Optimization of FPGA Architecture for Uniform Random Number Generator Using LUT-SR Family

Rita Rawate[1], M. V. Vyawahare[2]

[1]Nagpur University, Priyadarshini College of Engineering, Nagpur

[2]Professor, Priyadarshini College of Engineering, Nagpur

Abstract: ***Field-Programmable Gate Arrays (FPGAs) are widely used to implement logic without going through an expensive fabrication process. Field-programmable gate array optimized random number generators (RNGs) are more resource-efficient than software-optimized RNGs because they can take advantage of bitwise operations and FPGA-specific features. The software community has developed a number of high-quality, long period Random Number Generators (RNGs), some of which have been adapted for use in FPGAs. However, these generators were designed to meet the needs of word-level instruction processors, and so are less efficient when mapped to the bit-level operations available in FPGAs. This paper describes a type of FPGA RNG called a LUT-SR RNG, which takes advantage of bitwise XOR operations and the ability to turn lookup tables (LUTs) into shift registers of varying lengths. This provides a good resource–quality balance compared to previous FPGA-optimized generators. This paper deals with optimization of FPGA and simulations is done in VHDL.***

Keywords: FPGA (Field Programming Gate Array) , LUT(Look up table), LUT-SR(Look up table shift register),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). 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.

Random numbers have applications in many areas: simulation, game-playing, cryptography, statistical sampling, and evaluation of multiple integrals, particle transport calculations, and computations in statistical physics, to name a few [1,6]. Since each application involves slightly different criteria for judging the ―worthiness" of the random numbers generated, a variety of generators have been developed, each with its own set of advantages and disadvantages. Many applications are reliant on random numbers, such as financial calculations, simulated equipment test beds, and simulation of communications channels. Such applications require large amounts of processing power, while providing many opportunities to exploit fine-grain and coarse-grain parallelism, and so are often ideally suited to implementation in FPGAs [5, 7]. In order to function correctly, these applications require many parallel streams of high quality, large period, uncorrelated uniform random number generators. These are most commonly used as input to transformation functions which will provide the non-uniform distributions, and typically require many uniform input bits for each nonuniform output sample [1,2]. This paper explains a family of generators which makes it easier to use FPGA-optimized generators by given a simple method instantiate an RNG. This helps to achieve 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.

This paper is structured as follows. Section II presents general idea of field-programmable gate array. Section III introduces uniform random number generator. Section IV explains lut-opt RNGs .Section V gives idea about lut-FIFO RNGs. Section VI introduces LUT-SR RNGs. Finally Section VII deal with device utilization summery. Section VIII gives idea about comparison of generators by recourse usage and simulation results and synthesis report are given in section IX and X. Section XI concludes the paper.

2. Field Programming Gate Array (FPGA)

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing—hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC).

FPGAs can be used to implement any logical function that an ASIC can perform. The ability to update the functionality after shipping, partial re-configuration of the portion of the design and the low non-recurring engineering costs relative to an ASIC design, offer advantages for many applications[1,6]. FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "connected together"—somewhat like a one-chip programmable breadboard. Logic blocks can be configured to perform complex combinational functions, or merely simple logic like AND and NAND[8]. Figure 1 show The most common FPGA architecture which consists of an array of logic blocks (called Configurable Logic Block, CLB, or Logic Array Block, LAB, depending on vendor), I/O pads, and routing channels. Generally, all the routing channels have the same width (number of wires). Multiple I/O pads may fit into the height of one row or the width of one column in the array.

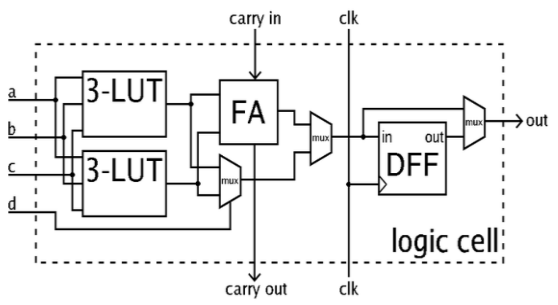


Figure 1: FPGA Architecture

In general, a logic block (CLB or LAB) consists of a few logical cells. A typical cell consists of a 4-input Lookup table (LUT), a Full adder (FA) and a D-type flip-flop, as shown. The LUT are in this figure split into two 3-input LUTs. In normal mode those are combined into a 4-input LUT through the left mux. In arithmetic mode, their outputs are fed to the FA. The selection of mode is programmed into the middle mux. The output can be either synchronous or asynchronous, depending on the programming of the mux to the right.

