A Gaussian Noise Generator for Hardware-Based Simulations

Dong-U Lee, *Student Member*, *IEEE*, Wayne Luk, *Member*, *IEEE*,
John D. Villasenor, *Senior Member*, *IEEE*, and Peter Y.K. Cheung, *Senior Member*, *IEEE*

Abstract—Hardware simulation offers the potential of improving code evaluation speed by orders of magnitude over workstation or PC-based simulation. We describe a hardware-based Gaussian noise generator used as a key component in a hardware simulation system, for exploring channel code behavior at very low bit error rates (BERs) in the range of 10^{-9} to 10^{-10}. The main novelty is the design and use of nonuniform piecewise linear approximations in computing trigonometric and logarithmic functions. The parameters of the approximation are chosen carefully to enable rapid computation of coefficients from the inputs while still retaining high fidelity to the modeled functions. The output of the noise generator accurately models a true Gaussian Probability Density Function (PDF) even at very high σ values. Its properties are explored using: 1) several different statistical tests, including the chi-square test and the Anderson-Darling test, and 2) an application for decoding of Low-Density Parity-Check (LDPC) codes. An implementation at 133MHz on a Xilinx Virtex-II XC2V4000-6 FPGA produces 133 million samples per second, which is seven times faster than a 2.6GHz Pentium-IV PC; another implementation on a Xilinx Spartan-IIE XC2S300E-7 FPGA at 62MHz is capable of a three 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 50 times faster than a 2.6GHz Pentium-IV PC. We illustrate the deterioration of clock speed with the increase in the number of instances.

Index Terms—Algorithms implemented in hardware, error-checking, gate arrays, simulation.

1 Introduction

Sequences of random numbers with Gaussian probability distribution functions are needed to simulate a wide variety of natural phenomena [18], [54]. Applications of such sequences include channel code evaluation [21], [25], watermarking [15], oscilloscope testing [51], simulation of economic systems [5], [49], and molecular dynamics simulations [22].

Numerical methods for Gaussian random number generation have a long history in mathematics and communications. As described in [23] and the references cited therein, most methods involve initially generating samples of a uniform random variable and then applying a transformation to obtain samples drawn from a unit-variance, zero-mean Gaussian PDF $f(x) = (1/\sqrt{2\pi})\, e^{-x^2/2}$. 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 [30] and provide strong motivation to examine the Gaussian noise generation problem in the framework of limited word length and limited computational and memory resources. For example, Low-Density Parity-Check (LDPC) codes are currently the focus of intensive interest in the coding community due to their ability to approach the Shannon bound very closely and with only moderate decoding complexity [17], [32]. Computer simulations to examine LDPC code behavior can be time-consuming, particularly when the behavior at low bit error rates (BERs) in the error floor region is being studied. Hardware-based simulation [30] offers the potential of speeding up code evaluation by several orders of magnitude, but is feasible only if suitably fast and high-quality noise generators can be implemented in hardware alongside the channel decoder.

The principal contribution of this paper is a hardware Gaussian noise generator that offers quality suitable for simulations involving very large numbers of noise samples. The noise generator occupies approximately 10 percent of the resources on a Xilinx Virtex-II XC2V4000-6 device [52], while producing over 133 million samples per second. In contrast to previous work, we focus specific attention on the accuracy of the noise samples in the high σ regions of the Gaussian PDF, which are particularly important in achieving accurate results during large simulations. The key novelties of our work include:

- *D. Lee and W. Luk are with the Department of Computing, Imperial College, London, UK. E-mail: {dong.lee, w.luk}@ic.ac.uk.*
- *J.D. Villasenor is with the Electrical Engineering Department, University of California, Los Angeles, CA 90095. E-mail: villa@icsl.ucla.edu.*
- *P.Y.K. Cheung is with the Department of Electrical and Electronic Engineering, Imperial College, London, UK. E-mail: p.cheung@ic.ac.uk.*

Manuscript received 26 July 2003; revised 18 Apr. 2004; accepted 11 June 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0102-0703.

- a hardware architecture which involves the use of nonuniform 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 Anderson-Darling test, as well as through application to decoding of Low-Density Parity-Check (LDPC) codes.

The rest of this paper is organized as follows: Section 2 covers background material and previous work. Section 3 briefly reviews the Box-Muller algorithm and discusses how each of its steps can be handled in a hardware architecture. Section 4 presents a method for function evaluation based on nonuniform segments. Section 5 explains how the function evaluation method is used to compute the functions in the Box-Muller algorithm. Section 6 describes technology-specific implementation of the hardware architecture. Section 7 presents our LDPC simulation framework. Section 8 discusses evaluation and results and Section 9 offers conclusions and future work.

2 Background

Previous work on Gaussian noise generation can be divided into two types: the generation of Gaussian noise using a combination of analog components and the generation of pseudorandom noise using purely digital components. 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. In addition, when simulating communication systems, we may wish to use pseudorandom 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 [3], [39], and [45].

Digital methods for generating random Gaussian variables are almost always based on transformations or operations on uniform random variables [50]. The most widely used methods are: various rejection-acceptance methods [1], [29], [31], [33], [36], the use of the central limit theorem [23], the inversion method [19], and the Box-Muller method [7]. The rejection-acceptance methods, while popular in software implementations, contain conditional loops such that the output rates are not constant, making them 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 [7], which generates random Gaussian variables by transforming two uniform random variables over $[0, 1)$. Properly implemented, it offers a predictable output rate and, in combination with the central limit theorem, extremely good Gaussian modeling.

There is very little previous work on digital hardware Gaussian noise generators. The most relevant publications are probably [6] and [55], which discuss designs targeting Field-Programmable Gate Arrays (FPGAs). We present a design with significantly improved efficiency which also passes statistical tests widely used for testing normality. In addition, previous work produces 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.

The Box-Muller algorithm requires the approximation of nonlinear functions. Advanced FPGAs enable the development of low-cost and high-speed function evaluation units, customizable to particular applications [37]. Applications that do not require high precision often employ direct table look-ups. However, this becomes impractical for precisions higher than a few bits because the size of the table is exponential in the input size.

