ZIGGURAT-BASED HARDWARE GAUSSIAN RANDOM NUMBER GENERATOR

Guanglie Zhang, Philip H.W. Leong [*] *Dong-U Lee*[†], *John D. Villasenor* [‡] *Ray C.C. Cheung*[§], *Wayne Luk* [¶]

Department of Computer Sci. & Eng.,
The Chinese University of Hong Kong,
Shatin, NT, Hong Kong
{glzhang, phwl}@cse.cuhk.edu.hk

Electrical Engineering Department,
University of California,
Los Angeles, USA
{dongu, villa}@icsl.ucla.edu

Department of Computing,
Imperial College London,
London, UK
{r.cheung, w.luk}@imperial.ac.uk

ABSTRACT

An architecture and implementation of a high performance Gaussian random number generator (GRNG) is described. The GRNG uses the Ziggurat algorithm which divides the area under the probability density function into three regions (rectangular, wedge and tail). The rejection method is then used and this amounts to determining whether a random point falls into one of the three regions. The vast majority of points lie in the rectangular region and are accepted to directly produce a random variate. For the non-rectangular regions, which occur 1.5% of the time, the exponential or logarithm functions must be computed and an iterative fixed point operation unit is used. Computation of the rectangular region is heavily pipelined and a buffering scheme is used to allow the processing of rectangular regions to continue to operate in parallel with evaluation of the wedge and tail computation. The resulting system can generate 169 million normally distributed random numbers per second on a Xilinx XC2VP30-6 device.

1. INTRODUCTION

Gaussian random number generators (GRNG) are used in a large number of computationally intensive modeling and simulation applications including heat transfer [1], communications systems [2], and evolutionary programming [3]. Given their importance, there has been surprisingly little research on their efficient hardware implementation. McCollum et al. used a lookup table followed by linear interpolation to compute the inverse cumulative distribution function in order to generate random variates with arbitrary distribution [4]. Lee et al. [2] proposed using the Box-Muller algorithm [5] to generate normally distributed random numbers. The elementary functions involved in its implementation are performed using non-uniform piecewise polynomial approximation. The Wallace method [6] involves transforming a pool of random numbers to a new pool of random numbers. Lee et al. made a hardware implementation of the Wallace method and showed the speed and size advantages over the Box-Muller method [7].

We propose the Ziggurat method [8] as an efficient algorithm to use in a GRNG. To date, no hardware implementations of this method have been reported. The Ziggurat algorithm allows fast integer operations to be used to produce most of its outputs in a single cycle. For a small percentage of outputs, elementary functions such as the natural logarithm and exponential function need to be computed. These are implemented using polynomial approximations using an arithmetic logic unit (ALU) and state machine. The resulting system can generate 169 million normally distributed random numbers per second on a Xilinx XC2VP30-6 device. The key contributions of this paper are:

We develop a 5-stage fully-pipelined architecture for the Ziggurat method that combines a single-cycle parallel datapath for the common cases, and a buffered sequential ALU supporting elementary function evaluation for the infrequent cases.

We advocate the Tausworthe uniform random number generator (URNG) for custom computing machines that perform simulation since it has strong theoretical support and better quality than the more commonly used linear feedback shift register. This paper shows that the hardware implementation of the Tausworthe generator is fast and has small area; furthermore, it has superior performance when evaluated using the DIEHARD random number testsuite.

We propose an on-chip ^{2} test circuit which allows the distribution of random numbers to be monitored at runtime. This greatly reduces the bandwidth for full speed testing of GRNGs and allows continuous quality checking of the GRNG.

We demonstrate that, for applications where accurate modeling of the tails of the Gaussian distribution are

[*]Research partially supported by the H.K. Government RGC Grant, Ref. No. CUHK4333/02E.

[†]Research partially supported by the Jet Propulsion Laboratory, NASA.

[‡]Research partially supported by the U.S. Office of Naval Research.

[§]Research partially supported by the Croucher Foundation.

[¶]Research partially supported by the U.K. EPSRC Grant, number GR/N 66599 and GR/R 31409.

required, our method is smaller and faster than all previously reported designs (Table 3).

The remainder of this paper is organized as follows. Section 2 describes the Ziggurat method for generating Gaussian distributed random numbers. Section 3 provides methods for evaluating elementary functions using polynomial approximations. Section 4 presents the architecture of our Gaussian random number generator. Section 5 reports results and Section 6 draws conclusions.