3. Uniform Random Number Geneartor (Rng)

Random values play a crucial role in several areas of science. In dependency on field of application the requirements for parameters of random sequence and generator of sequence itself may vary. Focusing on the sequence origin A random number generator (RNG) is a device designed to generate a sequence of numbers or symbols that don‗t have any pattern. Hardware-based systems for random number generation are widely used, but often fall short of this goal, albeit they may meet some of the statistical tests for randomness for ensuring that they do not have any ―de-codable‖ patterns. Methods for generating random results have existed since ancient times, including dice, coin flipping, the shuffling of playing cards, the use of yarrow stalks and many other techniques. [1], [2].

The many applications of randomness have led to many different methods for generating random data. These methods may vary as to how unpredictable or random they are, and how quickly they can generate random numbers. [3,6].

A. Binary Linear RNGs

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

$$\mathbf{x}_{i+1} = \mathbf{A}\mathbf{x}_i$$
$$\mathbf{y}_{i+1} = \mathbf{B}\mathbf{x}_{i+1}$$

where $\mathbf{x}_i = (x_{i,1}, x_{i,2}, \dots, x_{i,n})^T$ is the n-bit state of the generator, $\mathbf{y}_i = (y_{i,1}, y_{i,2}, \dots, y_{i,r})^T$ 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 $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots$ 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.

B. Linear feedback shift register

The LFSR (Linear Feedback Shift Register) is the most direct form of binary linear recurrence, as it simply implements the characteristic polynomial. For example, the polynomial x6 + x5 + x0 translates to the state transition function:

$$s_{i+1}^{T} = \left[s_{i,5} \oplus s_{i,6},\ s_{i,1},\ s_{i,2},\ s_{i,3},\ s_{i,4},\ s_{i,5} \right]^{T}$$

Note that only one bit, s1,i+1, represents a ―new” value. All the other state bits are simply shifted copies of values from the previous state. So although in principle the LFSR generates n bits per step, only one of them is actually useful. In the rest of this we will use the following terminology to refer to the two kinds of bits: ―Active Bits” are bits formed from a combination of two or more bits in the previous state, while ―FIFO Bits” are a direct copy of just one bit from the previous state. Only active bits can reasonably be considered as independent random bits, so the maximum number of random bits taken from an RNG per step is the number of active bits. An LFSR only has one active bit, so to generate w random bits per step it is necessary to use w separate LFSRs, and combine one bit from each . Unfortunately this means that wn bits of storage only produce a sequence of length 2^n −1, which is much less than the maximum possible period of 2^nw − 1.

LFSR has more active bits, they still have significant correlations and are completely unsuitable for use as independent random bits.

A linear feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state. the only linear function of single bits is XOR thus it is a whose input bit is driven by the exclusive or (xor) of some bits of the overall shift register value. The initial value of LFSR is called the seed, and because the operation of register is deterministic , the stream of values produced by the register is completely determined by its current (or previous) state. Likewise because the register has a finite number of possible state, it must eventually enter a repeating cycle. However a

LFSR with a well chosen feedback function can produce a sequence.

C. 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 operators 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.

4. LUT-Optimized (LUT-OPT) RNGs

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 mor.
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.

5. 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. LUT-FIFO generators can provide long periods such as $2^{11213} - 1$ and 2^{19937}.

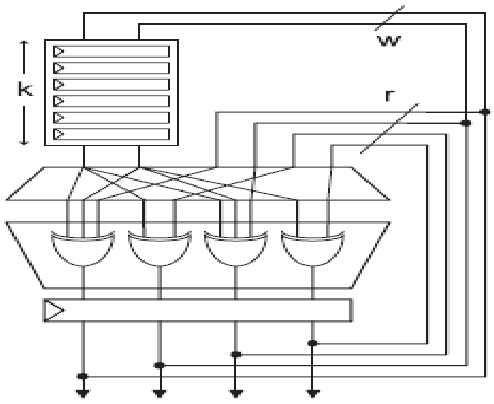


Figure 3: LUT-FIFO RNG

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.

