A Hardware Gaussian Noise Generator Using the Wallace Method

Dong-U Lee, *Member, IEEE*, Wayne Luk, *Member, IEEE*, John D. Villasenor, *Senior Member, IEEE*, Guanglie Zhang, and Philip H. W. Leong, *Senior Member, IEEE*

***Abstract*—We describe a hardware Gaussian noise generator based on the Wallace method used for a hardware simulation system. Our noise generator accurately models a true Gaussian probability density function even at high σ values. We evaluate its properties 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. Our design is implemented on a Xilinx Virtex-II XC2V4000-6 field-programmable gate array (FPGA) at 155 MHz; it takes up 3% of the device and produces 155 million samples per second, which is three times faster than a 2.6-GHz Pentium-IV PC. Another implementation on a Xilinx Spartan-III XC3S200E-5 FPGA at 106 MHz is two times faster than the software version. Further improvement in performance can be obtained by concurrent execution: 20 parallel instances of the noise generator on an XC2V4000-6 FPGA at 115 MHz can run 51 times faster than software on a 2.6-GHz Pentium-IV PC.**

***Index Terms*—Channel coding, communication channels, field-programmable gate arrays (FPGAs), Gaussian noise, high-performance, Monte Carlo methods, reconfigurable-computing, technology-mapping.**

I. Introduction

THE AVAILABILITY of high quality Gaussian random numbers is critical to many simulation, graphics and Monte Carlo applications. Currently, the majority of such simulations are performed using systems based on microprocessors, digital signal processors, or other software-programmable devices. In these systems, the trigonometric, exponential, and other functions involved in many of the methods for obtaining Gaussian random variables can be performed using software libraries [1]. As a result, not much research has been reported concerning efficient hardware methods for implementation of Gaussian noise generators. However, well-optimized hardware implementations can often operate one or more orders of magnitude faster than similarly optimized software implementations. Recent advances in field-programmable gate array (FPGA) technology have substantially improved performance and cost effectiveness of hardware implementations, and they provide a strong motivation for us to reexamine the issue of Gaussian noise generation in hardware.

The work described here is originally motivated by ongoing advances in communications relating to channel codes [2], and in particular by the development of new generations of channel codes that operate on very long (thousands to tens of thousands of bits each) blocks of data. For these codes, it is often desirable to perform simulations of extremely large numbers of blocks in order to assess the bit-error rate (BER) performance at rates as low as 10^{-12}.

There are many other applications in which very large simulations using Gaussian noise are valuable as well. These include financial modeling [3], simulation of economic systems [4], and molecular dynamics simulations [5]. For all of these applications, hardware-based simulation offers the potential to speed up simulation by several orders of magnitude, but is feasible only if suitably fast and high-quality noise generators can be implemented. In principle, one could generate the noise samples on a PC and transfer them to the hardware device performing the simulation. However in such approaches, generation of the noise samples is often the performance bottleneck. Hence, it is desirable to have a hardware noise generator. On-chip noise generation is significantly faster and does not suffer from transfer overheads.

In addition, since deviation from an ideal Gaussian probability density function (pdf) can degrade simulation results, very large simulations have stringent requirements on the quality of the pdf in the tails. Samples that lie at large multiples of σ (standard deviation) away from the mean are rare, but they are also exactly the noise realizations that are most likely to induce events of high interest in understanding the behavior of the overall system. To accurately obtain good characteristics in the tails requires the combination of: 1) an underlying method that creates high σ values with the proper frequency and 2) a hardware implementation of the method that preserves the requisite precision at all stages to ensure that high σ behavior is not compromised.

The principal contribution of this paper is a hardware Gaussian noise generator based on the Wallace method [6] that offers quality suitable for simulations involving very large numbers of noise samples. The noise generator occupies approximately 3% of the resources of a Xilinx Virtex-II

Manuscript received August 20, 2004; revised March 23, 2005. This work was supported in part by Xilinx Inc., by the U.K. Engineering and Physical Sciences Research Council under Grants GR/N 66599 and GR/R 31409, by the U.S. Office of Naval Research, and by the Research Grants Council of the Hong Kong Special Administrative Region, China, under Project CUHK4333/02E.

D. Lee was with the Department of Computing, Imperial College London, London SW7 2AZ, U.K. He is now with the Electrical Engineering Department, University of California, Los Angeles, CA 90024 USA (e-mail: dongu@icsl.ucla.edu).

W. Luk is with the Department of Computing, Imperial College London, London 2SW 2BT, U.K. (e-mail: w.luk@imperial.ac.uk).

J. D. Villasenor is with the Electrical Engineering Department, University of California, Los Angeles, CA 90024 USA (e-mail: villa@icsl.ucla.edu).

G. Zhang and P. H. W. Leong are with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Shatin, Hong Kong (e-mail: glzhang@cse.cuhk.edu.hk, phwl@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TVLSI.2005.853615

XC2V4000-6 FPGA device, while producing over 155 million samples per second. The key contributions of our work include:

- a hardware architecture for the Wallace method;
- 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 (A-D) test, as well as through application to a large communications simulation involving low-density parity-check (LDPC) channel codes [7].

The rest of this paper is organized as follows. Section II covers background material and previous work. Section III provides an overview of the Wallace method. Section IV describes our Wallace architecture, and discusses how each of its steps can be handled in a hardware implementation. Section V describes technology-specific implementation of the hardware architecture. Section VI discusses evaluation and results, and Section VII offers conclusions and future work.

II. Background

Most methods for generating random Gaussian variables are based on transformations or operations on uniform random variables. Widely used methods include various rejection-acceptance methods [8]–[10], the use of the central limit theorem [11], the inversion method [12] and the Box–Muller method [13]. While there are many software implementations of these methods, there is little previous work on digital hardware Gaussian noise generation.

The rejection-acceptance methods are popular in software approaches. However, they contain conditional loops such that the output rates are not constant, which is undesirable in a hardware simulation environment. While in principle the central limit theorem can be used to produce Gaussian samples if a suitable number of samples are involved, in practice an impractically large number of samples would be required to achieve an accurate representation of the ideal Gaussian pdf.

The Box–Muller method, either alone or in combination with the central limit theorem, has been the focus of most efforts in hardware implementation. For example, Boutillon *et al.* [14] present a hardware Gaussian noise generator based on the Box–Muller algorithm in conjunction with the central limit theorem. Their design occupies 437 logic cells on an Altera Flex 10K100EQC240-1 FPGA, and outputs 24.5 million noise samples per second at a clock speed of 98 MHz. Recently, the "Additive White Gaussian Noise (AWGN) Core 1.0" [15] has been released by Xilinx, which is based on the Boutillion *et al.* architecture.

