LINEAR FEEDBACK SHIFT REGISTER GENETICALLY ADJUSTED FOR SEQUENCE COPYING

Ugalde-Franco Juan Manuel, Martinez-Gonzalez Ricardo Francisco
and Mejia-Perez Juan Francisco

Electrics – Electronics Department, TecNM – Instituto Tecnologico de Veracruz,
Veracruz, Mexico

ABSTRACT

The present manuscript proposes a comparison between two types of shift registers; the first one, a traditional one with a linear feedback (LFSR); on the other corner, a register designed one register with genetically-controlled feedback loop (NLFGR). Being more traditional, the linear feedbacked register works as reference to offer comparison metrics, meanwhile the NLFGR is our research focus, since our proposal relies in its good features to replicate certain behaviours, with shorter parsing time than linear feedback shift register.

KEYWORDS

Lineal Feedback Shift Register, Non-linear Feedback Genetically-controlled Register, Heuristic search, Genetic Algorithms, Sequence reproduction.

1. INTRODUCTION

Generating pseudorandom sequences has application in many fields, since cryptography, steganography, encoders and decoders for different communication protocols [1], [2], as a security core used in satellite networks for cell phone communications [3], [2], even into Monte Carlo test applications, developing and simulation environments [4].

A Pseudo Random Binary Sequence (PRBS) generator is an n-bit binary number generator circuit that make a number per clock cycle for 2^{n-1} clock cycles [5]. Popularly a PRBS is implemented as a linear feedback shift register [6], [7], [8]. Expressed in simple terms, a PRBS refers to its function, while an LFSR is described as the implemented circuit. When an LFSR includes non-linear elements such as genetic algorithms, it evolves from LFSR to become a non-linear feedback genetic register (NLFGR), which has interesting properties in randomness fields for generating sequences not deeply explored. However to exploit its full potential it has to be simulated in software and implemented in hardware to evaluate its efficiency. A hardware implementation in a FPGA fulfils translation from a high-level language as Python, that is a very useful tool for testing draft designs

In this paper, we will discuss a comparison between LFSR and NLFGR features to replicate a random sequences. Firstly, an LFSR and NLFGR is implemented using Python, and generating a specific sequence, then we compare parsing times to estimate efficiency as generating engine

1.1. Linear feedback shift register (LFSR)

It is a shift register whose input bit is a linear function of its previous state; the most commonly used linear function for handling individual bits is the exclusive-or (XOR). Therefore, an LFSR is often a shift register whose input bit is driven by an XOR function of some bits of the same shift register, creating a closed loop. Figure 1 shows a schematic that can help to illustrate prior statement.

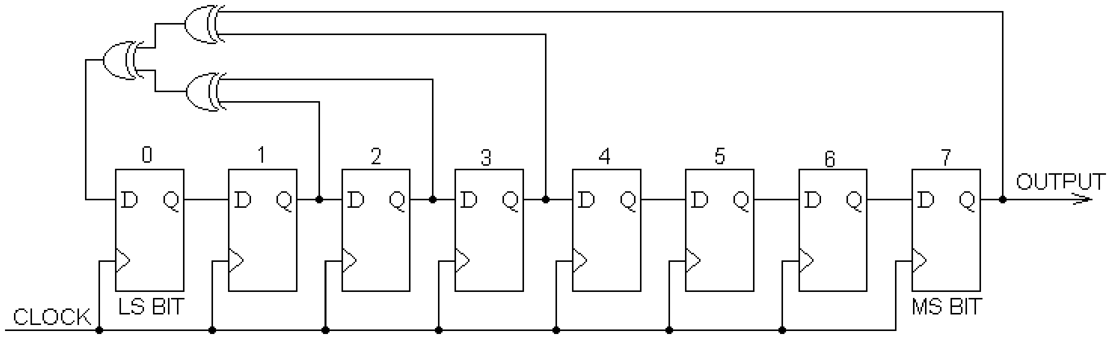

Figure 1. Example for 8-bits LFSR

In its operation, the initial value of the LFSR is called seed, and it must be different from '0'; since the operation of the register is deterministic, and its produced values are completely locked to its current state. In the same way, because the register has a finite number of possible states, it must eventually enter into a repeating cycle.