2. THE ZIGGURAT METHOD

The Ziggurat method uses the rejection method to generate a random variate from an arbitrary decreasing probability density function. Our description of the Ziggurat method in this section follows the notation of Marsaglia [8].

The rejection method for generating a random variate can be described as follows. Let $=$ ( ) be a function with finite area, $\mathcal{C}$ be the set of points ( ) under the curve and $\mathcal{Z}$ be a finite area superset of $\mathcal{C}$, i.e. $\mathcal{Z}$ $\mathcal{C}$. Random points ( ) are taken uniformly from $\mathcal{Z}$ until ( ) $\in \mathcal{C}$ and is returned as the random variate [5, 8]. The density of such an will be ( ), with a normalizing value that makes ( ) a probability density function (i.e. $\int$ ( ) $= 1$).

For a normal distribution, we use ( ) $=$ exp[$\frac{x^2}{2}$], 0. $\mathcal{C}$ is the area under this curve. The distribution is made two-sided by the introduction of a randomly chosen sign, and is scaled to ensure unit area as described above. $\mathcal{Z}$ is chosen as the union of sections, $_i$ (0), made up of (1) rectangles and a bottom strip which tails off to infinity. The rectangles and bottom strip are chosen so that they are all of equal area, and their right-hand edge is denoted $_i$. The leftmost rectangle, $_0$ is assumed to be empty and $_0 = 0$. The following pseudocode describes the complete Ziggurat algorithm (which shows the rectangle, wedge and tail regions):

```
01 INITIALIZATION
02     $n$ is the size of the $w$ and $k$ tables.
03     i = 0..n-1 and $r = x_{n-1}$.
04     $w_0 = 0.5^{32}v/f(r); k_0 = \lfloor 2^{32}rf(r)/v \rfloor$.
05     $w_i = .5^{32}x_i; k_i = \lfloor 2^{32}(x_{i-1}/x_i) \rfloor$.
06     $\{f_i\}$, where $f_i = e^{-i^2/2}$.
07     U(0,1) is a uniform random number generator over [0,1).
08 REPEAT
09     Generate a signed random 32-bit integer $j$.
10     Set index: $i \leftarrow j$ & $(2^n - 1)$. Set $x \leftarrow jw_i$.
11     IF $|j| < k_i$ THEN RETURN $x$. /* **rectangle** */
12     IF $i = 0$ THEN /* **tail** */
13         DO
14             Generate i.i.d. uniform (0,1) variates $u_1, u_2$.
15             $x \leftarrow -\ln(u_1)/r, y \leftarrow -\ln(u_2)$.
16         WHILE $u_2 + u_2 < u_1^2$.
17         RETURN $x > 0$ ? $(r + x) : -(r + x)$.
18     IF $(f_i + (f_{i-1} - f_i)U(0,1)) < e^{-x^2/2}$ /* **wedge** */
19         RETURN $x$.
20 UNTIL FALSE.
```

The values of $_i$ ($= 1$ 2 (1)) are needed for the tables in the hardware implementation and are determined by equating the area of each of the rectangles with that of the base region. If this area is , the equations are as follows:

$$v \quad = \quad x_i[f(x_{i-1}) - f(x_i)] = rf(r) + \int_r^\infty f(x)dx. \qquad (1)$$

In order to determine , a function ( ) is first defined as follows:

```
FUNCTION z(r)
    $x_{n-1} = r$.
    $v = rf(r) + \int_r^\infty f(x)dx$.
    FOR i = (n-2) DOWNTO 1
        $x_i = f^{-1}(v/x_{i+1} + f(x_{i+1}))$.
    RETURN $(v - x_1 + x_1 f(x_1))$.
```

The root of ( ) (i.e. the value of such that ( ) $= 0$) is found numerically, e.g. using the bisection method. The values for the $_i$ are then calculated from .

