipe4;
421 reg signed [FBg:0] g1_pipe4;
422 reg signed [FBg:0] g0_f;
423 reg signed [FBg:0] g1_f;
424
425 wire signed [57:0] f_g_extended;
426 wire signed [57:0] g0_f_extended;
427 wire signed [57:0] g1_f_extended;
428
429 wire signed [57:0] n1_next_raw;
430 wire signed [57:0] n2_next_raw;
431 wire signed [23:0] n1_next;
432 wire signed [23:0] n2_next;
433
434 // valid bit pipeline
435 reg valid_pipe;
436 reg valid_pipe0;
437 reg valid_pipe1;
438 reg valid_pipe2;
439 reg valid_pipe3;
440 reg valid_pipe4;
441 reg valid_pipe5;
```

```
442 reg valid_pipe6;
443 reg valid_pipe7;
444 reg valid_pipe8;
445 reg valid_pipe9;
446 reg valid_pipe_last;
447
448 //------------------------------------------------------------------------------//
449
450     TAUS TAUS_a (
451                     .reset(reset),
452                     .clk(clk),
453                     .seed(seed_a),
454                     .u0(a),
455                     .scan_in0(scan_in0),
456                     .scan_en(scan_en),
457                     .test_mode(test_mode),
458                     .scan_out0(scan_out0)
459                 );
460
461     TAUS TAUS_b (
462                     .reset(reset),
463                     .clk(clk),
464                     .seed(seed_b),
465                     .u0(b),
466                     .scan_in0(scan_in0),
467                     .scan_en(scan_en),
468                     .test_mode(test_mode),
469                     .scan_out0(scan_out0)
470                 );
471     TAUS TAUS_c (
472                     .reset(reset),
473                     .clk(clk),
474                     .seed(seed_c),
475                     .u0(c),
476                     .scan_in0(scan_in0),
477                     .scan_en(scan_en),
478                     .test_mode(test_mode),
479                     .scan_out0(scan_out0)
480                 );
481     LZD_u0 LZD_LN (
482                     .reset(reset),
483                     .clk(clk),
484                     .val(u0),
485                     .LZ_count(u0_LZD_cnt),
486                     .scan_in0(scan_in0),
487                     .scan_en(scan_en),
488                     .test_mode(test_mode),
489                     .scan_out0(scan_out0)
490                 );
491     C3_LN C3_LN (
492                     .reset(reset),
493                     .clk(clk),
```

```
494                     .sel(sel_D4e),
495                     .val(raw_val_C3e),
496                     .scan_in0(scan_in0),
497                     .scan_en(scan_en),
498                     .test_mode(test_mode),
499                     .scan_out0(scan_out0)
500                 );
501     C2_LN C2_LN (
502                     .reset(reset),
503                     .clk(clk),
504                     .sel(sel_D3e),
505                     .val(raw_val_C2e),
506                     .scan_in0(scan_in0),
507                     .scan_en(scan_en),
508                     .test_mode(test_mode),
509                     .scan_out0(scan_out0)
510                 );
511     C1_LN C1_LN (
512                     .reset(reset),
513                     .clk(clk),
514                     .sel(sel_D1e),
515                     .val(raw_val_C1e),
516                     .scan_in0(scan_in0),
517                     .scan_en(scan_en),
518                     .test_mode(test_mode),
519                     .scan_out0(scan_out0)
520                 );
521     C0_LN C0_LN (
522                     .reset(reset),
523                     .clk(clk),
524                     .sel(sel_ye),
525                     .val(raw_val_C0e),
526                     .scan_in0(scan_in0),
527                     .scan_en(scan_en),
528                     .test_mode(test_mode),
529                     .scan_out0(scan_out0)
530                 );
531     LZD_e LZD_SQRT (
532                     .reset(reset),
533                     .clk(clk),
534                     .val(e),
535                     .LZ_count(e_LZD_cnt),
536                     .scan_in0(scan_in0),
537                     .scan_en(scan_en),
538                     .test_mode(test_mode),
539                     .scan_out0(scan_out0)
540                 );
541     C1_SQRT_1_2 C1_SQRT_1_2 (
542                     .reset(reset),
543                     .clk(clk),
544                     .sel(sel_D0f),
545                     .val(raw_val_C1f_1_2),
```

```
546                     .scan_in0(scan_in0),
547                     .scan_en(scan_en),
548                     .test_mode(test_mode),
549                     .scan_out0(scan_out0)
550                 );
551     C1_SQRT_2_4 C1_SQRT_2_4 (
552                     .reset(reset),
553                     .clk(clk),
554                     .sel(sel_D0f),
555                     .val(raw_val_C1f_2_4),
556                     .scan_in0(scan_in0),
557                     .scan_en(scan_en),
558                     .test_mode(test_mode),
559                     .scan_out0(scan_out0)
560                 );
561     C0_SQRT_1_2 C0_SQRT_1_2 (
562                     .reset(reset),
563                     .clk(clk),
564                     .sel(sel_yf),
565                     .val(raw_val_C0f_1_2),
566                     .scan_in0(scan_in0),
567                     .scan_en(scan_en),
568                     .test_mode(test_mode),
569                     .scan_out0(scan_out0)
570                 );
571     C0_SQRT_2_4 C0_SQRT_2_4 (
572                     .reset(reset),
573                     .clk(clk),
574                     .sel(sel_yf),
575                     .val(raw_val_C0f_2_4),
576                     .scan_in0(scan_in0),
577                     .scan_en(scan_en),
578                     .test_mode(test_mode),
579                     .scan_out0(scan_out0)
580                 );
581     C2_COS C2_COS_a (
582                     .reset(reset),
583                     .clk(clk),
584                     .sel(sel_D2g_a),
585                     .val(raw_val_C2g_a),
586                     .scan_in0(scan_in0),
587                     .scan_en(scan_en),
588                     .test_mode(test_mode),
589                     .scan_out0(scan_out0)
590                 );
591     C2_COS C2_COS_b (
592                     .reset(reset),
593                     .clk(clk),
594                     .sel(sel_D2g_b),
595                     .val(raw_val_C2g_b),
596                     .scan_in0(scan_in0),
597                     .scan_en(scan_en),
```

