DESIGN OF HIGH SPEED AWGN COMMUNICATION CHANNEL EMULATOR

Emmanuel Boutillon[1], Jean-Luc Danger[2], Adel Ghazel[3]

[1]LESTER, University of Bretagne Sud, Centre de recherche - BP 92116, 56321 Lorient Cedex, France
[2] Ecole Nationale Supérieure des Télécommunications, ComElec, 46 rue Barrault, 75634 Paris Cedex 13, France
[3] UTIC - Ecole Supérieure des Communications, Rte de Raoued km 3.5 – 2083 El Ghazala –Tunisia
emmanuel.boutillon@univ-ubs.fr , danger@enst.fr, adel.ghazel@supcom.rnu.tn,

Abstract: This paper presents a method for designing a high accuracy white gaussian noise generator suitable for communication channel emulation. The proposed solution is based on the combined use of the Box-Muller method and the central limit theorem. The resulting architecture provides a high accuracy AWGN with a low complexity architecture for a digital implementation in FPGA. The performance is studied by means of MATLAB simulations and various complexity figures are given.

Keywords: AWGN, channel emulator, FPGA, central limit theorem, Box-Muller

1 Introduction

The design of a digital system for a communication application (error control coding, demodulation) is a very complex task requiring often trade-off between complexity and performances. In the ideal case, the formal expression of the Bit Error Rate (BER) can generally be expressed [1] and used to predict the performance of the system. But, in practice, the non-linearity of the system (fixed precision implementation) and/or the choice of a sub-optimal algorithm lead to a formal expression of the BER, which is too complex to derive. In that case, BER is evaluated using Monte-Carlo simulation. The real system is emulated with an exact software model of the transmission system (transmitter, channel and receiver) and its statistical behavior is estimated by software emulation of the transmission of thousand of bits. Monte-Carlo simulations are easy to set-up but they are time consuming. For example, 10^9 calculation iterations are needed to get an accurate (+-3.3%) estimation of a BER around 10^{-6}. Thus, the exploration of the solution space for obtaining a good trade-off performance/complexity is bounded by the simulation time needed to obtain reliable estimation of the BER.

To overcome this problem, some authors propose to speed up Monte-Carlo simulation using a cluster of computer working in parallel. In this method, each computer performs its own Monte-Carlo simulation of the system with a reduced number of iterations. Then, all the results generated by each computer are collected and summed to obtain a reliable estimation of the BER. For a

turbo-code application, effective data rate of 1.2 Mbit/s can be emulated using a cluster of 14 PCs for DVB-RCS turbo-decoder [2].

The complementary approach is to replace software emulation by hardware emulation (using FPGA circuit) in order to speed-up the simulation by a few orders of magnitude. Compared to a software compilation, this method is less flexible since each modification of the system requires the synthesis of the design from a Register Transfer Level (RTL) model and the place&route operations on the FPGA. But, once this is done, the simulation can run at a very high speed and precise BER evaluation can be obtained. Note that at the moment, the hardware emulation is not currently used, mainly because it requires both algorithm and hardware skills, but we believe that this type of method will be much more developed in the future. First, thanks to the progress of the CAD tools, configuration of FPGA becomes more and more easier for a non-specialist. Second, the increasing trade of Intellectual Properties (IP), also named Virtual Circuits (VC), generates the need for a client to evaluate and validate the IP. In that case, hardware emulation can be efficiently used.

The hardware emulation of a communication link contains at least three parts: the emitter, the channel and the receiver. In this paper, we are interested on the channel emulation and we focus specifically on the White Gaussian Noise Generator (WGNG). From this White Gaussian Noise Generator, the Additive White Gaussian Noise (AWGN) channel can be emulated and, with some additional computation, a large class of models of channel can also be derived from the WGNG (using ARMA filter for example) [3].

The main difficulty in emulating the WGNG is the faithful representation of the normal distribution $N(0,\sigma)$ that has a zero mean and a standard deviation of σ. The accuracy measurement of a random variable $X(x)$ is here indicated by the relative error $\xi_X(x)$ between the probability density function (p.d.f.) of X and the normal distribution $N(0,1)$.

$$\xi_X(x) = \frac{X(x) - N(0,1)(x)}{N(0,1)(x)} \quad (1)$$

The following parameters of the generated random variable *X* have been considered in the paper:

- $\xi_X(x) < 0.2\%$ for $|x| < 4\sigma$ (or a (0.2%, 4σ) accuracy);
- *b* bits (at least 6) of resolution after the decimal point;
- periodicity greater than 10^{18} samples (or 2^{60});
- flat spectrum;
- high sampling rate (> 10 MHz).

