FPGA Implication of the LUT-SR Family for Uniform Random Number Generation

M.V.Vyawahare, Rita Rawate
Department of Electronics Engineering,
Priyadarshini College of Engineering, Nagpur

ABSTRACT

Field-programmable gate array (FPGA) optimized random number generators (RNGs) can take advantage of bitwise operations and FPGA-specific features, hence they are more resource-efficient than software-optimized RNGs. This paper describes a type of RNG called a LUT-SR RNG, which takes advantage of bitwise XOR operations and the ability to configure lookup tables (LUTs) into decoders & shift registers of varying lengths. This provides good quality compared to others. The LUT-SR generators is implemented by using VHDL (very high speed integrated circuit hardware description language).

KEYWORDS

Equidistribution, field-programmable gate array (FPGA), uniform random number generator (RNG).

1. INTRODUCTION

MONTE CARLO applications are ideally suited to field-programmable gate arrays (FPGAs) because of the highly parallel nature of the applications, and because it is possible to take advantage of hardware features to create very efficient random number generators (RNGs). In particular, uniform random bits are extremely cheap to generate in an FPGA, as large numbers of bits can be generated per cycle at high clock rates using lookup tables [1], or first-in-first- out (FIFO) queues [2]. In addition, these generators can be customized to meet the exact requirements of the application, both in terms of the number of bits required per cycle, and for the FPGA architecture of the target platform.

Despite these advantages, FPGA-optimized generators are not widely used in practice, as the process of constructing a generator for a given parameterization is time consuming, in terms of both developer man hours and CPU time. While it is possible to construct all possible generators ahead of time, the resulting set of cores would require many megabytes, and be difficult to integrate into existing tools and design flows. Faced with these unpalatable choices, engineers under time constraints understandably choose less efficient methods, such as combined Tausworthe generators [3] or parallel linear feedback shift registers (LFSRs).

This paper describes a family of generators which makes it easier to use FPGA-optimized generators by providing a simple method for engineers to instantiate an RNG that meets the specific needs of their application. Specifically, it shows how to create a family of generators called LUT-SR RNGs, which use LUTs as shift registers to achieve high quality and long periods, while requiring very few resources. The main contributions of this paper are as follows:

1) A type of FPGA-optimized uniform RNG called a LUT-SR generator is presented which uses LUT-based shift registers to implement generators with periods of $2^{1024}-1$ or more, using two LUTs and two flip flops (FFs) per generated random bit.
2) An algorithm for describing LUT-SR RNGs using five integers is given, along with a set of open-source test benchs and tools.
3) Tables of 60 LUT-SR RNGs are presented, covering output widths from 32 up to 624, with periods from $2^{1024}-1$ up to $2^{19937}-1$.
4) A theoretical quality analysis of the given RNGs in terms of equidistribution and a comparison with other software and hardware RNGs are carried out.

2. BINARY LINEAR RNGS

Binary linear recurrences operate on bits (binary digits), where addition and multiplication of bits is implemented using exclusive-or ($\oplus$) and bitwise-and ($\otimes$). The recurrence of an RNG with n-bit state and r-bit outputs is defined as:

$$\mathbf{x}_{i+1} = \mathbf{A}\mathbf{x}_i \quad (1)$$

$$\mathbf{y}_{i+1} = \mathbf{B}\mathbf{x}_{i+1} \quad (2)$$

where $\mathbf{x}_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,n})^{\mathrm{T}}$ is the n-bit state of the generator, $\mathbf{y}_i = (y_{i,1}, y_{i,2}, \ldots, y_{i,r})^{\mathrm{T}}$ is the r-bit output of the generator, $\mathbf{A}$ is an $n \times n$ binary transition matrix, and $\mathbf{B}$ is an $r \times n$ binary output matrix. Because the state is finite, and the recurrence is deterministic, eventually the state sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ must start to repeat. The minimum value p such that $\mathbf{x}_{i+p} = \mathbf{x}_i$ is called the period of the generator, and one goal in designing RNGs is to achieve the maximum period of $p = 2^n - 1$. A period of 2^n cannot be achieved because it is impossible to choose $\mathbf{A}$ such that $\mathbf{x}_0 = \mathbf{0}$ maps to anything other than $\mathbf{x}_1 = \mathbf{0}$. This leads to two sequences in a maximum period generator: a degenerate sequence of length 1 which contains only zero, and the main sequence which iterates through every possible nonzero n-bit pattern before repeating. A necessary and sufficient condition for a generator is to have maximum period.