```
598                     .test_mode(test_mode),
599                     .scan_out0(scan_out0)
600                 );
601     C1_COS C1_COS_a (
602                     .reset(reset),
603                     .clk(clk),
604                     .sel(sel_D1g_a),
605                     .val(raw_val_C1g_a),
606                     .scan_in0(scan_in0),
607                     .scan_en(scan_en),
608                     .test_mode(test_mode),
609                     .scan_out0(scan_out0)
610                 );
611     C1_COS C1_COS_b (
612                     .reset(reset),
613                     .clk(clk),
614                     .sel(sel_D1g_b),
615                     .val(raw_val_C1g_b),
616                     .scan_in0(scan_in0),
617                     .scan_en(scan_en),
618                     .test_mode(test_mode),
619                     .scan_out0(scan_out0)
620                 );
621     C0_COS C0_COS_a (
622                     .reset(reset),
623                     .clk(clk),
624                     .sel(sel_yg_a),
625                     .val(raw_val_C0g_a),
626                     .scan_in0(scan_in0),
627                     .scan_en(scan_en),
628                     .test_mode(test_mode),
629                     .scan_out0(scan_out0)
630                 );
631     C0_COS C0_COS_b (
632                     .reset(reset),
633                     .clk(clk),
634                     .sel(sel_yg_b),
635                     .val(raw_val_C0g_b),
636                     .scan_in0(scan_in0),
637                     .scan_en(scan_en),
638                     .test_mode(test_mode),
639                     .scan_out0(scan_out0)
640                 );
641 // Actual Inputs
642     assign u0 = {a,b};
643     assign u1 = c;
644 
645 ////////////////////////////////////////////////////////////////////
646 // LN UNIT //// LN UNIT //// LN UNIT //// LN UNIT //// LN UNIT ///
647 ////////////////////////////////////////////////////////////////////
648 
649 // LN Range Reduction
```

```
650     assign exp_e = u0_LZD_cnt + 1;
651     assign x_e =  u0 << (exp_e-1);
652
653 // LN Degree Three Approximation
654     assign xm_e = x_e[FBx_e-1:FBx_e-k_ln];
655     assign xl_e = x_e[FBx_e-k_ln-1:0];//
656
657 // D4e = xl_e * C3e;
658     // UQ47 = UQFBx_l_e * UQ47;
659     assign D4e_next_raw = xl_e_D4e * {raw_val_C3e,{(FBxl_e - FBC3e){1'b0}}};
660     assign D4e_next = D4e_next_raw[2*FBxl_e-1:2*FBxl_e-FBD4e] +
        ↪  D4e_next_raw[2*FBxl_e-FBD4e-1]; // w/ Round to nearest
661
662 // D3e = D4e + C2e;
663     // Q47 = UQ47 + Q47;
664     assign val_C2e = {{(RBC2e+1){1'b1}},raw_val_C2e};   // Executes sign extension.
        ↪  C2 stored in 2s comp w.o SB
665     assign D3e_next = D4e + val_C2e;
666
667 // D2e = xl_e * D3e;
668     // Q48 = QFBxl_e * Q47
669     assign D2e_next_raw = {{55{1'b0}},xl_e_D2e} * {{55{D3e[47]}},D3e,{(FBxl_e -
        ↪  FBD3e){1'b0}}};
670     assign D2e_next =
        ↪  {D2e_next_raw[2*(FBxl_e+1)-1],D2e_next_raw[2*FBxl_e-1:2*FBxl_e-FBD2e]} +
        ↪  D2e_next_raw[2*FBxl_e-FBD2e-1]; // w/ Round to nearest plus SB stuff
671
672 // D1e = D2e + C1e;
673     // Q48 = Q48 + Q47;
674     assign val_C1e = {{1'b0},raw_val_C1e,1'b0};
675     assign D1e_next = D2e + val_C1e;
676
677 // D0e = xl_e * D1e;
678     // UQ48 = UQFBx_e * UQ48;
679     assign D0e_next_raw = {xl_e_D0e,{k_ln{1'b0}}} * {D1e,{(FBx_e - FBD1e){1'b0}}};
680     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
681
682 // ye = D0e + C0e;
683     // UQ43 = UQ48 + UQ47;
684     assign ye_next_raw = D0e + {raw_val_C0e,1'b0};
685     assign ye_next = ye_next_raw[47:5] + ye_next_raw[4];   //[47:47-43+1] w/ Round to
        ↪  nearest
686
687 // Range Reconstruction
688 // ln2 UQ49
689     `define LN2 49'h162E42FEFA39F   // 390207173010335 -- 49 FB
690 // e_p = exp_e * LN2;
691     // e_p UQ6.43     UQ7.0 * UQ49
692     assign e_p_raw = exp_e_e * `LN2;        // Q6.49 TODO: 7.49?
693     assign e_p = e_p_raw[54:6] + e_p_raw[5];            // 49FB -> 43FB Q6.43 w/
        ↪  Round to nearest
```