The bits in the LFSR that influence the input are called taps. An LFSR produces a 2^{n-1} sequence of states within itself, except for the '0' combination.

1.2. Genetic Algorithms

This is an iterative search technique inspired by the principles of natural selection. The GA does not seek to model biological evolution but they origin optimization strategies. The concept is based on the generation of individuals by the reproduction of their parents, and so on. During the course of evolution, genes with slow evolution were replaced by genes with better evolutionary adaptability; therefore, highly efficient strategies are expected in the individuals.

Many problems have complex objective functions, and their optimization tends to end at local minimums-maximums ends. The idea of the GA is to optimize, by finding the maximum or minimum of an objective function, for such, the principles of natural selection are used on the function parameters.

The components of a genetic algorithm are:

- A function that you want to optimize.
- A group of candidates for the solution.
- An evaluation function that measures how candidates optimize the function.
- Mating function.

Simplifying, the GA contains genes (initial population), genetic evaluation (fitness function), reproduction of alleles or gene values (crossover or recombination), allele mutation (randomly changing a bit, or by using a specific function), until satisfy the aptitude function or exceeding certain number of iterations.

2. STATE OF ART

The pseudo-random number generation technique implemented by a linear feedback shift register (LFSR), or an LFSR arrangement, for granting non-linear attributes [9], [10], [11] at the data encryption moment, it gives an important impact in security area, since it improves vulnerabilities in the systems, highly increasing the security levels [2], [12]. However, the LFSR, at being a linear device, it remains susceptible to attacks.

Antagonistically, cryptanalysis has shown that if the noisy prefix of the output sequence is known, it is possible to reconstruct the initial state of the LFSR by a generalized correlation attack. This kind of attack is based on resolution of the restricted edition distance between sequences, and such is determined by the initial state of shift registers and intercepted noisy output sequence. Where FPGAs have been successfully applied for cryptanalytic purposes, particularly in the exhaustive search of keys which is a highly parallelisable task [13].

One of the proposals is to make the systems more robust, the passwords or keys must be analysed; for such activity, cryptanalysts require ways to automate the process so that cryptographic systems can be more efficiently tested. Evolutionary algorithms provide one of these features since they are able to search optimal global solutions very quickly. The cuckoo search algorithm (Cuckoo Search, CS) has been effectively used in cryptanalysis of conventional encrypted systems [14].

Another way to find satisfactory solutions to these problems is to use heuristic goals. That is an algorithm designed to solve a wide range of optimization problems without having to deeply adjust after each problem [15].

3. DEVELOPING THE USED MODEL

3.1. LFSR

Taking as a core an 8-bit register, with taps in bits 6, 4, 3 and 1 with right-side shifting, the LFSR shown in Figure 2 is obtained.