CORDIC [53] has been a popular method for evaluating functions, involving only shift and add operations. However, it has an execution time which is linearly proportional to the number of bits of the operands and is not suitable for applications requiring high accuracy and speed. High radix CORDIC [2] algorithms can be used to reduce the number of iterations. However, they suffer from increased complexity in digit selection process for microrotation.

Piñeiro et al. [46] divide the interval into several uniform segments. For each segment, they store the second degree minimax polynomial approximation coefficients and accumulate the partial terms in a fused accumulation tree. Such approximations using uniform segments [8], [20], [44] are suitable for functions with linear regions, but are inefficient for nonlinear functions, especially when the function varies exponentially. It is desirable to choose the boundaries of the segments to cater to the nonlinearities of the function. Highly nonlinear 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.

Methods that use nonuniform segment sizes to cater for the nonlinearities of functions have been proposed before. Cantoni [9] uses optimally placed segments and presents an algorithm to find such segment boundaries. However, although this approach minimizes the number of segments required, such arbitrarily placed segments are impractical for actual hardware implementation since the hardware circuitry to find the right segment for a given input would be too complex. Combet et. al. [12] and Mitchell [40] use segments that increase by powers of two to approximate the base two logarithm. The principles of their approaches are similar to ours, but we use a more advanced segmentation scheme by using segments that increase and decrease by powers of two. We also employ a hierarchy of uniform segments and segments that vary by powers of two to cover the nonlinearities of different functions better. Moreover, we present a hardware architecture which is suitable for FPGA implementation.

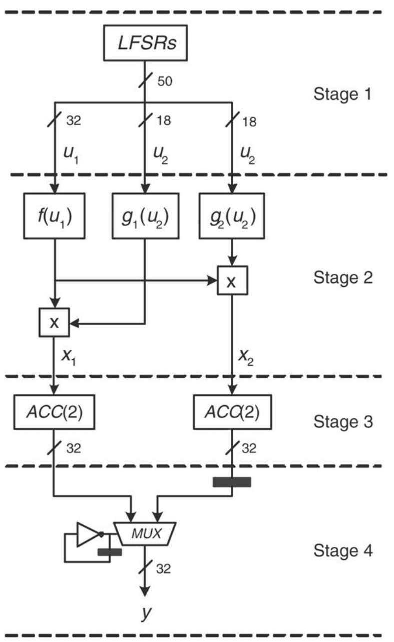

Fig. 1. Gaussian noise generator architecture. The black boxes are buffers.

3 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 is presented in Section 6.

The Box-Muller method is conceptually straightforward. Given two independent 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

$$f(u_1) = \sqrt{-\ln(u_1)}, \tag{1}$$

$$g_1(u_2) = \sqrt{2}\,\sin(2\pi\, u_2), \tag{2}$$

$$g_2(u_2) = \sqrt{2}\,\cos(2\pi\, u_2), \tag{3}$$

the products,

$$x_1 = f(u_1)\, g_1(u_2), \tag{4}$$

$$x_2 = f(u_1)\, g_2(u_2), \tag{5}$$

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

The above equations lead to an architecture that has four stages (Fig. 1).

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 [6]; what distinguishes our work is the detail of the functional implementation developed to deal with: 1) Gaussian noise with high σ values and 2) evaluations using commonly used statistical tests.

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) [11]. 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 m-bit LFSRs with irreducible polynomials can produce random numbers with periodicity of $2^m - 1$, the 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 σ value 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. In addition, the precisions of u_1, u_2, g_1, and g_2 should be large enough so that there are enough diversities in the outputs. Low precisions will cause the statistical tests to fail.

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 functions using methods such as CORDIC [53] leads to prohibitively long computation times. A direct 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, we use a two-step process based on nonuniform piecewise linear approximation. Our approach is described in Sections 4 and 5.

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 $x_1, x_2, \ldots, x_l$ with unit variance and zero mean, the distribution of

$$\frac{x_1 + x_2 + ... + x_l}{\sqrt{l}}$$

tends to be normally distributed as $l \to \infty$. We find that $l = 2$ is sufficient, so we use an accumulator (the $ACC(2)$ component shown in Fig. 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 $\sqrt{2}$ ((2) and (3)), this multiplication is, in effect, canceled 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].

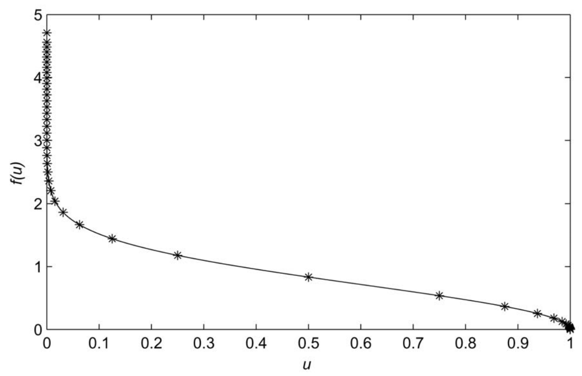

Fig. 2. The f function. The asterisks indicate the boundaries of the linear approximations.

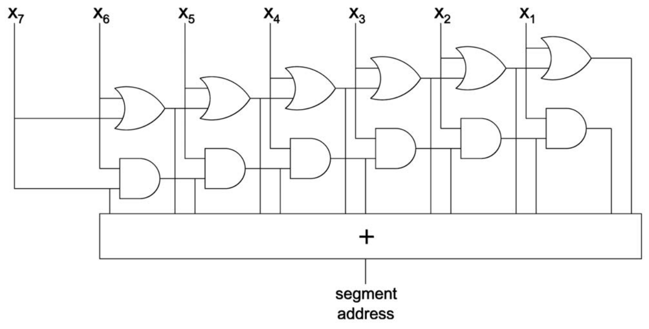

Fig. 3. Circuit to calculate the segment address for a given input x. The adder counts the number of ones in the output of the two prefix circuits. Note that the least-significant bit x_o is not required.

