POZNAN UNIVERSITY OF TECHNOLOGY ACADEMIC JOURNALS
No 80 Electrical Engineering 2014

Piotr KOZIERSKI*
Marcin LIS*
Andrzej KRÓLIKOWSKI*

IMPLEMENTATION OF FAST UNIFORM RANDOM NUMBER GENERATOR ON FPGA

The article presents approach to implementation of random number generator on FPGA unit. The objective was to select a generator with good properties (correlation values and matching of probability density function were taken into account). Design focused on logical elements so that the pseudo-random number generation time depend only on the electrical properties of the system. The results are positive, because the longest time determining the pseudorandom number was 16.7ns for the "slow model" of the FPGA and 7.3ns for "fast model", while one clock cycle lasts 20ns.

KEYWORDS: random number generator, uniform noise, FPGA unit, logic functions

1. INTRODUCTION

The FPGA unit is primarily intended for parallel computations. Its use can reduce calculation time even by several orders of magnitude [6]. However, the disadvantage of the system is the lack of many functions, which are basic in other languages. One of those functions, on which article is focused, is the calculation of pseudo-random number with uniform distribution. It is also an element required for other noise generators, as for example Ziggurat Method [5], Alias Method [1] or Ratio Method [4]. However, there are also methods that do not use uniform noise, such as Wallace Method [3]. The approach proposed in this article assumes implementation of a standard algorithm to generate random numbers. The difference is that the whole algorithm should be made based only on logical gates, so that it will have a very high speed, and the subsequent generation of the random number will be able to take place in each clock cycle (every 20 ns).

The second section describes the type of the random number generator, which has been selected for implementation. In the third chapter, one can read about parameter selection of pseudo-random number generator. The method for module implementing on FPGA is presented in chapter four, while in the fifth chapter results of the time simulation were discussed. Chapter six concludes the article.

* Poznan University of Technology.

2. RANDOM NUMBER GENERATOR

The algorithm can be represented by short formula

$$X_{n+1} = (a \cdot X_n + c) \bmod m \tag{1}$$

where X is random number. Parameters a, c and m are chosen by the programmer. This algorithm has been proposed already in the 50s of the twentieth century [2], but it is still often used in less sophisticated random number generators. This type of generator was chosen for implementation on FPGA unit.

All generator parameters were selected in such a way to reduce the number of performed calculations, as much as possible. Therefore, the value m is equal to 2^{32} (assumed that the number has to be 32-bit), to save time on calculating the modulo. Sometimes one can come across with a proposal to establish the parameter m larger than is needed, to increase the period after which sequentially generated numbers will be repeated.

In the particular case it can be assumed that parameter c is equal to 0, however, in this case, all generated numbers would be even (or odd). One can check that in order to generate numbers of both even and odd, parameters a and c must be odd.

Indication has been made that the logical elements must be used and in FPGA all numbers greater than 1 are represented by the bit vector. Thus, in order to reduce the number of operations, chosen parameters should have the minimum number of non-zero bits (especially parameter a).

Below is shown how big is the difference in multiplying the 8-bit number by 179 (10110011_2) and by 193 (11000001_2). One can see, that each additional non-zero bit in the parameter a increases the number of logic elements required to implement the algorithm.

$$
\begin{array}{cccccccccccccccc}
 & & & & & & & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 \\
 & & & & & & & \bullet & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
\hline
 & & & & & & & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 \\
 & & & & & & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & \\
 & & & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & & & & \\
 & & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & & & & & \\
+ & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & & & & & & & \\
\hline
Y_{15} & Y_{14} & Y_{13} & Y_{12} & Y_{11} & Y_{10} & Y_9 & Y_8 & Y_7 & Y_6 & Y_5 & Y_4 & Y_3 & Y_2 & Y_1 & Y_0
\end{array}
$$

$$\begin{array}{cccccccccccccccc}
 & & & & & & & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 \\
 & & & & & & & \bullet & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
 & & & & & & & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 \\
 & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & & & & & & \\
+ & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & & & & & & & \\
\hline
Z_{15} & Z_{14} & Z_{13} & Z_{12} & Z_{11} & Z_{10} & Z_9 & Z_8 & Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0
\end{array}$$

Therefore were selected only 86 primes, in which the number of non-zero bits is 2 or 3, and among them a satisfactory value for the generator were sought.