3. LUT-OPTIMIZED (LUT-OPT) RNGS

LUT-OPT generators [1] are a family of generators with a matrix A where each row and column contains $t-1$ or t *1s*. In hardware terms, this means that each row maps to a $t-1$ or t input XOR gate, and so can be implemented in a single t input LUT. Thus if the current vector state is held in a register, each bit of the new vector state can be calculated in a single LUT, and an r-bit generator can be implemented in r fully utilized LUT-FFs. The basic structure of a LUT-OPT generator is shown in Fig. 1.

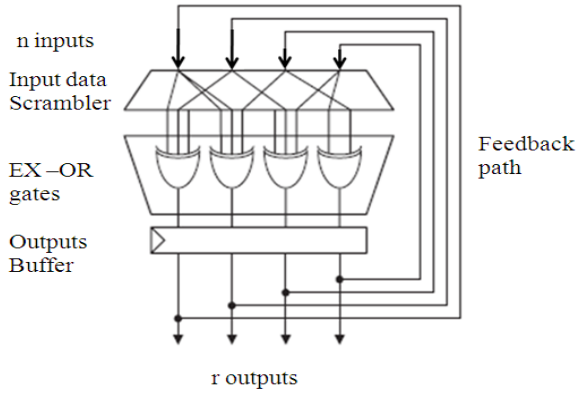


Fig. 1. LUT-Optimized (LUT-OPT) RNG.

A simple example of a maximum period LUT-OPT generator with $r = 6$ and $t = 3$ is given by

$$\mathbf{A} = \begin{bmatrix} 0\,1\,1\,0\,0\,0 \\ 0\,1\,1\,0\,0\,1 \\ 0\,1\,0\,1\,0\,0 \\ 1\,0\,0\,0\,1\,0 \\ 1\,0\,0\,0\,0\,1 \\ 1\,0\,0\,1\,1\,0 \end{bmatrix}, \quad \begin{bmatrix} x_{i+1,1} \\ x_{i+1,2} \\ x_{i+1,3} \\ x_{i+1,4} \\ x_{i+1,5} \\ x_{i+1,6} \end{bmatrix} = \begin{bmatrix} x_{i,2} \oplus x_{i,3} \\ x_{i,2} \oplus x_{i,3} \oplus x_{i,6} \\ x_{i,2} \oplus x_{i,4} \\ x_{i,1} \oplus x_{i,5} \\ x_{i,1} \oplus x_{i,6} \\ x_{i,1} \oplus x_{i,4} \oplus x_{i,5} \end{bmatrix}$$

LUT-OPT generators have two key advantages.

1) Resource efficiency: Each additional bit requires one additional LUT and FF, so resource usage scales linearly, and generating r bits per cycle requires r LUT-FFs.
2) Performance: The critical path in terms of logic is a single LUT delay, so the generators are extremely fast, so usually the clock net is the limiting factor, with routing delay and congestion only becoming a factor for large n.

Some disadvantages of LUT-OPT generators are following:

1) Complexity: Each (r, t) combination requires a unique matrix of connections, which must be found using specialized software. If these matrices are randomly constructed (as in previous work), then it is difficult to compactly encode these matrices, so it is difficult for FPGA engineers to make use of the RNGs.
2) Quality: The random bits are formed as a linear combination of random bits produced in the previous cycle— when $t = 3$, some of the new bits will be a simple two-input XOR of bits from the previous cycle. The input of this lag-1 linear dependence is minimal in modern FPGAs where $t \geq 5$, and also diminishes quickly as r is increased, but remains a source of concern.
3) Period: In order to achieve a period of $2^n - 1$, it is necessary to choose $r = n$, even if far fewer than n bits are needed per cycle. An absolute minimum safe period for a hardware generator is $2^{64} - 1$, but it is preferable to have much larger periods of $2^{1000} - 1$ or more.
4) Seeding: It is necessary to initialize RNGs with a chosen state at run time, so that different hardware instances of the same RNG algorithm will generate different random streams. In a LUT-optimized generator, it is possible to implement serial loading of state using one LUT input per RNG bit to select between RNG and load mode, but in practice, for a randomly chosen matrix A, only parallel loading is possible.

3. LUT-FIFO RNGS

One way of removing the quality and period problems is provided by LUT-FIFO generators [2]. These augment the r bits of state held in FFs with an additional depth-k width-w first-in-first-out (FIFO), for a total period of $2^n - 1$, where $n = r + wk$, shown in Fig. 1(b). LUT-FIFO generators can provide long periods such as $2^{11213} - 1$ and 2^{19937}.

