Hardware-Optimized Ziggurat Algorithm for High-Speed Gaussian Random Number Generators

Hassan M. Edrees, Brian Cheung, McCullen Sandora, David Nummey, Deian Stefan
S*ProCom2 // The Cooper Union for the Advancement of Science and Art
51 Astor Place
New York, NY 10003
{edrees, cheung4, sandor, nummey, stefan}@cooper.edu

Abstract

Many scientific and engineering applications, which are increasingly being ported from software to reconfigurable platforms, require Gaussian-distributed random numbers. Thus, the efficient generation of these random numbers using few resources and allowing for high clocking rates is an important design factor in the application performance. In this paper, we demonstrate scalable implementations of the Ziggurat algorithm, a Gaussian random number generator, which we have modified for optimal performance on the Xilinx Virtex-4 FX12 FPGA. The resource-efficient design uses a small number of slices (233) while delivering a high throughput of 240 million samples per second. A two-way parallelizable design is discussed and the estimated throughput scales almost linearly. The generation of multiple Gaussian random numbers per cycle allows for the implementation of multiple, concurrent simulations on FPGAs with minimal resource overhead.

Keywords: RNGs, FPGA, Reconfigurable Hardware

1. Introduction

The Gaussian distribution has many important analytical properties that make it a standard choice for modeling and approximating systems for which little knowledge is available [7], [1]. Two examples from communication theory that rely on the use Gaussian-distributed random numbers include 1) modeling of communication channel imperfections as additive white Gaussian noise (AWGN) [18], and 2) simulating multipath fading channels using low-pass Gaussian noise sources [19]. In machine learning and data mining applications, it is very common to assume the distribution of the target data (conditioned on the inputs) as Gaussian, or the weighted sum of multiple Gaussian processes [1]; for example, keystroke durations can modeled as Gaussian-distributed variables [6]. Furthermore, Gaussian random numbers find application in numerical integration; for example, solutions to parabolic differential equations, which include the diffusion equation and Schrödinger's wave equation, involve integrating over a Gaussian. Very often, such integrals cannot be solved analytically and the only means of obtaining a solution involves sampling the Gaussian arising in the problem [16].

Unlike the generation of uniform random numbers, which can be implemented using linear methods and are therefore easily scaled to hardware, generating Gaussian random numbers (GRNs) is a non-linear, thus more complex process. The most common methods of generating GRNs involve the transformation of multiple uniform random numbers. In this paper we focus on one of the fastest of such methods in software, the Ziggurat method, which we have modified for efficient implementation on Field-Programmable Gate Arrays (FPGAs).

This paper is organized as follows. We introduce Gaussian random number generators and various implementation methods, followed by a detailed description of the Ziggurat method and our hardware-oriented modification to the algorithm. We then describe the architectural and algorithmic optimizations used and finally present the implementation details and results.

2. Background and Related Work

Gaussian Random Number Generators (GRNGs) are used in a variety of simulation environments, and the need for faster implementations has increased drastically with the advent of computationally-driven research. These fields range from communication systems to artificial intelligence to finance.

A number of general algorithms have been created for the efficient generation of Gaussian-distributed

random numbers given a uniformly-distributed random number as input. These methods can be divided into four general techniques: 1) Inversion of the cumulative distribution function (CDF), 2) Transformation, 3) Recursion, and 4) Acceptance-rejection.

2.1. CDF Inversion

The CDF inversion method is the most direct approach to generating Gaussian-distributed random numbers. The CDF, $\Phi(t)$, is defined as:

$$\Phi(t) = \int_{-\infty}^{t} f(x)\ \mathrm{d}x = \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{t-\mu}{\sigma\sqrt{2}}\right)\right). \quad (1)$$

Here $f(t)$ is the probability density function (PDF) of the Gaussian distribution with mean μ and variance σ^2:

$$f(t) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(t-\mu)^2}{2\sigma^2}\right). \quad (2)$$

This method simply takes a uniformly-distributed random number, $U_1 \in [0, 1)$, and calculates the probability of such a number occurring through the inversion of the CDF, $\Phi^{-1}(U_1)$ [21]. Theoretically, exact precision from this method can be achieved through numerical integration, however at a very high computational cost. Thus, practical implementations usually involve the use of polynomial approximations [2].