6. LUT-SR

LUT-SR generator sits between the LUT-optimized and LUT-FIFO generators. 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[7]. 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.

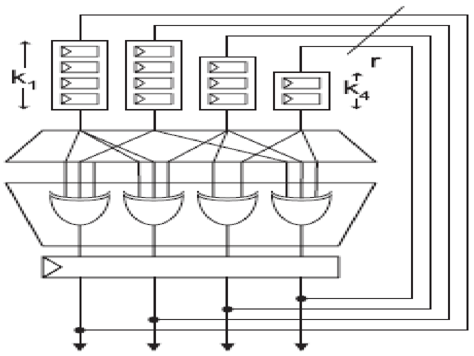


Figure 4: LUT-SR RNG

There are four stages to develop uniform random numbers.

6.1 Create Initial Seed Cycle

A cycle of length r is created through the r XOR gates at the output of the RNG. FPGA optimized uniform random number generator with a large period and with the ability to generate large quantities of uniform random numbers from a single seed.At this stage there are no FIFO bits, or equivalently there are r FIFOs of length is 0, is shown in Figure 6.1

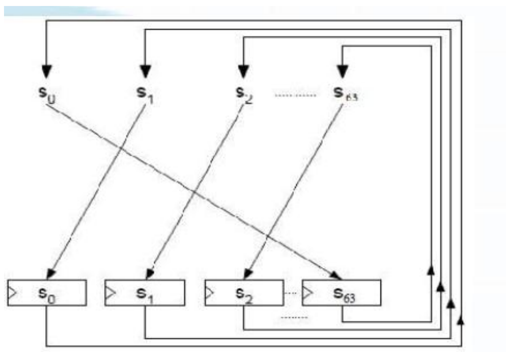


Figure 6.1: Create Seed Cycle

6.2. FIFO Extension

The cycle is randomly extended until a total cycle length of *n is reached, by randomly selecting a FIFO and increasing* its length by 1, while maintaining the known cycle is shown in Figure 6.2. A FIFO is a sequential data buffer that is very easy to use.Very small FIFOs can be implemented with flip flops or register arrays, sometimes even with shift registers.

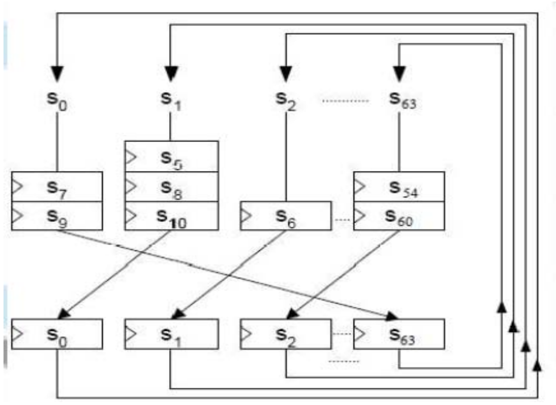


Figure 6.2: ADD LOADING CONNECTIONS

The known cycle is added to the —tap,” which describes the matrix. The cycle describes the FIFO connections completely, and also describes the first input to each of the *r XOR gates*

6.3. ADD XOR CONNECTIONS

The cycle provides one input for each of the XOR gates, so now the additional t – 1 random inputs are added over t−1 rounds. Each round is constructed from a permutation of the FIFO outputs, which ensures that at the end each FIFO output is used at most t times is shown in Figure 6.3. *Some bits will be assigned the same* FIFO bit in multiple rounds, and so will have fewer than *t inputs: to achieve a maximum period* generator, and also provides an entry point into the cycle for seed loading.