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. The buffer after the second $ACC(2)$ is needed to ensure one valid noise sample is fed to the multiplexor every clock cycle, rather than two valid samples every two clock cycles.

Two further remarks about this architecture can 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 by using larger LFSRs and higher σ values can be obtained using more bits for u_1, both with very little increase in complexity.

4 Function Evaluation for Nonuniform Segmentation

This section presents a method for function evaluation based on an innovative technique involving nonuniform segmentation. The interval of approximation is divided into a set of subintervals, called segments. The best-fit straight line, in a minimax sense (minimize worst-case error), to each segment is found. A look-up table is used to store the coefficients for each line segment and the functions can then be evaluated using a multiplier and an adder to calculate the linear approximation. Uniform segmentation methods have been proposed which involve similar hardware [37].

Using well-known methods that compute elementary functions such as CORDIC [53], the evaluation of compound functions is a multistage process. Consider the evaluation of the f function as defined in (1) over the interval $(0, 1)$ (Fig. 2). Using CORDIC, the computation of this function is a two-stage process: the logarithm of x followed by the square root. With our approach, we look at the entire function over the given domain and, therefore, we do not need to have two stages. As shown in Fig. 2, the greatest nonlinearities of the f function occur in the regions close to zero and one. If uniform segments are used, a large number of small segments would be required to get accurate approximations in the nonlinear 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 nonlinear 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, 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: 1) The segment lengths used in a given region depend on the local linearity, with more segments deployed for regions of higher nonlinearity, and 2) the boundaries between segments are chosen such that the task of identifying which segment to use for a given input can be rapidly performed.

As an example to illustrate our approach, consider approximating f with an 8-bit input. Using the traditional approach, the most-significant bits of u 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 adopt small segments for nonlinear 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 horizontal axis of Fig. 2. 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 (Fig. 3) for a given input x. It checks 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 a shorter critical path but requiring more area can be used [24]. 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 Fig. 3, 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 the path from x_6

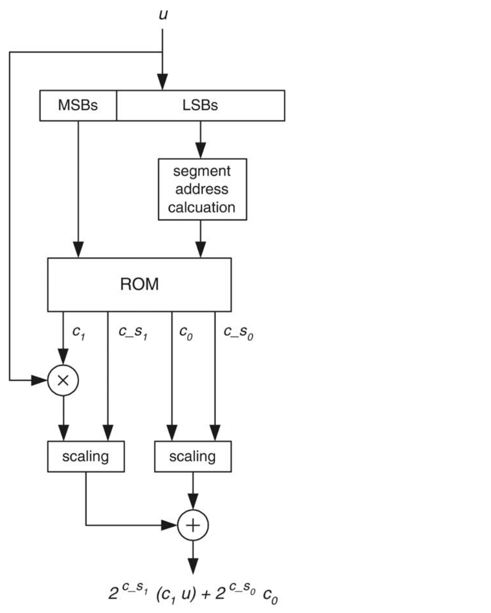

Fig. 4. Function evaluator architecture based on nonuniform segmentation.

or x_7 to the output of the adder. By introducing pipeline registers between the gates, higher throughput can easily be achieved.

When approximating f with 32-bit inputs based on polynomials of the form

$$p(u) = c_1 \times u + c_0, \tag{6}$$

the gradient of the steepest part of the curve is on 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, resulting in variation of 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. Note that the precision of the approximation $p(x)$ depends on the maximum error desired between $p(x)$ and the actual function.

It is also possible to divide the input interval into uniform or nonuniform intervals and have uniform or nonuniform 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 nonuniform segments. This hybrid combination of nested uniform and nonuniform segments provides a flexible way to choose the segment boundaries.

The architecture of our function evaluator, shown in Fig. 4, is based on first order polynomials. 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 ROM outputs the four coefficients for the

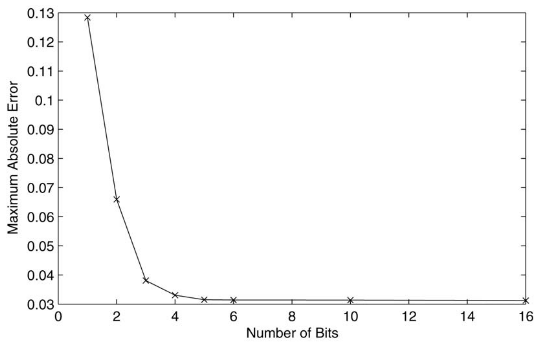

Fig. 5. Variation of function approximation error with number of bits for the gradient of the f function.

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.

This work on function evaluation has been extended by employing a hierarchy of uniform and nonuniform segments. Details about this hierarchical segmentation scheme can be found in [27].

5 Function Evaluation for Noise Generator

This section explains in detail how the function evaluation method based on nonuniform segmentation is used to compute the f and g functions for Gaussian noise generation ((1)-(3)). We first consider the f function. As stated earlier, the greatest nonlinearities of this function occur in the regions close to zero and one. To be consistent with the change in linearity, we use line segment locations to boundaries at locations 2^{n-32} for $0 < u \leq 0.5$ and $1 - 2^{-n}$ for $0.5 < u \leq 1$, where $0 \leq n < 32$. A total of 59 segments are used to approximate this function, as shown in Fig. 2. Since f approaches infinity for u values close to zero, the smallest u value is 2^{-32}, resulting in a maximum output value of around 4.7.

