2010

An Efficient Implementation of an Exponential Random Number Generator in a Field Programmable Gate Array (FPGA)

Smitha Gautham
Virginia Commonwealth University

An Efficient Implementation of an Exponential Random Number Generator in a Field Programmable Gate Array (FPGA)

Smitha Gautham

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University

Virginia Commonwealth University, 2010

Director – Dr. James M. McCollum
Assistant Professor of Electrical & Computer Engineering

Dedication

In loving memory of my late grandmother

Nagarathnamma

Acknowledgements

First and foremost, I would like to thank my advisor Dr. James M. McCollum for his constant support and encouragement. I am grateful for his patient guidance and valuable suggestions throughout my research. I would like to thank Dr. Robert H. Klenke, Dr. Alen Docef and Dr. Vojislav Kecman for serving on my thesis committee. I want to thank Dr. Robert H. Klenke for sharing his group's random number generator code to interface with my exponential distribution module. I would also like to thank Dr. Roslyn Hobson for her support.

It was great pleasure to be a part of the VCU Computer Systems Laboratory. I would like to thank all my lab members who made working in the lab always enjoyable. I would like to specially acknowledge Jake Berlier for his help in understanding the Xilinx tools.

I would like to thank my friends Sai Priya, Sravanthy and Meena who made my stay at VCU very memorable.

I would like to thank my parents Latha and Muralidhar for their love and unwavering support. Their encouragement and blessing helped me reach where I am today. I fondly remember my late grand-mother who inspired me to study well and take up a professional career, waking me early in the morning with a hot cup of coffee during my exams. My grandfather was also a constant source of support. I would like to thank my in laws Anupama and Jayasimha for always being there for me. Their love and encouragement helped me a lot. I would like to thank my sister Sapna and brother Sharath with whom I can share all my aspirations and feelings. I would like to thank my sister in law Nirupama for her good wishes.

I am fortunate to have a husband like Atul, whose love and support motivates me to achieve anything I aspire for. I would like to thank him for being so special.

Abstract

Many physical, biological, ecological and behavioral events occur at times and rates that are exponentially distributed. Modeling these systems requires simulators that can accurately generate a large quantity of exponentially distributed random numbers, which is a computationally intensive task. To improve the performance of these simulators, one approach is to move portions of the computationally inefficient simulation tasks from software to custom hardware implemented in Field Programmable Gate Arrays (FPGAs).

In this work, we study efficient FPGA implementations of exponentially distributed random number generators to improve simulator performance. Our approach is to generate uniformly distributed random numbers using standard techniques and scale them using the inverse cumulative distribution function (CDF). Scaling is implemented by curve fitting piecewise linear, quadratic, cubic, and higher order functions to solve for the inverse CDF.

As the complexity of the scaling function increases (in terms of order and the number of pieces), number accuracy increases and additional FPGA resources (logic cells and block RAMs) are consumed. We analyze these tradeoffs and show how a designer with particular accuracy requirements and FPGA resource constraints can implement an accurate and efficient exponentially distributed random number generator.

Table of Contents

List of Tables

Page

List of Figures

Page

1. Introduction

This chapter discusses the motivation for accelerating the performance of exponentially distributed random number generation using FPGAs, key contributions made by this work, and provides an outline for the thesis.

1.1 Motivation

Many natural systems have properties, rates, or events that are exponentially distributed. The arrival of cars at traffic lights are exponentially distributed [1]. The rates at which phone numbers are called in a telemarketing system are exponentially distributed [1]. The inter-arrival time of packets in a network router and processing tasks on a computer are exponentially distributed [3]. Engineers model these systems to optimize the capacity and efficiency of their design, in turn using simulators that rely heavily on exponentially distributed numbers.

Scientists use the exponential distribution to model the radioactive decay time of a nuclear material [4]. Also, the decrease in the intensity of electromagnetic radiation in a medium follows an exponential distribution [4]. Models can then be used to calculate the thickness of a shield used in a nuclear reactor.

Similarly, the rate of a chemical reaction can be modeled by the exponential distribution. In pharmacology, for instance, the time it takes for a chemical to be metabolized also usually follows an exponential decay [5]. This has implications on the absorption of a medicine by the body as well as how it is spatially distributed to different parts of the body over time.

Because exponentially distributed random numbers are used to model such a wide variety of physical, biological, ecological, and behavioral systems, accelerating the performance of exponentially distributed random number generation stands to benefit many areas of scientific inquiry and engineering design.

Field Programmable Gate Arrays (FPGAs), which have been used to accelerate a wide variety of computationally intensive tasks, are an obvious platform for developing accelerated simulators that are either partially or fully implemented in hardware. The first step in constructing FPGA-accelerated simulators is to develop re-usable hardware elements that can be assembled into working solutions. Here we study a fundamental issue in the development of hardware accelerated simulators by developing an optimized and accurate exponential random number generator.

1.2 Contribution

McCollum et al [6] first designed a system capable of generating exponentially distributed random numbers using a piecewise linear approximation of the inverse CDF using a 256 entry data lookup table. This work is significantly expanded here by exploring the tradeoffs in FPGA resources and number accuracy by varying the data lookup table size and increasing the order of the approximation function. The study shows how a designer can generate the most accurate exponential numbers given a particular limitation on FPGA resources. The study also shows how a designer can use the least amount of FPGA resources to achieve a specific accuracy value. An exponential distribution generator is then implemented on a Virtex 5 FPGA capable of generating 100,000,000 exponentially distributed random numbers per second.

1.3 Outline

Introductions to probability theory, curve fitting, FPGAs, and prior work are presented in the Section 2. Details of how the random number generator was constructed and how performance and error analysis was conducted are given in the Section 3. Section 4 describes the results of the research and gives a detailed description of FPGA resource utilization for different order polynomials and different data look up table sizes. Section 5

concludes this thesis by summarizing the key results and discussing the future areas of research.

2. Background

Information needed to understand this thesis and description of previous work performed in the proposed research area is described in this section. First, probability density functions [7] and their importance in generation of random numbers that follow a particular distribution are described. The emphasis is on the exponential distribution. Second, floating point numbers [8] that are key to implementing any such distribution on hardware and operations involving them are discussed. Third, the use of curve fitting to approximate these distribution functions in various intervals is described. Finally, the work performed by other researchers is discussed.

2.1 Probability

The concept of probability can be explained with a simple example involving tossing of coins or rolling of dice. When a coin is tossed, there is equal chance of getting a head or tail. Thus the probability of getting a head or tail is 0.5. When a dice is rolled, there is one in six chance of getting a specific number between 1 and 6, the probability of any of these events is 0.1333.

The probability of an event occurring is thus the number of ways the event can occur divided by all of the possible ways of getting the outcomes. The probability of an event occurring is always less than or equal to 1. A probability of 1 implies that there is one hundred percent chance of occurrence of the event. A probability of zero implies that there is no chance of occurrence of the event.

For example, when a pair of dice is flipped, we can make the following observations.

- Number of ways of getting a sum of 2 or 12 is one in twelve
- Number of ways of getting a sum of 3 or 11 is two in twelve
- Number of ways of getting a sum of 4 or 10 is three in twelve

- Number of ways of getting a sum of 5 or 9 is four in twelve
- Number of ways of getting a sum of 6 or 8 is five in twelve
- Number of ways of getting a sum of 7 is six in twelve

It can be observed that all of these outcomes do not have equal probabilities. Suppose the sum is plotted on the x-axis and the probability of obtaining this sum is plotted on the y-axis, it gives a graph of the probability of each and every event (in this case the sum) as shown in Figure 2.1. A similar graph can be obtained when we toss a coin a hundred times and record the number of consecutive heads or tails. These are graphs of the probability density function (PDF) of the event.

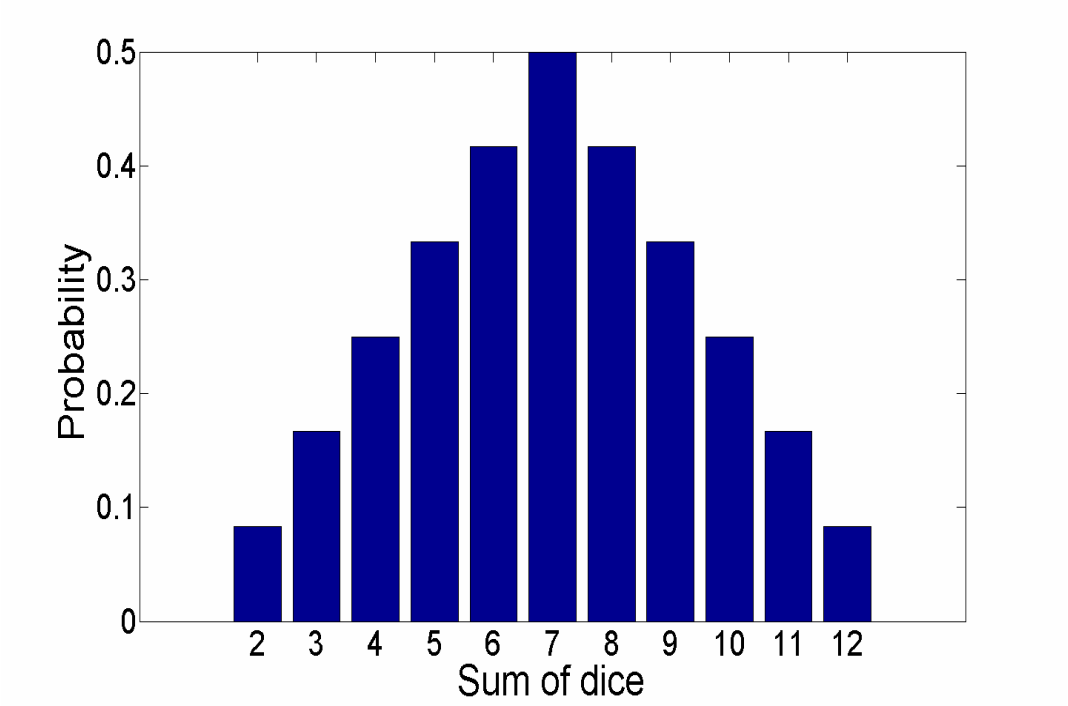

Figure 2.1: Probability density function for rolling of a dice.

2.1.1 Probability distributions

Many systems can be modeled using probability distributions. There are different standard forms of probability distribution functions, such as the exponential, normal, weibull, and uniform, that commonly mimic the behavior of real-world events. These distributions can be observed in various applications in our daily activities.

As discussed in the introduction section, radio-active decay time of a material and waiting time at a traffic light follow an exponential distribution, which will be discussed in later sections.

Marks scored by students in a class and performance of employees in a company follow the normal distribution, an example of which is shown in Figure 2.2.

.

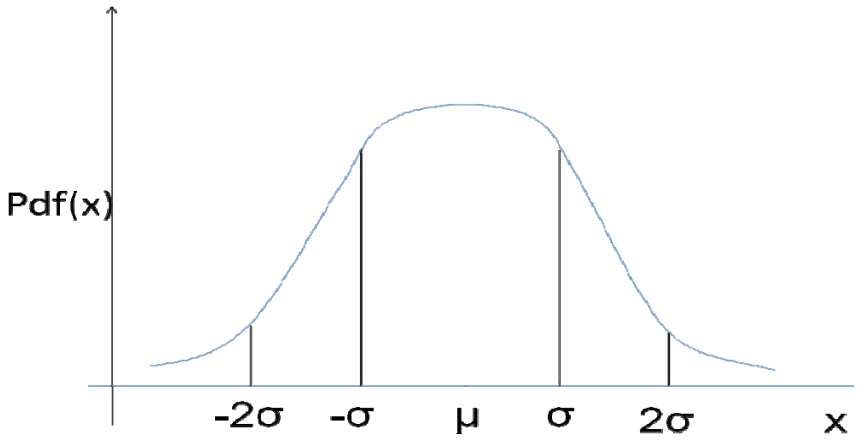

Figure 2.2. PDF of a normal distribution function.

2.1.2. Probability density function

The probability density function (PDF) represents the likelihood of a random number falling in a given sample space [9]. For example, when a coin is tossed a hundred times and the number of consecutive heads or tails are plotted on X axis and the probability of

getting the outcome is plotted on the Y axis, then we get a graph of the Probability density function of the event.

The PDF is described by the following equation

$$P(a \leq x \leq b) = \int_a^b f(x)dx \qquad (2.1).$$

For example, if the marks scored by students in a class is plotted and it gives us a probability distribution function as shown in Figure 2.3, then the probability of the marks falling between a and b where a and b are marks of the students can be calculated by using equation (2.1) where f(x) is the probability distribution function defining the marks scored by the students.

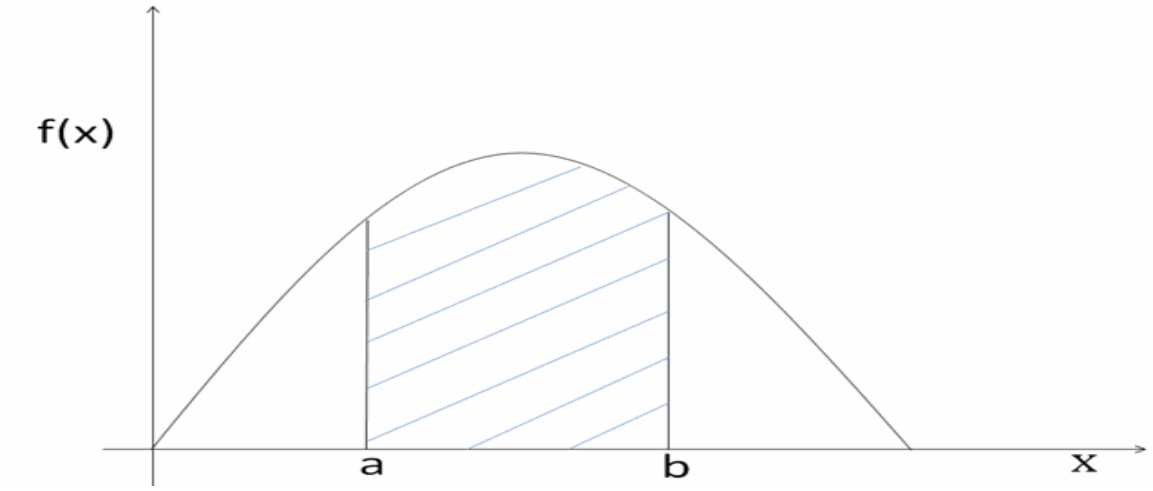

Figure 2.3. Probability of x falling between a and b, given PDF=f(x).

2.1.3 Cumulative distribution function

The cumulative distribution function is the integral of the probability density function. CDF (x) represents the probability of a number being less than or equal to x. Since, the total probability can never exceed one, the CDF can never take a value greater than one.

It usually asymptotes to 1 as x tend to infinity as shown in Figure 2.4. The cumulative distribution function of a number x can be described by the equation

$$F(x) = \int f(x)dx \tag{2.2}$$

where F(x) is the probability density function.

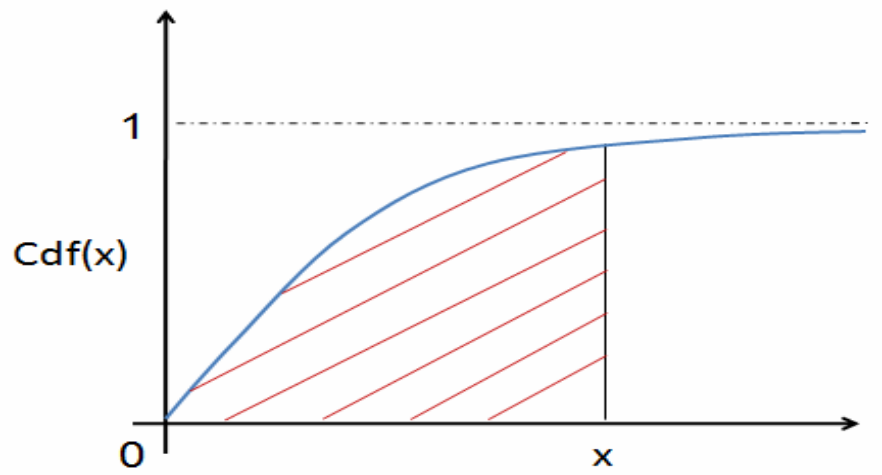

Figure 2.4. Cumulative distribution function (CDF).