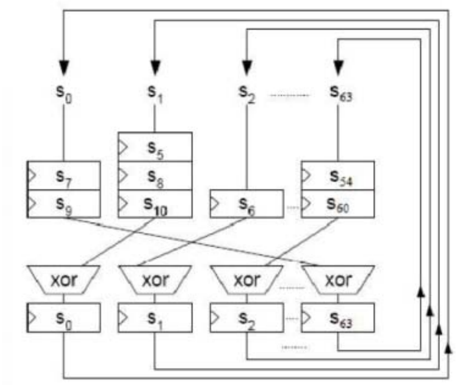


Figure 6.3: Add XOR Connections

6.4 Output Permutations

The simple dependency between adjacent bits is masked using a final output permutation is shown in Figure. Each permuted output bit is used at most times. Some bits will be assigned the same FIFO bit in multiple rounds. The XOR outputs are given to the SR and fed back to the FIFO extensions.

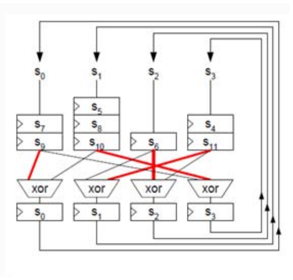


Figure 6.4: Random Permutation

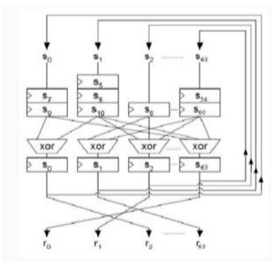


Figure 6.5: Output Permutation

The initial seed for an 8-bit RNG is given.

A shift register is an n-bit register that shifts its stored data by one bit position for every clock tick.

The resulting sequence is fed back to the FIFO SR.

Permutation of the resulting outputs is given to the XOR gates, where the XOR gate outputs are shifted and thus random number generation takes place successfully.

The same scheme is carried out for N bit RNG.

The permuted bits output is given to the XOR gates. For 8-bit RNG the number of XOR gates is 8(t=8).

The concept of permutation is used up for improving randomness among bits and thus employing unpredictability. The first and last bits are interchanged

The same concept of permutation is used for different bit RNGs. The permuted outputs are fed into the XOR gates and for remaining inputs to XOR gates round basis is used. The resulting outputs generate the random number cycle. The cycle is fed into the [FIFO] of varying lengths (length=k). The length should not exceed r. As each bit crosses the flip-flop, it will be set to zero.

Thus random number generation takes place.

The resulting random numbers are generated such that their period is 2^n -1.

The count of all zero state is reduced since the all zero state leads to idle condition.

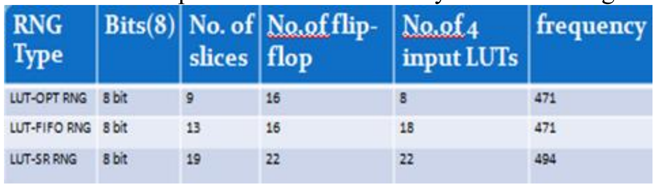
The period is the duration after which the entire sequence goes on repeating based on the initial seed and the permutations.

Register-Transfer-Level abstraction is used in VHDL languages for the formation of high level representation of the circuit and it clearly depicts the amount of LUTs used.

7. Device Utilization Summery

The device utilization summary results for 8-bit, RNG shows the number of (resources) flip-flops and LUTs utilized

Table 1: Comparison of Generators By Resource Usage

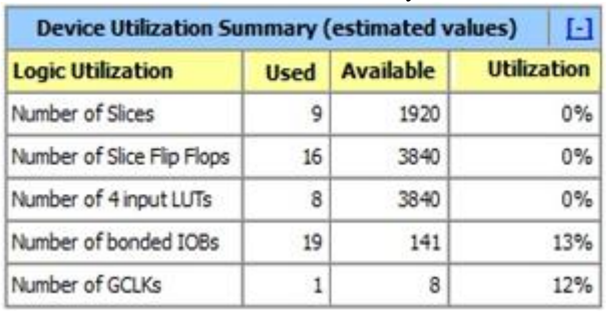


RNG Type	Bits(8)	No. of slices	No.of flip-flop	No.of 4 input LUTs	frequency
LUT-OPT RNG	8 bit	9	16	8	471
LUT-FIFO RNG	8 bit	13	16	18	471
LUT-SR RNG	8 bit	19	22	22	494

