Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 675130, 11 pages
doi:10.1155/2012/675130

Research Article

A Hardware Efficient Random Number Generator for Nonuniform Distributions with Arbitrary Precision

Christian de Schryver,[1] Daniel Schmidt,[1] Norbert Wehn,[1] Elke Korn,[2] Henning Marxen,[2] Anton Kostiuk,[2] and Ralf Korn[2]

[1] *Microelectronic Systems Design Research Group, University of Kaiserslautern, Erwin-Schroedinger-Straße, 67663 Kaiserslautern, Germany*
[2] *Stochastic Control and Financial Mathematics Group, University of Kaiserslautern, Erwin-Schroedinger-Straße, 67663 Kaiserslautern, Germany*

Correspondence should be addressed to Christian de Schryver, schryver@eit.uni-kl.de

Received 30 April 2011; Accepted 23 November 2011

Academic Editor: Ron Sass

Copyright © 2012 Christian de Schryver et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Nonuniform random numbers are key for many technical applications, and designing efficient hardware implementations of non-uniform random number generators is a very active research field. However, most state-of-the-art architectures are either tailored to specific distributions or use up a lot of hardware resources. At ReConFig 2010, we have presented a new design that saves up to 48% of area compared to state-of-the-art inversion-based implementation, usable for arbitrary distributions and precision. In this paper, we introduce a more flexible version together with a refined segmentation scheme that allows to further reduce the approximation error significantly. We provide a free software tool allowing users to implement their own distributions easily, and we have tested our random number generator thoroughly by statistic analysis and two application tests.

1. Introduction

The fast generation of random numbers is essential for many tasks. One of the major fields of application are Monte Carlo simulation, for example widely used in the areas of financial mathematics and communication technology.

Although many simulations are still performed on high-performance CPU or general-purpose graphics processing unit (GPGPU) clusters, using reconfigurable hardware accelerators based on field programmable gate arrays (FPGAs) can save up to at least one order of magnitude of power consumption if the random number generator (RNG) is located on the accelerator. As an example, we have implemented the generation of normally distributed random numbers on the three mentioned architectures. The results for the achieved throughput and the consumed energy are given in Table 1. Since one single instance of our proposed hardware design (together with a uniform random number generator) consumes less than 1% of the area on the used Xilinx Virtex-5 FPGA, we have introduced a line with the extrapolated values for 100 instances to highlight the enormous potential of hardware accelerators with respect to the achievable throughput per energy.

In this paper, we present a refined version of the floating point-based nonuniform random number generator already shown at ReConFig 2010 [1]. The modifications allow a higher precision while having an even lower area consumption compared to the previous results. This is due to a refined synthesis. The main benefits of the proposed hardware architecture are the following:

(i) The area saving is even higher than the formerly presented 48% compared to the state-of-the-art FPGA implementation of Cheung et al. from 2007 [2].

(ii) The precision of the random number generator can be adjusted and is mainly independent of the output resolution of the auxiliary uniform RNG.

Table 1: Normal random number generator architecture comparison.

Implementation	Architecture	Power consumption	Throughput [M samples/s]	Energy per sample
Fast Mersenne Twister, optimized for SIMD	Intel Core 2 Duo PC 2.0 GHz, 3 GB RAM, one core only	~100 W	600	166.67 pJ
Nvidia Mersenne Twister + Box-Muller CUDA	Nvidia GeForce 9800 GT	~105 W	1510	69.54 pJ
Nvidia Mersenne Twister + Box-Muller OpenCL			1463	71.77 pJ
Proposed architecture, only one instance [1]	Xilinx FPGA Virtex-5FX70T-3 380 MHz	~1.3 W	397	3.43 pJ
Proposed architecture, 100 instances		~1.9 W	39700	0.05 pJ

(iii) Our design is exhaustively tested by statistical and application tests to ensure the high quality of our implementation.

(iv) For the convenience of the user, we provide a free tool that creates the lookup table (LUT) entries for any desired nonuniform distribution with a user-defined precision.

The rest of the paper is organized as follows. In Section 2, we give an overview about current techniques to obtain uniform (pseudo-)random numbers and to transform them to nonuniform random numbers. Section 3 shows state-of-the-art inversion-based FPGA nonuniform random number generators, as well as a detailed description of the newly introduced implementation. It also presents the LUT creator tool needed for creating the lookup table entries. How floating point representation can help to reduce hardware complexity is explained in Section 4. Section 5 shows detailed synthesis results of the original and the improved implementation and elaborates on the excessive quality tests that we have applied. Finally, Section 6 concludes the paper.

2. Related Work

The efficient implementation of random number generators in hardware has been a very active research field for many years now. Basically, the available implementations can be divided into two main groups, that are

(i) random number generators for uniform distributions,

(ii) circuits that transform uniformly distributed random numbers into different target distributions.

Both areas of research can, however, be treated as nearly distinct. We will give an overview of available solutions out of both groups.

2.1. Uniform Random Number Generators. Many highly elaborate implementations for uniform RNGs have been published over the last decades. The main common characteristic of all is that they produce a bit vector with n bits that represent (if interpreted as an unsigned binary-coded integer and divided by $2^n - 1$) values between 0 and 1. The set of all results that the generator produces should be as uniformly as possible distributed over the range $(0, 1)$.

A lot of fundamental research on uniform random number generation has already been made before 1994. A comprehensive overview of the work done until that point in time has been given by L'Ecuyer [3] who summarized the main concepts of uniform RNG construction and their mathematical backgrounds. He also highlights the difficulties of evaluating the quality of a uniform RNG, since in the vast majority of the cases, we are dealing not with truly random sequences (as, e.g., Bochard et al. [4]), but with pseudorandom or quasirandom sequences. The latter ones are based on deterministic algorithms. *Pseudorandomness* means that the output of the RNG looks to an observer like a truly random number sequence if only a limited period of time is considered. *Quasirandom* sequences, however, do not aim to look very random at all, but rather try to cover a certain bounded range in a best even way. One major field of application for quasirandom numbers is to generate a suitable test point set for Monte Carlo simulations, in order to increase the performance compared to pseudorandom number input [5, 6].

One of the best investigated high-quality uniform RNGs is the Mersenne Twister as presented by Matsumoto and Nishimura in 1998 [7]. It is used in many technical applications and commercial products, as well as in the RNG research domain. Well-evaluated and optimized software programs are available on their website [8]. Nvidia has adapted the Mersenne Twister to their GPUs in 2007 [9].

A high-performance hardware architecture for the Mersenne Twister has been presented in 2008 by Chandrasekaran and Amira [10]. It produces 22 millions of samples per second, running at 24 MHz. Banks et al. have compared their Mersenne Twister FPGA design to two multiplier pseudo-RNGs in 2008 [11], especially for the use in financial mathematics computations. They also clearly show that the random number quality can be directly traded off against the consumed hardware resources.