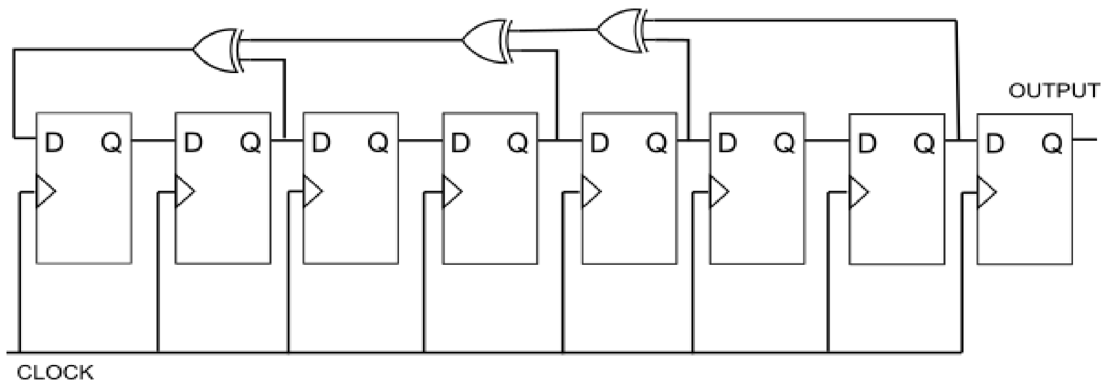

Figure 2. LSFR with taps in bits 6, 4, 3 and 1

Which will be used as a model, to be implemented in high-level language Python, and then implemented in an FPGA using VHDL language.

Its operating structure will be described in the flow diagram shown in Figure 3, but not before explaining the variables used in diagram. N = Number, K = Iteration control, B = Search , NA = Random number, cor = LFSR function.

To start operation, initialize K = 0, then request N that is necessary for program startingand basically it can be one of two options; enter ‘0’ will look for a default value "10101010" or when entering another number, the program will require an 8-bit binary number to be assigned to B, then an 8-bit random number is generated, which will enter in a while loop that applies the function *cor* to NA and adds 1 in each K cycle until variable B is equal to the variable NA or it exceeds the 1000 iterations.

From the brief previous description, it should be noted that the function *cor* is essentially the LFSR with the features of Figure 3. N is the element to find, and NA is the initial seed for the LFSR operation.

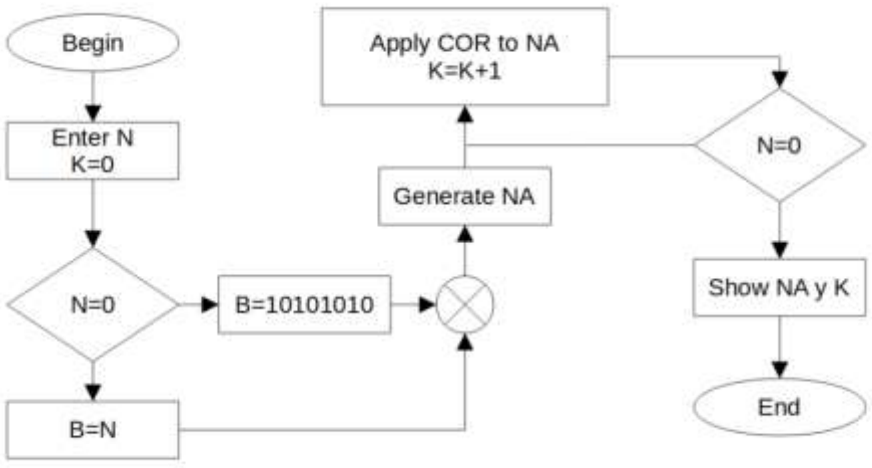

Figure 3. Flow chart of LFSR behaviour

3.2. NLFGR

Taking as a core an 8-bit register with an evaluation function for the mating execution and mutation function in each cycle, the whole process is repeated until reach stop criterion. The processing loop is depicted in the block diagram in Figure 4.

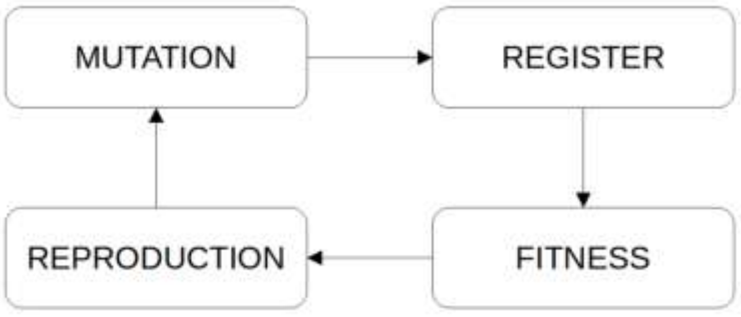

Figure 4. Flow chart for the proposed genetic algorithm

Its operating structure will be described in the flow diagram shown in Figure 5, but not before explaining the used variables of the diagram. N = Number, K = iteration control, B = Search, NA = Random Number, and Fitness, Reproduction and Mutation = function NLFGR.

