Reconfigurable Hardware for Function Evaluation and LDPC Coding

MPhil/PhD Transfer Report

Dong-U Lee
Department of Computing
Imperial College London
United Kingdom
dong.lee@ic.ac.uk

July 2003

Abstract

Three main topics are presented in this report: function evaluation, Gaussian noise generation, and Low-Density Parity-Check (LDPC) code encoding. The function evaluator uses first order polynomials to approximate elementary or special purpose functions. The novelty of our approach is the use of non-uniform segments, in which the segment sizes can be adjusted to conform to the non-linearities for the function to be approximated. A simple cascade of AND and OR gates can be used to rapidly calculate the segment addresses for a given input. Scaling factors are used to deal with large polynomial coefficients, trading precision with range. Two functions are used for case studies: $\sqrt{-\ln(x)}$ and $\cos(2\pi x)$. Results indicate significant improvements of our non-uniform segment approach over the traditional uniform approach, especially for functions with high non-linearities. The hardware Gaussian noise generator is designed to facilitate channel code simulations involving very large numbers of samples. Our non-uniform segment function evaluator is used to compute the trigonometric and logarithmic functions needed for the generation of the noise samples. The design occupies approximately 10% of a Xilinx Virtex-II XC2V4000-6 chip and 90% of a Xilinx Spartan-IIE XC2S300E-7, and can produce 133 million samples per second. Various statistical tests as well as application to LDPC decoding are used to confirm the quality of the noise samples. Finally, an efficient hardware encoder for irregular LDPC codes is developed. It is based on Richardson and Urbanke's encoding algorithm and takes up 3% of a Xilinx Virtex-II XC2V4000-6 device. It runs at 133MHz and is capable of a throughput of 10 million bits per second. The codewords generated from our encoder are verified against our software model for correctness.

Contents

1 Introduction **4**
1.1 Objectives and Contributions 4
1.2 History 4
1.3 Overview of our Approach 5

2 Background **9**
2.1 Introduction 9
2.2 Function Evaluation 9
2.2.1 CORDIC 9
2.2.2 Bipartite Method 10
2.2.3 Polynomial Approximation 11
2.2.4 Types of Errors 12
2.3 Gaussian Noise Generation 12
2.4 LDPC Codes 13
2.4.1 Basics of LDPC Codes 13
2.4.2 RU LDPC Encoding Method 15
2.4.3 Hardware Aspects of LDPC codes 20
2.5 Summary 20

3 Function Evaluation **21**
3.1 Introduction 21
3.2 Non-uniform Segmentation 21
3.3 Hardware Architecture 24
3.4 Placement of Segment Boundaries 24
3.5 Evaluation and Results 26
3.6 Summary 30

4 Gaussian Noise Generation **31**
4.1 Introduction 31
4.2 Architecture 32
4.3 Implementation 36
4.4 Evaluation and Results 38
4.5 Wallace's Method 42
4.6 Summary 43

5 LDPC Encoding **44**
5.1 Introduction 44
5.2 Overview 44
5.3 Preprocessing 45
5.4 Hardware Encoder Architecture 45
5.4.1 Matrix-vector multiplication 45
5.4.2 Back-substitution 46
5.5 Implementation and Results 48
5.6 Summary 49

6 Conclusion **50**
6.1 Summary 50
6.2 Research Plan 51

Chapter 1

Introduction

1.1 Objectives and Contributions

The objective of this report is to explore the use of reconfigurable hardware for function evaluation and Low-Density Parity-Check (LDPC) coding. Our main contributions are:

- An efficient function evaluator based on polynomial approximation with non-uniform segments.
- A hardware Gaussian noise generator that satisfies the chi-square and Kolmogorov-Smirnov tests, which is capable of producing 133 million samples per second with 10% resource usage on a Xilinx XC2V4000-6 FPGA.
- An encoder for irregular LDPC codes capable of a throughput of 10 million bits per second with 3% resource usage on a Xilinx XC2V4000-6 FPGA.

The rest of this chapter provides historical information and an overview of the material in Chapters 3-5. Chapter 2 covers background material and previous work. Chapter 3 describes a hardware function evaluator based on polynomial approximation with non-uniform segments. Chapter 4 presents a hardware Gaussian noise generator used for channel code evaluation. Chapter 5 describes an efficient hardware encoder for irregular LDPC codes, and Chapter 6 offers conclusion and future work.

1.2 History

Error correcting coding (ECC) is a critical part of modern communications systems, where it is used to detect and correct errors introduced during a transmission over a channel [10], [73]. It relies on transmitting the data in an encoded form, such that the redundancy introduced by the coding allows a decoding device at the receiver to detect and correct errors. In this way, no request for retransmission is required, unlike systems which only detect errors (usually by means of a checksum transmitted with the data). In many applications, a substantial portion of the baseband signal processing is dedicated to ECC. The wide range of ECC applications [22] include space and satellite communications,

data transmission, data storage and mobile communications. NASA's Voyager, Galileo and Cassini missions would not have been possible without the use of ECC.

In 1948, Claude Shannon founded the field of study "Information Theory" which is the basis of modern ECC with his discovery of the noisy channel coding theorem [104]. The theoretical contribution of Shannon's work was a useful definition of "information" and several "channel coding theorems" which gave explicit upper bounds, called the channel capacity, on the rate at which information could be transmitted reliably on a given communication channel. In the context of our work, the result of primary interest is the "noisy channel coding theorem for continuous channels with average power limitations". This theorem states that the capacity C (which is now known as the Shannon limit) of a bandlimited Additive White Gaussian Noise (AWGN) channel with bandwidth W, a channel model that approximately represents many practical digital communication and storage systems, is given by

$$C = W \log_2(1 + E_s/N_0) \text{ bits per second (bps)} \tag{1.1}$$

where E_s is the average signal energy in each signaling interval of duration $T = 1/W$, and $N_0/2$ is the two-sided noise power spectral density. Perfect Nyquist signalling is assumed. The proof of this theorem demonstrates that for any transmission rate R less than or equal to the channel capacity C, there exists a coding scheme that achieves an arbitrarily small probability of error; conversely, if R is greater than C, no coding scheme can achieve reliable performance. Since this theorem was published, an entire field of study has grown out of attempts to design coding schemes that approach the Shannon limit of various channels.

In the past few years, LDPC codes have received much attention because of their excellent performance, and have been widely considered as the most promising candidate ECC scheme for many applications in telecommunications and storage devices [79], [107], [122]. LDPC codes were first proposed by Gallager in 1962 [33], [34]. He defined an (n, d_v, d_c) LDPC code as a code of block length n in which each column of the parity check matrix contains d_v ones and each row contains d_c ones. Due to the regular structure (uniform column and row weight) of Gallager's codes, they are now called regular LDPC codes. Gallager provided simulation results for codes with block lengths of the order of hundreds of bits. The results indicated that LDPC codes have very good potential for error correction. However, the storage and computation requirements interrupted the research on LDPC codes. After the discovery of Turbo codes by Berrou et al. in 1993 [7], MacKay [63] reestablished the interest in LDPC codes during the mid to late 1990s.

1.3 Overview of our Approach

Evaluations of LDPC codes are based on computer simulations which can be time consuming, particularly when the behavior at low bit error rates (BERs) in the error floor region is being studied. Tremendous efforts have been devoted to analyze and improve their error-correcting performance, but little consideration

has been given to the practical LDPC codec hardware implementations. If the binary Hamming distance between all combinations of codewords (the distance spectrum) is known, then analytic techniques for describing the performance of the codes in the presence of noise is available. However, in the case of capacity achieving random linear codes (such as LDPC codes), the problem of finding the distance spectrum of the code is intractable and researchers resort to the use of Monte Carlo simulation in order to characterize various code constructions in terms BER versus signal to noise ratio (SNR). At very low SNRs, errors occur often and a sufficient statistic can be gathered readily within a workstation. However at higher SNRs where errors occur rarely, the situation is different. Thorough characterization of a code in this region may require simulation of $10^{10}-10^{12}$ code symbols, and workstation simulations provide inadequate means of finding statistically sufficient set of error events.

Hardware-based simulation [57] offers the potential of speeding up code evaluation by several orders of magnitude. Such simulation framework consists of three main blocks: encoder, noise channel and decoder, where the noise channel is generally modelled with Gaussian noise. Our LDPC code simulations are run on a reconfigurable engine, which consists of a PC and a reconfigurable platform [53]. The reconfigurable platform used is a Virtex-II FPGA prototyping board from Nallatech [78] shown in Figure 1.1. A block diagram of our LDPC simulation framework is provided in Figure 1.2. The LDPC encoder follows an algorithm suggested in [95]. Our noise generator block improves the overall value of the system as a Monte Carlo simulator, since noise quality at high signal to noise ratios (tails of the Gaussian) is essential. Since the LDPC decoding process is iterative and the number of required iterations is non-deterministic, a flow control buffer is used to greatly increase the throughput of the overall system.

The generation of Gaussian noise involves the computation of two functions: $\sqrt{-\ln(x)}$ and $\cos(2\pi x)$. The accuracy and speed in computing these functions are essential for generating high-quality Gaussian noise rapidly. We present in Chapter 3 a hardware function evaluator based on polynomial approximation with non-uniform segments. The novel use of non-uniform segments enables us to approximate non-linear regions of a function particularly well. The appropriate segment address for a given function can be rapidly calculated in run time by a simple combinatorial circuit. Scaling factors are used to deal with large polynomial coefficients and to trade precision with range. Our function evaluator is based on first-order polynomials, and is suitable for applications requiring high performance with small area, at the expense of accuracy.

The Gaussian noise generator in Chapter 4 is used as a key component in a hardware simulation system, for exploring channel code behavior at very low BERs in the range of 10^{-9} to 10^{-10}. The output of the noise generator accurately models a true Gaussian PDF even at very high σ values. Its properties are explored using: (a) several different statistical tests, including the chi-square test and the Kolmogorov-Smirnov test, and (b) an application for decoding of LDPC codes. An implementation at 133MHz on a Xilinx Virtex-II XC2V4000-6 FPGA produces 133 million samples per second, which is 40 times faster than a 2.13GHz PC; another implementation on a Xilinx Spartan-IIE XC2S300E-7

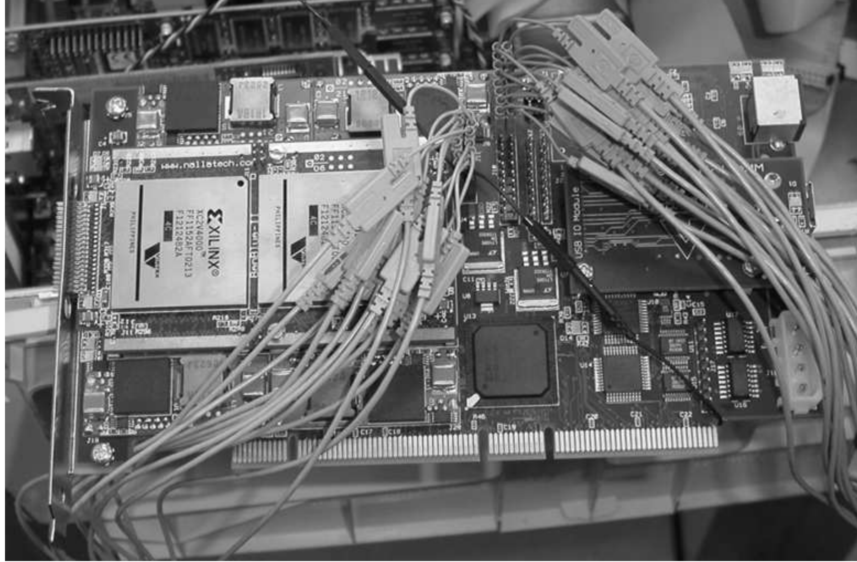

Figure 1.1: The BenONE board from Nallatech used to run our designs. It consists of two Xilinx Virtex-II XC2V4000-6 FPGAs and 4MB of SRAM. The board can be connected to a PC via the PCI bus or USB. The grey wires are connected to a logic analyzer for debugging purposes.

FPGA at 62MHz is capable of a 20 times speedup. The performance can be improved by exploiting parallelism: an XC2V4000-6 FPGA with nine parallel instances of the noise generator at 105MHz can run 300 times faster than a 2.13GHz PC.

In Chapter 5, we describe a hardware implementation of an efficient encoder for irregular LDPC codes. The design takes just 3% of resources on a Xilinx Virtex-II XC2V4000-6 device. It is capable of running at 133MHz resulting in a throughput of 10MBps. The codewords generated from our encoder have been validated against our software simulation model for correctness. This block will be placed in front of the noise generator in our LDPC simulation framework.

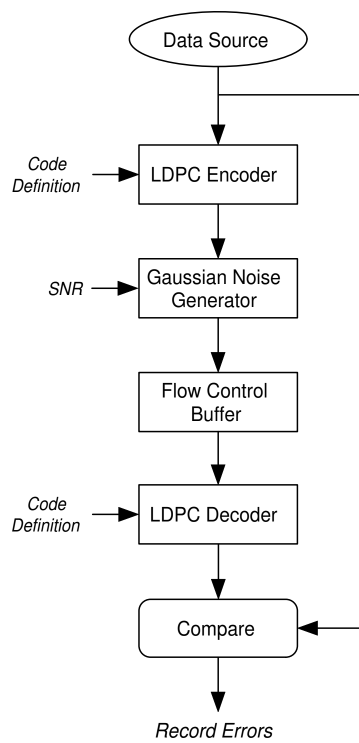

Figure 1.2: LDPC simulation framework.

Chapter 2

Background

2.1 Introduction

The purpose of this chapter is to present the background material and related work of this report. Section 2.2 describes three popular methods for approximating functions: CORDIC, the bipartite method and polynomial approximation. We also describe the different kinds of errors that can occur when functions are approximated. Section 2.3 discusses various ways of generating Gaussian noise and explores the existing work in this area. Finally, Section 2.4 introduces the basics of LDPC codes, describes Richardson and Urbanke's (RU) method for efficiently encoding irregular LDPC codes and looks at previous work on hardware related issues on LDPC codes.

2.2 Function Evaluation

Many FPGA applications including digital signal processing, computer graphics and scientific computing require the evaluation of elementary or special purpose functions. For applications that require low precision approximation at high speeds, full look-up tables are often employed. However, this becomes impractical for precisions higher than a few bits, because the size of the table is exponential in the input size. Three well known methods are described, which are generally used for evaluating functions.

2.2.1 CORDIC

CORDIC is an acronym for COordinate Rotations DIgital Computer and offers the opportunity to calculate desired functions in a rather simple and elegant way. The CORDIC algorithm was first introduced by Volder [116] for the computation of trigonometric functions, multiplication, division and data type conversion, and later generalized to hyperbolic functions by Walther [118]. It has found its way into diverse applications including the 8087 math coprocessor [27], the HP-35 calculator, radar signal processors and robotics.

It is based on simple iterative equations, involving only shift and add operations and was developed in an effort to avoid the time consuming multiply and

divide operations. The general CORDIC algorithm consists of the following three iterative equations:

$$
\begin{aligned}
x_{k+1} &= x_k - m\delta_k y_k 2^{-k} \\
y_{k+1} &= y_k + \delta_k x_k 2^{-k} \\
z_{k+1} &= z_k - \delta_k \sigma_k
\end{aligned}
$$

The constants m, δ_k and σ_k depend on the specific computation being performed, as explained below.

- m is either 0, 1 or -1. $m = 1$ is used for trigonometric and inverse trigonometric functions. $m = -1$ is used for hyperbolic, inverse hyperbolic, exponential and logarithmic functions, as well as square roots. Finally, $m = 1$ is used for multiplication and division.
- δ_k is one of the following two signum functions:

$$
\delta_k = \text{sgn}(z_k) = \begin{cases} 1, & z_k \geq 0 \\ -1, & z_k < 0 \end{cases} \quad \text{or} \quad \delta_k = \text{-sgn}(y_k) = \begin{cases} 1, & y_k < 0 \\ -1, & y_k \geq 0 \end{cases}
$$

  The first is often called the rotation mode, in which the z values are driven to zero, whereas the second is the vectoring mode, in which the y values are driven to zero. Note that δ_k requires nothing more than a comparison.
- The numbers σ_k are constants stored in a table which depend on the value of m. For $m = 1$, $\sigma_k = \tan^{-1} 2^{-k}$; for $m = 0$, $\sigma_k = 2^{-k}$; and for $m = -1$, $\sigma_k = \tanh^{-1} 2^{-k}$.

To use these equations, appropriate starting values x_1, y_1 and z_1 must be given. One of these inputs, say z_1, might be the number whose hyperbolic sine we wish to approximate, $\sinh z_1$. In all cases, the starting values must be restricted to a certain interval about the origin in order to ensure convergence. As the iterations proceed, one of the variables tends to zero while another variable approaches the desired approximation.

The major disadvantage of the CORDIC algorithm is its linear convergence resulting in an execution time which is linearly proportional to the number of bits in the operands. In addition, CORDIC is limited to a relatively small set of elementary functions. A comprehensive study of CORDIC algorithms on FPGAs can be found in [3].