Tian and Benkrid have presented an optimized hardware implementation of the Mersenne Twister in 2009 [12], where they showed that an FPGA implementation can outperform a state-of-the-art multicore CPU by a factor of about 25, and a GPU by a factor of about 9 with respect to the throughput. The benefit for energy saving is even higher.

We will not go further into details here since we concentrate on obtaining nonuniform distributions. Nevertheless, it is worth mentioning that quality testing has been a big issue for uniform RNG designs right from the beginning [3]. L'Ecuyer and Simard invented a comprehensive test suite named *TestU01* [13] that is written in C (the most recent version is 1.2.3 from August, 2009). This suite combines a lot of various tests in one single program, aimed to ensure the quality of specific RNGs. For users without detailed knowledge about the meaning of each single test, the TestU01 suite contains three test batteries that are predefined selections of several tests:

(i) *Small Crush*: 10 tests,

(ii) *Crush*: 96 tests,

(iii) *Big Crush*: 106 tests.

TestU01 includes and is based on the tests from the other test suites that have been used before, for example, the Diehard Test Suite by Marsaglia from 1995 [14] or the fundamental considerations made by Knuth in 1997 [15].

For the application field financial mathematics (what is also our main area of research), McCullough has strongly recommended the use of TestU01 in 2006 [16]. He comments on the importance of random number quality and the need of excessive testing of RNGs in general.

More recent test suites are the very comprehensive Statistical Test Suite (STS) from the US National Institute of Standards and Technology (NIST) [17] revised in August, 2010, and the Dieharder suite from Robert that was just updated in March, 2011 [18].

2.2. Obtaining Nonuniform Distributions. In general, nonuniform distributions are generated out of uniformly distributed random numbers by applying appropriate conversion methods. A very good overview of the state-of-the-art approaches has been given by Thomas et al. in 2007 [19]. Although they are mainly concentrating on the normal distribution, they show that all applied conversion methods are based on one of the four underlying mechanisms:

(i) transformation,

(ii) rejection sampling,

(iii) inversion,

(iv) recursion.

Transformation uses mathematical functions that provide a relation between the uniform and the desired target distribution. A very popular example for normally distributed random numbers is the Box-Muller method from 1958 [20]. It is based on trigonometric functions and transforms a pair of uniformly distributed into a pair of normally distributed random numbers. Its advantage is that it provides a pair of random numbers for each call deterministically. The Box-Muller method is prevalent nowadays and mainly used for CPU and GPU implementations. A drawback for hardware implementations is the high demand of resources needed to accurately evaluate the trigonometric functions [21, 22].

Rejection sampling can provide a very high accuracy for arbitrary distributions. It only accepts input values if they are within specific predefined ranges and discards others. This behavior may lead to problems if quasirandom number input sequences are used, and (especially important for hardware implementations) unpredictable stalling might be necessary. For the normal distribution, the Ziggurat method [23] is the most common example of rejection sampling and is implemented in many software products nowadays. Some optimized high-throughput FPGA implementations exist, for example, by Zhang et al. from 2005 [24] who generated 169 millions of samples per second on a Xilinx Virtex-2 device running at 170 MHz. Edrees et al. have proposed a scalable architecture in 2009 [25] that achieves up to 240 Msamples on a Virtex-4 at 240 MHz. By increasing the parallelism of their architecture, they predicted to achieve even 400 Msamples for a clock frequency of around 200 MHz.

The *inversion method* applies the inverse cumulative distribution function (ICDF) of the target distribution to uniformly distributed random numbers. The ICDF converts a uniformly distributed random number $x \in (0, 1)$ to one output $y = \mathrm{icdf}(x)$ with the desired distribution. Since our proposed architecture is based on the inversion method, we go more into details in Section 3.

The so far published hardware implementations of inversion-based converters are based on piecewise polynomial approximation of the ICDF. They use lookup tables (LUTs) to store the coefficients for various sampling points. Woods and Court have presented an ICDF-based random number generator in 2008 [26] that is used to perform Monte Carlo simulations in financial mathematics. They use a nonequidistant hierarchical segmentation scheme with smaller segments in the steeper parts of the ICDF, what reduces the LUT storage requirements significantly without losing precision. Cheung et al. have shown a very elaborate multilevel segmentation approach in 2007 [2].

The *recursion method* introduced by Wallace in 1996 [27] uses linear combinations of originally normally distributed random numbers to obtain further ones. He provides the source code of his implementation for free [28]. Lee et al. have shown a hardware implementation in 2005 [29] that produces 155 millions of samples per second on a Xilinx Virtex-2 FPGA running at 155 MHz.

3. The Inversion Method

The most genuine way to obtain nonuniform random numbers is the *inversion method*, as it preserves the properties of the originally sampled sequence [30]. It uses the ICDF of the desired distribution to transform every input $x \in (0, 1)$ from a uniform distribution into the output sample $y = \mathrm{icdf}(x)$ of the desired one. In case of a continuous and strictly monotone cumulative distribution (CDF) function F, we have

$$F_{\mathrm{out}}(\alpha) = \mathbb{P}(\mathrm{icdf}(U) \leq \alpha) = \mathbb{P}(U \leq F(\alpha)) = F(\alpha). \quad (1)$$

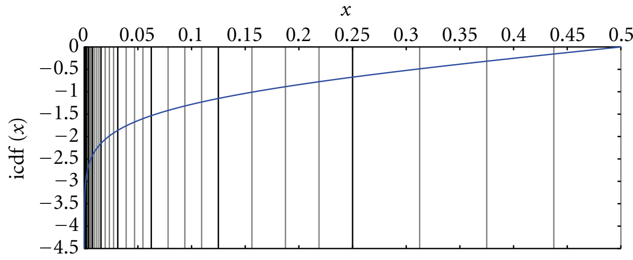


FIGURE 1: Segmentation of the first half of the Gaussian ICDF.

Identical CDFs always imply the equality of the corresponding distributions. For further details, we refer to the works of Korn et al. [30] or Devroye [31].

Due to the above mechanism, the inversion method is applicable to transform also quasirandom sequences. In addition to that, it is completable with variance reduction techniques, for example, antithetic variates [26]. Inversion-based methods in general can be used to obtain any desired distribution using memory-based lookup tables. This is especially advantageous for hardware implementations, since for many distributions, no closed-form expressions for the ICDF exist, and approximations have to be used. The most common approximations for the Gaussian ICDF (see Peter [32] and Moro [33]) are, however, based on higher-grade rational polynomials, but, for that reason, they cannot be efficiently used for a hardware implementation.