2.1.4 Inverse cumulative density function

The Inverse CDF involves finding the value "x", so that the CDF (x) = p, where $0 \leq p \leq 1$. In other words given a certain probability p between 0 and 1, the inverse CDF (p) gives the number x described above.

It can be seen that if p is a uniformly distributed random number between 0 and 1, and the exponential CDF is chosen, then taking the inverse CDF yields exponentially distributed random numbers. If we choose the CDF to be Weibull, Normal, etc we would get Weibull or normally distributed random numbers by taking the inverse CDF of these functions respectively. This makes inverse CDF a very useful method to generate numbers with different distributions.

2.1.5 The Exponential distribution

The exponential distribution is used significantly in modeling statistical data and simulation. The probability distribution function (PDF) of exponential distribution is given by [7]

$$f(t) = \lambda e^{-\lambda t}, t \geq 0 \qquad (2.3).$$

where **λ** is a positive parameter. A plot of the probability density function for different values of **λ** is shown in Figure 2.5.

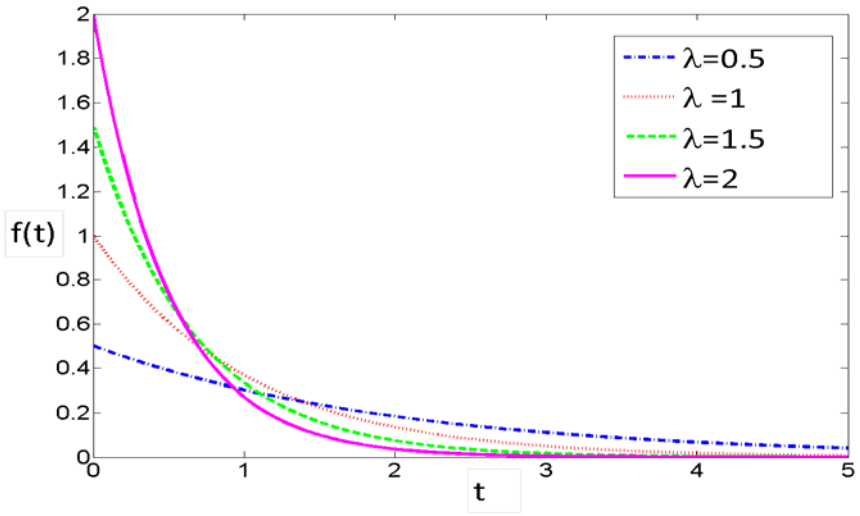

Figure 2.5. The PDF for exponential distribution for different values of λ.

The cumulative distribution function (CDF) of exponential distribution, which is the integral of the PDF, is given by [7]

$$F(t) = 1 - \lambda e^{-\lambda t}, t \geq 0 \qquad (2.4).$$

where λ is a positive parameter. A plot of the cumulative distribution function is shown in Figure 2.6.

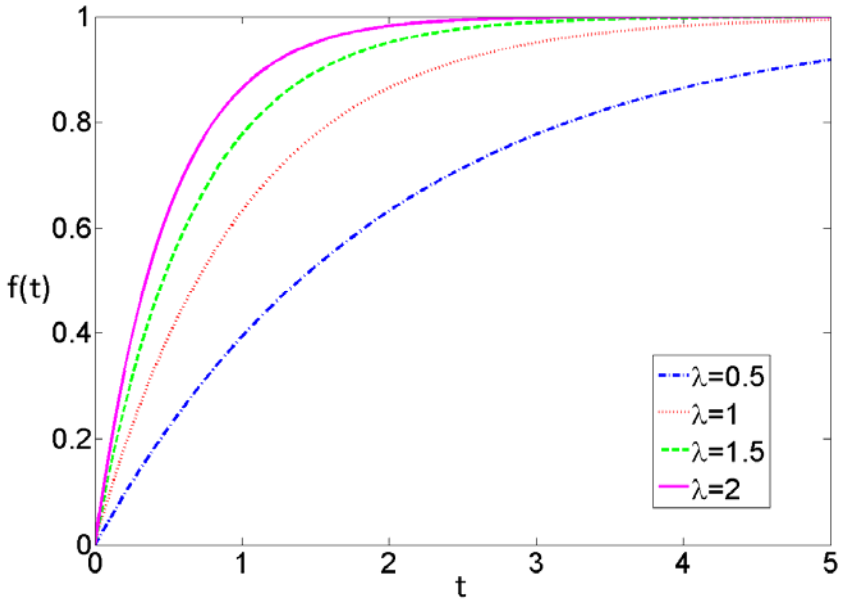

Figure 2.6. CDF of exponential distribution for different values of λ.

The inverse CDF can be calculated as $t = -\ln(1 - F(t))$ where F (t) is the CDF of the exponential distribution function.

The approach used in this thesis will be to generate uniformly distributed random numbers using standard FPGA methods, then scale the uniform random numbers to the exponential distribution using the inverse CDF. The challenge in implementing this approach in hardware will be to accurately and efficiently calculate the inverse CDF function in hardware.

2.2 Floating-Point Number Representation

This section describes floating-point number representation, floating-point addition and floating-point multiplication. There are data look up tables (DLUT) used in the design,

which have numbers in the integer format. These numbers are converted to floating point form to perform further operations. The description below describes real to float conversion. Floating point addition and multiplication are also performed in the design when we do curve fitting using different polynomials. Hence, these operations are also described in this section.

2.2.1 Floating point Numbers

Floating point numbers are used to represent very large and very small numbers. They can be represented in single precision or double precision format. In single precision format the number is represented as a 32 bit binary number. In double precision format the number is represented as a 64 bit binary number. In our research implementation, single precision floating point operations are performed.

A 32 bit floating point number has three parts. It has one sign bit, eight bits for the exponent and 23 bits for the mantissa. The sign bit is used to indicate if the number is positive or negative. Sign bit is zero for a positive and one for negative number. The exponent ranges from -126 to +127. All bits are zeros in the exponent for representing zero and all bits are ones for representing infinity or NaN. The mantissa part has 23 bits. There is an implicit 1 at the beginning of the mantissa. Therefore the mantissa has a precision of 24 bits [8].

Figure 2.7 shows a single bit floating point representation.

Figure 2.7. Floating point representation.

The following steps describe the procedure to convert a real number to a 32 bit floating point format [10]:

1. The integral and fraction part of the given number is converted to binary. The fraction part in converted to binary by repeated multiplication by 2.

2. 2^0 is appended at the end of the number. This does not change the value of the number as it is same as multiplying by 1.

3. The number is then normalized. Normalization of a number means, the decimal point is shifted such that there is only one number to the left of the decimal point. The exponent is adjusted such that the value of the number is not changed.

4. A bias is added to the exponent and the new exponent is placed in the exponent field of the floating point number. The bias is given by 2^{k-1} - 1 where k is the number of bits in the exponent field of the floating point number. For a 32 bit floating point number, the exponent field has 8 bits. Therefore the bias is 2^{8-1} -1= 127.

5. The sign bit is set for the given number. If the number is positive, sign bit is zero, otherwise the sign bit is one.

Ex: Convert 3.25 to floating point

- Convert to binary
  3 in binary is11
  0.25 in binary
  0.25 ×2 =**0**.50 0
  0.50×2 = **1**.0 1
- 3.25 in binary is 11.01
- Insert 2^0 11.01 * 2^0
- Shift decimal point and adjust exponent:
  1.101 * 2^1
- Normalize: 1+ 127=128 128 = 10000000

- Floating point representation of number 3.25 is
  0 10000000 10100000000000000000000

2.2.2 Floating point Multiplication

Single precision floating point multiplication consists of multiplying two 32 bit numbers with an 8 bit exponent and a 23 bit mantissa. The exponents are added and the mantissas are multiplied. Mantissa multiplication can be done by repeated additions. The result is then normalized.

2.2.3 Floating Point addition

Single precision floating point addition involves addition of two 32 bit floating point numbers. The two numbers must have the same exponent value to do the addition operation, or they must be shifted numerous times until they have the same exponent. The mantissas are then added.

2.3 Curve fitting

The exponential distribution implemented on hardware in this research involves curve fitting with different polynomials. This section gives a brief description of curve fitting.

2.3.1. Polynomial fit

A pool of data points can be defined by a mathematical equation or well defined curve by the method of curve fitting. Curve fitting can be done in many ways. One of the methods of curve fitting is to fit polynomial equations to the data set. The polynomial which best fits the data set is used to define the data points. Best fit depends on the nature of the data points. In this thesis curve fitting was done by polynomial curve fit. A set of data points are fit by different polynomials like linear, quadratic, cubic etc.

The equation for a linear polynomial is given by

$$y=ax+b \tag{2.5}$$

where x is a set of input data points. The co-efficient a and b are calculated so that ax+b is close to the output y.

The expression for higher order polynomials is given by

$$Y=a_0 + a_1\, x + a_2\, x^2 + a_3\, x^3 \ldots\ldots\ldots + a_{m-1}\, x^{m-1} + a_m\, x^m \tag{2.6}$$

where m is the order of the polynomial.

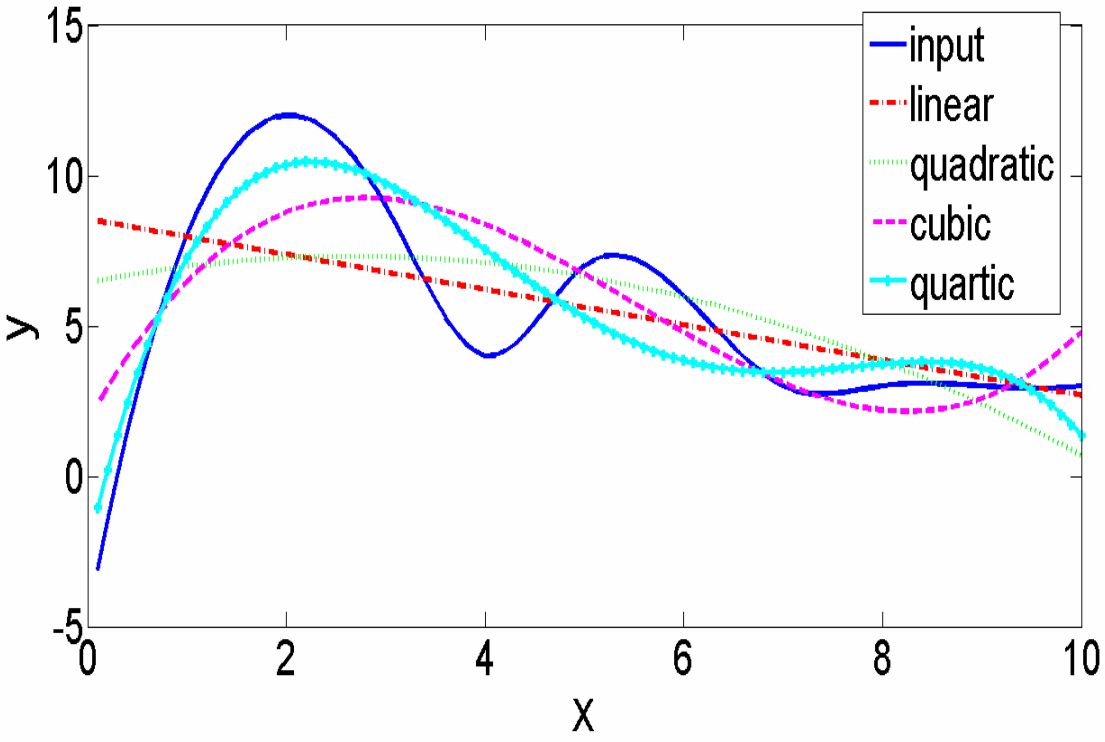

Figure 2.8. Polynomial curve fit for linear, quadratic, cubic and quartic polynomials.

Figure 2.8 is curve fit using linear, quadratic, cubic and quartic polynomials. The coefficients of the polynomials can be calculated numerically. This is easy for linear polynomials. But for higher order polynomials, numeric calculations become tedious. There are tools available for doing curve fit. The curve fitting toolbox available in Matlab was used in this research. "Polyfit" is used to do the curve fit in Matlab. "Polyval" is used to evaluate the curve fit polynomial and compare against input data and get an error estimate between the given curve and the polynomial curve fit.

2.3.2 Mean square error

The error between the given curve and the polynomial fit is squared and the mean of this square of the errors between the data points is taken. This gives the error estimate in the polynomial fit. The lower the MSE, the polynomial is considered a better fit.

2.4 Field Programmable Gate Arrays

Field programmable gate arrays (FPGA) are an array of logical blocks which can be configured based on the design requirements. The interconnections between the cells are field programmable and hence the name field programmable gate arrays. Each logical block consists of logic gates, flip flops, and multiplexers which can be programmed to perform the required function [11].

FPGAs have a lot of advantages over Application Specific Integrated Circuits (ASIC). They have more design flexibility since there is no need to draw the layout and fabricate the IC for each design [12]. The designs can be tested instantaneously on hardware without the need to wait for a fabricated IC to test the design. Alterations on the original design can be made easily. ASICs on the other hand take more time to manufacture and thereby take more time for the design to be tested. They are not flexible. Once an ASIC is fabricated, design changes cannot be made.

In this research, a Xilinx Virtex 5 FPGA was used. Virtex 5 FPGA consists of about 330,000 logic blocks and 16.4 Mbits of Block RAM [13]. Block RAM is large memory blocks configured to store data continuously. In Virtex 5 FPGA, BRAMS can be used as one 36Kb or two 18 Kb RAM. Each logical block in a Virtex 5 FPGA consists of 4 LUTs and 4 flip flops. It also has an IBM Power PC RISC processor core.

2.5 Previous work by other researchers

Random numbers have many applications in gaming, biological simulations etc. The exponential distribution module that is designed can be used with a random number generator to get exponentially distributed random numbers. This section gives a brief description on random numbers generation. A summary of work of others researchers on generating true random numbers and random numbers that follow different distributions is presented in this section.

2.5.1 True random numbers

Random numbers can be truly random or pseudo random. In case of pseudorandom numbers, the sequence of random numbers repeat after certain set of data. The time for the first repeat of data depends on the initial seed. In case of true random numbers, the next number that is generated cannot be predicted. Random numbers can be generated by hardware or software. Software random numbers generally have a code written which is initiated when random number generation is set. They are usually pseudo random. Pseudorandom numbers are preferable sometimes to true random numbers because you can repeat the simulations for a certain set of data and repeat the simulation for the same set of random data and verify the results [7]. Just the initial seed has to be changed to get a new set of random data.

True random numbers are used in cryptography so that secrecy is maintained [14].They are also used in gaming [14]. Thermal noise, photo electric effect is used as some of randomness in a hardware random number generator [14].

A hardware random number generator generated by Mitchum [14] was interfaced to the exponential distribution module to get a hardware exponential true random number generator. In Mitchum [14] a true random number generator is designed such that after generation of each number there are multiple paths that can be followed to generate the next successive number. It is difficult to predict the outcome of the generator as there are many ways that may be followed to get the number. Many digital components like ring oscillator, counters, shift registers, multiplexers and transposers were used in the design [14]. The ring oscillator has series of inverters connected back to back and the output of the last inverter is fed back to the first one. The period of the output wave of the ring oscillator is unpredictable and is the source of randomness in this paper. Divergent paths were introduced by using Linear feedback shift registers (LFSRs) of different lengths with ring oscillators to clock them. Thus true random numbers are generated and the randomness is tested using different methods.

2.5.2 Random numbers that follow different distributions

Wallace [15] method uses a conventional random number generator to generate a pool of random numbers. This is multiplied by an orthogonal matrix to get a new pool of random numbers. This is based on closure property where the new pool generated will also belong to the same random distribution as the old pool. Mixing of the numbers in the new pool is performed so that each number in the old pool can affect every element in the new pool after many successive transformations. This method may be used for different distributions but has been shown only for the exponential case. Here statistical errors get built up due to rounding off with successive transformations.

Ziggurat [16] is an efficient method for normal distributions but can be used for any distribution that increases monotonically to a peak and then decreases. It is based on the fact that most points lie in the mid-region of an exponential distribution and very few points lie in the tail of the curve. The points which fall in the mid region can be calculated easily and require less computation but points in the tail region require more

computation. A pipelined approach is used so that computation of the tail and the mid region can be done in parallel by hardware to generate a lot of mid-region points and few tail points. These are mixed together at the final step. The main disadvantage of this method is it cannot be used to for all kinds of distributions.

