RAPID PROTOTYPING AND DESIGN OF A FAST RANDOM NUMBER GENERATOR

Juan Franco

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

May 2012

APPROVED:

Saraju P. Mohanty, Major Professor
Elias Kougianos, Co-Major Professor
Mahadevan Gomathisankaran, Committee Member
Barrett R. Bryant, Chairman, Department of Computer Science and Engineering
Costas Tsatsoulis, Dean, College of Engineering
James D. Meernik, Acting Dean of the Toulouse Graduate School

Franco, Juan. Rapid prototyping and design of a fast random number generator. Master of Science (Computer Science), May 2012, 52 pp., 10 tables, 22 figures, references, 29 titles.

Information in the form of online multimedia, bank accounts, or password usage for diverse applications needs some form of security. The core feature of many security systems is the generation of true random or pseudorandom numbers. Hence reliable generators of such numbers are indispensable. The fundamental hurdle is that digital computers cannot generate truly random numbers because the states and transitions of digital systems are well understood and predictable. Nothing in a digital computer happens truly randomly. Digital computers are sequential machines that perform a current state and move to the next state in a deterministic fashion. To generate any secure hash or encrypted word a random number is needed. But since computers are not random, random sequences are commonly used. Random sequences are algorithms that generate a pattern of values that appear to be random but after some time start repeating. This thesis implements a digital random number generator using MATLAB, FGPA prototyping, and custom silicon design. This random number generator is able to use a truly random CMOS source to generate the random number. Statistical benchmarks are used to test the results and to show that the design works. Thus the proposed random number generator will be useful for online encryption and security.

Copyright 2012

by

Juan Franco

ACKNOWLEDGMENTS

I deeply acknowledge my major professor, Dr. Saraju P. Mohanty, for his time, feedback, and sincere encouragement for this research. His knowledge helped me to get involved in and complete this thesis topic. I also thank my co-major professor, Dr. Elias Kougianos, for the time and all the key suggestions he provided for this research and implementation. I thank my committee member Dr. Mahadevan Gomathisankaran. I also thank the Department of Computer Science and Engineering (`http://www.cse.unt.edu`), which is a key unit in the College of Engineering at the University of North Texas (`http://www.unt.edu`), for making all of this possible. I thank all the members of the NanoSystem Design Laboratory (NSDL, `http://nsdl.cse.unt.edu`) for all the discussions we had for this research.

CONTENTS

CHAPTER 1

INTRODUCTION

In this chapter a brief description of some example applications of random number generators are provided that served as the motivation of the research undertaken in this thesis. Two emerging multimedia security frameworks, a secured digital camera and a net-centric multimedia processor are briefly discussed focusing on their security parts. An overview of related research in random number generators is also provided along with a classification of random number generators.

1.1. Case Study Embedded Systems

1.1.1. Secure Digital Camera (SDC)

An embedded system which is used in day-to-day life is a digital camera. The digital camera may be a stand alone system or part of smart mobile phones. They may employ charge-coupled-device (CCD) sensors or complimentary metal-oxide sensors (CMOS). Cameras may be regular digital cameras or digital single-lens reflex (DSLR) among other forms. However, the digital signal processor (DSP) is the key component that performs the image of video processing in the camera. A digital camera is able to record multimedia, store them in a digital format, or transmit them over the Internet. For images or still pictures, formats like JPEG, TIFF, RAW, etc. are used. For videos, formats like H.264, MPEG, MPEG2, AVI, etc. are used.

One of the key advantages of using a digital format is that it allows for high-visual quality and easy processing. Because it is so easy to process digital media, the need for digital rights management (DRM) [16, 15] rises. A system-level solution is a secure digital camera (SDC). The important blocks of the SDC are provided in Fig. 1.1. The use of encryption along with digital watermarking is necessary to accomplish effective DRM. Encryption is a method of transforming a digital media to a secure form so that unauthorized users can not view or tamper with the media. Encryption allows us to perform access control to the media

[6]. Watermarking allows us to embed digital rights on a media so that we can determine the origin of the digital media [18, 14].

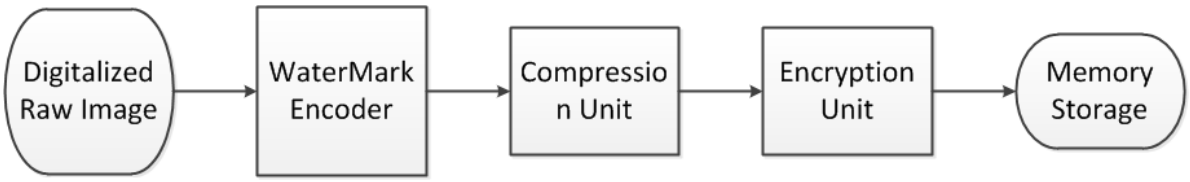

FIGURE 1.1. The key components of the secure digital camera (SDC).

1.1.2. Net-Centric Multimedia Processor (NMP)

Digital broadcasting is preferred over traditional analog broadcasting due to several advantages of analog transmission: digital TV has a single standard compared to analog TV, low production cost, low bandwidth requirements, capability of displaying computer applications over the same screen as that of the TV. Such digital signal transmission-reception is possible over terrestrial, satellite, and cable links, carrying compressed streams. The Internet Protocol (IP) is an important protocol and widely used for data transmission over the Internet. Thus, coexistence of the digital TV and IP protocols is necessary and intuitively will be cost effective. The net-centric multimedia processor (NMP) is a system that is needed for this application [13, 16]. The NMP has the added features to ensure security and copyright issues of digital video, which will need separate engines for compression, watermarking, scrambling, and cryptography.

A diagrammatic view of the NMP system is presented in Fig. 1.2. The NMP consists on several processing elements (PE). In the NMP, one or more PEs process video and others process Internet packets for broadcasting with added security and protection. Video is one of the most important digital multimedia [13]. In [16], the Net-Centric Multimedia Processor (NMP) was introduced along with power efficient and reconfigurability features. This is a video processor that is able to do integrated encryption, watermarking and video compression in real time. The advantage of such a system is that it could be integrated into current systems to allow easy sharing of the media in existing protocols such as IP. Each

of the elements are optimized for specific tasks to increase performance and lower area and power consumption [16].

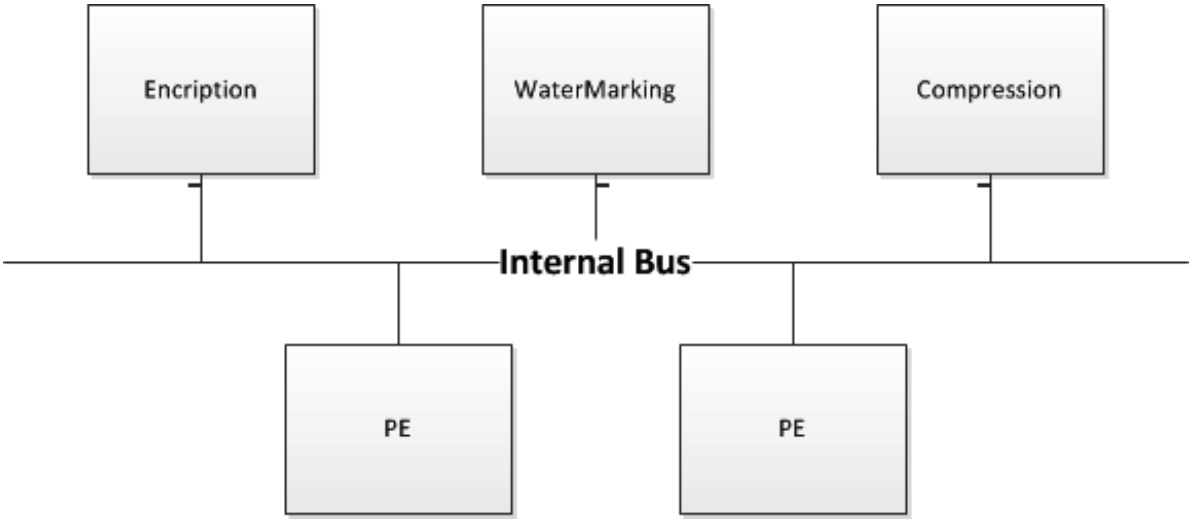

FIGURE 1.2. Key components of the net-centric multimedia processor (NMP). The NMP is designed using a multiple core approach for speed and power dissipation tradeoffs.

1.2. The Need of Security in Embedded Systems

There is a growing need for security in embedded systems for multiple reasons and perspectives. Security and copyright protection can deal with the content that embedded systems handle. Security can also deal with the protection of the embedded system itself from the process of reverse engineering and duplications. However, the scope of this thesis is the security of the content and information that an embedded system handles.

One way of securing embedded systems is by using watermarked multimedia. Hardware assisted watermarking in the context of embedded systems for DRM at the source has several advantages [11]. In general watermarking is composed of three parts: the encoder, the decoder and the comparator. The Encoder grabs the original multimedia along with a watermark and provides a watermarked multimedia file. The decoder grabs the test image and a watermark, then it attempts to extract the watermark. The comparator gets the output from the decoder and the original multimedia and compares them by correlating them

and provides a number. Depending of the value of this number one can determine if the original multimedia and the test multimedia are the same image.

Hardware assisted DRM uses simultaneous watermarking and encryption of images or video and is suitable for real-time applications [15]. The sequence of the use of watermarking and encryption schemes depends on the target application. The encoding process encrypts the watermark logo using an encryption algorithm and a public key. Then the result is embedded on to the original image to get a watermarked image. To authenticate if a given image is the original image is done by extracting the encrypted logo from the watermarked image and comparing it with the original encrypted logo. The decoding process needs the original logo to compare and also needs the encryption process and the public key. This add complexity to the system and allows for access control to the authentication process.

1.3. Sample Applications of Random Number Generators

Random number generators are used in many applications including statistical sampling, statistical analysis, Monte Carlo simulation, cryptography, and watermarking. Each of these applications actually depend very heavily on random numbers. It can be seen that all of these application are performed today in digital computers, where the generation of random numbers is less trivial than may initially appear.

Statistical sampling is the technique of sampling large amounts of data by just sampling a small amount of data that is representative of the entire set [3]. The small amount of samples is chosen by using rules such as probability characteristics of the set of all samples. Random numbers are needed to select the set of data that will be used to sample so that fast sampling can be performed as well as the representative data can be obtained.

Data analysis is the method of analyzing data to infer about their nature and fidelity [27]. When analyzing very large amounts of data, statistical analysis can be used to analyze a smaller amount of data. Thus random number generators are needed to determine or predict the rest of the data that is not analyzed.

An area where random number generators are used is in simulation. Simulation is widely used in weather prediction, IC design, material reaction, etc. Random number

generators becomes a crucial part of the simulation when trying to simulate complex systems. A very well known application or process that uses this is the Monte Carlo Simulation [1].

Cryptography is the process of securing information in a way that only allowed users can read it. Normally this process is performed by taking the information one wants to secure and then using some algorithm to change the information in a way that it can't be read. An other application of cryptography is to change the data back to its original format. These processes require the use of random numbers to generate seeds. A seed is the initial state where a random number generator starts [19].

Digital watermarking involves hiding some form of data or information inside digital multimedia [17]. The watermark is generated by a pseudorandom sequence generator. The unit that generates the binary watermark consists of linear feedback shift registers (LFSR). The LFSR is a very crucial unit in watermark security and detection. It is a sequential shift register with combinational feedback logic around it that causes it to cycle pseudo randomly through a sequence of binary values. The LFSR consists of flip-flops (FFs) as sequential elements with feedback loops.

1.4. Random Number Generators: A Broad Prospective

Random number generators have been an important topic of research since the invention of the digital computer. The root cause of the problem is that digital computers cannot generate truly random numbers due to their deterministic nature of computing. A system that can generate truly random numbers is shown in Fig. 1.3. The problem is that this simple mechanism can't be implemented in a digital computer to generate random number. This is because the logic functions, states and transitions of digital systems are well understood and deterministic. Digital hardware designers always try to avoid getting into unknown states or situations where they don't know what the output could be. Nothing in a digital computer happens truly randomly.

Random number generators are mainly used for encryption and security. To generate any secure hash or encrypted word, a random number is needed. The same is true in cryptography. Most cryptographic protocols use random number generators. However, as

FIGURE 1.3. Tossing a coin to generate random numbers.

the computers are not random, researchers have came up with random sequences. Random sequences are algorithms that generate a pattern of values from a given seed that appears to be random but after some time it starts repeating itself in a loop.

A true random number generator using only Complementary Metal-Oxide- semiconductor (CMOS) technology is important. The advantage of this is that it makes it much easier to include in existing circuits. This has an advantage of using the high density in nanoscale CMOS [10]. The RNG in [5] is used as a case study for this thesis. The design takes advantage of an unstable state of a digital system and it uses the randomness of real phenomena to stabilize it and produce random results.

These bits will be used to manipulate a LFSR to produce good quality random num-

bers. The reason for this is that the cells themselves are too sensitive to process variation and this produces a need for a conditioning logic. The final goal is to produce good random numbers that will pass very strict randomness tests to prove that these numbers can be used for encryption, watermarking, or other security applications.

1.5. Motivation of Research for this Thesis

The primary motivation for the research undertaken in this thesis is the urgent need for a robust random number generator to meet security demands. The need of good random number generation in digital systems has been around for a long time and several research and development activities have been undertaken. Many different ideas, types, and alternatives have being attempted. From the hardware point of view, pure digital implementation or a mixed-signal implementation have been used.

Digital implementations predominately use CMOS technology. The problem with this approach is that CMOS technology is very deterministic and predictable. Particularly, the switching of devices is predictable. In these implementations, pseudorandom number generators or a random sequence generator are being used. These are algorithms that produce a sequence of numbers in a random like way by using two things: an initial condition and seed [21]. The initial condition will determine where the sequence will start. The seed determines the pattern the sequence will follow. Even though digital implementations appear to be random, they are cyclical. In other words the pattern will repeat after certain cycles. Also since they depend on two parameters (initial condition, seed), they can be cracked. In other words, the sequence can be predicted if the present state can be replicated [28].

Mixed-signal implementations use both CMOS and analog components. Some of the most common designs use some analog component such as resistors, capacitors, or even RF receivers. Then they convert the analog signal to digital using Analog To Digital Converters (ADC).

1.6. Organization of this Thesis

The thesis is organized in six chapters: Introduction, State or the Art in Random Number Generators, Field Programable Gate Array (FPGA) Prototyping of the Random Number Generator, Circuit Design of the Random Number Generator, Experiments and Evaluation, Conclusion and Future Research.

In Chapter 1, the thesis discussed embedded systems and the security in them. This chapter also discusses some application of random number generators and provide some background in random number generators. Finally the motivation of this research has been discussed.

In Chapter 2, State of the Art in Random Number Generators, the thesis briefly outlines different types of random number generators. In this chapter other research that has been done in this topic is highlighted. Analog RNG, pseudorandom RNG, and digital RNG, are discussed. The contributions of this thesis are then presented.

In Chapter 3, Field Programable Gate Array (FPGA) Prototyping of the Random Number Generator, the thesis discusses the implementation of the proposed design in an FPGA. The MATLAB simulation is also performed as a proof of concept. Then the implementation on the FPGA including the blocks and ports is presented. The Register-Transfer-Level (RTL) Synthesis of the FPGA design is performed to obtain a hardware cost perspective.

In Chapter 4, Circuit Design of Random Number Generator, the circuit and the layout design of the RNG cell are implemented in Cadence Virtuoso.

In Chapter 5, Experiments and Evaluation, the thesis presents the experiments used to test the designs and the results from them. This chapter describes how the RNG cell was tested. The experimental results are also presented. The test of the FPGA and MATLAB design are discussed.

In Chapter 6, Conclusion and Future Research, the thesis briefs the conclusions of this research and also some future research that could be done to this thesis is presented.

CHAPTER 2

STATE OF THE ART IN RANDOM NUMBER GENERATORS

This chapter describes research which is related to the scope of this thesis. The chapter is divided into sections for more clarity of the discussion. The classification of the prior research research is presented as shown in Fig. 2.1. The first one is analog random number generators. These are designs that require analog computing. The second one is pseudorandom number generators. This is where the numbers are constructed by using a function where the values appear random. The last one is digital random number generators. These generators are designed using complementary metal-oxide semiconductor (CMOS) technology.

2.1. Analog Random Number Generator

Many attempts to create a random number generator using analog devices have been presented in the literature. Intel in 1999 introduce a truly random number generator using analog components [8]. This random number generator uses a thermal noise source. It uses the noise captured and digitalizes it to create a specific number. This is a very efficient way to create random numbers. They only problem is the cost due to the use of analog components. Analog components can make a chip more complex to design since they differ is size and power requirements. They tested their results with various benchmark tests including, DIEHARD, FIPS, and Knuth's tests. Their design passed all of the above tests.

A random number generator for communication systems was introduced in [23]. They demonstrated a 30 Gbit/s generator using superconductive materials. A superconductive random number generator is designed by using a single-flux-quantum balanced comparator and a current source. Physical phenomena such as thermal noise and electronic noise [23] are used in this design. They used the same tool to test their results, i.e. the NIST statistical test suite.

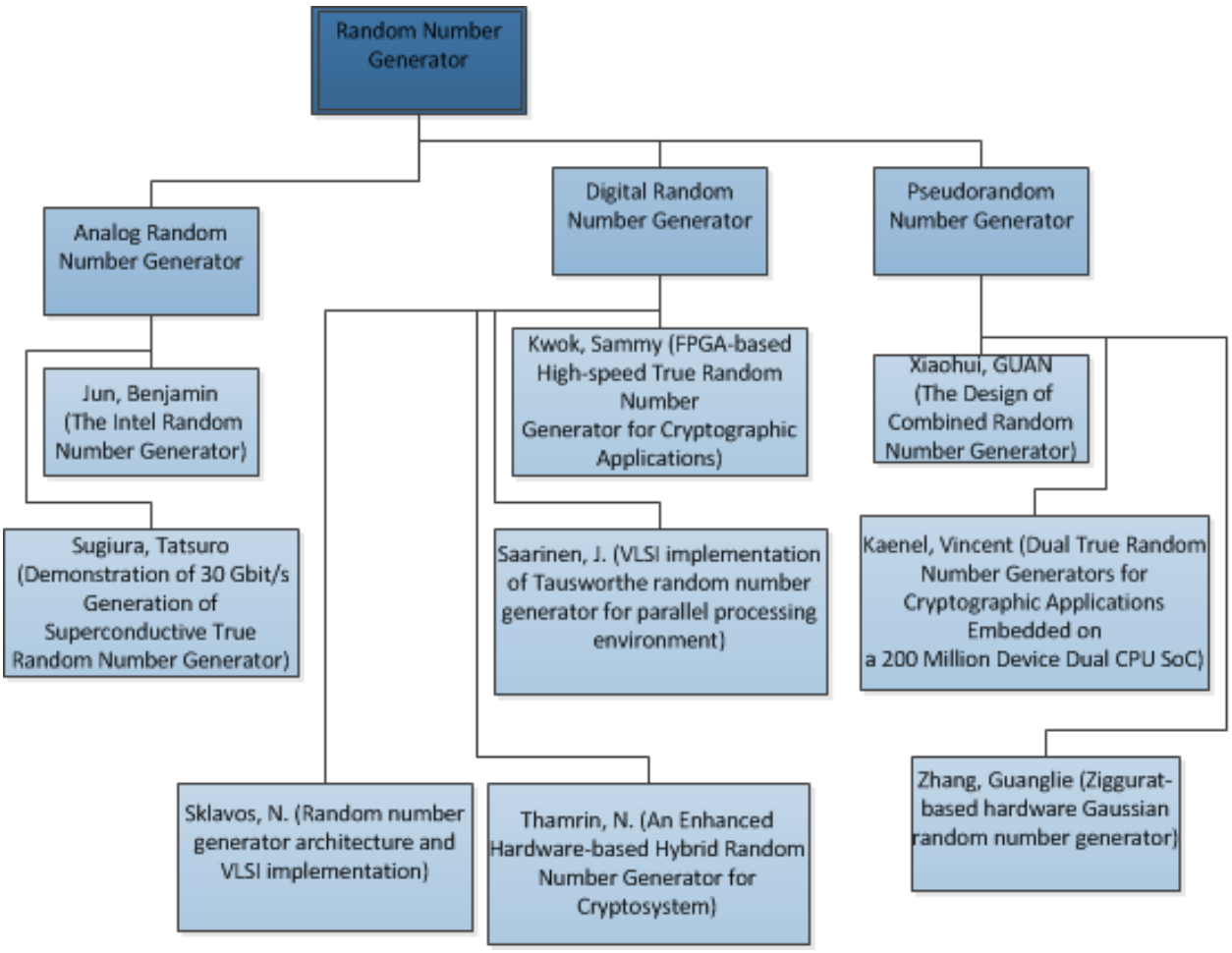

FIGURE 2.1. The state-of-the-art in random number generator design.

2.2. Pseudorandom Number Generators

Some research results show that by combining different algorithms to generate numbers that look random will generate a more robust random number sequence as depicted in Fig. 2.2. The research presented in [26] demonstrated that the combination of the logistic chaotic mapping [4], linear congruential method [29] , and hybrid optical chaotic mapping [25] will generate a combined algorithm than can produce very strong pseudorandom numbers that will pass the randomness tests. These research works have shown the results from various experimental runs test and they passed common randomness tests. This was only showing a pseudorandom sequence which will eventually repeat and it can also be predicted since it is purely deterministic.

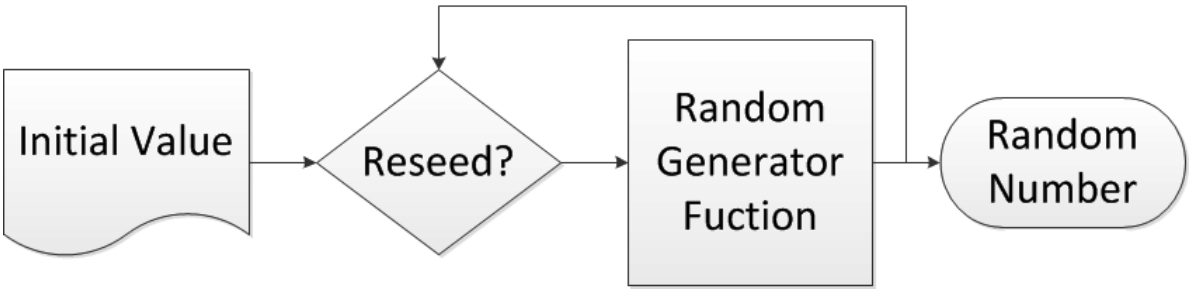

FIGURE 2.2. Block diagram of a pseudorandom number generator.

Another way researchers have tried to generate random sequences is by using a Gaussian noise simulator algorithm and obtain a sequence from there. Gaussian noise simulators are used to generate white noise. They are extensively used in various science and engineering applications to simulate and test radio frequency systems. In [28], the author implemented a Gaussian random number generator (GRNG) using hardware. Similar to our research, the authors used a LFSR based implementation to test it. So in other words, they used a GRNG instead of the random number generator cells. They have compared their results (presented in Table 2.1) with a Tausworthe implementation.

Table 2.1: Statistical test results for randomness of various designs in the prior related research

TEST	Tausworthe [28]	LFSR [28]	Sugiura ([23])	Thamrin [24]	Kwok(T=2, M=1) [12]
Birthday	0.908125	0.718022	N/A	N/A	0.5468
OPERM5	0.659361	0.894649	N/A	N/A	0.7081
Binary Rank (31 x 31)	0.782536	0.894649	N/A	0.294517	0.3917

Binary Rank (32 x 32)	0.357046	0.768956	N/A	N/A	0.9890
Binary Rank (6 x 8)	0.324027	0.261791	N/A	N/A	0.8859
Bitstream	0.598578	0.443253	N/A	N/A	1.0000
OPSO	0.431957	0.571626	N/A	N/A	1.0000
OQSO	0.492004	0.559064	N/A	N/A	1.0000
DNA	0.432068	0.525910	N/A	N/A	0.3098
Stream Count-the-1	0.459779	0.564513	N/A	N/A	0.9977
Byte Count-the-1	0.609182	0.560009	N/A	N/A	0.0212
Parking Lot	0.941697	0.460448	N/A	N/A	0.5201
Minimum Distance	0.337831	0.999999	N/A	N/A	1.0000
3D Spheres	0.952286	0.634016	N/A	N/A	0.6350
Squeeze	0.113189	0.855206	N/A	N/A	0.7088
Overlapping Sums	0.139815	0.717343	0.066882	0.587053	0.5779
Runs Up	0.555513	0.575984	0.017912	0.572333	0.0239
Runs Down	0.253845	0.552341	N/A	N/A	0.9794

Craps	0.395120	0.841941	N/A	N/A	0.7242
Frequency: Monobit	N/A	N/A	0.122325	0.326473	0.6789
Frequency: Block	N/A	N/A	0.122325	0.255473	0.7607
Cumulative Sums-Forward	N/A	N/A	0.122325	0.565004	0.9205
Cumulative Sums-Reverse	N/A	N/A	0.122325	N/A	0.9307

2.3. Digital Random Number Generators

Different random number generators have been proposed using different architectures and algorithms and prototyped in FPGAs [2]. In [12] a truly random number generator using an FPGA is presented. Using a Xilinx Vertex II Pro FPGA they were able to use the built-in clock to get random seeds. This FPGA has a Delay-Locked-Loop (DLL) clock generator. The problem with this clock is that is susceptible jitter during transitions. This is normally a problem, but in this paper they were able to use the jitter to advantage for generating the random numbers. The problem with the values generated is that they could be biased to ones or zeros. By using a parity filter, they were able to get the bias close to 50-50.

Many implementations have been made for digital random number generators. In [21], the authors have presented the implementation of Tausworthe random number generator optimized for parallel processing environments. They did a VLSI implementation in silicon and simulated using Monte-Carlo simulation. The advantage of this simulation is that it

takes in consideration process variation. In this paper the authors demonstrated a successful simulation of the random number generator passing some of the standard randomness tests including the following: moment test, coupon collector's test, serial test, poker test, the equidistribution test, run test, Gap test, Kolmogorov-Smirnov test, and Visual test [21].

Implementations of random number generators have been proposed to predict the present and future state of an architecture. In [22], the authors have introduced such an architecture. The authors proposed a random number generator architecture that produces 160-bit words using a SHA technique. The size of the word is ideal for many security applications. They implemented their design using VHDL and synthesized it for a Xilinx FPGA. They tested the architecture with the FIPS randomness test suite. The architecture was able to pass all tests. This shows that it could be used for cryptographic applications.

In [24], a very similar implementation is presented. The architecture is shown in Fig. 2.3. The authors used an LFSR to generate bits with the combination of a truly random source. The truly random source is generated by an unstable clock frequency [24]. This is a clock that is oscillating at a very high speed and with high sensibility to ambient conditions. The proposed architecture used a simple XOR of the output from the LFSR and the bits from the truly random source. This implementation passed the NIST statistical test suite tests. The architecture has also been implemented using an FPGA.

2.4. Contribution of this Thesis

The random generator is simulated using MATLAB. FPGA prototyping of the random number generator is presented along with the register-transfer level (RTL) synthesis results. This thesis has introduced a random number generator based on cells. These cells can generate random bits using only CMOS technology. These bits are used to control an LFSR to produce a random number sequence that can't be reproduced. Unlike most random number generators that either require analog components or just produce a sequence that can be reproduced, this random number generator does not require analog components and the sequence can't be reproduced. Exhaustive experiments using state-of-the-art benchmarks were performed to test the quality of the numbers. The sequence produced passed several

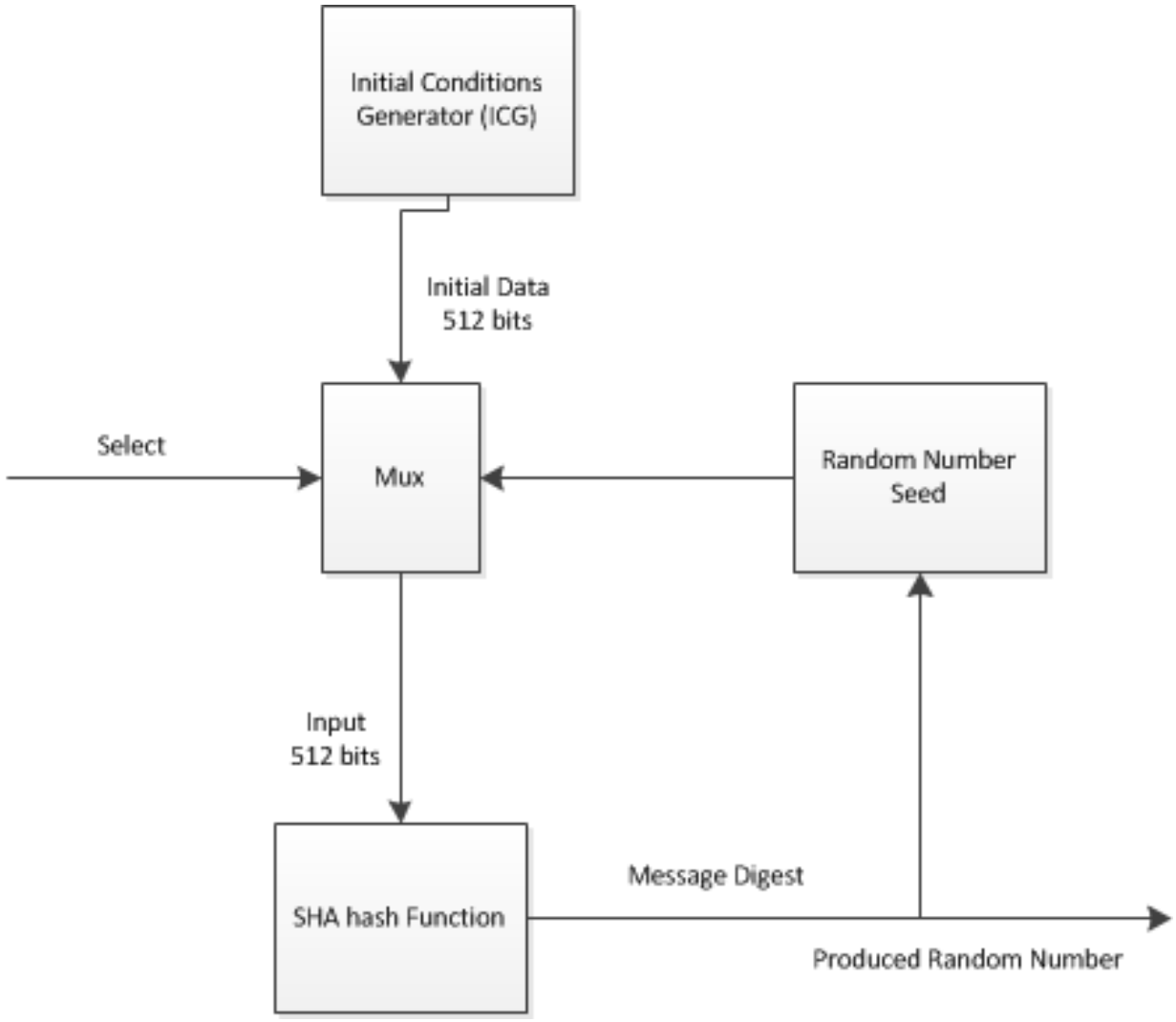

FIGURE 2.3. Architecture of a digital random number generator [22].

statistical randomness tests. This means that this can be used for encryption to a military standard.

CHAPTER 3

FPGA IMPLEMENTATION OF THE RANDOM NUMBER GENERATOR

In this chapter the simulation of the proposed random number generator in MATLAB and its rapid prototyping using field programmable gate-array (FPGAs) are discussed. The steps and theory of the design are explained. First the chapter discusses the MATLAB implementation. In this section the theory behind the design is explained and the simulation of the whole design is then presented. Then the implementation of the design in Verilog for an FPGA is explained. This is followed by the RTL synthesis of the design.

3.1. Architecture of the Proposed Random Number Generator using MATLAB

A recent random number cell presented in [5] is considered for a full-fledged random number generator implementation in this chapter. The cells on their own are not be able to pass randomness tests. The cells need additional logic to make the results "more random". This is accomplished by evenly distributing bits and making the sequence more random like.

The overall idea of making the conditioner and the pseudorandom-number generator is depicted in Fig. 3.1. The overall design will be using the bits obtained from step 1. At this step, a cell is designed using 45nm CMOS technology. The cell generates all the bits and the resulting bits are stored. The overall architecture uses the XOR technique for the conditioner. A LFSR is used for the pseudorandom-number generation.

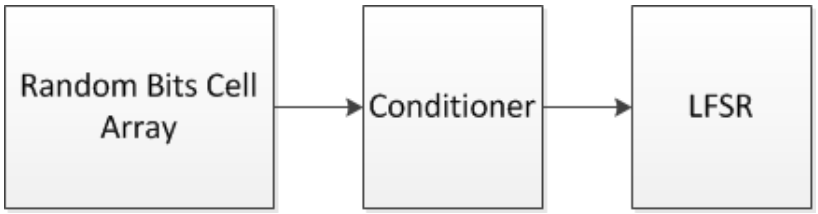

FIGURE 3.1. Three step process for the random number generator.

3.1.1. The XOR Conditioner

The conditioner is the block which is used to reduce the possible bias in bits from the random generator cells. Cells, because of manufacturing parasitics, will never give perfect

results. That is, their results will not be evenly distributed for ones and zeros.

In the current design, the conditioner raw bits are generated from the cells by performing XORing operation of the two bits. It is observed from the experiments that by XORing two randomly generated bits with bias probabilities for ones and zeros, one will be able to improve the bias (that is making it more like a 50-50 change on one and zero) by approx. 200%.

It is now discussed how the XOR operation will improve the bias of two cells. Assume that there are two cells A and B. The XOR operation of the values of the two cells is performed using Fig. 3.2. Let us assume the probability of the two as follows:

$$P(A_0) = PA_0, \tag{1}$$

$$P(A_1) = PA_1, \tag{2}$$

$$P(B_0) = PB_0, \text{ and} \tag{3}$$

$$P(B_1) = PB_1. \tag{4}$$

Here, $P(A_0)$ is the probability of cell A being zero. $P(A_1)$ is the probability of cell A being one. $P(B_0)$ is the probability of cell B being zero. $P(B_1)$ is the probability of cell B being one.

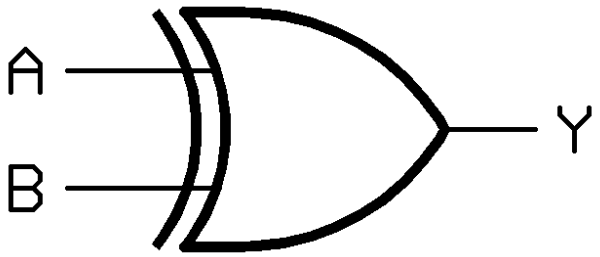

FIGURE 3.2. XOR gate with input of Cell A and B.

The truth table of an XOR gate looks as in table 3.1. By substituting the values from the truth table with probabilities, the values in Table 3.2 are obtained.

By multiplying the probabilities of each row the probability of Y in each row is

TABLE 3.1. Truth table of XOR logic gate.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 3.2. Truth table of XOR gate with probability from the cells.

A	B	Y
PA_0	PB_0	$PA_0 * PB_0$
PA_0	PB_1	$PA_0 * PB_1$
PA_1	PB_0	$PA_1 * PB_0$
PA_1	PB_1	$PA_1 * PB_1$

obtained as follows:

$$PY_0 = PA_0 * PB_0 + PA_1 * PB_1, \tag{5}$$

$$PY_1 = PA_0 * PB_1 + PA_1 * PB_0. \tag{6}$$

Where PY_0 is the probability of Y being zero and PY_1 is the probability of Y being one. The bias of a cell is obtained by using the following equation:

$$PB = \left(\frac{|0.50 - P_0| + |0.50 - P_1|}{2} \right), \tag{7}$$

where PB is the probability bias of the cell, P_0 is the probability of the cell being zero and P_1 is the probability of the cell being one. From this, the probability bias of two cells A and B is calculated as follows:

$$PB_{AB} = \left(\frac{|0.50 - PA_0| + |0.50 - PA_1| + |0.50 - PB_0| + |0.50 - PB_1|}{4} \right), \tag{8}$$

where PB_{AB} is the probability bias of both cells. We can also obtain the probability bias of Y as follows:

$$PB_Y = \left(\frac{|0.50 - (PY_0)| + |0.50 - (PY_1)|}{2} \right), \tag{9}$$

$$PB_Y = \left(\frac{|0.50 - (PA_0 * PB_0 + PA_1 * PB_1)| + |0.50 - (PA_0 * PB_1 + PA_1 * PB_0)|}{2} \right).$$

3.1.2. The Pseudorandom-Number Generator

The next step after the conditioner is the pseudorandom generator. This step is used to make the final bits for the random sequence. It uses the bits from the XOR conditioner to generate the final bits. In this design, a 32 bit liner feedback shift register (LFSR) is used for this step. An LFSR (Fig. 3.3) is made of a shift register of a given size, a few XOR gates and a tap. A tap determines which bits from the register will be XORed to obtain a new bit to shift in the register. The seed is the initial values in the register.