3.1. State-of-the-Art Architectures. In 2007, Cheung et al. proposed to implement the inversion using the piecewise polynomial approximation [2]. It is based on a fixed point representation and uses a hierarchical segmentation scheme that provides a good trade-off between hardware resources and accuracy. For the normal distribution (as well as any other symmetric distribution), it is also common to use the following simplification: due to the symmetry of the normal ICDF around $x = 0.5$, its approximation is implemented only for values $x \in (0, 0.5)$, and one additional random bit is used to cover the full range. For the Gaussian ICDF, Cheung et al. suggest to divide the range $(0, 0.5)$ into nonequidistant segments with doubling segment sizes from the beginning to the end of the interval. Each of these segments should then be subdivided into inner segments of equal size. Thus, the steeper regions of the ICDF close to 0 are covered by more smaller segments than the regions close to 0.5, where the ICDF is almost linear. This segmentation of the Gaussian ICDF is shown in Figure 1. By using a polynomial approximation of a fixed degree within each segment, this approach allows to obtain an almost constant maximal absolute error over all segments. The inversion algorithm first determines in which segment the input x is contained, then retrieves the coefficients c_i of the polynomial for this segment from a LUT, and evaluates the output as $y = \sum c_i \cdot x^i$ afterwards.

Figure 2 explains how, for a given fixed point input x, the coefficients of the polynomial are retrieved from the lookup table (that means how the address of the corresponding

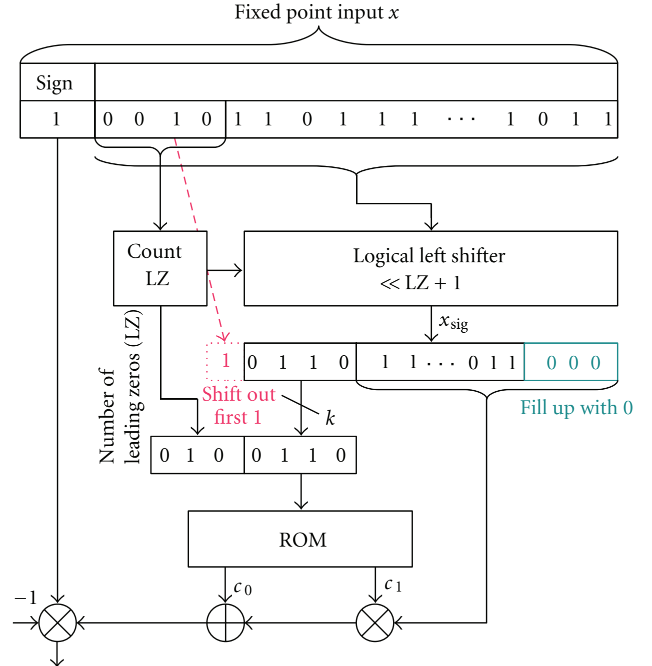


FIGURE 2: State-of-the-art architecture.

segment in the LUT is generated). It starts with counting the number of leading zeros (LZ) in the binary representation of x. It uses a bisection technique to locate the segment of the first level: that means numbers with the most significant bit (MSB) 1 lie in the segment $[0.25, 0.5)$ and those with 0 correspondingly in $(0, 0.25)$, numbers with second MSB 1 (i.e., $x = 01\ldots$) lie in the segment $[0.125, 0.25)$ and those with 0 (i.e., $x = 00\ldots$) in $(0, 0.125)$, and so forth. Then the input x is shifted left by $LZ + 1$ bits, such that x_{sig} is the bit sequence following the most significant 1-bit in x. The k MSBs of x_{sig} determine the subsegments of the second level (the equally sized ones). Thus, the LUT address is the concatenation of LZ and $MSB_k(x_{\text{sig}})$. The inverted value equals the approximating polynomial for the ICDF in that segment evaluated on the remaining bits of x_{sig}. The architecture for the case of linear interpolation [2] is presented in Figure 2. It approximates the inversion with a maximum absolute error of $0.3 \cdot 2^{-11}$.

The works of Lee et al. [34, 35] are also based on this segmentation/LUT approach. They use the same technique to create generators for the log-normal and the exponential distributions, with only slight changes in the segmentation scheme. For the exponential distribution, the largest segment starts near 0, sequentially followed by the twice smaller segments towards 1. For the log-normal distribution, neighboring segments double in size starting from 0 until 0.5 and halve in size towards 1.

But this approach has a number of drawbacks as follows.

(i) *Two uniform RNGs needed for a large output range*: due to the fixed point implementation, the output range is limited by a number of input bits. The smallest positive value that can be represented by an

m bit fixed point number is 2^{-m}, what in the case of a 32-bit input value leads to the largest inverted value of $\text{icdf}(2^{-32}) = 6.33\sigma$. To obtain a larger range of normal random variable up to 8.21σ, the authors of [2] concatenate the input of two 32-bit uniform RNGs and pass a 53-bit fixed point number into the inversion unit, at the cost of one additional uniform RNG. The large number of input bits results in the increased size of the LZ counter and shifter unit, that dominate the hardware usage of the design.

(ii) *A large number of input bits is wasted:* as a multiplier with a 53-bit input requires a large amount of hardware resources, the input is quantified to 20 significant bits before the polynomial evaluation. Thus, in the region close to the 0.5, a large amount of the generated input bits is wasted.

(iii) *Low resolution in the tail region:* for the tail region (close to 0), there are much less than 20 significant bits left after shifting over the LZ. This limits the resolution in the tail of the desired distribution. In addition, as there are no values between 2^{-53} and 2^{-52} in this fixed point representation, the proposed RNG does not generate output samples between $\text{icdf}(2^{-52}) = 8.13\sigma$ and $\text{icdf}(2^{-53}) = 8.21\sigma$.

3.2. Floating Point-Based Inversion. The drawbacks mentioned before result from the fixed point interpretation of the input random numbers. We therefore propose to use a floating point representation.

First of all, we do not use any floating point arithmetics in our implementation. Our design does not contain any arithmetic components like full adders or multipliers that usually blow up a hardware architecture. We just exploit the representation of a floating point number consisting of an exponent and a mantissa part. We also do not use IEEE 754 [36] compliant representations, but have introduced our own optimized interpretation of the floating point encoded bit vector.

3.2.1. Hardware Architecture. We have enhanced our formerly architecture presented at ReConFig 2010 [1] with a second *part* bit that is used to split the encoded half of the ICDF into two parts. The additionally necessary hardware is just one multiplexer and an adder with one constant input, that is, the offset for the address range of the LUT memory where the coefficients for the second half are located.

Figure 3 shows the structure of our proposed ICDF lookup unit. Compared to our former design, we have renamed the *sign_half* bit to *symmetry* bit. This term is more appropriate now since we use this bit to identify in which half of a symmetrical ICDF the output value is located. In this case, we also only encode one half and use the symmetry bit to generate a symmetrical coverage of the range (0, 1) (see Section 3.1).