To star operation, initialize K=0 then request N that it is necessary for program starting, and it also consists of two options. Entering ‘0’, operation will look for a default value "10101010"; meanwhile entering another 8-bit number, it will be assigned to B, and used for an 8-bit random number generation, which will enter in a while loop, where the fitness function is applied to NA, returning eight parameters indicating 1 when B=NA or 0 when they are different in each bit; afterwards, the mating function uses the parameters delivered by the fitness function to indicate evaluated population improvement, this to determine which pair of bits is more suitable to mating. Such activity involves crossing bits (10-01), or determine to which bit will be applied Mutation process; nevertheless this process needs to be non-recurrent; therefore, in each iteration, when data enters the function, it must satisfy two precedents, that states which data bits receives the NOT operation, and it must be in the Fitness parameters, and 0-to-7 random number must be equal to the bit weight after being inverted and being in the Fitness parameter, on the contrary, function will return an unchanged NA. After adding 1 to K in each cycle, until B equals NA or K reach fifty iterations.

From the previous description, it is necessary to specify the optimization strategies that integrates the described genetic algorithm, where genes are the randomly generated number (NA), after introduce the searching parameters (B = N), the allele is each bit that integrates NA, the evaluation function is implemented by *Fitness,* where it is reflected the key of the genetic algorithm; in other words, the suitable points, the reproduction function is implemented by the same name function, that is responsable for crosssing the bits for a new generation. The mutation function is implemented by same name function that randomly adds parameters of Fitness, and avoids stagnation in the production of generations, attending a tendency toward improvement.

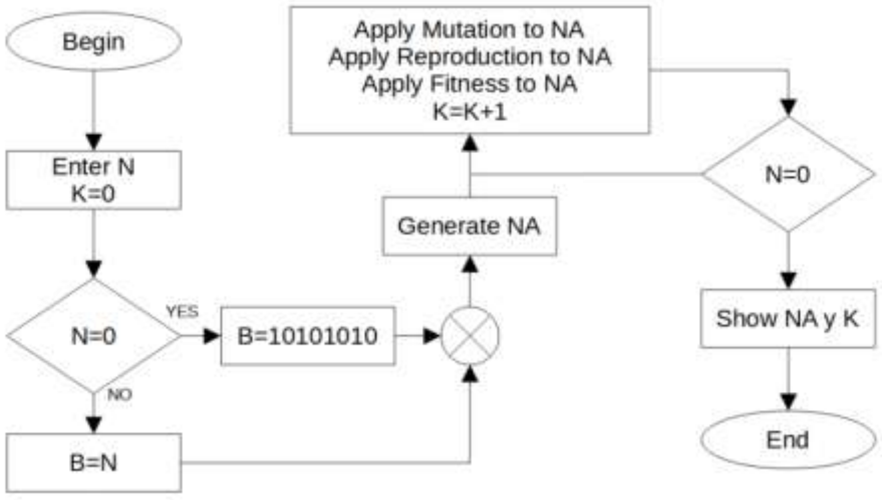

Figure 5. Flow chart of NLFGR behaviour

4. RESULTS

From the LSFR and NLFGR algorithms, a series of fifty tests with the same word was realised to reach "10101010" for measuring its operation performance and the following was determined:

- NLFGR consumes 10 times fewer iterations to find the result from a random seed.
- LFSR consumes 20 times more time to find the result from a random seed.
- The normal distribution of the two linear feedback shift registers show us the number of iterations respectively to the standard deviation and its average result, verifying with its graph the stability and operational performance under an observation period.
- The Table 1 shows a performance (seed-result) between experiment and iteration number with regard to the two algorithms operating with a random seed.