The maximum absolute error of this approximation is 0.020 (compared against IEEE double precision). However, this is the case only if we have infinite precision for the coefficients and data paths, which is not realistic. Multipliers take a 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. Fig. 5 shows how the maximum absolute error varies with the number of bits used for the gradient of the f function. Similar tests are performed for the y-intercept coefficients and various data paths. The figure indicates that six bits are sufficient to give a maximum absolute error of 0.031. Our requirement is faithfully rounded results [16] (results are rounded to the nearest or next nearest), where the approximation should differ from the true value by less than one unit in the last place (ulp). With this error, it is sufficient to give an output accuracy of eight bits (three bits for integer and five for fraction). If uniform segments are used, small segment size

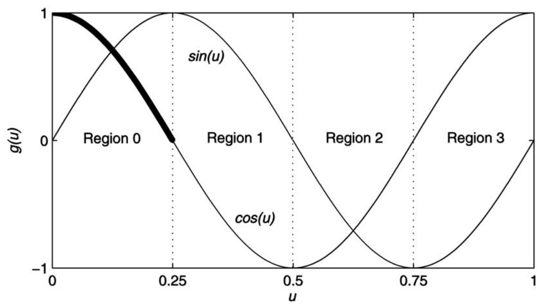

Fig. 6. The g functions. Only the thick line is approximated; see Fig. 4. The most significant two bits of u_2 are used to choose which of the four regions to use; the remaining bits select a location within Region 0.

would be needed in order to cope with the highly nonlinear 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 nonuniform approach. It is clear that our approach works well, especially for functions with exponential behavior.

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, labeled 0 to 3 in Fig. 6. To evaluate the functions g_1 and g_2, due to the symmetry of the sine and cosine functions, only the input range $[0, 1/4)$ for g_1 needs to be approximated [41]. The specific axis-partitioning technique for f is unsuitable for g_1 since the nonlinearities of the two functions are different. If the same technique is 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 Fig. 7. The curve is divided into four uniform intervals and, within each interval, nonuniform segmentation is applied. Note that, for each interval, not all taps are taken from the segment address calculator. The boundaries are chosen in a way to minimize the approximation error. For the first three intervals, nonuniform segments increasing and decreasing by powers of two with six segments each are used. For the last interval, only three segments are used by omitting taps. Since this interval is the most nonlinear, sufficiently good accuracy can be achieved with only a few segments. We use a total of 21 segments to approximate this function.

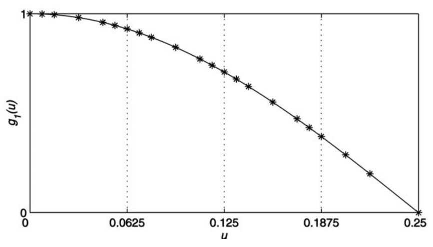

Fig. 7. Approximation for g_1 over $[0, 1/4)$. The asterisks indicate the segment boundaries of the linear approximations.

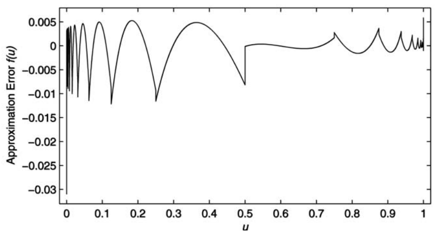

Fig. 8. Approximation error to f. The worst case and average errors are 0.031 and 0.000048, respectively.

With finite precision on the coefficients and data paths, the maximum absolute error of this approximation is 0.0035, which is sufficient to give an output accuracy of eight 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 nonlinearities.

The maximum absolute errors to the two functions, 0.031 and 0.00079, may seem to be rather high. However, the average errors for the two functions are in fact 0.000048 and 0.0000012, respectively. Lower average approximation errors to the functions ensure overall higher noise quality. The error plots for the approximations to f and g_1 are shown in Figs. 8 and 9.

Table 1 shows a comparison of the number of segments for the two functions for nonuniform and uniform segmentation in order to achieve the same worst-case error. Note that, for uniform segmentation, the number of segments needs to be a power of two. This is because the most-significant n bits are used for addressing. For instance, the actual number of uniform segments needed for the f function is 617 million, but one billion segments are used, which is the next power of two (2^{30}). We do not have this kind of restriction with our nonuniform addressing scheme. The table also shows the number of bits used for each coefficient in the look-up tables. The look-up table sizes are $59 \times (6 + 5 + 32 + 5) = 2,832$ bits for the f function and $21 \times (8 + 4 + 16 + 4) = 672$ bits for the g_1 function, giving a total look-up table size of just 3,504 bits for all three functions. With such a small look-up table size, all the coefficients can be stored on-chip for fast access. Note that the g_2 function shares the same look-up table with g_1.

6 Implementation

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

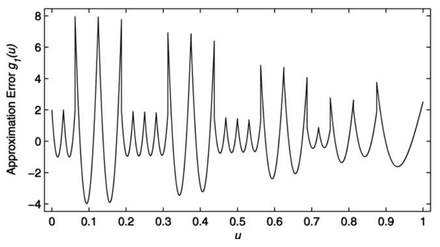

Fig. 9. Approximation error to g_1. The worst case and average errors are 0.00079 and 0.0000012, respectively.

TABLE 1
Comparing Two Segmentation Methods

function	non-uniform	uniform	c_1	c_s_1	c_0	c_s_0
f	59	1 billion	6	5	32	5
g_1	21	32	8	4	16	4

The second column shows the comparison of the number of segments for nonuniform segmentation. The third column shows the number of bits used for the coefficients to approximate f and g_1.

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 (lower bit widths cause the statistical tests to fail). 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. In terms of hardware resources, the size of these uniform random number inputs (u_1, u_2, g_1, and g_2) affects the size of the multipliers and adders (see Fig. 4). The more bits there are, the more one needs larger multipliers and adders.

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. In order for the LFSRs to iterate through this large period, they are configured with polynomials which will produce maximum sequence lengths for a given LFSR size [38].

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 23,040 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. A 60-bit LFSR using SRL16s instead of flipflops can be packed into three slices instead of 32 [38]. So, we just need 150 slices for the 50 LFSRs. Note that all 50 LFSRs are initialized with random seeds.

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 trade off 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 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 six 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.