Each part itself is divided further into *octaves* (formerly *segments*), that are halved in size by moving towards the outer borders of the parts (compare with Section 3.1). One exception is that the both very smallest octaves are equally

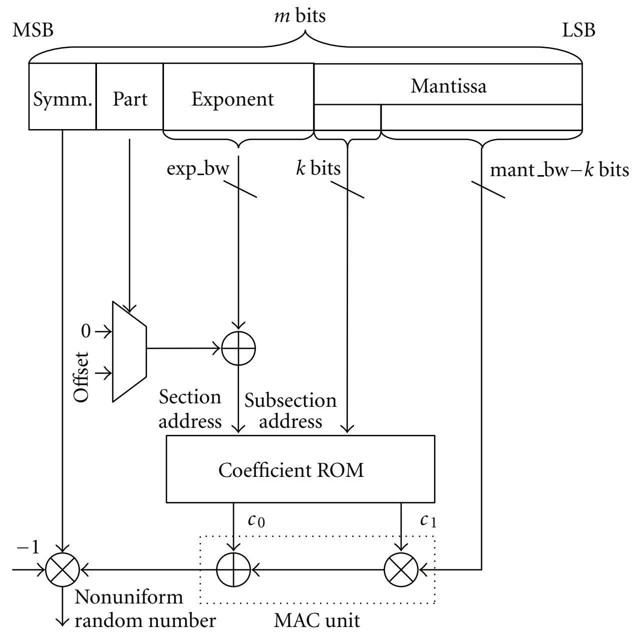


FIGURE 3: ICDF lookup structure for linear approximation.

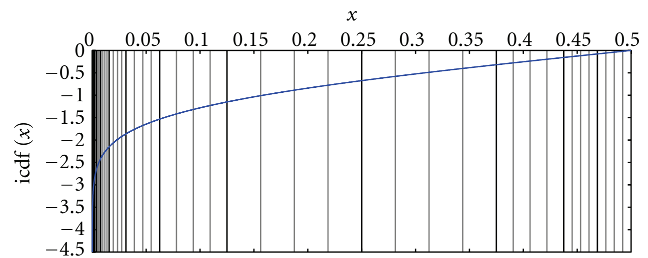


FIGURE 4: Double segmentation refinement for the normal ICDF.

sized. In general, the number of octaves for each part can be different. As an example, Figure 4 shows the left half of the Gaussian ICDF with a nonequal number of octaves in both parts.

Each octave is again divided into 2^k equally sized *subsections*, where k is the number of bits taken from the mantissa part in Figure 3. k therefore has the same value for both parts, but is not necessarily limited to powers of 2.

The input address for the coefficient ROM is now generated in the following way.

(i) The offset is exactly the number of subsections in part 0, that means all subsections in the range from 0 to 0.25 for a symmetric ICDF:

$$\text{offset} = 2^k \cdot \text{number of octaves in part 0}. \tag{2}$$

(ii) In part 0, the address is the concatenation of the exponent (giving the number of the octave) and the k dedicated mantissa bits (for the subsection).

(iii) In part 1, the address is the concatenation of (exponent + offset) and the k mantissa bits.

Table 2: Selected tool configuration for provided error values.

Parameter	Value
Growing octaves	54
Diminishing octaves	4
Subsection bits (k)	3
Mantissa bits ($mant_bw - k$)	18
Output precision bits	42

Table 3: Maximum approximation errors for different distributions.

Distribution	Symmetry	Maximum absolute error
Normal	Point	0.000383397
Log-normal (0, 1)	None	0.00233966
Gamma (0, 1)	None	0.00787368
Laplace (1)	Point	0.000901326
Exponential (1)	None	0.000787368
Rayleigh (1)	None	0.000300666

This floating point-based addressing scheme efficiently exploits the LUT memory in a hardware friendly way since no additional logic for the address generation is needed compared to other state-of-the-art implementations (see Sections 2.2 and 3.1). The necessary LUT entries can easily be generated with our freely available tool presented in Section 3.2.2.

3.2.2. The LUT Creator Tool. For the convenience of the users who like to make use of our proposed architecture, we have developed a flexible C++ class package that creates the LUT entries for any desired distribution function. The tool has been rewritten from scratch, compared to the one presented at ReConFig 2010 [1]. It is freely available for download on our website (http://ems.eit.uni-kl.de/).

Most of the detailed documentation is included in the tool package itself. It uses Chebyshev approximation, as provided by the GNU Scientific Library (GSL) [37]. The main characteristics of the new tool are as follows.

(i) It allows any function defined on the range (0, 1) to be approximated. However, the GSL already provides a large number of ICDFs that may be used conveniently.

(ii) It provides configurable segmentation schemes with respect to

  (i) symmetry,

  (ii) one or two parts,

  (iii) independently configurable number of octaves per part,

  (iv) number of subsections per octave.

(iii) The output quantization is configurable by the user.

(iv) The degree of the polynomial approximation is arbitrary.

Our LUT creator tool also has a built-in error estimation that directly calculates the maximum errors between the provided optimal function and the approximated version. For linear approximation and the configuration shown in Table 2, we present a selection of maximum errors in Table 3. For optimized parameter sets that take the specific characteristics of the distributions into account, we expect even lower errors.

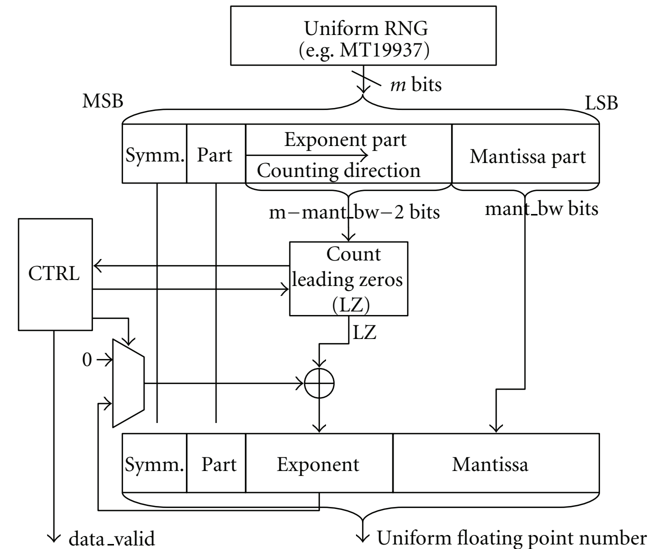


Figure 5: Architecture of the proposed floating point RNG.

4. Generating Floating Point Random Numbers

Our proposed LUT-based inversion unit shown in Section 3.2.1 requires dedicated floating point encoded numbers as inputs. In this section, we present an efficient hardware architecture for generating these numbers. Our design consumes an arbitrary-sized bit vector from any uniform random number generator and transforms it into the floating point representation with adjustable precisions. In Section 5.2, we show that our floating point converter maintains the properties of the uniform random numbers provided by the input RNG.

Figure 5 shows the structure of our unit and how it maps the incoming bit vector to the floating point parts. Compared to our architecture presented on ReConFig 2010 [1], we have enhanced our converter unit with an additional *part* bit. It provides the information if we use the first or the second segmentation refinement of the ICDF approximation (see Section 3.2).