Thomas and Luk [17,18] develop a very general random number generator that is capable of generating random numbers with any kind of distribution. This uses a good mix of both hardware and software methods. The LUTs for different distributions are generated by software. A combination of different distributions that approximates the target distribution well is also chosen by software. The hardware generates a uniform random number and then uses the combination of distributions that has already been selected to generate random numbers that follow the desired distribution. The hardware also uses another random variable to pick one of the component distributions at random for calculating the transformation of each of the uniformly distributed random point. This is quick as it is only selecting a component distribution randomly and not using a combination of tables for calculating a given point. This method is applicable to different distributions. The method described in [17] uses equal weights for all selected tables, but the weight or probability of use of selected tables can be changed in method described in [18]. This allows it to approximate target distributions better. The drawback of this method is approximation of the target distribution by software is time consuming and complex. If the target distribution is known in advance then this is not an efficient method to generate random numbers.

McCollum [6] uses a LUT based method. The inverse CDF of a distribution is divided into many intervals. Each interval is curve fitted with linear polynomial and the coefficients are stored in a Data Look Up Tables (DLUTs). Thus random numbers can be generated by interpolation. This method can be used for all kinds of distributions.

In this research a similar approach to McCollum [6] is used. Here each interval is curve fitted with higher order polynomials like quadratic and cubic. It was found that the accuracy improves. By reducing the number of intervals the size of the DLUT can be

decreased while accuracy can be maintained by using higher order polynomial fits. The next chapter describes the detailed implementation of this scheme.

3. Implementation

The design and implementation of the exponential distribution on an FPGA based on using the inverse CDF was performed in two steps as described below:

1. In the first step Matlab is used to generate Data Look Up Tables (DLUT) and optimize the interval size. This is purely implemented in software, where the effect of the size of intervals on the mean square error as well as the size of the data look-up tables required is studied. This is used to select an optimum interval size for which the data lookup tables are generated for linear, quadratic and cubic curve fits.

2. In the second step, a hardware implementation of the inverse CDF, using the data look-up tables generated for the optimum interval size is performed. This is implemented on a Virtex 5 XC5VF70T-2FFG1136 FPGA.

These two steps are explained in detail in the "Matlab implementation" and "Hardware implementation" sections.

3.1 Matlab implementation

The aim of this research project is to generate the exponential distribution in hardware. To achieve this we have to implement the inverse CDF of the distribution. A 32 bit uniform number between 0 and 1 input to this inverse CDF will generate an exponentially distributed number as output.

The inverse CDF of exponential distribution function is given below [7].

$$t = - \ln (1-F(t)) \qquad (3.1)$$

This involves implementing the natural log in hardware, which is difficult. Hence, a DLUT (Data look up table) based method is used implement the natural log in hardware. The values of the output for all possible inputs can be stored in a DLUT. Then, for a 32 bit input there are 2^{32} i.e. 4,294,967,296 possible output combinations to be stored in the DLUT. This would require a huge amount of memory for implementation. Hence, a more efficient way is proposed. Instead of storing all the output values in DLUT, the inputs are divided into intervals. The inverse CDF of the numbers in each interval is calculated and curve fitted with polynomials. The coefficients of the polynomials are stored in LUT. By choosing the coefficients for the correct interval from the DLUT and interpolating, the natural log of a number can be calculated.

Figure 3.1 shows the inverse CDF, curve fitted by a linear polynomial Ax+B where A and B are polynomial coefficients. Input numbers from 0 to 100 were curve fitted by linear polynomial and coefficients A= 3.951 and B= -0.808 were obtained.

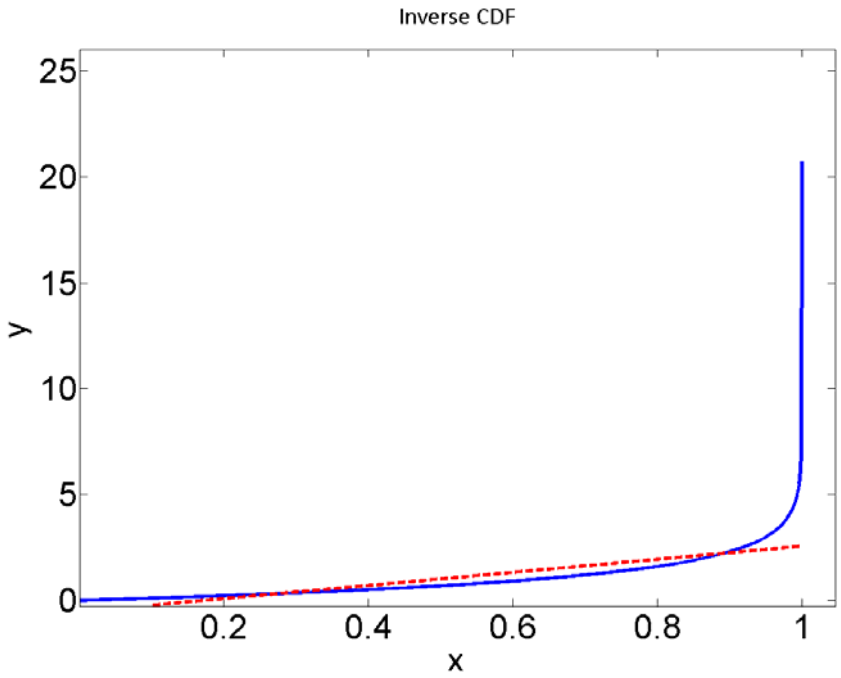

Figure 3.1. Inverse CDF curve fitted by linear polynomial.

Figure 3.2 shows the inverse CDF divided into two intervals and then curve fitted by a linear polynomial Ax+B and quadratic polynomial Ax^2+Bx+C where A,B and C are polynomial coefficients. Polynomial coefficients obtained in each interval are shown below.

For the first interval:

Linear fit : A = 1.365 B= -0.034. The linear equation is (1.365 × x) +(-0.034)

Quadratic fit A= 0.935 B= 0.897 and C= 0.004. The quadratic equation is

(0.935 × x^2) +(0.897 × x) +(0.004)

For the second interval:

Linear fit: A= 7.838 B= -4.024. Therefore linear equation is (7.838 × x) +(-4.024)

Quadratic fit: A= 37.451 B= -48.339 and C= 16.245. The quadratic equation is

(37.451 × x^2) + (-48.339 × x) +16.245

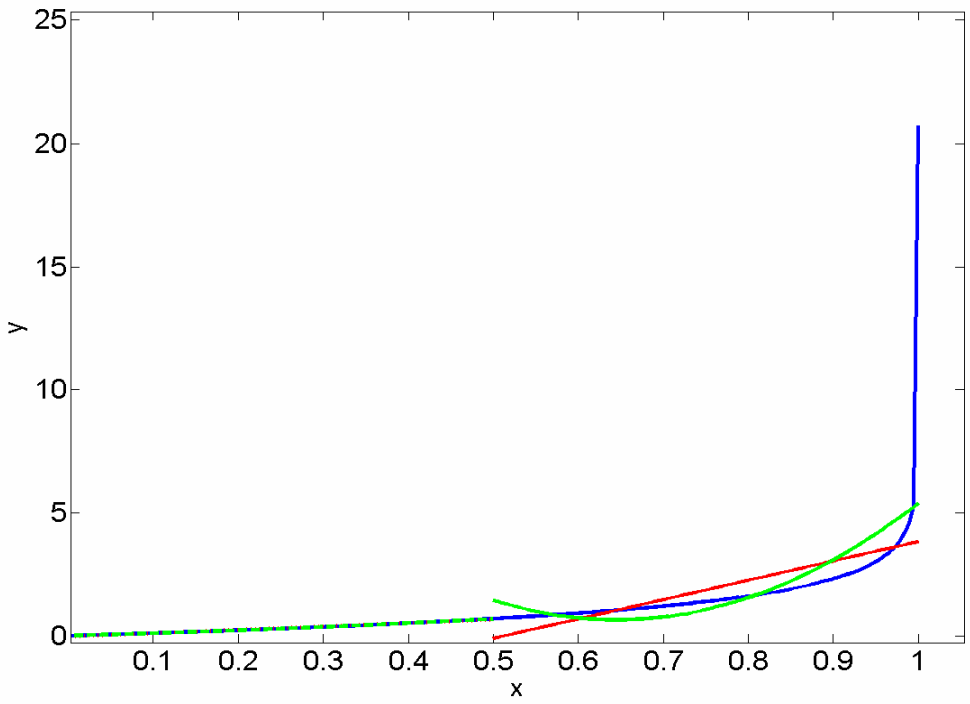

Figure 3.2 . Inverse CDF function divided into 2 intervals and curve fitted with linear and quadratic polynomials

The coefficients are stored in Data Look Up Tables (DLUTs). Thus for an input x, the inverse exponential CDF is calculated by fetching the coefficient values from the LUTs and interpolating using the polynomials.

Table-3.1 shows that the DLUT entries increases exponentially with the bit size corresponding to the number of intervals. For example, if there are 8-bits that are allocated to the number of intervals, then we need 2^8= 256 DLUT entries. In fact, we will need a multiple of this number depending on the order of the interpolation polynomial used.

Bit-size allocated to Number of Intervals	**Number of entries in LUT**
4	16
8	256
12	4096
16	65536
20	1,048,576
24	16,777,216
28	268,435,456
32	4,294,967,296

Table3.1. Increase in LUT size as the number of intervals increases.

The intervals are curve fitted with linear, quadratic and cubic polynomials. Curve fitting was done using polyfit () and the interpolations are performed using polyval () function in Matlab.

Table 3.2 shows the DLUT size for linear, quadratic and cubic polynomials for different interval lengths the inverse CDF is divided into.

Intervals	Linear	Quadratic	Cubic
4	32	48	64
8	512	768	1024
12	8,192	12,288	16,384
16	131,072	196,608	262,144
20	2,097,152	3,145,728	4,194,304
24	33,554,432	50,331,648	67,108,864
28	536,870,912	805,306,368	1,073,741,824

Table 3.2 . Scaling of DLUT entries with number of intervals and order of polynomial fit.

From Table 3.2 we can see that as the order of the polynomial increases the DLUT entries also increases as higher order polynomials will have more coefficients to be stored in the DLUT. As the number of intervals increase, there is a large increase in DLUT size.

Linear, quadratic and cubic polynomials used are:

A_1x+B_1, (3.2)

$A_2x^2+B_2x+C_2$ (3.3)

$A_3x^3+B_3x^2+C_3x+D_3$ (3.4)

Thus, as we use higher order polynomials to curve fit, the number of coefficients increase. Thus LUT size also increases. The number of intervals that the inverse CDF is divided into is also important. From Table 3.2 we can see that as the interval size increases, there will be bigger LUTs for coefficients in each interval.

The Matlab code which implements the design is given below.

```
total_points=2^32;
no_intervals=2^8;
points_interval=2^24;
for j=1: No_intervals;
x=linspace ((j-1)/ No_intervals, (j/ No_intervals)-(1/ total_points),
points_interval);
y=-log (1 - x); %Exponential distribution
poly1=polyfit(x,y,1);
poly2=polyfit(x,y,2);
poly3=polyfit(x,y,3);
end
```

- The input is a 32 bit uniform number. Therefore there are 2^{32} points. We chose to divide them into 2^8 intervals (one particular case). Therefore there are 2^{24} points in each interval.
- The for loop iterates from 1st to the 2^8th interval.
- "x" holds the starting point and end point for each interval. Matlab function linspace generates an array of values from the start point to the end point.
- For all points in each interval, the inverse CDF is calculated. "y" holds the inverse CDF values for each interval.
- Points in each interval are curve fitted by the polyfit () function in Matlab. Linear, quadratic and cubic polynomials are used to fit the intervals.

It should be noted that in dealing with natural log function, ln (0) $\rightarrow\infty$. In case of the inverse CDF of exponential distribution the log (1-x) term tends to infinity when x tends to one. To avoid this situation the code was modified to bypass the last bit by subtracting $1/2^{32}$ from the each interval so "1" is not included in the last interval.

Another important consideration is the Mean Square error (MSE). This is used to measure the accuracy of the curve fitted polynomial.

Figure 3.3 shows the inverse CDF fitted by a linear polynomial. We can see that there is a difference between actual curve and the curve fit, which results in an error. Mean square error is calculated by the mean of square of the difference between actual and curve fit.

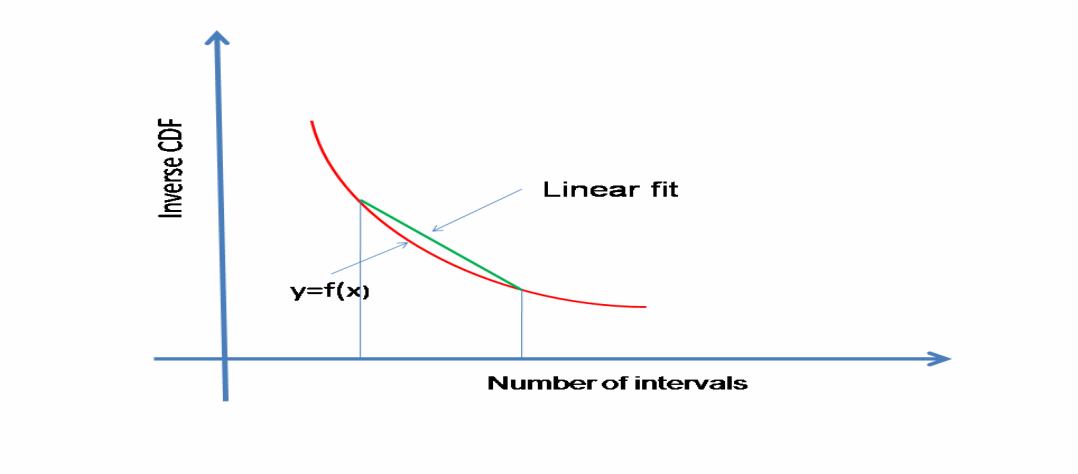

Figure 3.3. Schematic to show how polynomial fit in an interval introduces an error.

Our objective is to generate an optimum DLUT(Data look up table) size, that minimizes mean square error. When the number of intervals is increased the mean square error decreases but the DLUT size increases. When the number of intervals is decreased the error increases while DLUT size decreases. A trade off has to be achieved between the MSE and the DLUT size. Using higher order polynomials can improve the accuracy but would require more entries in the DLUT.

Figure 3.4 shows the MSE for linear, quadratic, cubic, quartic (4^{th} order) and quintic (5^{th} order) polynomials.

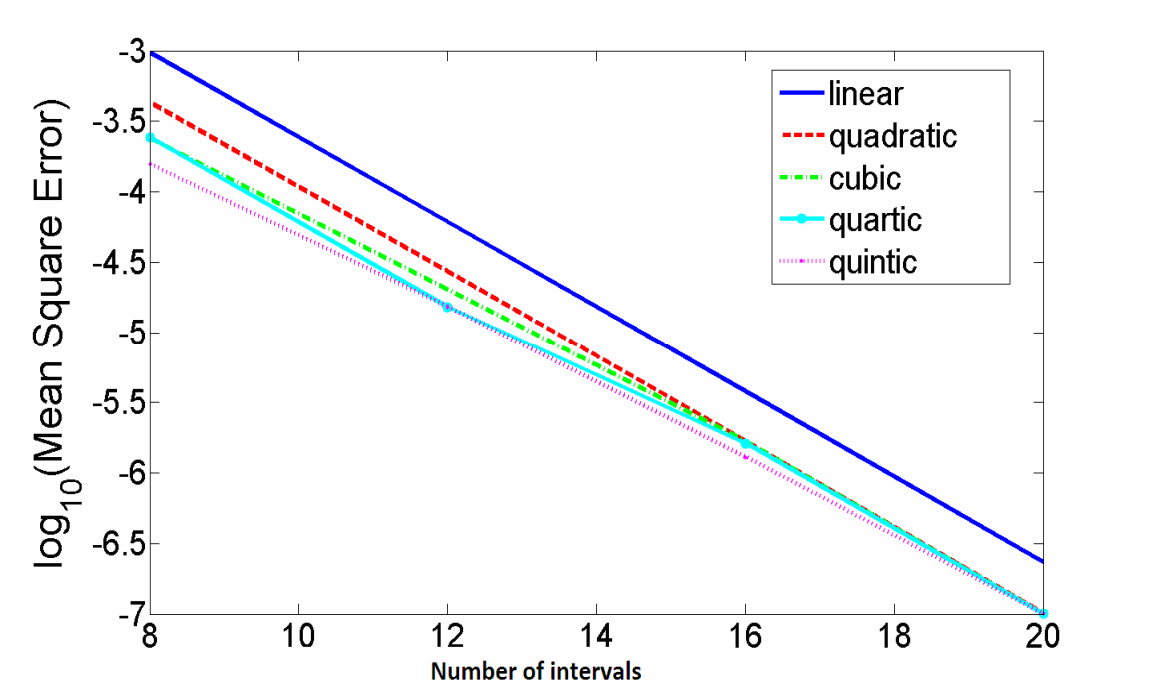

Figure 3.4 .MSE vs. number of intervals (2^n) for linear, quadratic and cubic polynomials for inverse CDF of exponential distribution.