2.2. Transformation

Box and Muller proposed a method which exploits the wide availability of library functions for the calculation of trigonometric functions, square roots, and natural logarithms on general-purpose processors. Unlike the previous technique, this method provides flexibility and high precision, particularly in the tail of the distribution [2]. The Box-Muller Transform takes two uniformly-distributed random samples, U_1 and U_2, and transforms them to two Gaussian-distributed random samples, Z_1 and Z_2, with mean $\mu = 0$ and variance $\sigma^2 = 1$ according to [2]:

$$Z_1 = \sqrt{-2\ln U_1}\cos(2\pi U_2) \quad (3)$$

$$Z_2 = \sqrt{-2\ln U_1}\sin(2\pi U_2). \quad (4)$$

2.3. Recursion

A recursion method generates new Gaussian-distributed random numbers from linear combinations of previously generated Gaussian-distributed random numbers, taking advantage of the fact that the sum of two Gaussian random variables is also Gaussian-distributed [8], [21]. An example of a recursion method is the Wallace random number generator (RNG), which takes a vector X of length K such that each element of X is an independent random number from the Gaussian distribution. These K numbers are then normalized so that their average squared value is 1. X is then multiplied by an orthogonal matrix A to generate a new vector of Gaussian numbers $X' = AX$. This process is then repeated R times to generate $X^{(n)} = AX^{(n-1)}$, $n = 2, \ldots, R$. Although this repetition decreases the correlation between the output samples, the method is inexact [21].

2.4. Acceptance-Rejection

An acceptance-rejection method involves conditionality in generating Gaussian-distributed random numbers: the generation of random numbers is done by accepting correct random values (i.e., those part of the target distribution) and rejecting incorrect ones. The Ziggurat method described in this paper is an example of an acceptance-rejection method. Unlike the inversion method described earlier, this method does not require the calculation of the CDF or its inverse, thereby eliminating the need for inefficient numerical methods when the CDF is not expressible in closed form (i.e., the CDF of the Gaussian distribution) [8]. Although the Ziggurat method is inexact, highly accurate results can be achieved with increasing computational resources.

This paper details the hardware implementations of a modified, inexact Ziggurat method.

3. The Ziggurat Method

The Ziggurat method (or algorithm) is among the fastest and most efficient methods for generating GRNs, and is the default generator in various statistical and scientific libraries and applications [5], [15]. Following [14], we define:

$$\mathcal{C} = \{(x, y) : y = f(x)\}, \quad (5)$$

a set of the points making up the area under the Gaussian distribution $f(x)$, and a subset $\mathcal{Z} \subset \mathcal{C}$ of C blocks B_i:

$$\mathcal{Z} = \bigcup_{i=0}^{C} B_i, \quad (6)$$

where the area of each B_i is v. As shown in Figure 1, blocks B_i, $i > 0$ are rectangles horizontally extending from $x = 0$ to x_i and vertically extending from $f(x_i)$ to $f(x_{i+1})$. This is highlighted by the darker block B_i of Figure 1. The bottom-most block B_0 extends to $x = \infty$ and is composed of a rectangle and the 'tail' of the Gaussian.