2.2.2 Bipartite Method

The bipartite method was originally introduced by Das Sarma and Matula [24], with the aim of getting accurate reciprocals. Generalizations and improvements were suggested by Schulte and Stine [102], [103], Muller [77], and de Denechin and Tisserand [25].

Assume an n-bit, binary fixed-point system, and assume that n is a multiple of 3, $n = 3k$. We wish to design a table-based implementation of function f.

A full look-up table would lead to a table of size $n \times 2^n$. Instead, we split the input word x into three k-bit words x_0, x_1, and x_2, that is,

$$x = x_0 + x_1 2^{-k} + x_2 2^{-2k} \tag{2.1}$$

where x_0, x_1 and x_2 are multiples of 2^{-k} that are less than 1. The original bipartite method consists in approximating the first order Taylor expansion

$$f(x) = f(x_0 + x_1 2^{-k}) + x_2 2^{-2k} f'(x_0 + x_1 2^{-k}) + x_2^2 2^{-4k} f''(\xi), \quad \xi \in [x_0 + x_1 2^{-k}, x] \tag{2.2}$$

by

$$f(x) = f(x_0 + x_1 2^{-k}) + x_2 2^{-2k} f'(x_0). \tag{2.3}$$

That is, $f(x)$ is approximated by the sum of two functions $\alpha(x_0, x_1)$ and $\beta(x_0, x_2)$, where

$$\begin{cases} \alpha(x_0, x_1) & = & f(x_0 + x_1 2^{-k}) \\ \beta(x_0, x_2) & = & x_2 2^{-2k} f'(x_0) \end{cases}$$

The error of this approximation is roughly proportional to 2^{-3k}. Instead of directly tabulating function f, functions α and β are tabulated. Since they are functions of $2k$ bits only, each of these tables has $2^{2n/3}$ entries. This results in a total table size of $2n \times 2^{2n/3}$ bits, which is a significant improvement over the full look-up table.

2.2.3 Polynomial Approximation

Polynomial approximation [31], [39], [93] involves approximating a continuous function f with one or more polynomials p of degree n on a closed interval $[a, b]$. There are two kinds of approximations: least squares approximations that minimize the average error, and least maximum approximations that minimize the worst-case error [76]. In both cases, the aim is to minimize a distance $\|p - f\|$. For least squares approximations, that distance is:

$$\|p - f\|_2 = \sqrt{\int_a^b w(x)(f(x) - p(x))^2 dx}, \tag{2.4}$$

where w is a continuous weight function for selecting parts of $[a, b]$ where we want the approximation to be more accurate. For least maximum (minimax) approximations, the distance is:

$$\|p - f\|_\infty = \max_{a \le x \le b} |f(x) - p(x)|. \tag{2.5}$$

Our work is based on minimax polynomial approximations, which involves minimizing the worst-case error. Since we are interested in fixed-point number representation in our work, we will be concerned with the worst-case absolute errors. If we were to work with floating-point numbers, we would have to minimize the relative errors. A recent study of minimax polynomial approximation

on FPGAs can be found [105]. Much of the work on function evaluation are generally concerned with producing highly accurate approximation with highly complex designs. Instead, we will focus on applications that require high speed and small area but relatively low accuracy. Examples of such applications include Gaussian noise generation [54] and belief propagation in LDPC decoding [106]. We will focus on first-order polynomials of the form $p(x) = c_1 \times x + c_0$, where c_1 is the gradient and c_0 is the y-intercept, which can be computed by two table look-ups, a multiplication and an addition.

Previous work on polynomial approximations involves equally sized segments [14], [24], [26], [40], [46], [58], [91], [101], [102]. Many variants of this general idea have been suggested. For instance, Piñeiro et al. [91] divide the input interval into around 2^8 subintervals. They store, for each subinterval a degree-2 minimax approximation, and accumulate the partial terms in a fused accumulation tree. Cao et al. [14] store function values instead of coefficients and perform interpolation, using fewer look-up table memory entries, at the expense of additional hardware and extra time for calculating the coefficients on-the-fly. Approximations using such uniform segments are suitable for functions with linear regions, but they can be inefficient for non-linear functions. It is desirable to choose the boundaries of the segments to cater for the non-linearities of the function. Highly non-linear regions may need smaller segments than linear regions. This approach minimizes the amount of storage required to approximate the function, leading to more compact and efficient designs.

2.2.4 Types of Errors

Classically, we have three different kinds of error which affect to the global error of an evaluation of a function:

- The input quantization error measures the fact that an input number usually represents a small interval centered around this number.
- The approximation error measures the difference between the pure mathematical function and the approximate mathematical function that will be used to evaluate it.
- Output rounding errors measure the difference between the approximated mathematical function and the closest machine-representable value.

2.3 Gaussian Noise Generation

Sequences of random numbers with Gaussian probability distribution functions are needed to simulate a wide variety of natural phenomena [35], [119]. Applications of such sequences include channel code evaluation [54], watermarking [28], oscilloscope testing [113], simulation of economic systems [6], [98] and molecular dynamics simulations [47].

Previous work on Gaussian noise generation can be divided into two types: the generation of Gaussian noise using a combination of analog components [88], [97], [127], and the generation of pseudo random noise using purely digital components [4], [11], [18], [23], [36], [38], [45], [56], [75], [81], [109], [117], [120].

The first method tends to be practical only in highly restricted circumstances, and suffers from its own problems with noise accuracy. The second method is often more desirable, because of its flexibility and high performance. In addition, when simulating communication systems we may wish to use pseudo random noise so that we can adopt the same noise for different systems. Also, if the system fails we may wish to know which noise samples cause the system to fail. Comprehensive but rather dated comparisons of such digital methods can be found in [5], [74] and [87].

Digital methods for generating random Gaussian variables are almost always based on transformations or operations on uniform random variables. There are four well-known methods [100]: the Ziggurrat method, the polar method, the use of the central limit theorem and the Box-Muller method. The Ziggurrat method [66], [67] is not considered for our work because it can produce large errors in the tail areas of the distribution. The polar method, while popular in software implementations, contains a conditional loop such that the output rate is not constant, making it less amenable to a hardware simulation environment. The central limit theorem can, in principle, be used to produce Gaussian samples, if a suitable number of samples are involved. In practice however, approaching a Gaussian PDF to a high accuracy using the central limit theorem alone would require an impractically large number of samples. Our choice for hardware implementation is based on the Box-Muller algorithm, which generates random Gaussian variables by transforming two uniform random variables over [0,1]. Properly implemented, it offers predictable output rate and, in combination with the central limit theorem, extremely good Gaussian modelling.

There is very little previous work on practical digital hardware Gaussian noise generators. The most relevant publications are probably [11], [23] and [120], which discuss designs targeting Field-Programmable Gate Arrays (FPGAs). The designs in these papers are based on the technique described by Ghazel et al. [36], which employs a recursive direct look-up table structure. Unfortunately, our tests show that their noise samples fail the normality tests. Moreover, their designs produce noise samples that are targeted primarily for the output region below about 4σ, and therefore does not specifically address the high σ values of 4σ to 6σ and beyond; these are critical in the large simulations motivating our work. Our design has significantly improved efficiency, passes statistical tests widely used for testing normality and produces high σ values.

2.4 LDPC Codes

2.4.1 Basics of LDPC Codes

LDPC codes exhibit performance extremely close to the best possible as determined by the Shannon capacity formula. For the AWGN channel, the best code of rate on-half presented in [94] has a threshold within 0.06dB from capacity, and their simulation results show that the best LDPC code of length 10^6 achieves a bit error probability of 10^{-6} less than 0.13dB away from capacity, beating the best codes known so far. A comparison of codes is illustrated in Figure 2.1.

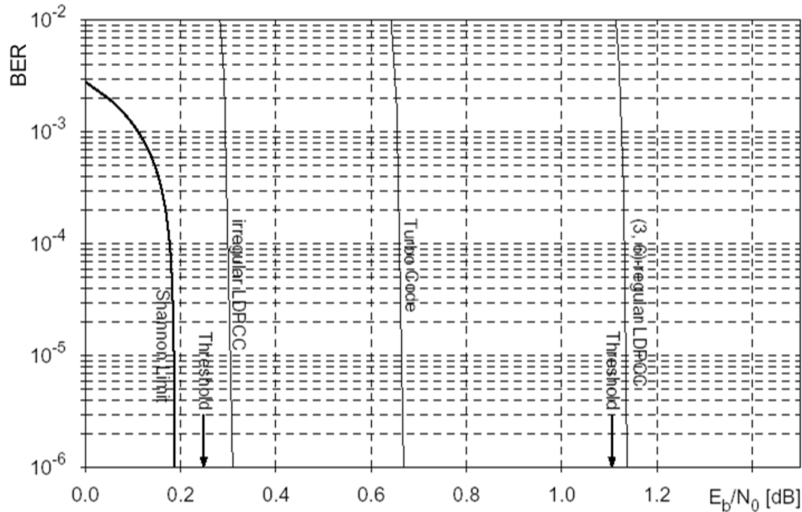

Figure 2.1: Comparison of (3,6)-regular LDPC code, Turbo code and optimized irregular LDPC code. All codes are length of 10^6 and rate one-half. The bit error rate for the AWGN channel is shown as a function of E_b/N_0 (signal to noise ratio per bit in dB).

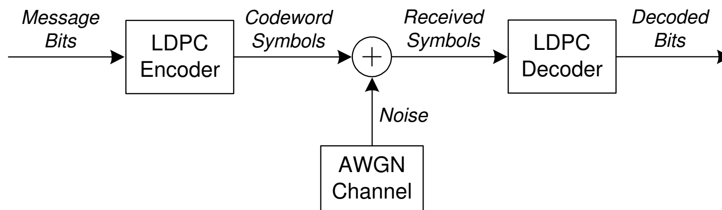

Figure 2.2: LDPC communication system model.

The communication system model we consider comprises of LDPC encoder, decoder and an AWGN channel as shown in Figure 2.2. Message bits are given as inputs to the LDPC encoder, which creates parity bits for a block of message generating codewords. A binary antipodal modulation is assumed at the transmitter. The signal gets corrupted by AWGN noise during the transmission over the channel. At the receiver end, the demodulator demodulates the received signal, filters it and performs A/D conversion on it. This is further fed to the LDPC decoder, which iteratively decodes the received block of codeword and provides decoded bits at the output end.

As originally suggested by Tanner [110], LDPC codes are well represented by bipartite graphs in which one set of nodes, the variable nodes, corresponds to elements of the codeword and the other set of nodes, the check nodes, corresponds to the set of parity-check constraints which define the code. Regular LDPC codes are those for which all nodes of the same type have the same degree. For example, a (3,6)-regular LDPC code has a graphical representation in which all variable nodes have degree 3 and all check nodes have degree 6.

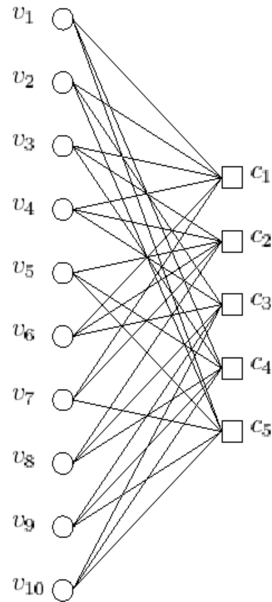

Figure 2.3: A (3,6)-regular LDPC code of length 10 and rate one-half. There are 10 variable nodes and five check nodes. For each check node check node C_i the sum (over GF(2)) of all adjacent variable node is equal to zero.

The bipartite graph determining such a code is shown in Figure 2.3. Irregular LDPC codes were introduced in [59] and [62] and were further studied in [60], [61] and [64]. For such an irregular LDPC code, the degrees of each set of nodes are chosen according to some distribution. Thus, an irregular LDPC code might have a graphical representation in which half the variable nodes have degree 3 and half have degree 5, while half the constraint nodes have degree 6 and half have degree 8.

LDPC codes are linear codes. Hence, they can be expressed as the null space of a parity check matrix H, i.e., x is a code word if and only if

$$Hx^T = 0^T. \tag{2.6}$$

The sparseness of H enables efficient (sub-optimal) decoding, while the randomness ensures (in the probabilistic sense) a good code. The H matrix corresponding to the bipartite graph in Figure 2.3 is shown below.

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{2.7}$$

2.4.2 RU LDPC Encoding Method

In this section, we shall describe the RU algorithm for constructing efficient encoders for LDPC codes as described in [95]. The efficiency of the encoder

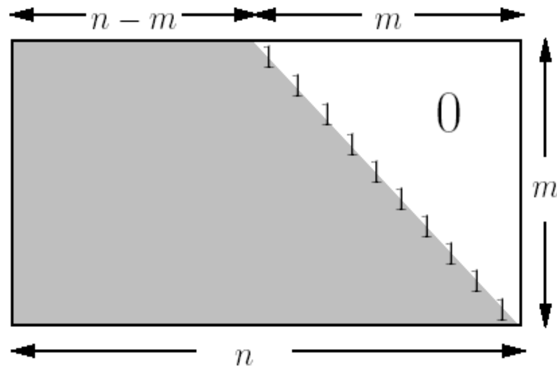

Figure 2.4: An equivalent parity-check matrix in lower triangular form.

arises from the sparseness of the parity check matrix H and the algorithm can be applied to any (sparse) H. Although our example is binary, the algorithm applies generally to matrices H whose entries belong to a field F. We assume throughout that the rows of H are linearly independent. If the rows are linearly dependent, then the algorithm which constructs the encoder will detect the dependency and either one can choose a different matrix H or one can eliminate the redundant rows from H in the encoding process.

Assume we are given an $m \times n$ parity-check matrix H over F. By definition, the associated code consists of the set of n-tuples x over F such that

$$Hx^T = 0^T. \tag{2.8}$$

The most straight forward way of constructing an encoder for such a code is the following. By means of Gaussian elimination bring H into an equivalent lower triangular form as shown in Figure 2.4. Split the vector x into a systematic part s, $s \in F^{n-m}$, and a parity part p, $p \in F^m$, such that $x = (x, p)$. Construct a systematic encoder as follows: i) Fill s with the $(n - m)$ desired information symbols. ii) Determine the m parity-check symbols using back-substitution. More precisely, for $l \in [m]$ calculate

$$p_l = \sum_{j=1}^{n-m} H_{l,j} s_j + \sum_{j=1}^{l-1} H_{l,j+n-m} p_j. \tag{2.9}$$

Bringing the matrix H into the desired form requires $O(n^3)$ operations of preprocessing. The actual encoding then requires $O(n^2)$ operations since, in general, after the preprocessing the matrix will no longer be sparse.

Given that the original parity-check matrix H is sparse, one might wonder if encoding can be accomplished in $O(n)$. As it will be shown, typically for codes which allow transmission at rates close to capacity, linear time encoding is indeed possible.

Assume that by performing row and column permutations only we can bring the parity check matrix into the form indicated in Figure 2.5. We say that H is in approximate lower triangular form. Note that since this transformation was

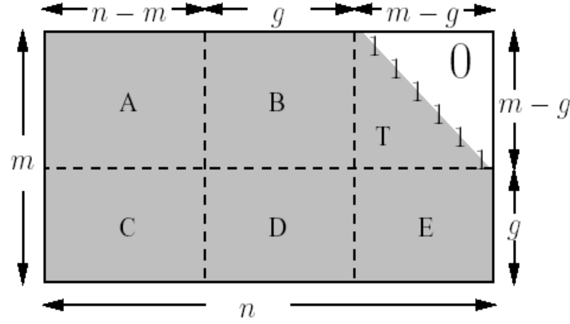

Figure 2.5: The parity-check matrix in approximate lower triangular form

accomplished solely by permutations, the matrix is still sparse. More precisely, assume that we bring the matrix in the form

$$H = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix} \tag{2.10}$$

where A is $(m-g)\times(n-m)$, B is $(m-g)\times g$, C is $g\times(n-m)$, D is $g\times g$, and, finally, E is $g\times(m-g)$. Further, all these matrices are sparse and T is lower triangular with ones along the diagonal. Multiplying this matrix from the left by

$$\begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix} \tag{2.11}$$

we get

$$\begin{bmatrix} A & B & T \\ -ET^{-1}A+C & -ET^{-1}B+D & 0 \end{bmatrix} \tag{2.12}$$

Let $x = (s, p_1, p_2)$ where s denotes the systematic part, p_1 and p_2 combined denote the parity part, p_1 has length g, and p_2 has length $(m-g)$. The defining equation $Hx^T = 0^T$ splits naturally into two equations, namely

$$As^T + Bp_1^T + Tp_2^T = 0 \tag{2.13}$$

and

$$(-ET^{-1}A+C)s^T + (-ET^{-1}B+D)p_1^T = 0. \tag{2.14}$$

Define $\phi := -ET^{-1}B + D$ and assume for the moment that ϕ is nonsingular. The general case will be discussed shortly. Then from (2.14) we conclude that

$$p_1^T = -\phi^{-1}(-ET^{-1}A+C)s^T. \tag{2.15}$$

Hence, once the $g\times(n-m)$ matrix $-\phi^{-1}(-ET^{-1}A+C)$ has been precomputed, the determination of p_1 can be accomplished in complexity $O(g\times(n-m))$ simply by performing a multiplication with this matrix. This complexity can be further

reduced as shown in Table 2.1. Rather than precomputing $-\phi^{-1}(-ET^{-1}A+C)$ and then multiplying with s^T we can determine p_1 by breaking the computation into several smaller steps, each of which is efficiently computable.