8. Comparison of Generators by Resource Usage

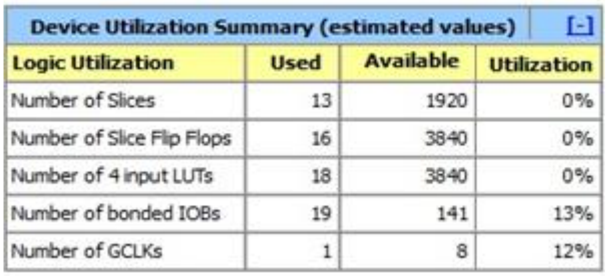
The device utilization summary results for 8-bit, RNG shows the number of (resources) flip-flops and LUTs utilized. The device utilization summary table is displayed by Xilinx Design Suite soon after the RTL implementation is completed.

Table 2: Device utilization summary of LUT-OPT RNG



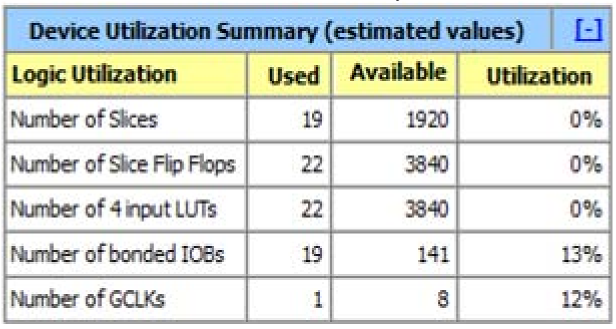
Device Utilization Summary (estimated values) [-]			
Logic Utilization	Used	Available	Utilization
Number of Slices	9	1920	0%
Number of Slice Flip Flops	16	3840	0%
Number of 4 input LUTs	8	3840	0%
Number of bonded IOBs	19	141	13%
Number of GCLKs	1	8	12%

Table 3: Device utilization summary of LUT-FIFO RNG



Device Utilization Summary (estimated values) [-]			
Logic Utilization	Used	Available	Utilization
Number of Slices	13	1920	0%
Number of Slice Flip Flops	16	3840	0%
Number of 4 input LUTs	18	3840	0%
Number of bonded IOBs	19	141	13%
Number of GCLKs	1	8	12%

Table 4: Device utilization summary of LUT-SR RNG



Device Utilization Summary (estimated values) [-]			
Logic Utilization	Used	Available	Utilization
Number of Slices	19	1920	0%
Number of Slice Flip Flops	22	3840	0%
Number of 4 input LUTs	22	3840	0%
Number of bonded IOBs	19	141	13%
Number of GCLKs	1	8	12%

9. Simualtion Result

The proposed method is simulated in VHDL .Figure 1 shows simulation result of LUT-OPT RNG. Figure 2 shows simulation result of LUT-FIFO RNG. and Figure 3 shows the simulation result of LUT-SR RNG respectively.