Chen *et al.* [16] use a cummulative distribution function (cdf) conversion table to transform uniform random variables to Gaussian random variables. They have implemented the Gaussian noise generator as part of a readback-signal generator on a Xilinx Virtex-E XCV1000E FPGA at 70 MHz. The method they employ is basically the inversion method [12] implemented with a look-up table. However, the number of table entries are insufficient. To produce high quality noise samples with their direct table look-up approach, one would lead to an impractically large table.

Fan *et al.* [17] present a hardware Gaussian noise generator based on the polar method [11] in conjunction with the central limit theorem. Their design is implemented on an Altera Mercury EP1M120F484C7 FPGA; it takes up 336 logic elements and has a clock speed of 73 MHz generating a sample every clock. The polar method is a variant of the Box–Muller method and is a class of the rejection-acceptance methods, hence the output rate is not constant. In order to overcome this problem, they employ a first-in first-out (FIFO) buffer with the read speed set to half of the write speed.

The drawback of the hardware designs mentioned above are revealed by statistical tests applied to evaluate the noise samples produced. The authors carry out relative error comparisons between the ideal Gaussian pdf and the obtained pdf for a small number of samples and limited maximum σ value. Such tests are not enough to ensure the quality of the noise samples; one should thoroughly apply well known goodness-of-fit tests such as the chi-square test and the A-D test over large numbers of samples. Designs which fail such statistical tests are inadequate for high quality hardware communications simulations such as LDPC codes. This issue is discussed in more detail in Section VI.

In [18], we present a hardware Gaussian noise generator based on the Box–Muller method and central limit theorem. The idea is similar to [14], but we employ more sophisticated approximation techniques for the mathematical functions of the Box–Muller method, resulting in significantly more statistically accurate noise samples. The design occupies 2514 slices, two block RAMs and eight MULT18X18s (18×18 embedded multiplier blocks) on a Xilinx Virtex-II XC2V4000-6 FPGA. It operates at 133 MHz generating a noise sample every clock and passes the statistical tests widely used for testing normality discussed in Section VI.

All of the methods described above produce normal variables by performing operations on uniform variables. In contrast, Wallace proposes an algorithm using an evolving pool of normal variables to generate additional normal variables [6]. The approach draws its inspiration from uniform random number generators that generate one or more new uniform variables from a set of previously generated uniform variables. Given a set of normally distributed random variables, a new set of normally distributed random variables can be generated by applying a linear transformation. Brent [19] has implemented a fast vectorized Gaussian random number generator using the Wallace method on the Fujitsu VP2200 and VPP300 vector processors. In [20], Brent and Rüb outline possible problems associated with the Wallace method and discuss ways of avoiding them.

III. Wallace Method

Wallace proposes a fast algorithm for generating normally distributed pseudorandom numbers which generates the target distributions directly using their maximal-entropy properties [6]. This algorithm is particularly suitable for high throughput

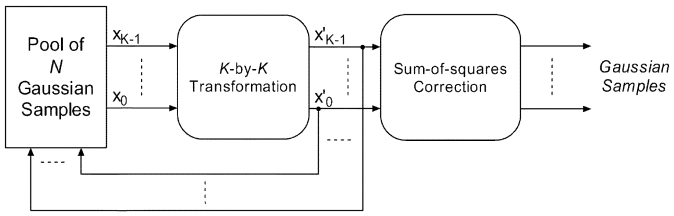


Fig. 1. Overview of the Wallace method.

hardware implementation since no transcendental functions such as $\sqrt{x}$, $\log(x)$ or $\sin(x)$ are required.

An overview of the Wallace method is described in Fig. 1. It takes a pool of $N = KL$ normally distributed random numbers from the normal distribution. These values are normalized so that their average squared value is one. In L transformation steps, K numbers are treated as a vector X, and transformed into K new numbers from the components of the K vector $X_1 = AX$ where A is an orthogonal matrix. If the original K values are normally distributed, then so are the K new values. Furthermore, this transformation preserves the sum of squares.

The process of generating a new pool of normally distributed random numbers is called a "pass." After a pass, a pool of new Gaussian random numbers is formed. As there are KL variables in the data pool, L transformation steps are performed during each pass. A K-vector X is multiplied with the orthogonal matrix A in performing a transformation step.

As stated by Wallace, it is desirable that any value in the pool should eventually contribute to every value in the pools formed after several passes. In Wallace's original method, the old pool is treated as an L-by-K array stored in row-major order, and the new pass is treated as an L-by-K array stored in column major order. Hence, each pass effectively transposes the values in the pool. If L is odd, the transposition is sufficient to ensure eventual mixing of the values. However if L is even, transposition alone is not sufficient. We describe in Section IV how we overcome this problem to reduce correlation even further.

The initial values in the pool are normalized so that their average squared value is one. Because A is orthogonal, the subsequent passes do not alter the sum of the squares. This would be a defect, since if $x_1, \ldots, x_N$ are independent samples from the normal distribution, we would expect $\sum_{i=1}^{i=N} x_i^2$ to have a chi-squared distribution χ_N^2. In order to overcome this defect, a variate from the previous pool is used to approximate a random sample S from the χ_N^2 distribution. A scaling factor is introduced to ensure that the sum of the squares of the values in the pool is S, the random sample.

One concern of the Wallace method is the issue of correlations given the use of previous outputs to generate new outputs. This can be problematic in the case of realizations with large absolute values lying in the tails of the Gaussian, since each value contributes K values in the subsequent block, K^2 values in the next block, and so on with diminishing influence. With proper choice of parameters such as pool size and transformation size, the effect of these correlations can be minimized for a given set of requirements with respect to number of noise samples, tail accuracy, and noise quality.