To this end, we first determine As^T, which has complexity $O(n)$ since A is sparse. Next, we multiply the result by T^{-1}. Since $T^{-1}[As^T] = y^T$ is equivalent to the system $[As^T] = Ty^T$ this can also be accomplished in $O(n)$ by back-substitution, since T is lower triangular and also sparse. The remaining steps are fairly straightforward. It follows that the overall complexity of determining p_1 is $O(n+g^2)$. In a similar manner noting from (2.13) that $p_2^T = -T^{-1}(As^T + Bp_1^T)$, we can accomplish the determination of p_2 in complexity $O(n)$ as shown step by step in Table 2.2.

Table 2.1: Efficient computation of $p_1^T = -\phi^{-1}(-ET^{-1}A + C)s^T$.

Operation	Comment	Complexity
As^T	multiplication by sparse matrix	$O(n)$
$T^{-1}[As^T]$	$T^{-1}[As^T] = y^T \Leftrightarrow [As^T] = Ty^T$	$O(n)$
$-E[T^{-1}As^T]$	multiplication by sparse matrix	$O(n)$
Cs^T	multiplication by sparse matrix	$O(n)$
$[-ET^{-1}As^T] + [Cs^T]$	addition	$O(n)$
$-\phi^{-1}[-ET^{-1}As^T + Cs^T]$	multiplication by dense $g \times g$ matrix	$O(g^2)$

Table 2.2: Efficient computation of $p_2^T = -T^{-1}(As^T + Bp_1^T)$.

Operation	Comment	Complexity
As^T	multiplication by sparse matrix	$O(n)$
Bp_1^T	multiplication by sparse matrix	$O(n)$
$[As^T] + [Bp_1^T]$	addition	$O(n)$
$-T^{-1}[As^T + Bp_1^T]$	$-T^{-1}[As^T + Bp_1^T] = y^T \Leftrightarrow -[As^T + Bp_1^T] = Ty^T$	$O(n)$

A summary of the RU encoding procedure is given in Table 2.3. It consists of two steps. A preprocessing step and the actual encoding step. In the preprocessing step, we first perform row and column permutations to bring the parity-check matrix into approximate lower triangular form with as small a gap

g as possible. We also need to check whether $\phi := -ET^{-1}B + D$ is nonsingular. Rather than premultiplying by the matrix $\begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix}$, this task can be accomplished efficiently by Gaussian elimination. If, after clearing the matrix E the resulting matrix ϕ is seen to be singular we can simply perform further column permutations to remove this singularity. This is always possible when H is not rank deficient, as assumed. The actual encoding then entails the steps listed in Tables 2.1 and 2.2.

Table 2.3: Summary of the RU encoding procedure. It entails two steps: a preprocessing step and the actual encoding step.

Preprocessing: Input: Non-singular parity-check matrix H. Output: An equivalent parity check matrix of the form $\begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix}$ such that $-ET^{-1}B+D$ is non-singular.

1. Perform row and column permutations to bring the parity check matrix H into approximate lower triangular form

$$H = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix} \tag{2.16}$$

   with as small a gap g as possible. We will see in subsequent sections how this can be accomplished efficiently.

2. Use Gaussian elimination to effectively perform the pre-multiplication.

$$\begin{bmatrix} I & 0 \\ ET^{-1} & I \end{bmatrix} \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix} = \begin{bmatrix} A & B & T \\ -ET^{-1}A + C & -ET^{-1}B + D & 0 \end{bmatrix} \tag{2.17}$$

   in order to check that $-ET^{-1}B + D$ is non-singular, performing further column permutations is necessary to ensure this property. (Singularity of H can be detected at this point.)

Encoding: Input: Parity-check matrix of the form $\begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix}$ such that $-ET^{-1}B + D$ is non-singular and a vector $s \in F^{n-m}$. Output: The vector $x = (s, p_1, p_2), p_1 \in F^g, p_2 \in F^{m-g}$, such that $Hx^T = 0^T$.

1. Determine p_1 as shown in Table 2.1.

2. Determine p_2 as shown in Table 2.2.

2.4.3 Hardware Aspects of LDPC codes

Hardware based channel code simulations are desirable, since they are several orders of magnitudes faster than software based methods. FPGAs are ideal candidates for this purpose because of their speed and flexibility. Although there have been plenty of publications on the hardware implementation of Viterbi [15], [17], [82], [108], [121] and Turbo codes [42], [68], [89], [90], [114] very little attention has been given to the hardware implementation issues of LDPC codes.

Levine and Schmidt [57] present a simple hardware architecture for the LDPC codec, but it has not been implemented and is perhaps not practical for a real design due to the large size and inefficiency. Zhang et al. investigate the finite precision effects in regular LDPC decoders in [124]. They introduce their hardware architecture in [125] and present a FPGA based (3,6)-regular LDPC decoder in [126]. Their design is capable of 54MBps, but its error correcting performance is rather poor due to the use of regular LDPC codes and simplicity of their design. In [8], Bhatt et al. present a regular LDPC implementation on a fixed-point DSP, which achieves a bit rate of just 133.33KBps. In [43], Howland et al. present their parallel regular LDPC decoding architecture, and later published their ASIC based LDPC decoder chip in [9] and [44]. Their chip is claimed to be capable of 1GBps, but are again based on regular LDPC codes, which have significantly lower error correcting performance compared to irregular LDPC codes. Our closet competitors are probably Metha et al. and Flarion Technologies Inc. Metha et al. are working on a FPGA based regular LDPC simulation platform and recently published a technical report on their preliminary architecture [70]. Flarion offer intellectual properties for LDPC encoder and decoders [32]. Their FPGA decoder is reported to operate at up to 384MBps and their ASIC decoder at 10GBps. However, very little details are known due to them being commercial products.

We are currently working on a FPGA based irregular LDPC simulation framework with Prof. John Villasenor's group at University of California, Los Angeles. The work is still in progress and a paper describing our preliminary decoder architecture has been published by UCLA at MILCOM 2003 [106].

2.5 Summary

In this chapter, we have presented the background material and related work of this report. We have first described the different methods for approximating elementary and special purpose functions, which are the CORDIC method, the bipartite method and polynomial approximation. We also discussed the different types of errors that can occur when functions are approximated. Secondly, we have looked at various ways of generating Gaussian noise, which are used for various applications including channel code simulations. Finally, we have introduced the basics of LDPC codes, described the RU algorithm for efficient LDPC encoders and looked at previous work that deals with the hardware related issues of LDPC codes.

Chapter 3

Function Evaluation

3.1 Introduction

The purpose of this chapter is to present our hardware function evaluator based on polynomial approximations with non-uniform segments. Section 3.2 describes our non-uniform segmentation approach. Section 3.3 illustrates our hardware architecture. Section 3.4 describes a near optimum approach to place the boundaries of the segments. Section 3.5 evaluates our design and offers benchmarks, and Section 3.6 gives a summary of this chapter.

The evaluation of functions is often the performance bottleneck of many compute-bound applications. Examples of these functions include elementary functions such as $\ln(x)$ or $\sqrt{x}$, and compound functions such as $\sqrt{-\ln(x)}$ or $\tan^2(x)+1$. Computing these functions quickly and accurately is a major goal in computer arithmetic; software implementations are often too slow for numerically intensive or real-time applications. For instance, over 60% of the total run time is devoted to function evaluation operations in a simulation of a jet engine reported by O'Grady and Wang [80]. The performance of such applications depends on the design of a hardware function evaluator. Advanced FPGAs enable the development of low-cost and high-speed function evaluation units, customizable to particular applications. The principal contribution of this paper is a fast and efficient hardware function evaluator using polynomial approximations. The key novelties of our work include:

- a method for polynomial approximations with non-uniform segments;
- hardware architecture and implementation of the proposed method;
- evaluation of this method with a logarithmic function and a cosine function.

3.2 Non-uniform Segmentation

The interval of approximation $[a, b]$ is divided into a set of sub-intervals, called segments. The best-fit straight line, in a minimax sense, to each segment is found. A look-up table is used to store the coefficients for each line segment,

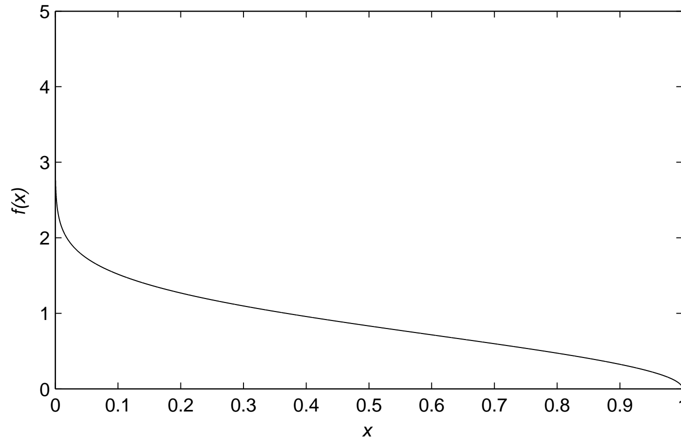

Figure 3.1: $\sqrt{-\ln(x)}$ over $(0, 1]$.

and the functions can then be evaluated using a multiplier and an adder to calculate the linear approximation [71].

Consider the evaluation of the function $\sqrt{-\ln(x)}$ over the interval $(0, 1]$. As shown in Figure 3.1, the greatest non-linearities of this function occur in the regions close to zero and one. It can be seen that if uniform segments are used, a large number of small segments would be required in order to get accurate approximations in the non-linear regions. However, in the middle part of the curve where it is relatively linear, accurate approximation can be obtained using relatively few segments. It would be efficient to use small segments for the non-linear regions, and large segments for linear regions. Arbitrary sized segments would enable us to have the least error for a given number of segments. However this approach has one disadvantage: the hardware to calculate the segment address for a given input can be complex. Our objective is to provide near arbitrary-sized segments with a simple circuit to find the segment address for a given input. We have developed a novel method which can construct piecewise linear approximation such that: (a) the segment lengths used in a given region depends on the local linearity, with more segments deployed for regions of higher non-linearity; and (b) the boundaries between segments are chosen such that the task of identifying which segment to use for a given input can be rapidly performed.

The proposed method consists of five steps.

1. Determine optimal placement of segment boundaries (see Section 3.4) – this would include dividing into regions such that in each region the function either monotonically increases or decreases.

2. For a non-linear region, if the non-linearity is monotonically increasing, then increase segment size by a factor of two or more at each step; if the non-linearity is monotonically decreasing, then reduce segment size by a factor of two or more at each step.

3. The segment addresses can be obtained by computing the prefixes [51]

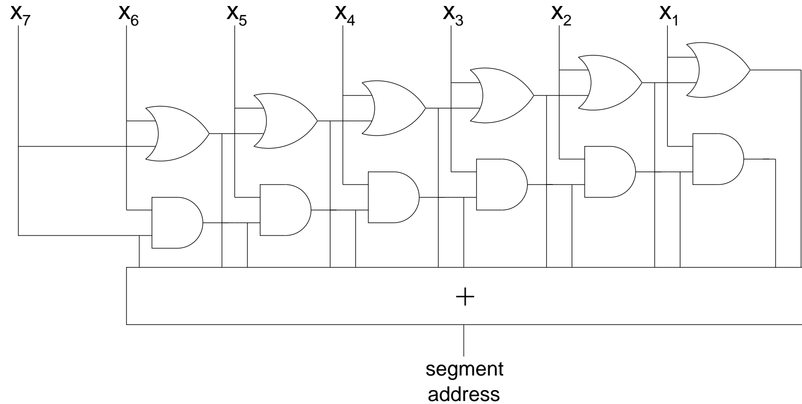

Figure 3.2: Circuit to calculate the segment address for a given input x. The adder counts the number of ones in the output of the two prefix circuits.

with a simple combinational or pipelined circuit.

4. If necessary, divide the function into several intervals, then apply step 1–3 (see the function $\cos(2\pi, x)$ in Section 3.5).

5. If necessary, repeat the above steps with higher-order terms.

As an example to illustrate our approach, consider approximating $\sqrt{-\ln(x)}$ with an 8-bit input (Figure 3.1). Using the traditional approach, the most-significant bits of x are used to index the uniform segments. For instance if the most-significant four bits are used, 16 uniform segments are used to approximate the function. Using our approach, it is possible to use small segments for non-linear regions (regions near 0 and 1), and large segments for linear regions (regions around 0.5). The idea is to use segments that grow by a factor of two from 0 to 0.5, and segments that shrink by a factor of two from 0.5 to 1 in the x-axis of Figure 3.1. We use segment boundaries at locations 2^{n-8} and $1 - 2^{-n}$ where $0 \leq n < 8$. Up to 14 segments can be formed this way. A circuit based on prefix computation can be used for calculating segment addresses (Figure 3.2) for a given input x. It checks for the number of leading zeros and ones to work out the segment address. A cascade of OR gates is used for segments that grow by factors of two, and a cascade of AND gates is used for segments that shrink by factors of two; these circuits can be pipelined and a circuit with shorter critical path but requiring more area can be used [51]. Note that the choice of segments does not have to be factors of two, it could be more. The appropriate taps are taken from the cascades depending on the choice of the segments and are added to work out the segment address. In Figure 3.2, the maximum available taps are taken, giving 14 segment addresses. Some taps would not be taken if the segments grow or shrink by more than a factor of two. It can be seen that the critical path of this circuit is from x_6 or x_7 to the output of the adder. By introducing pipeline registers between the gates, higher throughput can be easily achieved.

When approximating $\sqrt{-\ln(x)}$ with 32-bit inputs based on polynomials of the form $p(x) = c_1 \times x + c_0$, the gradient of the steepest part of the curve is

in the order of 10^8, thus large multipliers would be required. To overcome this problem, we use scaling factors of multiples of two to reduce the magnitude of the gradient, essentially trading precision for range. This is appropriate since the larger the gradient, the less important precision becomes. The use of scaling factors provides the user the ability to control the precision for both c_1 and c_0. It is especially useful to control the size of the multiplier and adder. Hence for each segment four coefficients are stored: c_1 and its scaling factor, c_0 and its scaling factor.

It is also possible to divide the input interval into uniform or non-uniform intervals, and have uniform or non-uniform segments inside each interval. In this case, the most-significant bits are used to address the intervals and the least-significant bits are used to address the segments inside each interval. It can be seen that one can have any number of nested combinations of uniform and non-uniform segments. This hybrid combination of nested uniform and non-uniform segments provides a flexible way to choose the segment boundaries. Currently, this segmentation step is done by hand, which is slow and far from optimal. A possible approach to automate this step is discussed in Section 3.4.

3.3 Hardware Architecture

The architecture of our function evaluator shown in Figure 3.3 is based on polynomials of the form $p(x) = c_1 \times x + c_0$. The most-significant bits are used to select the interval, and the least-significant bits are passed through the segment address calculator which calculates the segment address within the interval. The design shown is developed for the common cases, and has been used in the examples of this paper. For other cases, one could divide the input bits into more than two parts and apply the segment address calculation depending on whether the parts use uniform or non-uniform segments.

The ROM outputs the four coefficients for the chosen interval and segment. c_1 is multiplied by the input x and c_s_1 is used to scale the output. The scaling circuit involves shifters, which increase or decrease the value by powers of two. This scaled multiplication value is added to the scaled c_0 coefficient to produce the final result.

For high throughput applications, the segment address calculator, the multiplier and the adder can be pipelined. For typical applications targeting FPGAs, the ROM would be small and could be implemented on-chip using distributed RAM or block RAM. Often the multiplier would be the part taking up a significant portion of the area. Therefore it is important to minimize the multiplier size by finding out the minimum bit width for the coefficient c_1. Also recent FPGAs, such as Xilinx Virtex-II devices, provide dedicated hardware resources for multiplication which can benefit the proposed architecture.

3.4 Placement of Segment Boundaries

Let f be a continuous function on $[a, b]$, and let an integer $m \geq 2$ specify the number of contiguous intervals into which $[a, b]$ has been partitioned: $a = u_0 \leq$

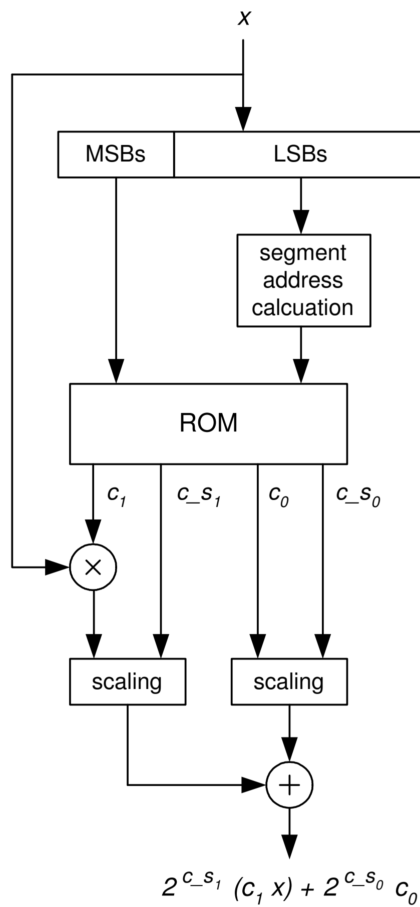

Figure 3.3: Our function evaluator architecture.