Table 1. General results of the experiment

Average of Iterations and Computational Time to Find Results		
Element	**Iteration**	**Time**
LFSR	121.88	261.8 ms
NLFGR	19.9	10.7 ms

4.1. Statistical distribution LFSR

With reference to a normal distribution, the equation 1 is applied to the data obtained from the realised experiments, the general behaviour of the device can be graphically determined.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mu)^2}{2\sigma^2}} \qquad (1)$$

Where:
μ = average
π= Pi constant
σ= typical deviation
σ^2= variance
e = Euler constant
x = data

We observe the normal distribution in Figure 6 with a standard deviation of 75.8 iterations, a mean value of 121.88 iterations, a variance of 5745.64. The x-axis represents the data, meanwhile the y-axis density value. After graphing the Gauss bell, we can determine:

• The number of iterations needed to find a result, in general, is from 101 to 144 iterations.
• The consumption of minimum iterations is not a characteristic of this device, because it depends properly on its initial state.

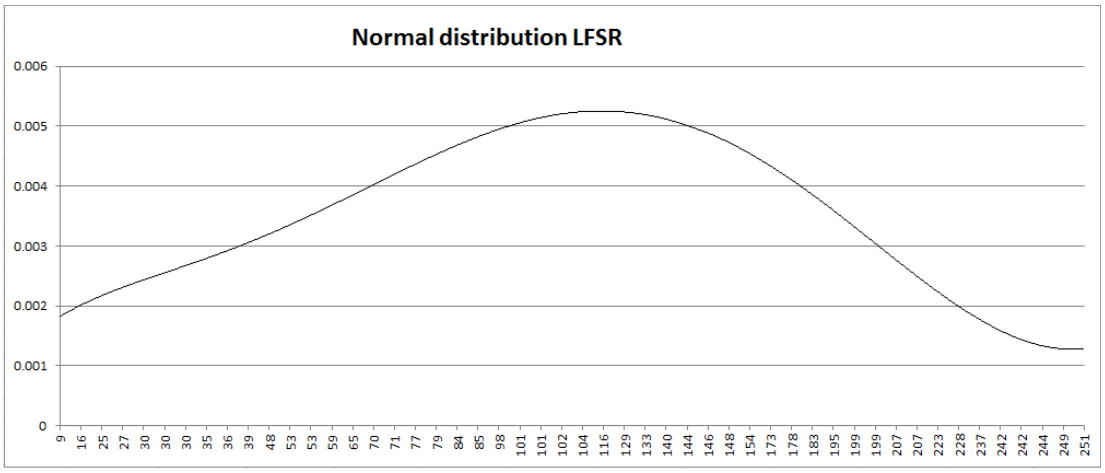

Figure 6. Normal distribution of an LFSR

4.2. Statistical distribution NLFGR

We observe the normal distribution in Figure 7 with a standard deviation of 8.90 iterations, a mean value of 10.9 iterations, a variance of 79.21. The x-axis represents data, and y-axis the density value. After graphing, the Gauss bell you can determine:

• Few iterations are required for obtaining a satisfying result.
• The number of iterations needed to find a result, in general, is between 7 to 17.