For each floating point random number that is to be generated, we extract the symmetry and the part bit in the first clock cycle, as well as the mantissa part that is just mapped to the output one to one. The mantissa part in our case is encoded with a hidden bit, for a bit width of

mant_bw bits, it can therefore represent the values 1, $1 + (1/2^{mant_bw})$, $1 + (2/2^{mant_bw}), \ldots, 2 - (1/2^{mant_bw})$.

The exponent in our floating point encoding represents the number of leading zeros (LZs) that we count from the exponent part of the incoming random number bit vector. We can use this exponent value directly as the segment address in our ICDF lookup unit described in Section 3.2.1. In the hardware architecture, the leading zeros computation is, for efficiency reasons, implemented as a comparator tree.

However, if we would only consider one random number available at the input of our converter, the maximum value for the floating point exponent would be $m - mant_bw - 2$, with all the bits in the input exponent part being zero. To overcome this issue, we have introduced a parameter determining the maximum value of the output floating point exponent, *max_exp*. If now all bits in the input exponent part are detected to be zero, we store the value of already counted leading zeros and consume a second random number where we continue counting. For the case that we have again only zeros, we consume a third number and continue if either one is detected in the input part or the predefined maximum of the floating point exponent, *max_exp*, is reached. In this case, we set the *data_valid* signal to 1 and continue with generating the next floating point random number.

For the reason that we have to wait for further input random numbers to generate one floating point result, we need a stalling mechanism for all subsequent units of the converter. Nevertheless, depending on size of the exponent part in the input bit vector that is arbitrary, the probability for necessary stalling can be decreased significantly. A second random number is needed with the probability of $P_2 = 1/2^{m-mant_bw-2}$, a third with $P_3 = 1/2^{2\cdot(m-mant_bw-2)}$, and so on. For an input exponent part with the size of 10 bits, for example, $P_2 = 1/2^{10} = 0.976 \cdot 10^{-3}$, which means that on average one additional input random number has to be consumed for generating about 1,000 floating point results.

We have already presented pseudocode for our converter unit at the ReConFig 2010 [1] that we have enhanced now for our modified design by storing two sign bits. The modified version is shown in Algorithm 1.

```
rn ← get_random_number();
symmetry ← rn.get_symmetry();
part ← rn.get_part();
mant ← rn.get_mantissa();
exp ← rn.get_exponent();
LZ ← exp.count_leading_zeros();
while (exp == 0) and (LZ < max_exp) do
  | rn ← get_random_number();
  | exp ← rn.get_exponent();
  | LZ ← LZ + exp.count_leading_zeros();
end
LZ ← min(LZ, max_exp);
return symmetry, part, mant, LZ
```

Algorithm 1: Floating point generation algorithm.

Table 4: ReConFig 2010 [1]: optimized for speed.

	Slices	FFs	LUTs	BRAMs	DSP48E
Floating point converter	30	62	40	—	—
LUT evaluator	12	47	—	1	1
Complete design	40	108	39	1	1

Table 5: Proposed design: optimized for speed.

	Slices	FFs	LUTs	BRAMs	DSP48E
Floating point converter	30	62	40	—	—
LUT evaluator	18	47	7	1	1
Complete design	42	109	46	1	1

Table 6: ReConFig 2010 [1]: optimized for area.

	Slices	FFs	LUTs	BRAMs	DSP48E
Floating point converter	13	11	26	—	—
LUT evaluator	12	47	—	1	1
Complete design	26	84	26	1	1

5. Synthesis Results and Quality Test

In addition to our conference paper presented at ReConFig 2010 [1], we provide detailed synthesis results in this section on a Xilinx Virtex-5 device, for both speed and area optimization. Furthermore, we show quality tests for the normal distribution.

5.1. Synthesis Results. Like for the proposed architecture from ReConFig 2010, we have optimized the bit widths to exploit the full potential of the Virtex-5 DSP48E slice that supports an $18 \cdot 25$ bit + 48 bit MAC operation. We therefore selected the same parameter values that are as follows: input bitwidth $m = 32$, $mant_bw = 20$, $max_exp = 54$, and $k = 3$ for subsegment addressing. The coefficient c_0 is quantized to 46 bits, and c_1 has 23 bits.

We have synthesized our proposed design and the architecture presented at ReConFig with the newer Xilinx ISE 12.4, allowing a fair comparison of the enhancement impacts. Both implementations have been optimized for area and speed, respectively. The target device is a Xilinx Virtex-5 XC5FX70T-3. All provided results are post place and route.

From Tables 4 and 5, we see that just by using the newer ISE version, we already save area of the whole nonuniform random number converter compared to the ReConFig result that was 44 slices (also optimized for speed) [1]. The maximum clock frequency is now 393 MHz compared to formerly 381 MHz.

Even with the ICDF lookup unit extension described in Section 3.2.1, the new design is two slices smaller than in the former version and can run at 398 MHz. We still consume one 36 Kb BRAM and one DSP48E slice.

The synthesis results for area optimization are given in Tables 6 and 7. The whole design now only occupies 31 slices on a Virtex-5 and still runs at 286 MHz instead of 259 MHz formerly. Compared to the ReConFig 2010 architecture, we

Table 7: Proposed design: optimized for area.

	Slices	FFs	LUTs	BRAMs	DSP48E
Floating point converter	13	11	26	—	—
LUT evaluator	18	47	7	1	1
Complete design	31	85	34	1	1

therefore consume about 20% more area by achieving a speedup of about 10% at a higher precision.

5.2. Quality Tests. Quality testing is an important part in the creation of a random number generator. Unfortunately, there are no standardized tests for nonuniform random number generators. Thus, for checking the quality of our design, we proceed in three steps: in the first step, we test the floating point uniform random number converter, and then we check the nonuniform random numbers (with a special focus on the normal distribution here). Finally, the random numbers are tested in two typical applications: an option pricing calculation with the Heston model [38] and the simulation of the bit error rate and frame error rate of a duo-binary turbo code from the WiMax standard.

5.2.1. Uniform Floating Point Generator. We have already elaborated on the widely used TestU01 suite for uniform random number generators in Section 2.1. TestU01 needs an equivalent fixed point precision of at least 30 bits, and for the big crush tests even 32 bits. The uniformly distributed floating point random numbers have been created as described in Section 4 with a mantissa of 31 bits from the output of a Mersenne Twister MT19937 [7].

The three test batteries small crush, crush, and big crush have been used to test the quality of the floating point random number generator. The Mersenne Twister itself is known to successfully complete all except two tests. These two tests are linear complexity tests that all linear feedback shift-register and generalized feedback shift-register-based random number generators fail (see [13] for more details). Our floating point transform of Mersenne random numbers also completes all but the specific two tests successfully. Thus, we conclude that our floating point uniform random number generator preserves the properties of the input generator and shows the same excellent structural properties.