Some disadvantages are following:

1) For reasonable efficiency, the FIFO needs to be implemented using a block RAM, a relatively expensive resource which one would usually prefer to use elsewhere in a design.
2) The wordwise granularity of block-RAM-based FIFOs reduces the flexibility in the choice of r, as it can only be varied in multiples of k.

These are mild disadvantages when compared to the quality and period problems of LUT-optimized generators that have been eliminated, but LUT-FIFO generators also make the problems of complexity and efficient initialization slightly worse. If extremely high quality and period are needed, then LUT-FIFO generators present the fastest and most efficient solution, but few applications actually require such high levels of quality, particularly given the need for expensive block- RAM resources.

4. SOFTWARE RNGS

In addition to the hardware-optimized LUT-OPT and LUT-FIFO generators, a number of generators designed for software architectures have been ported to FPGA architectures.

Combined Tausworthe [3]—Software generators which use word-level shift, XOR, and AND operations to construct simple recurrences with distinct periods, which are then combined using XOR to produce a much longer period generator.

Mersenne Twister [5]—This uses the same word-level oper- ators as the Combined Tausworthe, combined with a large RAM-based queue, to create a software generator with a fairly good equidistribution and the extremely long period of $2^{19937} - 1$.

WELL [10]—This generator uses techniques similar to the Mersenne Twister, but uses a more complex recurrence step involving multiple memory accesses per sample, to achieve the maximum possible equidistribution at the same period as the Mersenne Twister.

All the software generators are designed with word-level instructions in mind, and so tend to be inefficient in terms of resources consumed per bit generated.

5. LUT-SR RNG

This RNG will have the time period very longer than the available RNG's. This RNG will be faster than other RNG's. This RNG will be will have very less complexity than LUT-FIFO RNG. It does not require block of RAM. But it is the improved form of LUT –SR RNG.

LUT-SR generator It fixes all problems related to complexity and serial seeding found with both generators, and provides much higher periods than LUT-OPT generators for a cost of

one extra LUT-FF per bit, while eliminating the block- RAM resource needed for an LUT-FIFO RNG. LUTs can be configured in a number of different ways, such as basic ROMs, RAMs, and shift registers. Configuring LUTs as shift registers provides an attractive means of adding more storage bits to a binary linear generator.
The improved LUT- SR RNG is shown in figure. 2.

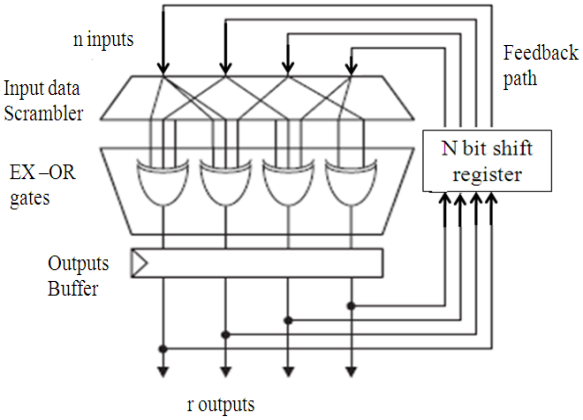


Fig. 2. Improved LUT-LUT SR RNG.

It has the time period $N2^r$ times than LUT- SR RNG. The block diagram shown above is described below:

1. There are "n" inputs initially "0000" (4 bits considered).
2. These n bits input are fed to the scrambler.
3. This scrambler block scrambles or randomizes the input at its output.
4. Hence we get the different output w.r.t input.
5. Then this output is fed to the EX –OR gates.
6. The "r" bit output of this block is very random w.r.t input data.
7. Then this "r" bit output is fed to the N bit shift register.
8. This N bit shift register shifts the output at every clock pulse.
9. This shifted data is then fed back to the input again.
10. But in this way the input will not have any correlation with output & with the feedback data.
11. Hence output pattern will be unpredictable as compared to the binary RNG.
12. Because the time period of the pattern repetition will be $N2^r$ times more compared to the binary RNG.
13. The simulation results are shown in figure below.