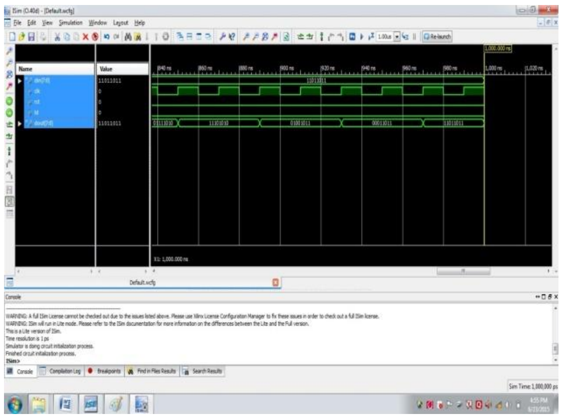


Figure 1: Simulation result of 8 bit LUT-OPT RNG

Figure 2: simulation result of LUT-FIFO RNG

Figure 3: Simulation result of 8 bit LUT-SR RNG

10. Synthesis Results

A. LUT-OPT RNG

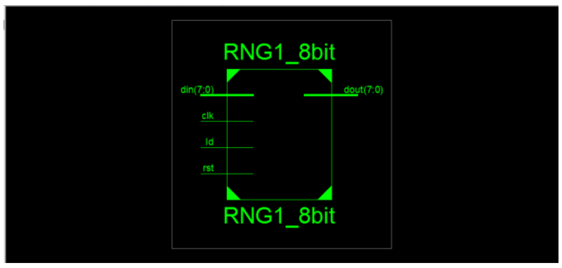


Figure 1: 8- Bit LUT-OPT RNG Block

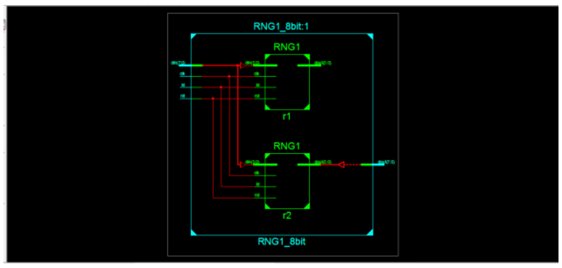


Figure 2: RTL Schematic of 8- Bit LUT-OPT RNG

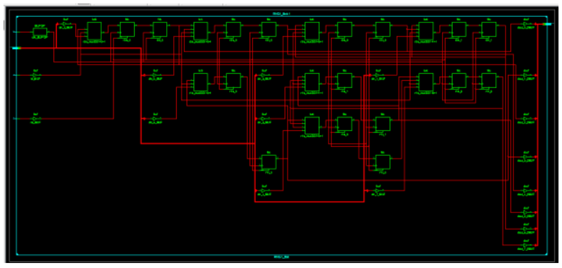
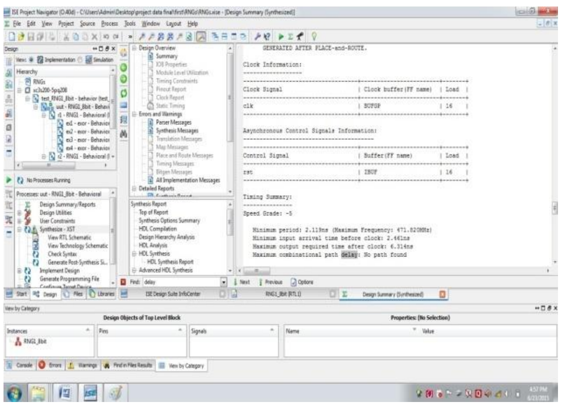


Figure 3: Technology Schematic of 8 Bit LUT-OPT RNG



```
GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 16    |
-----------------------------------+------------------------+-------+

Asynchronous Control Signals Information:
----------------------------------------
-----------------------------------+------------------------+-------+
Control Signal                     | Buffer(FF name)        | Load  |
-----------------------------------+------------------------+-------+
rst                                | IBUF                   | 16    |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 2.119ns (Maximum Frequency: 471.820MHz)
   Minimum input arrival time before clock: 2.461ns
   Maximum output required time after clock: 6.314ns
   Maximum combinational path delay: No path found
```

Figure 4: Delay of 8 bit LUT-OPT RNG

B. LUT-FIFO RNG

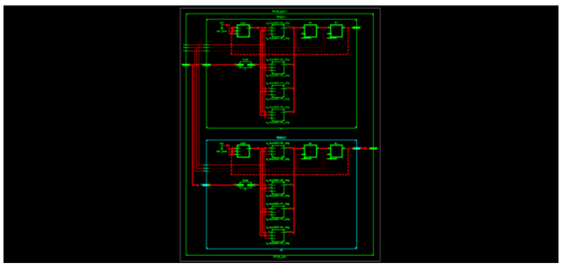
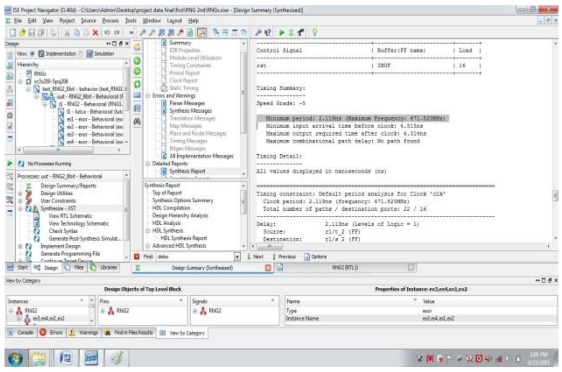