We observe that the MSE decreases as the number of intervals increases. As we fit higher order polynomials the mean square error does not decrease significantly. We can see from the plot that after certain intervals, the MSE of quadratic and cubic polynomials are very close. There is no significant decrease in MSE as we move to higher order polynomials like quartic and quintic for larger interval lengths. From this we can conclude that we cannot see considerable increase in accuracy as we fit higher order polynomials. It may be better to fit a quadratic instead of cubic for certain interval lengths as the MSE are close and quadratic has less coefficients to store in LUTs.

For our design interval length of 8 was chosen. This gives a DLUT size of 256 entries. It also gave a sufficient accuracy. For example if we want to get a MSE of about $0.5*10^{-3}$ then we can choose a linear fit with interval size of 2^{10}. From the Figure 3.4 we can see that even a cubic fit of interval length 2^8 gives the same accuracy. This would reduce the

DLUT size for the same accuracy. Hence, a cubic is a better fit for this case. There may be other cases where higher order polynomials may not provide the best fit.

A similar approach can be used to implement any distribution function on hardware. We have to just change the DLUTs for the distribution in the Matlab code that generates the coefficients.

3.2 Hardware implementation

In this section the implementation of inverse CDF of exponential distribution on an FPGA is discussed. The design was implemented on a Virtex 5 XC5VF70T-2FFG1136 FPGA and verified in Modelsim simulator. The exponential distribution module was interfaced with a true random number generator to produce an exponentially distributed random number generator.

3.2.1 Hardware implementation of inverse CDF for generating exponential numbers

The DLUTs generated in Matlab are in real format. These variables of type "real" can be simulated in Modelsim but is not synthesizable on Xilinx ISE. They have to be converted to integer or floating point form for synthesis. A VHDL code was written to convert all the real LUTs to integer LUTs that can be used in Xilinx.

Here is a part of the VHDL code written to convert the LUT contents of type "real" to type "integer". Here linA has the coefficient of linear fit from Matlab. These are converted to floating point by the to_float function. The floating point numbers are converted to "std_logic_vector" type and then to type "integer" as there is no direct conversion function from "float" to "integer". The integer coefficients are stored in linAINT. These can be stored in DLUTs and are synthesizable. The DLUT (Data Look Up Table) values are stored in integer format in the design. They can also be stored in floating point format. Conversion of the DLUT numbers to floating point using the to_float function can be avoided if the DLUT numbers are already stored in float format.

The design was implemented using floating point DLUTs and there was no significant decrease in the logic utilization by the design. Here is a part of the code which implements the above design:

```
process(clk)
 begin
 if clk='1' and clk' event then
   for index in 0 to 255 loop
     linAfloat(index)<=To_float(linA(index)); --converts from real to float
     linASTD(index)<=to_slv(linAfloat (index)); -- converts from float to
std_logic_vector
     linAINT(index)<=conv_integer(linASTD (index)); std_logic _vector to
integer
    end loop;
   end if;
end process;
```

Implementation of exponential distribution on hardware involves floating point operations. A floating point library created by David Bishop[19,20] was used to perform the floating point addition and multiplication operations. The to_float function used in the code above which converts a number of any type to a single precision floating point number is also part of the floating point library.

The design was implemented in VHDL. The algorithm of the design is as follows. The input is a 32 bit uniformly distributed number less than 1. The higher 8 bits of the input is used as an index to get the polynomial coefficients from the DLUT. This would give the interval that the input number falls into. The coefficients that correspond to this particular interval are used to perform the interpolation to get the inverse CDF. Thus the inverse CDF to convert a number between 0 and 1 to an exponentially distributed number is realized.

Interpolation was performed using linear, quadratic and cubic polynomials. Figures 3.5, 3.6 and 3.7 describe the design operations. The upper 8 bits of the 32 bit input number is used to get the coefficients from the data look up table. Depending on the polynomial

chosen for interpolation, the DLUT coefficients are selected. Interpolation using linear polynomial will have two DLUTs, quadratic will have three DLUTs, and cubic will have four DLUTs and so on. Interpolation is performed after the coefficient values are fetched from the DLUTs. Cubic interpolation will require the most logic followed by quadratic and then linear. Thus the higher the order of the interpolation polynomial, the more logic is needed. For example, the cubic polynomial can be represented as $A_3x^3+B_3x^2+C_3x+D_3$ and the linear polynomial can be represented as A_1x+B_1 where A_3, B_3, C_3 and D_3 are the cubic polynomial coefficients, $A_1$and B_1 are linear polynomial coefficients and x is the 32 bit input number. All of the numbers are in floating point format.

Cubic interpolation involves finding the cube and square of the input floating point number, which requires three floating point multiplications and three floating point additions. Whereas, linear interpolation involves one floating point multiplication and one floating point addition. Thus, the higher the order of the polynomial used for curve fitting, the more logic would to be implemented and more DLUTs would be required. Thus depending on the memory available to store the DLUTs and to implement the logic, a suitable polynomial has to be selected. The accuracy needed must also be taken into consideration while selecting the polynomial order. As shown in the Matlab implementation section, there may not be significant improvement in accuracy for certain interval sizes as we move to higher order polynomials for example choosing 4^{th} order instead of a 3^{rd} order may not reduce the interpolation error significantly.

Block diagrams for the linear, quadratic, and cubic implementations are given in Figures 3.5, 3.6 and 3.7.

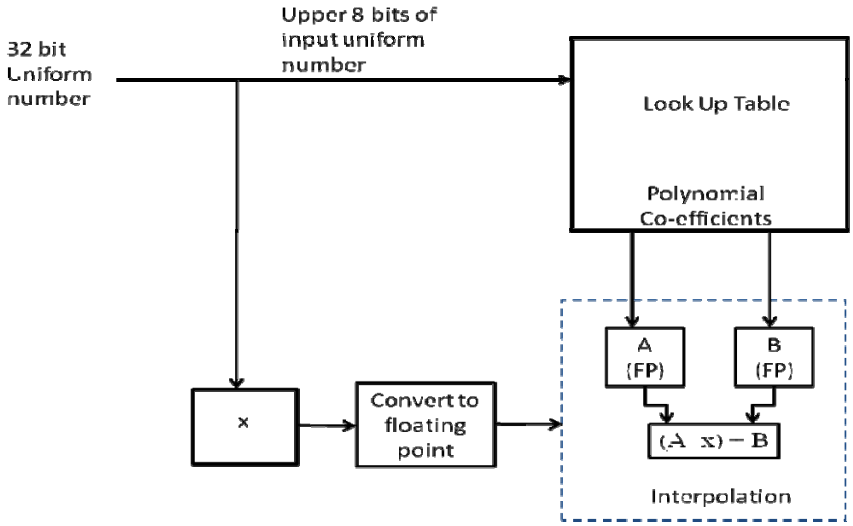

Figure 3.5. Block diagram representation linear interpolation.

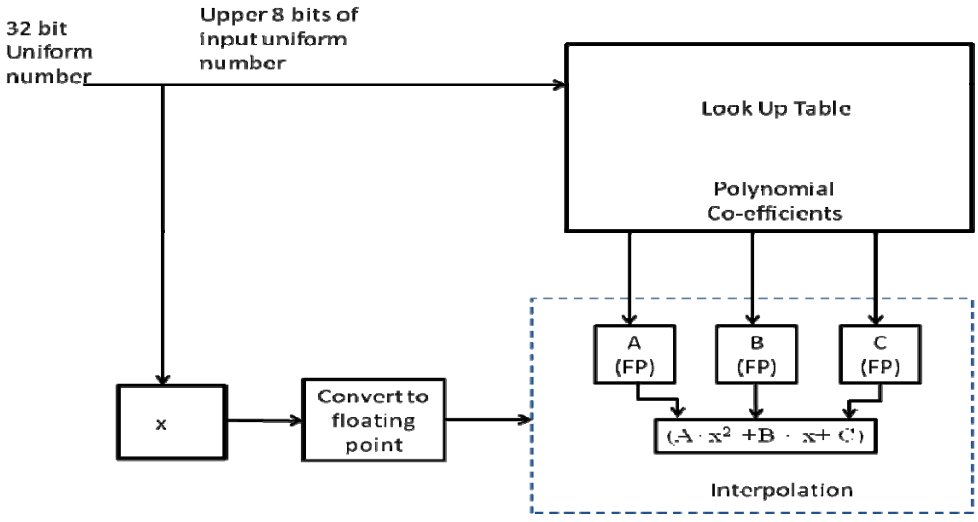

Figure 3.6. Block diagram representation quadratic interpolation.

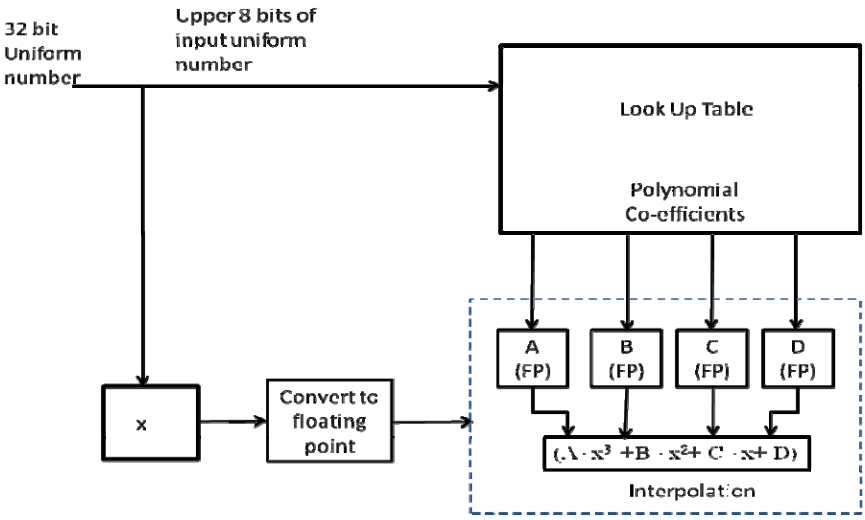

Figure 3.7. Block diagram representation cubic interpolation.

3.2.2 True Random Number Generator

The exponential distribution that is implemented on hardware is interfaced to a true random number generator to get an exponentially distributed random number. The true random number generator is designed by Mitchum and Klenke [14]. The input to the exponential distribution module is taken from the true random number generator output.

4. Results

The previous sections discussed selection of the optimum interval size for generating data look up tables (DLUTs) for linear, quadratic and cubic fits. To achieve this, the effect of the interval size on mean square error (MSE) for different polynomial fits was determined. Then the effect of the interval size on memory required for DLUTs for different polynomials was also studied. The tradeoff between the accuracy and memory required for DLUTs resulted in the selection of an optimum interval size and polynomial order. The hardware implementation of the exponential distribution was also discussed.

In this section, the results of the implementation of the design in the previous section (summarized above) on Virtex 5 XC5VF70T-2FFG1136 FPGA are shown. The speed achieved by the design and memory utilized for design for different interval lengths are described.

4.1 Results and verification

The inverse exponential distribution of a number is given by

$$Y = -\ln(1-x); \tag{4.1}$$

where x is the input 32 bit number and y is the 32 bit exponential output. An example of the execution of this function by the actual FPGA is shown to confirm that the implementation was correct. The FPGA output is shown below.

Input value: 80000000	**FPGA Output: 3F317346**	**Actual value: 3F317218**
Input value: 567854	**FPGA Output: 3AAD0FC3**	**Actual value: 3AAD0DE4**
Input value: 0	**FPGA Output: 0**	**Actual value: 0**

The input is in the hexadecimal (hex) format. This input hex number is converted to 32 bit binary number and a decimal point is appended in the beginning of the number. Thus the input is always less than 1. The FPGA output is found by implementing the equation in hardware. The actual value gives the correct value of the output. We can see that the FPGA output is almost equal to the actual output. The difference between the FPGA output and the actual output is due to the interpolation error. The results shown above were obtained by performing linear polynomial curve fit with a interval length of 8.

E.g.: For input 800000000

- 32 Binary conversion gives 10000000000000000000000000000000
- After appending decimal point, the number becomes 0.10000000000000000000000000000000 i.e. 0.5 in decimal
- Output = -ln(1-0.5)=0.6931471806
- Converting output to floating point, we get 3F317218. The output was converted to floating point for verification using IEEE floating point conversion [21].
- The FPGA output is slightly different (3F317346) due to small error caused by the curve fit.

4.2 Resource Utilization

The Virtex 5 XC5VFX70T is a Xilinx Power PC FPGA. It consists of about 330,000 logical cells, 1200 user I/Os and 18 Mb of block RAM [22]. FPGA resource utilization for linear, quadratic and cubic polynomial fits was studied. The design is mapped on the logical blocks and the DLUTs used in the design are mapped on to the BRAMs on the FPGA. The logic block utilization is almost the same for a given polynomial. However, the number of block RAMs used increases with the interval size. When the DLUT sizes are varied, only the BRAM utilization changes, but the logic usage remains the same.

Table 4.1 shows the FPGA device utilization for linear, quadratic and cubic polynomial for different interval sizes. Figure 4.1 shows the graphical representation for the same.

Log (LUT size)	Linear			Quadratic			Cubic		
	slice registers used	slice LUTs used	Block RAMs used	slice registers used	slice LUTs used	Block RAMs used	slice registers used	slice LUTs used	Block RAMs used
6	265	4606	2	393	8097	3	517	11158	4
8	261	4708	2	394	7964	3	519	11158	4
10	267	4809	2	395	8563	3	517	12614	4
12	270	4820	8	398	8076	11	519	12076	15
14	273	4607	32	400	7346	48	525	11158	64
16	271	4811	122	403	8192	192 (error)	518	12332	242 (error)

Table 4.1. Resource utilization for various LUT sizes.

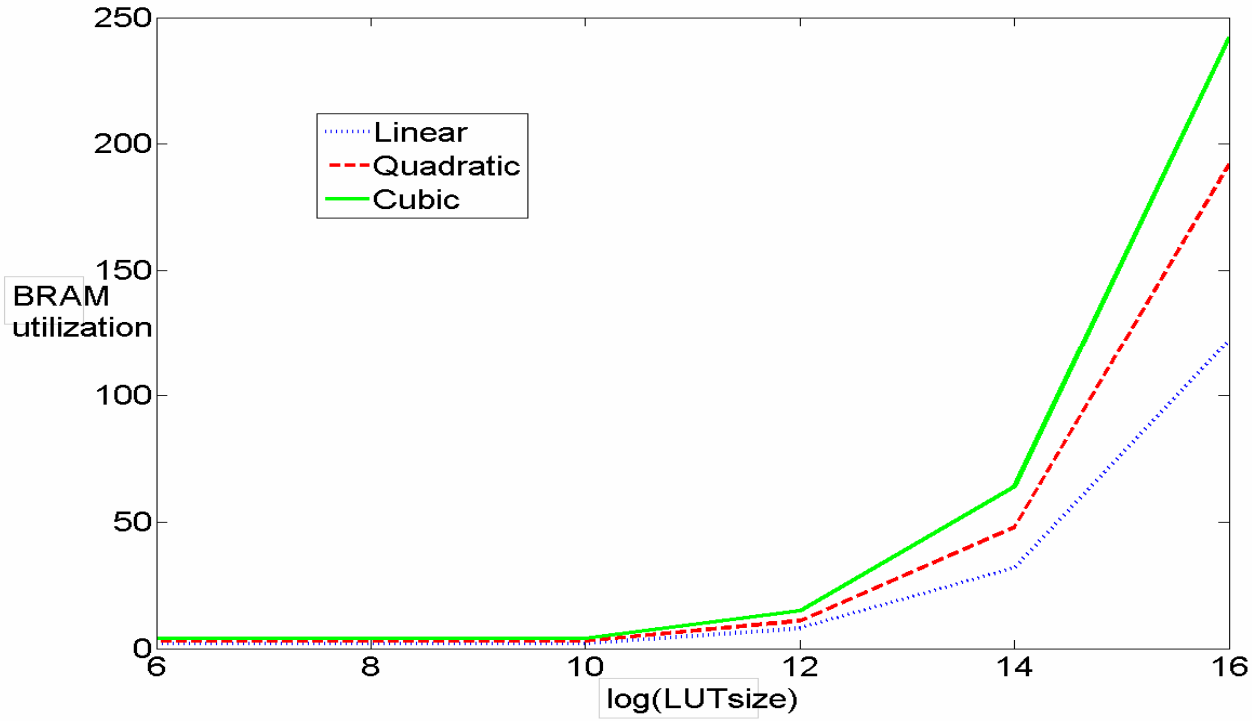

Figure 4.1. Graphical representation of utilization of BRAM by linear, quadratic and cubic polynomials**.**

From Figure 4.1 it can be seen that the device utilization increases as we go from linear to quadratic to higher order polynomials. The block RAM size increases as we move to higher order polynomials and as the DLUT size increases. The BRAM size reaches the maximum limit of the Virtex 5 FPGA for LUT size of 2^{16} entries in case of quadratic and cubic fit. Thus as we fit higher order polynomials, the maximum DLUT size that can be used in the design decreases. Similarly, for a smaller DLUT size, higher order polynomial fits can be used to fit the given data.