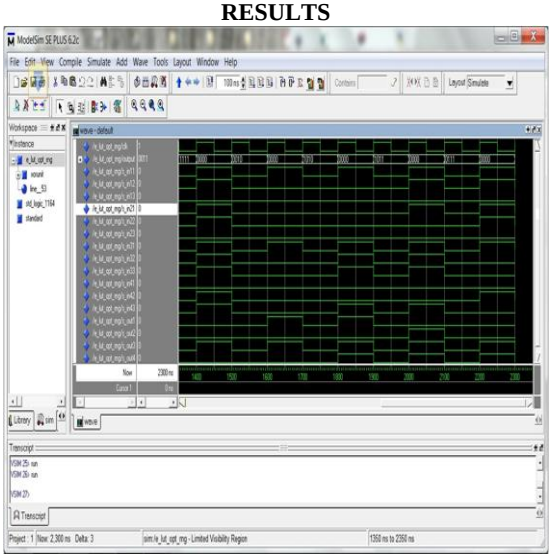
This RNG will increase the reliability of system as the time period of pattern repetition will be huge as compared to other RNG's. It will be used to implement in spread spectrum technique as PN sequence generator to increase the security.

CONCLUSION

This paper presents improved LUT-SR RNGs with implication on FPGA. These RNGs take advantage of LUTs to configure as independent shift registers, for realizing high-quality long-period generators.

The advantage of this LUT-SR generator over previous FPGA-optimized uniform RNGs will be less complexity than LUT-FIFO RNG, faster speed, huge time period of pattern repetition & no requirement of RAM block. In concert with the tables of maximum period generators, this allows FPGA engineers to use the new RNGs without needing to find generator instances themselves. This improved LUT-SR RNG has been realized by in VHDL using Modelsim.

SIMULATION RESULTS



REFERENCES

[1] D.b. Thomas and w.luk "FPGA optimized uniform random number generators using lut and shift registers "in proc.conf.feild program. logic appl.2010,pp 77-82.

[2] D. B. Thomas and W. Luk, "High quality uniform random number generation using LUT optimised state-transition matrices," J. VLSI Signal Process., vol. 47, no. 1, pp. 77–92, 2007.

[3] D. B. Thomas and W. Luk, "FPGA-optimised high-quality uniform random number generators," in Proc. Field Program. Logic Appl. Int. Conf., 2008, pp. 235–244.

[4] P. L'Ecuyer, "Tables of maximally equidistributed combined LFSR generators," Math. Comput., vol. 68, no. 225, pp. 261–269, 1999.

[5] D. B. Thomas and W. Luk, "FPGA-optimised uniform random number generators using luts and shift registers," in Proc. Int. Conf. Field Program. Logic Appl., 2010, pp. 77–82.

[6] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623- dimensionally equidistributed uniform pseudo-random number generator," ACM Trans. Modeling Comput. Simulat., vol. 8, no. 1, pp. 3–30, Jan. 1998.

[7] M. Saito and M. Matsumoto, "SIMD-oriented fast mersenne twister: A 128-bit pseudorandom number generator," in Monte-Carlo and Quasi-Monte Carlo Methods. New York: Springer-Verlag, 2006, pp. 607–622.

[8] F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," ACM Trans. Math. Software, vol. 32, no. 1, pp. 1–16, 2006.

[9] M. Matsumoto and Y. Kurita, "Twisted GFSR generators II," ACM Trans. Modeling Comput. Simulat., vol. 4, no. 3, pp. 254–266, 1994.

[10] P. L'Ecuyer and R. Simard. (2007). TestU01 Random Number Test Suite [Online].

Available:http://www.iro.umontreal.ca/~imardr/indexe.html.

[11] F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," ACM Trans. Math. Software, vol. 32, no. 1, pp. 1–16, 2006.

[12] V. Shoup. (1997, Jan. 15). NTL: A Library for Doing Number Theory [Online]. Available: http://www.shoup.net/ntl/

[13] M. Albrecht and G. Bard. (2010). The M4RI Library - Version 20100817 [Online]. Available: http://m4ri.sagemath.org

[14] S. Duplichan. (2003). PPSearch: A Primitive Polynomial Search Program [Online]. Available: http://users2.ev1.net/~sduplichan/ primitivepolynomials/

[15] V. Sriram and D. Kearney, "A high throughput area time efficient pseudo uniform random number generator based on the TT800 algorithm," in Proc. Int. Conf. Field Program. Logic Appl., 2007, pp. 529–532.

[16] S. Konuma and S. Ichikawa, "Design and evaluation of hardware pseudorandom number generator mt19937," IEICE Trans. Inf. Syst., vol. 88, no. 12, pp. 2876–2879, 2005.

[17] Y. Li, P. C. J. Jiang, and M. Zhang, "Software/hardware framework for generating parallel long-period random numbers using the well method," in Proc. Int. Conf. Field Program. Logic Appl., Sep. 2011, pp. 110–115.