Several FPGA implementations have been developed, using the Handel-C hardware compiler from Celoxica [10]. We have mapped and tested the design onto a hardware platform with a Xilinx Virtex-II XC2V4000-6 device. This design occupies 2,514 slices, eight block multipliers, and two block RAMs, which takes up around 10 percent of the device. Stage two, the function evaluator, takes up 2,137 slices or 85 percent of the slices used. A pipelined version of our design operates at 133MHz, 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 62MHz and has 2,829 slices and eight block RAMs, which requires over 90 percent of this device. This implementation can produce 133 million samples in around two 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 high fan-out of the clock tree. For instance, a design with three instances of our noise generator takes up around 32 percent of the resources in an XC2V4000-6 device; it runs at 126MHz, producing 378 million noise samples per second.

In Section 8, the performance of the hardware designs presented above is compared with those of software implementations.

7 LDPC Simulation System

Error correcting coding (ECC) [42] is a critical part of modern communications systems, where it is used to detect and correct errors introduced during a transmission over a channel. In the past few years, LDPC codes have received a lot attention because of their excellent performance [32]. They have been widely considered as the most promising candidate ECC scheme for many applications in telecommunications and storage devices.

If the binary Hamming distance between all pairs 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

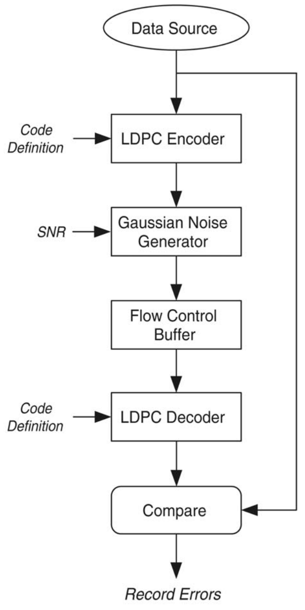

Fig. 10. LDPC simulator.

code constructions in terms of bit error rate (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 $10^{10} \sim 10^{12}$ code symbols and workstation simulations provide inadequate means of finding a statistically sufficient set of error events. We are working on a hardware LDPC simulation framework that is several orders of magnitude faster than simulations provided by workstations or an array of workstations.

Our real-time LDPC simulation framework has been implemented on a Virtex-II prototyping board from Nallatech [43] and a block diagram is provided in Fig. 10. The LDPC encoder follows an algorithm suggested in [47] and has been implemented in hardware [28]. Details of our LDPC decoder are described in [21]. Our noise generator offers significant value to the system as a Monte Carlo simulator since noise quality at high SNR (tails of the Gaussian) is essential.

8 Evaluation and Results

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

In order to ensure the randomness of the uniform random samples u_1 and u_2, we have tested the LFSR with the Diehard tests [34]. The LFSR passed all the tests, indicating that the uniform random samples generated are indeed uniformly randomly distributed.

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 Anderson-Darling (A-D) test [13]..

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 t 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 given by

$$\chi^2 = \sum_{i=1}^{k} \frac{(Y_i - tp_i)^2}{tp_i}. \tag{7}$$

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 extend 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 A-D test deals with continuous properties. It is a modification of the Kolmogorov-Smirnov (K-S) test [23] and gives more weight to the tails than the K-S test does. The K-S test is distribution free in the sense that the critical values do not depend on the specific distribution being tested. The A-D test makes use of the specific distribution (normal in our case) in calculating critical values. For comparing a data set to a known CDF $F(x)$, the A-D statistic A^2 is defined by

$$A^2 = \sum_{i=1}^{N} \frac{1-2i}{N} \left[\ln F(x_i) + \ln(1 - F(x_{N+1-i}))\right] - N, \tag{8}$$

where x_i is the ith sorted and standardized sample value and N is the sample size.

A p-value [13] can be obtained from the tests, which is the probability that the deviation of the observed from that expected is due to chance alone. 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.

Figs. 11, 12, and 13 illustrate the effect on the PDF of different implementation choices. Fig. 11 shows the PDF obtained when 17 and 6 linear approximations are used for f and g_1, 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

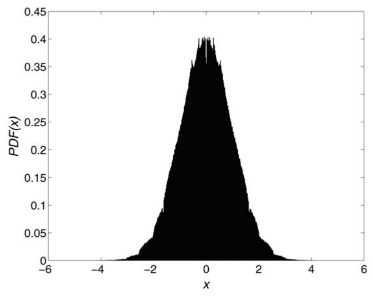

Fig. 11. PDF of the generated noise with 17 approximations for f and six for g for a population of four million. The p-values of the χ^2 and A-D tests are 0.00002 and 0.0084, respectively.

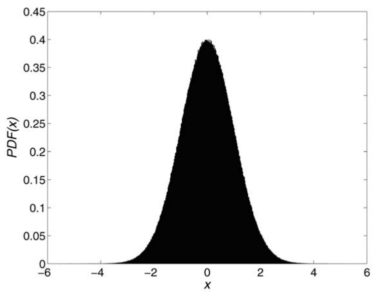

Fig. 13. PDF of the generated noise with 59 approximations for f and 21 for g with two accumulated samples for a population of four million. The p-values of the χ^2 and A-D tests are 0.3842 and 0.9058, respectively.

in the approximation of f and g_1. These distinct errors cause the χ^2 and A-D tests to fail. Increasing the number of linear approximations to 59 and 21, respectively, leads to the PDF shown in Fig. 12. It is clear that the error regions have decreased significantly. However, although this passes the A-D test, it fails the χ^2 test when the sample size is sufficiently large. When the further enhancement of summing two successive samples as discussed earlier is added, the PDF of Fig. 13 results.

This implementation passes the statistical tests even with extremely large numbers of samples. We have run a simulation of 10^{10} samples to calculate the p-values for the χ^2 and A-D test. For the χ^2 test, we use 100 bins for the x axis over the range $[-7, 7]$. The p-values for the χ^2 and A-D tests are found to be 0.3842 and 0.9058, respectively, which are well above 0.05, indicating that the generated noise samples are indeed normally distributed. To test the noise quality in the high σ regions, we run a simulation of 10^7 samples over the range $[-7, -4]$ and $[4, 7]$ with 100 bins. This is equivalent to a simulation size of over 10^{11} samples. The p-values for the χ^2 and A-D tests are found to be 0.6432 and 0.9143, showing that the noise quality even in the high σ regions is high.