The random point is only accepted if it falls under the pdf curve, otherwise it is rejected. The probability of accepting a point, $_{accept}$ is given by:

$$P_{accept} \quad = \quad area(\mathcal{C})/area(\mathcal{Z}) = \frac{\int_{-\infty}^{\infty} e^{-x^2/2}dx}{2vn}. \qquad (2)$$

Finally, the probability that a point is not drawn from a rectangular region, $_{nrect}$ (i.e. it is a wedge, tail or rejected) can be calculated as follows:

$$\begin{aligned} P_{nrect} &= 1 - area(\text{rect})/area(\mathcal{Z}) \\ &= \sum_{i=1}^{n-2} x_i(f(x_i) - f(x_{i+1})) + x_{n-1}f(x_{n-1}) \qquad (3) \end{aligned}$$

3. ELEMENTARY FUNCTION EVALUATION

The fast and accurate computation of elementary functions are necessary for an efficient implementation of the Ziggurat method. In order to achieve high accuracy using minimal resources, polynomial approximations are used.

We consider the exponential and logarithm functions required in the Ziggurat algorithm and describe their implementation using polynomial evaluation as proposed in [9] for single precision floating point. A general polynomial $_n$( ) can be written as $_n$( ) $=$ $_n$ ^{n} $+$ $_{n-1}$ $^{n-1}$ $+$ $+$ $_1$ $+$ $_0 = \sum_{i=0}^n$ $_i$ ^{i} where is the degree of the polynomial, and $_i$ is the coefficient of the th term. A function is approximated over a specified input interval [ ]. If the input values are outside this interval, range reduction is employed. Thus the process for computing the exponential or logarithm function involves three steps: the reduction of the given argument to a related argument in a interval [ ], the computation of the exponential for the reduced argument , and the reconstruction of the desired function from its components. For the methods described below, the peak relative errors are 7 6 10^{-8} and 7 1 10^{-8} for the exponential and logarithm functions respectively if all the computations are done in single precision floating point [9].

In our implementation, we use 36-bit accumulators and 18 18 bit multipliers. The format of the two's complement fixed point fractions used as inputs to the multipliers are 3 integer bits and 15 fractional bits and the peak relative error is 2^{-15} for both the exponential and logarithm functions.

The range reduction for the exponential function is accomplished by the identities $^{X} = 2^{k} \ {}^{x} = {}^{k\ln 2} \ {}^{x} = {}^{x+k\ln 2}$. An integer is found such that the fraction is within a specified interval. Then $= \ + \ \ln 2$. As is restricted to the interval $0\ 5\ln 2 \quad x \quad +0\ 5\ln 2$, can be calculated by $= \quad \log_2 e + 0\ 5$ where represents the integer truncation function which returns the largest integer that is less than or equal to its argument. Following range reduction, the exponential function for reduced argument is computed according to the following formula ($\in [\ \frac{\ln 2}{2}\ \frac{\ln 2}{2}]$).

$$e^x \approx (1.9875691500 \cdot 10^{-4}x^5 + 1.3981999507 \cdot 10^{-3}x^4 \\ +8.3334519073 \cdot 10^{-3}x^3 + 4.1665795894 \cdot 10^{-2}x^2 \\ +1.6666665459 \cdot 10^{-1}x + 5.0000001201 \cdot 10^{-1})x^2 + x + 1.0 \tag{4}$$

To evaluate the natural logarithm of a given argument , the range reduction for the reduced argument is computed using the identities

$$\ln X = \ln(2^k x) = \ln x + \ln 2^k = \ln x + k\ln 2. \tag{5}$$

We first find the such that $= 2^k$ and in the range $[0\ 5\ 1)$. This is done by using a left shifter to find the first set bit in . The logarithm function for the reduced argument is then computed by the function approximation.

4. ARCHITECTURE

The $\mathcal{Z}$ used in the Ziggurat algorithm is designed so that random points falling in the rectangular region occur the vast majority of the time. Computation in these cases is extremely fast. For a small percentage of situations (1.5% for $= 256$ as calculated from (3)), wedge and tail region acceptance must be handled and the computation of elementary functions is necessary. Fig. 1 is a block diagram showing the main components of the GRNG. We use a pipelined datapath (stage 1-3) to compute the rectangular regions and an iterative operation unit (OU) is used to handle the wedge and tail regions. First-in-first-out (FIFO) buffers are used so that a number of wedge/tail computations can be queued for processing by the operation unit while new random numbers from rectangular regions are being generated. An optional histogram unit (HU) allows full speed testing of the design and can be used to apply a ^{2} test. The HU allows the quality of the GRNG to be continuously monitored.