$u_1 \leq ... \leq u_m = b$. Let n_i and $d_i (i = 1, ..., m)$ be non-negative integers and let P_i denote the set of rational functions p_i whose numerators and denominators are polynomials of degrees less or equal to n_i and d_i, respectively. For $i = 1, ..., m$, define

$$h_i(u_{i-1}, u_i) = \min_{p_i \in P_i} \max_{u_{i-1} \leq x \leq u_i} |f(x) - p_i(x)|. \tag{3.1}$$

Let $\mu = \mu(u) = \max_{1 \leq i \leq m} h_i(u_{i-1}, u_i)$. Lawson states in his paper [52] that the segmented rational minimax approximation problem is that of minimizing μ over all partitions u of $[a, b]$. It can be shown that if the error norm is a non-decreasing function of the length of the interval of approximation, that the function to be approximated is continuous and that the goal is to minimize the maximum error norm on each interval, then a balanced error solution is optimal; the term "balanced error" means that the error norms on each interval are equal.

Pavlidis and Maika present an iterative scheme for segmentation in their paper [83] which results in a suboptimal balanced error solution. The scheme is based on an iteration of the form

$$u_m^{k+1} = u_m^k + c(e_{m+1}^k - e_m^k), \quad m = 1, ..., n-1. \tag{3.2}$$

Here u_m^k is the value of the m-th point and the k-th iteration, e_m^k is the error on $(u_{m-1}^k, u_m^k]$ and c is an appropriate small positive number. It can be shown that for sufficiently small c the scheme converges to a solution [83]. In this algorithm, the number of segments is fixed and the maximum error is worked out. However in many cases, it may be more useful to fix the accuracy desired and let the number of segments vary. Starting from a or b one could apply polynomial approximation in small increments, until the desired accuracy is reached. Then start a new segment from that point.

Once the segment boundaries have been found by using one of the two approaches above, the next step is to match the boundaries based on our addressing scheme as close to the suboptimum ones as possible. As discussed in Section 3.2, our addressing scheme is based on nested uniform and non-uniform segments. By carefully using these combinations of segments, it is possible to get a close approximation to the suboptimum segment boundaries. The aim is for the user to input constraints such as maximum error norm and to apply the segmentation automatically to produce look-up tables and the corresponding circuits such as the one shown in Figure 3.3. A possible approach of such an automated method is shown in Figure 3.4.

3.5 Evaluation and Results

Our function evaluator has been successfully implemented for the Gaussian noise generator presented in the next Chapter. Three functions are approximated: $\sqrt{-\ln(x)}$, $\cos(2\pi x)$ and $\sin(2\pi x)$ over $[0, 1]$. 32-bit inputs are used for $\sqrt{-\ln(x)}$ and 16-bit inputs are used for $\cos(2\pi x)$ and $\sin(2\pi x)$.

We first consider the function $\sqrt{-\ln(x)}$. As stated earlier, the greatest non-linearities of this function occur in the regions close to zero and one. To

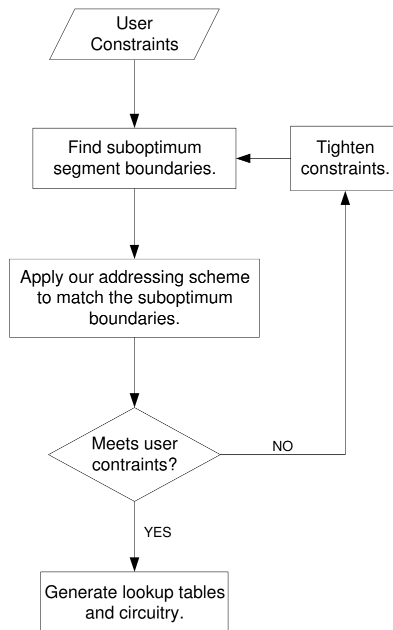

Figure 3.4: Steps for automating segmentation.

be consistent with the change in linearity, we use line segment locations to boundaries at locations 2^{n-32} for $0 < x \leq 0.5$, and $1 - 2^{-n}$ for $0.5 < x \leq 1$, where $0 \leq n < 32$. A total of 59 segments are used to approximate this function as shown in Figure 3.5. Since $\sqrt{-\ln(x)}$ approaches infinity for x values close to zero, the smallest x value is 2^{-32}, resulting in a maximum output value of around 4.7.

The maximum absolute error of this approximation is 0.020. However this is the case only if we have infinite precision for the coefficients, which is not practical. Multipliers take significant amount of resources on FPGAs, therefore the coefficients for the gradient should be as small as possible. Tests are carried out to find the optimum number of bits for the gradient coefficients that provides the least absolute error. Figure 3.6 shows how the maximum absolute error varies with the number of bits used for the gradient of $\sqrt{-\ln(x)}$. The figure indicates that 6 bits are sufficient to give a maximum absolute error of 0.031. Our requirement is that the approximation should differ from the true value by less than one unit in the last place (ulp) [49]; the least significant bit of the fraction of a number in its standard representation is defined to be the last place. This requirement is known as faithful rounding [24]. With this error, it is sufficient to give an output accuracy of 8 bits (three bits for integer and five for fraction). If uniform segments are used, small segment size would be needed in order to cope with the highly non-linear parts of the curve. In fact, one would require around 617 million segments to get the same maximum absolute error with uniform segments. This is a good example to demonstrate the effectiveness of our non-uniform approach. It is clear that our approach works well especially for functions with exponential behavior.

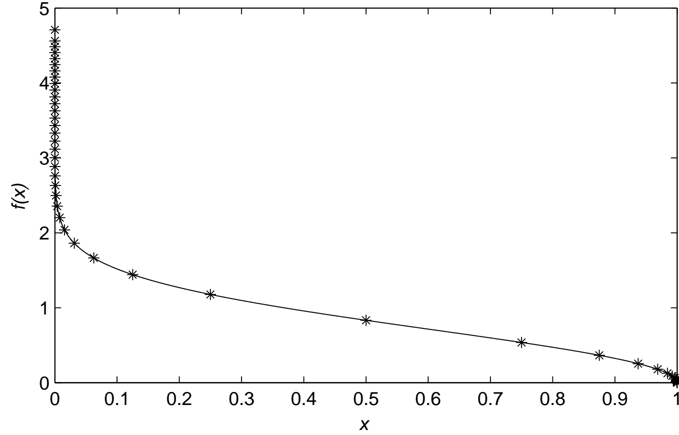

Figure 3.5: The segments used to approximate $\sqrt{-\ln(x)}$ with 32-bit inputs. The asterisks indicate the segment boundaries of the linear approximations.

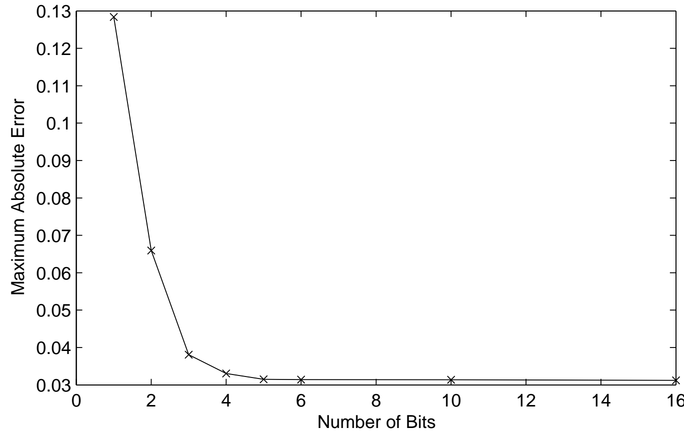

Figure 3.6: Variation of function approximation error with number of bits for the gradient of $\sqrt{-\ln(x)}$.

To evaluate the functions $\cos(2\pi x)$ and $\sin(2\pi x)$, due to the symmetry of the sine and cosine functions, only the input range $[0, 1/4]$ for $\cos(2\pi x)$ needs to be approximated. This technique is known as range reduction [118], [76]. The specific axis-partitioning technique for $\sqrt{-\ln(x)}$ is unsuitable for $\cos(2\pi x)$, since the non-linearities of the two functions are different. If the same technique was used, there would be many unnecessary segments near the beginning and end of the curve, and not enough segments in the middle regions. As before we consider both the local linearity of the curve, and the computational concerns with respect to choosing specific segment boundary locations, leading to the approximations shown in Figure 3.7. The curve is divided into four uniform intervals and within each interval, non-uniform segmentation is applied. We use 21 segments to approximate this function.

With finite precision on the coefficients, the maximum absolute error of this approximation is 0.0035, which is sufficient to give an output accuracy of 8

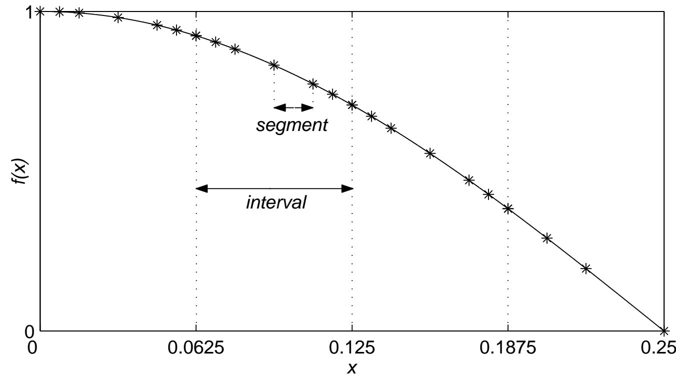

Figure 3.7: Approximation for $\cos(2\pi x)$ over $[0, 1/4]$. The asterisks indicate the segment boundaries of the linear approximations.

Table 3.1: Second column shows the comparison of the number of segments for non-uniform and uniform segmentation. Third column shows number of bits used for the coefficients to approximate the $\sqrt{-\ln(x)}$ and $\cos(2\pi\, x)$ functions.

function	non-uniform	uniform	c_1	c_s_1	c_0	c_s_0
$\sqrt{-\ln(x)}$	59	617 million	6	5	32	5
$\cos(2\pi\, x)$	21	27	8	4	16	4

bits (all eight bits for fraction). Using uniform segments the same error can be obtained with a slightly larger number of segments; this is because the curve does not have high non-linearities.

Table 3.1 shows a comparison of the number of segments for the two functions for non-uniform and uniform segmentation in order to achieve the same worst-case error. It also shows the number of bits used for each coefficient in the look-up tables. The look-up tables for the three functions $\sqrt{-\ln(x)}$, $\cos(2\pi x)$ and $\sin(2\pi x)$ have a size of just 3504 bits. With such small look-up table size, all the coefficients can be stored on-chip for fast access.

The function evaluators for the three functions are written using the Handel-C hardware compiler from Celoxica [16], and are mapped and tested on a Xilinx Virtex-II XC2V4000-6 device [115]. The design occupies 1864 slices, four block multipliers and two block RAMs, and takes up around 7% of the device. A fully pipelined version of our design operates at 133 MHz with a latency of 14 clock cycles, and the function evaluators are capable of 133 million operations per second; the completion time for each input is given by 14 / 133 million = 105 ns. The design has also been implemented on a low cost Xilinx Spartan-IIE XC2S300E-7, which occupies 70% of the chip and is capable of 62 million operations per second. Our hardware implementations have been compared with software implementations as shown in Table 3.2. The Virtex-based FPGA implementation is 158 times faster than the Athlon-based PC in terms of throughput, and 11 times faster in terms of completion time.

Table 3.2: Performance comparison: computation of $\sqrt{-\ln(x)}$, $\cos(2\pi x)$ and $\sin(2\pi x)$. All the PCs are equipped with 512MB DDR RAM. The XC2V4000-6 FPGA belongs to the Xilinx Virtex-II family, while the XC2S300E-7 belongs to the Xilinx Spartan-IIE family. The software implementations are written in C generating single precision floating point numbers, and is compiled with the GCC 3.3 compiler [37].

platform	clock speed (MHz)	latency (clock cycles)	area (slices)	throughput (operations / second)	completion time (ns)
XC2V4000-6 FPGA	133	14	1864	133 million	105
XC2S300E-7 FPGA	62	14	2129	62 million	226
AMD Athlon PC	1400	-	-	842460	1187
Intel Pentium 4 PC	2400	-	-	793021	1261

Comparing our function evaluator with other well known methods such as SBTM [102], [103], our approach has lower accuracy, but much smaller look-up table size. With further work on automatic segmentation, we hope to get as good accuracy as SBTM, but with smaller size and complexity.

3.6 Summary

We have presented a novel method for evaluating functions using polynomial approximations by employing non-uniform segments. The non-uniform segments deal with the non-linearities of functions which occur frequently. A simple cascade of AND and OR gates can be used to rapidly calculate the segment address for a given input. Scaling factors are used to deal with large polynomial coefficients, trading precision with range. Two functions developed for the generation of Gaussian noise are used as examples to illustrate and to evaluate our approach. Results show the advantages of using non-uniform segments over uniform ones. Current and future work includes the automation of the selection of boundaries and exploring the use of higher order polynomials for more accurate approximations. This would enable us to apply our approach to a wide range of functions and to obtain detailed comparison with other methods. We will also look at how our function generator can be used to speed up addition and subtraction functions in logarithmic number systems [20], [41], which are exponentially varying functions.

Chapter 4

Gaussian Noise Generation

4.1 Introduction

The purpose of this chapter is to present our hardware Gaussian noise generator, which is based on the Box-Muller method and central limit theorem. Section 4.2 briefly reviews the Box-Muller algorithm, and discusses how each of its steps can be handled in a hardware architecture. Section 4.3 describes technology-specific implementation of the hardware architecture. Section 4.4 discusses evaluation and results. Section 4.5 describes Wallace's method for generation Gaussian noise, and Section 4.6 offers summary and future work.

Numerical methods for Gaussian random number generation have a long history in mathematics and communications. As described in [48], [92] and the references cited therein, most methods involve initially generating samples of a uniform random variable and then applying the Box-Muller algorithm to obtain samples drawn from a unit-variance, zero-mean Gaussian PDF $f_X(x) = (1/\sqrt{2\pi})\, e^{-x^2/2}$. In the overwhelming majority of cases, this occurs in environments such as computer-based simulation where functions such as sine, cosine, and square roots are easily performed, and where there is sufficient precision so that finite-word length effects are negligible.

There has been far less attention focused on efficient hardware implementation of Gaussian noise generators, as the noise in real hardware systems is of course supplied by the environment and does not typically need to be generated internally. Recent advances in coding, however, have made the case for hardware-based simulation of channel codes much more compelling, and provide strong motivation to examine the Gaussian noise generation problem in the framework of limited word length, and limited computational and memory resources.

The principal contribution of our work is a hardware Gaussian noise generator that offers quality suitable for simulations involving very large numbers of noise samples. The noise generator is simple, occupying approximately 10% of the resources on a Xilinx Virtex-II XC2V4000-6 device [115], while producing over 133 million samples per second. In contrast with previous work, we focus specific attention on the accuracy of the noise samples in the high σ regions of the PDF, which are particularly important in achieving accurate results during

large simulations. The key novelties of our work include:

- a hardware architecture which involves the use of non-uniform piecewise linear approximations in computing trigonometric and logarithmic functions;
- exploration of hardware implementations of the proposed architecture targeting both advanced high-speed FPGAs and low-cost FPGAs;
- evaluation of the proposed approach using several different statistical tests, including the chi-square test and the Kolmogorov-Smirnov test, as well as through application to decoding of LDPC codes.

4.2 Architecture

This section provides an overview of the Box-Muller method and the associated four-stage hardware architecture. The implementation of this architecture in FPGA technology will be presented in Section 4.3.

The Box-Muller method [12] is conceptually straightforward. Given two realizations u_1 and u_2 of a uniform random variable over the interval [0,1), and a set of intermediate functions f, g_1 and g_2 such that

$$f(u_1) = \sqrt{-\ln(u_1)} \tag{4.1}$$
$$g_1(u_2) = \sqrt{2}\,\sin(2\pi\,u_2) \tag{4.2}$$
$$g_2(u_2) = \sqrt{2}\,\cos(2\pi\,u_2) \tag{4.3}$$

the products

$$x_1 = f(u_1)\,g_1(u_2) \tag{4.4}$$
$$x_2 = f(u_1)\,g_2(u_2) \tag{4.5}$$

then provide two samples of a Gaussian distribution $N(0, 1)$.

The above equations lead to an architecture that has four stages.

1. A shift register-based uniform random number generator,
2. implementation of the functions f, g_1, g_2 and the subsequent multiplications,
3. a sample accumulation step that exploits the central limit theorem to overcome quantization and approximation errors, and
4. a simple multiplexor-based circuit to support generation of one result per clock cycle.

A similar basic approach has been taken in other hardware Gaussian noise implementations [23]; what distinguishes our work is the detail of the functional implementation developed to deal with: (a) Gaussian noise with high σ values, and (b) evaluations using commonly-used statistical tests.