```
694 // e = (e_p-ye)*2;
695     // e UQ7.40       UQ6.43 - UQ43
696     assign e_offset = (e_p-ye);             // Q6.43
697     assign e_next_raw = {e_offset,1'b0};    // Q7.43
698     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)
699
700 ////////////////////////////////////////////////////////////
701 // SQRT UNIT //// SQRT UNIT //// SQRT UNIT //// SQRT UNIT/
702 ////////////////////////////////////////////////////////////
703
704 // Range Reduction
705 // expf = 5-LZD(e);
706     assign exp_f = 5 - e_LZD_cnt;
707
708 // x_f_p = e >> exp_f;
709 // UQ2.45 = UQ7.40 >> exp_f;
710     assign x_f_p = e << e_LZD_cnt;
711
712 // x_f = exp_f[0] ? x_f_p/2 : x_f_p;
713 // UQ2.46
714     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
715
716 // SQRT Degree One Approximation
717     assign xm_f = exp_f[0] ? x_f[FBx_f-1:FBx_f-k_f] : x_f[FBx_f:FBx_f-k_f+1];
718     assign xl_f = exp_f[0] ? x_f[FBx_f-k_f-1:0]     : x_f[FBx_f-k_f:1];
719
720 // D0f = xl_f * C1f;
721     // UQ25 = UQFBxl_f * UQ26;
722     assign val_C1f = exp_f_D0f[0] ? {1'b0,raw_val_C1f_1_2}: raw_val_C1f_2_4;
723     assign D0f_next_raw = xl_f_D0f * {val_C1f,{(FBxl_f - FBC1f){1'b0}}};
724     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
725
726 // yf = D0f + C0f;
727     // UQ1.23 = UQ25 + UQ1.25;
728     assign val_C0f = exp_f_yf[0] ? raw_val_C0f_1_2 : raw_val_C0f_2_4;
729     assign yf_next_raw = D0f + val_C0f;
730     assign yf_next = yf_next_raw[25:2] + yf_next_raw[1];   // w/ Round to nearest
731
732 // Range Reconstruction
733 // exp_f_p = exp_f[0] ? exp_f+1>>1 : exp_f>>1;
734     // Q5.0 from Q6.0
735     assign exp_f_p_raw = exp_f_f[0] ? exp_f_f + 1 : exp_f_f;
736     assign exp_f_p = exp_f_p_raw[6:1];
737
738 // f UQ4.20
739 // f = y_f << exp_f_p;
740     assign f_next_raw = yf << (20 + exp_f_p);
741     assign f_next = f_next_raw[46:23] + f_next_raw[22];
742
```

```
743 ////////////////////////////////////////////////////////
744 // SIN/COS UNIT //// SIN/COS UNIT //// SIN/COS UNIT ////
745 ////////////////////////////////////////////////////////
746
747 // Range Reduction
748 // UQ2.0
749 // Q = u1[31:30]
750     assign Q = u1[31:30];
751
752 // xg UQ30
753     assign xg_a = u1[FBx_g-1:0];
754     assign xg_b = ~xg_a;
755
756 // Degree Two Function Approximation
757     assign xm_g_a = xg_a[FBx_g-1:FBx_g-k_g];
758     assign xl_g_a = xg_a[FBx_g-k_g-1:0];
759
760     assign xm_g_b = xg_b[FBx_g-1:FBx_g-k_g];
761     assign xl_g_b = xg_b[FBx_g-k_g-1:0];
762
763 // D2g Q27
764     // Q27 = Q23 * Q27
765     // D2g = xl_g * C2g;
766     assign val_C2g_a = {{(RBC2g+1){1'b1}},raw_val_C2g_a};
767     assign val_C2g_b = {{(RBC2g+1){1'b1}},raw_val_C2g_b};
768
769     assign D2g_next_raw_a = {{28{1'b0}},xl_g_D2g_a,4'b0000} * {{28{1'b1}},val_C2g_a};
        ↪  // 2s complement multiplication requires full end range sign extension
770     assign D2g_next_raw_b = {{28{1'b0}},xl_g_D2g_b,4'b0000} * {{28{1'b1}},val_C2g_b};
        ↪  // 2s complement multiplication requires full end range sign extension
771
772     assign D2g_next_a = D2g_next_raw_a[2*FBC2g:2*FBC2g-27] +
        ↪  D2g_next_raw_a[2*FBC2g-28];    // w/ round to nearest
773     assign D2g_next_b = D2g_next_raw_b[2*FBC2g:2*FBC2g-27] +
        ↪  D2g_next_raw_b[2*FBC2g-28];    // w/ round to nearest
774
775 // D1g Q28
776     // Q28 = Q27 + Q27;
777     // D1g = D2g + C1g;
778     // All negative except 0 index
779     assign val_C1g_a = sel_D1g_a == 7'b0000000 ? 28'h0000000 :
        ↪  {{(RBC1g+1){1'b1}},raw_val_C1g_a};
780     assign val_C1g_b = sel_D1g_b == 7'b0000000 ? 28'h0000000 :
        ↪  {{(RBC1g+1){1'b1}},raw_val_C1g_b};
781
782     assign D1g_next_a = {(D2g_a + val_C1g_a),1'b0};
783     assign D1g_next_b = {(D2g_b + val_C1g_b),1'b0};
784
785 // D0g Q28
786     // Q28 = Q23 * Q28
787     // D0g = xl_g * D1g;
788
```