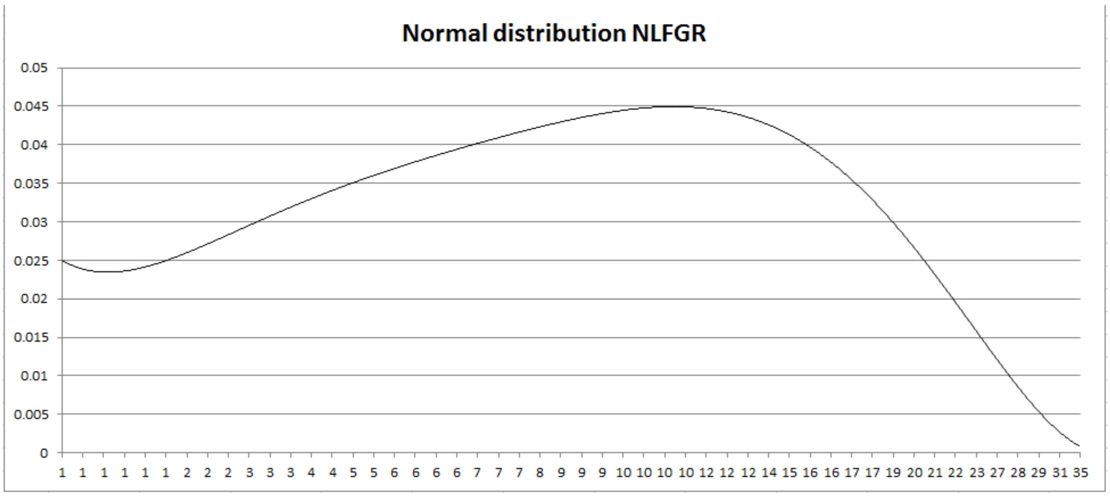

Figure 7. Normal Distribution of an NLFGR

4.3. Graphical response of the experiment-number of iterations relation

The Figure 8 depicts relation between the experiment and number of iterations necessary to reach control result, it highlights the reduced number of iterations necessary to maintain or success the NLFGR result, with regard to the conventional LSFR.

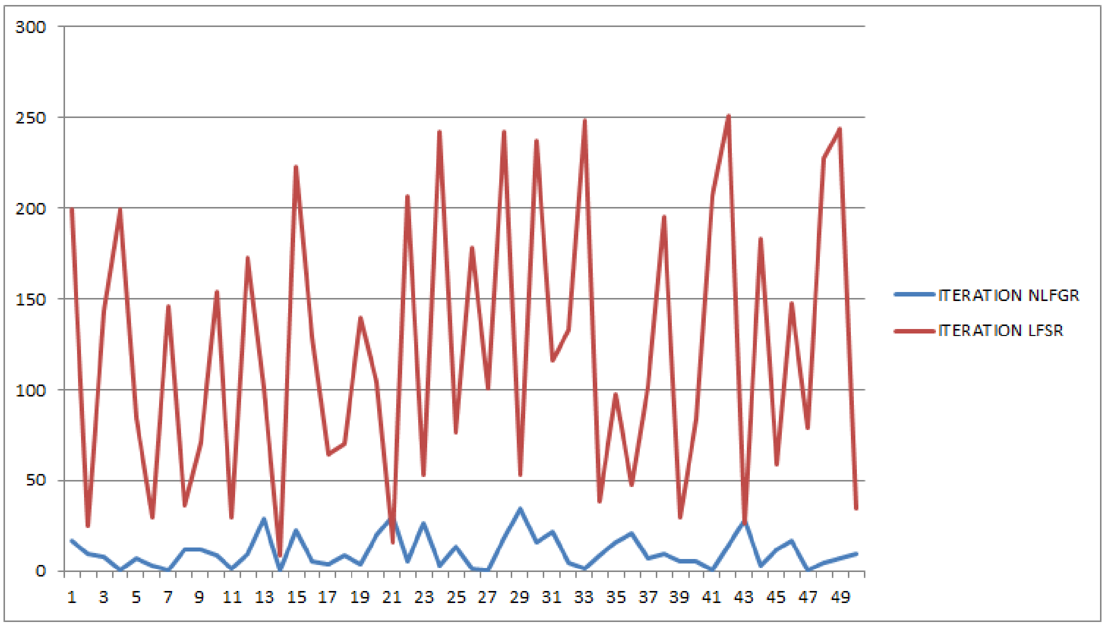

Figure 8. Response experiment-iteration

4.4. Graph for the computational time

The relationship between experiment and the needed time to find the control result is depicted in Figure 9. We can highlight the efficiency of NLFGR with respect to the conventional LFSR, and its tendency to use a same computational time.