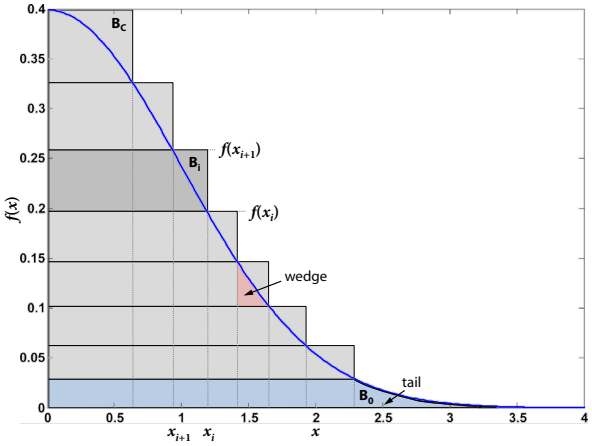

Figure 1. Ziggurat blocks partitioning the Gaussian distribution.

3.1. Original Algorithm

Following the acceptance-rejection approach, a random number z is generated such that $z \in \mathcal{Z}$, and is *accepted* if $z \in \mathcal{C}$, otherwise it is *rejected*. This method, following [4], is detailed in Algorithm 1. It is important to note that the edges of the rectangle x_i and $f(x_i)$ can be pre-computed and stored in memory (lines 1.1, 1.5-6, 1.12-13) for greater efficiency. Moreover, the exponential function only needs to be evaluated to determine whether z is in the wedge (lines 1.10-14); the wedge area is shown in Figure 1.

Algorithm 1: Ziggurat Algorithm

input : Number of blocks C
output: Gaussian random number z

1.1 Generate points $f_i = f(x_i)\ i = 0, \ldots, C$;
1.2 **while true do**
`// Choose a random block`
1.3 $i \leftarrow$ random index with probability $1/C$;
1.4 $U_0 \leftarrow \mathcal{U}(0,1)$; `// uniform random number`
1.5 $z \leftarrow U_0 x_i$;
1.6 **if** $z < x_{i+1}$ *and* $0 < i \leq C$ **then**
1.7 **return** z; `// In bulk`
1.8 **else if** $i = 0$ **then**
1.9 **return** z from tail;
1.10 **else**
1.11 $U_1 \leftarrow \mathcal{U}(0,1)$;
1.12 $y \leftarrow U_1 |f_i - f_{i+1}|$;
1.13 **if** $y < f(z) - f_i$ **then**
1.14 **return** z; `// In wedge`
1.15 **end**

3.2. Modified Algorithm

In the original Ziggurat method, the probability distribution function (PDF) of the Gaussian distribution must be calculated in the wedge region when $U_0 x_i \geq x_{i+1}$ (line 1.10-14). Based on the average of the ratios $(x_i - x_{i+1})/x_i$, the PDF of the Gaussian distribution must be calculated roughly 2% of the time. Traditionally, in hardware this requires the use of a Coordinate Rotation Digital Computer (CORDIC) block. However, a finite power series expansion can be used to reduce the problem to that of multiplication. Because the use of a CORDIC block is impractical for resource-constrained implementations, we consider a power series expansion. When incorporating this approximation, however, it is important to consider that the precision of a power series degrades in representing values increasingly far away from zero as the degree of the polynomial decreses. Following (2), we note that a point value is squared and negated before being exponentiated. Thus, proper error bounding needs to be considered to ensure good convergence. The largest point value for a ziggurat with $C = 256$ blocks is 3.64, therefore the quantity in the exponential could potentially be as large as 6.7. Using the normal Taylor series approximation about zero would require the first 20 terms to be retained in order to achieve accuracy to 2 decimal places, which would ultimately amount to 96 multiplications. This, too, is unreasonable for a resource-efficient implementation of the Ziggurat method.