The logic blocks utilized by the design for the exponential distribution module are significantly less than the available logic resources on the FPGA. However, as mentioned earlier the DLUT limits are reached very fast (For 2^{16} entries). The design is easily fit in the FPGA for all the linear, quadratic and cubic polynomials. The cubic fit occupies only 24% of the available logic. Thus higher order polynomials can be used if necessary to get better accuracy as there are lot of logic available unused. However, we will have to use

smaller DLUTs to stay within the BRAM memory limit. Virtex 5 FPGA has 148 BRAMs of 36Kb each. They can be used as one 36KB block or two 18 KB blocks. The maximum limit is reached for BRAM utilization as the data look up table (DLUT) size increases. The BRAM can hold a total of 5,328Kb of data. Thus, other FPGAs like Virtex6 can be used if more BRAM is needed. The BRAMs needed for a DLUT size of 2^{16} for quadratic and cubic is mapped to be 192 and 242 respectively, which exceeds the available BRAM size of 148. This results in an "error" message as shown in the last row of Table 4.1. Thus another FPGA with higher BRAM memory can be used for these implementations.

From the data obtained in Table 4.1 by running the exponential distribution VHDL code for different interval sizes and different polynomial fits, the BRAM utilization for higher order polynomials for different interval lengths was predicted. These values are shown in Table 4.2.

Mean Square Error (MSE) was calculated in Matlab for different DLUT sizes and for various polynomial fits from linear to 8^{th} order polynomial fit. This is shown in Table 4.3.

Poly Order ---------- **Log** **(LUT size)**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	Block RAMs used							
10	**2**	**3**	**4**	*5*	*6*	*7*	*8*	*9*
12	**8**	**11**	**15**	*19*	*22*	*26*	*30*	*33*
14	**32**	**48**	**64**	*81*	*96*	*110*	*126*	*141*
16	**122**	**192 (error)**	**242 (error)**	*300*				

Table 4.2 . Predicted BRAM usage for different LUT sizes for different polynomials
BRAMS usage shown bold was obtained by implementing the design on Virtex 5 FPGA.
BRAMS usage shown in italics was predicted based on trends.

Log (LUT size)	1	2	3	4	5	6	7	8
	Mean square Error	Mean square Error	Mean square Error	Mean square Error	Mean square Error	Mean square Error	Mean square Error	Mean square Error
10	2.4447 e-004	1.0849 e-004	6.1054 e-005	6.1010 e-005	3.9081 e-005	3.9049 e-005	3.9046 e-005	3.9030 e-005
12	6.1103 e-005	2.7110 e-005	2.0215 e-005	1.5243 e-005	1.5240 e-005	1.5164 e-005	1.4669 e-005	1.4133 e-005
14	1.5265 e-005	6.7678 e-006	6.7542 e-006	3.8173 e-006	3.8027 e-006	3.8025 e-006	3.8020 e-006	3.8030 e-006
16	3.8075 e-006	1.6847 e-006	1.6826 e-006					

Table 4.3 . Mean Square Error for different LUT sizes for different polynomials.

The results obtained in the design report of FPGA implementation agree with the Matlab simulations results described in the implementation section.

From Table 4.3 we can see that the MSE decreases as we move to higher order polynomials. But after a certain point, there is no significant decrease in error. For example, for a LUT size of 2^{12}, there is a large decrease in MSE as we move from linear fit to quadratic and then cubic. But the MSE remains almost the same after quartic (4^{th} order) polynomial fit. Instead of using a higher order polynomial fit like 7^{th} or 8^{th} order fit for a LUT size of 2^{12},we can use 4^{th} order polynomial fit which gives almost the same MSE.

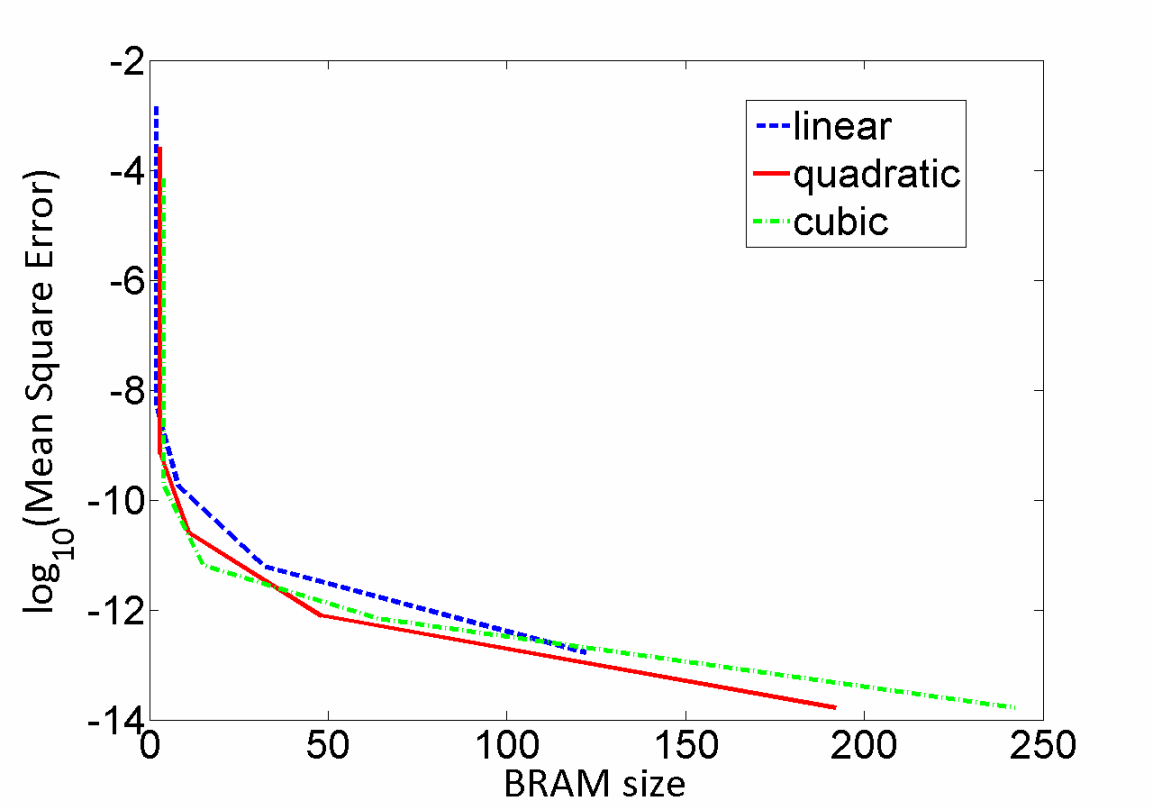

Figure 4.2 . Variation of Mean square error for different BRAM sizes

Figure 4.2 shows the variation of Mean Square Error(MSE) for different BRAM sizes for linear, quadratic and cubic polynomials. We can see that as the BRAM size increases, the MSE decreases drastically.

Figure 4.3 shows the variation of mean square error for different interval lengths from 2^0 to 2^{16} and for polynomial fits from 1^{st} order to 12^{th} order.

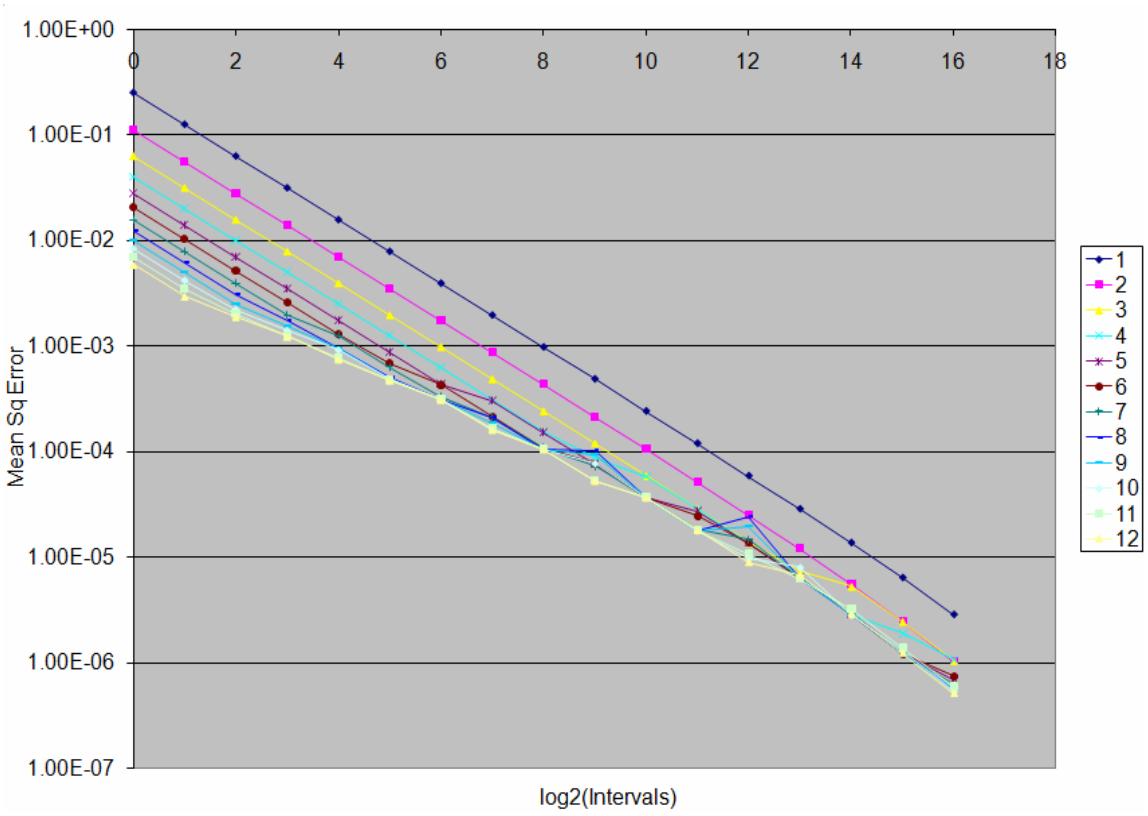

Figure 4.3. Variation of Mean square error for different interval lengths and different polynomial fits

From Figure 4.3 it can be seen that as the interval length increases, many higher order polynomial fits after cubic have almost similar MSEs. The MSEs after 10^{th} order are almost the same. Thus as the order of the polynomial fit is increased there is no significant improvement in MSE and it is thus better to use a lower order polynomial fit with a similar MSE. This way the number of logical blocks utilized will be smaller making the implementation more efficient. The MSE decreases drastically as we use bigger DLUTs, but the BRAMs size increases. From Figure 4.3 we can see that even with small DLUTs we can get better accuracy by using higher order polynomials like 3^{rd} or 4^{th} order for curve fitting. Thus depending on the availability of BRAMs and logic and also depending on the MSE requirement, an appropriate DLUT size and polynomial fit has to be chosen.

	Linear				Quadratic				Cubic			
Log	Block	Slice	Slice	MSE	Block	Slice	Slice	MSE	Block	Slice	Slice	MSE
(LUT size)	RAMs	Reg	LUTs		RAMs	Reg	LUTs		RAMs	Reg	LUTs	
2	2	263	4604	6.26E-02	3	392	8090	2.78E-02	4	515	11153	1.56E-02
4	2	266	4605	1.56E-02	3	394	8091	6.94E-03	4	517	11155	3.90E-03
6	2	265	4606	3.91E-03	3	393	8097	1.73E-03	4	517	11158	9.72E-04
8	2	261	4708	9.73E-04	3	394	7964	4.30E-04	4	519	11158	2.41E-04
10	2	267	4809	2.44E-04	3	395	8563	1.08E-04	4	517	12614	6.11E-05
12	8	270	4820	6.11E-05	11	398	8076	2.71E-05	15	519	12076	2.02E-05
14	32	273	4607	1.53E-05	48	400	7346	6.77E-06	64	525	11158	6.75E-06
16	122	271	4811	3.81E-06	192	403	8192	1.68E-06	242	528	12332	1.68E-06
	10%				18%				26%			

Table 4.4. Device utilization for different DLUT sizes and polynomials.

In Table 4.4 it can be seen that the device utilization is similar for all DLUT sizes less than 2^{10}. Thus DLUT sizes less than 2^{10} is eliminated, as better accuracy can be achieved with 2^{10} DLUT than with smaller DLUTs, for the same device utilization. For a DLUT size of 2^{14} the MSE of quadratic is almost similar to cubic. Thus it is better to use quadratic as has less device utilization. The quadratic and cubic polynomials for a DLUT size of 2^{16} cannot be used as the BRAM utilization for these designs exceed the available BRAM on the FPGA. Thus depending on the resources available and the MSE needed, the DLUT size and the polynomial fit can be chosen for the design from the available choices.

4.2.1Timing analysis

The exponential distribution was successfully implemented on the FPGA and the results were verified. The speed the design had to be calculated to know how many exponentially distributed numbers can be generated each second. A timer was initiated in Xilinx Platform Studio (XPS) to count the number clock pulses taken to get the output exponential number. The software part of the design was done in Xilinx SDK. A C code was written to start the timer when input is ready and stop the timer when the output is generated.

A FIFO was created using Xilinx coregen which holds input numbers. The exponent of the number that was input was calculated and displayed as output on the Hyper terminal. The clock pulse count was stored in timer register which was read and thus the timing was verified.

Number of clock pulses taken to find the exponent of numbers was monitored. The FIFO size was 2^{16}. Therefore, 2^{16} numbers were given as input to the exponential module. The timer register displayed a value of 65,722. A system clock of 100 MHz was used. The following calculations were performed to determine the number of exponential numbers that are generated every second.

$((100\text{MHz})^{-1} \times 65{,}722)^{-1} \times 2^{16} = 99{,}716{,}989$ numbers per second

The results shows that one exponential number is generated every clock pulse. This implies that almost 100 million exponential numbers can be generated per second.

The static timing report in the Xilinx ISE showed the maximum frequency of operation of 50-60 MHz for linear, quadratic and cubic polynomial fits. But the exponential module was run at a 100MHz clock successfully. The static timing report takes the longest path to calculate the maximum frequency of operation and this path may be unused in the design. Therefore, the exponential distribution generator was able to run at 100MHz. The timer register values also showed that 100 million random numbers were generated every second.

The floating point library that is used in the design is not pipelined. Using a more efficient pipelined floating point unit would increase the frequency in the timing report. The Xilinx Core generator's floating point adder and multiplier was used in the design instead of the floating point library. The floating point library was only used to convert the input uniform number to float before performing addition or multiplication operations. It was found that the frequency of operation in the Xilinx ISE timing report went up to 91M Hz for the linear fit. The logic utilization also increased for the pipelined

Coregen floating point unit. Thus by using a pipelined floating point adder and multiplier and an efficient conversion of the input uniform random number to float, the design can be run at higher frequencies. The design implemented in this thesis uses the floating point library for all its operations.