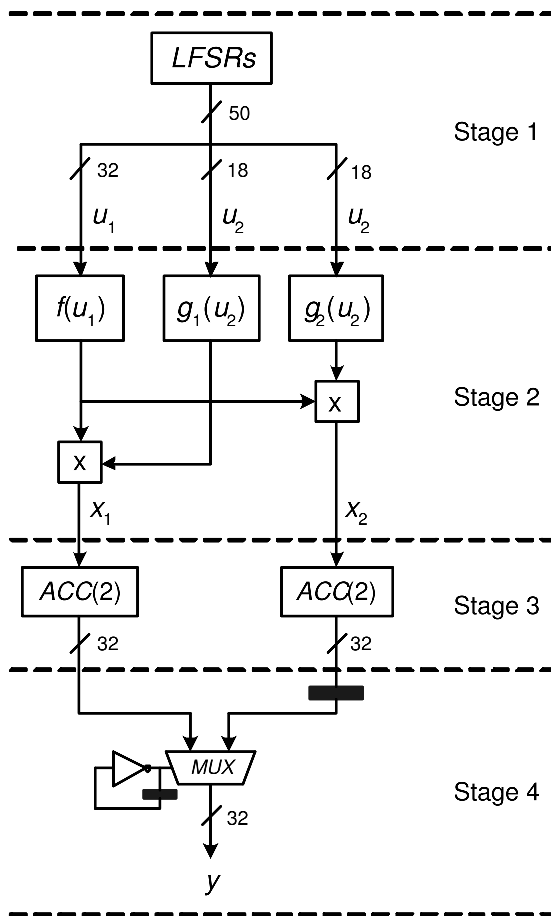

Figure 4.1: Gaussian noise generator architecture.

In the following, each of the four stages in our architecture is described in detail.

The first stage. This stage involves generation of the uniformly distributed realizations u_1 and u_2. The implementation of this stage is straightforward, and can be accomplished using well-known techniques based on Linear Feedback Shift Registers (LFSRs) [19]. To ensure maximum randomness, we use an independent shift register for each bit of u_1 and u_2. The resources needed are related to the periodicity desired in the shift registers. Since n-bit LFSRs with irreducible polynomials can produce random numbers with periodicity of $2^n - 1$, hardware required will be proportional to the number of bits of precision needed in u_1 and u_2.

The necessary precisions of u_1 and u_2 are related to the maximum values that the full system will produce. Since g_1 and g_2 are bounded by $[-\sqrt{2},\sqrt{2}]$, the maximum output is determined by f, which in turn takes on its largest values when u_1 is smallest. For example, when 16 bits are used for u_1, the maximum possible Gaussian sample has an absolute value of 4.7σ.

The second stage. This stage involves the most interesting challenges: efficient implementation of the functions f, g_1 and g_2. Direct computation of the logarithm and trigonometric functions leads to prohibitively long computation times. A look-up table would allow outputs to be obtained in only a few clock cycles, but this leads to prohibitively large memory requirements. For example, a look-up table for $f(u_1)$ with sufficient resolution for u_1 would require 2^{32} entries. Instead, the function evaluator from Chapter 3 is used. The best-fit straight line, in minimax sense, to each segment is found. A look-up table is used to store the gradient and the y-intercept for each line segment, and the functions can then be evaluated using a multiplier and an adder to calculate the linear approximation [71]. The key idea is to construct the piecewise linear approximation such that: (a) the segment lengths used in a given region depends on the local linearity, with more segments deployed for regions of higher non-linearity; and (b) the boundaries between segments are chosen such that the task of identifying which segment to use for a given input can be rapidly performed.

The f function is approximated using 59 segments as described earlier. The computation of g_1 and g_2 is carried out in a similar way. Given the symmetry of the sine and cosine functions, the axis can be considered in four regions related by symmetry, labelled 0 to 3 in Figure 4.2. The look-up table for g_1 and g_2 then only needs to hold coefficients corresponding to the input range [0,1/4]. We use the most significant 2 bits of u_2 to select a random region and the least significant 16 bits to select within the region. The outputs are generated using the symmetry by applying appropriate shifts and sign changes.

The computation of g_1 and g_2 is carried out in a similar way. Given the symmetry of the sine and cosine functions, the axis can be considered in four regions related by symmetry, labelled 0 to 3 in Figure 4.2. The look-up table for g_1 and g_2 then only needs to hold coefficients corresponding to the input range [0,1/4]. We use the most significant 2 bits of u_2 to select a random region and the least significant 16 bits to select within the region. The outputs are generated

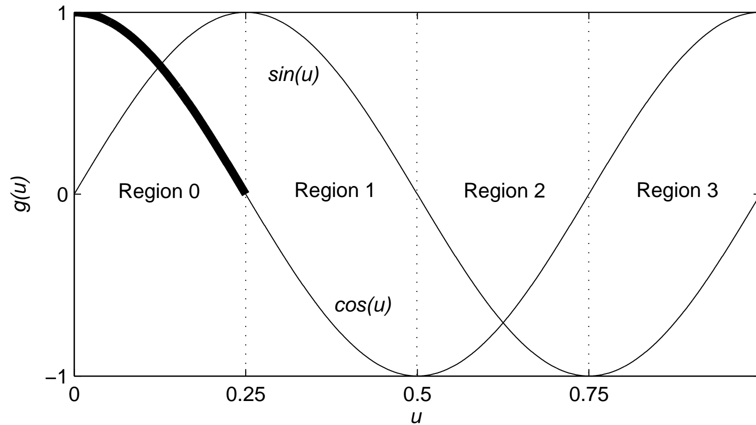

Figure 4.2: The g functions. Only the thick line is approximated; see Figure 3.7. The most significant 2 bits of u_2 are used to choose which of the four regions to use; the remaining bits select a location within Region 0.

using the symmetry by applying appropriate shifts and sign changes. Within a single region, the specific axis partitioning technique for f is unsuitable for g_1 and g_2 because the non-linearities of the functions are different. However, as before we consider both the local linearity of the curve and the computational concerns with respect to choosing specific segment boundary locations, leading to the approximations shown in Figure 3.7.

For each region we apply the same addressing technique as before: the most significant 2 bits of u_2 are used to choose which of the 4 regions to get the correct coefficients. Some regions need fewer approximations than others, if the function is relatively linear. It turns out that four regions are sufficient, although we could have used more. However, the more regions there are, the more complex it becomes to choose the appropriate region, which affects the complexity of the multipliers. Since the gradients of the g functions are not large, the scaling method for f is not used.

The third stage. This stage involves a sample accumulation step that exploits the central limit theorem to overcome quantization and approximation errors. As is well known, given a sequence of realizations of independent and identically distributed random variables $x_1, x_2, ..., x_n$ with unit variance and zero mean, the distribution of

$$\frac{x_1 + x_2 + ... + x_n}{\sqrt{n}}$$

tends to be normally distributed as $n \to \infty$. We find that $n = 2$ is sufficient, so we use an accumulator (the $ACC(2)$ component shown in Figure 1) that sums two successive inputs to produce an output every other cycle. The central limit theorem calls for a division by $\sqrt{2}$, which is potentially problematic in hardware. Fortunately, since computation of g_1 and g_2 involves a multiplication by $\sqrt{2}$ (Equations (4.2) and (4.3)), this multiplication is in effect cancelled by the subsequent division, so it can be dispensed with in both places in the

implementation. This optimization also alters the range of g as implemented to [-1,1].

The fourth stage. This stage involves a multiplexor-based circuit to select one of the two $ACC(2)$ component outputs in alternate clock cycles. The multiplexor is controlled by a circuit that toggles its output. This enables producing an output every clock cycle, rather than two outputs every other cycle.

Four further remarks about this architecture will be made. First, it is possible to speed up the output rate further by having multiple noise generators running in parallel, provided that the LFSRs are initialized with different random seeds. Second, the periodicity can be increased my using larger LFSRs and higher σ values can be obtained using more bits for u_1, both with very little increase in complexity.

Third, in addition to channel code evaluation, our noise generator can be used in various applications involving system-level characterization, such as digital watermarking [28], [111] and oscilloscope testing [113]. Fourth, for applications requiring a large dynamic range, floating-point arithmetic can be used for the components in our architecture.

4.3 Implementation

This section presents implementations of the four-stage architecture using FPGA technology.

We use 32 bits for u_1, allowing a maximum output of 6.7σ. Higher values of σ can be supported by increasing the number of bits for u_1; for instance 46 bits would yield a maximum output of 8σ. For u_2, 18 bits are found to be sufficient without loss of performance. This is because the trigonometric functions in g_1 and g_2 can be computed over [0,1/4] instead of [0,1], with symmetry used to derive the remainder of the [0,1] interval. The combination of 32 bits for u_1 and 18 bits for u_2 means that 50 shift registers are needed. We choose to target a period of about 10^{18} for the noise generator, which exceeds by several orders of magnitude even the most ambitious simulation size that can be contemplated with current hardware. Since 10^{18} is approximately 2^{60}, we use 60-bit LFSRs.

The 50 60-bit LFSRs can be implemented in configurable hardware using surprisingly few resources. Recent-generation reconfigurable hardware has a large amount of user-configurable elements. For instance the Xilinx Virtex-II XC2V4000-6 has 23040 user-configurable elements known as slices. The SRL16 primitive in Xilinx Virtex FPGAs enables a look-up table to be configured as a 16-bit shift register; so a 64-bit shift register using SRL16s instead of flipflops will use two slices instead of 32 [72]. Given that one 60-bit LFSR can be packed into two slices, so we just need 100 slices for the 50 LFSRs.

It could also be argued that application of the central limit theorem should be unnecessary if f, g_1 and g_2 are implemented with sufficient accuracy. However, there is hardware tradeoff involved in increasing the accuracy of these functions. We have found that application of the central limit theorem once (by summing two values as described above) results in a net reduction in complexity

Table 4.1: The number of bits used for each parameter of the f and g functions.

function	gradient	g-scale	y-intercept	y-scale
f	6	5	32	5
g	8	4	16	4

when the corresponding looser tolerances in the piecewise linear approximations are exploited.

Having a larger number of terms in the central limit theorem step would further simplify the linear approximations, but would slow the execution speed due to the need for accumulating more terms. For instance, when 17 approximations are used for f and 6 for g, eight values need to be summed in order to pass the statistical tests. When 59 approximations are used for f and 21 for g, without summing, the statistical tests fail after around 700 million samples. Therefore, we sum two samples to pass the tests.

Multipliers take significant amount of resources on FPGAs, therefore the coefficients for the gradient should be as small as possible. Tests are carried out to find the optimum number of bits for the gradient coefficients, that provide least error with small number of bits. Figure 3.6 in Section 3.5 shows how the worst-case error varies with the number of bits used for the gradient of the function f. The figure indicates that 6 bits are found to be sufficient.

Table 4.1 shows the number of bits used for each parameters in the look-up tables. Note that g_1 and g_2 share the same look-up table. 59 approximations are used for f, and 21 for g. The total look-up table has a size of 3504 bits for the function evaluator.

Several FPGA implementations have been developed, using the Handel-C hardware compiler from Celoxica [16]. We have mapped and tested the design onto a hardware platform with a Xilinx Virtex-II XC2V4000-6 device. This design occupies 2514 slices, eight block multipliers and two block RAMs, which takes up around 10% of the device. Stage two, which is the function evaluator, takes up 2137 slices or 85% of the slices used. A pipelined version of our design operates at 133 MHz, and hence our design produces 133 million Gaussian noise samples per second.

We have also implemented our design on a low-cost Xilinx Spartan-IIE XC2S300E-7 FPGA. This design runs at 62 MHz and has 2829 slices and 8 block RAMs, which requires over 90% of this device. This implementation can produce 133 million samples in around 2 seconds.

It is possible to increase the performance by exploiting parallelism. We have experimented with placing multiple instances of our noise generator in an FPGA, and find that there is a small reduction in clock speed probably due to the fan-out of the clock tree. For instance, a design with three instances of our noise generator takes up around 30% of the resources in an XC2V4000-6 device; it runs at 126 MHz, producing 378 million noise samples per second.

In the next section, the performance of the hardware designs presented

above will be compared with those of software implementations.

4.4 Evaluation and Results

This section describes the statistical tests that we use to analyze the properties of the generated Gaussian noise.

A software program in C has been developed to fully emulate the proposed hardware design with finite precision. The random Gaussian variables generated from this program are loaded into the MATLAB package [69] for analysis. There are numerous tests that can be applied to evaluate the quality of generated noise. Our tests to measure the quality of the generated noise include both goodness-of-fit for density and test for independence. We use two well-known goodness-of-fit tests to check the normality of the random variables: the chi-square (χ^2) test and the Kolmogorov-Smirnov (K-S) test [48], [92].

The χ^2 test involves quantizing the x axis into k bins, determining the actual and expected number of samples appearing in each bin, and using the results to derive a single number that serves as an overall quality metric. Let n be the number of observations, p_i be the probability that each observation fall into the category i and Y_i be the number of observations that actually do fall into category i. The χ^2 statistic is

$$\chi^2 = \sum_{i=1}^{k} \frac{(Y_i - np_i)^2}{np_i} \tag{4.6}$$

This test, which is essentially a comparison between an experimentally determined histogram and the ideal PDF, is sensitive not only to the quality of the noise generator itself, but also to the number and size of the k bins used on the x axis. For example, a noise generator that models the true PDF very accurately for low absolute values of x but fails for large x could yield a good χ^2 result if the examined regions are too closely centered around the origin. It is precisely for these high $|x|$ regions where a noise generator is critically important, and most likely to be flawed.

Consider a simulation involving generation of 10^{12} noise samples, conducted with the goal of exploring performance for a channel decoder in the range of BERs from 10^{-9} to 10^{-10}. In samples drawn from a true unit-variance Gaussian PDF, we would expect that approximately half a million samples from the set of 10^{12} would have absolute value greater than $x = 5$. These high σ noise values are precisely the ones likely to cause problems in decoding, so a hardware implementation that fails to faithfully produce them appropriately risks creating incorrect and deceptively optimistic results in simulation. To counter this, we extended the tests to specifically examine the expected versus actual production of high σ values.

While the χ^2 test deals with quantized aspects of a design, the K-S test deals with continuous properties. Given a hypothesized distribution function without discontinuities, the K-S test compares a CDF to the empirical distribution function of the samples. It is defined as the maximum value of the absolute

difference D between two cumulative distributions. Thus, for comparing a data set $Y(x)$ to a known CDF $F(x)$, the K-S statistic is

$$D = \max_{-\infty<x<\infty} | Y(x) - F(x) | \quad (4.7)$$

The χ^2 and K-S statistics are used to compute the p-values [21] for our outputs. The p-value is a probability. A sample set with a small p-value means that it is less likely to follow the target distribution. The general convention is to reject the null hypothesis – that the samples are normally distributed – if the p-value is less than 0.05.

Figures 4.3, 4.4 and 4.5 illustrate the effect on the PDF of different implementation choices. Figure 4.3 shows the PDF obtained when 17 and 9 linear approximations are used for f and $g1$ respectively. The figure (as well as the others in this section) is based on a simulation of four million Gaussian random variables. There are distinct error regions visible in the PDF, which occur when there are large errors in the multiplication of f and g_1. These distinct errors cause the various statistical tests to fail. Increasing the number of linear approximations to 59 and 21 respectively leads to the PDF shown in Figure 4.4. It is clear that the error regions have decreased significantly. However, this still fails the statistical tests when the sample size is sufficiently large. When the further enhancement of summing two successive samples as discussed earlier is added, the PDF of Figure 4.5 results.

This implementation passes the statistical tests even with extremely large numbers of samples. For the χ^2 test, we use 700 bins for the x axis over the range [-7,7]. The p-values calculated are around 0.5, which are well above 0.05, indicating that the generated noise samples are indeed normally distributed. The p-values for the K-S test are also well above 0.05. In order to explore the possibility of temporal statistical dependencies between the Gaussian variables [96], we generate scatter plots showing pairs y_i and y_{i+1}. An example based on 10000 Gaussian variables is shown in Figure 4.6, which displays no obvious correlations.

We have used our noise generator in LDPC decoding experiments. To obtain a benchmark, we performed LDPC decoding using a full precision (64-bit floating point representation) software implementation of belief propagation in which the noise samples are also of full precision. We then performed decoding using the LDPC algorithm but with noise samples created using the design presented in this paper. Over many simulations, we have found no distinguishable difference in code performance, even in the high E_b/N_0 regions where the error floor in BER is as low as 10^{-9}.

Our hardware implementations, described in Section 4.3, have been compared to several software implementations based on the polar method, which is the fastest method to generate Gaussian noise for instruction processors. The results are shown in Table 4.2. It can be seen that our hardware designs are faster than software implementations by 20–400 times, depending on the device used and the resource utilization.