In order to greatly reduce computational complexity, the exact Gaussian curve was replaced by a piecewise linear curve resembling stacked horizontal trapezoids, as shown in Figure 2. With this simplification only a single add and multiply are needed. Each trapezoid is characterized by three parameters, m_i, f_i and b_i , which can be thought of as the slope, base length and top length, respectively. Initial observations suggest that the base length and top length should simply lie on the distribution curve. However, this is unacceptable because the Gaussian has an inflection point at $x = 1$. The Gaussian curve between $0 \leq x < 1$ is concave-down; trapezoids below this point will always be underneath the curve and consequently, the area will be less than that of the Gaussian PDF. This would result in valid points being rejected in this region. Moreover, for $x > 1$, the Gaussian curve is concave-up, meaning trapezoidal approximations would always lie above the curve and result in invalid points being accepted. The combination of these errors leads to a skewing of the retained points which, among other errors, increases the mean by a significant amount. Care needs to be

taken so that each horizontal slice possesses equal area. This was achieved in our implementation by ensuring the bottom base of each trapezoid was on the curve, guaranteeing that all the slopes would be negative (otherwise, the so that all slopes would be guaranteed negative (otherwise, the Ziggurat method would need heavy modifications to treat curves that are not monotonically decreasing). Thus, the only free parameter was the top length of each trapezoid. Subject to the same area condition, these lengths were constrained as:

$$b_{i+1} = \frac{2}{y_{i+1} - y_i} \int_{y_i}^{y_{i+1}} g(y)\mathrm{d}y - x_i, \qquad (7)$$

where $g(y)$ is the inverse of the Gaussian function (2), and for clarity $y_k = f(x_k)$. From the above, the second parameter m_i is computed as $m_i = (y_i - y_{i+1})/(x_i - b_{i+1})$. Algorithm 2 shows these modifications from the original.

Algorithm 2: Modified Ziggurat Algorithm

input : Number of block C
output: Gaussian random number z

2.1 Generate f_i, b_i, m_i $i = 0, \ldots, C$;
2.2 **while true do**
2.3 $i \leftarrow$ random index with probability $1/C$;
2.4 $U_0 \leftarrow \mathcal{U}(0,1)$; `// uniform random number`
2.5 $z \leftarrow U_0 x_i$;
2.6 **if** $z < x_{i+1}$ *and* $0 < i \leq C$ **then**
2.7 **return** z; `// In bulk`
2.8 **else if** $i = 0$ **then**
2.9 **return** z from tail;
2.10 **else**
2.11 $U_1 \leftarrow \mathcal{U}(0,1)$;
2.12 $y \leftarrow U_1 |f_i - f_{i+1}|$;
2.13 **if** $y < m_i(z - x_i)$ **then**
2.14 **return** z; `// In wedge`
2.15 **end**

4. Optimizations

In its simplest form, the overall architecture is composed of a main unit and a tail unit, which compute GRNs from blocks in the main region (B_i, $i > 0$) and the tail region (B_0) of the Gaussian PDF, respectively. Various algorithmic and architectural optimizations for a hardware implementation of the modified Ziggurat algorithm are discussed in the following sections.

4.1. Architectural Optimizations

As discussed in previous sections, our algorithm has contingencies for the output random number to lie

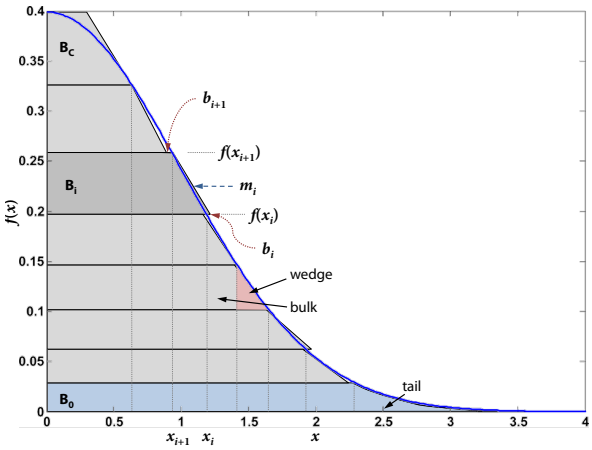

Figure 2. Modified Ziggurat with trapezoids partitioning the distribution.