Tausworthe Random Number Generator. Linear feedback shift registers (LFSR) are often used to generate uniform random numbers in hardware. While traditional LFSRs are sufficient for many purposes, the Tausworthe random number generator offers slightly superior randomness with modest hardware cost, so is preferable for applications such as ours in which extremely stringent noise quality standards are being applied [5, 10]. The Tausworthe URNG

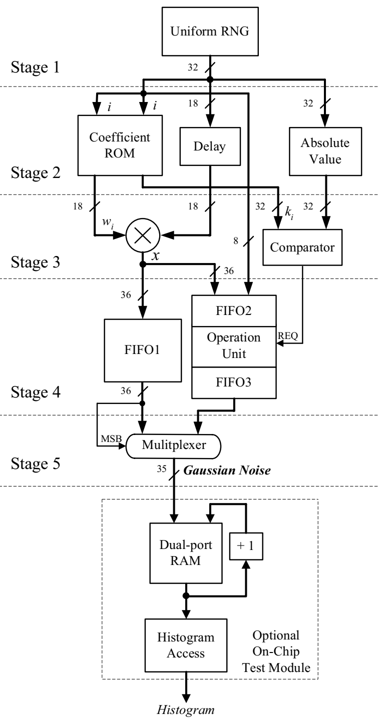

Fig. 1. Block diagram showing the architecture of the GRNG with optional on-chip test module.

combines three LFSR based random number generators to obtain improved statistical properties. A generator with period length 2^{88} is given in [10]. Fig. 2 shows the hardware implementation of the Tausworthe URNG.

Rectangular Region Datapath and FIFOs. In pseudocode (lines 9-11), it generates a random point and decides whether or not it is in the rectangular region. The computation involves a multiplication to get and comparison of $|\ |$ with $_i$. If the condition is true, the (otherwise unused) most significant bit (MSB) of is set and is written to the rectangular region output (FIFO1). This datapath should be made very efficient for the high acceptance rate of . Our pipelined design is shown in Fig. 1.

When the condition is false, the random point may correspond to a wedge or tail. Then a marker with the MSB being reset is written to FIFO1, and the value is written to FIFO2 which connects the rectangular region datapath with the OU. This buffering serves to decouple the computation of the rectangular regions, either a Gaussian random number or a marker indicating a wedge/tail is written to FIFO1 every clock cycle (unless it becomes full). The OU is responsible for processing inputs received via FIFO2 and must write its outputs to FIFO3. When the random point is from the tail region or accepted by the wedge region, the MSB of the value is also set and write to FIFO3.

When an output is read from the GRNG, a number is

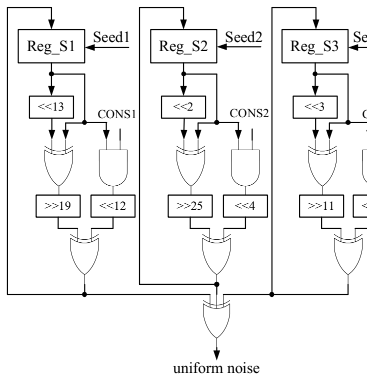

Fig. 2. Architecture of Tausworthe URNG.

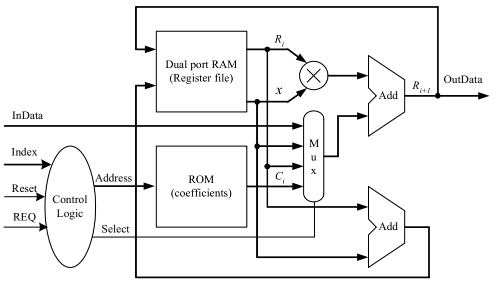

Fig. 3. Architecture of operation unit.

read from FIFO1 and its MSB is used to control a multiplexer. If the MSB is set, the value from FIFO1 is used, otherwise a read from FIFO3 is returned. Meanwhile, if the MSB of the data from FIFO3 is set, the ultimate output is valid random number, otherwise, it is a rejected trial for which no valid random number is generated (the probability of this occurring will be discussed in Section 5). Note that the MSB is used as a marker and does not form part of the random number.

Operation Unit (OU). The block diagram of the operation unit is shown in Fig. 3. It is organized as a register file and an ALU which includes two adders and a multiplier. In addition, a ROM, used to store polynomial coefficients to compute the elementary functions $\ln$ and $\exp$ is used. The OU is sequenced via a hardwired finite state machine.