Figure 4.7 shows how the number of noise generator instances affects the output rate. While ideally the output rate would scale linearly with the number of noise generator instances, in practice the output rate grows slower than

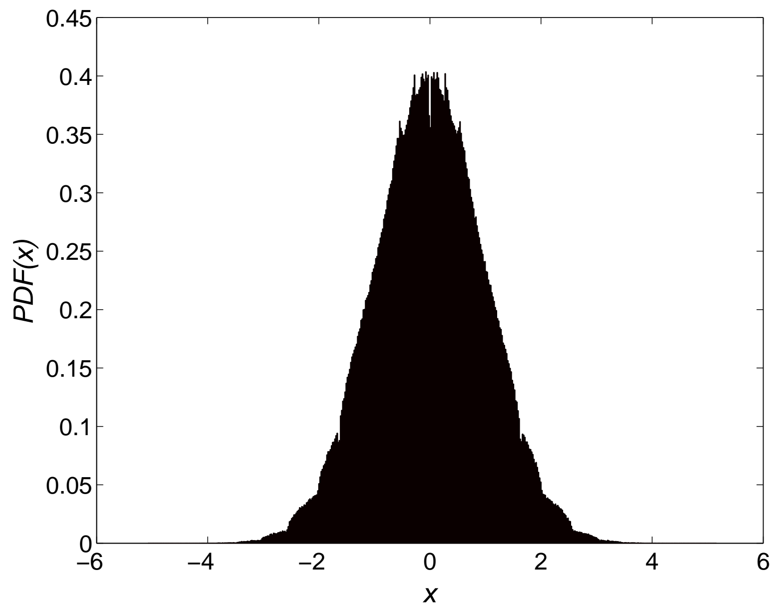

Figure 4.3: PDF of the generated noise with 17 approximations for f, 6 for g.

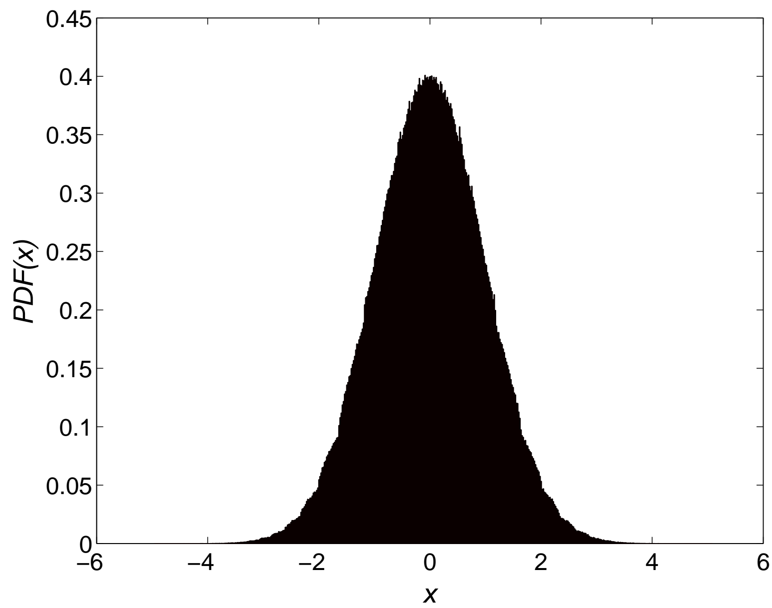

Figure 4.4: PDF of the generated noise with 59 approximations for f, 21 for g.

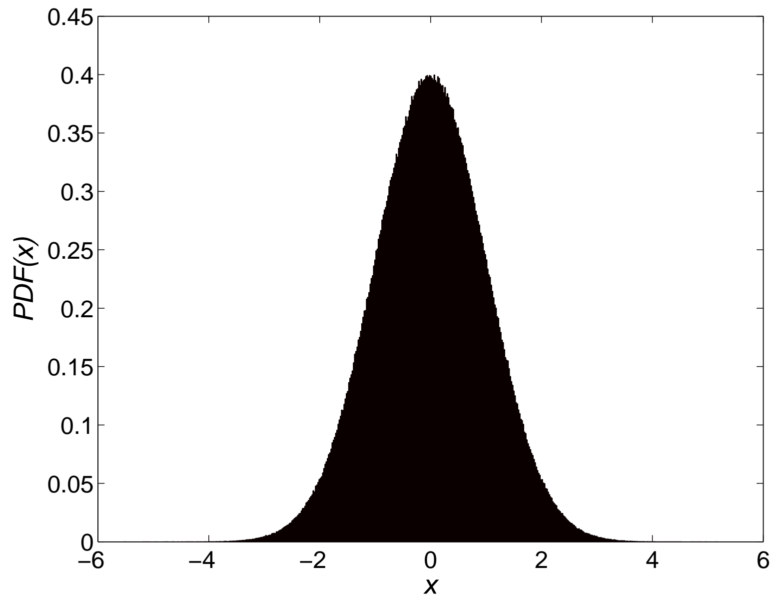

Figure 4.5: PDF of the generated noise with 59 approximations for f, 21 for g with two accumulated samples.

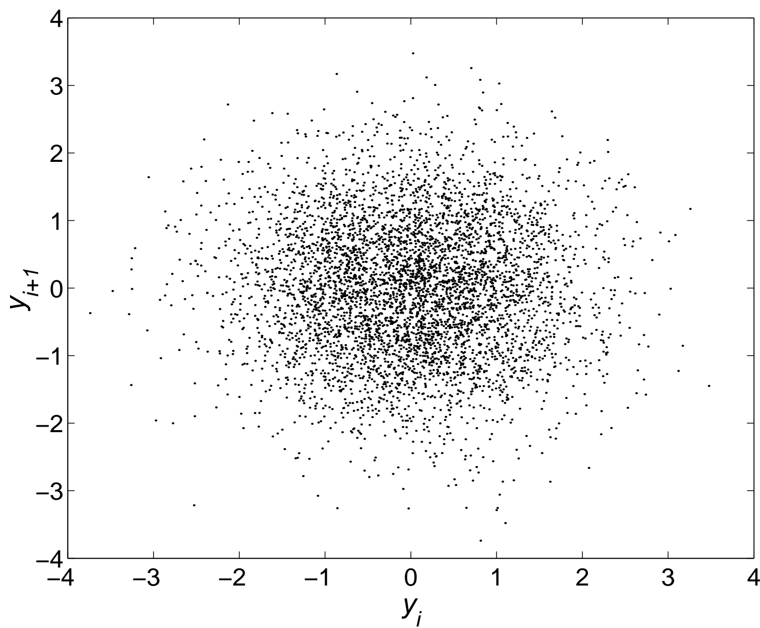

Figure 4.6: Scatter plot of two successive accumulative noise samples for a population of 10000.

Table 4.2: Performance comparison: time for producing 1 billion Gaussian noise samples. All PCs are equipped with 512MB DDR RAM. The XC2V4000-6 FPGA belongs to the Xilinx Virtex-II family, while the XC2S300E-7 belongs to the Xilinx Spartan-IIE family. The software implementations are written in C generating single precision floating point numbers, and are compiled with the GCC 3.3 compiler [37].

platform	**time (sec)**
XC2V4000-6 FPGA, 105MHz, 96% usage	1
XC2V4000-6 FPGA, 126MHz, 32% usage	2.6
XC2V4000-6 FPGA, 133MHz, 10% usage	7.5
XC2S300E-7 FPGA, 62MHz, 90% usage	16
AMD Athlon XP PC, 2.13GHz	300
AMD Athlon PC, 1.4GHz	338
Intel Pentium 4 PC, 2.0GHz	421

expected, because the clock speed of the design deteriorates as the number of noise generators increases. This deterioration is probably due to the increase in clock fan-out and loading.

4.5 Wallace's Method

Recently, Wallace [117] proposed a fast algorithm for generating normally distributed numbers which generates the target distributions directly from their maximal-entropy properties. This algorithm does not require a stream of uniform pseudo-random numbers or the evaluation of transcendental functions such as log, sqrt, sin or cos (needed by the Box-Muller method). This method exploits that if X is a K-vector of normally distributed numbers, and A is a $K \times K$ orthogonal matrix, then $Y = AX$ is another K-vector of normally distributed numbers. Thus, given a pool of KL normally distributed numbers, we can generated another pool of KL normally distributed numbers by performing L matrix-vector multiplications. The resulting pool of normally distributed numbers can be used to generate a new pool. This process of generating a new pool is known as a pass.

Although Wallace's method is simple and elegant there are some optimizations steps [13], [117] that need to be carried out to produce good quality noise samples. These involve operations which are not amendable for hardware implementation such as division. In addition, the variance of the noise samples tends to drift as more passed are performed [99], therefore the pool needs to be refreshed with new noise samples (from Box-Muller) every certain number of passes.

We have implemented a simplified version of Wallace in C and evaluated the noise samples with the χ^2 test. Our tests indicate that the pool needs

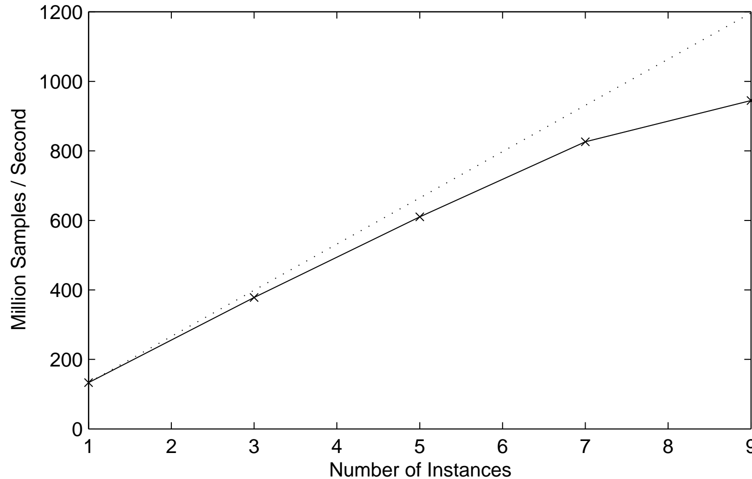

Figure 4.7: Variation of output rate against the number of noise generator instances. The dotted line shows the linear relationship between the output rate and the number of instances, if the clock speed does not deteriorate with the increasing number of instances.

to be refreshed less frequently when more optimization steps are performed. Currently, we are looking at modifying these steps so that they are more suitable for hardware implementation and ways of integrating Wallace with our Box-Muller implementation. The integration of the two methods is logical, since the pool in Wallace needs to be refreshed with fresh samples from Box-Muller.

4.6 Summary

We have presented a hardware-based Gaussian noise generator designed to facilitate channel code simulations implemented in hardware which involves very large numbers of samples. A key aspect of the design is the use of non-uniform piecewise linear approximations in computing trigonometric and logarithmic functions, with the boundaries between each approximation chosen carefully to enable rapid computation of coefficients from the inputs.

Our noise generator design is simple, occupying approximately 10% of a Xilinx Virtex-II XC2V4000-6 FPGA and 90% of a Xilinx Spartan-IIE XC2S300E-7, and can produce 133 million samples per second. Statistical tests as well as application in LDPC decoding have been used to confirm the quality of the noise samples. Ongoing and future work includes the implementation of hardware noise generators for different channels such as Rayleigh, Ricean and Nakagami-m [123] channels, and investigations how Wallace's method could be used in conjunction with our noise generator.

Chapter 5

LDPC Encoding

5.1 Introduction

The purpose of this chapter is to present our hardware implementation of an efficient LDPC encoder based on the RU method described in Section 2.4.2. Section 5.2 presents and overview of our approach. Section 5.3 shows our results after preprocessing the H matrix. Section 5.4 describes our hardware architecture. Section 5.5 offers our implementation results, and Section 5.6 summarizes our work.

5.2 Overview

We describe an efficient hardware encoder for irregular LDPC codes. Although LDPC codes achieve better performance and have low decoding complexity compared to Turbo codes, one of the major drawback of LDPC codes lies in their high encoding complexity. Whereas Turbo codes can be encoded in linear time, a straightforward implementation for a LDPC code has complexity quadratic in the block length.

It is suggested in [64] and [95] that using an approximate upper triangular parity check matrix to construct LDPC code can reduce the encoding complexity significantly without performance degradation. Our hardware implementation is based on the RU algorithm [95], which admits linear time encoding through optimizing irregular codes. All published hardware LDPC encoders in literature [44], [70], [125] employ the straightforward encoding method (Section 2.4.2). The only hardware LDPC encoder known that uses the RU method is the one from Flarion [32], but hardly any details are known due to it being a commercial product.

Our approach consists of two steps: preprocessing and hardware encoding. First, the original parity check matrix H is preprocessed with the RU algorithm to generate the appropriate look-up tables consisting of the six matrices needed by the hardware encoder. These matrices are generated from the RU algorithm and contain information on how the input message blocks are encoded to generate codewords. This preprocessing step is implemented in MATLAB. The hardware encoder itself is implemented on a FPGA and uses the look-up tables

generated from the preprocessing step to encode the message blocks.

5.3 Preprocessing

A software program is written in MATLAB to preprocess a 2499×5000 irregular H matrix. Row and column permutations are carried out to bring the parity-matrix into approximate lower triangular form with a gap g of 2. The resulting dimensions and the number of edges of the matrices generated from the RU algorithm are shown in Table 5.1.

Table 5.1: Dimensions and number of edges for the matrices A, B, T, C, D and E generated from a 2499×5000 irregular H matrix.

matrix	**dimension**	**edges**
A	2497×2501	14701
B	2497×2	40
T	2497×2497	6046
C	2×2501	11
D	2×2	0
E	2×2497	6046

This preprocessing step takes approximately 2 hours on a Pentium 4 2GHz PC. It needs to be performed only once for a given H matrix to generate the six matrices. Matrix A has 2501 columns, indicating that a message block (the systematic part s) needs to be 2501 bits.

5.4 Hardware Encoder Architecture

From Tables 2.1 and 2.2 in Section 2.4.2, it can be seen that the RU method computes two parity parts p_1 and p_2. A message block s is simply concatenated with its two parity parts to generate a codeword $x = (s, p_1, p_2)$. There are three types of operations required for the computation of the two parity parts, which are matrix by vector multiplications, back-substitutions and vector additions. Since we are dealing with a binary system, all operations are performed in $GF(2)$. Vector addition can be simply achieved by performing XOR operations on the corresponding elements of the two vectors. However, matrix-vector multiplication and back-substitution are non-trivial. We present our hardware architecture for the two operations.

5.4.1 Matrix-vector multiplication

This involves the computation of $X \times Y = Z$, where the matrices X and Y are known and Z is what were are trying to compute. We shall illustrate our

approach with an example. Assume we are multiplying a 5×6 matrix X by a 6×1 matrix Y (a vector since it has a single column) to obtain a resulting matrix Z. The matrix X is known from the preprocessing step and is sparse. It would be inefficient to store this matrix directly in a memory, since most of the locations will be zeros. Instead, the location of the edges (ones) of each row is stored, with an extra bit indicating the end of a row. For example, if

$$X = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{5.1}$$

it would be stored in memory as shown in Table 5.2. Memory #6 is a special

Table 5.2: Matrix X stored in memory.

address	**edge**	**end row**
0	3	0
1	5	1
2	1	1
3	2	0
4	4	0
5	6	1
6	0	1
7	3	0
8	4	1

case, indicating that the fourth row of matrix X has no edges. The location of the edges of a row in X are used as bit selectors for the vector Y. This bit selecting process has the same effect as performing AND operations with the bits of a row in X and the bits in vector Y. XOR is performed on the selected bits to calculate the resulting bits for Z. This operation is performed for each row of X starting from the first one. Figure 5.1 shows our matrix by vector multiplication circuit.

The Z index calculator works out the location of the Z matrix to be written. The index is simply incremented every time there an end of a row. It can be seen that the number of clock cycles required to compute Z is directly proportional on the number of edges in X.

