Proceedings of the Technical Sessions, 22 (2006)25-36
Institute of Physics – Sri Lanka

Hardware Implementation of Random Number Generators

W.A.S Wijesinghe[1], M.K Jayananda[2] and D.U.J Sonnadara[2]
[1]*Department of Electronics, Wayamba University of Sri Lanka*
[2]*Department of Physics, University of Colombo, Sri Lanka*

ABSTRACT

Random numbers are used in a wide variety of applications. True random number generators are slow and expensive for many applications while pseudo random number generators (RNG) suffice for most applications. Although a majority of random number generators have been implemented in software level, increasing demand exists for hardware implementation due to the advent of faster and high density Field Programmable Gate Arrays (FPGA). FPGAs make it possible to implement complex systems, such as numerical calculations, genetic programs, simulation algorithms etc., at hardware level. This paper discusses in detail the hardware implementation of several RNGs and their characteristics. Somewhat complex Cellular Automata based RNGs show slightly improved performance compared to the simplest Linear Feedback Shift Register RNG.

1. INTRODUCTION

Random numbers are widely used in various applications such as Monte Carlo simulations, cryptography, simulations of wireless communication systems, electronic circuit testing, genetic programming, data encryption, games etc. Usually, random numbers are generated using software algorithms. Although the sequence of numbers they produce seems random, they are not truly random. It is difficult to program a series of logical steps that produce numbers that do not follow some definite sequence. These random numbers are called Pseudo random numbers.

True random numbers can be generated from a physical process, such as measuring thermal noise or noise power level in a radio-frequency receiver, photoelectric effect or other quantum phenomena. These processes are, in theory, completely unpredictable [1]. True random number generators can be implemented by combining both analog and digital electronics. These generators generally tend to be expensive as well as slow.

High density and high speed programmable logic devices, such as Field Programmable Gate Arrays, have made it possible to implement complex systems completely embedded in hardware. For instance, some of the genetic algorithms and cryptography algorithms which had been originally implemented in software have now been implemented in hardware.

Most pseudo random number generating algorithms involve complex mathematical operations which are not suitable to be implemented in hardware. However, there are less complex methods that can be implemented in hardware. In this work we have studied several techniques that are used in generating random numbers and implement them using VHDL hardware description language.

First, the implementation of most simple and common random number generator, Linear Feedback Shift Register (LFSR) is discussed [2]. This generator tends to fail basic requirement of a RNG due to high correlation in the sequence [3]. The correlation problem that can be reduced by modifying the LFSR random number generator is discussed next. Then a cellular automata based random number generator is discussed. This is followed by a discussion which combines a 4-bit Cellular Automata generator and a 4-bit LFSR generator to construct 8-bit RNG. Finally, the utilization of resources, execution times, and a simple Monte Carlo test, which estimates the accuracy of the hardware generated random number sequences is presented.

2. HARDWARE IMPLEMENTATION OF RNG

The target hardware for this work was XC4005XL Field Programmable Gate Array (FPGA) on the XS40 prototype board running at 50 MHz. The basic building block of the FPGA is Configurable Logic Block (CLB). It contains two, four-input function generators (F and G) and one three-input function generator (H). These function generators are capable of implementing any arbitrarily defined Boolean function of four inputs and three inputs [4]. They are implemented as memory look up tables. The total number of user configurable logic gates available in this device is 5000.

2.1 8-bit LFSR RNG

The most common way to implement a random number generator is LFSR. Codes generated by the LFSR are actually pseudo random sequences because the sequence repeats itself after a certain number of cycles. It is known as the period of the generator. LFSR is based on the recurrence equation,

$$x_n = x_{n-1} \oplus x_{n-2} \oplus ... \oplus x_{n-m} \qquad \text{...............} (1)$$

The operator $\oplus$ is the exclusive OR (XOR) operator. The equation shows that n^{th} bit can be generated utilizing m previous values with XOR operators [2]. The value of m determines the period of the generator. The achievable maximum period is 2^m-1. For the 8-bit LFSR, the recurrence equation is,

$$x_n = x_{n-2} \oplus x_{n-3} \oplus x_{n-4} \oplus x_{n-8} \qquad \text{...............} (2)$$

Since new value x_n depends on previous m values, it is necessary to store previous m values to find the new value. This can be done with m single bit shift registers comprised with flip flops. According to the equation (2), XOR feedback tap positions are taken at 0th ,4th, 5th and 6th flip-flops. The maximum period of the generator is 2^8-1 (255). In each clock pulse, generated new bit is inserted to the shift register while the oldest bit shifts out. Output of the 8 flip-flops form the 8-bit random number.