In order to explore the possibility of temporal statistical dependencies [48] between the Gaussian variables, we generate scatter plots showing pairs y_i and y_{i+1}. This is to test serial correlations between successive samples, which can occur if the noise generator is improperly designed. If correlations exist, certain patterns can be seen in the scatter plot [48]. An example based on 10,000 Gaussian variables is shown in Fig. 14, which displays no obvious correlations.

Our hardware implementations, described in Section 6, have been compared to several software implementations based on the polar method [23] and the Ziggurat method [36], which are the fastest methods for generating Gaussian noise for instruction processors. The uniform random number generator used for the software implementations is the mrand48() C function in UNIX, which uses a linear congruential algorithm [23] and 48-bit integer arithmetic (period of 2^{48}). This algorithm can generate one billion 48-bit uniform random numbers on a Pentium-IV 2.6GHz PC in just 23 seconds.

The results are shown in Table 2. It can be seen that our hardware designs are faster than software implementations by 3-200 times, depending on the device used and the resource utilization. Looking at the PC results, we can see that the Ziggurat method performs significantly better than the polar method on both the Athlon and the Pentium-IV.

Fig. 15 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 expected because the clock speed of the design deteriorates as the number of noise generators increases. This deterioration is probably due to the increased routing congestion and delay.

We have used our noise generator in LDPC decoding experiments [21]. Although the output precision of our noise generator is 32 bits, 16 bits are found to be sufficient for our LDPC decoding experiments (other applications, such as financial modeling using the BGM model, require higher precisions [4]). To obtain a benchmark, we performed LDPC decoding using a full precision (64-bit floating-point representation) software implementation of

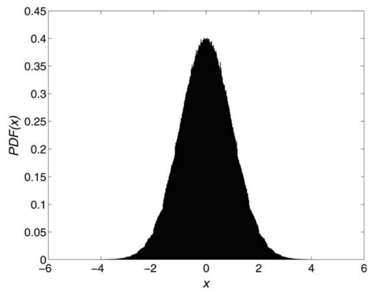

Fig. 12. PDF of the generated noise with 59 approximations for f and 21 for g for a population of four million. The p-values of the χ^2 and A-D test are 0.0012 and 0.3487, respectively.

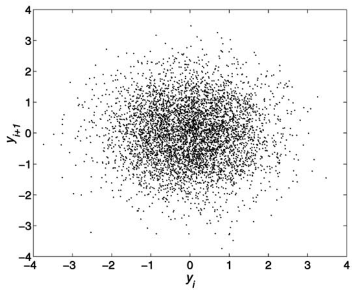

Fig. 14. Scatter plot of two successive accumulative noise samples for a population of 10,000. No obvious correlations can be seen.

TABLE 2
Performance Comparison: Time for Producing One Billion Gaussian Noise Samples

platform	speed [MHz]	method	time [s]
XC2V4000-6 FPGA	105	96% usage	1
XC2V4000-6 FPGA	126	32% usage	2.6
XC2V4000-6 FPGA	133	10% usage	7.5
XC2S300E-7 FPGA	62	90% usage	16
Intel Pentium-IV PC	2600	Ziggurat	50
AMD Athlon PC	1400	Ziggurat	72
Intel Pentium-IV PC	2600	Polar	147
AMD Athlon PC	1400	Polar	214

All PCs are equipped with 1GB DDR RAM. The XC2V4000-6 FPGA belongs to the Xilinix Virtex-II family, while the XC2S300E-7 belongs to the Xilinx Spartan II-E family. The software implementations are written in C, generating single precision floating-point numbers, and are compiled with the GCC 3.2.2 compiler.

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} (10^{12} codewords are simulated). To generate 10^{12} noise samples on a 2.6GHz Pentium-IV workstation, it takes over 11 hours, whereas a single instance of our hardware noise generator takes just over two hours. On a workstation where LDPC encoding, noise generation, and LDPC decoding are performed, the simulation time for 10^{12} codeword samples will be a lot longer than 10 hours since all three modules need to be performed. However, in our hardware simulation, we have the advantage of running all three modules in parallel. Although the hardware implementation of our hardware LDPC decoder is currently at a preliminary stage (implemented serially), it has a throughput of around 500Kbps, which is over 20 times faster than our workstation-based simulations. We are currently in the process of implementing a fully parallel scalable decoder, which we predict will be several orders of magnitude faster than traditional software simulations.

Comparing our implementation with other hardware Gaussian noise generators, the only implementation known on a Xilinx FPGA is the AWGN core [55] from Xilinx. This implementation follows the ideas presented in [6]. Although this core is around twice as fast as and four times smaller than our design, it is only capable of a maximum σ value of 4.7 (whereas we can achieve 6.7 σ and more). In addition, we have tested the design with our statistical tests and found out that the noise samples fails the χ^2 test after around 200,000 samples. Hence, we found the design to be inadequate for our low BER and high quality LDPC decoding experiments.

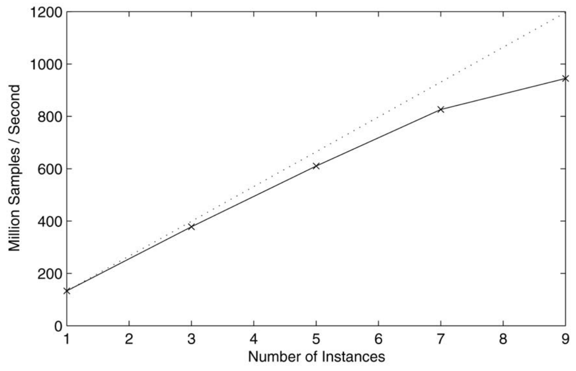

Fig. 15. Variation of output rate against the number of noise generator instances. The dotted line shows the linear relationship between the output and the number of instances if the clock speed does not deteriorate with the increasing number of instances.