5.4.2 Back-substitution

Consider the equation $X \times Z = Y$, where the matrices X and Y are known and Z is the matrix we want to compute. X is a lower triangular matrix, Y and Z

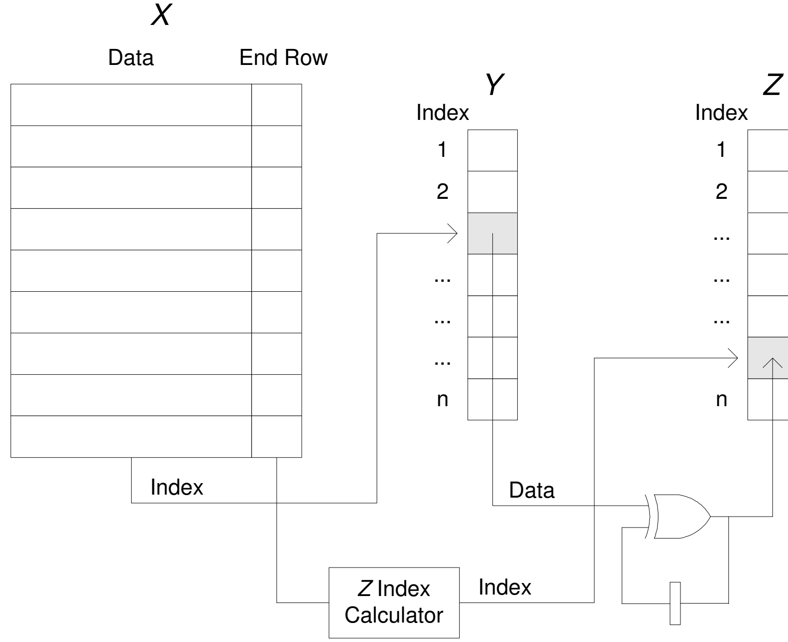

Figure 5.1: Circuit for matrix-vector multiplication.

are vectors as illustrated below.

$$\overbrace{\begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ x_{(2,1)} & 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ x_{(3,1)} & x_{(3,2)} & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ x_{(4,1)} & x_{(4,2)} & x_{(4,3)} & 1 & 0 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{(n,1)} & x_{(n,2)} & \cdots & \cdots & \cdots & \cdots & x_{(n,n-1)} & 1 \end{bmatrix}}^{X} \times \overbrace{\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ \vdots \\ z_n \end{bmatrix}}^{Z} = \overbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_n \end{bmatrix}}^{Y} \quad (5.2)$$

One way to approach this problem is to take the inverse of X and compute $Z = YX^{-1}$. However, matrix inversion is a complex procedure and requires a significant amount of processing time. A better way is to use back-substitution exploiting the fact that X is lower triangular. The elements of the vector Z can be computed with the following set of equations.

$$\begin{aligned} z_1 &= y_1 \\ z_2 &= y_2 \oplus x_{(2,1)} z_1 \\ z_3 &= y_3 \oplus x_{(3,1)} z_1 \oplus x_{(3,2)} z_2 \\ &\vdots \\ z_n &= y_n \oplus x_{(n,1)} z_1 \oplus x_{(n,2)} z_2 \oplus \cdots \oplus x_{(n,n-1)} z_{n-1} \end{aligned}$$

This can be generalized as:

$$z_i = y_i \oplus \bigoplus_{j=1}^{i-1} x_{(i,j)} z_j, \quad 1 \le i \le n \quad (5.3)$$

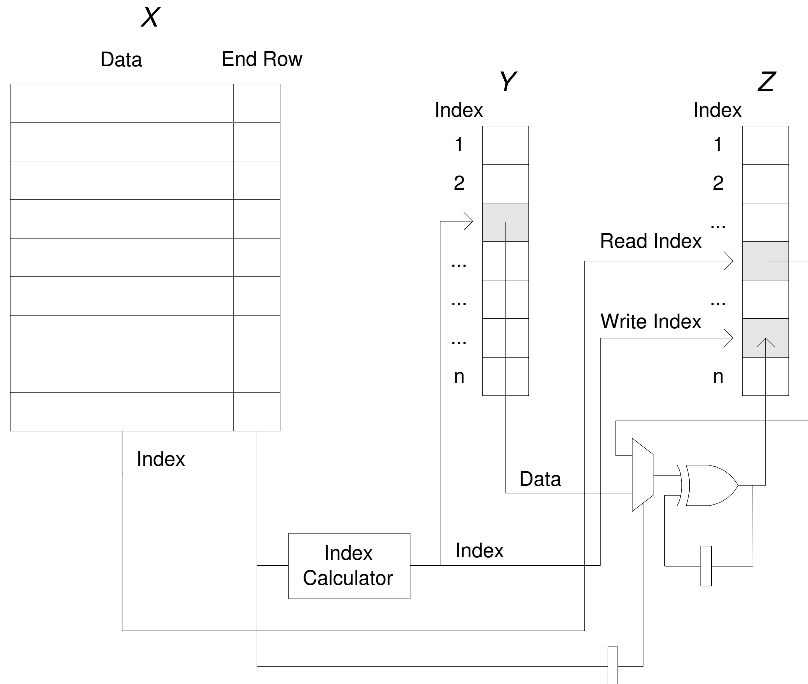

Figure 5.2: Circuit for back-substitution.

Just like the matrix-vector multiplication, to an element in Z, we need elements from X and Y. However, we also require all the previous elements of Z that have been computed previously. Therefore, the circuit for back-substitution is very similar to the one in Figure 5.1 with slight modifications as shown in Figure 5.2. The index calculator works out the memory location of Y to be read and Z to be written. As it can be seen from (5.3), these two address are identical. As in the matrix-vector multiplication case, the clock cycles for the computation is proportional to the number of edges in X.

5.5 Implementation and Results

We have implemented the encoder using the Verilog hardware description language [112]. The encoder is non-pipelined for keep it simple minimizing the resource usage and contains circuits for matrix by vector multiplication, back-substitution and vector addition. The codewords generated from our hardware encoder have been verified against our MATLAB model for correctness. The design has been synthesized on a Xilinx Virtex-II XC2V4000-6 chip. It takes up just 3% of the device and is capable of running at 133MHz.

Let $e(A)$ denote the number of edges for the matrix A. Matrix-vector multiplication with A and E needs to be performed for the computation of p_1 and back-substitution with T needs to be performed for the computation of both p_1 and p_2. So we have one Matrix-vector multiplication with A, one for E, and two back-substitutions for T. We have decided to implement one unit each

for matrix-vector multiplication and back-substitution to minimize the area on the chip. The use of multiples of these units are planned for future work to achieve higher throughput. With our current architecture, the number of cycles required for the generation of one codeword is proportional to $e(A)$, $e(E)$ and $e(T)$ (since $e(B)$, $e(C)$ and $e(D)$ are very small). Therefore, the number of clock cycles to generate a codeword is

$$e(A) + e(E) + 2e(T) = 14701 + 6046 + 2(6046) = 32839. \tag{5.4}$$

With a clock speed of 133 MHz we can generate $(133 \times 10^6)/32839 = 4050$ codewords per second. Since each message block is 2501 bits, we are capable of a throughput of $2501 \times 4050 = 10$MBps.

5.6 Summary

We have described a hardware implementation of an efficient LDPC encoder based on the RU method. The design is simple and takes just 3% of resources on a Xilinx Virtex-II XC2V4000-6 device. It is capable of running at 133MHz resulting in a throughput of 10MBps. The codewords generated from our encoder have been validated against our software simulation model for correctness. The encoder block will be placed in front of the noise generator in our LDPC simulation framework. Future work on the hardware encoder include optimizing the design to achieve higher throughput and exploring the use of run-time reconfiguration of FPGAs to support different modes of operation (e.g. support different H matrices at run-time).

Chapter 6

Conclusion

6.1 Summary

Three main topics have been presented in this report: function evaluation, Gaussian noise generation, and LDPC encoding.

The function evaluator uses first order polynomials to approximate elementary or special purpose functions. The novelty of our approach is the use of non-uniform segments, in which the segment sizes can be adjusted to conform to the non-linearities for the function the be approximated. A simple cascade of AND and OR gates can be used to rapidly calculate the segment addresses for a given input. Scaling factors are used to deal with large polynomial coefficients, trading precision with range. Two functions are used for case studies: $\sqrt{-\ln(x)}$ and $\cos(2\pi x)$. Results indicate the significant improvements by using our non-uniform segment approach over the traditional uniform approach, especially for functions with high non-linearities. The approximations of the two functions are successfully used in the Gaussian noise generator. Current and future work includes the automation of the selection of the boundaries and exploring the use of higher order polynomials for more accurate approximations. We will also look at how our function evaluator can be used to speed up addition and subtraction functions in logarithmic number systems, which are highly non-linear functions.

The hardware Gaussian noise generator is designed to facilitate channel code simulations, which involve very large numbers of samples. Our non-uniform segment function evaluator is used to compute the trigonometric and logarithmic functions needed for the generation of the noise samples. The design occupies approximately 10% of a Xilinx Virtex-II XC2V4000-6 chip and 90% of a Xilinx Spartan-IIE XC2S300E-7, and can produce 133 million samples per second. Various statistical tests as well as application to LDPC decoding have been used to confirm the quality of the noise samples. Gaussian probability distribution functions model many natural phenomena. Therefore, the application of our noise generator is not limited to channel code evaluation. It has a wide range of other applications including watermarking, oscilloscope testing, simulation of economic systems and molecular dynamics simulations. We plan to improve our noise generator to support different types of channels such as Rayleigh, Ricean

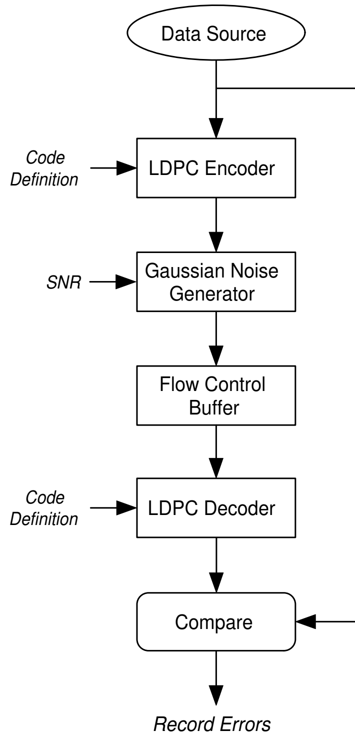

Figure 6.1: LDPC simulation framework.

and Nakagami-m. Also, we will investigate how the recent technique developed by Wallace can be used to speed up our design even further.

Finally, an efficient hardware LDPC encoder has been developed. It is based on the RU encoding method and takes up 3% of a Xilinx Virtex-II XC2V4000-6 device. It runs at 133MHz and is capable of a throughput of 10Mbps. The codewords generated from our encoder have been verified against our software model for correctness. Best to our knowledge this is the very first hardware implementation of such an encoder. Optimizations are planned to support a higher throughput.

The Gaussian noise generator and the LDPC encoder that we have developed will be integrated with our hardware LDPC simulation framework shown in Figure 6.1.

6.2 Research Plan

Jul 2003 - Aug 2003: More work on the function evaluator. The plan is to automate the segmentation based on the linearity of the function. Detailed error analysis will be carried out and the effects of second order polynomial approximation will be explored. Apply our functions evaluator on various functions and investigate how it can be used to speed up addition and subtraction functions in logarithmic number systems. Also, explore the use of Wallace's

method to generate Gaussian random variables.

Aug 2003 - Sep 2003: Improve the LDPC encoder. Revise the current architecture to a pipelined approach to achieve higher throughput. Explore the use of run-time reconfiguration to support different encoding modes (e.g. support different H matrices at run-time).

Oct 2003 - Apr 2004: Work on other parts of the LDPC simulation framework, such as some aspects of the decoder. The decoder is the most challenging part of the system, thus a lot of room for further research. It can be designed in a variety of different ways, and there is a potential of a adaptive LDPC decoder using run-time reconfiguration. This could be similar to the adaptive Viterbi decoders in [17] and [108].

Apr 2004 - Jul 2004: For our function evaluator, create a facility that would generate the appropriate hardware circuit and look-up tables for a given function and worst-case error. Also, look at how our noise generator can be modified to simulate different types of communication channels.

Aug 2004 - Oct 2004: Write up the PhD thesis.

The proposed thesis chapters are listed below.

Chapter 1 : Introduction
Chapter 2 : Background
Chapter 3 : Function Evaluation
Chapter 4 : Channel Simulation
Chapter 5 : LDPC Encoding
Chapter 6 : LDPC Decoding
Chapter 7 : Conclusion

Papers Published

D. Lee, W. Luk and P.Y.K. Cheung, "Incremental programming for reconfigurable engines", *In Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, 2002.

D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, "A hardware Gaussian noise generator for channel code evaluation", *In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2003.

D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, "Hardware function evaluation using non-linear segments", *In Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, Springer Verlag, 2003.

D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, "Hierarchical segmentation scheme for function evaluation", *Submitted to the IEEE International Conference on Field-Programmable Technology (FPT)*, 2003.

Acknowledgements

I would like to thank my supervisor Dr. Wayne Luk for his advice and direction on both academic and non-academic issues. I would also like to thank Prof. John Villasenor from UCLA and Prof. Peter Y.K. Cheung from EEE department for their help on LDPC codes and noise generation. Many thanks to my colleagues Jun Jiang, Altaf Abdul Gaffar and Shay Ping Seng for their assistance. The support of Celoxica Limited, Xilinx Inc., the U.K. Engineering and Physical Sciences Research Council (Grant number GR/N 66599 and GR/R 31409), and the U.S. Office of Naval Research is gratefully acknowledged.

Bibliography

[1] J.H. Ahrens and U. Dieter, "Computer methods for sampling from the exponential and normal distributions", *Comm. of the ACM*, vol. 15, no. 10, pp. 873–882, 1972.

[2] J.H. Ahrens and U. Dieter, "Efficient table-free sampling methods for the exponential, Cauchy, and normal distributions", *Comm. of the ACM*, vol. 31, no. 11, pp. 1330–1337, 1988.

[3] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers", *Proc. ACM/SIGDA Int. Symp. on Field Prog. Gate Arrays (FPGA)*, pp. 191–200, 1998.

[4] N.C. Beaulieu and C.C. Tan, "An FFT method for generating bandlimited Gaussian noise variates", *Proc. IEEE Global Comm. Conf.*, pp. 684–688, 1997.

[5] D.R. Barr and N.L. Sezak, "A comparison of multivariate normal generators", *Comm. of the ACM*, vol. 15, no. 12, pp. 1048–1049, 1972.

[6] A.R. Bergstrom, *Econometric Theory*, vol. 13, no. 467, 1997.

[7] C. Berrou, A. Glavieuxand and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes", *Proc. IEEE Conf. on Communications*, pp. 1064–1070, 1993.

[8] T. Bhatt, K. Narayanan and N. Kehtarnavaz, "Fixed-point DSP implementation of low-density parity check codes", *Proc. of the 9th IEEE DSP Workshop*, 2000.

[9] A.J. Blanksby and C.J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder", *IEEE J. of Solid-State Circ.*, vol. 37, no. 3, pp. 404-412, 2002.

[10] M. Bossert, *Channel Coding for Telecommunications*, John Wiley & Sons, 1999.

[11] E. Boutillon, J.L. Danger and A. Gazel, "Design of high speed AWGN communication channel emulator", *Analog Integrated Circuits and Signal Processing*, vol. 34, no. 2, pp. 133–142, Kluwer Press, 2003.

[12] G.E.P. Box and M.E. Muller, "A note on the generation of random normal deviates", *Ann. Math. Statist.*, vol. 29, pp. 610–611, 1958.

[13] R.P. Brent, "A fast vectorised implementation of Wallace's normal random number generator", *ANU Computer Science Technical Reports*, no. TR-CS-97-07, The Australian National University, 1997.

[14] J. Cao, B.W.Y. We and J. Cheng, " High-performance architectures for elementary function generation", *Proc. 15th IEEE Symp. on Comput. Arith.*, 2001.

[15] J. Cavallaro and M. Vaya, "VITURBO: A reconfigurable architecture for Viterbi and Turbo decoding", *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, 2003.

[16] Celoxica Limited, *Handel-C Language Reference Manual*, ver. 3.1, document no. RM-1003-3.0, 2002.

[17] K. Chadha and, J. Cavallaro, "A reconfigurable Viterbi decoder architecture", *Proc. Asilomar Conf. on Sig. Syst. and Comput.*, pp. 66-71, 2001.

[18] P.L. Chu, "Fast Gaussian noise generator", *IEEE Trans. on Acoustics, Speech and Signal Processing*, vol. 37, no. 10, 1989.

[19] P.P. Chu and R.E. Jones, "Design techniques of FPGA based random number generator", *Proc. Military and Aerospace Applications of Prog. Devices and Tech. Conf.*, 1999.

[20] J.N. Coleman, E. Chester, C.I. Softley and J. Kadlec, "Arithmetic on the European logarithmic microprocessor", *IEEE Trans. on Comput. Special Edition on Comput. Arith.*, vol. 49, no. 7, pp. 702–715, 2000.

[21] W.J. Conover, *Practical Nonparametric Statistics*, John Wiley and Sons, 1971.

[22] D.J. Costello, J. Hagenauer, H. Imai and S.B. Wicker, "Applications of error control and coding", *IEEE Trans. on Inform. Theory*, vol. 44, no. 6, pp. 2531–2560, 1998.

[23] J.L. Danger, A. Ghazel, E. Boutillon and H. Laamari, "Efficient FPGA implementation of Gaussian noise generator for communication channel emulation", *Proc. 7th IEEE Int. Conf. on Elect., Circ. and Syst.*, 2000.

[24] D. Das Sarma and D.W. Matula, "Faithful bipartite rom reciprocal tables", *Proc. 12th IEEE Symp. on Computer Arithmetic*, pp. 17–28, 1995.

[25] F. de Dinechin and A. Tisserand, "Some improvements on multipartite table methods", *Proc. IEEE Symp. on Comput. Arith.*, pp. 128–135, 2001.

[26] J. Detrey and F. de Dinechin, "Multipartite tables in JBits for the evaluation of functions on FPGAs", *Proc. IEEE Int. Parallel and Distributed Processing Symp.*, 2002.

[27] J. Duprat and J.M. Muller, “The CORDIC algorithm: new results for fast VLSI implementation”, *IEEE Trans. on Comput.*, vol. 42, pp. 168–178, 1993.

[28] J.J. Eggers, J.K. Su and B. Girod, “Robustness of a blind image watermarking scheme”, *Proc. IEEE Int. Conf. on Image Processing*, vol. 3, pp. 17–20, 2000.

[29] J.F. Fernandez and J. Rivero, “Fast algorithm for random numbers with exponential and normal distributions”, *Computers in Physics*, vol. 60, no. 1, 1996.

[30] J.F. Fernandez and C. Criado, “Algorithm for normal random numbers”, *Physics Review*, vol. 60, pp. 3361–3365, 1999.