Figure 4.5 shows the resource utilization and the frequency of operation obtained in the timing report for linear and cubic curve fits. In the first case, pipelined coregen floating point adder and multiplier were used to perform floating point addition and multiplication and float library was used to convert the input uniform number to floating point number. The DLUTs (Data Look Up Tables) were stored in float format. Hence the conversion of the polynomial coefficients in DLUTs to floating point is not needed. The design is pipelined and hence both the linear and cubic get the same frequency. In the second case, floating point library was used to convert the input to floating point and also to do addition and multiplication operations. The DLUTs were stored in integer formats. Thus the conversions of coefficients in the DLUTs have to be converted before performing the interpolation.

Floating point unit	**Linear**			**Cubic**		
	SR	SLUT	Freq	SR	SLUT	Freq
Pipelined Coregen	2,000	2,066	91.397MHz	6,045	5,635	91.397MHz
Float library	238	3,262	63.11MHz	517	11,158	52.722MHz

Table 4.5. Resource utilization and frequency of operation of Pipelined Coregen floating point unit vs. floating point library

SR: Slice register

SLUT: Slice LUT

Freq: maximum Frequency of operation in Xilinx timing report

4.3 True random number interface

A true random number generator [14] was interfaced with the exponential module and the results were verified. 32 bit exponentially generated random numbers were generated by connecting the output the true random number generator to the input of the exponential module. The design was performed for linear, quadratic and cubic polynomial fits with a DLUT size of 2^{10}.

5. Conclusion

An efficient exponential distribution generator was implemented on hardware by dividing the inverse CDF into intervals and curve fitting each interval with linear, quadratic and cubic polynomials. This method of implementing exponential distribution on hardware can be extended to other probability distribution functions with a well defined inverse CDF function. The design has the following attributes:

Speed/Number of exponentially distributed numbers generated per second

The exponential module that is designed is time efficient as it produces one exponential number every clock cycle, i.e. 100 million numbers/second with a 100 MHz clock.

Efficiency

The design is also very efficient. In case of exponential distribution, the mean square error decreases as we fit higher order polynomials up to cubic polynomials into each interval, although this improvement may not be significant as me fit higher order polynomial than cubic.

Tradeoff between number of intervals and polynomial order: optimization

A thorough study of fitting different polynomials to the intervals and getting an optimum size data look up table was performed. Thus depending on the resources available and accuracy needed, the DLUT size and the polynomial fit is decided. The design was implemented on a Virtex 5 FPGA and the resources of the FPGA were well utilized.

5.1 Future work

The research performed in this thesis gives a very efficient exponential distribution generator. But there is still a lot of scope for improvement. More efficient methods to handle the data look up tables can be implemented. Instead of storing the DLUT values in the memory, different methods like changing the values of the data look up tables at run time can be investigated. This way, we can change the probability distribution we want to

generate at run time and thus get a universal probability distribution generator which is more general and widely applicable. For example certain data that behaves like an exponential distribution initially and normal distribution later can be handled by hardware using the above method of changing the DLUT values during runtime.

We can try to implement the design in other FPGAs like Virtex 6, which have more block RAM and can thus store big DLUTs. A better floating point unit that performs floating point addition and multiplication operations efficiently and with less resource utilization can be used instead of the floating point library used in this research.

6. References

1. Linnartz , Jean-Paul (2009). Telephony Traffic Models. Retrieved April,2010, from http://www.wirelesscommunication.nl/reference/chaptr04/erlang/erlang.htm
2. Adan , Ivo, & Resing, Jacques (February 14, 2001). Queueing Theory. Retrieved April 2010, from http://www.win.tue.nl/~iadan/queueing.pdf
3. Beasley, J. E. (22-Feb-2005). OR-Notes. Retrieved April,2010, from http://people.brunel.ac.uk/~mastjjb/jeb/or/queue.html
4. Qian, Hong (2009). Exponential Distribution and the Process of Radioactive Decay. Retrieved April,2010, from http://www.amath.washington.edu/courses /423-winter-2010/note2.pdf
5. How Alcohol Is Metabolized in the Human Body. (2007). Retrieved April,2010, from http://www.angelfire.com/planet/hamshrn/metabolism/
6. J. M. McCollum, J. M. Lancaster, D. W. Bouldin, and G. D. Peterson. Hardware acceleration of pseudo-random number generation for simulation applications. In IEEE Southeastern Symposium on System Theory, pages 299–303, 2003.
7. R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design Measurement, Simulation, and Modeling, John Wiley & Sons, Inc., NY: 1991, pp. 439-444, 488-489
8. Malik, Ali (August 2005). Design Tradeoff Analysis of Floating point adders in FPGA. Master's thesis, University of Saskatchewan
9. Evans, Merran, Hastings, Nicholas, & Peacock, Brian (3 edition, June 15, 2000). Statistical Distributions. NY: Wiley, John & Sons, Incorporated.
10. Bennet, Thomas W Decimal to Floating-Point Conversions. Retrieved April, 2010, from http://sandbox.mc.edu/~bennet/cs110/flt/dtof.html
11. Roth, Charles (1998). Digital Systems Design using VHDL. Belmont, CA, USA : Wadsworth Publ. Co.
12. Xilinx, FPGA vs. ASIC. Retrieved April, 2010, from http://www.xilinx.com/company/gettingstarted/fpgavsasic.htm
13. Xilinx, Virtex 5 family overview. http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf
14. Mitchum, S.T.; Klenke, R.H.," Divergent Path Random Number Generators", proceedings of: Southeastcon, 2009. SOUTHEASTCON '09. IEEE
15. C. Wallace, "Fast pseudorandom generators for normal and exponential variates", ACM Transactions on Mathematical Software, vol. 22, no. 1, pp. 119.127, 1996.

16. G. L. Zhang, P. H. Leong, D.-U. Lee, J. D. Villasenor, R. C. Cheung, and W. Luk, "Ziggurat-based hardware Gaussian random number generator," in International Conference on Field Programmable Logic and Applications. IEEE Computer Society Press, 2005, pp. 275–280
17. D. B. Thomas and W. Luk, "Efficient hardware generation of random variates with arbitrary distributions," in IEEE Symposium on FPGAs for Custom Computing Machines, 2006
18. Thomas, D.B.; Luk, W., Non-Uniform Random Number Generation Through Piecewise Linear Approximations International Conference on Field Programmable Logic and Applications, 2006
19. Bishop, David (2005). Fixed- and floating-point packages for VHDL 2005. DVCON, 2005
20. Bishop, David (2005). Floating point package user's guide. Retrieved April,2010, from http://www.eda-stds.org/vhdl-200x/vhdl-200x-ft/packages old/ Float_ug.pdf
21. Wen, Quanfei (February 1998). IEEE-754 Floating-Point Conversion. Retrieved April,2010, from http://babbage.cs.qc.edu/IEEE-754/Decimal.html
22. Xilinx, Virtex-5 FPGA User Guide. http://www.xilinx.com/support/documentation/user_guides/ug190.pdf

Appendix

Linear

```
----------------------------------------------------------------------------------
--Function: Calculates exponential of a number by curve fitting using linear polynomial
-- Input: clk - std_logic,urn -32 bit std_logic_vector
-- Output: exprand -32 bit std_logic_vector
----------------------------------------------------------------------------------

 LIBRARY IEEE;
 library ieee_proposed;
 use ieee_proposed.float_pkg.all;
 use IEEE.Std_Logic_1164.all;
 use IEEE.STD_LOGIC_ARITH.ALL;
 use IEEE.STD_LOGIC_UNSIGNED.ALL;
 use ieee_proposed.fixed_pkg.all;

 entity linear is
  port(
    clk :in std_logic;
    urn : in std_logic_vector(31 downto 0);
    exprand : out std_logic_vector (31 downto 0)
      );
  end linear;


 architecture design of linear is
  signal ind:std_logic_vector(5 downto 0):="000000";
  signal index,Xi:integer:=0;
  signal exp : float32;
  signal Alin,Blin,X,Xd:float32;
  signal expl:float32;
  constant f:float32:="00110000000000000000000101010001";
  signal stda_linear,stdb_linear:std_logic_vector(31 downto 0);
  signal Al:float32;

type LUT is array (0 to 63) of integer;

 constant LINEARa1 :LUT := (1051438193, 1051572411, 1051706629,1051844202,
          1051981775, 1052126059,1052270343,1052414627,1051438193,1051572411,
          1051706629, 1051844202,1051981775,1052126059,1052270343,1052414627,
          1051438193,1051572411,1051706629, 1051844202,1051981775,1052126059,
          1052270343,1052414627,1051438193, 1051572411,1051706629,1051844202,
          1051981775,1052126059,1052270343,1052414627,1051438193, 1051572411,
          1051706629, 1051844202,1051981775,1052126059,1052270343,1052414627,
          1051438193,1051572411, 1051706629, 1051844202,1051981775,1052126059,
          1052270343,1052414627,1051438193, 1051572411, 1051706629,1051844202,
          1051981775,1052126059, 1052270343,1052414627,1051438193,1051572411,
          1051706629, 1051844202,1051981775,1052126059,1052270343,1052414627);


constant LINEARb1 :LUT :=(1051438193, 1051572411, 1051706629,1051844202,
        1051981775,1052126059, 1052270343,1052414627,1051438193,1051572411,
        1051706629, 1051844202,1051981775,1052126059,1052270343,1052414627,
        1051438193,1051572411, 1051706629,1051844202,1051981775,1052126059,
        1052270343,1052414627,1051438193,1051572411,1051706629, 1051844202,
        1051981775,1052126059,1052270343,1052414627,1051438193, 1051572411,
        1051706629,1051844202,1051981775,1052126059, 1052270343,1052414627,
        1051438193, 1051572411, 1051706629,1051844202,1051981775,1052126059,
        1052270343,1052414627,1051438193, 1051572411, 1051706629,1051844202,
        1051981775,1052126059, 1052270343,1052414627,1051438193, 1051572411,
        1051706629, 1051844202,1051981775,1052126059, 1052270343,1052414627);

    begin
```

```
      process(clk)
      begin
        if clk='1' and clk' event then
                  ind<=urn(31 downto 26);
                  index<=conv_integer(ind);
             Xi<=conv_integer(urn(31 downto 0));
             Xd<=to_float(Xi);
             X<= Xd *f;
             stda_linear<=conv_std_logic_vector(LINEARa1(index),32);
             stdb_linear<=conv_std_logic_vector(LINEARb1(index),32);
             exp<=expl;
             exprand<=to_slv(exp);
         end if;
    end process;

--Converts a and b to floating point
   toflinear:process(clk)
   begin
       if clk='1' and clk' event then
           Alin<=To_float(stda_linear);
           Blin<=To_float(stdb_linear);
       end if;
  end process toflinear;

--Linear polynomial fit ax+b
   Lin: process(clk)
   begin
        if clk='1' and clk' event then
       Al<= Alin*X;
       end if;
   end process Lin ;

  explin:process(clk)
   begin
       if clk='1' and clk' event then
           expl<=Al+Blin;
      end if;
 end process explin;

 end design;
```

Quadratic

```
--------------------------------------------------------------------------------
--Function: Calculates exponential of a number by curve fitting using quadratic
--                  polynomial
-- Input: clk - std_logic,urn -32 bit std_logic_vector
-- Output: exprand -32 bit std_logic_vector
--------------------------------------------------------------------------------

  LIBRARY IEEE;
  library ieee_proposed;
  use ieee_proposed.float_pkg.all;
  USE IEEE.Std_Logic_1164.all;
  use IEEE.STD_LOGIC_ARITH.ALL;
  use IEEE.STD_LOGIC_UNSIGNED.ALL;
  use ieee_proposed.fixed_pkg.all;

  entity quad is
    port(
         clk :in std_logic;
         urn : in std_logic_vector(31 downto 0);
         exprand : out std_logic_vector (31 downto 0)
         );
  end quad;


architecture design of quad is

signal ind:std_logic_vector(5 downto 0):="000000";
signal index,Xi:integer:=0;
signal exp : float32;
signal Aqua,Bqua,Cqua,X,Xd,Xsq,ABq: float32;
signal expq:float32;
constant f:float32:="00110000000000000000000101010001";
signal stda_quad :std_logic_vector(31downto 0);
signal stdb_quad :std_logic_vector(31downto 0);
signal stdc_quad :std_logic_vector(31downto 0);
signal Aq,Bq :float32;
signal stdexp:std_logic_vector(31 downto 0);

type LUT is array (0 to 63) of integer;

constant quada2 :LUT :=(379663292, 217424109, 557450834, 984198893, 870656109,
 521483550, 517574259, 941418453, 706227913, 319552454, 412345498, 986391343,
334652128,  833511935, 435739226, 962474219, 813280190, 696487943, 464112399, 306213272,
719818415, 239194206, 346640482, 71970826, 608006172, 103701628,
771529494, 799659653, 984077195, 920412516, 268481594, 359803676, 771747174,
263179087, 825648194, 55913812, 619520027, 830043195, 208033776, 845628895,
671874810, 603660393, 710539471, 314441421, 475378998, 80127875, 133820893,
120324693, 776010575, 478285937,       278050042, 834087114, 617109346,728459085,
873597457, 389275420, 587303300,      594957368, 769990562, 85133768, 339352445,
412813635, 479459311, 860062059);

constant quadb2 :LUT := (121415088    , 876323916, 937253381, 787972678, 491496488,
491548451, 211578144, 549758566, 235474273, 893442683, 396331566,   351517566, 758149530,
378562418, 715679974,  732341334, 479861249, 855946707,      218110383, 472371576,
661612092, 834545553,  687540855, 335799729 , 928947794, 255354729, 983216576, 855787557,
607557367,     305535176, 887455705, 306758910,603568373, 6654410, 813297585,
       220110657, 968826781, 816286366,      147223167, 645666974,
818602366, 397014267, 204749175, 304511201, 777464834, 487484221,45571940, 405885702,
781029560, 647883417,  752281163, 745985463, 479095154,      290316292, 839241273,
828608954, 631612334,  486439999, 506442288, 402922110,      323987124, 957742691,
54475232, 929801280 ) ;

constant quadc2 :LUT := (116533356, 685389924, 588238150, 917946112 ,802764422, 486309328,
171037299, 109506285, 298733345, 134091071,535036947,223590236, 336624974, 274192888,
557952511,     601695545, 313887946, 944638288,      186290869, 920805636, 48972236,
796665943, 250775239, 96080636, 63773320, 264533359, 782670867, 331673103, 712652608,
       453599047, 463071040, 271878626,393015413, 790971877, 772021279,  905223561
       , 368027015, 304088417, 759467617,114782895,
738470734, 921290860, 448683070,      347114952, 673705094, 949198404,      949284024,
```

```
895386837, 305771981, 271051071,      620423888, 961468376, 161029073,      462841332,
340057507, 863919883, 295979126,      259954604, 884450728, 602332351,      49687773,
121532074, 281923731, 857720864) ;

 begin

         process(clk)
         begin
            if clk='1' and clk' event then
               ind<=urn(31 downto 26);
               index<=conv_integer(ind);
               Xi<=conv_integer(urn(31 downto 1));
                           Xd<=to_float(Xi);
               X<= Xd *f;
                              stda_quad<=conv_std_logic_vector(quada2(index),32);
               stdb_quad<=conv_std_logic_vector(quadb2(index),32);
                           stdc_quad<=conv_std_logic_vector(quadc2(index),32);
                               exp<=expq;
                       end if;
               exprand<=to_slv(exp);


         end process;

        ----Converts a,b and c to floating point
         tofquad:process(clk)
         begin
            if clk='1' and clk' event then
               Aqua<=To_float(stda_quad);
               Bqua<=To_float(stdb_quad);
                           Cqua<=To_float(stdc_quad);
            end if;
         end process tofquad;

        -- A*x*x
        Quad1: process(clk)
        begin
            if clk='1' and clk' event then
                     Aq<= Aqua*Xsq;
            end if;
                 end process Quad1 ;

        --B*x
        Quad2: process(clk)
        begin
           if clk='1' and clk' event then
                        Bq<=Bqua*X;
                end if;
                end process Quad2 ;

        --A*x*x+B*x
        Aqadd: process(clk)
        begin
      if clk='1' and clk' event then
                   ABq<=Aq+Bq;
           end if;
        end process Aqadd;

                --Ax*x+B*x+C
        expquad:process(clk)
        begin
           if clk='1' and clk' event then
              expq<= ABq+Cqua;
          end if;
        end process expquad;

        --x*x
        Xsquare:       process(clk)
        begin
                 if clk='1' and clk' event then
                    Xsq<=X*X;
```

```
                end if;
        end process Xsquare;

end design;
```