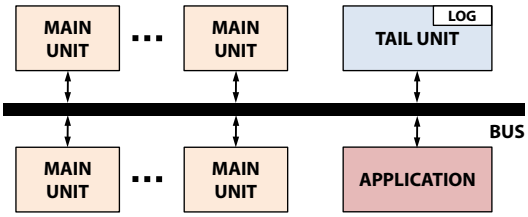

Figure 3. Architecture generating multiple Gaussian random numbers.

in three distinct regions of the Gaussian probability distribution curve, namely the bulk, wedge and tail regions (Figure 2), which correspond to the three different conditions on lines 2.6, 2.8, and 2.10.

The section of the algorithm generating random numbers from the tail region is independent of that for the bulk and wedge regions. Therefore, the design can be split into two modules which generate random numbers from the bulk/wedge and from the tail (denoted *main unit* and *tail unit* for convenience) in parallel. Because the majority of the Gaussian random numbers (over 98%) are generated from the main unit, the tail unit can be clocked at lower rates to save power, be implemented as a folded design to save resources [17], or implemented in software running on an embedded processor. Furthermore, for applications requiring multiple Gaussian random numbers, several main units can share one tail unit, which buffers the tail output until a number is requested by one of the main units (line 2.8) in our design. In addition, due to the very high throughputs and parallelizable architectures of uniform random number generators (URNGs) [3], [20], a common URNG is used to feed the main and tail units, thus keeping resource usage low. Figure 3 shows the overall architecture.

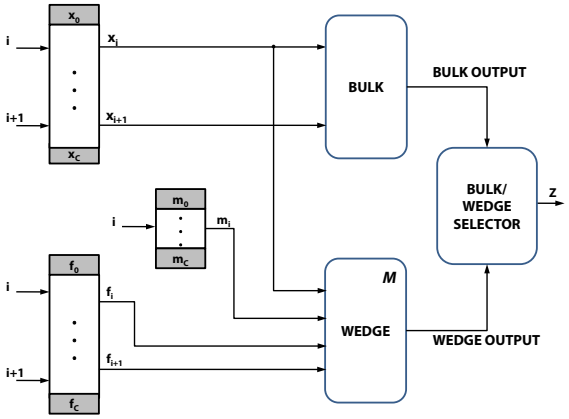

Figure 4. Direct design of the main unit.

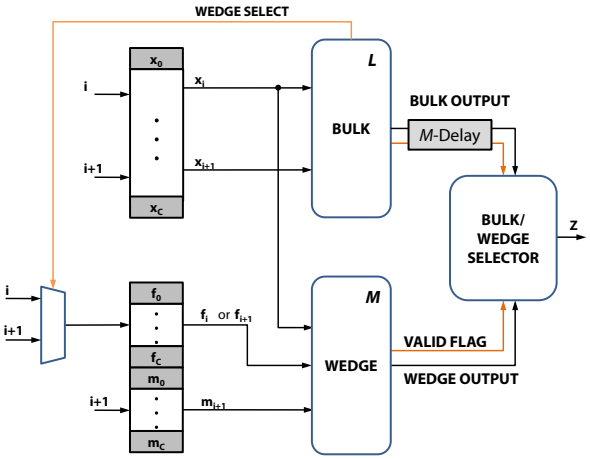

Figure 5. Main unit with wedge feedback control.

4.2. Main Unit

The most straight-forward approach to implementing our modified algorithm involves merely translating the existing algorithm to hardware, as shown in Figure 4. Since the design is a feed-forward or streaming architecture, it can easily be pipelined [17]. However, this approach suffers from excessive memory usage (3 Block RAMs for $C = \{128, 256\}$), and thus cannot be scaled efficiently for laying out multiple main units.