9 Conclusion

We have presented a hardware-based Gaussian noise generator designed to facilitate channel code simulations implemented in hardware which involve very large numbers of samples. A key aspect of the design is the use of nonuniform 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 occupies approximately 10 percent of a Xilinx Virtex-II XC2V4000-6 FPGA and 90 percent of a Xilinx Spartan-IIE XC2S300E-7 and can produce 133 million samples per second. The performance can be improved by exploiting parallelism: An XC2V4000-6 FPGA with nine parallel instances of the noise generator at 105MHz can run 50 times faster than a 2.6GHz Pentium-IV PC. Statistical tests, including the χ^2 test and the A-D test, as well as application in LDPC decoding, have been used to confirm the quality of the noise samples. The output of the noise generator accurately models a true Gaussian PDF even at very high σ values. Ongoing and future work includes further refinement of our hardware noise generator architecture for various applications, for instance, those which involve different channels such as Rayleigh [14], Ricean and Nakagami-m [56] channels.

Acknowledgments

The authors thank Altaf Abdul Gaffar, Jun Jiang, Shay Ping Seng, Chris Jones, Tao Tian, Ray Cheung, and the reviewers for their assistance. The support of Celoxica Limited, Xilinx Inc., the UK Engineering and Physical Sciences Research Council (Grant number GR/N 66599 and GR/R 31409), and the US Office of Naval Research is gratefully acknowledged.

References

[1] J.H. Ahrens and U. Dieter, "An Alias Method for Sampling from the Normal Distribution," *Computing*, vol. 42, nos. 2-3, pp. 159-170, 1989.

[2] E. Antelo, T. Lang, and J.D. Bruguera, "Very-High Radix CORDIC Vectoring with Scalings and Selection by Rounding," *Proc. IEEE Symp. Computer Arithmetic,* pp. 204-213, 1999.

[3] D.R. Barr and N.L. Sezak, "A Comparison of Multivariate Normal Generators," *Comm. ACM,* vol. 15, no. 12, pp. 1048-1049, 1972.

[4] A. Brace, D. Gatarek, and M. Musiela, "The Market Model of Interest Rate Dynamics," *Math. Finance,* vol. 7, no. 2, pp. 127-155, 1997.

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

[6] 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, 2003.

[7] G.E.P. Box and M.E. Muller, "A Note on the Generation of Random Normal Deviates," *Annals Math. and Statistics,* vol. 29, pp. 610-611, 1958.

[8] J. Cao, B.W.Y. We, and J. Cheng, "High-Performance Architectures for Elementary Function Generation," *Proc. 15th IEEE Symp. Computer Arithmetic,* 2001.

[9] A. Cantoni, "Optimal Curve Fitting with Piecewise Linear Functions," *IEEE Trans. Computers,* vol. 20, no. 1, pp. 59-67, 1971.

[10] Celoxica Ltd., *Handel-C Language Reference Manual,* version 3.1, document number RM-1003-3.0, 2002.

[11] P.P. Chu and R.E. Jones, "Design Techniques of FPGA Based Random Number Generator," *Proc. Military and Aerospace Applications of Programming Devices and Techniques Conf.,* 1999.

[12] M. Combet, H. Van Zonneveld, and L. Verbeek, "Computation of the Base Two Logarithm of Binary Numbers," *IEEE Trans. Electronic Computers,* vol. 14, no. 6, pp. 863-867, 1965.

[13] R.B. D'Agostino and M.A Stephens, *Goodness-of-Fit Techniques.* Marcel Dekker Inc., 1986.

[14] D. Derrien and E. Boutillon, "Quality Measurement of a Colored Gaussian Noise Generator Hardware Implementation Based on Statistical Properties," *Proc. IEEE Int'l Symp. Signal Processing and Information Technology,* 2002.

[15] J.J. Eggers, J.K. Su, and B. Girod, "Robustness of a Blind Image Watermarking Scheme," *Proc. IEEE Int'l Conf. Image Processing,* vol. 3, pp. 17-20, 2000.

[16] D. Das and D.W. Matula, "Faithful Bipartite Rom Reciprocal Tables," *Proc. 12th IEEE Symp. Computer Arithmetic,* pp. 17-28, 1995.

[17] R.G. Gallager, "Low-Density Parity-Check Codes," *IEEE Trans. Information Theory,* vol. 8, pp. 21-28, 1962.

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

[19] W. Hörmann and J. Leydold, "Continuous Random Variate Generation by Fast Numerical Inversion," *ACM Trans. Modeling and Computer Simulation,* vol. 13, no. 4, pp. 347-362, 2003.

[20] V.K. Jain, S.A. Wadecar, and L. Lin, "A Universal Nonlinear Component and Its Application to WSI," *IEEE Trans. Components, Hybrids, and Manufacturing Technology,* vol. 16, no. 7, pp. 656-664, 1993.

[21] C. Jones, E. Vallés, M. Smith, J. Villasenor, "Approximate-Min* Constraint Node Updating for LDPC Code Decoding," *Proc. IEEE Military Comm. Conf. (MILCOM),* 2003.

[22] B. Jung, H. Lenhof, P. Müller, and C. Rüb, "Langevin Dynamics Simulations of Macromolecules on Parallel Computers," *Macromolecular Theory Simululation,* pp. 507-521, 1997.

[23] D.E. Knuth, "Seminumerical Algorithms" *The Art of Computer Programming,* volume 2, third ed., Addison-Wesley, 1997.

[24] R.E. Ladner and M.J. Fischer, "Parallel Prefix Computation," *J. ACM,* vol. 27, no. 4, pp. 831-838, 1980.

[25] D. Lee, W. Luk, J. Villasenor, and P.Y.K. Cheung, "A Hardware Gaussian Noise Generator for Channel Code Evaluation," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* pp. 69-78, 2003.

[26] D. Lee, W. Luk, J. Villasenor, and P.Y.K. Cheung, "Hardware Function Evaluation Using Non-Linear Segments," *Proc. Field-Programmable Logic and Applications,* pp. 796-807, 2003.

[27] D. Lee, W. Luk, J. Villasenor, and P.Y.K. Cheung, "Hierarchical Segmentation Schemes for Function Evaluation," *Proc. IEEE Int'l Conf. on Field-Programmable Technology,* pp. 92-99, 2003.