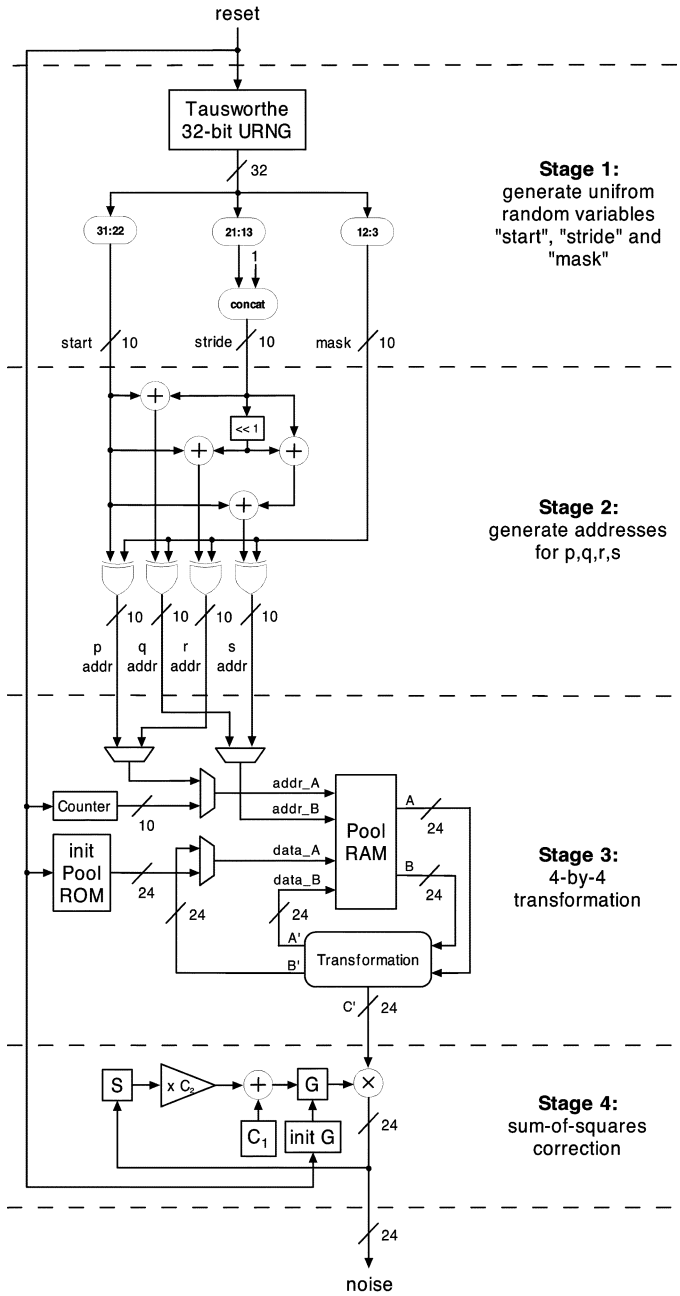


Fig. 2. Overview of our Gaussian noise generator architecture based on the Wallace method. The triangle in Stage 4 is a constant coefficient multiplier.

IV. Architecture

This section provides an overview of the hardware design for the Wallace method, which involves a four-stage hardware architecture shown in Fig. 2. The implementation of this architecture in FPGA technology will be presented in Section V. In Fig. 2, the select signals for the multiplexors and the clock enable signals for the registers are omitted for simplicity.

In our design illustrated in Fig. 2, we choose $K = 4$ and $L = 256$ resulting in a pool size N of 1024. Although one can choose any K and L, in this work, we follow Wallace's original description. On-chip, true dual read/write port synchronous RAM is used to implement the pool. The dual-port RAM allows two values to be read and written simultaneously, improving the memory bandwidth.

As all the variables from the pool are used to generate the new pseudo random numbers, the indexes should cover all the numbers in the pool and at the same time reduce the correlations between them. The addresses which index the pool are started from a random origin "start," stepped by a random odd "stride" and XOR is performed with a random "mask." The combination of these three operations is critical to achieve good mixing between the Gaussian samples in the pool. Our tests show that if one of them is not performed, it causes degradation in the overall Gaussian noise quality.

In order to achieve better mixing of the Gaussian random number generator, more pass types can be used during a pass by introducing different orthogonal matrices. As in Wallace's original implementation, two orthogonal matrices A_0 and A_1 are chosen for our design

$$A_0 = \frac{1}{2}\begin{pmatrix} 1 & -1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix}$$

$$A_1 = \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & 1 \\ -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{pmatrix}.$$

During a pass, A_0 is used for first half and A_1 is used for the second half of the pass. As the elements of the matrices A_0 and A_1 are only 1 or -1, only simple integer addition and shift operations are required. The Gaussian random variables in the pool are held as 24 bit two's complement integers. For the given set of four values p, q, r, s to be transformed, and with our choice of A_0 and A_1, the new values p', q', r', s' can be calculated from the old ones as follows:

$$p' = p - t;\ q' = t - q;\ r' = t - r;\ s' = t - s \quad (1)$$

$$p' = t - p;\ q' = q - t;\ r' = r - t;\ s' = s - t \quad (2)$$

where $t = (1/2)(p + q + r + s)$. Note that the operations above are performing the actual transformations, hence there is no need to store A_0 and A_1.

A. First Stage

This stage involves generation of the uniformly distributed realizations start, stride and mask. While traditional linear feedback shift registers (LFSRs) [21] are sufficient as a uniform random number generator (URNG), Tausworthe URNGs [22] offer better randomness with modest hardware cost. The Tausworthe URNG we employ follows the algorithm presented in [22]. It combines three LFSR based random number generators to obtain improved statistical properties, generates a 32-bit uniform random number per clock, and has a period of around 2^{88}. Since we use a pool size of 1024, 10 bits are needed for the three variables. Since stride needs to be odd at all times, we concatenate a one after the least significant bit. Hence, for each pass, altogether 29 bits are used for start, stride, and mask. The remaining three bits are left unused.

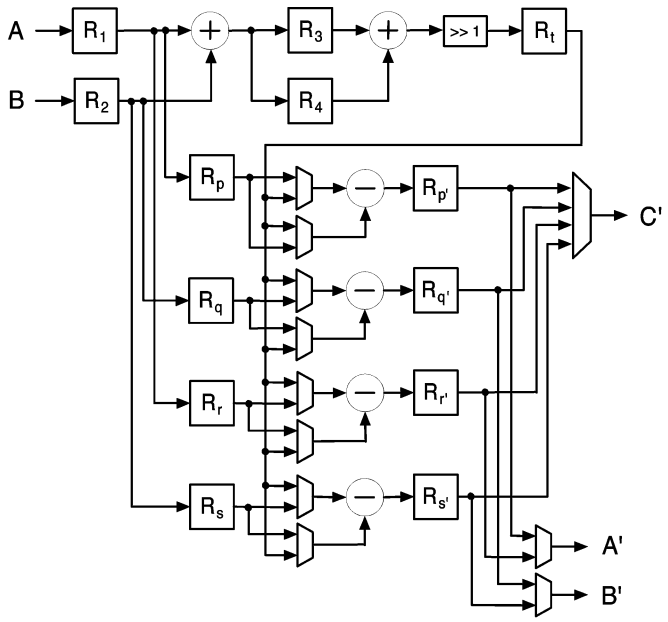


Fig. 3. Transformation circuit of Stage 3. The square boxes are registers. The select signals for the multiplexors and the clock enable signals for the registers are omitted for simplicity.