A direct mapping of the algorithm would require simultaneous access to the coefficients $x_i, x_{i+1}, f_i, f_{i+1}$, and m_i. Taking advantage of the dual-port Block RAMs available on FPGAs, the coefficients can be interleaved such that the first port accesses x_i and the second accesses x_{i+1} (this is identical for the f_i and f_{i+1} coefficients), while the m_i coefficients would require a single port Block RAM. Due to the high input/output requirements from RAM, a Block RAM must be dedicated to x_i's, f_i's and m_i's, if a random number is to be computed during each cycle. For $C < 512$, the direct design makes inefficient use of the Block RAMs holding the x_i and f_i 32-bit fixed-point coefficients; a Block RAM can hold up to 18Kbits, so for $C = \{128, 256\}$, as in our implementations, less than half of each Block RAM is occupied.

Considering the areas of the wedges in Figure 2 are substantially smaller than their corresponding bulk areas, the number of points needed from the bulk is much higher. Consequently, most of the points generated from the wedge are discarded. Then, regarding a wedge output as a contingency, it is possible to implement the architecture with the bulk portion of the main unit running continuously and being interrupted only when the condition for a wedge output is true (line 2.10). Thus, we propose a RAM-efficient design where the m_i and f_i coefficients are interleaved, reading x_i, x_{i+1}, f_i and m_i during each cycle, and only stalling the pipeline to read f_{i+1} when a wedge point is required (lines 2.10-14).

In order to circumvent stalling the whole system and flushing the pipeline, the coefficients used as inputs to the wedge block are the same as those in the bulk. However, the inputs to the wedge block are delayed by L cycles, the latency of the bulk block. The inputs to the Block RAMs are also delayed by $L+1$ cycles, so if the output of the bulk is invalid, the f_{i+1} coefficient is read on the next cycle. Because x_i, x_{i+1}, f_i and m_i were previously read and buffered (due to the L-delay input to the wedge block), a point from the wedge can be generated thusly. It is important to note that none of the pipeline stages in the bulk are stalled, but a flag indicating the status of the output is percolated through the system; only the input to the Block RAMs (the new i) has to be stalled or discarded. Figure 5 illustrates this modified design.

Finally, since the outputs from the bulk and the wedge has a corresponding bit flag indicating its validity, the bulk output and flag can each be delayed by M, the latency of the wedge block. Thus, the final output from the main unit can be chosen from the bulk or wedge; their latencies matching.

4.3. Tail Unit

In the event that $i = 0$ (line 2.8), a GRN must be chosen from the tail of the distribution. Unlike the other regions of the Ziggurat, which were divided into rectangles, the tail region requires a non-linear approximation. The generation of tail outputs in the Ziggurat method was implemented using a new algorithm developed by Marsaglia [13], [21]. The most computationally-intensive part of computing a number from the tail region is the evaluation of the logarithm.

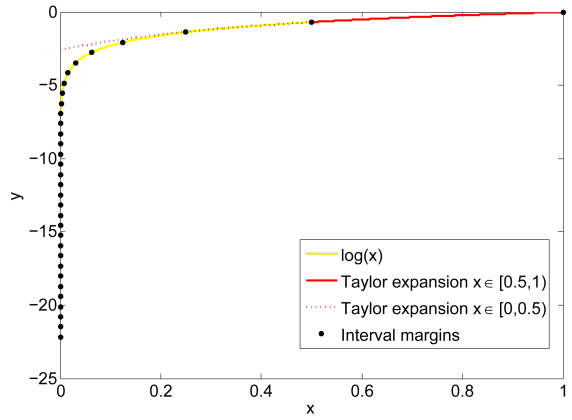

Figure 6. Taylor series expansion of the logarithm used for the tail, with range reduction method highlighted. Regions for which the 7^{th} order Taylor series inaccurately represents (shown in blue) are scaled and recast into the accurate $x \in [0.5, 1)$ range (shown in red).