A group of flip-flops connected in series are used with XOR gates to construct the LFSR random number generator (see figure 1).

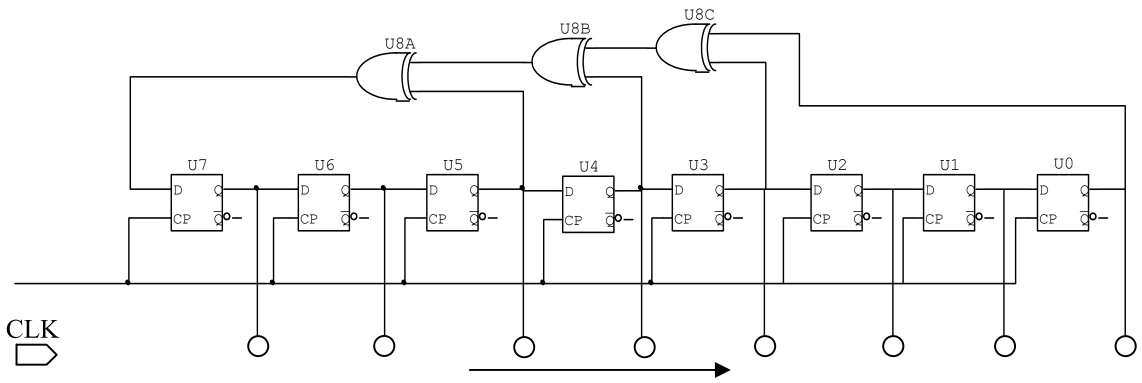

Figure 1: 8-bit LFSR RNG

The recurrence equation depends on the number of bits. Table 1 shows recurrence equations for bits 2 to 8.

Table 1: Taps for Maximal-Cycle LFSRs with 2 to 8 bits

No. of bits	Tap positions	Maximum Period
2	[1,0]	3
3	[2,0]	7
4	[3,0]	15
5	[3,0]	31
6	[5,0]	63
7	[6,0]	127
8	[6,5,4,0]	255

Hardware Implementation of the 8-bit LFSR was straightforward. The LFSR contained 8 D-type flip-flops and 3-two-input XOR gates. The behavior of the circuit was described using the VHDL and the XC4005XL was configured using Webpack software available from Xilinx. It consumed one FG Function generator out of 392 available in the XC4005XL and 8 CLB flip-flops out of 392.

To test the speed, another subsystem was created which comprised a counter (0 to 999). In each clock pulse the counter incremented by one as a new random number was generated. The system gave a pulse when the counter reaches to 999. The time duration between two pulses equal to the generation of 1000 random numbers. The average time taken to generate a random number was about 21.8 ns.

A sample of random numbers generated by LFSR is shown in figure 2. The obvious weakness of the LFSR is the correlation in the sequence. At any time step *t* there is 50% probability that the value at time *t+1* can be predicted. For an n-bit LFSR, if the present value is *v*, then the next value will be $v/2$ or $v/2+2^{n-1}$ [3]. This is shown in figure 3. In LFSR generator, a new random value contains (*m-1*) bits from the old value and only one bit of new information appears in the new value. This could be the reason for high correlation.

