A Hardware Gaussian Noise Generator for Channel Code Evaluation

Dong-U Lee and Wayne Luk
Department of Computing
Imperial College
London
United Kingdom
{dong.lee, wl}@ic.ac.uk

John Villasenor
Electrical Engineering Department
University of California
Los Angeles
USA
villa@icsl.ucla.edu

Peter Y.K. Cheung
Department of EEE
Imperial College
London
United Kingdom
p.cheung@ic.ac.uk

Abstract

Hardware simulation of channel codes 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 non-uniform 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 extremely high fidelity to the modelled functions. 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 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 40 times faster than a 2.13GHz PC; another implementation on a Xilinx Spartan-IIE XC2S300E-7 FPGA at 62MHz is capable of a 20 times speedup. The performance can be improved by exploiting parallelism: an XC2V4000-6 FPGA with three parallel instances of the noise generator at 126MHz can run 100 times faster than a 2.13GHz PC. We illustrate the deterioration of clock speed with the increase in the number of instances.

1 Introduction

Numerical methods for Gaussian random number generation have a long history in mathematics and communications. As described in [1] 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. 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 [2]. 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 [3] 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 is simple, occupying approximately 10% of the resources on a Xilinx Virtex-II XC2V4000-6 device [4], 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 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 describes technology-specific implementation of the hardware architecture. Section 5 discusses evaluation and results, and Section 6 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 pseudo random 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 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.

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 [5]: the Ziggurrat method, the polar method, the use of the central limit theorem, and the Box-Muller method. The Ziggurrat method is not considered here 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 digital hardware Gaussian noise generators. The most relevant publications are probably [6], [7] and [8], 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.

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 will be presented in Section 4.

The Box-Muller method [9] 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

$$f(u_1) = \sqrt{-\ln(u_1)} \quad (1)$$
$$g_1(u_2) = \sqrt{2}\,\sin(2\pi\, u_2) \quad (2)$$
$$g_2(u_2) = \sqrt{2}\,\cos(2\pi\, u_2) \quad (3)$$

the products

$$x_1 = f(u_1)\, g_1(u_2) \quad (4)$$
$$x_2 = f(u_1)\, g_2(u_2) \quad (5)$$

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 [6]; 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.

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

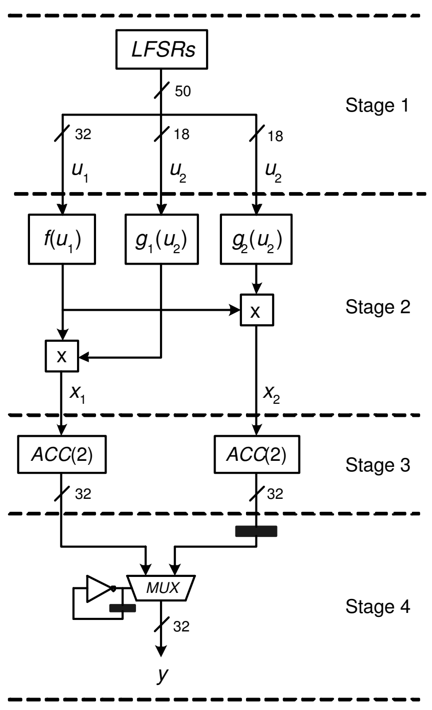

Figure 1. Gaussian noise generator architecture.

be accomplished using well-known techniques based on Linear Feedback Shift Registers (LFSRs) [10]. 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

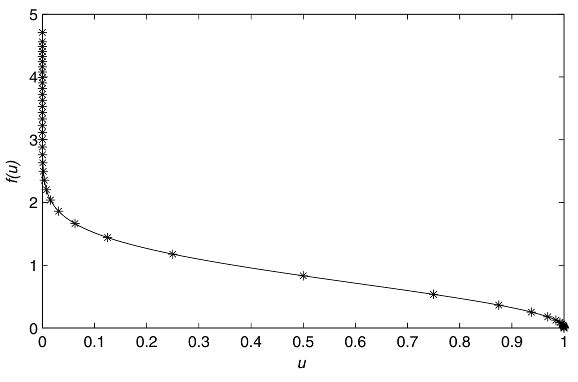

Figure 2. The f function. The asterisks indicate the boundaries of the linear approximations.