While the logarithm cannot be approximated by a piecewise linear approximation, due to the limited precision of fixed-point numbers, an 'exact' determination could be achieved using a CORDIC unit. Alternatively, it could be estimated by a Taylor expansion. The former, though more accurate, would increase the number of slices required by the design more than threefold, while the latter would need a large number of polynomials to adequately represent the whole range of $x \in [0, 1)$. A suitable approximation is attained for the range $x \in [0.5, 1)$ by a 7^{th} order expansion having mean and maximum error of 0.0916×10^{-3} and 0.8854×10^{-3}, respectively. A range reduction method, as presented in [10], is implemented to rescale the inputs to the log unit to the interval $[0.5, 1)$ according to:

$$\log(x) = \log(2^a x) - \log(2^a), x \in \left[\frac{1}{2^a}, \frac{1}{2^{a-1}}\right), \quad (8)$$

where $a \geq 1$, $\log(2^a x) \in [0.5, 1)$, and $\log(2^a)$ is a constant. The above procedure divides $\log(x)$ into 32 sub-intervals, whose edges are multiples of 2, as shown in Figure 6, so that the inputs to the log unit can be rescaled to $[0.5, 1)$ using one bitwise-shift operation. For illustration, we can calculate $\log(x)$ using $\log(2x) = \log(2) + \log(x)$ where $2x \in [0.5, 1)$. Similarly for $x \in [0.125, 0.25)$, $\log(x) = \log(4x) - \log(4)$, where $4x \in [0.5, 1)$.

5. Implementations and Discussion

We implemented the direct, feed-forward design and the RAM-efficient design on the Xilinx Virtex-4 FX12 FPGA (XC4VFX12-FF668, speed grade 10C), hosted on the ML403 development board. The design and implementation were conducted with Xilinx's System Generator package for MATLAB and synthesized using Synplify Pro; we prefer this to traditional hardware description languages (HDLs) as it speeds up the implementation process with minimal performance loss when compared to direct HDL implementations. The arithmetic and coefficients stored in the Block RAMs were all 32-bit fixed-point numbers. The direct implementation ($C = 128$) was clocked at 178.799MHz, and the RAM-efficient design was clocked at 240.442MHz, while using fewer number of slices, and fewer Block RAMs. Table 1 summarizes the results for $C = \{128, 256, 512\}$. The RAM-efficient module outperformed the direct module in terms of speed, slice count and Block RAM usage.

Design	Freq (MHz)	Slices	DSP Slices	Block RAMs	Samp/ sec
Modified Ziggurat					
Direct-128	178.80	262	12/32	3/36	179M
Direct-256	168.52	263	12/32	3/36	169M
Direct-512	182.15	264	12/32	4/36	182M
RAM-Eff-128	240.44	233	12/32	2/36	240M
RAM-Eff-256	241.31	234	12/32	2/36	241M
RAM-Eff-512	231.59	234	12/32	4/36	231M
Approx-2-MU	200.00	500	24/32	4/36	400M
Previous GRNG hardware implementations					
Ziggurat [22]	170	891	2/120	4/120	169M
Wallace [11]	155	770	6/120	4/120	155M
Box-Muller [9]	133	2514	8/120	2/120	133M
Box-Muller [12]	233	1528	3/120	12/120	466M

Table 1. Resource and Performance Evaluation

The tail unit implementation is maximally clocked at 103.445MHz, taking up 1,024 slices and 12 DSP blocks. Although the tail unit uses a large number of slices, it is shared among multiple applications due to the relaxed throughput demand. Furthermore, the design can be folded so that a RAM block is used to store the multiplier coefficients with a common multiplier for the Taylor expansion, thus taking multiple cycles to complete an iteration. In more resource-constrained implementations, the logarithm can be evaluated in software using the PowerPC core on Virtex-4 FPGA's with a minimal performance loss.