```
789     assign D0g_next_raw_a = {{29{1'b0}},xl_g_D0g_a,5'b00000} *
        ↪  {{29{D1g_a[FBD1g]}},D1g_a};       // 2s complement multiplication requires
        ↪  full end range sign extension
790     assign D0g_next_raw_b = {{29{1'b0}},xl_g_D0g_b,5'b00000} *
        ↪  {{29{D1g_b[FBD1g]}},D1g_b};       // 2s complement multiplication requires
        ↪  full end range sign extension
791
792     assign D0g_next_a = D0g_next_raw_a[2*FBD1g:2*FBD1g-28] +
        ↪  D0g_next_raw_a[2*FBD1g-29];    // w/ round to nearest
793     assign D0g_next_b = D0g_next_raw_b[2*FBD1g:2*FBD1g-28] +
        ↪  D0g_next_raw_b[2*FBD1g-29];    // w/ round to nearest
794
795 // yg UQ24
796     // UQ24 = Q28 + Q1.27;
797     // yg = D0g + C0g;
798
799     assign val_C0g_a = {{(RBC0g+1){1'b0}},raw_val_C0g_a};
800     assign val_C0g_b = {{(RBC0g+1){1'b0}},raw_val_C0g_b};
801
802     // add sign extend to D0g
803     assign yg_next_raw_a = {D0g_a[FBD0g],D0g_a} + {val_C0g_a,1'b0};
804     assign yg_next_raw_b = {D0g_b[FBD0g],D0g_b} + {val_C0g_b,1'b0};
805
806     assign yg_next_raw_rtn_a = yg_next_raw_a[27:4] + yg_next_raw_a[3];
807     assign yg_next_raw_rtn_b = yg_next_raw_b[27:4] + yg_next_raw_b[3];
808
809     // [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)
810     // in case one (greater than 1), we force the highest representable value,
        ↪  1-2^-FB, and in case two we force the lowest
811     // representable value, 0. Also, there is the case when round to nearest extends
        ↪  above representable range (1-2^-FB) and therefore
812     // forced to grabbing original result before rtn.
813     assign yg_next_a = yg_next_raw_a[28] ? (yg_next_raw_a[29] ? 24'h000000 :
        ↪  24'hFFFFFF) : yg_next_raw_rtn_a[24] ? yg_next_raw_a[27:4] :
        ↪  yg_next_raw_rtn_a[23:0];      // w/ round to nearest
814     assign yg_next_b = yg_next_raw_b[28] ? (yg_next_raw_b[29] ? 24'h000000 :
        ↪  24'hFFFFFF) : yg_next_raw_rtn_b[24] ? yg_next_raw_b[27:4] :
        ↪  yg_next_raw_rtn_b[23:0];      // w/ round to nearest
815
816 // Range Reconstruction
817 // switch(Q)
818 //   case 0: g0 = yg_b;     g1 = yg_a;
819 //   case 1: g0 = yg_a;     g1 = -yg_b;
820 //   case 2: g0 = -yg_b;    g1 -yg_a;
821 //   case 3: g0 = -yg_a;    g1 = yg_b;
822                                          // 0              1
                                            ↪  2                3
823     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_a} + 1);
824     assign g1_next = ~Q_g[1] ? (~Q_g[0] ? {1'b0,yg_a} : ~{1'b0,yg_b} + 1) : (~Q_g[0]
        ↪  ? ~{1'b0,yg_a} + 1 : {1'b0,yg_b});
```

```
825
826 // Noise Creation
827     // n1 = f_g * g0;
828     // n2 = f_g * g1;
829     // Q4.19 = Q4.20 * Q24;
830     assign f_g = {1'b0,f};
831
832     assign f_g_extended = {{29{1'b0}},f_g,4'b0000};
833     assign g0_f_extended = {{33{g0_f[24]}},g0_f};
834     assign g1_f_extended = {{33{g1_f[24]}},g1_f};
835
836     assign n1_next_raw = f_g_extended * g0_f_extended;
837     assign n2_next_raw = f_g_extended * g1_f_extended;
838
839     assign n1_next = {n1_next_raw[57],n1_next_raw[51:29]} + n1_next_raw[28];
840     assign n2_next = {n2_next_raw[57],n2_next_raw[51:29]} + n2_next_raw[28];
841
842 always @(posedge reset or posedge clk)
843 begin
844     if (reset)
845     begin
846         // D4e
847         sel_D4e <= 0;
848         xl_e_D4e <= 0;
849         D4e <= 0;
850         exp_e_D4e <= 0;
851
852         // D3e
853         sel_D3e <= 0;
854         xl_e_D3e <= 0;
855         D3e <= 0;
856         exp_e_D3e <= 0;
857
858         // D2e
859         sel_D2e <= 0;
860         xl_e_D2e <= 0;
861         D2e <= 0;
862         exp_e_D2e <= 0;
863
864         // D1e
865         sel_D1e <= 0;
866         xl_e_D1e <= 0;
867         D1e <= 0;
868         exp_e_D1e <= 0;
869
870         // D0e
871         sel_D0e <= 0;
872         xl_e_D0e <= 0;
873         D0e <= 0;
874         exp_e_D0e <= 0;
875
876         // ye
```

```
877         sel_ye <= 0;
878         ye <= 0;
879         exp_e_ye <= 0;
880 
881         // e
882         e <= 0;
883         exp_e_e <= 0;
884 
885         //D0f
886         sel_D0f <= 0;
887         xl_f_D0f <= 0;
888         exp_f_D0f <= 0;
889         D0f <= 0;
890 
891         // yf
892         sel_yf <= 0;
893         exp_f_yf <= 0;
894         yf <= 0;
895 
896         // f
897         exp_f_f <= 0;
898         f <= 0;
899 
900         // D2g_a
901         sel_D2g_a <= 0;
902         xl_g_D2g_a <= 0;
903         D2g_a <= 0;
904 
905         // D2g_b
906         sel_D2g_b <= 0;
907         xl_g_D2g_b <= 0;
908         D2g_b <= 0;
909 
910         // D1g_a
911         sel_D1g_a <= 0;
912         xl_g_D1g_a <= 0;
913         D1g_a <= 0;
914 
915         // D1g_b
916         sel_D1g_b <= 0;
917         xl_g_D1g_b <= 0;
918         D1g_b <= 0;
919 
920         // D0g_a
921         sel_D0g_a <= 0;
922         xl_g_D0g_a <= 0;
923         D0g_a <= 0;
924 
925         // D0g_b
926         sel_D0g_b <= 0;
927         xl_g_D0g_b <= 0;
928         D0g_b <= 0;
```

```
929
930         // yg_a
931         sel_yg_a <= 0;
932         yg_a <= 0;
933
934         // yg_b
935         sel_yg_b <= 0;
936         yg_b <= 0;
937
938         // Q
939         Q_D2g <= 0;
940         Q_D1g <= 0;
941         Q_D0g <= 0;
942         Q_yg <= 0;
943         Q_g <= 0;
944
945         // g
946         g0 <= 0;
947         g1 <= 0;
948         g0_pipe0 <= 0;
949         g1_pipe0 <= 0;
950         g0_pipe1 <= 0;
951         g1_pipe1 <= 0;
952         g0_pipe2 <= 0;
953         g1_pipe2 <= 0;
954         g0_pipe3 <= 0;
955         g1_pipe3 <= 0;
956         g0_pipe4 <= 0;
957         g1_pipe4 <= 0;
958         g0_f <= 0;
959         g1_f <= 0;
960
961         // Valid Bit
962         valid_pipe <= 0;
963         valid_pipe0 <= 0;
964         valid_pipe1 <= 0;
965         valid_pipe2 <= 0;
966         valid_pipe3 <= 0;
967         valid_pipe4 <= 0;
968         valid_pipe5 <= 0;
969         valid_pipe6 <= 0;
970         valid_pipe7 <= 0;
971         valid_pipe8 <= 0;
972         valid_pipe9 <= 0;
973         valid_pipe_last <= 0;
974         valid <= 0;
975
976         // LGBMGNG
977         n1 <= 0;
978         n2 <= 0;
979     end
980     else
```

