Design Techniques of FPGA Based Random Number Generator

Pong P. Chu[1] and Robert E. Jones[2]

[1] Department of Electrical and Computer Engineering, Cleveland State University, Cleveland, Ohio 44115

[2] NASA Glen Research Center, Cleveland, Ohio 44112

ABSTRACT

Random numbers are required in a wide variety of applications. As digital systems become faster and denser, it is feasible, and frequently necessary, to implement random number generators directly in hardware. In this paper, we describe techniques suitable for hardware implementation, including one-bit true random number generator, one-bit LFSR (Linear Feedback Shift Register) generator, multiple-bit LFSR generator, multiple-bit leap-forward LFSR generator, and multiple-bit lagged Fibonnaci generator. We also discuss the benefit of each technique and its circuit complexity, particularly for implementation utilizing FPGA (Field Programmable Gate Array) devices.

I. INTRODUCTION

Random numbers are required in a wide variety of applications, including data encryption, circuit testing, system simulation and Monte Carlo method. In the past, the random number generation was mostly done by software. However, as digital systems become faster and denser, it is feasible, and frequently necessary, to implement the generator directly in hardware. Although the software-based methods are well understood [3] [4] [5] [7], they frequently require complex arithmetic operations and thus are not feasible to be constructed in hardware.

Ideally, the generated random numbers should be uncorrelated and satisfy any statistical test for randomness. A generator can be either "truly random" or "pseudo random". The former exhibits true randomness and the value of next number is unpredictable. The later only appears to be random. The sequence is actually based on specific mathematical algorithms and thus the pattern is repetitive and predictable. However, if the cycle period is very large, the sequence appears to be non-repetitive and random. Although it is possible to implement a true random number generator in hardware, it is slow and relatively expensive. The main focus of this paper is on pseudo random number generators. For simplicity, we will drop the word pseudo in our discussion.

This paper provides an overview on the methods that are suitable for hardware implementation, outlines the basic design and discusses their relative benefits. In remaining paper, section II discusses the implementation of a true, thermal-noised based random number generator; sections III and IV describe single-bit and multiple-bit generator based on LFSR; section V shows a special leap-forward LFSR implementation for multiple-bit generator; section VI discusses a more complicated lagged Fibonnaci approach; and last section summarizes the paper.

II. TRUE RANDOM NUMBER GENERATOR

True randomness can be derived from certain physical phenomena, such as the time between tics from a Geiger counter exposed to radioactive materials. In electronic circuit, thermal noise is frequently used as the source of randomness because of its well-qualified spectral and statistical properties. A representative implementation [8] is shown in Figure 1. In this circuit, the source of the noise is the thermal noise of a precision resistor, which is represented as V_{noise}. It is amplified by a low-noise amplifier and then passed to a high-speed comparator. The threshold of the comparator (V_{ref}) corresponds to the mean voltage of the input

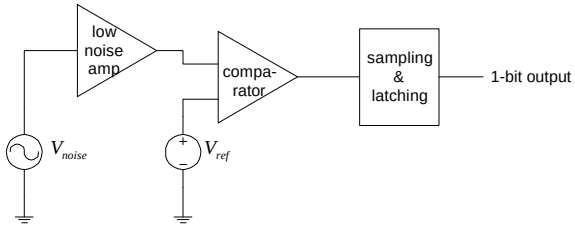

Figure 1. A True 1-bit Random Number Generator

noise signal. The output of the comparator is sampled and latched to a register. The latched signal is a one-bit binary signal that exhibits true randomness.

The true random number generator is fairly involved since it needs to preserve and amplify the thermal noise, and at the same time shield all external disturbances. It consists of mainly analog components and cannot be implemented by pure digital circuitry. The mixed-signal implementation significantly increases the system complexity. This implementation is also relatively slow and cannot match the high-speed digital circuit. One major application of the true random number is to generate the initial seed for pseudo random number generator.

III. SINGLE-BIT RANDOM NUMBER GENERATOR USING LFSR

A single bit random number generator produces a value of 0 or 1. The most efficient implementation is to use an LFSR (Linear Feedback Shift Register) [2]. It is based on the recurrence equation:

$$x_n = a_1 \bullet x_{n-1} \oplus a_2 \bullet x_{n-2} \oplus \cdots \oplus a_m \bullet x_{n-m}$$

Here, x_i is the i^{th} number generated, a_i is a pre-determined

constant that can be either 0 or 1, and $\bullet$ and $\oplus$ are AND operator XOR (e) operator respectively. equation implies that a new number (x_n) utilizing m ($x_{n-1}, x_{n-2}, \cdots, x_{n-m}$) through a quence of AND XOR operations.

generated pattern will repeat itself after a certain . know as the *period* of the nerator. In an LFSR, the achievable period is determined by m which is $2^m - 1$. In order to achieve the maximum period, a special set of a_i . In these sets, most a_is are 0, and only two to four of them are 1. Thus, the actual recurrence equation is fairly simple. Despite of its simplicity, the recurrence equations are different for different values of *m*. Many texts, such as [1] [2], have tables that list the recurrence equations exhaustively. Table in Figure 2 lists the recurrence equations for *m* with values from 2 to 8.

m	Recurrence equation
	$_{n-1} \oplus x_{n-2}$
3	$x_{-1} \oplus _{n}$
4	$x_{n-1} \oplus x_{n-4}$
5	$x_{n-2} \oplus x_{n-5}$
6	$x_{n-1} \oplus x_{n-6}$
7	$x_{n-1} \oplus x_{n-7}$
8	$x_{n-2} \oplus x_{n-3} \oplus x_{n-4} \oplus x_{n-8}$

Figure 2. Sample Recurrence Equations

For example, when *m* is 4, the equation becomes:

$$x_n = x_{n-1} \oplus x_{n-4}$$

Assume the initial seed (i.e., $x_1\ x_2, x_3, x_4$) is 1000. The random number sequence can be obtained by the equation:

100011110101100 100011110101100 ……

Note that the pattern repeats itself after 15 numbers.

The next step is to implement the recurrence equation in hardware. The new value, $_n$, depends on *m* previous values, $x_{n-}, x_{2}, \cdots _{n-m}$, and thus *m* 1-bit registers are required to store these values. After a new value is generated, the oldest stored value is no longer needed for future generation and can be discarded to make place for the new value. This can be done by an *m*-slot shift register, which shifts out the oldest value and shifts in a new value in every clock cycle. In addition to the register, few XOR gates are also required to perform exclusive-OR operation. Let us use the previous example of *m*=4 again. We need 4 1-bit registers to store the required values. Let q_3, q_2, q_1 and q_0 be the outputs of registers, and q_{3_next}, q_{2_next}, q_{1_next} and q_{0_next} be their next values. The Boolean equations for these registers can be written as:

$$q_{0\ next} = q_1$$
$$_{1\ next} = q_2$$
$$q_2 \quad _3$$
$$q_{3\ next} = q_3 \oplus _0$$

implementation is shown in Figure 3 (the ynchronous connections are omitted for clarity).

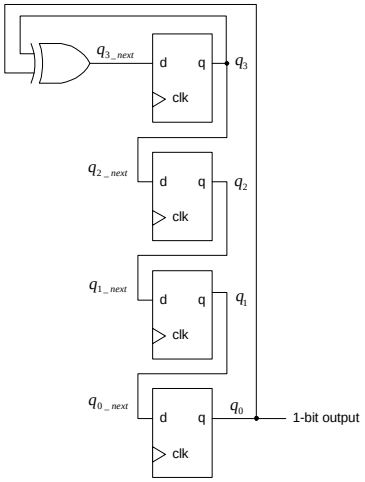

Figure 3. 4-bit LFSR

An LFSR random number generator is a very efficient n -bit shift register and 1 to

its operation is extremely fast. In some FPGA devices, anized as shift registers and further reduce the

LFSR implementation has shown many nice statistical

number. Furthermore, since the period grows exponentially with the size of the register, large non-repetitive sequences

running at 1 GHz, the period is more than 500 years.

n initial seed is needed The seed corresponds to the initial condition of the registers. It can be any state except for all 0 combination (i.e., 00 , which causes to be stuck at zero fo One way to fix the problem is to add a special circuit that s …01 conditions and value accordingly. The circuit known as e Bruijn counter, consists an *m* . It $2^m - 1$ to 2^m. equations for a 4-bit de Bruijn counter are

$$q_{0\_next} = q_1$$
$$q_{1\_next} = q_2$$
$$q_{2\_next} = q_3$$
$$q_{3\_next} = (q_3 \oplus q_0) \oplus (q_3' \bullet q_2' \bullet q_1')$$

The random sequence now has an extra 0 and is shown below (the new 0 is underlined):

1000<u>0</u>11110101100 1000<u>0</u>11110101100 ……

Its block diagram is shown in Figure 4. Note that the de Bruijn counter destroys the "linearity" of the system and the recurrence equation can no longer be applied for all x_i .

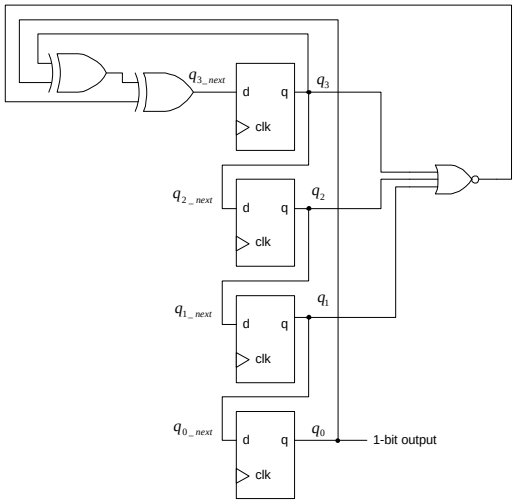

Figure 4. 4-bit de Bruijn Counter

IV. Multiple-Bit Random Number Generator Using LFSR

Some applications require more accuracy and need more than a single-bit random number. Since the numbers produced by a single-bit LFSR random number generator are uncorrelated, one way to obtain a multiple-bit random number is to accumulate several single-bit numbers. There are several techniques to achieve this and they are discussed in below.

A. Single-LFSR Method

The Single-LFSR method requires only one LFSR. It utilizes the values stored in shift register to form a multiple-bit number. For example, if a 4-bit random number is needed, we can use the output of register of the 4-bit LFSR shown in Figure 3 (i.e., q_0, q_1, q_2 and q_3). The generated 4-bit sequence is shown in the left column of Figure 5. Note that all possible 4-bit combinations, except for 0000, appear in the sequence.

Although this implementation is simple, the generated random numbers are highly correlated and fail many statistical tests. This should not come to a surprise since a new random number keeps most bits from the old number and contains only 1-bit new information.

To overcome the correlation problem, it is necessary to replace all bits in the random number rather than just one bit. There are several ways to do it and they are discussed in the subsequent subsections.

Single LFSR $q_0\ q_1\ q_2\ q_3$		Leap-forward LFSR $q_0\ q_1\ q_2\ q_3$	
1000 (8)	0101 (5)	1000 (8)	0011 (3)
0001 (1)	1011 (11)	1111 (15)	1101 (13)
0011 (3)	0110 (6)	0101 (5)	0110 (6)
0111 (7)	1100 (12)	1001 (9)	0100 (4)
1111 (15)	1001 (9)	0001 (1)	0111 (7)
1110 (14)	0010 (2)	1110 (14)	1010 (10)
1101 (13)	0100 (4)	1011 (11)	1100 (12)
1010 (10)		0010 (2)	

Figure 5. 4-bit Random Sequence of from Single-LFSR and Leap-forward LFSR Methods

B. Single-LFSR with a Counter Method

This implementation consists of a single LFSR and a counter. This method replaces new bits one at a time. A *k*-bit random number requires *k* shift operations of an LFSR to form a new number. As long as *k* is relatively prime to the period of the LFSR, the *k*-bit number will cycle through all possible states. In order to keep track the number of shift operation, an additional modulo-*k* counting circuit is required. The counter will generate a special enable signal that is asserted once every *k* clock cycles. The LFSR will operate as usual. However, its output is interpreted as valid only when the enable signal of the counter is asserted. The disadvantage of this approach, of course, is the operation speed. Clearly, the random number generator is slower by a factor *k*.

C. Parallel-LFSR method

Parallel-LFSR method is a straightforward extension of the previous single-bit random number generator method. It utilizes *k* copies of identical one-bit generator hardware to generate *k* bits concurrently. The major advantage of this method is the operation speed, which is identical to that of a single-bit generator. However, this method also requires a large amount of hardware. First, *k* copies of LFSRs are required. Second, each LFSR must have a different seed in order to avoid correlation. Since logic cells of most FPGA chips does not contains both reset and preset inputs, additional initialization circuit is required. Recall that the

original LFSR needs only few XOR gates. The initialization circuit may significantly increase the circuit complexity.

Another concern of parallel LFSR method is the poor utilization of FPGA's resource. FPGA devices are made of a collection of generic logic cells, which normally contain a programmable combinational circuit plus one or two registers. Since LFSRs require little combinational circuitry, logic cells are not fully utilized. A better approach is to use the embedded SRAM, which will be discussed in section VI.

V. Multiple-Bit Leap-forward LFSR

Leap-forward LFSR method utilizes only one LFSR and shifts out several bits. However, unlike the counter method of section IV, all shifts are performed in one clock cycle; i.e., multiple steps are done in the recurrence equation. This method is based on the observation that an LFSR is a linear system and the register state can be written in vector format:

$$\mathbf{q}(i+1) = \mathbf{A} \bullet \mathbf{q}(i)$$

In this equation, $\mathbf{q}(i+1)$ and $\mathbf{q}(i)$ are the content of shift register at $(i+1)^{\text{th}}$ and i^{th} steps, and $\mathbf{A}$ is the transition matrix.

After the LFSR advances k steps, the equation becomes

$$\begin{aligned}\mathbf{q}(i+k) &= \mathbf{A} \bullet \mathbf{q}(i+k-1) \\ &= \mathbf{A} \bullet (\mathbf{A} \bullet \mathbf{q}(i+k-1)) \\ &= \mathbf{A}^2 \bullet \mathbf{q}(i+k-2) \\ &= \mathbf{A}^3 \bullet \mathbf{q}(i+k-3) \\ &= \cdots\cdots \\ &= \mathbf{A}^k \bullet \mathbf{q}(i)\end{aligned}$$

We can calculate $\mathbf{A}^k$ and determine the XOR structure accordingly. The new circuit can leap k steps in one clock cycle. It still consists the identical shift register although the feedback combinational circuitry becomes more complex. The actual implementation can be best demonstrated by an example.

Let us use the 4-bit LFSR in Figure 3 as an example. It can be written as

$$\mathbf{q}(i+1) = \mathbf{A} \bullet \mathbf{q}(i)$$

where $\mathbf{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$ and $\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$

Assume that we need a four-bit random number generator and thus we have to advance 4 steps at a time. The new transition matrix, $\mathbf{A}^4$, becomes:

$$\mathbf{A}^4 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

For the purpose of circuit implementation, the equation can be written as

$$\begin{bmatrix} q_{0\_next} \\ q_{1\_next} \\ q_{2\_next} \\ q_{3\_next} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

After performing matrix multiplication, we can derive the feedback equation for each signal:

$$\begin{aligned} q_{0\_next} &= q_0 \oplus q_3 \\ q_{1\_next} &= q_0 \oplus q_1 \oplus q_3 \\ q_{2\_next} &= q_0 \oplus q_1 \oplus q_2 \oplus q_3 \\ q_{3\_next} &= q_0 \oplus q_1 \oplus q_2 \end{aligned}$$

The corresponding block diagram is shown in Figure 6 and the four-bit output random sequence is shown in the right column of Figure 5.

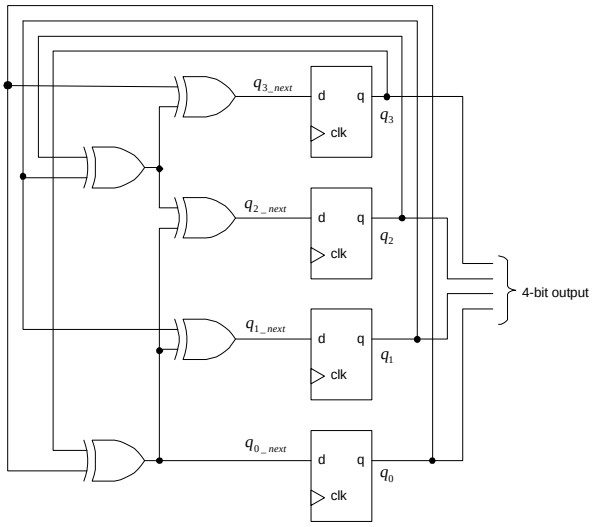

Figure 6. 4-bit Leap-forward LFSR

Leap-forward LFSR method achieves its goal by utilizing extra combinational circuit instead of duplicated LFSRs. For small k, the combinational circuit is not very complex. It is ideal for FPGA devices since it balances register and combinational circuitry and fully utilizes the resource of logic cells. However, for very large k, the XOR structure grows very large and becomes the dominant factor.

VI. Multiple-Bit Lagged Fibonnaci Generator

Lagged Fibonnaci method processes an n-bit word directly. It is governed by a recurrence equation:

$$\mathbf{x}_k = \mathbf{x}_{k-l} \otimes \mathbf{x}_{k-m} \quad \text{where } l > m > 0$$

In this equation, $\mathbf{x}_k$, $\mathbf{x}_{k-l}$ and $\mathbf{x}_{k-m}$ are all n-bit words, and represent the values at k^{th}, $(k\text{-}l)^{\text{th}}$ and $(k\text{-}m)^{\text{th}}$ steps respectively. The $\otimes$ symbol denotes an operator, which can

be bit-wise XOR, addition or multiplication. In order to obtain good randomness properties and the largest possible period, *l*, *m* and $\otimes$ have to be carefully chosen. Tables of recurrence equations are available in the literature [4] [7]. In general, complex operator performs better than simple operator does since it provides more mutation among the individual bits. Addition is the most commonly used operator and the corresponding implementation has been studied in more detail. Lagged Fibonnaci method is considered to be the best pseudo random generator [5] [6] [7]. Note that the parallel-LFSR method can be considered as a special case of the lagged Fibonnaci method.

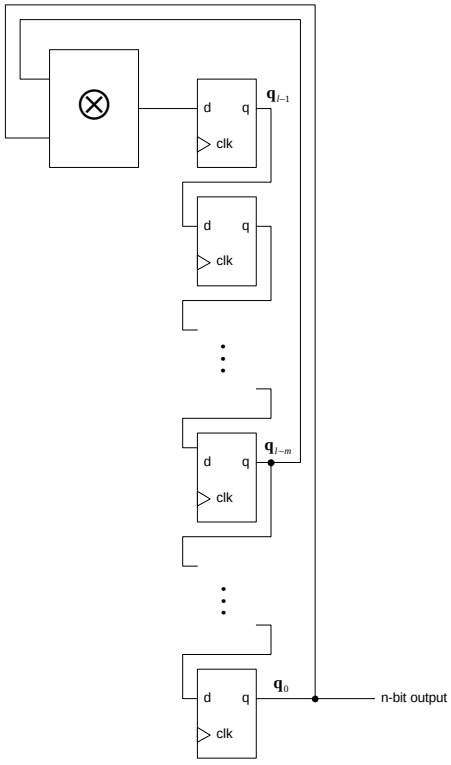

Figure 7. Direct Implementation of Lagged Fibonnaci Generator

To implement this method, we need *l* *n*-bit registers to memorize *l* past values and a circuit to perform $\otimes$ operation. The block diagram is shown in Figure 7. This approach requires a large number of registers, which causes poor logic cell utilization in FPGA, as we discussed in previous section. A better alternative is to utilize the embedded SRAM. Note that the access pattern is very regular and thus the register file can be replaced by a FIFO (first-in-first-out) buffer, which can be implemented by SRAM. The conceptual block diagram is shown in Figure 8. The SRAM is organized as a circular queue, with a pointer *head* pointing to the oldest element of the buffer (i.e., $\mathbf{x}_{k-l}$). The other two locations (i.e., $\mathbf{x}_{k-m}$ and) are constants relative to *head* and can be easily derived. In this implementation, $\mathbf{x}_{k-l}$ and $\mathbf{x}_{k-m}$ are read from SRAM and passed to $\otimes$ circuit, and the result is

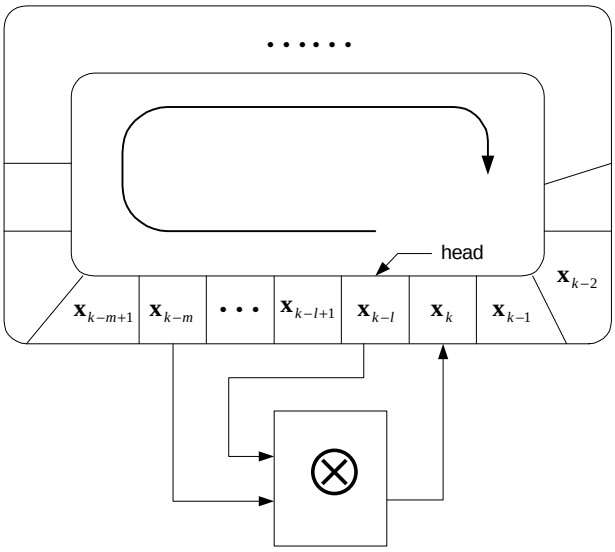

Figure 8. Conceptual SRAM Implementation of Lagged Fibonnaci Generator

written to $\mathbf{x}_k$. In next clock cycle, *head* moves ahead one step (i.e., the value of *head* is increased by 1) and the process is repeated. Note that the old value of is $\mathbf{x}_{k-l}$ discarded and overwritten by the new value of $\mathbf{x}_k$. The rough block diagram is shown in Figure 9. Since the structure of the embedded SRAM varies significantly for FPGA chips, the actual implementation is device dependent. Basically, the operation needs two memory reads and one memory write. It requires three cycles for a single port SRAM and two cycles for a dual-port SRAM. In many FPGA chips, SRAM is organized as a collection of small SRAM modules distributed over the logic blocks. It is possible to perform two reads at different SRAM modules at the same time and further reduce the number of cycles.

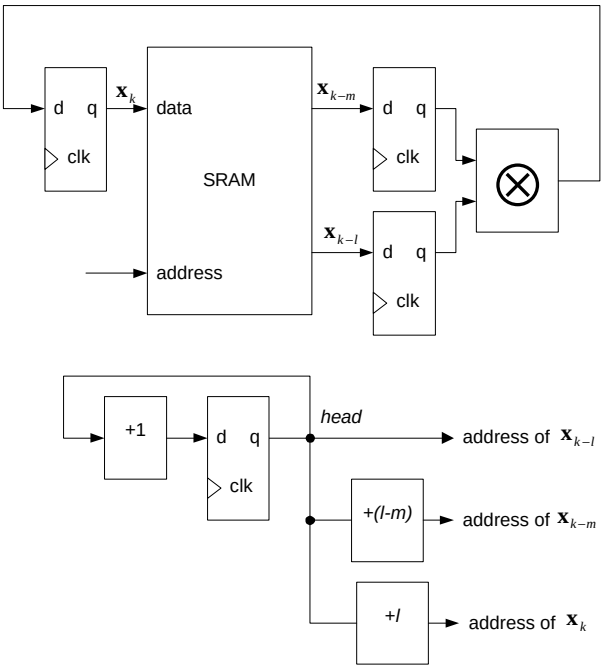

Figure 9. Block Diagram of SRAM-Based Lagged Fibonnaci Generator

In FPGA implementation, addition is preferred for the operator $\otimes$ since it provides proper permutation between individual bits and addition circuit can be efficiently synthesized in most devices. One other concern is the initialization of lagged Fibonnaci method, which requires *l n*-bit wide seeds. Frequently, it is done by using a small one-bit LFSR generator to fill the memory during the system initialization.

VII. Summary

We have examined several deign techniques for hardware random number generators and their feasibility for FPGA devices. For a single-bit random number generator, LFSR is the most effective method. When multiple bits are required, LFSR can be extended by utilizing extra time (as in counter method) or extra circuitry (as in parallel LSFR method and leap-forward LSFR method). For a small number of bits, leap-forward LFSR method is ideal since it balances the combinational circuitry and register and thus fully utilizes the FPGA's resource. For a large number of bits, lagged Fibonnaci method is preferred. Its circuit is more complex and requires embedded SRAM for efficient implementation.

Acknowledgements

Part of Dr. Pong Chu's work was supported by NASA Grant NAG3-2040.

References

[1]. P. Alfke, "Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators," *Xilinx Application Note*, 1995.

[2]. P. H. Bardell, W. H. McAnney and J. Savir, *Build-in Test for VLSI: Pseudo-random Techniques*, John Wiley and Sons, 1987.

[3]. F. James, "A Review of Pseudo-random Number Generators," *Computer Physics Communications* 60, 1990.

[4]. D.E. Knuth, *The Art of Computer Programming Vol. 2: Seminumerical Methods* (2nd edition), Addison-Wesley, Reading, Mass., 1981.

[5]. P. L'Ecuyer, "Random Numbers for Simulation," *Comm. ACM* 33:10, 1990.

[6]. P. L'Ecuyer, "Efficient and Portable Combined Random Number Generators," *Comm. ACM* 31:6, 1988.

[7]. G.A. Marsaglia, "A Current View of Random Number Generators," *Computational Science and Statistics: The Interface*, ed. L. Balliard, Elsevier, Amsterdam, 1985.

[8]. Quantum World Corporation, *QNG Model J20KP True Random Number Generator Users Manual*, 1998