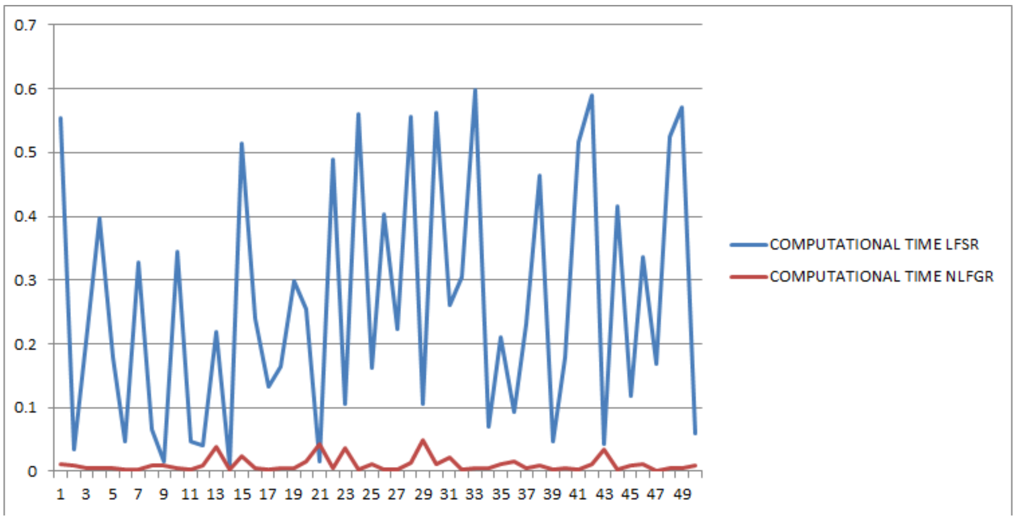

Figure 9. Comparison of LFSR and NLFGR computational times

4.5. VHDL Simulation

We recreate the Python coding in VHDL language, and it implements the same behaviour in a FPGA developing board. In Table 2, we present output sequences for both, Python and VHDL, codes using same seed. It gives congruent results that represents certainty about VHDL implementation. The Python data was collected from the console where the program runs.

Table 2. Results of the VHDL simulation

Output sequences		
Iteration	**Python**	**VHDL**
1	10100011	10100011
2	01010001	01010001
3	10101000	10101000
4	01010100	01010100
5	10101010	10101010

With the Python data, the next step is the VHDL simulation, which is presented in Figure 10. The upper signal is the clock signal; meanwhile, the next signal is our interested one, NLFGR output. Values are delivered each rising edge of clock signal, but for the last one, that occurs because such data is our desired one, so the systems do not generate any other result.

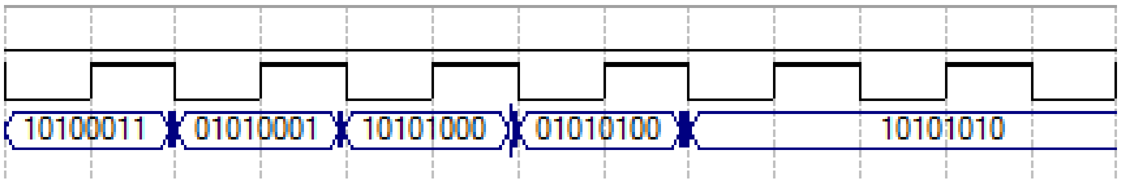

Figure 10. Simulation of the sequence delivered from the NLFSR described in VHDL code

Having explained the previous process, it can be confirmed that both designs exhibit the same behaviour.

5. CONCLUSION

It is determined that an NLFGR is more efficient than an LFSR to find a result, since its working features are focused on optimization strategies but its initial state. It is highlighted that elements obtained by experimentation show a number of iterations relative low, constant and consistent to satisfactorily find values. Another remarkable point is the consistency in the output data parsing time for the NLFGR algorithm, because it reaches the goal output around the same number of iterations for each try.