```
981     begin
982         // D4e
983         sel_D4e <= xm_e;
984         xl_e_D4e <= xl_e;
985         D4e <= D4e_next;
986         exp_e_D4e <= exp_e;
987 
988         // D3e
989         sel_D3e <= sel_D4e;
990         xl_e_D3e <= xl_e_D4e;
991         D3e <= D3e_next;
992         exp_e_D3e <= exp_e_D4e;
993 
994         // D2e
995         sel_D2e <= sel_D3e;
996         xl_e_D2e <= {1'b0,xl_e_D3e};            // unsigned to signed conversion for
            ↪  the signed arithmetic
997         D2e <= D2e_next;
998         exp_e_D2e <= exp_e_D3e;
999 
1000        // D1e
1001        sel_D1e <= sel_D2e;
1002        xl_e_D1e[FBxl_e-1:0] <= xl_e_D2e;       // signed to unsigned conversion
            ↪  (xl_e_D2e Always pos)
1003        D1e <= D1e_next[FBD1e-1:0];             // D1e_next was signed for
            ↪  arithmetic. but D1e always pos. ?todo;
1004        exp_e_D1e <= exp_e_D2e;
1005 
1006        // D0e
1007        sel_D0e <= sel_D1e;
1008        xl_e_D0e <= xl_e_D1e;
1009        D0e <= D0e_next;
1010        exp_e_D0e <= exp_e_D1e;
1011 
1012        // ye
1013        sel_ye <= sel_D0e;
1014        ye <= ye_next;
1015        exp_e_ye <= exp_e_D0e;
1016 
1017        // e
1018        e <= e_next;
1019        exp_e_e <= exp_e_ye;
1020 
1021        // D0f
1022        sel_D0f <= xm_f;
1023        xl_f_D0f <= xl_f;
1024        exp_f_D0f <= exp_f;
1025        D0f <= D0f_next;
1026 
1027        // yf
1028        sel_yf <= sel_D0f;
1029        exp_f_yf <= exp_f_D0f;
```

```
1030            yf <= yf_next;
1031
1032            // f
1033            exp_f_f <= exp_f_yf;
1034            f <= f_next;
1035
1036            // D2g_a
1037            sel_D2g_a <= xm_g_a;
1038            xl_g_D2g_a <= {1'b0,xl_g_a}; // unsigned to signed (add 0 SB)
1039            D2g_a <= D2g_next_a;
1040
1041            // D2g_b
1042            sel_D2g_b <= xm_g_b;
1043            xl_g_D2g_b <= {1'b0,xl_g_b}; // unsigned to signed (add 0 SB)
1044            D2g_b <= D2g_next_b;
1045
1046            // D1g_a
1047            sel_D1g_a <= sel_D2g_a;
1048            xl_g_D1g_a <= xl_g_D2g_a;
1049            D1g_a <= D1g_next_a;
1050
1051            // D1g_b
1052            sel_D1g_b <= sel_D2g_b;
1053            xl_g_D1g_b <= xl_g_D2g_b;
1054            D1g_b <= D1g_next_b;
1055
1056            // D0g_a
1057            sel_D0g_a <= sel_D1g_a;
1058            xl_g_D0g_a <= xl_g_D1g_a;
1059            D0g_a <= D0g_next_a;
1060
1061            // D0g_b
1062            sel_D0g_b <= sel_D1g_b;
1063            xl_g_D0g_b <= xl_g_D1g_b;
1064            D0g_b <= D0g_next_b;
1065
1066            // yg_a
1067            sel_yg_a <= sel_D0g_a;
1068            yg_a <= yg_next_a;
1069
1070            // yg_b
1071            sel_yg_b <= sel_D0g_b;
1072            yg_b <= yg_next_b;
1073
1074            // Q
1075            Q_D2g <= Q;
1076            Q_D1g <= Q_D2g;
1077            Q_D0g <= Q_D1g;
1078            Q_yg <= Q_D0g;
1079            Q_g <= Q_yg;
1080
1081            // g
```

```
1082            g0 <= g0_next;
1083            g1 <= g1_next;
1084            g0_pipe0 <= g0;
1085            g1_pipe0 <= g1;
1086            g0_pipe1 <= g0_pipe0;
1087            g1_pipe1 <= g1_pipe0;
1088            g0_pipe2 <= g0_pipe1;
1089            g1_pipe2 <= g1_pipe1;
1090            g0_pipe3 <= g0_pipe2;
1091            g1_pipe3 <= g1_pipe2;
1092            g0_pipe4 <= g0_pipe3;
1093            g1_pipe4 <= g1_pipe3;
1094            g0_f <= g0_pipe4;
1095            g1_f <= g1_pipe4;
1096
1097            // Valid Bit
1098            valid_pipe <= 1'b1;
1099            valid_pipe0 <= valid_pipe;
1100            valid_pipe1 <= valid_pipe0;
1101            valid_pipe2 <= valid_pipe1;
1102            valid_pipe3 <= valid_pipe2;
1103            valid_pipe4 <= valid_pipe3;
1104            valid_pipe5 <= valid_pipe4;
1105            valid_pipe6 <= valid_pipe5;
1106            valid_pipe7 <= valid_pipe6;
1107            valid_pipe8 <= valid_pipe7;
1108            valid_pipe9 <= valid_pipe8;
1109            valid_pipe_last <= valid_pipe9;
1110            valid <= valid_pipe_last;
1111
1112            // LGBMGNG
1113            n1 <= n1_next;
1114            n2 <= n2_next;
1115        end
1116    end
1117
1118    endmodule // LGBMGNG
```

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

I.2 32-bit Tausworthe URNG