Table 1 compares our implementation with previous hardware Gaussian random number generators, including a Ziggurat-based implementation [22]. Although the overall slice count (tail unit + main unit) is slightly higher than the implementations in [22] and [11], the throughput is 42% higher. We note that while the performance of our RAM-efficient implementation is comparable to previous FPGA implementations, it has been designed to be scaled for the generation of

multiple-Gaussian random numbers. This is illustrated in Table 1, which also shows an estimate of a 2-main unit parallel implementation (2-MU); because the slice count of the RAM-efficient design is very low and only a miniscule of additional control logic is needed, we conjecture that the 2-MU parallel implementation scales linearly. This design is comparable in the number of resources (Block RAMs + slices) used, with a much higher throughput (excepting the Box-Muller method presented in [12]). Such a design would be targeted for the DSP-enhanced FPGAs containing a large number of multipliers (e.g Virtex 4SX series). Furthermore, the high quality of the generator has been confirmed using the Kolmogorov-Smirnov test and the χ^2 goodness-of-fit test ($p = 0.9961$).

6. Conclusion

A scalable architecture of the Ziggurat algorithm for hardware generation of Gaussian random numbers was presented for high-speed scientific and engineering simulations. Throughputs on the order of 240 million samples per second were achieved and estimates for further parallelization scale almost linearly to 400 million samples per second. Future work includes the redesign of the tail unit to a resource-efficient folded design and the implementation of a highly-parallelized design on DSP-enhanced FPGAs for the generation of multiple Gaussian random numbers per cycle.

References

[1] C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] G.E.P. Box and M.E. Muller. A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[3] I.L. Dalal and D. Stefan. A hardware framework for the fast generation of multiple long-period random number streams. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays (FPGA 2008).*, pages 245–254. ACM New York, NY, USA, 2008.

[4] J.A. Doornik. An improved Ziggurat method to generate normal random samples. Technical report, working paper.

[5] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. GNU Scientific Library Reference Manual, Revised Second Edition (v1.8). *Network Theory Ltd*, 2006.

[6] J. Ilonen. Keystroke dynamics. *Advanced Topics in Information Processing–Lecture*, 2003.

[7] S.M. Kay. *Fundamentals of Statistical Signal Processing, Volume 2: Detection Theory*. Prentice Hall PTR, 1998.

[8] V. Krishnan. *Probability and random processes*. Wiley-Interscience, 2006.

[9] D.U. Lee, W. Luk, J. Villasenor, and PYK Cheung. A hardware Gaussian noise generator for channel code evaluation. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 69–78, 2003.

[10] D.U. Lee, W. Luk, J.D. Villasenor, and P.Y.K. Cheung. A Gaussian Noise Generator for Hardware-Based Simulations. *IEEE Transactions on Computers*, pages 1523–1534, 2004.

[11] 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*, 13(8):911–920, 2005.

[12] 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*, pages 659–671, 2006.

[13] G. Marsaglia and W.W. Tsang. A Fast, Easily Implemented Method for Sampling from Decreasing or Symmetric Unimodal Density Functions. *SIAM Journal on Scientific and Statistical Computing*, 5:349, 1984.

[14] G. Marsaglia and W.W. Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7, 2000.

[15] C.B. Moler. *Numerical Computing with* MATLAB. Society for Industrial Mathematics, 2004.

[16] B.K. Øksendal. *Stochastic Differential Equations: An Introduction With Applications*. Springer, 2003.

[17] K.K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. Wiley-Interscience, 1999.

[18] J.G. Proakis. *Digital Communications*. Osborne-McGraw-Hill, 2001.

[19] T.S. Rappaport. *Wireless communications*. Prentice Hall PTR, Upper Saddle River, NJ, 2002.

[20] D.B. Thomas and W. Luk. FPGA-optimised high-quality uniform random number generators. In *Proceedings of FPGA 2008*, pages 235–244. ACM New York, NY, USA, 2008.

[21] D.B. Thomas, W. Luk, P.H.W. Leong, and J.D. Villasenor. Gaussian random number generators. 2007.

[22] 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*, pages 275–280, 2005.