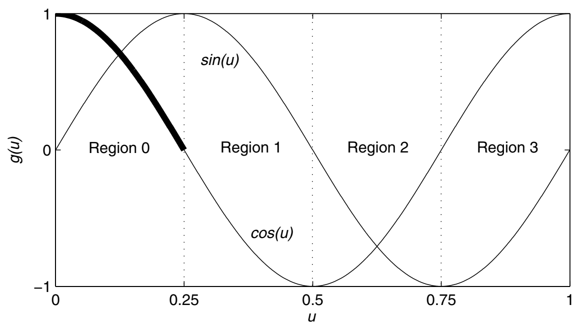

Figure 3. The g functions. Only the thick line is approximated; see Figure 4. The most significant 2 bits of u_2 are used to choose which of the four regions to use; the remaining bits select a location within Region 0.

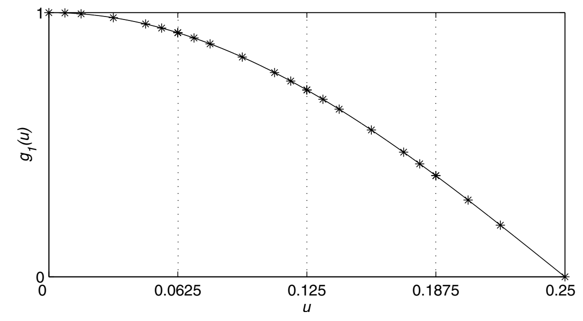

Figure 4. Approximation for the g functions which corresponds to the thick line in Figure 3. The asterisks indicate the boundaries of the linear approximations.

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 non-uniform piecewise linear approximation. The best-fit straight line, in a least squares 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 [12]. 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.

We first consider the f function (Figure 2). The greatest non-linearities of this function occur in the regions close to zero and one. To be consistent with the change in linearity, we use line segment boundaries at locations 2^{n-32} for $0 < u \leq 0.5$, and $1 - 2^{-n}$ for $0.5 < u \leq 1$, where $0 \leq n < 32$. The resulting boundaries are shown by the asterisks in Figure 2. A cascade of AND gates and OR gates can be used to produce the address of the appropriate coefficients. Since the largest gradient (for the steepest linear segment given the above partitioning) is of the order of 10^8, 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 variable precision for both the gradient and the y-intercept. Hence for each approximation four coefficients are stored: the gradient, the gradient scaling factor, the y-intercept, and the y-intercept scaling factor.

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 3. 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. 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 4.

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 (2) and (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 [13] and oscilloscope testing [14]. Fourth, for applications requiring a large dynamic range, floating-point arithmetic can be used for the components in our architecture.

4 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 lookup 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 [11]. 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 applica-

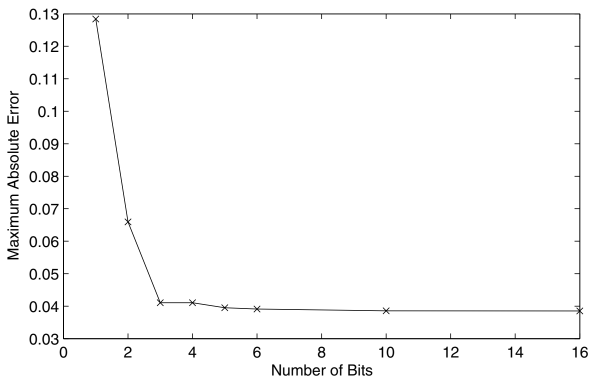

Figure 5. Variation of function approximation error with number of bits for the gradient of the f function.

tion 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 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 absolute error with small number of bits. Figure 5 shows how the 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 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 lookup table has a size of 3504 bits for the function evaluator.

Several FPGA implementations have been developed, using the Handel-C hardware compiler from

Table 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

Celoxica [15]. 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.

5 Evaluation and Results

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

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 [1].

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

$$\chi^2 = \sum_{i=1}^{k} \frac{(Y_i - np_i)^2}{np_i} \tag{6}$$

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

$$D = \max_{-\infty < x < \infty} |\, Y(x) - F(x) \,| \tag{7}$$

The χ^2 and K-S statistics are used to compute the p-values [16] 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 6, 7, and 8 illustrate the effect on the PDF of different implementation choices. Figure 6 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 7. 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 8 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 [17], we generate scatter plots showing pairs y_i and y_{i+1}. An example based on 10000 Gaussian variables is shown in Figure 9, 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

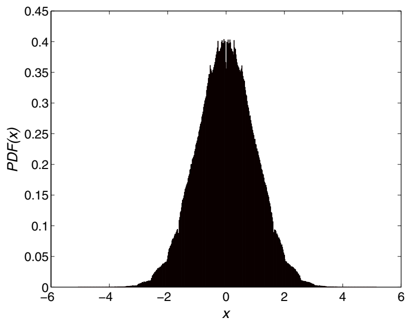

Figure 6. PDF of the generated noise with 17 approximations for f, 6 for g.