Nevertheless their performance, NLFGRs have some limitations. They need prior adjustment to tune, and works adequately. With no adjustment, the register simply does not converge. Another limitation is the complexity, in fact it need lesser time to converge, but each parsing cycle requires more operation.

Finally, we can express that following top-bottom design paradigm is reliable, since we started at high-level language generator algorithm, for next describing the same algorithm in low-level design as VHDL, and both succeed to gives correct data, giving confidence and reliability to our proposal.

REFERENCES

[1] MAO, Yaobin; CAO, Liu; LIU, Wenbo. Design and FPGA implementation of a pseudo-random bit sequence generator using spatiotemporal chaos. En 2006 International Conference on Communications, Circuits and Systems. IEEE, 2006. p. 2114-2118.

[2] ZENG, Guang; DONG, Xiaodai; BORNEMANN, Jens. Reconfigurable feedback shift register based stream cipher for wireless sensor networks. IEEE Wireless Communications Letters, 2013, vol. 2, no 5, p. 559-562.

[3] SHAH, Trishla; UPADHYAY, Darshana. Design analysis of an n-Bit LFSR-based generic stream cipher and its implementation discussion on hardware and software platforms. En Proceedings of the International Congress on Information and Communication Technology: ICICT 2015, Volume 2. Springer Singapore, 2016. p. 607-621.

[4] JUSTIN, Remya; MATHEW, Binu K.; ABE, Susan. FPGA implementation of high quality random number generator using LUT based shift registers. Procedia Technology, 2016, vol. 24, p. 1155-1162.

[5] BABITHA, P. K.; THUSHARA, T.; DECHAKKA, M. P. FPGA based N-bit LFSR to generate random sequence number. International Journal of Engineering Research and General Science, 2015, vol. 3, no 3, p. 6-10.

[6] ZULFIKAR, Zulfikar; AWAY, Yuwaldi; RAFIQA, Shahnaz Noor. FPGA-based design system for a two-segment Fibonacci LFSR random number generator. Int. J. Electr. Comput. Eng.(IJECE), 2017, vol. 7, p. 1882.

[7] FÚSTER-SABATER, Amparo; CARDELL, Sara D. Cryptographic Properties of Equivalent Ciphers. Procedia Computer Science, 2016, vol. 80, p. 2236-2240.

[8] QU, Bo, et al. Differential power analysis of stream ciphers with LFSRs. Computers & Mathematics with Applications, 2013, vol. 65, no 9, p. 1291-1299.

[9] MA, Zhen; QI, Wen-Feng; TIAN, Tian. On the decomposition of an NFSR into the cascade connection of an NFSR into an LFSR. Journal of Complexity, 2013, vol. 29, no 2, p. 173-181.

[10] PEINADO, Alberto; FÚSTER-SABATER, Amparo. Generation of pseudorandom binary sequences by means of linear feedback shift registers (LFSRs) with dynamic feedback. Mathematical and Computer Modelling, 2013, vol. 57, no 11-12, p. 2596-2604.

[11] HU, Chunqiang; LIAO, Xiaofeng; CHENG, Xiuzhen. Verifiable multi-secret sharing based on LFSR sequences. Theoretical Computer Science, 2012, vol. 445, p. 52-62.

[12] SHINY, M. I., et al. LFSR based secured scan design testability techniques. Procedia computer science, 2017, vol. 115, p. 174-181.

[13] BOJANIĆ, Slobodan, et al. FPGA for pseudorandom generator cryptanalysis. Microprocessors and Microsystems, 2006, vol. 30, no 2, p. 63-71.

[14] DIN, Maiya, et al. Applying Cuckoo Search for analysis of LFSR based cryptosystem. Perspectives in Science, 2016, vol. 8, p. 435-439.

[15] BOUSSAÏD, Ilhem; LEPAGNOT, Julien; SIARRY, Patrick. A survey on optimization metaheuristics. Information sciences, 2013, vol. 237, p. 82-117.