The rest of the paper is organized in five sections. Section 2 presents the current techniques to generate AWGN. Then section 3 proposes a new method based on the association of a quantized version of Box-Muller method and the use of the central limit theorem. Section 4 presents the architecture of the proposed method. Section 5 gives the design results in accuracy and complexity for different cases, results of a specific design are also given. Finally, conclusions are drawn in section 6.

2. State of the art

Different works has been done to generate AWGN. The real WGNG based on the thermal noise of a resistor is first presented. Then methods emulating the effect of the channel in a simple case are described. Finally, two methods allowing to generate a gaussian distribution, namely the method using the central limit theorem and the Box-Muller method are presented.

2.1 Method using thermal noise

The “natural” method to generate AWGN is the use of a true analog white Gaussian noise source associated to an Analog to Digital Converter (ADC). Generally, the noise source is obtained with

the amplification of the thermal noise of a resistor. This method is the only one that gives a true WGNG but first, its implementation is not really easy (need a costly high-speed high-quality ADC), second, it is impossible to generate twice the same sequence of data. Unfortunately, this last feature is particularly useful for the debug of the communication system: if the system has a transient wrong behavior on a sequence of data, it is important to be able to rerun the same simulation in order to detect the error. Thus, this solution is not taken into account in the rest of the paper.

2.2 Method using pre-computed probabilities

When the precision p of the ADC of the emulated system is below 4 bits, the number of possible received symbol $\{y_j\}_{0 \le j < 2^p}$ is low. In this case, instead of emulating the channel by the addition of a gaussian noise on the transmitted symbol, it is more efficient to emulate only the effect of the channel considering directly the received symbol. In fact, for a given Signal to Noise Ratio (SNR), it is possible to pre-compute the probability $p(y_j/x_i)$ of sending symbol x_i and receiving symbol y_j. Thus, the emulation of the WGNG, when symbol x_i is transmitted, is done by the generation of a random variable (r.v.) A_i with a probability law:

$$P(A_i = j) = p(y_j / x_i) \quad 0 \le j < 2^p \tag{2}$$

The r.v. A_i can also be characterized by its density function $F_{A_i}(j)$:

$$F_{A_i}(j) = P(A_i \le j) = \sum_{k=0}^{j} P(A_i = k) \tag{3}$$

An approximation of this law can be obtained when using q Linear Feedback Shift Registers (LFSR) [4] to obtain an uniformly distributed r.v. U^q over $\{0, 1, \ldots 2^q\text{-}1\}$. (see section 4.2 below). From a draw u of U^q, the index j of y_j is obtained simply as $j = F_{A_i}^{-1}(u/2^q)$. To perform this

operation directly into hardware, the value of u is compared to the 2^p-1 pre-computed values $2^q F_{A_i}(j)$, j=1..2^p-1. The value of j is easily deduced from the results of these comparisons [5].

Note that in the general case, symmetrical considerations on the constellation used for the transmission and the ADC allows reducing the number of r.v. A_i needed to emulate the effect of the AWGN.

In terms of hardware, q LFSRs are needed to generate U^q and 2^p-1 q-bit comparators are needed to compute the index j of the received symbol while the precision of this method depends directly on the parameter q. When p is important (at least p = 8 for our requirement) and high precision is required, the complexity of this method becomes very high.

Some authors proposed also the use of the r.v. U^q as an address in a 2^q word-RAM containing pre-defined real draws of the normal distribution. In [6], RAMs of 256 K words (q=18) are used for the hardward emulation of a Turbo-decoder. Once again, the complexity of this method becomes very high for a high precision WGNG.

2.3 Method using central limit theorem

The central limit theorem tells us that if X is a random real variable of mean m_x and standard deviation σ_x, the random variable X_N defined as:

$$X_N = \frac{1}{\sigma_x \sqrt{N}} \sum_{i=0}^{N-1} (x_i - m_x) \qquad (4)$$

where x_i, with i=0..N-1, are N independent determinations of the variable X, tends toward the normal distribution $N(0,1)$ of mean 0 and standard deviation σ=1, when N tends toward infinity.

Let us define the q bits r.v. U^q_N according to (4) with a number N of independent draws of the r.v. U^q. According to the central limit theorem, U^q_N tends toward $N(0,1)$ but this convergence is very

slow. For example, if q=4, N should be greater than 512 to fit the requirements, i.e. 2048 binary variables are needed to generate one sample of the WGNG. The hardware for the LFSR required to generate 2048 uniformly distributed random variables, is important.