B. Second Stage

This stage generates the addresses for the four values p, q, r, s from start, stride, and mask. The four addresses are calculated as follows:

$$p\ \text{addr} = \text{start} \oplus \text{mask} \quad (3)$$

$$q\ \text{addr} = (\text{start} + \text{stride}) \oplus \text{mask} \quad (4)$$

$$r\ \text{addr} = (\text{start} + \text{stride} \times 2) \oplus \text{mask} \quad (5)$$

$$s\ \text{addr} = (\text{start} + \text{stride} \times 3) \oplus \text{mask}. \quad (6)$$

The multiplication by two is implemented simply by a left shift, and the multiplication by three is implemented by a left shift followed by an addition. This addressing scheme ensures that the correlations between variables are kept at a minimum.

C. Third Stage

This stage involves the most interesting challenge: efficiently performing the actual transformation. This stage contains the "Pool RAM" which holds the pool of 1024 Gaussian random variables. Dual-port RAM is used to implement the pool. Since each variable in the pool is 24 bits, the total size of the pool is $1024 \times 24 = 24\,576$ bits. The "init Pool ROM" and the counter are used to initialize the pool with the original pool contents when the reset signal is set. This ROM is single-ported and has the same size as the pool. The contents of this ROM are generated in software using the Box–Muller method, and the variables are normalized so that their sum of squares is equal to one.

Fig. 3 shows how we perform the transformation steps described in (1) and (2). The timing diagram of this circuit and the "Pool RAM" is illustrated in Fig. 4. We can see that the dual-port RAM is fully utilized. t is calculated in three steps

$$x = p + q \quad (7)$$

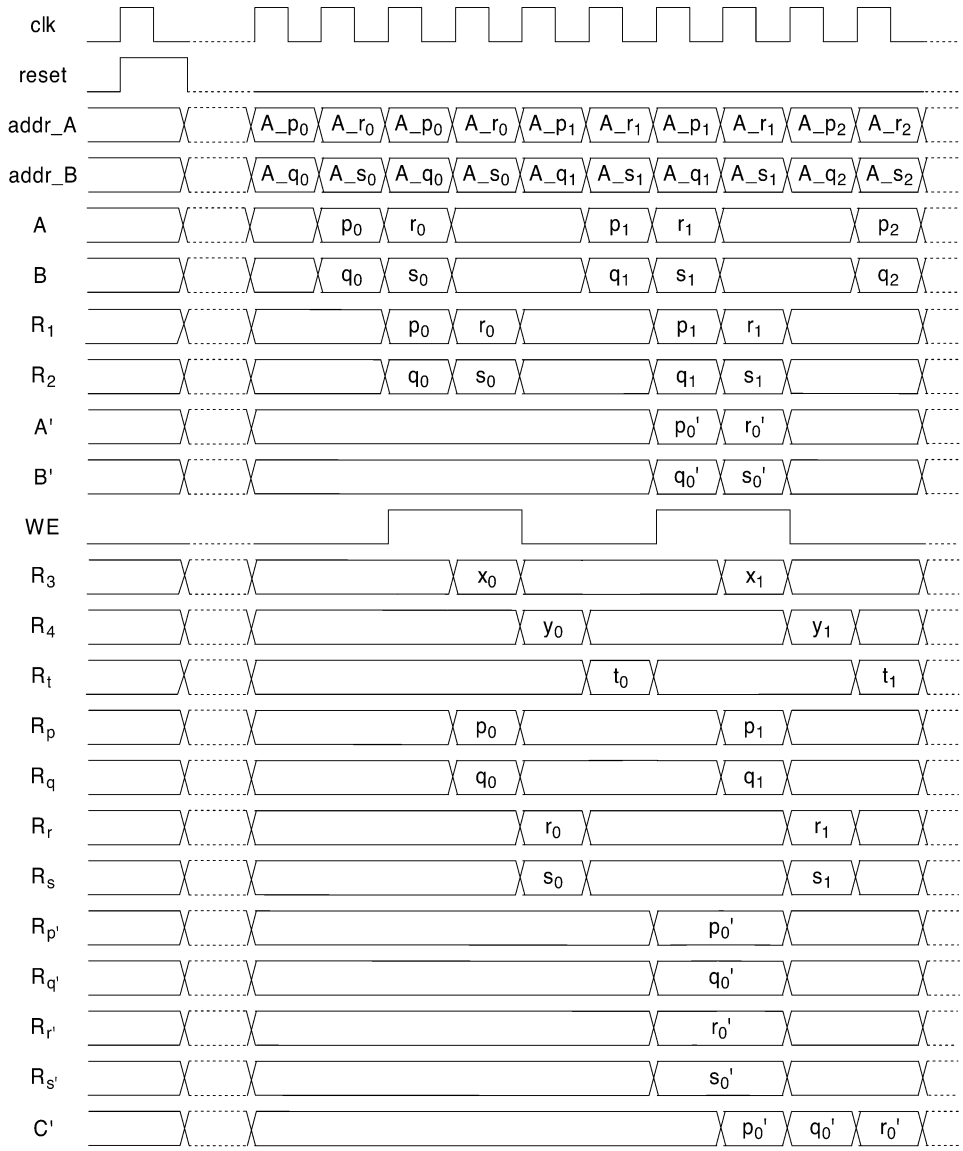


Fig. 4. Detailed timing diagram of the transformation circuit and the dual-port "Pool RAM." A_z indicates the address of the data z and WE is the write enable signal of the "Pool RAM." All ports and registers of the transformation circuit and ports of the dual-port RAM are shown. We observe that the dual-port RAM is fully utilized.

$$y = r + s \tag{8}$$

$$t = \frac{1}{2}(x + y). \tag{9}$$

In principle, we could share a single adder in conjunction with multiplexors to perform all the operations of the transformation circuit. However, high-speed adders are efficiently implemented on FPGAs by fast-carry chains. In fact, both a two-input 24-bit multiplexor and a 24-bit adder occupy 14 slices (user-configurable elements on the FPGA) in a Xilinx Virtex-II FPGA. In addition the use of multiplexors would increase the delay significantly. For these reasons, we decide to use separate adders/subtractors for each operation. For other devices such as Application-Specific Integrated Circuits (ASICs), it can be more efficient to adopt the former approach involving hardware sharing. The critical path of the entire Wallace design is from R_p to $R_{p'}$ which is just a multiplexor followed by a subtractor.