Polynomial Evaluation. The OU must compute the $\ln$ and $\exp$ functions in lines 15 and 18 of the Ziggurat pseudocode respectively. This turns out to be the most computationally expensive part of the Ziggurat algorithm, but fortunately, since these correspond to the wedge and tail regions which occur with much lower probability than the rectangular regions, the speed with which one must compute these elementary functions is not so critical. As an example, for $= 256$, as used in our implementation, the combined probability of a wedge, tail or the sample being rejected is 1.5% (3) and so, the speed for these sections need only be 1.5% of the speed for a rectangular region.

The direct evaluation of a polynomial involves evaluating each monomial $_i\ ^i$ individually. This takes mul-tiplications for each monomial and additions, resulting in $(\ + 1)\ 2$ multiplications for a polynomial of degree . Horner's rule [11], as typically used in both software and hardware implementations, achieves better numerically stability and efficiency by factoring the terms as: $_n(\ ) = (((\ _n + \ _{n-1}) + \ _{n-2}) + \ + \ _1) + \ _0$. Computation starts from the innermost parentheses using the coefficients of the highest degree monomials and works outwards.

Control Logic with State Machine. We use a finite state machine to implement the control logic. One-hot encoding is employed to maximize speed and minimize implementation area. The state machine begins at "S_Start" and first waits for the REQ signal which corresponds to FIFO2 being non-empty. When the REQ signal is valid, the state machine makes a choice among the different state sequences for wedge and tail regions according to whether the INDEX signal is 1 or 0 respectively. The state machine generates the address of the coefficients ROM, read and write signals for the register file and the select signal for the multiplexor. In the case of a wedge (lines 18-19 in pseudocode), the function $^{-x^2/2}$ is evaluated in the "S_EXP" state and compared with $_i + [\ _{i-1}\ \ _i]$ in the "S_Wedge" state. If INDEX$=$ 0, we evaluate a tail region (lines 12-17 in pseudocode). The state machine will calculate the two logarithms and do the comparison in the "S_LOG" and "S_Tail" states respectively. Output from the wedge or tail computations is written to FIFO3 in the "S_End" state. After this operation, the state machine will settle in the "S_NULL" state to wait for another REQ signal to "S_Start" and for the next wedge or tail region.

In the histogram unit, the initial values of the RAM are all zero. The higher order bits of the GRNG output form the address of a dual-port RAM and the location corresponding to the generated output is incremented. The other port of the dual-port RAM can be used to access the histogram without interrupting the operation of the GRNG.

5. RESULTS

An implementation of the architecture described in Section 4 with $= 256$ is made on a Xilinx Virtex-II Pro XC2VP30-6FF896C FPGA and all results in this section are for this part unless otherwise specified. In order to achieve high performance, some special on-chip features of the Virtex-II FPGA are used, in particular, the SRL16 shift registers, delay locked loop, dedicated 18 18 bit hardware multipliers and dual-port block RAM resources. The design is written in VHDL and synthesized using Synplify Pro 7.3. Place and route is performed using the Xilinx ISE 6.2i.

In the subsections that follow, we will first compare the

Table 1. DIEHARD test results for Tausworthe and LFSR URNGs (failed tests in bold).

TEST	Tausworthe	LFSR
Birthday	0.908125	0.718022
OPERM5	0.659361	0.894649
Binary Rank (31×31)	0.782536	0.894649
Binary Rank (32×32)	0.357046	0.768956
Binary Rank (6×8)	0.324027	0.261791
Bitstream	0.598578	0.443253
OPSO	0.431957	0.571626
OQSO	0.492004	0.559064
DNA	0.432068	0.525910
Stream Count-the-1	0.459779	0.564513
Byte Count-the-1	0.609182	0.560009
Parking Lot	0.941697	0.460448
Minimum Distance	0.337831	**0.999999**
3D Spheres	0.952286	0.634016
Squeeze	0.113189	0.855206
Overlapping Sums	0.139815	0.717343
Runs Up	0.555513	0.575984
Runs Down	0.253845	0.552341
Craps	0.395120	0.841941

Tausworthe URNG and an LFSR-based URNG in terms of area utilization and the quality of the generated random numbers. We then describe an implementation of the GRNG on an FPGA, discuss its performance and compare it with previously published Box-Muller and Wallace implementations. In the last subsection, we present some statistical tests for the GRNG.