2.4 Box-Muller method

The Box-Muller method is widely used in simulation software to generate a random variable (see [7] for a C program example). This method includes 3 steps: the first two ones generate independent values x_1 and x_2 of a random variable uniformly distributed over [0, 1]. In the third step, the functions, $f(x_1)$ and $g(x_2)$ are derived from x_1 and x_2 by:

$$f(x_1) = \sqrt{-\ln(x_1)} \tag{5}$$

$$g(x_2) = \sqrt{2}\cos(2\pi x_2) \tag{6}$$

Finally,

$$n = f(x_1)g(x_2) \tag{7}$$

gives a sample of the $N(0,1)$ distribution.

Using a 32-bit floating point CPU, equations (5), (6) and (7) are efficiently computed in a small number of clock cycles. Unfortunately, these operations (square root of a logarithm, cosine function, multiplication) require a lot of hardware.

In conclusion, none of those methods lead to a low complexity high quality WGNG. Other methods have to be explored.

3. Design of accurate AWGN reference model

In order to obtain a low complexity high precision WGNG, we propose to combine a quantized version of the Box-Muller method and the central limit theorem. The idea is to generate a

Gaussian noise sample in two steps: first, the quantized version of the Box-Muller method is performed to obtain a good approximation of the Gaussian distribution. Second, several samples thus obtained are accumulated by taking advantage of the central limit theorem to smooth the fluctuation of the distribution obtained with the quantized Box Muller method [8], [9].

3.1. Quantized Box-Muller method

To reduce the complexity, a quantized version of (5) and (6) using pre-computed values in Look-Up Table (LUT) is proposed by the authors. Let us first focus our attention on equation (5).

3.1.1 Non uniform quantization of function f(x_1*)*

The plot of the function $f(x_1)$ is shown in figure 1.

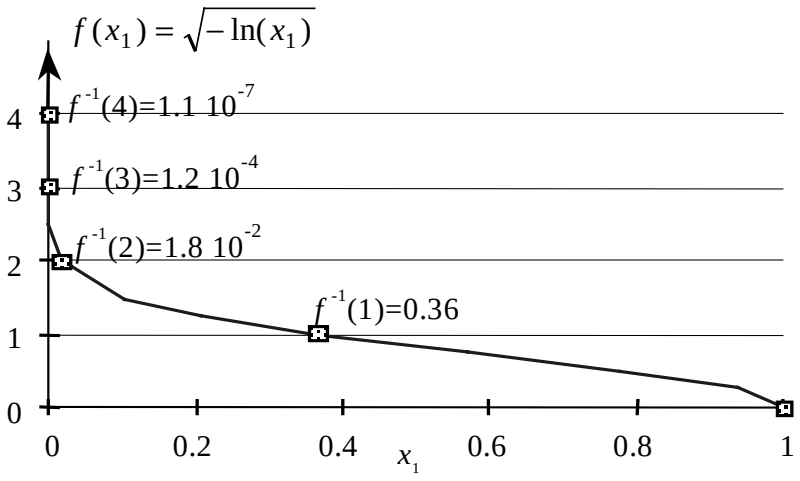

Figure 1: Plot of function $f(x_1)$

Since the average of $g(x)$ is close to 1 ($2\sqrt{2}/\pi$ exactly), a value of $f(x)$ greater than 4 should be generated in order to generate sample n greater than 4. To do so, the quantized step in segment [0,1] should be very small, i.e. of the order of $f^{-1}(4) < 10^{-7}$. Since only the vicinity of 0 should be accurately quantized, a recursive non-uniform quantization of segment [0,1] is proposed (see figure 2):

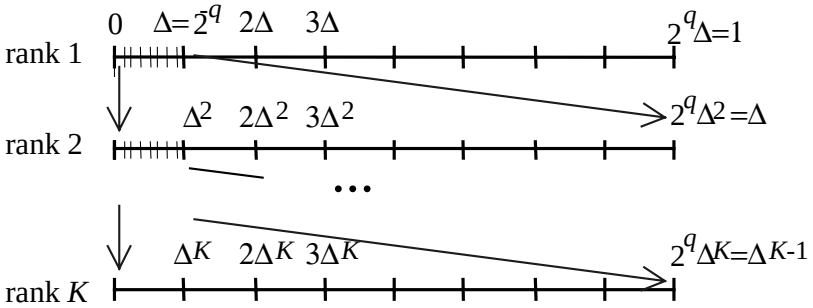

Figure 2: Non uniform quantization of segment [0,1]