D. The Fourth Stage

This stage performs the sum of squares correction described in Section III. It follows the approach used by Wallace in the FastNorm implementations [23].

A random sample S with an approximate χ^2_N distribution can be obtained as

$$S = \frac{1}{2}(C + A \times x)^2 \tag{10}$$

where x has unit normal distribution, $A = 1 + 1/(8N)$ and $C = \sqrt{2N - A^2}$ for large N. Hence, S can be computed as

$$S = \sqrt{\frac{1}{2N}} \times A \times (B + x) \tag{11}$$

where $B = C/A$. We set $C_1 = A \times \sqrt{2N}$ and $C_2 = B$.

The noise sample C', generated from the transformation circuit of Stage 3, is multiplied by G to correct the sum of the squares and hence the final noise sample. G is obtained by

$$G = S \times C_2 + C_1. \tag{12}$$

Since C_1 and C_2 are constants, they are precalculated in software and stored as constants in the hardware design.

Before a pass, S is assigned with a variable from a previous pass, and G is updated. For the very first pass when the reset signal is set, G is initialized to $1/\sqrt{N/ts}$ where ts is the sum of squares of the initial pool. Note that we are using a pool size of $N = 1024$.

V. Implementation

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

The Tausworthe URNG in Stage 1 can be implemented in configurable hardware using a small amount of resources. Recent FPGAs have a large number of user-configurable elements: for instance, the Xilinx Virtex-II XC2V4000-6 device has 23 040 user-configurable elements known as slices. We use the primitive pentanomial $x^{88} + x^{87} + x^{17} + x^{16} + 1$ over $GF(2)$ with a random initial state. This takes up just 77 slices.

Xilinx Virtex-II devices have embedded memory elements and multipliers, which are known as block RAMs and MULT18X18s. Each block RAM can hold 18 kb of data and each MULT18X18 can implement a 18 bit $\times$ 18 bit multiplication. If the data or the multiplication are larger than 18 kb or 18 bit $\times$ 18 bit, the Xilinx tools will use multiple block RAMs and MULT18X18s to implement them. The Xilinx Virtex-II XC2V4000-6 device has 120 block RAMs and 120 MULT18X18s in total. The "init Pool ROM" is implemented using single-port block RAM, while the "Pool RAM" is implemented using dual-port block RAM. Since, we use a pool of 1024 and use 24 bits for each noise samples, the size of "init Pool ROM" and "Pool RAM" are both 24 576 bits, occupying two block RAMs each. The constant coefficient multiplier in Stage 4 uses two block RAMs to implement part of the multiplication. The 24 bit $\times$ 24 bit multiplier in Stage 4 occupies four MULT18X18s.

Several FPGA implementations have been developed, using Xilinx System Generator 6.3 [24]. All designs are heavily pipelined to maximize throughput. Synplicity Synplify Pro 7.7 is used for synthesis. For place-and-route, Xilinx ISE 6.3 is used with the maximum effort level and the clock constraints are carefully tuned to give the fastest clock frequency. We have mapped and tested the Wallace design onto a hardware platform with a Xilinx Virtex-II XC2V4000-6 FPGA. The design occupies 770 slices, six block RAMs, and four MULT18X18s, which takes up around 3% of the device. The pipelined design operates at 155 MHz, and hence our design produces 155 million Gaussian noise samples per second. The resource usage of each of the four stages is shown in Table I.

The latency of our design is 1138 clock cycles ($\approx$ 7 μs at 155 MHz); 1024 cycles are used to initialize the "Pool RAM" with the initial pool of Gaussian random samples. The other 114 cycles are needed to fill up the pipelines of the design. Although the latency is very large, it is not important since we only care about the throughput in a hardware-based simulation.

TABLE I
Resource Utilization for the Four Stages of the Noise Generator on a Xilinx Virtex-II XC2V4000-6 FPGA

stage	slices	block RAMs	MULT18X18s
1	77	-	-
2	121	-	-
3	342	4	-
4	230	2	4
total	770	6	4

TABLE II
Hardware Implementation Results of the Noise Generator Using Different Types of FPGA Resources on a Xilinx Virtex-II XC2V4000-6 FPGA

FPGA resources used	slices	block RAMs	MULT 18X18s	speed [MHz]
slices + block RAMs + MULT18X18s	770	6	4	155
slices + block RAMs	1095	6	-	151
slices + MULT18X18s	3548	-	4	119
slices	3890	-	-	112

From a hardware designer's point of view, it is interesting to explore the tradeoffs between using different types of hardware resources. For instance, a table can be implemented using block RAM or distributed RAM using slices. Table II shows our noise generator implemented using different FPGA resources. We observe that the design using slices only is more than four times the number of slices and has significantly lower clock speed than our original design. Also, the area and speed penalty of using slices to implement tables instead of block RAMs is especially high. Hence, in our opinion, dedicated FPGA resources such as block RAMs and MULT18X18s should be used wherever applicable.

We have also implemented our design on a low-cost Xilinx Spartan-III XC3S200E-5 FPGA, utilizing the slices, block RAMs, and MULT18X18s available on the chip. The design runs at 106 MHz and takes up the same amount of resources as the Virtex-II design above, which requires around half of the resources in the device.

VI. Evaluation and Results

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

To ensure the randomness of the uniform random numbers start, stride and mask, we have tested the Tausworthe URNG with the Diehard tests [25]. The Tausworthe URNG pass 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 A-D test [26]. 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. This test is essentially a comparison between an experimentally determined histogram and the ideal pdf. In contrast to the χ^2 test which deals

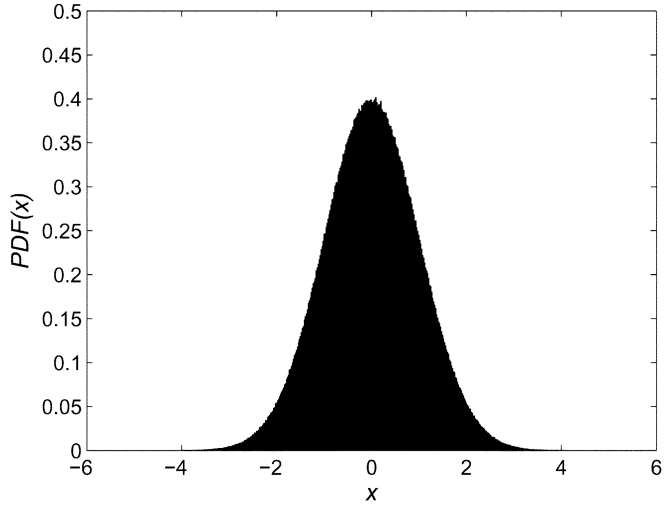


Fig. 5. The pdf of the generated noise from our design for a population of four million samples. The p-values of the χ^2_{99} and A-D tests are 0.7303 and 0.8763, respectively.

with quantized aspects of a design, the A-D test deals with continuous properties. It is modified from the Kolmogorov-Smirnov (K-S) test [11] to give 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.

