Uniform Random Number Generator using Leap-Ahead LFSR Architecture

GU Xiao-chen, ZHANG Min-xuan
School of Computer
National University of Defense Technology
Changsha, China
specialsjtu@163.com

***Abstract*—Uniform Random Number Generator (URNG) is a key element in most applications which run on FPGA based hardware accelerators. As multi-bits is required and a normal LFSR could only generate one bit per cycle, more than one LFSR is needed in a URNG. In this paper, we introduce a new kind of URNG using Leap-Ahead LFSR Architecture which could generate an m-bits random number per cycle using only one LFSR. We analyze its architecture, present the expression of the period and point out how to choose the taps of the LFSR. Finally, a 18-bits URNG is implemented on Xilinx Vertex Ⅳ FPGA.. By comparison, the Leap-Ahead LFSR Architecture URNG consumes less than 40 slices which is only 10% of what the Multi-LFSRs architecture consumes and acquires very good Area Time performance and Throughput performance that are 2.18×10-9 slices×sec per bit and 17.87×109 bits per sec.**

Keywords-FPGA; Uniform Random Number Generator; LFSR

I. INTRODUCTION

With the improving performance of FPGAs, they have been introduced as hardware accelerators frequently. Most of the applications implemented on these FPGAs are computation intensive, such as Monte-Carlo simulations[4][5][6]. In these applications, random number generator (RNG) is a very important and frequently used element.

Among all kinds of RNGs, Uniform Random Number Generator (URNG) is the most important one, because all the other RNGs which have different distributions could be transformed from URNG[7]. There are many methods to implement a URNG, such as "MT19937", and LFSR based architecture is one of the most popular because it could be easily described with HDL language and prototyped in FPGAs.

A normal LFSR could only generate one random bit per cycle. As multi-bits is required to form a random number in most applications, Multi-LFSRs architecture is used to implement a URNG. This means 32 different LFSRs are needed in a 32-bit output URNG. But Leap-Ahead architecture could avoid this and generate one multi-bits random number per cycle using only one LFSR.

In this paper, we introduce the architectures of Leap-Ahead LFSRs of both Galois type and Fibonacci type. We primarily analyze the characteristics of the transform matrix and present the period formula of the generated random numbers. Finally, we implement the Leap-Ahead LFSR based URNGs on Xilinx Vertex Ⅳ FPGA, and analyze the results in detail with compare to other reported ones.

II. LEAP-AHEAD ARCHITECTURE

A. The architecture

There are two types of LFSRs: Galois type and Fibonacci type as illustrated in Fig. 1. DFFi is a register, c_1-c_{n-1} are the taps, x_i is the output of the ith DFF.

Both of the two types of LFSRs could be described by the following formula:

$$X(t+1) = AX(t) \qquad (1)$$

X(t) is the output of all the DFFs at current time; X(t+1) is the output of all the DFFs at the next clock cycle; A is the transform matrix. Here, only x_n is the "active output" per cycle. If we use m bits ($x_{n-(m-1)}$ to x_n) as the output of the URNG, the random numbers generated would have very close correlation and this is unacceptable.

978-0-7695-3906-5/09 $26.00 © 2009 IEEE
DOI 10.1109/ICCCS.2009.11

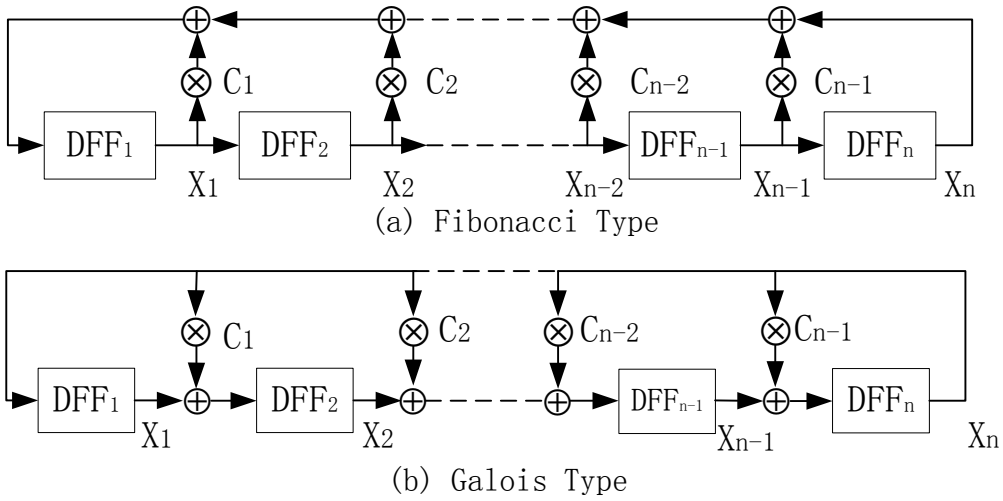

Figure 1 Structure of Galois type and Fibonacci type LFSRs

If we use formula (1) m times, we could acquire the m-cycle-late outputs of all the DFFs as follows:

$$X(t+m)=AX(t+m-1)=A(AX(t+m-2))=...=A^mX(t) \tag{2}$$

This could also be represented as

$$X'(t+1)=A^mX'(t) \tag{3}$$

If we design the feedback circuit, like the structure in Figure 1 but using the new transform matrix A^m, we could acquire the m-cycle-late outputs of all the DFFs in Figure 1 within only one cycle in the new architecture. Because the outputs of all the DFFs leap m cycle ahead in the new architecture, we just call it Leap-Ahead LFSR architecture. Here, X'(t+1) in formula (3) represents the new outputs.

If $DFF_{n-(m-1)}$ to DFF_n in Figure 1 operate as shift registers, then, in Leap-Ahead architecture, the m-bit outputs $x'_{n-(m-1)}$ to x'_n per cycle just equal to the m-cycle outputs of x_n in Figure 1. Now, we could use the m-bit outputs $x'_{n-(m-1)}$ to x'_n as an m-bit random number, because they no longer have close correlation.

This could be easily satisfied in Fibonacci Type LFSR, because all the DFFs in Figure 1 (a) operate as shift registers. But in Galois Type LFSR in Figure 1 (b), we have to carefully choose the taps and make sure that $DFF_{n-(m-1)}$ to DFF_n also operate as shift registers.

B. The period of the generated random numbers

The generated random numbers are pseudo random numbers and they have a period. We could choose the taps of the original LFSR carefully to make sure that the one-bit stream generated has a maximum period of 2^n-1. Then, we could figure out that the period of the random numbers generated by the corresponding Leap-Ahead LFSR have the following relationship with n and m:

$$T=\frac{[2^n-1,m]}{m} \tag{4}$$

Here, $[2^n-1, m]$ is the lease common multiple of 2^n-1 and m; n is the number of the stages of the LFSR; m is the number of the output bits of the URNG; T is the period of the generated random numbers.

Obviously, T could get its maximum when 2^n-1 and m could not divide by each other. But at the same time, the generated random numbers, again, would have close correlation with each other, because they would have the same content as the output of $x_{n-(m-1)}$-x_n from the original LFSR. So, only when 2^n-1 could divide by m, the generated random numbers would acquire the best quality. But the cost would be that T would get its minimum. So, there is a tradeoff between the period and the quality of the generated random numbers.

C. The transform matrix

The transform matrix A in formula (1) is a special matrix. It could be expressed as follows:

$$A_{Galois}=\begin{pmatrix} 0_{1\times(n-1)} & C_{n\times 1} \\ I_{(n-1)\times(n-1)} & \end{pmatrix}_{n\times n}$$

$$A_{Fibonacci}=\begin{pmatrix} C_{1\times n} \\ I_{(n-1)\times(n-1)} \quad 0_{(n-1)\times 1} \end{pmatrix} \tag{5}$$

Here, $C_{n\times 1}$ (or $C_{1\times n}$) is the vector of taps; $I_{(n-1)\times(n-1)}$ is an identity matrix; $0_{1\times(n-1)}$ is a zero matrix. According to this expression, we could acquire A^m as follows:

$$A_{Galois}^{\ m}=\begin{pmatrix} 0_{m\times(n-m)} & C_{1\times 1} & A\times C_{1\times 1} & \cdots & A^{m-2}\times C_{1\times 1} & A^{m-1}\times C_{1\times 1} \\ I_{(n-m)\times(n-m)} & & & & & \end{pmatrix}_{n\times n}$$

$$A_{Fibonacci}^{\ m}=\begin{pmatrix} C_{1\times n}\times A^{m-1} \\ C_{1\times n}\times A^{m-2} \\ \cdots \\ C_{1\times n}\times A \\ C_{1\times n} \\ I_{(n-m)\times(n-m)} \quad 0_{(n-m)\times m} \end{pmatrix}_{n\times n} \tag{6}$$

In Galois type architecture, when the taps have the following relationship:

$$c_{n-1}=c_{n-2}=\ldots=c_{n-(m-1)}=0 \tag{7}$$

We could acquire:

$$A^{m-1}\times C_{1\times 1}=\left(\underbrace{0\ \ 0\ \ \ldots\ \ 0}_{(m-1)\text{个}}\ \ 1\ \ c_1\ \ c_2\ \ \ldots\ \ c_{n-(m+1)}\ \ c_{n-m}\right)^T_{1\times n} \tag{8}$$

Also, in Fibonacci type architecture, when the taps have the following relationship:

$$c_1=c_2=\ldots=c_{m-1}=0 \tag{9}$$

We could acquire:

$$C_{1\times n}\times A^{m-1}=(c_m,c_{m+1},c_{m+2},\ldots,c_{n-1},1,\underbrace{0,0,\ldots,0}_{(m-1)\text{个}})_{1\times n} \tag{10}$$

Obviously, when taps could satisfy the relationship in (7) and (9), we would acquire $(A^{m-1}\times C_{n\times 1})^T$ and $C_{1\times n}\times A^{m-1}$ by only simply right-shifting $C_{n\times 1}^T$ and left-shifting $C_{1\times n}$. This just shows us an easy way to figure out the new transform matrix A^m.

III. IMPLEMENTATION AND RESULTS

According to the above analysis, we implement both Galois type and Fibonacci type Leap-Ahead architecture based 18-bits output URNGs on a Xilinx Vertex Ⅳ FPGA. Here, the required period of the generated random numbers is 2^{18}. So, according to formula (4), n should be 22. But we could not acquire proper taps that would satisfy formula (7) and (9), when n equals to 22. So, we have to increase n from 22 to 23。

For comparison, we also implement the Multi-LFSRs architecture URNGs which have the same output bit width and period. We use ISE 10.1 to synthesize the Verilog. The results is listed in Table Ⅰ.

TABLE Ⅰ Synthesis results of URNGs using different architecture

Architecture	Period	Bit-width	No. of LFSRs used	Stages of each LFSR	Slices	Frequency
Leap-Ahead (Galois)	2^{18}	18	1	23	39	993 MHz
Leap-Ahead (Fibonacci)	2^{18}	18	1	23	37	993 MHz
Multi-LFSR (Galois)	2^{18}	18	18	18	393	1146 MHz
Multi-LFSR (Fibonacci)	2^{18}	18	18	18	383	1010 MHz

As listed in TABLE Ⅰ, the Leap-Ahead architecture consumes less than 10% of slices which the Multi-LFSR architecture consumes. One of the reasons for this is that the Leap-Ahead architecture has only 1 LFSR in the URNG hardware, while the Multi-LFSRs architecture has 18. The other reason is that every register in the URNG has to be initialed separately when the circuit is restarted, and the logic for this is complicated. As the Multi-LFSR architecture has 18×18 registers, while the Leap-Ahead architecture has only 23 registers, it needs more slices for the initializing function.

According to formula (4), we could acquire the following relationship:

$$n_{Leap-Ahead} - \log_2^m = n_{Multi-LFSR} \qquad (11)$$

Here, $n_{Leap-Ahead}$ and $n_{Multi-LFSR}$ are the numbers of stages of a single LFSR in Leap-Ahead architecture or Multi-LFSR architecture. This means, for acquiring the same period, the Leap-Ahead architecture needs $\log_2^m$ more stages in a single LFSR than which the Multi-LFSR architecture needs. But, because there are m LFSRs in one Multi-LFSRs architecture URNG, the overall number of stages used here is $m \times n_{Multi-LFSR}$. This number is much larger than $n_{Leap-Ahead}$. As listed in TABLE Ⅰ, it is 18×18 to 23.

From the results in TABLE Ⅰ, we could also acquire the conclusion that the Leap-Ahead architecture works slower than the Multi-LFSR architecture. This is because the feedback logic is much more complicated in the Leap-Ahead architecture. And the Fan-Out of each register is larger, too. This drawback is much obvious in Galois type architecture, because not only the Fan-Out increases but also the logic stages of the feedback circuit increases from 1 to 4. So, the working frequency decreases from 1146 MHz to 993 MHz.

In Leap-Ahead architecture, Galois type no longer has advantages in working speed, because it has almost the same complicated feedback logic as Fibonacci type does. As listed In TABLE Ⅰ, they have the same working frequency as 993 MHz. Galois type architecture consumes a bit more slices than Fibonacci type architecture (39 slices to 37 slices). This is only because its structure is more suitable for the optimization methods of the ISE tools.

Performance comparison results of different URNGs are listed in TABLE Ⅱ. As having a simple structure, the Leap-Ahead architecture acquires much higher working frequency, while consumes much less slices. So, in TABLE Ⅱ, the Area Time performance of Leap-Ahead architecture is 2.18 slices×sec per bit, much better than the other ones. Though our implementation is a 18-bit output URNG, it still acquires very good Throughput performance of 17.87×109 bits per sec because of the high working frequency. The Multi-LFSR architecture has a little better Throughput performance, but it has the worst Area Time performance as a cost. A more intuitional comparison results are denoted in Fig. 2.

TABLE Ⅱ Performance comparison of different URNGs

	Area Time slices×sec per bit $\times 10^{-9}$ (smaller is better)	Throughput Bits per sec $\times 10^{9}$ (larger is better)
Leap-Ahead (Galois) [this work]	2.18	17.87
Multi-LFSR(Galois) [this work]	19.05	20.63
MT19937[1]	7.04	11.136
MT19937[2]	9	6.13
MT19937 software[3]	-	3.10
TT800[8]	10.63	7.68

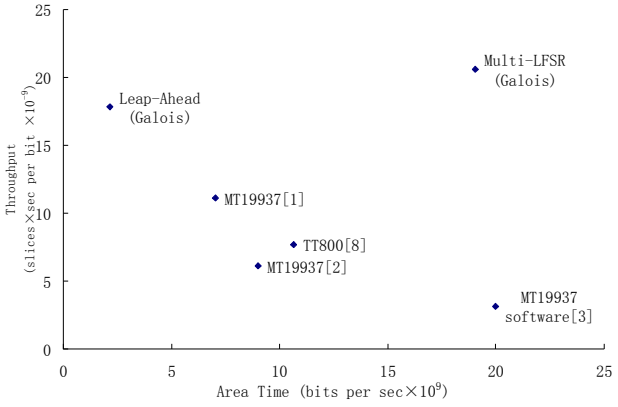

Figure 2 Comparison results of different URNGs

IV. CONCLUSION

We introduce a new kind of URNG using Leap-Ahead LFSR architecture; present the period expression and point out how to choose the taps for easy computation of transform matrix. By implementing the Leap-Ahead LFSR architecture and Multi-LFSR architecture of both Galois type and Fibonacci type on Xilinx Vertex Ⅳ FPGA, we acquire the conclusion that, with only very little lost in speed, Leap-Ahead LFSR architecture consumes only 10% slices of what the Multi-LFSR architecture does to generate the random numbers that have the same period. By comparison with other URNGs, Leap-Ahead LFSR architecture has very good Area Time performance and Throughput performance that are 2.18×10^{-9} slices×sec per bit and 17.87×10^{9} bits per sec.

ACKNOWLEDGMENT

This paper is supported by the National High Technology Research and Development Program ("863"Program) of China 2009AA01Z124.

REFERENCES

[1] I. L. Dalal, D. Stefan, “A hardware framework for the fast generation of multiple long-period random number streams”, Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays (FPGA ‘08), Monterey, California, USA, 2008. pages:245-254

[2] V. Sriram, D. Kearney, “High throughput multi-port MT19937 uniform random number generator”, Eighth International Conference on, Parallel and Distributed Computing, Applications and Technologies, 2007. PDCAT '07, Adelaide, SA, 2007, pages: 157-158

[3] M. Matsumoto, and Nishimura, T. (1998) Mersenne Twister, “A 623-dimensionally equidistributed uniform pseudo-random number generator”, ACM Transactions on Modeling and Computer Simulation, 8, 3-30.

[4] D U Lee, J D Villasenor, W Luk, P H W Leong. A hardware Gaussian noise generator using the Box-Muller method and its error analysis . Computers, IEEE Transactions on, 2006, Volume: 55, Issue: 6, pages:659-671.

[5] D U Lee, W Luk, J D Villasenor, G Zhang, P H W Leong. A hardware Gaussian noise generator using the Wallace method . Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2005, Volume 13, Issue 8, pages:911-920

[6] D U Lee, W Luk, J D Villasenor, P Y K. Cheung. A Gaussian noise generator for hardware-based simulations . Computers, IEEE Transactions on, 2004, Volume 53, Issue 12, pages: 1523-1534

[7] S. Banks, P. Beadling, A. Ferencz. FPGA Implementation of Pseudo Random Number Generators for Monte Carlo Methods in Quantitative Finance. Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on, Cancun, 2008, pages: 271 - 276

[8] V. Sriram, D. Kearney, “A high throughput area time efficient pseudo uniform random number generator based on the TT800 algorithm”, International Conference on, Field Programmable Logic and Applications, 2007. FPL 2007, Amsterdam, 2007, pages: 529 - 532