[31] C.T Fike, *Computer Evaluation of Mathematical Functions*, Prentice-Hall, 1968.

[32] Flarion Technologies Inc., *Vector-Low-Density Parity-Check Coding Solution Data Sheet*, http://www.flarion.com, 2002.

[33] R.G. Gallager, “Low-density parity-check codes”, *IEEE Trans. on Inform. Theory*, vol. 8, pp. 21–28, 1962.

[34] R.G. Gallager, *Low-Density Parity-Check Codes*, MIT Press, 1963.

[35] C.W. Gardiner, *Handbook of Stochastic Methods*, Springer-Verlag, 1990.

[36] A. Ghazel, E. Boutillon, J.L. Danger, G. Gulak and H. Laamari, “Design and performance analysis of a high speed AWGN communication channel emulator”, *Proc. IEEE Pacific Rim Conf. on Commun. Comput. and Sig. Proc.*, vol. 2, pp. 374–377, 2001.

[37] GNU Project, *GCC 3.3 Manual*, http://gcc.gnu.org, 2003.

[38] N. Golshan, “A novel digital implementation of a Gaussian noise generator”, *Proc. 6th IEEE Instrumentation and Measurement Tech. Conf.*, pp. 256–257, 1989.

[39] J.F. Hart et al., *Computer Approximations*, Wiley, 1968.

[40] H. Hassler and N. Takagi, “Function evaluation by table look-up and addition”, *Proc. of the IEEE 12th Symp. on Comp. Arith.*, pp. 10–16, 1995.

[41] A. Hermanek, J. Kadlec, R. Matousek, M. Licko, C.I. Softley and J.N. Coleman, “Pipelined Logarithmic 32-bit ALU for Celoxica DK1”, *UTIA Research Report*, no. 2034, UTIA AV CR, 2001.

[42] S. Hong and W.E. Stark, “Design and implementation of a low complexity VLSI Turbo-Code decoder architecture for low energy mobile wireless communications”, *J. of VLSI Sig. Proc.*, pp. 2350–2354, 2000.

[43] C.J. Howland and A.J. Blanksby, "Parallel decoding architectures for low density parity check codes", *Proc. IEEE Int. Symp. on Circ. and Syst.*, vol. 4, pp. 742-745, 2001.

[44] C.J. Howland and A.J. Blanksby, "A 220mW 1 Gb/s 1024-Bit rate-1/2 low density parity check code decoder", *Proc. IEEE Custom Integrated Circ. Conf.*, pp. 293-296, 2001.

[45] F.C. Ionescu, "Theory and practice of a fully controllable white noise generator", *Proc. IEEE Int. Semiconductor Conf.*, vol. 2, pp. 319–322, 1996.

[46] V.K. Jain, S.A. Wadecar and L. Lin, "A universal nonlinear component and its application to WSI", *IEEE Trans. on Components, Hybrids and Manufacturing Tech.*, vol. 16, no. 7, pp. 656–664, 1993.

[47] B. Jung, H. Lenhof, P. Müller and C. Rüb, "Langevin dynamics simulations of macromolecules on parallel computers", *Macromol. Theory Simul.*, pp. 507–521, 1997.

[48] D.E. Knuth, "Seminumerical algorithms", *The Art of Computer Programming*, vol. 2, third edition, Addison-Wesley, 1997.

[49] I. Koren, *Computer Arithmetic Algorithms*, Prentice Hall, 1993.

[50] R.A Kronmal and A.V. Peterson, "The alias and alias-rejection-mixture methods for generating random variables from probability distributions", *Proc. of the 11th Conf. on Winter Simulation*, pp. 269–280, 1979.

[51] R.E. Ladner and M.J. Fischer, "Parallel prefix computation", *J. of the ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[52] C.L. Lawson, "Characteristic properties of the segmented rational minimax approximation problem", *Numer. Math.*, vol. 6, pp. 293–301, 1964.

[53] D. Lee, W. Luk and P.Y.K. Cheung, "Incremental programming for reconfigurable engines", *Proc. IEEE Conf. on Field-Prog. Tech.*, pp. 411–415, 2002.

[54] D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, "A hardware Gaussian noise generator for channel code evaluation", *Proc. IEEE Symp. on Field-Prog. Cust. Comput. Mach.*, 2003.

[55] D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, "Hardware function evaluation using non-linear segments", *Proc. Field-Prog. Logic and Applications*, 2003.

[56] J.L. Leva, "A fast normal random number generator", *ACM Trans. Math. Software*, vol. 18, no. 4, pp. 449–453, 1992.

[57] B. Levine, R.R. Taylor and H. Schmit, "Implementation of near Shannon Limit error-correcting codes using reconfigurable hardware", *Proc. IEEE Symp. on Field-Prog. Cust. Comput. Mach.*, pp. 217–226, 2000.

[58] D.M. Lewis, "Interleaved memory function interpolators with application to an accurate LNS arithmetic unit", *IEEE Trans. on Comput.*, vol. 43, no. 8, pp. 974–982, 1994.

[59] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman and V. Stemann, "Practical loss-resilient codes", *Proc. 29th Annu. ACM Symp. Theory of Computing*, pp. 150–159, 1997.

[60] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, "Analysis of low density codes and improved designs using irregular graphs", *Proc. 30th Annu. ACM Symp. Theory of Computing*, pp. 249–258, 1998.

[61] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, "Improved low-density parity-check codes using irregular graphs and belief propagation", *Proc. IEEE Symp. on Information Theory*, pp. 117, 1998.

[62] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman and V. Stemann, "Practical loss-resilient codes", *IEEE Trans. on Inform. Theory*, vol. 47, pp. 569–584, 2001.

[63] D.J.C MacKay, "Good error-correcting codes based on very sparse matrices", *IEEE Trans. on Inform. Theory*, 1999.

[64] D.J.C MacKay, S. Wilson and M.Davey, "Comparison of constructions of irregular Gallager codes", *IEEE Trans. on Comm.*, vol. 47, pp. 1449–1454, 1999.

[65] G. Marsaglia, "Generating discrete random variables in a computer", *Comm. of the ACM*, vol. 6, no. 1, pp. 37–38, 1963.

[66] G. Marsaglia, M.D. MacLaren and T.A. Bray, "A fast procedure for generating normal random variables", *Comm. of the ACM*, vol. 7, no. 1, pp. 4–10, 1964.

[67] G. Marsaglia and W.W. Tsang, "The Ziggurat method for generating random variables", *J. of Statistical Software*, vol. 5, issue 8, pp. 1–7, 2000.

[68] G. Masera, G. Piccinini, M. Ruo Roch and M. Zamboni, "VLSI architectures for Turbo codes", *IEEE Trans. on VLSI Systems*, vol. 7, no. 3, pp. 369–379, 1999.

[69] The MathWorks Inc., *MATLAB Manual*, ver. 6.5, http://www.mathworks.com, 2002.

[70] G. Mehta and H. Lee, "An FPGA implementation of the graph encoder-decoder for regular LDPC codes", *CRL Technical Report*, no. 8-4-2002-1, Communications Research Laboratory, University of Pittsbugh, 2002.

[71] O. Mencer, N. Boullis, W. Luk and H. Styles, "Parameterized function evaluation for FPGAs", *Field-Programmable Logic and Applications*, LNCS 2147, pp. 544–554, 2001.

[72] A. Miller and M. Gulotta, "PN generators using the SRL macro", *Xilinx Application Note XAPP211* (v1.1), January 2001.

[73] R.H. Morelos-Zaragoza, *The Art of Error Correcting Coding*, John Wiley & Sons, 2002.

[74] M.E. Muller, "A comparison of methods for generating normal deviates on digital computers", *J. of the ACM*, vol. 6, no. 3, pp. 376–383, 1959.

[75] G.S Muller and C.K. Pauw, "On the generation of a smooth Gaussian random variable to 5 standard deviations", *Proc. IEEE Southern African Conf. on Comm. and Sig. Processing*, pp. 62–66, 1988.

[76] J.M. Muller, *Elementary Functions: Algorithms and Implementation*, Birkhauser Verlag AG, 1997.

[77] J.M. Muller, "A few results on table-based methods", *Reliable Computing*, vol. 5, no. 3, pp. 279–288, 1999.

[78] Nallatech, *BenONE User Guide*, http://www.nallatech.com, 2002.

[79] T. Oenning and J. Moon, "Low density parity check coding for magnetic recording channels with media noise", *Proc. IEEE Conf. on Communications*, vol. 7, pp. 2189–2193, 2001.

[80] E.P. O'Grady and C.H. Wang, "Performance limitations in parallel processor simulations", *Trans. Soc. Comput. Simulation*, vol. 4, pp. 311–330, 1987.

[81] K. Page and E.M. Chau, "A FPGA ASIC communication channel systems emulator", *Proc. 6th IEEE ASIC Conf.*, pp. 345–348, 1993.

[82] B. Pandita and S.K. Roy, "Design and implementation of a Viterbi decoder using FPGAs", *Proc. IEEE Int. Conf. on VLSI Design*, pp. 611–614, 1999.

[83] T. Pavlidis and A.P Maika, "Uniform piecewise polynomial approximation with variable joints", *J. of Approximation Theory*, vol. 12, pp. 61–69, 1974.

[84] T. Pavlidis and S.L. Horowitz, "Segmentation of plane curves", *IEEE Trans. Comput.*, vol. C-23, pp. 860–870, 1974.

[85] T. Pavlidis, "Optimal piecewise polynomial L_2 approximation of functions of one and two variables", *IEEE Trans. on Comput.*, vol. C-24, pp. 98–102, 1975.

[86] T. Pavlidis, "Polygonal Approximations by Newton's Method", *IEEE Trans. on Comput.*, vol. C-26, pp. 800–807, 1977.

[87] W.H. Payne, "Normal random numbers: using machine analysis to choose the best algorithm", *ACM Trans. Math. Software*, vol. 3, no. 4, pp. 346–358, 1977.

[88] C.S. Petrie and J.A. Connelly, "The sampling of noise for random number generation", *Proc. IEEE Int. Symp. on Circ. and Syst.*, vol. 6, pp. 26–29, 1999.

[89] S.S Pietrobon, "A Turbo/MAP decoder for use in satellite circuits", *IEEE Int. Conf. on Inf. Comm. and Sig. Proc. (ICICS)*, vol. 1, pp. 427–431, 1997.

[90] S.S Pietrobon, "Implementation and performance of a Turbo/MAP decoder", *IEEE Int. J. of Satellite Comm.*, vol. 16, pp. 23–46, 1998.

[91] J.A Piñeiro, J.D. Bruguera and J.M. Muller, "Faithful powering computation using table look-up and a fused accumulation tree", *Proc. 15th IEEE Symp. on Comput. Arith.*, 2001.

[92] W.H. Press et. al., *Numerical Recipes in C*, second edition, Cambridge University Press, 1993.

[93] J.R. Rice, *The Approximation of Functions*, vol. 1,2, Addison-Wesley, 1964, 1969.

[94] T. Richardson, A. Shokrollahi and R. Urbanke, "Design of provably good low-density parity check codes", *IEEE Trans. on Inform. Theory*, 1999.

[95] T. Richardson and R. Urbanke, "Efficient encoding of low-density parity-check codes", *IEEE Trans. on Inform. Theory*, vol. 47, pp. 638–656, 2001.

[96] B.D. Ripley, *Stochastic Simulation*, Wiley, 1987.

[97] S. Rocchi and V. Vignoli, "A chaotic CMOS true-random analog/digital withe noise generator", *Proc. IEEE Int. Symp. on Circ. and Syst.*, pp. 463–466, 1999.

[98] C. Rose, *J. of Economic Dynamics and Control*, vol. 19, no. 1391, 1997.

[99] C. Rüb, "On Wallace's method for the generation of normal variates", *MPI Informatik Research Reports*, no. MPI-I-98-1-020, Max-Planck-Institut für Informatik, Germany, 1998.

[100] M.F. Schollmeyer and W.H. Tranter, "Noise generators for the simulation of digital communication systems", *Proc. 24th Ann. Simulation Symp.*, pp. 264–275, 1991.

[101] M.J. Schulte and E.E. Swartzlander Jr, "Hardware designs for exactly rounded elementary functions", *IEEE Trans. on Comput.*, vol. 43, no. 8, pp. 964–973, 1994.

[102] M.J. Schulte and J.E. Stine, "Symmetric bipartite tables for accurate function approximation", *Proc. 13th IEEE Symp. on Comput. Arith.*, vol. 48, no. 9, pp. 175–183, 1997.

[103] M.J. Schulte and J.E. Stine, "Approximating elementary functions with symmetric bipartite tables", *IEEE Trans. on Comput.*, vol. 48, no. 9, pp. 842-847, 1999.

[104] C.E. Shannon, "A mathematical theory of communication", *Bell System Technical Journal*, no. 27, pp. 379–423, 1948.

[105] N. Sidahao, G.A. Constantinides and P.Y.K. Cheung, "Architectures for function evaluation on FPGAs", *Proc. IEEE Int. Symp. on Circ. and Syst.*, vol. 2, pp .804–807, 2003.

[106] M. Smith, J. Villasenor, C. Jones and E. Vallés, "A high throughput low complexity decoder architecture for irregular LDPC codes", *Proc. IEEE Military Comm. Conf. (MILCOM)*, 2003.

[107] H. Song, J. Liu and B.V.K. Vijaya Kumar, "Low-density parity-check (LDPC) codes for optical data storage", *Joint Int. Symp. on Optical Memory (ISOM) and Optical Data Storage (ODS)*, 2002.

[108] S. Swaminathan, R. Tessier, D. Goeckel, and W. Burleson, "A dynamically reconfigurable adaptive Viterbi decoder", *Proc. ACM/SIGDA Symp. on Field-Prog. Gate Arrays*, 2002.

[109] K. Tae, J. Chung and D. Kim, "Noise generation system using DCT", *Proc. IEEE Int. Symp. on Circ. and Syst.*, vol. 4, pp. 29–32, 2002.

[110] M. Tanner, "A recursive approach to low complexity codes", *IEEE Trans. on Inform. Theory*, vol. IT-27, pp. 533–547, 1981.

[111] V. Thilak and A. Nosratinia, "Robust bandlimited watermarking with trellis coded modulation", *Proc. IEEE Int. Conf. on Image Processing*, 2002.

[112] D.E. Thomas and P.R. Moorby , *The Verilog Hardware Description Language* , third edition, Kluwer Academic Publishers, 1996.

[113] J. Vedral and J. Holub, "Oscilloscope testing by means of stochastic signal", *Measurement Science Review*, vol. 1, no. 1, 2001.

[114] F. Viglione, G. Masera, G. Piccinini, M. Ruo Roch and M. Zamboni, "A 50 Mbit/s iterative Turbo-decoder", *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pp. 176–180, 2000.

[115] Xilinx Inc., *Virtex-II User Guide v1.5*, http://www.xilinx.com, 2002.

[116] J.E. Volder, "The CORDIC trigonometric computing technique", *IEEE Trans. on Elec. Comput.*, vol. EC-8, no. 3, pp. 330–334, 1959.

[117] C.S. Wallace, "Fast pseudorandom generators for normal and exponential variates", *ACM Trans. on Math. Software (TOMS)*, vol. 22, no. 1, pp. 119–127, 1996.

[118] J.S Walther, “A unified algorithm for elementary functions”, *Proc. AFIPS Spring Joint Comput. Conf.*, pp. 379–385, 1971.

[119] N. Wax, *Noise and Stochastic Processes*, Donver Publications Inc., 1954.

[120] “Additive White Gaussian Noise (AWGN) Core v1.0”, *Xilinx Product Specification*, October 2002.

[121] D. Yeh, G. Feygin and P. Chow, “RACER: A reconfigurable constraint-length 14 Viterbi decoder”, *Proc. IEEE Symp. on Field-Prog. Cust. Comput. Mach.*, pp. 60–69, 1996.

[122] E. Yeo, P. Pakzad, B. Nikolic and V. Anantharam, “VLSI architectures for iterative decoders in magnetic recording channels”, *IEEE Trans. on Magnetics*, vol. 37, no. 2,

[123] K.W. Yip and T.S. Ng, “A simulation model for Nakagami-m fading channels, m<1”, *IEEE Trans. on Comm.*, vol. 48, no. 2, pp. 214–221, 2000.

[124] T. Zhang, Z. Wang and K.K. Parhi, “On finite precision implementation of low-density parity-check codes decoder”, *IEEE Int. Symp. on Circuits and Systems (ISCAS)*, vol. 6, pp. 202–205, 2001.

[125] T. Zhang and K.K. Parhi, “VLSI implementation-oriented (3,k)-regular low-density parity-check codes”, *Proc. IEEE Workshop on Signal Processing Systems (SiPS): Design and Implementation*, pp. 25–36, 2001.

[126] T. Zhang and K.K. Parhi, “54 Mbps (3,6)-regular FPGA LDPC decoder”, *Proc. IEEE Workshop on Signal Processing Systems (SiPS): Design and Implementation*, 2002.

[127] H. Zhun and H. Chen, “A truly random number generator based on thermal noise”, *Proc. IEEE Int. Conf. on ASIC*, pp. 862–864, 2001.