The probability that the deviation of the observed from the expected is due to chance alone can be obtained from a p-value [26] based on the above tests. 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.

Our hardware Wallace 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_{99} and A-D tests are found to be 0.5385 and 0.7372 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_{99} and A-D tests are found to be 0.6839 and 0.7662, showing that the noise quality even in the high σ regions is high.

If (x, y) is a pair of random numbers with Gaussian distributions, then $u = e^{(x^2+y^2)/2}$ should be uniform over $[0, 1]$. Six million Gaussian variables, randomly picked from a population of 10^{10} samples generated from our design are transformed using this identity, resulting in three million uniform random variables. These uniform variables are tested with the Diehard tests [25] for uniformity. They pass all tests indicating that the transformed numbers are indeed uniformly distributed. Fig. 5 shows the pdf obtained from our Gaussian noise generator for a population of four million samples. The samples pass both the χ^2 and the A-D test resulting in a smooth bell-shaped Gaussian distribution.

TABLE III
COMPARISONS OF DIFFERENT HARDWARE GAUSSIAN NOISE GENERATORS IMPLEMENTED ON XILINX VIRTEX-II XC2V4000-6 FPGAS. ALL DESIGNS GENERATE A NOISE SAMPLE EVERY CLOCK

	Xilinx [24]	Lee [18]	this design
slices	653	2514	770
block RAMs	4	2	6
MULT18X18s	8	8	4
speed [MHz]	168	133	155
pass χ^2 test	no	yes	yes
pass A-D test	no	yes	yes

We compare our design with two other designs: "White Gaussian Noise Generator" block available in Xilinx System Generator 6.3 [24] and the design presented in [18]. The "White Gaussian Noise Generator" block is based on the "Additive White Gaussian Noise (AWGN) Core 1.0" from Xilinx [15]. The Xilinx core follows the architecture presented by Boutillon *et al.* in [14], which uses the Box–Muller method in conjunction with the central limit theorem. The block is slightly slower and larger than the core, since it is less optimized. The design in [18] is also based on the Box–Muller method and central limit theorem, but we employ more sophisticated approximation techniques for the mathematical functions in the Box–Muller method, resulting in significantly more statistically accurate noise samples. We test the noise samples generated from the Xilinx block with the χ^2_{99} and the A-D test. We find that the samples fail the tests after just 160 000 samples. We also tested the Boutillon *et al.* [14] design with the χ^2_{99} test, and found that it fails after just 20 000 samples. This is primary due to the limited resolution problem of their nonuniform direct look-up table approach. We suspect that the tables would have to be impractically large to generate high quality noise samples with this approach.

Table III compares the Xilinx block, our Box–Muller design in [18], and our Wallace design. We can see that the Xilinx block uses less resources and is slightly faster than our Wallace design, but as mentioned above the block fails the statistical tests after a small number of samples. Both of our Box–Muller and Wallace designs pass the statistical tests, even with very large numbers of samples. However, our Wallace design is around three times smaller and slightly faster.

Fig. 6 shows the variation of the χ^2_{99} value with sample size for the Xilinx block, and various Wallace implementations at different data path and noise sample bit-widths. The dotted horizontal line is the 0.05 confidence level (p-value), i.e., values below this line pass the χ^2_{99} test. We observe that the Xilinx block fails after a small number of samples. For the Wallace implementations, we observe that with increasing precision more samples are required to fail the χ^2_{99} test. This is due to a combination of two factors: 1) insufficient precision in the Wallace arithmetic leading to low quality noise samples and 2) because of finite precision of the noise samples, some bins of the χ^2_{99} test will be more biased than others, this effect is reduced with increasing noise precision.

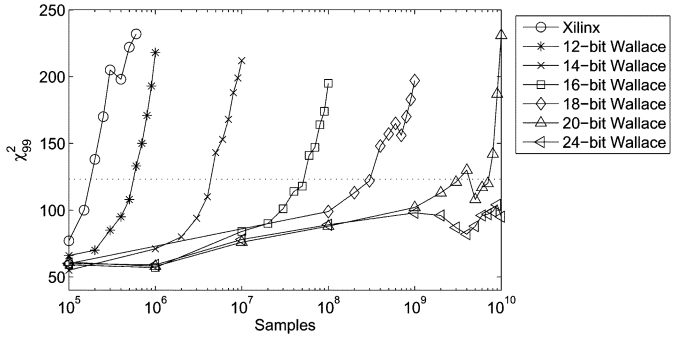


Fig. 6. Variation of the χ^2_{99} value with sample size for the Xilinx block and various Wallace implementations at different precisions.

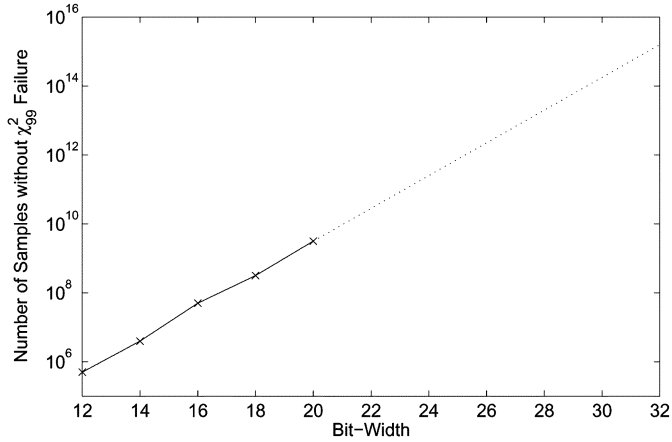


Fig. 7. Wallace bitwidth versus number of samples without χ^2_{99} failure. We see a linear behavior with the y axis in logarithmic scale.

In Fig. 7 we plot the Wallace bitwidth versus the number of samples without χ^2_{99} failure. With the y axis in logarithmic scale, we can see a linear behavior. The dotted line shows the trend at higher bit-widths. For instance, if 32 bit Wallace is used, up to around 10^{15} samples would pass the χ^2_{99} test. Increasing the Wallace bit-width would require more block RAMs for the pool, and larger multipliers for the sum-of-squares correction, and more slices for the larger adders and multiplexors.