Tausworthe URNG. We compare the Tausworthe URNG with a maximum length LFSR-based URNG. The LFSR uses the primitive pentanomial $^{88} + \ ^{87} + \ ^{17} + \ ^{16} + 1$ over (2) and a random initial state. The 32-bit Tausworthe URNG uses 64 slices and can operate above 300MHz. The 88 tap LFSR can be implemented in 3.5 slices since each slice can implement a 34-bit shift register and an extra lookup table is required to compute the feedback. 32 of these are required to have a parallel 32-bit output so the resource utilization of the LFSR is 112 slices.

It is not possible to prove a sequence is random. However, the DIEHARD test developed by Marsaglia [12] is widely considered to be one of the most stringent URNG tests. Although the DIEHARD test suite is one of the most comprehensive publically available sets of randomness tests, unfortunately there are no well-defined pass criteria. Intel calculated that the entire 250 test suite passes with a 95% confidence interval for p-values between 0.0001 and 0.9999 [13], and this method is used for our testing.

The two outputs of the URNGs are compared with the DIEHARD testsuite. Results are shown in Table 1 with failed tests shown in bold. As it can be seen, the Tausworthe generator passes all tests whereas the LFSR fails the minimum distance test.

Table 2. Implementation results for GRNG on XC2VP30-6 and XC3S200-4 FPGAs (without test module).

	XC2VP30-6	XC3S200-4
SLICEs	868 out of 13,696 (6%)	908 out of 1,920 (47%)
Block RAMs	4 out of 136 (2%)	4 out of 12 (33%)
MULT18X18s	2 out of 136 (13%)	2 out of 12 (16%)
DCMs	1 out of 8 (12%)	1 out of 4 (25%)
Period of "CLK"	5.88ns (170MHz)	6.11ns (164MHz)
Period of "CLK2"	11.76ns (85MHz)	12.21ns (82MHz)

Gaussian Random Number Generator (GRNG). In the GRNG implementation, $= 256$ is used. A single dual-port 512 36 bit block RAM is used to store both and arrays, each being 256 entries in size.

The GRNG produces 35-bit outputs and 32-bit values are maintained in the rectangular region datapath for all operations except the multiplier which takes 18 18 bit inputs and produces a 36-bit result. Similarly, the OU's datapath is 36-bit except for the inputs to the multiplier which are 18 bits. The data width and size of the major blocks used in the datapath of the GRNG are summarized in Fig. 1. Place and route results from the CAD tools, showing maximum clock frequency and resource utilisation are given in Table 2. The implementations do not include the optional on-chip test module, which would require around 12 more slices and 1 more block RAM.

The random point generated in the rejection method is accepted with probability of ${}_{accept}$ (as shown in (2)) which is 0 993 for $= 256$. In the case that the cycle results in the point being rejected, no output is produced. The throughput of the implementation, can thus be calculated by:

$$= {}_{CLK} \quad {}_{accept}$$

Our implementation shows 170MHz 0 993 which is 169 million samples / second. It is possible that the distribution of random numbers causes several wedge and tail requests to be queued in the OU, and the GRNG will stall waiting for an output of the OU. Our simulations show that this effect occurs infrequently enough that it does not significantly affect the throughput of the implementation since there are only 50 stalled cycles during 10^9 cycles of simulation. We envisage that most applications would not be able to process data at this rate and this will result in even fewer stalled cycles.

To make a fair comparison, the design is retargeted for a Virtex-II XC2V4000-6 device using the same 170MHz target frequency. Table 3 shows a comparison between the Ziggurat design, Wallace [7] and Box-Muller [2] designs for both hardware and software designs. It can be seen that the Ziggurat implementation uses less resources and is faster than the other two implementations. One drawback of the Wallace design is the use of previously generated random numbers, leading to poor distributions in those regions [14]. The Ziggurat method does not suffer from this effect.

Table 3. Comparisons of different Gaussian random number generators implemented on a XC2V4000-6 FPGA (one sample is generated in every clock cycle). The software implementations run on a P4-2GHz PC.