Figure 5: Technology Schematic Of 8 Bit LUT-FIFO RNG



```
-----------------------------------+------------------------+-------+
Control Signal                     | Buffer(FF name)        | Load  |
-----------------------------------+------------------------+-------+
rst                                | IBUF                   | 16    |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 2.119ns (Maximum Frequency: 471.820MHz)
   Minimum input arrival time before clock: 4.515ns
   Maximum output required time after clock: 6.314ns
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 2.119ns (frequency: 471.820MHz)
  Total number of paths / destination ports: 22 / 16
-------------------------------------------------------------------------
Delay:               2.119ns (Levels of Logic = 1)
  Source:            r1/t_2 (FF)
  Destination:       r1/s_2 (FF)
```

Properties of Instance: ex3,ex4,ex1,ex2

Name	Value
Type	exor
Instance Name	ex3,ex4,ex1,ex2

Figure 6: Delay of 8- bit LUT-FIFO RNG

C. LUT-SR RNG

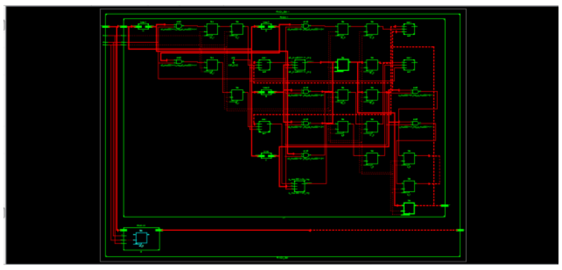
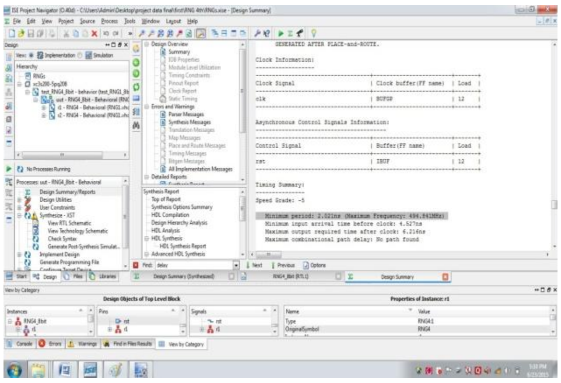


Figure 7: Technology schematic of 8 bit LUT-SR RNG



```
GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 12    |
-----------------------------------+------------------------+-------+

Asynchronous Control Signals Information:
----------------------------------------
-----------------------------------+------------------------+-------+
Control Signal                     | Buffer(FF name)        | Load  |
-----------------------------------+------------------------+-------+
rst                                | IBUF                   | 12    |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 2.021ns (Maximum Frequency: 494.841MHz)
   Minimum input arrival time before clock: 4.527ns
   Maximum output required time after clock: 6.216ns
   Maximum combinational path delay: No path found
```

Figure 8: Delay of 8 bit LUT-SR RNG

11. Conclusion

A family of FPGA-optimized uniform random number generator, called a LUT-SR RNG. LUT-SR RNGs takes advantage of the ability to configure LUTs as independent shift-registers, allowing high-quality long period generators to be implemented using only a small amount of logic. In addition the period and quality scale with the number of output bits, unlike generators adapted from software.

A key advantage of the LUT-SR generators over previous FPGA-optimized uniform random number generators is that they can be reconstructed using a simple algorithm ,new RNGs without needing to find generator instances themselves.
This paper uses a hardware description language called VHDL to design LUT-OPT RNG, LUT-FIFO RNG and LUT –SR RNG. In this dissertation, strategies & implementation of different RNGs is described. The LUT-OPT RNG, LUT-FIFO RNG, LUT -SR RNGs coded in VHDL and VHDL code executing on the Xilinx ISE 13.1i VHDL tools.

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] 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.

Author Profile

Rita S. Rawate She born in BHANDARA,(M.S) on May 7th 1986. She completed her B.E (ECE). From MIET, Gondiya (M.S). She is pursuing her M.TECH in VLSI from Priyadarshini college of engineering and technology, Nagpur, Maharashtra, India