Cubic

```
LIBRARY IEEE;
library ieee_proposed;
use ieee_proposed.float_pkg.all;
USE IEEE.Std_Logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee_proposed.fixed_pkg.all;

entity cubic is
port(
         clk :in std_logic;
         urn : in std_logic_vector(31 downto 0);
         exprand : out std_logic_vector (31 downto 0)
       );
end cubic;


architecture design of cubic is

 signal ind:std_logic_vector(5 downto 0):="000000";
 signal index,Xi:integer:=0;
 signal exp : float32;
 signal Acu,Bcu,Ccu,Dcu: float32;
 signal  X,Xd,Xsq,Xcube,ABq,ABc,ABCc: float32;
 signal expl,expq,expc:float32;
 constant f:float32:="00110000000000000000000101010001";
 signal stda_cubic,stdb_cubic,stdc_cubic,stdd_cubic:std_logic_vector(31 downto 0);
 signal Al,Aq,Bq,Ac,Bc,Cc :float32;
 signal stdexp:std_logic_vector(31 downto 0);

type LUT is array (0 to 63) of integer;

constant cubica3 :LUT := (379663292, 217424109, 557450834, 984198893, 870656109,
521483550, 517574259, 941418453, 706227913, 319552454, 412345498,    986391343,
334652128, 833511935, 435739226,      962474219, 813280190, 696487943,      464112399,
306213272, 719818415, 239194206,      346640482, 71970826, 608006172,103701628,
771529494, 799659653, 984077195,      920412516, 268481594, 359803676, 771747174,
263179087, 825648194, 55913812, 619520027, 830043195, 208033776, 845628895,
 671874810, 603660393, 710539471      , 314441421, 475378998, 80127875,     133820893,
120324693, 776010575, 478285937,      278050042, 834087114 , 617109346      ,728459085,
873597457, 389275420, 587303300,      594957368, 769990562, 85133768, 339352445,
412813635, 479459311, 860062059);

constant cubicb3 :LUT := (121415088,876323916,       937253381, 787972678, 491496488,
491548451, 211578144, 549758566, 235474273, 893442683, 396331566, 351517566,
758149530, 378562418, 715679974, 732341334, 479861249, 855946707, 218110383,
 47237157, 661612092, 834545553, 687540855, 335799729, 928947794, 255354729,
983216576, 855787557, 607557367,      305535176, 887455705, 306758910, 603568373, 6654410,
813297585,     220110657, 968826781, 816286366,      147223167,645666974,
818602366, 397014267, 204749175, 304511201, 777464834, 487484221,   45571940,
405885702, 781029560, 647883417,      752281163, 745985463, 479095154,      290316292,
839241273, 828608954, 631612334,      486439999, 506442288, 402922110,      323987124,
957742691, 54475232, 929801280);

constant cubicc3 :LUT := (116533356, 685389924, 588238150, 917946112,802764422,
486309328, 171037299, 109506285, 298733345, 134091071, 535036947,   223590236,
336624974, 274192888, 557952511,      601695545, 313887946, 944638288,      186290869,
920805636, 48972236, 796665943, 250775239, 96080636, 63773320,264533359, 782670867,
331673103, 712652608,  453599047, 463071040, 271878626, 393015413, 790971877, 772021279,
905223561, 368027015, 304088417, 759467617,  114782895,
738470734, 921290860, 448683070,      347114952, 673705094, 949198404,      949284024,
895386837, 305771981, 271051071,      620423888, 961468376, 161029073,      462841332,
340057507, 863919883, 295979126,      259954604, 884450728, 602332351,      49687773,
121532074, 281923731, 857720864       );

constant cubicd3 :LUT := (130710016, 993703648, 621858027, 225919470, 832077741,
186164517, 525982250, 770532095, 939741118, 132138646, 315853674,   923454197,
```

```
167011105, 567929095, 986745034,      940106894, 11956416, 97614603, 36312462, 66631738,
892626284, 480767608, 246844394, 580470295, 285074586, 375070160,
671252573, 637588574, 769963702,      691609959, 172331480, 11025438, 174979182,
283864137, 395094907, 598297564,      490808539, 369907567, 807638979,      939738014,
252858533, 325845605, 279103687,      788820452, 809478616, 71936329, 35278644,
912086769, 777284329, 598828301,      779668006, 353686607, 484865129,      222577064,
24709292, 506605854, 364860657, 232837914, 32846030, 363947809, 69853403,
547689061, 46961589 ,857957253);

   begin

  process(clk)
  begin
     if clk='1' and clk' event then
        ind<=urn(31 downto 26);
        index<=conv_integer(ind);
        Xi<=conv_integer(urn(31 downto 1));
        Xd<=to_float(Xi);
        X<= Xd *f;
        stda_cubic<=conv_std_logic_vector(cubica3(index),32);
        stdb_cubic<=conv_std_logic_vector(cubicb3(index),32);
        stdc_cubic<=conv_std_logic_vector(cubicc3(index),32);
        stdd_cubic<=conv_std_logic_vector(cubicd3(index),32);
        exp<=expc;
    end if;
     exprand<=to_slv(exp);
 end process;

--a,b,c and d to floating point
 tofcubic:process(clk)
 begin
    if clk='1' and clk' event then
       Acu<=To_float(stda_cubic);
       Bcu<=To_float(stdb_cubic);
       Ccu<=To_float(stdc_cubic);
       Dcu<=To_float(stdd_cubic);
    end if;
 end process tofcubic;

--a*x*x*x
 cubic1: process(clk)
 begin
    if clk='1' and clk' event then
       Ac<= Acu*Xcube;
    end if;
 end process cubic1 ;

--b*x*x
 cubic2: process(clk)
 begin
    if clk='1' and clk' event then
       Bc<=Bcu*Xsq;
    end if;
 end process cubic2 ;

--c*x
 cubic3: process(clk)
 begin
    if clk='1' and clk' event then
       Cc<=Ccu*X;
    end if;
 end process cubic3 ;

-- a*x^3 + b*x^2
 Acadd: process(clk)
 begin
    if clk='1' and clk' event then
       ABc<=Ac+Bc;
    end if;
 end process Acadd;
```

```
-- a*x^3 +b*x^2 + c*x
 ABcadd: process(clk)
 begin
    if clk='1' and clk' event then
       ABCc<=ABc+Cc;
    end if;
 end process ABcadd;

-- a*x^3 +b*x^2 + c*x +d
 expcubic:process(clk)
 begin
    if clk='1' and clk' event then
       expc<= ABCc+Dcu;
    end if;
 end process expcubic;

-- x*x
 Xsquare:      process(clk)
 begin
    if clk='1' and clk' event then
       Xsq<=X*X;
    end if;
 end process Xsquare;

 --x*x*x
  Xcu:  process(clk)
  begin
     if clk='1' and clk' event then
        Xcube<= X*Xsq;
     end if;
  end process Xcu;

end design;
```

Linear interpolation using Coregen Floating point adder and multiplier

(uses float library to convert input to floating point)

```
LIBRARY IEEE;
library ieee_proposed;
use ieee_proposed.float_pkg.all;
USE IEEE.Std_Logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee_proposed.fixed_pkg.all;
Library XilinxCoreLib;

entity linear is
       port(clk :in std_logic;
             urn : in std_logic_vector(31 downto 0);
             exprand : out std_logic_vector (31 downto 0));
end linear;


   architecture design of linear is

  signal ind:std_logic_vector(7 downto 0):="00000000";
  signal index,Xi:integer:=0;
  signal exp : float32;
  signal Alin,Blin,X,Xd:float32:="00000000000000000000000000000000" ;
  constant f1:std_logic_vector:="00110000000000000000000101010001";
  signal stda_linear,stdb_linear:std_logic_vector(31 downto 0);
  signal Al:float32;
  signal Als,Blins,Alins,Xs,Xds,fs:std_logic_vector(31 downto 0);

  type LUT is array (0 to 255) of std_logic_vector(31 downto 0);

  constant LINEARa1 :LUT := (x"3F804189",    x"3F80C155",   x"3F814467",   x"3F81C77A",
        x"3F824A8C",   x"3F82D0E5",   x"3F83573F",   x"3F83DD98",   x"3F8463F1",
x"3F84ED91",   x"3F857A78",   x"3F860419",   x"3F869100",   x"3F87212D",   x"3F87AE14",
        x"3F883E42",   x"3F88D1B7",   x"3F89652C",
x"3F89F8A1",   x"3F8A8F5C",   x"3F8B22D1",   x"3F8BBCD3",   x"3F8C56D6",   x"3F8CF0D8",
        x"3F8D8ADB",   x"3F8E2824",   x"3F8EC8B4",
x"3F8F6944",   x"3F9009D5",   x"3F90AA65",   x"3F915183",   x"3F91F55A",   x"3F929C78",
        x"3F9346DC",   x"3F93F141",   x"3F949BA6",
x"3F954952",   x"3F95F6FD",   x"3F96A7F0",   x"3F9758E2",   x"3F980D1B",   x"3F98C49C",
        x"3F997C1C",   x"3F9A339C",   x"3F9AEE63",
x"3F9BAC71",   x"3F9C6A7F",   x"3F9D288D",   x"3F9DE9E2",   x"3F9EAE7D",   x"3F9F7319",
        x"3FA03AFB",   x"3FA10625",   x"3FA1D14E",
x"3FA29FBE",   x"3FA36E2F",   x"3FA43FE6",   x"3FA514E4",   x"3FA5E9E2",   x"3FA6C227",
        x"3FA79DB2",   x"3FA8793E",   x"3FA95810",
x"3FAA3A2A",   x"3FAB1C43",   x"3FAC01A3",   x"3FACEA4B",   x"3FADD639",   x"3FAEC227",
        x"3FAFB4A2",   x"3FB0A71E",   x"3FB1999A",
x"3FB292A3",   x"3FB38BAC",   x"3FB48B44",   x"3FB58ADB",   x"3FB68DB9",   x"3FB793DE",
        x"3FB89D49",   x"3FB9A6B5",   x"3FBAB6AE",
x"3FBBC9EF",   x"3FBCDD2F",   x"3FBDF6FD",   x"3FBF10CB",   x"3FC03127",   x"3FC15183",
        x"3FC2786C",   x"3FC3A29C",   x"3FC4CCCD",
x"3FC5FD8B",   x"3FC73190",   x"3FC86C22",   x"3FC9A6B5",   x"3FCAE48F",   x"3FCC28F6",
        x"3FCD70A4",   x"3FCEBB99",   x"3FD00D1B",
x"3FD161E5",   x"3FD2B9F5",   x"3FD41893",   x"3FD57A78",   x"3FD6DFA4",   x"3FD84B5E",
        x"3FD9BA5E",   x"3FDB2FEC",   x"3FDCA8C1",
x"3FDE2824",   x"3FDFAACE",   x"3FE1374C",   x"3FE2C3CA",   x"3FE45A1D",   x"3FE5F3B6",
        x"3FE793DE",   x"3FE93A93",   x"3FEAE48F",
x"3FEC985F",   x"3FEE4F76",   x"3FF01062",   x"3FF1D495",   x"3FF3A29C",   x"3FF573EB",
        x"3FF74F0E",   x"3FF930BE",   x"3FFB18FC",
x"3FFD07C8",   x"3FFF0069",   x"40007FCC",   x"4001844D",   x"40028C15",   x"400398C8",
        x"4004AA65",   x"4005BF48",   x"4006D917",
x"4007F7CF",   x"40091B71",   x"400A425B",   x"400B6FD2",   x"400CA234",   x"400DDB23",
        x"400F1759",   x"40105A1D",   x"4011A36E",
x"4012F1AA",   x"40144674",   x"4015A027",   x"4017020C",   x"401868DC",   x"4019D7DC",
        x"401B4D6A",   x"401CC986",   x"401E4C30",
x"401FD8AE",   x"40216BBA",   x"402306F7",   x"4024AA65",   x"40265604",   x"40280B78",
        x"4029C91D",   x"402B8EF3",   x"402D6042",
```

```
x"402F3B64",   x"4031205C",   x"40330F28",   x"4035096C",   x"403710CB",   x"403921FF",
       x"403B3EAB",   x"403D6A16",   x"403FA0F9",
x"4041E4F7",   x"404437B5",   x"40469931",   x"404907C8",   x"404B86C2",   x"404E17C2",
       x"4050B780",   x"40536944",   x"40562B6B",
x"405902DE",   x"405BEC57",   x"405EE979",   x"4061FD8B",   x"406526E9",   x"40686595",
       x"406BBE77",   x"406F2FEC",   x"4072B9F5",
x"4076617C",   x"407A233A",   x"407E0419",   x"4081020C",   x"40831340",   x"408534D7",
       x"40876873",   x"4089AEE6",   x"408C0903",
x"408E786C",   x"4090FDF4",   x"40939B3D",   x"40965048",   x"40991F8A",   x"409C0AA6",
       x"409F126F",   x"40A23886",   x"40A57F63",
x"40A8E979",   x"40AC779A",   x"40B02D0E",   x"40B40C4A",   x"40B81893",   x"40BC538F",
       x"40C0C227",   x"40C566CF",   x"40CA4745",
x"40CF65FE",   x"40D4C986",   x"40DA75F7",   x"40E072B0",   x"40E6C49C",   x"40ED758E",
       x"40F48C15",   x"40FC12D7",   x"410209D5",
x"41064D6A",   x"410ADB23",   x"410FBA5E",   x"4114F488",   x"411A93DE",   x"4120A3D7",
       x"41273261",   x"412E5048",   x"41360FF9",
x"413E87FD",   x"4147D42C",   x"41521412",   x"415D6FD2",   x"416A182B",   x"417849BA",
       x"418427F0",   x"418D463F",   x"4197BEAB",
x"41A3E45A",   x"41B22752",   x"41C3229C",   x"41D7B296",   x"41F11B71",   x"4208A787",
       x"421DB98C",   x"423A7D56",   x"42641E6A",
x"4292E305",   x"42CE76E3",   x"432EA5C9",   x"443FCED7" );


        constant LINEARb1 :LUT :=( x"00000000",    x"00000000",   x"00000000",
       x"B8D1B717",   x"B951B717",   x"B951B717",   x"B99D4952",   x"B9D1B717",
       x"BA1D4952",
x"BA378034",   x"BA6BEDFA",   x"BA902DE0",   x"BAAA64C3",   x"BAC49BA6",   x"BADED289",
       x"BB03126F",   x"BB16BB99",   x"BB2A64C3",
x"BB3E0DED",   x"BB51B717",   x"BB6BEDFA",   x"BB83126F",   x"BB902DE0",   x"BB9D4952",
       x"BBAA64C3",   x"BBBAC711",   x"BBCB295F",
x"BBDB8BAC",   x"BBEBEDFA",   x"BBFF9724",   x"BC09A027",   x"BC1374BC",   x"BC1D4952",
       x"BC28C155",   x"BC343958",   x"BC3FB15B",
x"BC4CCCCD",   x"BC5844D0",   x"BC656042",   x"BC741F21",   x"BC809D49",   x"BC87FCB9",
       x"BC8F5C29",   x"BC978D50",   x"BC9EECC0",
x"BCA7EF9E",   x"BCB020C5",   x"BCB923A3",   x"BCC22681",   x"BCCB295F",   x"BCD4FDF4",
       x"BCDED289",   x"BCE978D5",   x"BCF41F21",
x"BCFEC56D",   x"BD04B5DD",   x"BD0A71DE",   x"BD1096BC",   x"BD16BB99",   x"BD1CE076",
       x"BD230553",   x"BD29930C",   x"BD3089A0",
x"BD378034",   x"BD3E76C9",   x"BD45D639",   x"BD4D35A8",   x"BD54FDF4",   x"BD5CC63F",
       x"BD64F766",   x"BD6D288D",   x"BD75C28F",
x"BD7E5C92",   x"BD83AFB8",   x"BD883127",   x"BD8CE704",   x"BD91D14E",   x"BD96BB99",
       x"BD9BA5E3",   x"BDA0F909",   x"BDA64C30",
x"BDAB9F56",   x"BDB126E9",   x"BDB6E2EB",   x"BDBC9EED",   x"BDC28F5C",   x"BDC8B439",
       x"BDCED917",   x"BDD566CF",   x"BDDBF488",
x"BDE28241",   x"BDE978D5",   x"BDF06F69",   x"BDF79A6B",   x"BDFEF9DB",   x"BE032CA5",
       x"BE0710CB",   x"BE0AF4F1",   x"BE0EF34D",
x"BE132618",   x"BE1758E2",   x"BE1BA5E3",   x"BE200D1B",   x"BE248E8A",   x"BE292A30",
       x"BE2DE00D",   x"BE32B021",   x"BE379A6B",
x"BE3C9EED",   x"BE41D7DC",   x"BE4710CB",   x"BE4C7E28",   x"BE5205BC",   x"BE57C1BE",
       x"BE5D7DBF",   x"BE636E2F",   x"BE6978D5",
x"BE6FB7E9",   x"BE761134",   x"BE7C84B6",   x"BE819653",   x"BE84F766",   x"BE8872B0",
       x"BE8C0831",   x"BE8FAACE",   x"BE9367A1",
x"BE973190",   x"BE9B22D1",   x"BE9F212D",   x"BEA339C1",   x"BEA76C8B",   x"BEABB98C",
       x"BEB013A9",   x"BEB49518",   x"BEB930BE",
x"BEBDF3B6",   x"BEC2C3CA",   x"BEC7BB30",   x"BECCCCCD",   x"BED1F8A1",   x"BED74BC7",
       x"BEDCC63F",   x"BEE25AEE",   x"BEE816F0",
x"BEEDFA44",   x"BEF404EA",   x"BEFA29C7",   x"BF004189",   x"BF0381D8",   x"BF06D5D0",
       x"BF0A43FE",   x"BF0DC5D6",   x"BF115B57",
x"BF150B0F",   x"BF18DB8C",   x"BF1CB924",   x"BF20B780",   x"BF24D6A1",   x"BF29096C",
       x"BF2D5CFB",   x"BF31D14E",   x"BF365FD9",
x"BF3B15B5",   x"BF3FE5C9",   x"BF44DD2F",   x"BF49F55A",   x"BF4F34D7",   x"BF549BA6",
       x"BF5A29C7",   x"BF5FDF3B",   x"BF65C28F",
x"BF6BD3C3",   x"BF720C4A",   x"BF787FCC",   x"BF7F1AA0",   x"BF82F838",   x"BF867D56",
       x"BF8A1FF3",   x"BF8DDCC6",   x"BF91BA5E",
x"BF95B8BB",   x"BF99D7DC",   x"BF9E17C2",   x"BFA27EFA",   x"BFA70A3D",   x"BFABBCD3",
       x"BFB096BC",   x"BFB59E84",   x"BFBAD42C",
x"BFC037B5",   x"BFC5CC64",   x"BFCB9581",   x"BFD19653",   x"BFD7CED9",   x"BFDE45A2",
       x"BFE4FAAD",   x"BFEBF488",   x"BFF33333",
x"BFFAB9F5",   x"C0014952",   x"C0055E9E",   x"C0099E84",   x"C00E0DED",   x"C012AE7D",
       x"C01781D8",   x"C01C8E8A",   x"C021D495",
```

```
x"C02758E2",   x"C02D205C",   x"C0332E49",   x"C0398937",   x"C040346E",   x"C047367A",
       x"C04E978D",   x"C0565AEE",   x"C05E8A72",
x"C0672FEC",   x"C07051EC",   x"C079FBE7",   x"C0821CAC",   x"C0878BAC",   x"C08D5254",
       x"C0937803",   x"C09A068E",   x"C0A106F7",
x"C0A88588",   x"C0B0902E",   x"C0B93405",   x"C0C283E4",   x"C0CC92A3",   x"C0D77732",
       x"C0E34D6A",   x"C0F032CA",   x"C0FE4DD3",
x"C106E4F7",   x"C10F6DC6",   x"C118E0DF",   x"C123652C",   x"C12F29C7",   x"C13C69AD",
       x"C14B6FD2",   x"C15C9B3D",   x"C17066CF",
x"C183BB64",   x"C191535B",   x"C1A19412",   x"C1B556A1",   x"C1CDDB23",   x"C1ED0DB9",
       x"C20B05BC",   x"C2271D98",   x"C24FEFB8",
x"C288496C",   x"C2C32D43",   x"C3287724",   x"C43DCC15");



--Coregen multiply
begin
coremul2:ENTITY floating_point_v3_1(floating_point_v3_1_a)
           port map(
                    a => Xds,
                    b => f1,
                    clk => clk,
                    result => Xs
                   ) ;

coremul1:ENTITY floating_point_v3_1(floating_point_v3_1_a)
           port map(
                   a => Alins,
                   b => Xs,
                   clk => clk,
                   result => Als
                  ) ;


       --Coregen add

coreadd:ENTITY floating_point_v3_0(floating_point_v3_0_a)
         port map(
                  a=>Als,
                  b =>Blins ,
                  clk => clk,
                  result =>exprand
                 ) ;



         process(clk)
         begin
            if clk='1' and clk' event then
              ind<=urn(31 downto 24);
              index<=conv_integer(ind);
              Xi<=conv_integer(urn(31 downto 0));
              Xd<=to_float(Xi);
             Xds<=to_slv(Xd);
           end if;
          end process;

       --Converts a and b to floating point
       toflinear:process(clk)
       begin
         if clk='1' and clk' event then
            Alins<=LINEARa1(index);
            Blins<=LINEARb1(index);
          end if;
       end process toflinear;


       end design;
```