	Ziggurat	Wallace [7]	Box-Muller [2]	Ziggurat [14]	Wallace [14]	Box-Muller [14]
Clock	170MHz	155MHz	133MHz	2GHz	2GHz	2GHz
SLICEs	891	770	2514	-	-	-
Block RAMs	4	6	2	-	-	-
MULT18X18s	2	4	8	-	-	-
Throughput (M/sec)	169	155	133	16.1	52.6	3.6

In order to find out how small of a device can implement our design, an XC3S200, the second smallest device in Spartan-3 FPGA family, is fitted in. The implementation results are shown in Table 2. As the maximal frequency of implementation is 164MHz, the throughput can reach 163 million samples / second in this case.

Quality of GRNG's Random Numbers. In order to check the distribution of the random numbers generated by our design, we apply the chi-square (^{2}) goodness-of-fit test [15, 16]. The ^{2} test determines if the sample under analysis is drawn from a population that follows the specified Gaussian distribution. For the ^{2} test computation, the data are divided into bins and the test statistic is defined as: $^2 = \sum_{i=1}^{k} \frac{(O_i-E_i)^2}{E_i}$ where $_i$ is the observed frequency for bin and $_i$ is the expected frequency for bin . The expected frequency can be calculated by the probability $_i$ that each observation falls into the category and the total number of observations . So the expected frequency is $_i = \ _i$. We generate 10^9 normal random numbers with our hardware design and test them using the ^{2} test based on 512 bins spaced uniformly over [8 8]. The $^2_{511}$ statistic is 422.539. For a 95% level of confidence, the critical value is 565, showing that our hardware Ziggurat design produces good quality results. If $_1$ and $_2$ are normally distributed random numbers, $^{-(y_1^2+y_2^2)/2}$ should be uniformly distributed over [0,1]. This transformation is applied and the DIEHARD test runs. The resulting samples pass all the tests in the DIEHARD testsuite.

6. CONCLUSION

An architecture for a hardware Gaussian random number generator which can generate 169 million random numbers per second is described. The Ziggurat algorithm combines a high speed parallel datapath for the common rectangular region case and a sequential circuit for the infrequent wedge and tail regions. The resulting implementation is compact, fast and generates high quality Gaussian random numbers with correct distribution in the tails. Future work includes automating selection of the best method for elementary function evaluation, and retargeting the design for different FPGAs.

7. REFERENCES

[1] M. Gokhale et al., "Monte Carlo radiative heat transfer simulation," in *Proc. Field Programmable Logic and Applications*. LNCS 3203, Springer, 2004, pp. 95–104.

[2] D. Lee et al., "A Gaussian noise generator for hardware-based simulations," *IEEE Transactions on Computers*, vol. 53, no. 12, pp. 1523–1534, 2004.

[3] K. Chellapilla, "Combining mutation operators in evolutionary programming," *IEEE Transactions on Evolutionary Computation*, vol. 2, no. 3, pp. 91–127, 1998.

[4] J.M. McCollum et al., "Hardware acceleration of pseudo-random number generation for simulations applications," in *Proc. of Annual Southeastern Symposium on System Theory*, 2003, pp. 299–303.

[5] D. Knuth, *Seminumerical Algorithms*, ser. The Art of Computer Programming. Addison-Wesley, 1997, vol. 2.

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

[7] D. Lee et al., "A hardware Gaussian noise generator using the Wallace method," *IEEE Transactions on VLSI Systems*, 2005, accepted for publication.

[8] G. Marsaglia and W. W. Tsang, "The Ziggurat method for generating random variables," *Journal of Statistical Software*, vol. 5, no. 8, 2000.

[9] S. Moshier, *Methods and Programs for Mathematical Functions*. Halsted Press, 1989.

[10] P. L'Ecuyer, "Maximally equidistributed combined Tausworthe generators," *Mathematics of Computation*, vol. 65, no. 213, pp. 203–213, 1996.

[11] I. Munro, "Optimal algorithms for parallel polynomial evaluation," *Journal of Computer and System Sciences*, vol. 7, pp. 189–198, 1973.

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

[13] Intel Platform Security Division, "The Intel random number generator," *Intel Technical Brief*, 1999, ftp://download.intel.com/design/security/rng/techbrief.pdf.

[14] D. Lee et al., "Design parameter optimization for the Wallace Gaussian random number generator," *ACM Transactions on Mathematical Software*, 2005, submitted.

[15] R. D'Agostino and M. Stephens, *Goodness-of-fit Techniques*. Marcel Dekker Inc., 1986.

[16] G. W. Snedecor and W. G. Cochran, *Statistical Methods*. Iowa State University Press, 1989.