Fig. 8 shows BER performance simulation results for a 972×1944 irregular LDPC code from [27]. Results show our Wallace noise generator, standard Box–Muller, and Box–Muller with the noise saturated at 4σ and 3σ. The Box–Muller implementations are written in software using double-precision floating point arithmetic. Two observations can be made from this figure. First, there are no distinguishable differences between our Wallace implementation and the software Box–Muller. Second, limiting the noise power to a certain σ multiple results in inaccurate simulation results, particularly in regions of lower BER. For instance at $E_b/N_o = 2.5$ dB, our Wallace gives a BER of $10^{-6.03}$, standard Box–Muller gives $10^{-6.01}$, whereas Box–Muller saturated at 4σ and 3σ give $10^{-6.16}$ and $10^{-6.44}$, respectively. Thus, the lack of noise of sufficient quality can lead to an overly optimistic (and incorrect) conclusions regarding the behavior of the code.

The performance of the noise generator can be improved by concurrent execution. We have experimented with placing multiple instances of our noise generator in an FPGA, and discovered that there is a reduction in clock speed due to increased routing congestion. We are able to fit up to 20 instances on the XC2V4000-6 FPGA running at 115 MHz, the number of block RAMs available on the device being the limit. Of course using a larger device such as the Virtex-4 XC4VFX140 FPGA, we are able to fit even more instances. Note that it is perfectly valid to use multiple instances of the noise generator, as long as the Tausworthe URNGs and pool RAMs are initialized with different random seeds and noise samples.

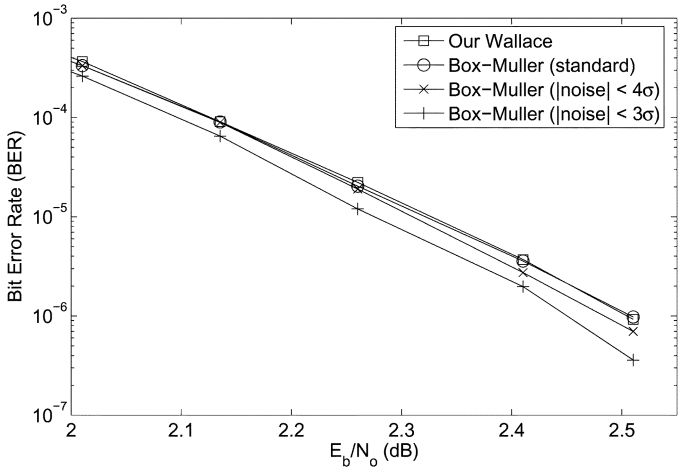


Fig. 8. BER performance simulation results for a 972×1944 irregular LDPC code. The Box–Muller implementations are written in software using double-precision floating point arithmetic.

TABLE IV
PERFORMANCE COMPARISON: TIME FOR PRODUCING ONE BILLION GAUSSIAN NOISE SAMPLES. THE XC2V4000-6 FPGA BELONGS TO THE XILINX VIRTEX-II FAMILY, WHILE THE XC3S200E-5 FPGA BELONGS TO THE XILINX SPARTAN-III FAMILY

platform	speed [MHz]	method	time [s]	ratio
XC2V4000-6 FPGA	115	20 inst	0.43	1
XC2V4000-6 FPGA	155	1 inst	6.5	15
XC3S200E-5 FPGA	106	1 inst	9.4	22
Intel Pentium-IV PC	2600	Wallace	22	51
Intel Pentium-IV PC	2600	Ziggurat	63	145
Intel Pentium-IV PC	2600	Polar	164	377
Intel Pentium-IV PC	2600	Box-Muller	265	609

Our hardware implementations have been compared to several software implementations based on the Wallace, Ziggurat [10], polar and Box–Muller method [11], which are known to be the fastest methods for generating Gaussian noise for instruction processors. For the Wallace and Ziggurat methods, FastNorm2[1] [23] and rnorrexp [10], which are publicly available, are used. In order to make a fair comparison, we use the same uniform number generator for all implementations. The mixed multiplicative congruential (Lehmer) generator [28] used in the FastNorm2 implementation is chosen. Software implementations are run on an Intel Pentium-IV 2.6-GHz PC is equipped with 1-GB DDR-SDRAM. They are written in ANSI C and compiled

[1]In FastNorm2, the pool is updated periodically, whereas in our hardware implementation it is not.

with the GNU gcc 3.2.2 compiler with -O3 optimization, generating double precision floating-point numbers. Note that all software implementations pass the χ^2 and A-D tests. The results are shown in Table IV. It can be seen that our hardware designs are faster than software implementations by 2–609 times, depending on the device used and the resource utilization. Looking at the PC results, we can see that the Wallace method performs significantly better than other methods.

VII. Conclusion

We have presented a hardware Gaussian noise generator using the Wallace method to support simulations which involve very large numbers of samples.

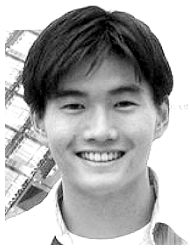
Our noise generator architecture contains four stages. It takes up approximately 3% of a Xilinx Virtex-II XC2V4000-6 FPGA and half of a Xilinx Spartan-III XC3S200E-5, and can produce 155 million samples per second. Further improvement in performance can be obtained by concurrent execution: 20 parallel instances of the noise generator on an XC2V4000-6 FPGA at 115 MHz can run 51 times faster than software on a 2.6-GHz Pentium-IV PC. The quality of the noise samples is confirmed by two statistical tests: the χ^2 test and the A-D test, and also by applications involving LDPC decoding. We are currently exploring methods for characterizing and optimizing Gaussian noise generators, such that the most appropriate noise generator can be selected for a given application.

Acknowledgment

The authors would like to thank R. P. Brent, A. Abdul Gaffar, A. C. H. Ng, R. C. C. Cheung, E. Vallés, and the reviewers for their assistance.