For computational complexity reasons, for the following tests, we have restricted the output bit width of the floating point converter software implementation to 23 bits. The resolution is lower than the fixed point input in some regions, whereas in other regions a higher resolution is achieved. Due to the floating point representation, the regions with higher resolutions are located close to zero. Figure 6 shows a zoomed two-dimensional plot of random vectors produced by our design close to zero. It is important to notice that no patterns, clusters, or big holes are visible here.

Besides the TestU01 suite, the equidistribution of our random numbers has also been tested with several variants of

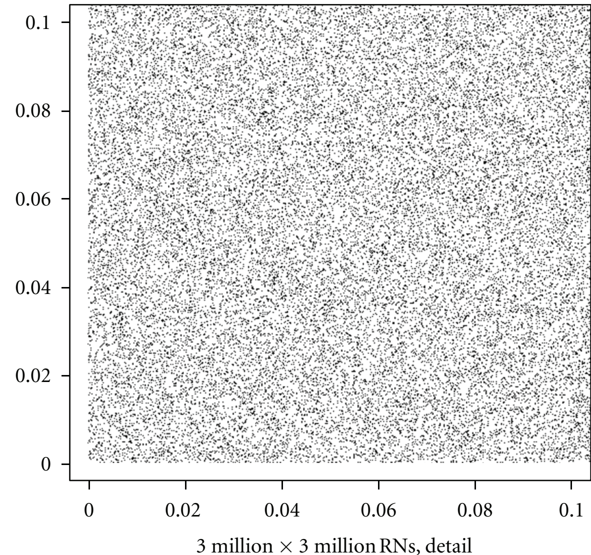


Figure 6: Detail of uniform 2D vectors around 0.

the frequency test mentioned by Knuth [15]. While checking the uniform distribution of the random numbers up to 12 bits, no extreme P value could be observed.

5.2.2. Nonuniform Random Number Generator. For the nonuniform random number generator, we have selected a specific set of commonly applied tests to examine and ensure the quality of the produced random numbers. In this paper, we focus on the tests performed for normally distributed random numbers, since those are most commonly used in many different fields of applications. Also the application tests presented below use normally distributed random numbers.

As a first step, we have run various χ^2-tests. In these tests, the empirical number of observations in several groups is compared with the theoretical number of observations. Test results that would only occur with a very low probability indicate a poor quality of the random numbers. This may be the case if either the structure of the random numbers does not fit to the normal distribution or if the numbers show more regularity than expected from a random sequence. The batch of random numbers in Figure 7 shows that the distribution is well approximated. The corresponding χ^2-test with 100 categories had a P value of 0.4.

The Kolmogorov-Smirnov test compares the empirical and the theoretical cumulative distribution function. Nearly all tests with different batch sizes were perfectly passed. Those not passed did not reveal an extraordinary P value. A refined version of the test, as described in Knuth [15] on page 51, sometimes had low P values. This is likely to be attributed to the lower precision in some regions of our random numbers, as the continuous CDF can not be perfectly approximated with random numbers that have fixed gaps. Other normality tests were perfectly passed, including the Shapiro-Wilk [39] test. Stephens [40] argues that the latter one is more suitable

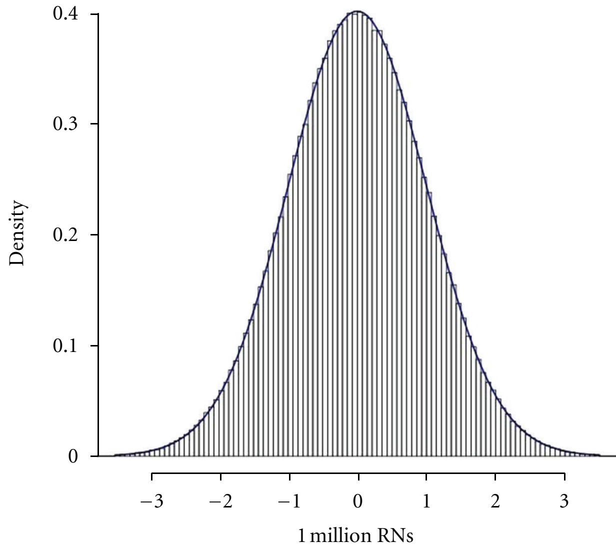


Figure 7: Histogram of Gaussian random numbers.

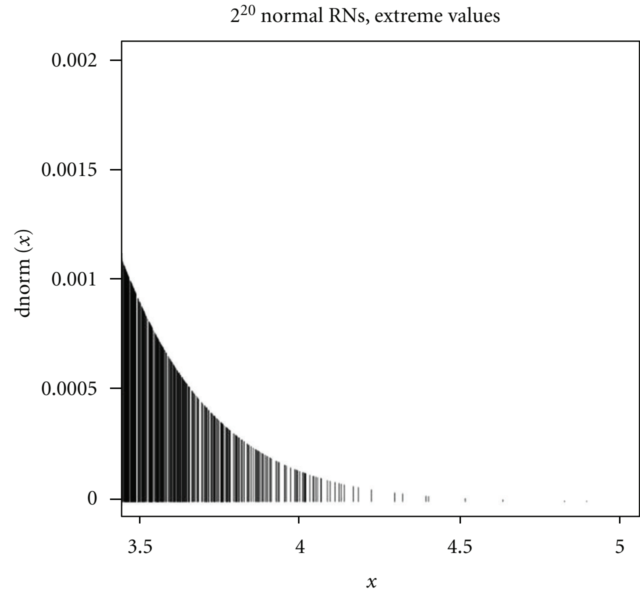


Figure 8: Tail of the empirical distribution function.

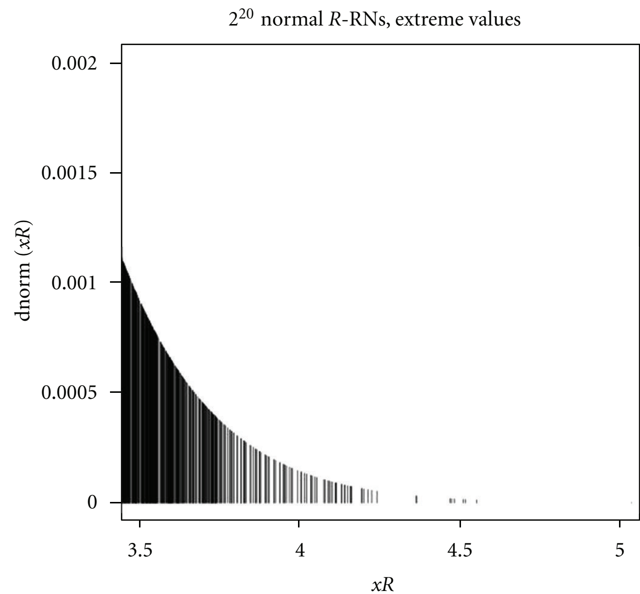


Figure 9: Tail of the empirical distribution function for the R RNG.

for testing the normality than the Kolmogorov-Smirnov test. The test showed no deviation from normality.