Modelsim code for converting real to std_logic;

(The code below converts one of the linear polynomial co-efficient from real to float integer format)

```
LIBRARY IEEE;
USE work.all;
library ieee_proposed;
use ieee_proposed.float_pkg.all;
USE IEEE.Std_Logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;
use ieee_proposed.math_utility_pkg.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;
USE ieee.MATH_REAL.ALL;


entity rand is
port(clk :in std_logic;
     urn : in std_logic_vector(31 downto 0); -- input 32 bit random number from LFSR
     --fit :in integer range 0 to 2; -- integer to indicate linear,quaddratic or cubic
intrapolation
     ans :out std_logic_vector(31 downto 0)); -- exponentially generated random number
end rand;


architecture design of rand is

signal ind:std_logic_vector(7 downto 0):="00000000";
signal index,Xi,i:integer:=0;
signal upper,A,B,C,D:real;
signal A1,B1,exp,X:float32;

type LUTf is array (0 to 255) of float32;
signal LINEARb1f :LUTf;

type LUTstd is array (0 to 255) of std_logic_vector(31 downto 0);
signal LINEARb1std :LUTstd;

type LUTint is array (0 to 255) of integer;
signal LINEARb1int :LUTint;

type LUTrevstd is array (0 to 255) of std_logic_vector(31 downto 0);
signal LINEARb1revstd :LUTrevstd;

type LUTrev is array (0 to 255) of float32;
signal LINEARb1rev :LUTrev;


type LUT is array (0 to 255) of real;
constant LINEARa1 :LUT := ( 1.0020, 1.0059, 1.0099, 1.0139, 1.0179, 1.0220, 1.0261,
1.0302, 1.0343,1.0385, 1.0428, 1.0470, 1.0513, 1.0557, 1.0600, 1.0644, 1.0689, 1.0734,
1.0779, 1.0825, 1.0870, 1.0917, 1.0964, 1.1011, 1.1058, 1.1106, 1.1155, 1.1204, 1.1253,
1.1302, 1.1353, 1.1403, 1.1454, 1.1506, 1.1558, 1.1610,1.1663, 1.1716, 1.1770, 1.1824,
1.1879, 1.1935, 1.1991, 1.2047, 1.2104,1.2162, 1.2220, 1.2278, 1.2337, 1.2397, 1.2457,
1.2518, 1.2580, 1.2642,1.2705, 1.2768, 1.2832, 1.2897, 1.2962, 1.3028, 1.3095, 1.3162,
1.3230,1.3299, 1.3368, 1.3438, 1.3509, 1.3581, 1.3653, 1.3727, 1.3801, 1.3875,1.3951,
1.4027, 1.4105, 1.4183, 1.4262, 1.4342, 1.4423, 1.4504, 1.4587,1.4671, 1.4755, 1.4841,
1.4927, 1.5015, 1.5103, 1.5193, 1.5284, 1.5375,1.5468, 1.5562, 1.5658, 1.5754, 1.5851,
1.5950, 1.6050, 1.6151, 1.6254,1.6358, 1.6463, 1.6570, 1.6678, 1.6787, 1.6898, 1.7010,
1.7124, 1.7239,1.7356, 1.7474, 1.7595, 1.7716, 1.7840, 1.7965, 1.8092, 1.8221, 1.8351,
1.8484, 1.8618, 1.8755, 1.8893, 1.9034, 1.9176, 1.9321, 1.9468, 1.9617,1.9768, 1.9922,
2.0078, 2.0237, 2.0398, 2.0562, 2.0729, 2.0898, 2.1070,2.1245, 2.1423, 2.1603, 2.1787,
2.1974, 2.2165, 2.2358, 2.2555, 2.2756,2.2960, 2.3168, 2.3379, 2.3595, 2.3814, 2.4038,
2.4266, 2.4498, 2.4734, 2.4976, 2.5222, 2.5473, 2.5729, 2.5990, 2.6257, 2.6529, 2.6806,
```

```
2.7090,2.7380, 2.7676, 2.7978, 2.8287, 2.8604, 2.8927, 2.9257, 2.9596, 2.9942,3.0296,
3.0659, 3.1031, 3.1411, 3.1801, 3.2202, 3.2612, 3.3033, 3.3464, 3.3908, 3.4363, 3.4830,
3.5311, 3.5805, 3.6312, 3.6835, 3.7373, 3.7926,3.8497, 3.9084, 3.9690, 4.0315, 4.0961,
4.1627, 4.2315, 4.3026, 4.3761,4.4522, 4.5310, 4.6127, 4.6973, 4.7851, 4.8763, 4.9710,
5.0694, 5.1718,5.2785, 5.3896, 5.5055, 5.6265, 5.7530, 5.8852, 6.0237, 6.1688,  6.3212,
6.4812, 6.6496, 6.8269, 7.0140, 7.2115, 7.4206, 7.6421, 7.8773, 8.1274,8.3939, 8.6785,
8.9830, 9.3097, 9.6611, 10.0400, 10.4498, 10.8946, 11.3789,11.9082, 12.4893, 13.1299,
13.8398, 14.6309, 15.5180, 16.5195, 17.6593, 18.9681,20.4865, 22.2692, 24.3919, 26.9622,
30.1384, 34.1636, 39.4312, 46.6224, 57.0297,73.4434, 103.2322, 174.6476, 767.2319);

constant LINEARb1 :LUT :=( -0.0000, -0.0000, -0.0000, -0.0001, -0.0002, -0.0002, -
0.0003, -0.0004, -0.0006, -0.0007, -0.0009, -0.0011, -0.0013, -0.0015, -0.0017, -0.0020,
-0.0023, -0.0026,-0.0029, -0.0032, -0.0036, -0.0040, -0.0044, -0.0048, -0.0052, -0.0057,
-0.0062, -0.0067, -0.0072, -0.0078, -0.0084, -0.0090, -0.0096, -0.0103, -0.0110, -0.0117,
-0.0125, -0.0132, -0.0140, -0.0149, -0.0157, -0.0166, -0.0175, -0.0185, -0.0194, -0.0205,
-0.0215, -0.0226, -0.0237, -0.0248, -0.0260, -0.0272, -0.0285, -0.0298,-0.0311, -0.0324,
-0.0338, -0.0353, -0.0368, -0.0383, -0.0398, -0.0414, -0.0431,-0.0448, -0.0465, -0.0483,
-0.0501, -0.0520, -0.0539, -0.0559, -0.0579, -0.0600, -0.0621, -0.0643, -0.0665, -0.0688,
-0.0712, -0.0736, -0.0760, -0.0786, -0.0812,-0.0838, -0.0865, -0.0893, -0.0921, -0.0950,
-0.0980, -0.1010, -0.1042, -0.1074,-0.1106, -0.1140, -0.1174, -0.1209, -0.1245, -0.1281,
-0.1319, -0.1357, -0.1396, -0.1437, -0.1478, -0.1520, -0.1563, -0.1607, -0.1652, -0.1698,
-0.1745, -0.1793,-0.1842, -0.1893, -0.1944, -0.1997, -0.2051, -0.2107, -0.2163, -0.2221,
-0.2280,-0.2341, -0.2403, -0.2466, -0.2531, -0.2597, -0.2665, -0.2735, -0.2806, -0.2879,
-0.2953, -0.3030, -0.3108, -0.3188, -0.3270, -0.3354, -0.3439, -0.3527, -0.3617,-0.3710,
-0.3804, -0.3901, -0.4000, -0.4101, -0.4205, -0.4312, -0.4421, -0.4533, -0.4648, -0.4766,
-0.4886, -0.5010, -0.5137, -0.5267, -0.5401, -0.5538, -0.5678,-0.5822, -0.5971, -0.6122,
-0.6278, -0.6439, -0.6603, -0.6772, -0.6946, -0.7124, -0.7308, -0.7496, -0.7690, -0.7889,
 -0.8094, -0.8305, -0.8522, -0.8745, -0.8975,-0.9212, -0.9455, -0.9707, -0.9965, -1.0232,
 -1.0507, -1.0791, -1.1083, -1.1385,-1.1697, -1.2019, -1.2351, -1.2695, -1.3050, -1.3417,
 -1.3796, -1.4189, -1.4596, -1.5017, -1.5453, -1.5905, -1.6374, -1.6860, -1.7365, -
1.7889,  -1.8434, -1.9000,-1.9588, -2.0201, -2.0839, -2.1503, -2.2196, -2.2919, -2.3673,
-2.4462,
-2.5286,-2.6148, -2.7051, -2.7997, -2.8990, -3.0032, -3.1127, -3.2280, -3.3493, -3.4772,
 -3.6123, -3.7550, -3.9060, -4.0660, -4.2358, -4.4163, -4.6084, -4.8133, -5.0321,-5.2663,
-5.5176, -5.7876, -6.0786, -6.3929, -6.7333, -7.1032, -7.5062, -7.9470, -8.4309, -8.9643,
 -9.5549, -10.2122, -10.9477,-11.7758, -12.7148, -13.7879, -15.0251,-16.4665, -18.1657, -
20.1973, -22.6673, -25.7320, -29.6317, -34.7556, -41.7789, -51.9841,-68.1434, 97.5884, -
168.4654, -759.1888);

   begin
  process(clk)
  begin
     if clk='1' and clk' event then
                for index in 0 to 255 loop
        LINEARb1f(index)<=To_float(LINEARb1(index));
        LINEARb1std(index)<=to_slv(LINEARb1f(index));
        LINEARb1int(index)<=conv_integer(LINEARb1std(index));
        LINEARb1revstd(index)<=conv_std_logic_vector(LINEARb1int(index),32);
        LINEARb1rev(index)<=To_float(LINEARb1revstd(index));
        end loop;
       end if;

end process;
end design;
```

Matlab

Matlab code used to get polynomial co-efficients

```
n_int_bits=8;
n_int=2^n_int_bits; % number of intervals
n_points_bits=32;
n_points=2^n_points_bits; % Number of bits in the given number
factor=1/2^1;
n_points_intv=factor*n_points/n_int; % Each interval has 2^32/2^8 points

for j=1:n_int;
    x=linspace ((j-1)/n_int,(j/n_int)-(1/n_points_intv),n_points_intv);
    lamda=1;
    y=-log (1 - x) /lamda; %inverse of exponent cdf


    poly1=polyfit(x,y,1); %linear fit
    A1(j)=poly1(1);
    B1(j)=poly1(2);

    poly2=polyfit(x,y,2); %quadratic fit
    A2(j)=poly2(1);
    B2(j)=poly2(2);
    C2(j)=poly2(3);

    poly3=polyfit(x,y,3); %quadratic fit
    A3(j)=poly3(1);
    B3(j)=poly3(2);
    C3(j)=poly3(3);
    D3(j)=poly3(4);
    j
end
```

Matlab code used to calculate Mean Square error

```
%polynomial co-efficients


N = 20000000;
x = 0:1:N-1;
x = x./N;
y = -log(1-x);

n_order = 12;
n_intervals = 16;

mse = zeros(n_order,n_intervals+1);


        order = 8
        interval_pow = 12

        n_int = 2^interval_pow;


        testy = zeros(1,N);
        for j = 0:n_int-1
            lowerindex = floor(N*(j)/n_int+1);
            upperindex = floor(N*(j+1)/n_int);
            poly1 = polyfit(x(lowerindex:upperindex),y(lowerindex:upperindex),order);
            testy(lowerindex:upperindex) = polyval(poly1,x(lowerindex:upperindex));
        end
```

mse(order,interval_pow+1) = sum((y-testy).^2) / N;