```
1  // Author: Lincoln Glauser
2  // August 2017
3  module TAUS (
```

```
4             reset,
5             clk,
6             seed,
7             u0,
8             scan_in0,
9             scan_en,
10            test_mode,
11            scan_out0
12        );
13
14  input
15      reset,                      // system reset
16      clk;                        // system clock
17
18  input [31:0]
19      seed;                       // seed
20
21  output [31:0]
22      u0;                         // 32 bit U(0,1)
23
24  input
25      scan_in0,                   // test scan mode data input
26      scan_en,                    // test scan mode enable
27      test_mode;                  // test mode select
28
29  output
30      scan_out0;                  // test scan mode data output
31
32  reg [31:0]
33      s0,
34      s1,
35      s2;
36
37  wire [31:0]
38      b0,
39      b1,
40      b2,
41      s0_next,
42      s1_next,
43      s2_next;
44
45  //---------------------------------------------------------------------------//
46
47  assign b0 = (((s0 << 13) ^ s0) >> 19);
48  assign s0_next = (((s0 & 32'hFFFFFFFE) << 12) ^ b0);
49
50  assign b1 = (((s1 << 2) ^ s1) >> 25);
51  assign s1_next = (((s1 & 32'hFFFFFFF8) << 4) ^ b1);
52
53  assign b2 = (((s2 << 3) ^ s2) >> 11);
54  assign s2_next = (((s2 & 32'hFFFFFFF0) << 17) ^ b2);
55
```

```
56  assign u0 = s0 ^ s1 ^ s2;
57
58  always @(posedge reset or posedge clk)
59  begin
60      if (reset)
61      begin
62          s0 <= seed;
63          s1 <= seed;
64          s2 <= seed;
65      end
66      else
67      begin
68          s0 <= s0_next;
69          s1 <= s1_next;
70          s2 <= s2_next;
71      end
72  end
73
74  endmodule // TAUS
```

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

I.3 Coefficient Look up Table

```
1   module C2_COS (
2     input[6:0] sel,
3     output reg[13:0] val,
4
5     // DFT ports
6     input reset,
7     input clk,
8     input scan_en,
9     input scan_in0,
10    input test_mode,
11    output scan_out0
12  );
13
14  always @(sel) begin
15    case(sel)
16      0   : val[13:0] = 14'h1886;
17      1   : val[13:0] = 14'h1887;
18      2   : val[13:0] = 14'h188a;
19      3   : val[13:0] = 14'h188f;
20      4   : val[13:0] = 14'h1895;
21      5   : val[13:0] = 14'h189d;
```

```
22      6   : val[13:0] = 14'h18a6;
23      7   : val[13:0] = 14'h18b0;
24      8   : val[13:0] = 14'h18bc;
25      9   : val[13:0] = 14'h18ca;
26      10  : val[13:0] = 14'h18d9;
27      11  : val[13:0] = 14'h18ea;
28      12  : val[13:0] = 14'h18fc;
29      13  : val[13:0] = 14'h1910;
30      14  : val[13:0] = 14'h1925;
31      15  : val[13:0] = 14'h193c;
32      16  : val[13:0] = 14'h1954;
33      17  : val[13:0] = 14'h196e;
34      18  : val[13:0] = 14'h1989;
35      19  : val[13:0] = 14'h19a6;
36      20  : val[13:0] = 14'h19c4;
37      21  : val[13:0] = 14'h19e3;
38      22  : val[13:0] = 14'h1a04;
39      23  : val[13:0] = 14'h1a27;
40      24  : val[13:0] = 14'h1a4b;
41      25  : val[13:0] = 14'h1a70;
42      26  : val[13:0] = 14'h1a97;
43      27  : val[13:0] = 14'h1ac0;
44      28  : val[13:0] = 14'h1ae9;
45      29  : val[13:0] = 14'h1b15;
46      30  : val[13:0] = 14'h1b41;
47      31  : val[13:0] = 14'h1b6f;
48      32  : val[13:0] = 14'h1b9f;
49      33  : val[13:0] = 14'h1bd0;
50      34  : val[13:0] = 14'h1c02;
51      35  : val[13:0] = 14'h1c36;
52      36  : val[13:0] = 14'h1c6b;
53      37  : val[13:0] = 14'h1ca1;
54      38  : val[13:0] = 14'h1cd9;
55      39  : val[13:0] = 14'h1d12;
56      40  : val[13:0] = 14'h1d4c;
57      41  : val[13:0] = 14'h1d88;
58      42  : val[13:0] = 14'h1dc5;
59      43  : val[13:0] = 14'h1e04;
60      44  : val[13:0] = 14'h1e43;
61      45  : val[13:0] = 14'h1e85;
62      46  : val[13:0] = 14'h1ec7;
63      47  : val[13:0] = 14'h1f0a;
64      48  : val[13:0] = 14'h1f4f;
65      49  : val[13:0] = 14'h1f96;
66      50  : val[13:0] = 14'h1fdd;
67      51  : val[13:0] = 14'h2026;
68      52  : val[13:0] = 14'h2070;
69      53  : val[13:0] = 14'h20bb;
70      54  : val[13:0] = 14'h2107;
71      55  : val[13:0] = 14'h2154;
72      56  : val[13:0] = 14'h21a3;
73      57  : val[13:0] = 14'h21f3;
```