[28] D. Lee, W. Luk, C. Wang, C. Jones, M. Smith, and J. Villasenor, "A Flexible Hardware Encoder for Low-Density Parity-Check Codes," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* 2004.

[29] J.L. Leva, "A Fast Normal Random Number Generator," *ACM Trans. Math. Software,* vol. 18, no. 4, pp. 449-453, 1992.

[30] B. Levine, R.R. Taylor, and H. Schmit, "Implementation of Near Shannon Limit Error-Correcting Codes Using Reconfigurable Hardware," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* pp. 217-226, 2000.

[31] J. Leydold, "Automatic Sampling with the Ratio-of-Uniforms Method," *ACM Trans. Math. Software,* vol. 26, no. 1, pp. 78-98, 2000.

[32] D.J.C. MacKay, "Good Error-Correcting Codes Based on Very Sparse Matrices," *IEEE Trans. Information Theory,* Mar. 1999.

[33] G. Marsaglia, M.D. MacLaren, and T.A. Bray, "A Fast Procedure for Generating Normal Random Variables," *Comm. ACM,* vol. 7, no. 1, pp. 4-10, 1964.

[34] G. Marsaglia, *Diehard: A Battery of Tests of Randomness,* http://stat.fsu.edu/~geo/diehard.html, 1997.

[35] G. Marsaglia and W.W. Tsang, "The Monty Python Method for Generating Random Variables," *ACM Trans. Math. Software,* vol. 24, no. 3, pp. 341-350, 1998.

[36] G. Marsaglia and W.W. Tsang, "The Ziggurat Method for Generating Random Variables," *J. Statistical Software,* vol. 5, no. 8, pp. 1-7, 2000.

[37] O. Mencer and W. Luk, "Parameterized High Throughput Function Evaluation for FPGAs," *J. VLSI Signal Processing Systems,* vol. 36, no. 1, pp. 17-25, 2004.

[38] A. Miller and M. Gulotta, "PN Generators Using the SRL Macro," *Xilinx Application Note XAPP211,* v1. 1, 2001.

[39] M.E. Muller, "A Comparison of Methods for Generating Normal Deviates on Digital Computers," *J. ACM,* vol. 6, no. 3, pp. 376-383, 1959.

[40] J.N. Mitchell Jr., "Computer Multiplication and Division Using Binary Logarithms," *IRE Trans. Electronic Computers,* vol. 11, pp. 512-517, 1962.

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

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

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

[44] A.S. Noetzel, "An Interpolating Memory Unit for Function Evaluation: Analysis and Design," *IEEE Trans. Computers,* vol. 38, pp. 377-384, 1989.

[45] 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.

[46] 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. Computer Arithmetic,* 2001.

[47] T. Richardson and R. Urbanke, "Efficient Encoding of Low-Density Parity-Check Codes," *IEEE Trans. Information Theory,* vol. 47, pp. 638-656, 2001.

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

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

[50] 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.

[51] J. Vedral and J. Holub, "Oscilloscope Testing by Means of Stochastic Signal," *Measurement Science Rev.,* vol. 1, no. 1, 2001.

[52] Xilinx Inc., *Virtex-II User Guide v1.5,* 2002.

[53] J.E. Volder, "The CORDIC Trigonometric Computing Technique," *IEEE Trans. Electronic Computers,* vol. 8, no. 3, pp. 330-334, 1959.

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

[55] "Additive White Gaussian Noise (AWGN) Core v1.0," Xilinx Product Specification, 2002.

[56] K.W. Yip and T.S. Ng, "A Simulation Model for Nakagami-m Fading Channels, $m < 1$," *IEEE Trans. Comm.,* vol. 48, no. 2, pp. 214-221, 2000.

Dong-U Lee received the BEng degree in information systems engineering from Imperial College, London, in 2001. He is currently finishing the PhD degree in computing at the same university. He visited the Electrical Engineering Department, University of California, Los Angeles, in 2002 and 2003 as a visiting scholar, where he developed hardware designs for LDPC codes. His research interests include reconfigurable computing, computer arithmetic, channel coding, and video processing. He is a student member of the IEEE.

Wayne Luk received the MA, MSc, and PhD degrees in engineering and computer science from the University of Oxford, Oxford, United Kingdom. He is a member of the academic staff in the Department of Computing, Imperial College of Science, Technology and Medicine and leads the Custom Computing Group there. His research interests include theory and practice of customizing hardware and software for specific application domains, such as graphics and image processing, multimedia, and communications. Much of his current work involves high-level compilation techniques and tools for parallel computers and embedded systems, particularly those containing reconfigurable devices such as field-programmable gate arrays. He is a member of the IEEE.

John D. Villasenor received the BS degree in 1985 from the University of Virginia, the MS degree in 1986 from Stanford University, and the PhD degree in 1989 from Stanford, all in electrical engineering. From 1990 to 1992, he was with the Radar Science and Engineering Section of the Jet Propulsion Laboratory in Pasadena, California, where he developed methods for imaging the earth from space. He joined the Electrical Engineering Department at the Univesity of California, Los Angeles (UCLA), in 1992 and is currently a professor. He served as vice chair of the department from 1996 to 2002. At UCLA, his research efforts lie in communications, computing, imaging and video compression, and networking. He is a senior member of the IEEE.

Peter Y.K. Cheung graduated from Imperial College of Science and Technology, University of London in 1973 with first class honors and was awarded the IEE prize. After working at Hewlett Packard for a few years, he returned to Imperial College as a lecturer in 1980. He runs an active research group in digital design, attracting support from many industrial partners. He was elected as one of the first Imperial College Teaching Fellows in 1994 in recognition of his innovation in teaching. He is currently a professor of digital systems and deputy head of the Department of Electrical & Electronic Engineering at Imperial College. His research interests include VLSI architectures for signal processing, asynchronous systems, reconfigurable computing using FPGA, and architectural synthesis. He is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.