References

[1] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes in C*. Cambridge, U.K.: Cambridge Univ. Press, 1993, vol. 2.
[2] B. Levine, R. Taylor, and H. Schmit, "Implementation of near Shannon limit error-correcting codes using reconfigurable hardware," in *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, 2000, pp. 217–226.
[3] A. Brace, D. Gątarek, and M. Musiela, "The market model of interest rate dynamics," *Math. Finance*, vol. 7, no. 2, pp. 127–155, 1997.
[4] A. Bergstrom, "Gaussian estimation of mixed-order continuous-time dynamic models with unobservable stochastic trends from mixed stock and flow data," *Econometric Theory*, vol. 13, no. 4, pp. 467–505, 1997.
[5] B. Jung, H. Lenhof, P. Müller, and C. Rüb, "Langevin dynamics simulations of macromolecules on parallel computers," in *Macromolecular Theory Simul.*, 1997, pp. 507–521.
[6] C. Wallace, "Fast pseudorandom generators for normal and exponential variates," *ACM Trans. Math. Softw.*, vol. 22, no. 1, pp. 119–127, 1996.
[7] R. Gallager, "Low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. IT–8, pp. 21–28, 1962.
[8] J. Ahrens and U. Dieter, "An alias method for sampling from the normal distribution," *Computing*, vol. 42, no. 2–3, pp. 159–170, 1989.
[9] J. Leva, "A fast normal random number generator," *ACM Trans. Math. Softw.*, vol. 18, no. 4, pp. 449–453, 1992.
[10] G. Marsaglia and W. Tsang, "The Ziggurat method for generating random variables," *J. Statist. Softw.*, vol. 5, no. 8, pp. 1–7, 2000.
[11] D. Knuth, *Seminumerical Algorithms*, 3rd ed, ser. The Art of Computer Programming. Reading, MA: Addison-Wesley, 1997, vol. 2.
[12] W. Hörmann and J. Leydold, "Continuous random variate generation by fast numerical inversion," *ACM Transactions on Modeling and Computer Simulation*, vol. 13, no. 4, pp. 347–362, 2003.
[13] G. Box and M. Muller, "A note on the generation of random normal deviates," *Ann. Math. Statist.*, vol. 29, pp. 610–611, 1958.
[14] E. Boutillon, J. Danger, and A. Gazel, "Design of high speed AWGN communication channel emulator," *Analog Integr. Circuits Signal Process.*, vol. 34, no. 2, pp. 133–142, 2003.
[15] (2002) Additive white Gaussian noise (AWGN) Core v1.0. Xilinx Inc. [Online]. Available: http://www.xilinx.com
[16] J. Chen, J. Moon, and K. Bazargan, "Reconfigurable readback-signal generator based on a field-programmable gate array," *IEEE Trans. Magn.*, vol. 40, no. 3, pp. 1744–1750, Mar. 2004.
[17] Y. Fan, Z. Zilic, and M. Chiang, "A versatile high speed bit error rate testing scheme," in *Proc. IEEE Int. Symp. Quality Electronic Design*, 2004, pp. 395–400.
[18] D. Lee, W. Luk, J. Villasenor, and P. Cheung, "A Gaussian noise generator for hardware-based simulations," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1523–1534, Dec. 2004.
[19] R. Brent, "A fast vectorised implementation of Wallace's normal random number generator," The Australian National University, ANU Computer Sci.Tech. Rep. TR-CS-97-07, 1997.
[20] C. Rüb, "On Wallace's method for the generation of normal variates," Max-Planck-Institut für Informatik, Germany, MPI Informatik Res. Rep. MPI-I-98-1-020, 1998.
[21] P. Chu and R. Jones, "Design techniques of FPGA based random number generator," presented at the Military and Aerospace Applications of Programmable Devices and Technology Conf., Laurel, MD, 1999.
[22] P. L'Ecuyer, "Maximally equidistributed combined Tausworthe generators," *Math. Comput.*, vol. 65, no. 213, pp. 203–213, 1996.
[23] C. Wallace. (2003) MDMC software—Random number generators. [Online]. Available: http://www.datamining.monash.edu.au/software/random
[24] (2004) Xilinx System Generator User Guide v6.3. Xilinx Inc.. [Online]
[25] G. Marsaglia. (1997) Diehard: A battery of tests of randomness. [Online]. Available: http://stat.fsu.edu/~geo/diehard.html
[26] R. D'Agostino and M. Stephens, *Goodness-of-Fit Techniques*: Marcel Dekker Inc., 1986.
[27] A. Vila Casado, W. Weng, and R. Wesel, "Multiple rate low-density parity-check codes with constant block length," in *Proc. IEEE Asilomar Conf. Signals, Systems and Computers*, 2004, pp. 2010–2014.
[28] C. Wallace, "A long-period pseudo-random generator," Monash Univ., Australia, Tech. Rep. TR89/123, 1989.



Dong-U Lee (S'01–M'05) received the B.Eng. degree in information systems engineering and the Ph.D. degree in computing, both from Imperial College, London, U.K., in 2001 and 2004, respectively.

He is currently a Postdoctoral Researcher at the Electrical Engineering Department, University of California, Los Angeles (UCLA), where he is developing hardware implementations of communication algorithms for deep-space communications with the Jet Propulsion Laboratory, NASA. He visited UCLA in 2002 and 2003 as a Visiting Scholar, where he developed hardware designs for LDPC codes. His research interests include reconfigurable computing, computer arithmetic, communications and video image processing.

Wayne Luk (S'85–M'89) received the M.A., M.Sc., and Ph.D. degrees in engineering and computer science from the University of Oxford, Oxford, U.K.

He is a Member of Academic Staff in the Department of Computing, Imperial College of Science, Technology, and Medicine, London, U.K., 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.

John D. Villasenor (M'90–SM'98) received the B.S. degree from the University of Virginia, Charlottesville, in 1985, and the M.S. and Ph.D. degrees from Stanford University, Stanford, CA, in 1988 and 1989, respectively, 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, University 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.

Guanglie Zhang received the B.S., M.A., and Ph.D. degrees in electrical engineering from Xi'an Jiaotong University, Xi'an, China, in 1997, 2000, and 2003, respectively.

He is currently a Postdoctoral Research Fellow in the Chinese University of Hong Kong. His research interests are in the area of reconfigurable computing, embedded system, video processing, and MEMS sensor technologies.

Philip H. W. Leong (M'88–SM'97) received the B.Sc., B.E., and Ph.D. degrees from the University of Sydney, Sydney, Australia, in 1986, 1988, and 1993, respectively.

In 1989, he was a research engineer at AWA Research Laboratory, Sydney Australia. From 1990 to 1993, he was a postgraduate student and research assistant at the University of Sydney, where he worked on low power analogue VLSI circuits for arrhythmia classification. In 1993, he was a consultant to SGS Thomson Microelectronics, Milan, Italy. He was a Lecturer at the Department of Electrical Engineering, University of Sydney from 1994 to 1996. He is currently an Associate Professor in the Department of Computer Science and Engineering, Chinese University of Hong Kong and the Director of the Custom Computing Laboratory. He is the author of more than 70 technical papers and five patents. His research interests include reconfigurable computing, digital systems, parallel computing, cryptography and signal processing.