```
74      58  : val[13:0] = 14'h2244;
75      59  : val[13:0] = 14'h2296;
76      60  : val[13:0] = 14'h22e9;
77      61  : val[13:0] = 14'h233e;
78      62  : val[13:0] = 14'h2393;
79      63  : val[13:0] = 14'h23ea;
80      64  : val[13:0] = 14'h2442;
81      65  : val[13:0] = 14'h249a;
82      66  : val[13:0] = 14'h24f4;
83      67  : val[13:0] = 14'h254f;
84      68  : val[13:0] = 14'h25ab;
85      69  : val[13:0] = 14'h2608;
86      70  : val[13:0] = 14'h2666;
87      71  : val[13:0] = 14'h26c5;
88      72  : val[13:0] = 14'h2725;
89      73  : val[13:0] = 14'h2785;
90      74  : val[13:0] = 14'h27e7;
91      75  : val[13:0] = 14'h284a;
92      76  : val[13:0] = 14'h28ae;
93      77  : val[13:0] = 14'h2912;
94      78  : val[13:0] = 14'h2977;
95      79  : val[13:0] = 14'h29de;
96      80  : val[13:0] = 14'h2a45;
97      81  : val[13:0] = 14'h2aad;
98      82  : val[13:0] = 14'h2b16;
99      83  : val[13:0] = 14'h2b7f;
100     84  : val[13:0] = 14'h2bea;
101     85  : val[13:0] = 14'h2c55;
102     86  : val[13:0] = 14'h2cc1;
103     87  : val[13:0] = 14'h2d2d;
104     88  : val[13:0] = 14'h2d9b;
105     89  : val[13:0] = 14'h2e09;
106     90  : val[13:0] = 14'h2e78;
107     91  : val[13:0] = 14'h2ee7;
108     92  : val[13:0] = 14'h2f57;
109     93  : val[13:0] = 14'h2fc8;
110     94  : val[13:0] = 14'h3039;
111     95  : val[13:0] = 14'h30ab;
112     96  : val[13:0] = 14'h311e;
113     97  : val[13:0] = 14'h3191;
114     98  : val[13:0] = 14'h3205;
115     99  : val[13:0] = 14'h3279;
116     100 : val[13:0] = 14'h32ee;
117     101 : val[13:0] = 14'h3363;
118     102 : val[13:0] = 14'h33d9;
119     103 : val[13:0] = 14'h344f;
120     104 : val[13:0] = 14'h34c6;
121     105 : val[13:0] = 14'h353d;
122     106 : val[13:0] = 14'h35b4;
123     107 : val[13:0] = 14'h362c;
124     108 : val[13:0] = 14'h36a5;
125     109 : val[13:0] = 14'h371d;
```

```
126     110 : val[13:0] = 14'h3796;
127     111 : val[13:0] = 14'h3810;
128     112 : val[13:0] = 14'h3889;
129     113 : val[13:0] = 14'h3903;
130     114 : val[13:0] = 14'h397d;
131     115 : val[13:0] = 14'h39f8;
132     116 : val[13:0] = 14'h3a72;
133     117 : val[13:0] = 14'h3aed;
134     118 : val[13:0] = 14'h3b68;
135     119 : val[13:0] = 14'h3be4;
136     120 : val[13:0] = 14'h3c5f;
137     121 : val[13:0] = 14'h3cdb;
138     122 : val[13:0] = 14'h3d56;
139     123 : val[13:0] = 14'h3dd2;
140     124 : val[13:0] = 14'h3e4e;
141     125 : val[13:0] = 14'h3eca;
142     126 : val[13:0] = 14'h3f46;
143     127 : val[13:0] = 14'h3fc2;
144   endcase
145 end // always @(sel) begin
146
147 endmodule // C2_COS
```

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

I.4 Leading Zero Detector

```
1  // 47-bit Leading Zero Detector
2  module LZD_e (
3             reset,
4             clk,
5             val,
6             LZ_count,
7             scan_in0,
8             scan_en,
9             test_mode,
10            scan_out0
11         );
12
13 input
14     reset,                      // system reset
15     clk;                        // system clock
16
17 input [46:0] val;
18
```

```
19  output reg [5:0] LZ_count;
20
21  input
22      scan_in0,                   // test scan mode data input
23      scan_en,                    // test scan mode enable
24      test_mode;                  // test mode select
25
26  output
27      scan_out0;                  // test scan mode data output
28
29  always @(val)
30  begin
31      casez (val)
32          47'b1??????????????????????????????????????????????: LZ_count = 0;
33          47'b01?????????????????????????????????????????????: LZ_count = 1;
34          47'b001????????????????????????????????????????????: LZ_count = 2;
35          47'b0001???????????????????????????????????????????: LZ_count = 3;
36          47'b00001??????????????????????????????????????????: LZ_count = 4;
37          47'b000001?????????????????????????????????????????: LZ_count = 5;
38          47'b0000001????????????????????????????????????????: LZ_count = 6;
39          47'b00000001???????????????????????????????????????: LZ_count = 7;
40          47'b000000001??????????????????????????????????????: LZ_count = 8;
41          47'b0000000001?????????????????????????????????????: LZ_count = 9;
42          47'b00000000001????????????????????????????????????: LZ_count = 10;
43          47'b000000000001???????????????????????????????????: LZ_count = 11;
44          47'b0000000000001??????????????????????????????????: LZ_count = 12;
45          47'b00000000000001?????????????????????????????????: LZ_count = 13;
46          47'b000000000000001????????????????????????????????: LZ_count = 14;
47          47'b0000000000000001???????????????????????????????: LZ_count = 15;
48          47'b00000000000000001??????????????????????????????: LZ_count = 16;
49          47'b000000000000000001?????????????????????????????: LZ_count = 17;
50          47'b0000000000000000001????????????????????????????: LZ_count = 18;
51          47'b00000000000000000001???????????????????????????: LZ_count = 19;
52          47'b000000000000000000001??????????????????????????: LZ_count = 20;
53          47'b0000000000000000000001?????????????????????????: LZ_count = 21;
54          47'b00000000000000000000001????????????????????????: LZ_count = 22;
55          47'b000000000000000000000001???????????????????????: LZ_count = 23;
56          47'b0000000000000000000000001??????????????????????: LZ_count = 24;
57          47'b00000000000000000000000001?????????????????????: LZ_count = 25;
58          47'b000000000000000000000000001????????????????????: LZ_count = 26;
59          47'b0000000000000000000000000001???????????????????: LZ_count = 27;
60          47'b00000000000000000000000000001??????????????????: LZ_count = 28;
61          47'b000000000000000000000000000001?????????????????: LZ_count = 29;
62          47'b0000000000000000000000000000001????????????????: LZ_count = 30;
63          47'b00000000000000000000000000000001???????????????: LZ_count = 31;
64          47'b00000000000000000000000000000001???????????????: LZ_count = 32;
65          47'b00000000000000000000000000000001???????????????: LZ_count = 33;
66          47'b00000000000000000000000000000001???????????????: LZ_count = 34;
67          47'b00000000000000000000000000000001???????????????: LZ_count = 35;
68          47'b00000000000000000000000000000001???????????????: LZ_count = 36;
69          47'b00000000000000000000000000000001???????????????: LZ_count = 37;
70          47'b00000000000000000000000000000001???????????????: LZ_count = 38;
```