where K is the number of recursions and q is the number of bits required to select one of the 2^q segments of length $\Delta^r = 2^{-rq}$ at the level r of the recursion. Thus, K q-bit random generators s_1, s_2... s_K are used to define the position of a sub segment of [0,1]. At the first level (r=1), the quantization step is $\Delta=2^{-q}$. The value s_1 defines the segment $[\Delta s_1, \Delta(s_1+1)[$, if s_1 is equal to 0, the sub-segment $[0, \Delta[$ is sub-divided in 2^q sub-segments of size Δ^2 indexed with s_2. Once again, if s_2 is equal to 0, the sub-segment $[0, \Delta^2[$ is sub-divided in 2^q sub-segments indexed by s_3. The process is repeated recursively K times. The probability to select a segment s of rank r is equal to Δ^r, i.e., the size of the segment. The process is thus uniformly distributed over [0,1]. The quantized value $f_r(s)$ associated to this segment is given by:

$$f_r(s) = \left\lfloor 2^m f((s+\delta)\Delta^r) \right\rfloor \quad (\times 2^{-m}) \tag{8}$$

Where m is the number of fractional bits used to represent $f_r(s)$ (i.e. $f_r(s)$ is coded on 3+m bits, 3 for the integer part, m for the fractional part) and $\lfloor x \rfloor$ denotes the largest integer lower than x and δ, a real number between 0 and 1, gives the relative position of the sample in the segment $[\Delta^r s, \Delta^r(s+1)[$.

*3.1.2 Quantization of function g(*x*)*

The problem is easier for the $g(x)$ function. Let us define s', a q' bit r.v. $U^{q'}$. $\Delta' = 2^{-q'}$ is the quantization step of segment [0,1/4] (the problem of sign is analyzed later), thus $g(x)$ is quantized as:

$$g(s') = \left\lfloor 2^{m'}\sqrt{2}\cos\left(\frac{\pi\Delta'(s'+\delta')}{2}\right)\right\rfloor \quad (\times 2^{m'}) \tag{9}$$

where δ' and m' have the same meanings as those of δ and m of equation (8). The function $g(s')$ is coded on $1+m'$ bits, 1 for the integer part, m' for the fractional part.

3.1.3 Final multiplication

From $f_r(s)$ and $g(s')$, the quantized Half Box-Muller random variable *HBM* with b bits after the dot is obtained using:

$$n^+ = \left\lfloor \frac{f_r(s)\times g(s')}{2^{m+m'-b}} \right\rfloor \quad (\times 2^b) \tag{10}$$

The probability P of obtaining a given couple $(f_r(s),g(s'))$ is:

$$P(f_r(s), g(s')) = 2^{-(rq+q')} \tag{11}$$

Let S_n be the subset of $\{0, ..., 2^q-1\}\times\{1, ..., K\}\times\{0,..., 2^{q'}-1\}$ of all triplets (s,r,s') giving n^+ using (10). The probability $P(HBM=n^+)$ is then given by:

$$P(HBM = n^+) \sum_{(s,r,s')\in S_n} P(f_r(s), g(s')) \tag{12}$$

Using (10), (11) and (12), the p.d.f. of *HBM* can easily be computed. Finally, the Box-Muller r.v. *BM* is obtained from the Half Box-Muller r.v. *HBM* using a binary r.v. *sign*:

$$n = (1-2\cdot sign)n^+ \tag{13}$$

Using (13), the p.d.f. of *BM* can also be computed.

3.2. Mixed method

The curve *a* of figure 3 illustrates the relative error $\xi_X(x)$ associated to the distribution BM_1 obtained with the Box Muller method and the parameters of Table 1.

Param.	*b*	*q*	*K*	*M*	δ	*q'*	*m'*	δ'
Matlab	b	q_f	K	m_f	d_f	q_g	m_g	d_g
BM_1	6	4	5	7	0.467	8	6	0.5

Table 1: Characteristics of the Box-Muller WGNG. The second line of the array gives the names of the variables used in the MATLAB program given in the appendix A.

With a reduced complexity (low value parameters) it is hardly possible with The Box-Muller to reach the initial constraint of (0.2%, 4 σ) accuracy.

To smooth the large variation of the distribution BM_1, a number *N* of independent r.v. BM_1 are accumulated (use of the central limit theorem) to generate a single sample. The resulting distributions BM_2 and BM_4. obtained for *N* = 2 and 4 are shown in figure 3.