3. CHOICE OF GENERATOR PARAMETERS

Certain parameters were chosen based on the calculated properties of pseudo-random numbers sequence – autocorrelation and histogram.

One of the good generator features should be low autocorrelation value [7]

$$\hat{R}_{xx}(i) = \frac{1}{N} \sum_{n=1}^{N-1-|i|} x(n) \cdot x^*(n-i) \tag{2}$$

for lags $i \neq 0$, where N is length of pseudo-random number sequence and $x(n)$ is n-th number of this sequence. For the calculation of the autocorrelation, function in Matlab was used, which calculates the value without scaling (default setting), so

$$\hat{R}(i) = \sum_{n=1}^{N-1-|i|} x(n) \cdot x^*(n-i) \tag{3}$$

Parameter, based on the autocorrelation values, has been proposed

$$c_{rl} = \sum_{i=1}^{100} \left(\hat{R}(i)\right)^2 \tag{4}$$

which value was directly compared between different pairs of generator parameters (a,c).

The second property, which has been studied, is the histogram. The sum of square errors between true value of probability density function (PDF) and the histogram value has been calculated. This can be represented by the formula

$$h^2 = \sum_{j=1}^{M} \left(\frac{O_j - E_j}{E_j} \right)^2 \tag{5}$$

where M is the number of intervals of probability density function, O_j means the number of randomly selected values from the j-th interval and E_j means theoretical number of values in j-th interval.

3.1. Simulations for different a and c parameters

Based on values of c_{rl} and h^2 the best pair of generator parameters (a,c) was searched. The length of generated sequence was $N = 10^5$ samples. The simulation was repeated 100 times for each pair of parameters, with different initial values. Table 1 shows some typical results obtained in the simulations.

Table 1. Results of h^2 and c_{rl} in few simulations for different generator parameters

a, c	h^2	$\sigma(h^2)$	c_{rl}	$\sigma(c_{rl})$
$a = 2^{11} + 2^5 + 2^0$ $c = 2^3 + 2^2 + 2^0$	0.0994	0.0156	0.000989	0.000133
$a = 2^{19} + 2^9 + 2^0$ $c = 2^{18} + 2^1 + 2^0$	0.0994	0.0141	0.001343	0.000217
$a = 2^{18} + 2^9 + 2^0$ $c = 2^{19} + 2^6 + 2^0$	0.0992	0.0140	0.000963	0.000153
$a = 2^{30} + 2^{13} + 2^0$ $c = 2^{12} + 2^1 + 2^0$	0.1278	0.0191	0.001877	0.000754
$a = 2^{21} + 2^{12} + 2^0$ $c = 2^{19} + 2^6 + 2^0$	0.0945	0.0130	0.000795	0.000110
$a = 2^{29} + 2^{15} + 2^0$ $c = 2^{30} + 2^{13} + 2^0$	0.0815	0.0090	0.000738	0.000214
$a = 2^{30} + 2^{13} + 2^0$ $c = 2^{20} + 2^{17} + 2^0$	0.0900	0.0132	0.007080	0.003048
$a = 2^{30} + 2^{19} + 2^0$ $c = 2^{20} + 2^{17} + 2^0$	0.0017	0.0002	0.001390	0.000316
$a = 2^{27} + 2^{21} + 2^0$ $c = 2^{14} + 2^{11} + 2^0$	0.0488	0.0020	0.000563	0.000089

Among the performed simulations the best was the last example in Table 1 (for $a = 2^{27} + 2^{21} + 2^0 = 136314881$ and $c = 2^{14} + 2^{11} + 2^0 = 18433$). Although one can notice better results for h^2 (second last example in Table 1), however the parameter based on correlation was finally considered as the most important, so pair of parameters obtained the best sequence in terms of c_{rl} was chosen.

4. MODULE CONSTRUCTION FOR FPGA

To better illustrate the operation and construction of the module, it will be shown on the 8-bit example, for generator parameters $(a = 41, c = 5)$. Below shows the multiplication:

$$
\begin{array}{cccccccccccccc}
 & & & & & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 \\
 & & & & & \bullet & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
\hline
 & & & & & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 \\
 & & & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & & & \\
+ & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & & & & & \\
\hline
 & & & & & & Y_7 & Y_6 & Y_5 & Y_4 & Y_3 & Y_2 & Y_1 & Y_0
\end{array}
$$

High-order bits are not shown, because the result of the module should be 8-bit number, so only $Y_{7:0}$ bits are visible. To the result of the multiplication should be added the value of $c = 5$, and therefore it can be presented as a sum:

$$
\begin{array}{ccccccccc}
 & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 \\
 & X_4 & X_3 & X_2 & X_1 & X_0 & & & \\
 & X_2 & X_1 & X_0 & & & & & \\
+ & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline
 & Y_7 & Y_6 & Y_5 & Y_4 & Y_3 & Y_2 & Y_1 & Y_0
\end{array}
$$

To calculate the Y_0 value, it must be perform the XOR operation on the values X_0 and 1. Y_1 value depends not only on X_1, but also on the previous sum – if there is a carry (P_w) or not. Additionally, in the case where four or more bits are summed, should be taken into account also carry affecting on the next bit (R_w). Therefore, the final form of the sum will be as follows:

$$
\begin{array}{ccccccccc}
 & R_5 & & & & & & & \\
 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 & \\
 & X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 \\
 & X_4 & X_3 & X_2 & X_1 & X_0 & & & \\
 & X_2 & X_1 & X_0 & & & & & \\
+ & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline
 & Y_7 & Y_6 & Y_5 & Y_4 & Y_3 & Y_2 & Y_1 & Y_0
\end{array}
$$

The smaller modules were created first, summing from 2 to 6 bits, and then were combined together. For example, 4 bits summing module has 4 inputs (number of bits) and 3 outputs (Y_w, P_w and R_w), where 1 output is the part of the result (Y_w) and 2 others (P_w and R_w) are inputs in next modules.

In a similar way 32-bit number generator was created, for parameters $a = 2^{27} + 2^{21} + 2^{0} = 136314881$ and $c = 2^{14} + 2^{11} + 2^{0} = 18433$.

4.1. Logical functions in modules

Functions are different depending on the number of inputs. Marks $\&$ and $|$ means respectively logical operations AND and OR, $\wedge$ means XOR, whereas $\sim$ means NOT. Logical functions describing the module outputs are presented below:

– for 2-bit summing module (inputs A and B)

$$Y_w = A \wedge B \tag{6}$$

$$P_w = A \,\&\, B \tag{7}$$

– for 3-bit summing module (inputs A, B and C)

$$Y_w = A \wedge B \wedge C \tag{8}$$

$$P_w = (A \,\&\, B) \,|\, (C \,\&\, (A \,|\, B)) \tag{9}$$

– for 4-bit summing module (inputs A, B, C and D)

$$Y_w = A \wedge B \wedge C \wedge D \tag{10}$$

$$P_w = (\sim R_w) \,\&\, (((A \,|\, B) \,\&\, (C \,|\, D)) \,|\, ((A \,|\, C) \,\&\, (B \,|\, D))) \tag{11}$$

$$R_w = A \,\&\, B \,\&\, C \,\&\, D \tag{12}$$

– for 5-bit summing module (inputs A, B, C, D and E)

$$Y_w = A \wedge B \wedge C \wedge D \wedge E \tag{13}$$

$$P_w = (\sim R_w) \,\&\, (((A \,|\, B \,|\, C) \,\&\, (D \,|\, E)) \,|\, ((A \,|\, B \,|\, D) \,\&\, (C \,|\, E)) \,|\, (A \,\&\, B)) \tag{14}$$

$$R_w = (A \,\&\, B \,\&\, C \,\&\, (D \,|\, E)) \,|\, ((A \,|\, B) \,\&\, C \,\&\, D \,\&\, E) \,|\, (A \,\&\, B \,\&\, D \,\&\, E) \tag{15}$$

– for 6-bit summing module (inputs A, B, C, D, E and F)

$$Y_w = A \wedge B \wedge C \wedge D \wedge E \wedge F \tag{16}$$

$$\begin{aligned} P_w = {} & ((\sim R_w) \,|\, (A \,\&\, B \,\&\, C \,\&\, D \,\&\, E \,\&\, F)) \,\& \\ & \&(((A \,|\, B \,|\, C) \,\&\, (D \,|\, E \,|\, F)) \,|\, ((A \,|\, D \,|\, F) \,\&\, (B \,|\, C \,|\, E)) \,|\, ((B \,|\, D) \,\&\, (C \,|\, F))) \end{aligned} \tag{17}$$

$$\begin{aligned} R_w = {} & ((A \,|\, B) \,\&\, (C \,|\, D) \,\&\, E \,\&\, F) \,|\, ((A \,|\, B) \,\&\, C \,\&\, D \,\&\, (E \,|\, F)) \,| \\ & |\, (A \,\&\, B \,\&\, (C \,|\, D) \,\&\, (E \,|\, F)) \,|\, (C \,\&\, D \,\&\, E \,\&\, F) \,| \\ & |\, (A \,\&\, B \,\&\, E \,\&\, F) \,|\, (A \,\&\, B \,\&\, C \,\&\, D) \end{aligned} \tag{18}$$

5. TIME SIMULATION RESULTS

The sequence of generated numbers obtained during simulation of created module was correct, which confirm the correctness of implementation.

Time after which module output was steady also has been taken into account. After generating 1000 consecutive numbers, the longest time period obtained for the "slow model" was 16.725 ns and for "fast model" – 7.338 ns. One can assumed that the maximum time generation of pseudo-random numbers on real FPGA unit will be between the values obtained from simulations.

All time simulations were made using ModelSim® Altera® 6c and Quartus® II 10.1 Web Edition programs.

6. SUMMATION

The article proposed the method of generating pseudo-random numbers by an appropriate choice of generator parameters – thus obtained numerical sequence had to have the best properties. Simultaneously take into account that the selected parameters should provide high-speed operation of the module (1 clock cycle on the test FPGA lasts 20 ns). Based on the simulation one can conclude that the module has been built properly.

Further research will aim to verify the operation of the generator on a real system and the implementation of pseudo-random number generator with a Gaussian distribution.

REFERENCES

[1] Ahrens J. H., Dieter U., An Alias Method for Sampling from the Normal Distribution, Computing, Vol. 42, No. 2-3, 1989, pp. 159-170.

[2] Knuth D. E., The Art of Computer Programming, Addison-Wesley Publishing Co., Vol. 2 Seminumerical Algorithms, 1981, pp. 1-37.

[3] Lee D. U., Luk W., Villasenor J. D., Zhang, G., Leong P. H. W., A Hardware Gaussian Noise Generator Using the Wallace Method. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, Vol. 13, No. 8, 2005, pp. 911-920.

[4] Leva J. L., A Fast Normal Random Number Generator, ACM Transactions on Mathematical Software, Vol. 18, No. 4, December 1992, pp. 449-453.

[5] Marsaglia G., Tsang W. W., The ziggurat method for generating random variables, Journal of Statistical Software, Vol. 5, No. 8, 2000, pp. 1-7.

[6] Mountney J., Obeid I., Silage D., Modular Particle Filtering FPGA Hardware Architecture for Brain Machine Interfaces, Conf Proc IEEE Eng Med Biol Soc. 2011, pp. 4617-4620.

[7] Zieliński T., Cyfrowe przetwarzanie sygnałów: Od teorii do zastosowań, Wydawnictwa Komunikacji i Łączności, Warszawa 2007, pp. 1-38.