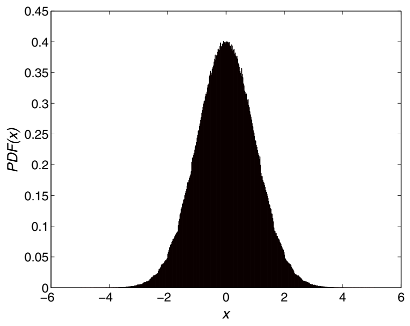

Figure 7. PDF of the generated noise with 59 approximations for f, 21 for g.

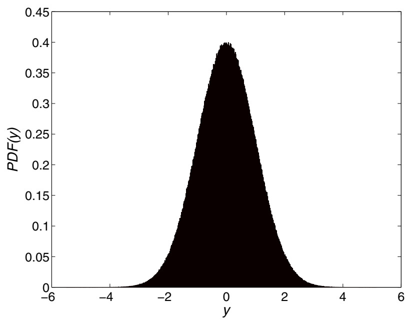

Figure 8. PDF of the generated noise with 59 approximations for f, 21 for g with two accumulated samples.

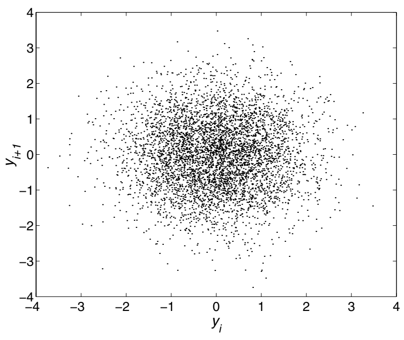

Figure 9. Scatter plot of two successive accumulative noise samples for a population of 10000.

as 10^{-9}.

Our hardware implementations, described in Section 4, have been compared to several software implementations based on the polar method. The results are shown in Table 2. It can be seen that our hardware designs are faster than software implementations by 40–100 times, depending on the device used and the resource utilization.

Figure 10 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 increase in clock fan-out and loading.

6 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 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. Sta-

Table 2. Performance comparison: time for producing 133 million samples per second. 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.

platform	time (sec)
XC2V4000-6 FPGA, 126MHz, 30% usage	0.4
XC2V4000-6 FPGA, 133MHz, 10% usage	1
XC2S300E-7 FPGA, 62MHz, 90% usage	2
AMD Athlon XP PC, 2.13GHz	40
AMD Athlon PC, 1.4GHz	45
Intel Pentium 4 PC, 2.0GHz	56

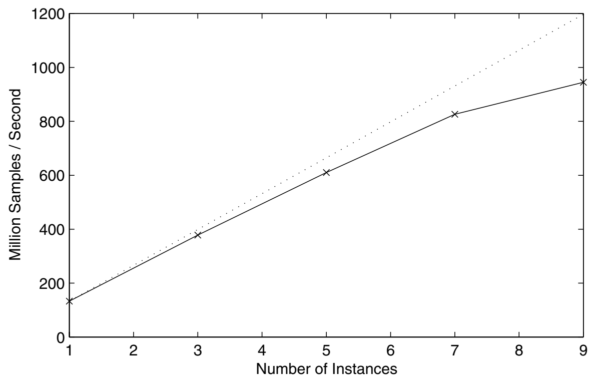

Figure 10. Variation of output rate against the number of noise generator instances. The dotted line shows the linear relationship between the output rate and the number of instances, if the clock speed does not deteriorate with the increasing number of instances.

tistical 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 [18] channels, and automating the design of the circuits for piecewise linear approximation to speed up the production of a wide variety of hardware noise generators.

Acknowledgment

The authors thank Altaf Abdul Gaffar, Jun Jiang, Shay Ping Seng and Tao Tian 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.

References

[1] D.E. Knuth, "Seminumerical algorithms", *The Art of Computer Programming*, Volume 2, Third Edition, Addison-Wesley, 1997.

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

[3] 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.

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

[5] 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.

[6] J.L. Danger et al., "Efficient FPGA implementation of Gaussian noise generator for communication channel emulation", *Proc. 7th IEEE Int. Conf. on Elect., Circ. and Syst. (ICECS2́K)*, 2000.

[7] A. Ghazel et al., “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.

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

[9] 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.

[10] 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.

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

[12] O. Mencer et al., “Parameterized function evaluation for FPGAs”, *Field-Programmable Logic and Applications*, LNCS 2147, pp. 544–554, 2001.

[13] 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.

[14] J. Vedral and J. Holub, “Oscilloscope testing by means of stochastic signal”, *Measurement Science Review*, Vol. 1, No. 1, 2001.

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

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

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

[18] 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.