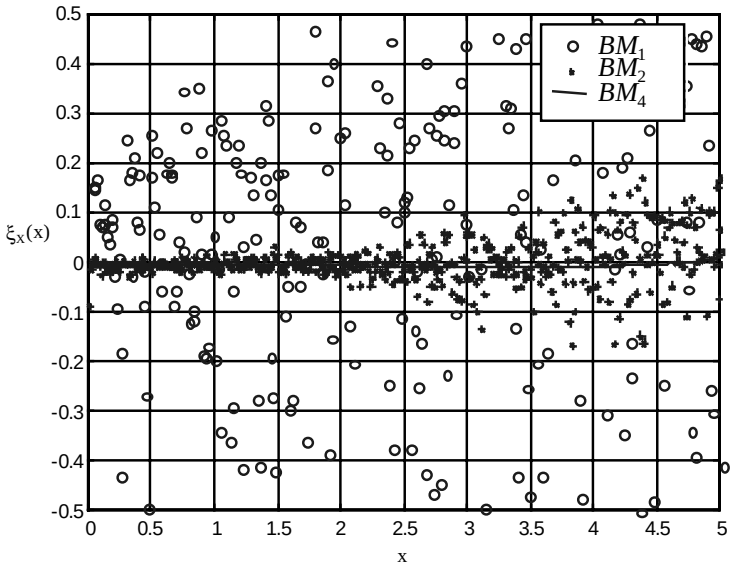

Figure 3: $\xi_X(x)$ for *X*= BM_1, BM_2 *and* BM_4

One can note that exact distribution of BM_2 can be computed as the auto-convolution of BM_1: $BM_2 = BM_1 \otimes BM_1$. The exact distribution of BM_4 can also be exactly computed ($BM_4 = BM_2 \otimes BM_2$). Result of figure 3 shows that BM_4 fulfills our initial requirement of a (0.2%, 4σ) accuracy. Appendix A gives the reference MATLAB program that compute the exact distribution according to the selected parameters.

4. Architecture

This chapter concerns the architectural aspect of the WGNG. It explains how to get an efficient and low complexity architecture with a FPGA device.

4.1. Overall architecture

Figure 4 shows the general architecture of the WGNG. The uniformly distributed random variables are generated by using several Linear Feedback Shift Registers (LFSR) (see chapter 4.2)

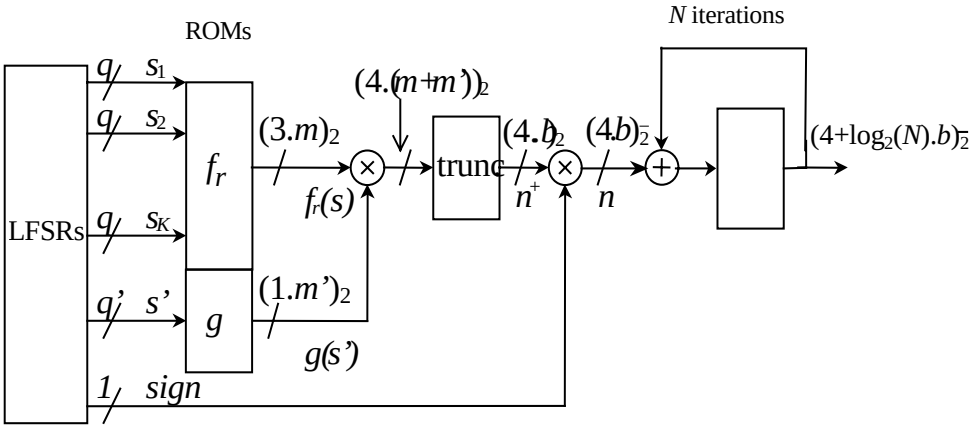

Figure 4: Overall architecture where $(a.b)_2$, respectively $(a.b)_{\bar{2}}$, indicates an unsigned (a two-complement) number with *a* bits before the dot and *b* bits after the dot.

The targeted FPGA structure is based on logic cells [10] allowing the generation of a logic function of 4 variables. Every logic cell output can be registered. Larger functions can be obtained

by combining logic cells. In Table 1 q=4 allows the designer to perform a f_r ROM of 3+m bits with only 3+m logic cells. The logic cells input being 4 uniformly distributed address bits.

As shown in Figure 3 (BM_4) and Table 1, good results are obtained with K=5 and m=7, which means that (3+m)K=50 logic cells are needed for the f_r ROMs. A 256-byte FPGA on-chip RAM can be used for the g ROM (m'=8-1=7). The parameters of Table 1 offer a good trade-off between performance and FPGA complexity.

Once the Box-Muller variable is generated a truncation is done to keep only b bits after the decimal point. In our example we truncated to get 6 bits after the decimal point. The multiplication by the sign permits to get a p.d.f. centered on 0 by using a 2's complement generator when the sign is 1. In this case the zero value of the p.d.f. is twice as big than the normal distribution (because 2's complement of 0 is 0). If the 1's complement is used instead of the 2's complement, the mean value becomes -2^{-b-1} instead of 0 before accumulation. After accumulation the mean and standard deviation are given by:

$$\text{mean}(BM_{N,b}) = -N \cdot 2^{-b-1} \tag{14}$$

$$\text{standard deviation}(BM_{N,b}) = \sqrt{N} \tag{15}$$

To be as close as possible to $N(0,1)$ when the 1's complement is used, a compensation has to be done at the back end stage. The back end stage consists in multiplying the noise according to the needed signal power and in adding the result to the signal. For instance if N=4, a mere left shift of the decimal point is enough to compensate σ (i.e. a division by $\sqrt{4}$ =2) and the addition with -2^{-b+1} compensates the mean.

4.2. Uniformly distributed variables generation with LFSRs.

A total of $K.q + q' + 1$ uniformly distributed variables are needed:

- $K.q$ to address the K f_r ROMs,

- q' to address the g ROM

- 1 to generate the *sign*.

It is known that a Linear Feedback Shift Register LFSR associated with its characteristic polynomial G[x] of order n can generate a very good random like binary variable of periodicity 2^n-1 [4]. Associating q independent LFSRs generate a q bit variable U^q uniformly distributed over $\{0, 1, 2, ..., 2^{q-1}\}$. The LFSR design in FPGA need only n logic cells, each of them with its own register. Figure 5 illustrates the LFSR structure called "one to many" with the polynomial $x^5 + x^2 +1$:

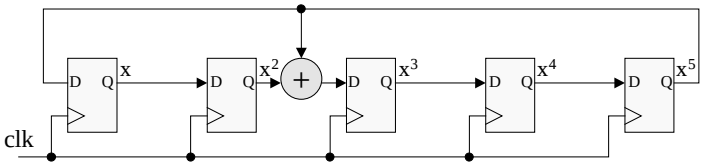

Figure 5: LFSR for $x^5 + x^2 +1$

Considering the parameters of Table 1, 29 uniformly distributed variables have to be generated. the number of LFSRs can be reduced if the address bits are grouped by packet of 4, necessiting only 7 LFSRs, 2 for the g ROM, one for each f_r ROM and one for the *sign*. At every clock cycle, 4 bits are used as outputs and "shifted". For instance for the LFSR of figure 5, t being the clock period, the register x^5 can be expressed as $x^5(t)= x^4(t\text{-}1) = x^2(t\text{-}3)+x^5(t\text{-}3) = x(t\text{-}4)+x^4(t\text{-}4)$. By considering operations every 4t, 4 virtual shift operations are done in one clock cycle.

This technique can be easily coded in VHDL and generates almost no extra FPGA logic cells. The code of the LFSR function generator is given in the Annex B for any number of outputs (parameter *nb_iter* in the code). Figure 6 illustrates the LFSR structure with polynomial $x^5 + x^2 +1$ and 4 ouputs.

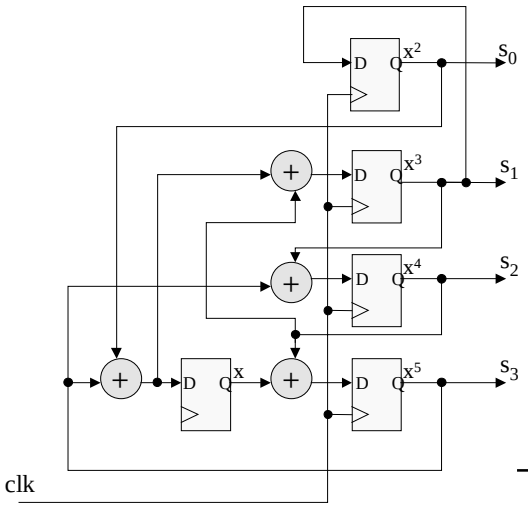

Figure 6: LFSR for $x^5 + x^2 + 1$ and 4 outputs

	LFSR 1 output					LFSR 4 outputs				
t	x	x^2	x^3	x^4	x^5	x	x^2	x^3	x^4	x^5
0	1	0	0	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0	0	0	1
2	0	0	1	0	0	1	0	1	1	0
3	0	0	0	1	0	0	1	1	1	0
4	0	0	0	0	1	1	1	0	1	1
5	1	0	1	0	0	0	0	1	1	0
6	0	1	0	1	0	0	1	1	1	1
7	0	0	1	0	1	0	1	1	0	1
8	1	0	1	1	0	0	1	0	0	0
9	0	1	0	1	1	1	0	1	0	0
10	1	0	0	0	1	0	1	0	1	1
11	1	1	1	0	0	0	0	1	1	1
12	0	1	1	1	0	1	1	0	0	1

Table 2: LFSR sequences

The sequence of the first 12 combinations is indicated in Table 2. The initial value is "00001". This table shows combinations of the 4-outputs LFSR corresponding to each fourth -combination of the 1-output LFSR.

In order to meet the periodicity constraint (i.e. a periodicity greater than 2^{60} cycles), at least a total of 60 registers of LFSRs are needed. In order to keep the highest period, the LFSRs need to have periods prime between them. To meet this condition, we propose to select the LFRS's length from the "Mersenne" numbers (number *d* so that 2^d–1 is a prime number).

5. Results

5.1 Accuracy

By considering the parameters of Table 1 and x between 0 and 4σ, the maximum relative error $\xi_X(x)$ between the ideal gaussian distribution $N[0,1]$ and the synthesized one is calculated by using a MATLAB model. The accuracy is given for different values of the number b of bits after the decimal point, and the number of accumulations N. Table 3 represents the maximum relative error expressed in 10^{-3} for different values of N and b. For every value of b the optimal value of δ is indicated.

Max $\xi_X(x) * 10^{-3}$ Between 0 and 4σ		N=2	N=3	N=4	N=5
b	1 δ=0.44	0.65	0.08	0.15	0.29
	2 δ=0.453	11.5	1.96	0.93	0.43
	3 δ=0.445	20.2	2.12	0.56	0.34
	4 δ=0.467	64.6	5.4	0.71	0.31
	5 δ=0.467	57.3	5.4	1.12	0.69
	6 δ=0.467	71.9	5.8	1.38	0.93
	7 δ=0.467	237	8.4	0.68	0.28
	8 δ=0.467	503	26.5	1.76	0.26

Table 3: maximum relative error for different N and b

5.2 Synthesis

Table 4 gives the results obtained with the parameters of Table 1, N=4, b=6 and LFSRs of length 22,21,20,17,13,7,5,15 registers for respectively g, f_r and *sign:*

FPGA device	Cells	mem block	clock rate	Output rate
10K100ARC240-1	434	1	74MHz	18.5MHz
10K100EQC240-1	437	0.5	98MHz	24.5MHz

Table 4: synthesis results

The synthesis has been done using FPGA Express™ and MAX+PLUSII™. The number of cells of the LFSR part is 149. Note that a sample can be obtained at a rate of 98MHz using 4 parallel Box-Muller generators and one 4-word adder. Figure 7 illustrates the relative error obtained with 10^9 samples.

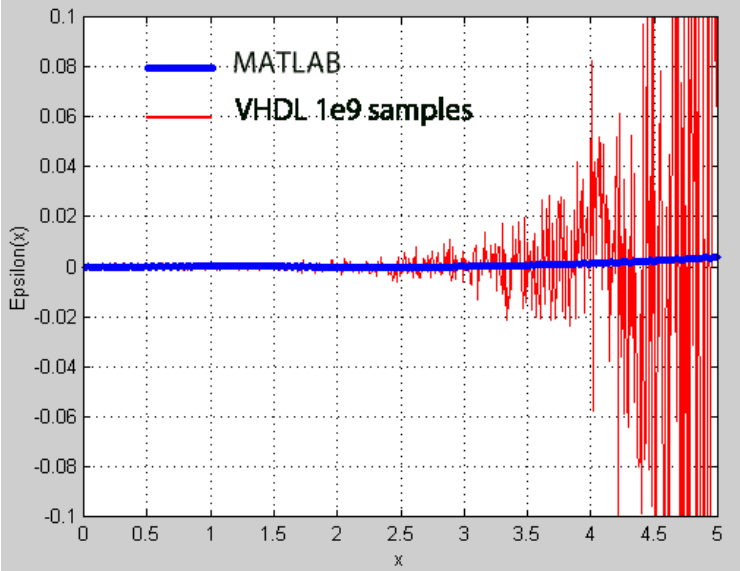

Figure 7: relative error with 10^9 samples

The difference between the theoretical distribution and the one obtained is due to the low number of samples obtained for high value of σ. When the number of samples increases, the distribution converges towards the result of the MATLAB simulation

6. Conclusion

In this work, we use both the Central limit theorem and the Box-Muller methods to obtain a probability density function with a (0.2%, 4σ) accuracy. Moreover, a MATLAB reference program is elaborated which takes the hardware architecture characteristics (memory size, data format,…) into account. For easy evaluation, a linear representation of the distribution, together with its deviation from the theoretical gaussian distribution, is defined. A generic MATLAB

model has been written to allow the designer to adapt the quality of the gaussian distribution to his needs and consequently to adjust the parameters of the VHDL model.

One can note that using filtering and appropriate mathematical functions, the WGNG can also be used to emulate the Rayleigh, the Ricean channel or even more complex channels. Studies are pursued in this direction.

Reference

[1] J.G. Proakis, "*Digital communications"*, Mc GRAW-HILL International Editions, Electrical Engineering Series, 1998.

[2] M. Jézéquel, ENST Bretagne, personal communication, june 2001.

[3] J.R. Ball, "A real time fading simulator for mobile radio", *The radio and Electronic Engineer, Vol 52*, N°10,October 1982.

[4] E.R. Berlekamp, "Algebric Coding Theory", McGraw-Hill, 1968

[5] J. Touch, Virtual Turbo, personal communication, june 2001.

[6] M. Jézéquel, C. Berrou, J. R. Inisan and Y. Sichez, "Test of a Turbo-Encoder/Decoder", TURBO CODING Seminar, Lund, Sweden, pp. 35-41, 28-29 August 1996.

[7] Donald E. Knuth, "*The Art of computer programming*", ADDISON-WESLEY, 1998

[8] J.L. Danger, A. Ghazel, E. Boutillon, H. Laamari, "Efficient FPGA Implementation of Gaussian Noise Generator for Communication Channel emulation", IEEE ICECS conference, Laslik, Lebanon, Dec 2000.

[9] A. Ghazel, E. Boutillon, J.L. Danger, G. Gulak, "Design and performance analysis of a high speed AWGN communication channel emulator", IEEE PACRIM conference, Victoria, B.C., Aug. 2001.

[10] ALTERA Data Book 1998

Appendix A: Reference MATLAB Program

(See table 1 for the name and the value of the parameters)

```
for r=1:K  % ROM fr(s)
  for s=1:pow2(q_f)-1
    rom_f(s,r)=floor(sqrt(log((s+d_f)*
pow2(r*q_f)))*pow2(m_f));
  end;
 rom_f(pow2(q_f), r) = 0;
end;

for u=1:pow2(q_g)  %ROM g(x)
  rom_g(u)=floor(sqrt(2)*cos(pi*((u-1+d_g)*pow2(-q_g-1)))*pow2(m_g));
end;

% Initialisation of the distribution HBM
scaling = pow2(m_f + m_g - b);
max = floor(1 + rom_f(1,K)*rom_g(1)/scaling)
HBM = zeros(1,max+1);

for s=1:pow2(q_f)-1 %Construction of HBM
  for r=1:K
    for u=1:pow2(q_g)
   n=floor((rom_f(s,r)*rom_g(u)/scaling));
   HBM(n+1) = HBM(n+1) + pow2(-(r*q_f + q_g));
   end;
 end; end;

for i = 1 : max   %Construction of BM
   BM(i) = 0.5*HBM(max+2-i);
   BM(2*max+2-i) = BM(i);
end;
BM(max+1)=HBM(1);

BMi = BM; % Construction of BM_N
for i = 1 : N-1
     BMi = conv(BMi , BM);
end;
```

Appendix B: LFSR VHDL code

```
FUNCTION gen_lfsr(
  pol     : std_logic_vector;
  en      : std_logic;
  nb_iter : natural
RETURN std_logic_vector IS

VARIABLE pol_int : std_logic_vector(pol'length-1 DOWNTO 0);
VARIABLE pol_gen : std_logic_vector(pol'length-1 DOWNTO 0);

BEGIN

CASE pol'length is
    when 22 => pol_gen := "0000000000000000000011";
    when 21 => pol_gen := "0000000000000000000101";
    when 20 => pol_gen := "0000000000000000001001";
    when 17 => pol_gen := "00000000000001001";
    when 13 => pol_gen := "0000000011011";
    when  7 => pol_gen := "0000011";
    when  5 => pol_gen := "00101";      -- x^5 + x^2 + 1
    when  4 => pol_gen := "0011";       -- x^4 + x + 1
    when  3 => pol_gen := "011";        -- x^3 + x + 1
    when  others => pol_gen := "11";    -- x^2 + x + 1
END CASE;

pol_int := pol;

iteration : FOR i in 1 to nb_iter LOOP
  IF en = '1' THEN
    IF pol_int(pol'length-1)='1' THEN
       pol_int := (pol_int(pol'length-2 DOWNTO 0)&'0') xor pol_gen;
    ELSE
       pol_int := (pol_int(pol'length-2 DOWNTO 0)&'0');
    END IF;
  ELSE   pol_int := pol_int;
  END IF;
END LOOP;

RETURN (pol_int);
END gen_lfsr;
```