We not only compared our random numbers with the theoretical properties, but also with those taken from the well-established normal random number generator of the R language. It is based on a Mersenne Twister as well. Again, we used the Kolmogorov-Smirnov test, but no difference in distribution could be seen. Comparing the mean with the t-test and the variance with the F-test gave no suspicious results. The random numbers of our generator seem to have the same distribution as standard random numbers, with an exception of the reduced precision in the central region and an improved precion in the extreme values. This difference can be seen in Figures 8 and 9. Both depict the empirical results of a draw of 2^{20} random numbers, the first with the presented algorithm and the second with the RNG of R.

The tail distribution of the random numbers of the presented algorithm seems to be better in the employed test set. The area of extreme values is fitted without large gaps in contrast to the R random numbers. The smallest value from our floating point-based random number generator is $1 \cdot 2^{-54}$, compared to $1 \cdot 2^{-32}$ in standard RNGs, thus values of -8.37σ and 8.37σ can be produced. Our approximation of the inverse cumulative distribution function has an absolute error of less than $0.4 \cdot 2^{-11}$ in the achievable interval. Thus, the good structural properties of the uniform random numbers can be preserved. Due to the good properties of our random number generator, we expect it to perform well in the case of a long and detailed approximation, where rare extreme events can have a huge impact (consider risk simulations for insurances, e.g.).

5.2.3. Application Tests. Random number generators are always embedded in a strongly connected application environment. We have tested the applicability of our normal RNG in two scenarios: first, we have calculated an option price with the Heston model [38]. This calculation was done using the Monte Carlo simulation written in Octave. The provided RNG of Octave *randn*() has been replaced by a bit true model of our presented hardware design. For the whole benchmark set, we could not observe any peculiarities with respect to the calculated results and the convergence behavior of the Monte Carlo simulation. For the second application, we have produced a vast set of simulations of a wireless communications system. For comparison to our RNG, a Mersenne Twister and inversion using the Moro approximation [33] has been used. Also in this test,

no significant differences between the results from both generators could be observed.

6. Conclusion

In this paper, we present a new refined hardware architecture of a nonuniform random number generator for arbitrary distributions and precision. As input, a freely selectable uniform random number generator can be used. Our unit transforms the input bit vector into a floating point notation before converting it with an inversion-based method to the desired distribution. This refined method provides more accurate random numbers than the previous implementation presented at ReConFig 2010 [1], while occupying roughly the same amount of hardware resources.

This approach has several benefits. Our new implementation saves now more than 48% of the area on an FPGA compared to state-of-the-art implementations, while even achieving a higher output precision. The design can run at up to 398 MHz on a Xilinx Virtex-5 FPGA. The precision itself can be adjusted to the users' needs and is mainly independent of the output resolution of the uniform RNG. We provide a free tool allowing to create the necessary look-up table entries for any desired distribution and precision.

For both components, the floating point converter and the ICDF lookup unit, we have presented our hardware architecture in detail. Furthermore, we have provided exhaustive synthesis results for a Xilinx Virtex-5 FPGA. The high quality of the random numbers generated by our design has been ensured by applying extensive mathematical and application tests.

Acknowledgment

The authors gratefully acknowledge the partial financial support from the Center for Mathematical and Computational Modeling $(CM)^2$ of the University of Kaiserslautern.

References

[1] C. de Schryver, D. Schmidt, N. Wehn et al., "A new hardware ecient inversion based random number generator for non-uniform distributions," in *Proceedings of the International Conference on Recongurable Computing and FPGAs (ReConFig '10)*, pp. 190–195, December 2010.

[2] R. C. C. Cheung, D.-U. Lee, W. Luk, and J. D. Villasenor, "Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 8, pp. 952–962, 2007.

[3] P. L'Ecuyer, "Uniform random number generation," *Annals of Operations Research*, vol. 53, no. 1, pp. 77–120, 1994.

[4] N. Bochard, F. Bernard, V. Fischer, and B. Valtchanov, "True-randomness and pseudo-randomness in ring oscillator-based true random number generators," *International Journal of Reconfigurable Computing*, vol. 2010, article 879281, 2010.

[5] H. Niederreiter, "Quasi-Monte Carlo methods and pseudo-random numbers," *American Mathematical Society*, vol. 84, no. 6, p. 957, 1978.

[6] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*, Society for Industrial Mathematics, 1992.

[7] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[8] M. Matsumoto, Mersenne Twister, http://www.math.sci.hiroshima-u.ac.jp/?m-mat/MT/emt.html, 2007.

[9] V. Podlozhnyuk, Parallel Mersenne Twister, http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/MersenneTwister/doc//MersenneTwister.pdf, 2007.

[10] S. Chandrasekaran and A. Amira, "High performance FPGA implementation of the Mersenne Twister," in *Proceedings of the 4th IEEE International Symposium on Electronic Design, Test and Applications (DELTA '08)*, pp. 482–485, January 2008.

[11] S. Banks, P. Beadling, and A. Ferencz, "FPGA implementation of Pseudo Random Number generators for Monte Carlo methods in quantitative finance," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 271–276, December 2008.

[12] X. Tian and K. Benkrid, "Mersenne Twister random number generation on FPGA, CPU and GPU," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '09)*, pp. 460–464, August 2009.

[13] P. L'Ecuyer and R. Simard, "TestU01: a C library for empirical testing of random number generators," *ACM Transactions on Mathematical Software*, vol. 33, no. 4, 22 pages, 2007.

[14] G. Marsaglia, Diehard Battery of Tests of Randomness, http://stat.fsu.edu/pub/diehard/, 1995.

[15] D. E. Knuth, *Seminumerical Algorithms, Volume 2 of The Art of Computer Programming*, Addison-Wesley, Reading, Mass, USA, 3rd edition, 1997.

[16] B. D. McCullough, "A review of TESTU01," *Journal of Applied Econometrics*, vol. 21, no. 5, pp. 677–682, 2006.

[17] A. Rukhin, J. Soto, J. Nechvatal et al., "A statistical test suite for random and pseudorandom number generators for cryptographic applications," http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/ SP800-22rev1a.pdf, Special Publication 800-22, Revision 1a, 2010.

[18] G. B. Robert, Dieharder: A Random Number Test Suite, http://www.phy.duke.edu/?rgb/General/dieharder.php, Version 3.31.0, 2011.

[19] D. B. Thomas, W. Luk, P. H. W. Leong, and J. D. Villasenor, "Gaussian random number generators," *ACM Computing Surveys*, vol. 39, no. 4, article 11, 2007.

[20] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610–611, 1958.

[21] A. Ghazel, E. Boutillon, J. L. Danger et al., "Design and performance analysis of a high speed AWGN communication channel emulator," in *Proceedings of the IEEE Pacific Rim Conference*, pp. 374–377, Citeseer, Victoria, BC, Canada, 2001.