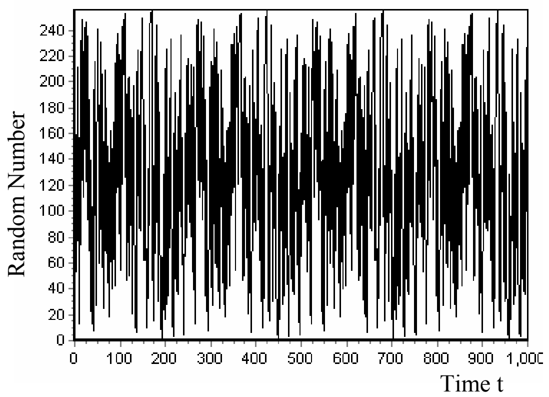

Figure 2: Sample of Random numbers

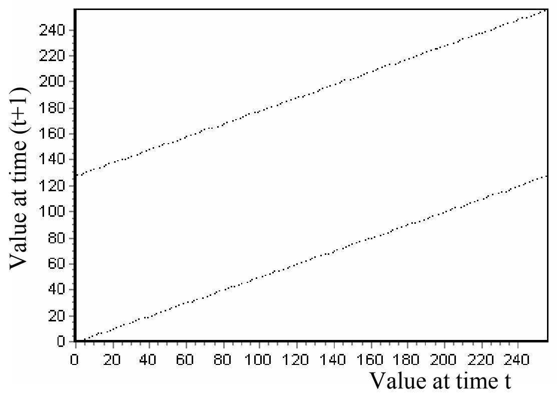

Figure 3: Correlation

However, the random number sequence is highly uniform (see figure 4). Thus, this method is still used in many applications with large registers. As the register size increases, the period becomes extremely large. For instance, if 64 bit register is taken, the period is 18,446,744,073,709,551,615.

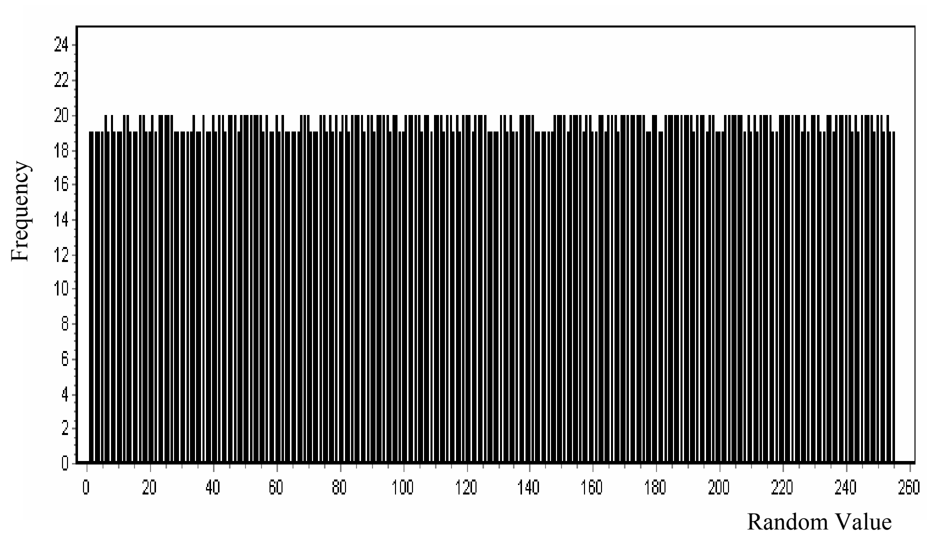

Figure 4: Distribution of Random Numbers

As the seed, a non-zero value should be given to the register before starting the process. If all zero value appears in the register, XOR operations continue to produce zeros and output becomes always zero.

2.2 Leap-forward LFSR RNG

The correlation effect of the LFSR method can be reduced with a slight modification. Instead of reading random numbers at each clock pulse, it is possible to shift all the old bits so that the new number will have only new bits to appear in the register. This can be easily done with a counter of modulo *m* for *m*-bit LFSR. The major drawback of this approach is that it reduces the speed. When this technique was applied, uniformity was similar to LFSR (see figure 6), but the correlation had been reduced significantly as shown in figure 5. Since the new random number is read after *m* cycles of *m*-bit LFSR, it is called Leap-Forward LFSR.

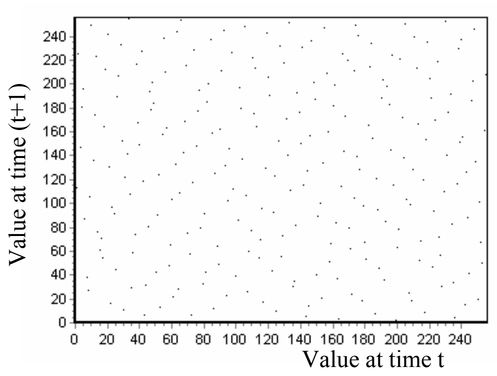

Figure 5: Correlation

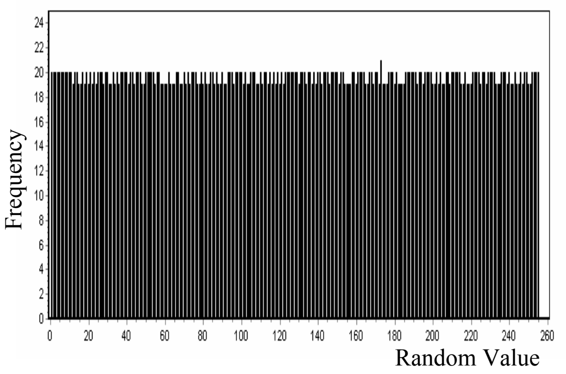

Figure 6: Distribution of random numbers

To construct n-bit random number generator it requires *n* number of flip-flops and several XOR gates. The recurrence equations show how the taps are formed using taking the outputs of middle stages. The nth bit generated sequence repeats after 2^k-1. Thus, the 8-bit random number sequence repeats after 255. This is called the cycle length. When the number of bits is increased the cycle length also increases exponentially.

To implement this generator in the FPGA, the LFSR RNG was extended with an internal 8-bit counter (0-7). The function of the counter was to keep the track of shifting the content of the shift register. With clock pulses the counter incremented by 1 in parallel to the function of the LFSR. When the counter reached to the maximum value 7, the shift register contents were loaded to the output lines. 21 FG function generators, 1 H function generator and 20 CLB flip-flops were utilized by the system. The average time taken to generate a random number was 162.4 ns which is approximately 8 times slower than the LFSR generator.

2.3 Cellular Automata RNG

Cellular Automata (CA) has been used in many scientific calculations and simulations. CA is based on a set of local rules which are defined as local relationships between it-self and the neighbouring cells. Conceptually, simple CA rules are used to produce complex behaviors. Resent studies have shown that CA can produce patterns whose features cannot readily be predicted, and in fact often seem completely random [5].

CA random number generator is based on linear finite machine each consisting of 1-D array of cells, each cell is allowed to communicate only with the immediate neighbors based on a rule. The connecting rule can be shown in a simple algorithm as follows [6].

```
$I_1[1] = O_2[2]$
$I_2[1] = O_2[N]$
For ( i =2, i<N, i++)
{
        $I_1[i]=O_2[i+1]$
        $I_2[i]=O_2[i-1]$
}
```

Here I_1 and I_2 are the inputs and O_1 and O_2 are the outputs of the random number generating cell. A group of cells are connected according to the above algorithm is shown in figure 7.

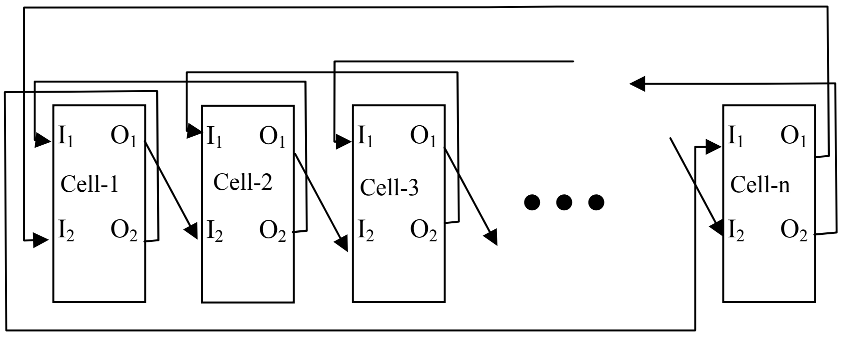

Figure 7: Cell Architecture of CA Random Number Generator

To create the 8-bit random number, 8 cells were connected according to the above rule. Output random numbers were read from the O_2 output of 8 cells. The required precision of the random number can be obtained by adding or removing cells.

Random number generating cell basically contains a RAM, XOR gates and flip-flops. The RAM holds 32 bits. The output O_1 and the 4-bit counter are concatenated so that feedback bit (O_1) becomes the MSB to form the 5-bit address line to access the 32 RAM cells. The new input bit I_1 is stored in the current address and previous value of that address is XOR to generate new bits (see figure 8).

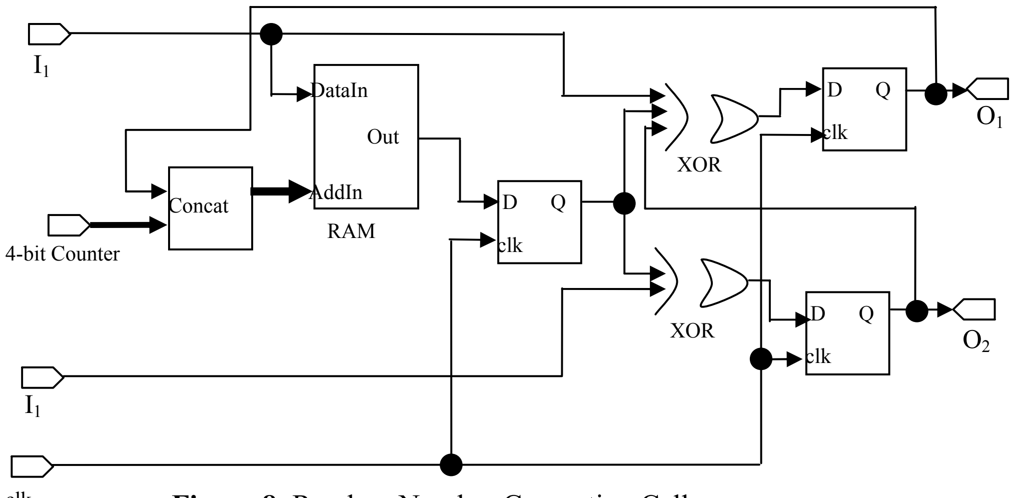

Figure 8: Random Number Generating Cell

To start the operation, a 32 bit pattern is stored in the RAM as the initial seed. Once started, the RAM is updated in each clock and random bits appear at the outputs. N-number of random number generating cells can be combined to construct an N-bit random number generator which produces an N-8 bit random number in each clock pulse.

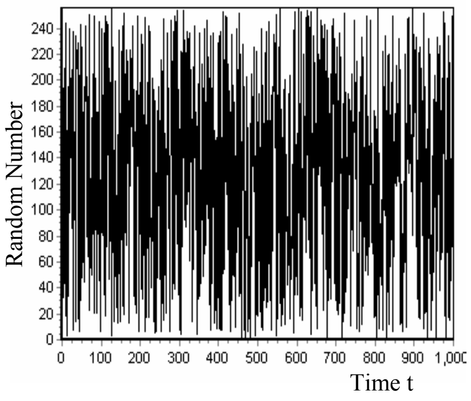

Figure 9: Sample of Random numbers

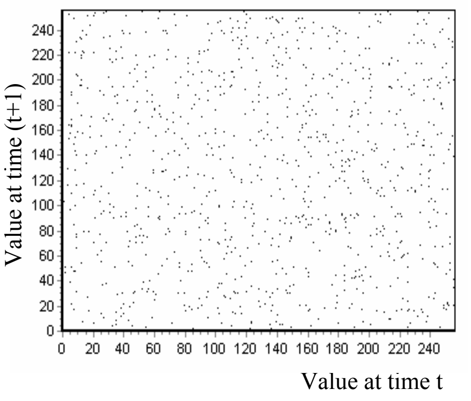

Figure 10: Correlation

Constructing the random number generating cell was the major challenge. Initially each component was constructed individually as VHDL sub-circuits and later combined them to form a single unit as the random number generating cell. Then, 8 random number generating cells were combined to create the 8-bit RNG. This RNG did not fit to the XC4005XL FPGA since it required more resources. To compare the performance of this system with other generators, simulated results were taken. The 'ModelSim' software was used for the simulation. Figure 9 shows the variation of random numbers with time. The relationship between numbers shows no apparent correlation (see figure 10).

The histogram of the distribution of random numbers shows that the sequence is not as uniform as in the previous methods (see figure 11).

The sequence generated by the 1-D CA does not seem to be repeating as in the LFSR methods. Since it is operated in parallel, it gives a new random number in each clock pulse. However, it consumes a large amount of hardware resources. Each random number generating cell contains single bit wide 32 registers, in addition, flip-flops for delay lines and XOR gates are used. When this cell is replicated to construct N-bit random number generator, the required resources are multiplied. This method is more suitable for modern configurable devices since they contain vast amount of resources.

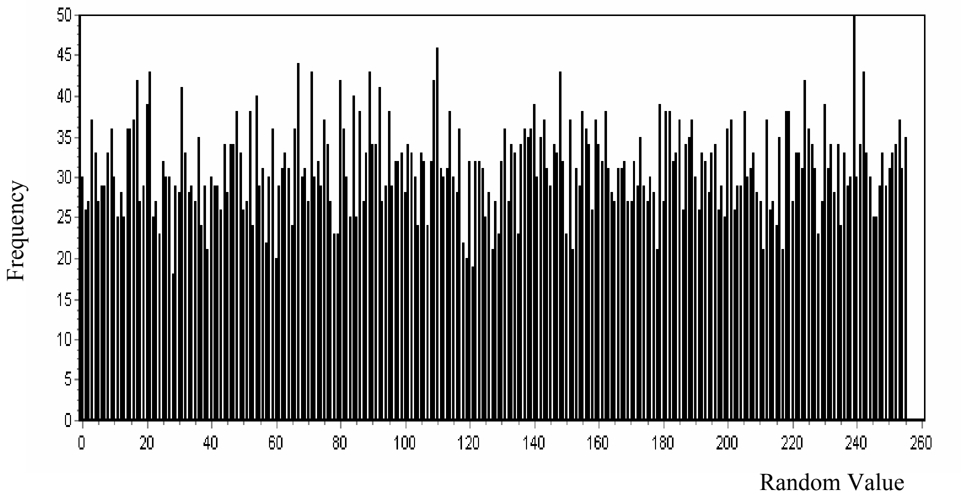

Figure 11: Distribution of Random numbers

2.4 Combined RNG

Since 1-D CA random number generator has high hardware resource requirements, it cannot be implemented in small configurable devices. Although it is possible to reduce the number of cells of the RNG, the resulting reduced precision can cause problems in a particular application.

A possible solution to this problem is to combine 4-bit CA random number generator with 4-bit LFSR RNG to construct 8-bit RNG.

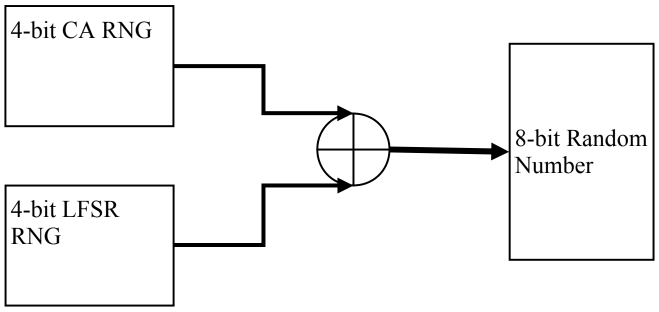

Figure 12: Combined Generator

To test the randomness, an 8-bit RNG was constructed combining a 4-bit CA RNG and 4-bit LFSR RNG as shown in figure 12. Once this system is configured in the FPGA, it utilized 243 FG function generators out of 392, 26 H function generators out of 196 and 150 CLB flip-flops out of 392. 4-bit CA generator performed correctly at 25 MHz. Therefore, 4-bit LFSR generator had also to be clocked at the same frequency. As in earlier cases, time taken to generate 1000 random numbers was measured and the average time was about 40.8 ns.

The random number sequence generated by this RNG shows better results than 8-bit LFSR random number generator (see figure 13). As in the LFSR random number generator there is a distinct pattern, but for each value of x_t there are several possible values for x_{t+1}. Further, it produced a relatively uniform random number sequence (see figure 14).

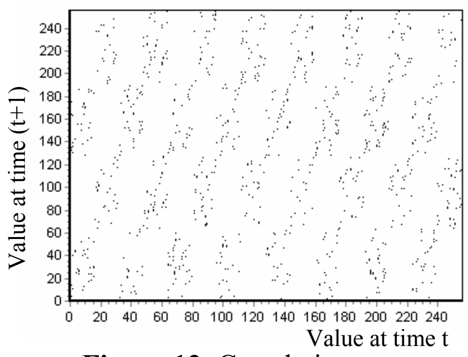

Figure 13: Correlation

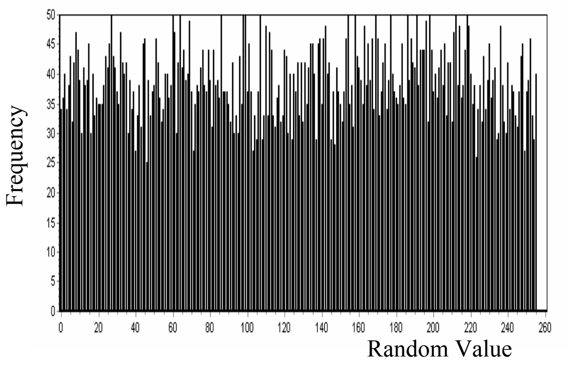

Figure 14: Distribution of random numbers

3. PERFORMANCE COMPARISON

3.1 Resource utilization

Resource utilization for each of the implemented random number generator is given in table 2. It is clear that the LFSR random number generator utilizes minimum amount of hardware resources while the combined generator uses considerable amount of logic resources. Thus, the CA based random number generators may not be economical to implement in low density configurable devices.

Table 2: Resource utilization of each random number generator

RNG	**FG Function Generators**	**H Function Generators**	**CLB Flip Flops**
LFSR	1/392 (0.3%)	0/196 (0.0%)	8/392 (2.0%)
LFSR with 8-bit Counter	21/392 (5.4%)	1/196 (0.5%)	20/392 (5.1%)
8-bit Cellular Automata	463/392 (118%)	48/196 (24%)	280/392 (71%)
Combined Generator	243/392 (62%)	26/196 (13%)	150/392 (38%)

3.2 Execution speed

Table 3 shows the average time taken to generate a random number by each generator. LFSR random number generator takes the minimum time to generate a random number while LFSR with a counter takes highest time per random number.

Table 3: Time taken to generate a single random number

	RNG (XC4005XL with 50 MHz clock)	**Time to generate one random number (ns)**
1	LFSR	21.8
2	Leap-Forward LFSR RNG	162.4
3	8-bit Cellular Automata	40.8
4	Combined Generator	40.8

3.3 Monte-Carlo test - accuracy

In order to check the accuracy of the random numbers generated above, the well known Monte Carlo method of calculating π was employed [7]. Using a simple C++ program into which the above generated random number sequences were fed, π was estimated. For each set of sequence the value of π obtained is shown in table 4. The sample size was 5000 points. The estimation of π using the standard random number generator supplied with the C++ library is also shown for comparison.

Table 4: Estimated values of π for each random number generator

	Random Number Generator	Estimated value of π
1	LFSR	2.994
2	Leap-Forward LFSR RNG	3.151
3	Cellular Automata RNG	3.129
4	Combinational Method	3.132
5	Software generator (C++)	3.158

According to the figures, it is clear that the random number sequence of Cellular Automata RNG gives close approximation to the actual value of π, while the value calculated with the LFSR generator deviates somewhat from the actual value.

4. CONCLUSION

This work shows that LFSR gives faster random number sequence with utilizing lower hardware resources, but the sequence of numbers are highly correlated. Although, Leap-forward LFSR gives slightly improved random sequence than that of LFSR, with consuming extra hardware resources, it is slow. Cellular Automata RNG gives random sequence that seems impossible to discern a pattern. However, it consumes a large amount of hardware resources. This method would be ideal for higher density devices while combined generator would be suitable for lower density configurable devices.

Acknowledgements: Financial assistance by IPPS, Uppsala University, Sweden (research grant number SRI:01/3) is acknowledged.

REFERENCES

1. Free Encyclopedia, Pseudo Random Number Generators, http://en.wikipedia.org/wiki/Random_number_generators
2. Horowitz P., Hill W., The Art of Electronics, 2nd Edition, Cambridge University Press (2001)
3. Martin P., An Analysis of Random Number Generators for a Hardware implementation of Genetic Programming using FPGAs and Handle-C, Technical Report, CMS-358 (2002)
4. Xilinx Product Specification, XC4000 series Field Programmable Gate Arrays, Version 1.6 (1999)
5. Wolfram S, Random Sequence Generation by Cellular Automata, www.stephenwlfram.com

6. Abdulai M., Inexpensive Parallel Random Number Generator for Configurable Hardware, SUPERB (2003)
7. Sezen T., The Basic Monte Carlo Simulation, http://www.stanford.edu/group/pandegroup/folding/education/montecarlo