```
71          47'b00000000000000000000000000000000000000001??????: LZ_count = 39;
72          47'b00000000000000000000000000000000000000001??????: LZ_count = 40;
73          47'b00000000000000000000000000000000000000001?????: LZ_count = 41;
74          47'b00000000000000000000000000000000000000001????: LZ_count = 42;
75          47'b00000000000000000000000000000000000000001???: LZ_count = 43;
76          47'b00000000000000000000000000000000000000001??: LZ_count = 44;
77          47'b00000000000000000000000000000000000000001?: LZ_count = 45;
78          47'b00000000000000000000000000000000000000001: LZ_count = 46;
79          default: LZ_count = 47;
80      endcase
81  end
82  endmodule // LZD_e
```

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

I.5 LGBMGNG Test Vector Testbench

```
1   // Author: Lincoln Glauser
2   // August 2017
3   module test;
4
5   wire  scan_out0;
6
7   reg  clk, reset;
8   reg  scan_in0, scan_en, test_mode;
9
10  reg [31:0]
11      seed_a,
12      seed_b,
13      seed_c;
14
15  wire [23:0]
16      n1,
17      n2;
18
19  wire valid;
20
21  integer error_cnt;
22  integer index;
23  integer count;
24  parameter pcount = 20000000;    // Need to change for different TV sizes manually
25
26  reg [31:0]  cfg [0:3];
27  reg [31:0] memory_N1 [0:pcount-1];
28  reg [31:0] memory_N2 [0:pcount-1];
```

```
29 reg [24:0] error_index [1:pcount];  // 2^25 -> Max ~20 million
30 reg [24:0] error_index_index;
31
32 LGBMGNG top(
33         .reset(reset),
34         .clk(clk),
35         .seed_a     (seed_a),
36         .seed_b     (seed_b),
37         .seed_c     (seed_c),
38         .n1         (n1),
39         .n2         (n2),
40         .valid      (valid),
41         .scan_in0(scan_in0),
42         .scan_en(scan_en),
43         .test_mode(test_mode),
44         .scan_out0(scan_out0)
45     );
46
47 initial
48 begin
49     $timeformat(-9,2,"ns", 16);
50 `ifdef SDFSCAN
51     $sdf_annotate("sdf/LGBMGNG_tsmc065_scan.sdf", test.top);
52 `endif
53     clk = 1'b0;
54     reset = 1'b0;
55     scan_in0 = 1'b0;
56     scan_en = 1'b0;
57     test_mode = 1'b0;
58
59 // Uncomment option 1 for debug and option 2 for test_vectors
60 // Comment the option task call that is not being used.
61     // Option 1 is for singular a,b,c value debug
62     // For Debug set the a,b,c values as the seeds and view first sample set
63
64     seed_a = 1120679337;
65     seed_b = 3622075866;
66     seed_c = 3221225471;
67
68     // debug;  // Comment for TV and uncomment for Debug
69
70     // Option 2 for Test Vector Test
71
72     test_vector; // Comment for debug and uncomment for Test Vector support
73 end
74
75 always begin
76     #1.25 clk = ~clk ; // 400MHz
77 end
78
79 task test_vector ;
80     begin
```

```
81          $readmemh("TV_config.txt",cfg);
82          $readmemh("TV_in_N1.txt",memory_N1);
83          $readmemh("TV_in_N2.txt",memory_N2);
84
85          count = cfg[0];
86          seed_a = cfg[1];
87          seed_b = cfg[2];
88          seed_c = cfg[3];
89
90          error_cnt = 0;
91
92          $display("\nCount = %d\nseed_a = %d\nseed_b = %d\nseed_c =
            ↪  %d\n",count,seed_a,seed_b,seed_c);
93
94          @(negedge clk) ;
95          reset = 1'b1;
96          @(negedge clk) ;
97          reset = 1'b0;
98          @(negedge clk) ;
99
100         @(posedge valid) ;
101         index = 0;
102         @(posedge clk) ;
103         repeat (count)
104         begin
105             @(posedge clk) ;
106             if(memory_N1[index][23:0] != n1) begin
107                 $display("Error: Mismatch @ index = %d\nmdl N1: %d\nN1:
                    ↪  %d\n",index,memory_N1[index][23:0],n1);
108                 error_cnt = error_cnt + 1;
109             end
110             if(memory_N2[index][23:0] != n2) begin
111                 $display("Error: Mismatch @ index = %d\nmdl N2: %d\nN2:
                    ↪  %d\n",index,memory_N2[index][23:0],n2);
112                 error_cnt = error_cnt + 1;
113             end
114             index = index + 1;
115             if (index%(pcount/100) == 0)
116             begin
117                 $display("Progress: %d",index);
118             end
119         end
120
121         if(error_cnt == 0) begin
122             $display("\n** \"Success\" **\n\n");
123         end
124         else begin
125             $display("\n** TEST FAILED **\n\n%d errors found\n\n",error_cnt);
126         end
127         $display("\nCount = %d\nseed_a = %d\nseed_b = %d\nseed_c =
            ↪  %d\n",count,seed_a,seed_b,seed_c);
128         $finish;
```

```
129     end
130 endtask
131 task debug ;
132     begin
133         @(negedge clk) ;
134         reset = 1'b1;
135         @(negedge clk) ;
136         reset = 1'b0;
137         @(negedge clk) ;
138 
139         repeat (30)
140         @(posedge clk)
141         begin
142             $display("N1: %d\nN2: %d\n",n1,n2);
143         end
144         $finish;
145     end
146 endtask
147 endmodule
```

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