[22] D.-U. Lee, J. D. Villasenor, W. Luk, and P. H. W. Leong, "A hardware Gaussian noise generator using the box-muller method and its error analysis," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 659–671, 2006.

[23] G. Marsaglia and W. W. Tsang, "The ziggurat method for generating random variables," *Journal of Statistical Software*, vol. 5, pp. 1–7, 2000.

[24] G. Zhang, P. H. W. Leong, D.-U. Lee, J. D. Villasenor, R. C. C. Cheung, and W. Luk, "Ziggurat-based hardware gaussian random number generator," in *Proceedings of the International*

Conference on Field Programmable Logic and Applications (FPL '05), pp. 275–280, August 2005.

[25] H. Edrees, B. Cheung, M. Sandora et al., "Hardware-optimized ziggurat algorithm for high-speed gaussian random number generators," in *Proceedings of the International Conference on Engineering of Recongurable Systems & Algorithms (ERSA '09)*, pp. 254–260, July 2009.

[26] N. A. Woods and T. Court, "FPGA acceleration of quasi-monte carlo in finance," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 335–340, September 2008.

[27] C. S. Wallace, "Fast pseudorandom generators for normal and exponential variates," *ACM Transactions on Mathematical Software*, vol. 22, no. 1, pp. 119–127, 1996.

[28] C. S. Wallace, MDMC Software—Random Number Generators, http://www.datamining.monash.edu.au/software/random/index.shtml, 2003.

[29] D. U. Lee, W. Luk, J. D. Villasenor, G. Zhang, and P. H. W. Leong, "A hardware Gaussian noise generator using the Wallace method," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 8, pp. 911–920, 2005.

[30] R. Korn, E. Korn, and G. Kroisandt, *Monte Carlo Methods and Models in Finance and Insurance*, Financial Mathematics Series, Chapman & Hull/CRC, Boca Raton, Fla, USA, 2010.

[31] L. Devroye, *Non-Uniform Random Variate Generation*, Springer, New York, NY, USA, 1986.

[32] J. A. Peter, An algorithm for computing the inverse normal cumulative distribution function, 2010.

[33] B. Moro, "The full Monte," *Risk Magazine*, vol. 8, no. 2, pp. 57–58, 1995.

[34] D.-U. Lee, R. C. C. Cheung, W. Luk, and J. D. Villasenor, "Hierarchical segmentation for hardware function evaluation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 1, pp. 103–116, 2009.

[35] D.-U. Lee, W. Luk, J. Villasenor, and P. Y. K. Cheung, "Hierarchical Segmentation Schemes for Function Evaluation," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '03)*, pp. 92–99, 2003.

[36] IEEE-SA Standards Board. IEEE 754-2008 Standard for Floating-Point Arithmetic, August 2008.

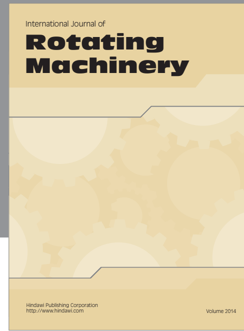
[37] Free Software Foundation Inc. GSL—GNU Scientic Library, http://www.gnu.org/software/gsl/, 2011.

[38] S. L. Heston, "A closed-form solution for options with stochastic volatility with applications to bond and currency options," *Review of Financial Studies*, vol. 6, no. 2, p. 327, 1993.

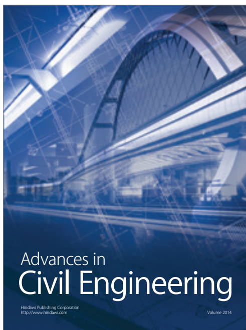
[39] S. S. Shapiro and M. B. Wilk, "An analysis-of-variance test for normality (complete samples)," *Biometrika*, vol. 52, pp. 591–611, 1965.

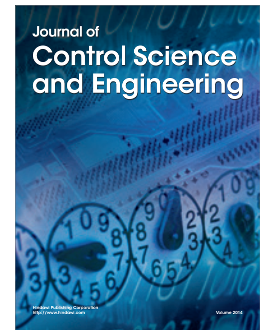
[40] M. A. Stephens, "EDF statistics for goodness of fit and some comparisons," *Journal of the American Statistical Association*, vol. 69, no. 347, pp. 730–737, 1974.

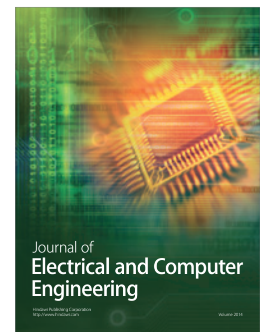
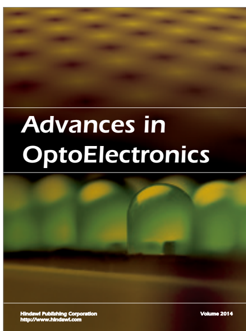
Journal of
Engineering

The Scientific
World Journal

International Journal of
Rotating
Machinery

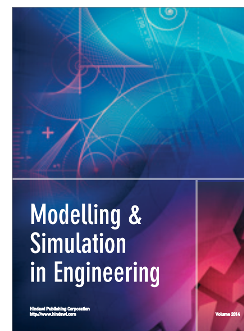
Journal of
Sensors

International Journal of
Distributed
Sensor Networks

Advances in
Civil Engineering

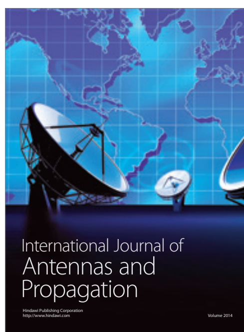
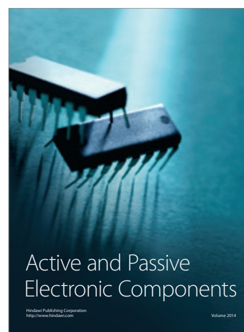
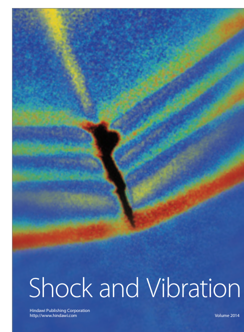
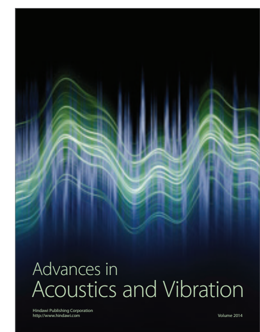
Hindawi
Submit your manuscripts at
http://www.hindawi.com

Journal of
Control Science
and Engineering

Journal of
Robotics

Journal of
Electrical and Computer
Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and
Observation

Modelling &
Simulation
in Engineering

International Journal of
Aerospace
Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and
Propagation

Active and Passive
Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration