Computer Applications in Electrical Engineering
Vol. 12 *2014*

Parallel uniform random number generator in FPGA

Piotr Kozierski, Marcin Lis, Andrzej Królikowski
Poznań University of Technology
60-965 Poznań, ul. Piotrowo 3a, e-mail: piotr.kozierski@gmail.com

The article presents approach to implementation of random number generator in FPGA unit. The objective was to select a generator with good properties (correlation values and fidelity of probability density function were taken into account). During the 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. Additionally the parallel random number generator has been proposed, composed of 10 simple generator modules. After modules connecting, maximum time for generation of 10 random numbers was equal 41.0ns for the "slow model" and 16.6ns for the "fast model".

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

1. Introduction

FPGA unit is primarily intended for parallel computations. Its use can reduce the time of calculations even by several orders of magnitude [7]. 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 [6], Alias Method [1] or Ratio Method [5]. However, there are also methods that do not use uniform noise, such as Wallace Method [4].

The approach proposed in this article assumes implementation of the standard algorithm for generation of the 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 20ns).

Then is has been assumed that in every clock cycle may be needed more than one random number. Therefore 10 modules have been connected in such a way that output of first module is input of second module, output of second module is input of third module, and so on. For such a connection the computation time was longer than 20ns, but definitely less than 10 clock cycles.

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.

This paper basis on the article [3] prepared on the ZKwE 2014 conference.

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, and parameters a, c and m are chosen by the programmer. This algorithm was 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 has been chosen to implementation in 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} (the assumption has been made that the numbers are 32-bit), to save time needed to calculating the modulo. Sometimes one can find the proposal to establish the parameter m larger than it is needed, to increase the period after which sequentially generated numbers will be repeated.

In the particular case it can be assumed that parameter $c = 0$, 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.

Indicated already that the logical elements must be used to implementation 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 it has been shown how big is the difference in multiplying the 8-bit number by 193 (11000001_2) and by 179 (10110011_2). One can see, that each additional non-zero bit in the parameter a increases the number of logical elements required in algorithm implementation.

								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
								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							
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

								X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0
							•	1	0	1	1	0	0	1	1
								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							
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

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

3. Choice of generator parameters

Certain parameters have been 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 [8]

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

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)

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

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 is the number of randomly selected values from the j-th interval and E_j is 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) has been chosen. The length of generated sequence of random numbers was equal to $N = 10^5$ samples. The simulation has been repeated 100 times for each pair of parameters, with different initial value. Some typical results, obtained in the simulations, have been shown in Table 1.

Table 1. Results of h^2 and c_{rl} obtained 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 the two parameters with the lowest value of c_{rl} have been chosen.

4. Construction of module in FPGA

To better illustrate the operation and construction of the module, it has been shown on the 8-bit example, for some generator parameters $(a = 41, c = 5)$. The multiplication has been shown below.

						X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0
					•	0	0	1	0	1	0	0	1
						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					
						Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0

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 the new random number can be represented by the sum:

	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
	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0

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, the carry affecting on the next bit (R_w) should be taken into account. Therefore, the final form of the sum should be as follows:

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

Therefore, the smallest modules which have been created are 2-6 bits summators, and then these modules have been combined together. For example, 4 bits summator module has 4 inputs (number of bits) and 3 outputs (Y_w , P_w and R_w), where one output is the part of the result (Y_w) and two other outputs (P_w and R_w) have been connected with inputs in next modules.

In a such way 32-bit number generator has been 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, ^ means XOR, whereas ~ means NOT. Logical functions describing the module outputs have been presented below:

- for 2-bit summator module (inputs A and B)

$$Y_w = A \,\hat{}\, B \tag{6}$$

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

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

$$Y_w = A \,\hat{}\, B \,\hat{}\, C \tag{8}$$

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

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

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

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

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

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

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

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

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

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

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

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

$$R_w = ((A \mid B) \,\&\, (C \mid D) \,\&\, E \,\&\, F) \mid ((A \mid B) \,\&\, C \,\&\, D \,\&\, (E \mid F)) \mid \mid (A \,\&\, B \,\&\, (C \mid D) \,\&\, (E \mid F)) \mid (C \,\&\, D \,\&\, E \,\&\, F) \mid \mid (A \,\&\, B \,\&\, E \,\&\, F) \mid (A \,\&\, B \,\&\, C \,\&\, D) \tag{18}$$

Based on these summators, module for 32-bits numbers generation has been created.

4.2. Parallel generator

Parallel generator has been created by serial connection of simple generator modules (see Fig. 1), where first module has updated with a clock signal register, whereas remained modules are updated continuously (responsive to changes in input).

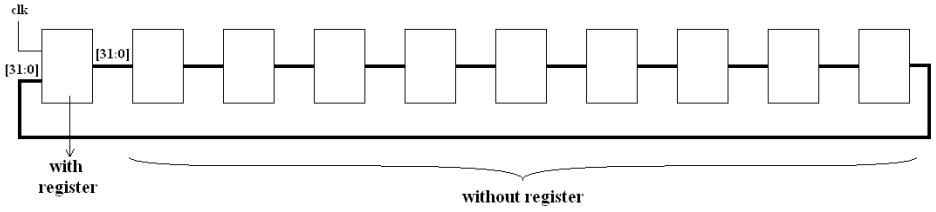

Fig. 1. Modules connection for parallel random numbers generation

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 16.725ns and for "fast model" – 7.338ns. One can assumed that the maximum time generation of pseudo-random numbers on real FPGA unit will be between the values obtained from simulations.

For parallel generation of 10 random numbers, steady output has been obtained after maximum 41.034ns for "slow model", and 16.634ns for "fast model" (100000 consecutive numbers have been calculated).

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

6. Summary

The method of generating pseudo-random numbers by an appropriate choice of generator parameters has been proposed in the paper. Simultaneously one should takes 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.

With the combination of modules one can obtain more random number, however in this case system clock (used in algorithm) must have lower frequency (e.g. 10 MHz).

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] Kozierski P., Lis M., Królikowski A., Implementation of Fast Uniform Random Number Generator on FPGA, Poznan University of Technology Academic Journals, Iss. 80, 2014, pp. 167-173.

[4] 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.

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

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

[7] 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.

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