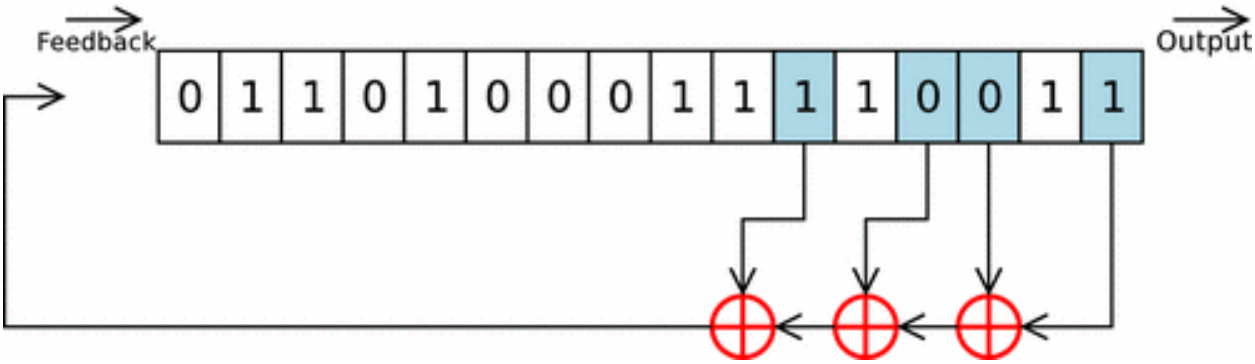

FIGURE 3.3. LFSR

This design uses the bits given by the XOR conditioner to determine the seed, the tap and the time of refresh (TR). The time of refresh (TR) is the time the design needs to wait until a new seed is supplied and a new tap to the LFSR. The units of TR are cycles. These are the cycles from the LFSR. TR is a 10-bit number in the proposed design. They come from the bits generated by the cells. The conditioner will output a 32 bit value. From these values the seeds, the tab and TR will feed. These values are denoted as LFSR_IN. A single bit from LFSR_IN is called bit(N) where N represents the bit number within LFSR_IN. A

function with LFSR_IN will look as follows:

$$LFSR\_IN = [bit(31), bit(30), ..., bit(1), bit(0)]\,. \tag{11}$$

The time of refresh is:

$$TR = [bit(22), bit(20), bit(18), bit(16), bit(12), bit(10), bit(6), bit(4), bit(2), bit(0)], \tag{12}$$

$$TR = LFSR\_IN[22, 20, 18, 16, 12, 10, 6, 4, 2, 0]. \tag{13}$$

The seeds will use all of the values from LFSR_IN so the function will be of the following form:

$$SEED = LFSR\_IN. \tag{14}$$

The following mechanism is used to have the tap values dependent on the cells. The bits and a tap are not selected at random. The taps that will give a maximal length are selected. For the longest length, the following cycle is obtained:

$$2^N - 1 = 2^{32} - 1 = 4294967296 - 1 = 4294967295. \tag{15}$$

A lookup table is obtained that contains 8 different taps that will give maximal length. Then 3 bits are picked from LFSR_IN to choose a row from the lookup table 3.3.

TABLE 3.3. Lookup table for taps used in this design.

Value	Bits	Tap
0	[000]	[31,30,28,0]
1	[001]	[31,30,4,3]
2	[010]	[31,29,7,2]
3	[011]	[31,29,6,3]
4	[100]	[31,28,5,4]
5	[101]	[31,28,5,3]
6	[110]	[31,25,14,6]
7	[111]	[31,30,15,1]

The equation for the tap is given by:

$$Tap = [bit(23), bit(10), bit(2)], \tag{16}$$

$$Tap = LFSR\_IN\,[23, 10, 2]\,. \tag{17}$$

Hence, if for example $LFSR\_IN[23, 10, 2] = 101$, then the tap values will be [31,28,5,3]. This is very useful to prevent picking a random tap with a very low cycle and run in a loop if the cycle is less that TR. The design for this part is shown in figure 3.4.

3.2. Rapid Prototyping using Field Programmable Gate Array (FPGA)

For the implementation of the algorithm the hardware description language Verilog is used. Verilog is a well known hardware description language used in the industry. The Altera Quartus II 9.0 software suite is used for the simulation and synthesis of the design. In Fig. 3.5 a design flow of the implementation is presented.

The design was separated into three distinct modules. The first module was the XOR handler. The second is the LFSR. The third is the controller. These modules are connected to generate 32-bit random bits using the cells as inputs. A memory element was made in the XOR handler to store all of the values from the cells. The memory was initialized before the time of simulation.

The main job of the XOR handler is to XOR a set of sixteen 32-bits word from the cells and output to the controller. The XOR handler has two inputs and two outputs. The two inputs are the clock and the signal called got_it. The clock is a global clock that has been set to 50 MHz. The other signal is a control signal that the controller has. This signal is used to tell the XOR handler that the controller already read the bits. After got_it is set to high, the XOR handler will generate another value. The XOR handler also has 2 outputs: the first one is a rdy_snd signal and a 32-bit word called bitsout. When the XOR handler is ready to send the 32-bits it generated it will raise the rdy_snd signal to 1.

The LFSR is an implementation of a regular LFSR but it also allows for change of tap. The LFSR has 4 inputs and only one output. The first input is the clock which is the global clock. The second input is reset. Every time reset is set to one (by the controller) it

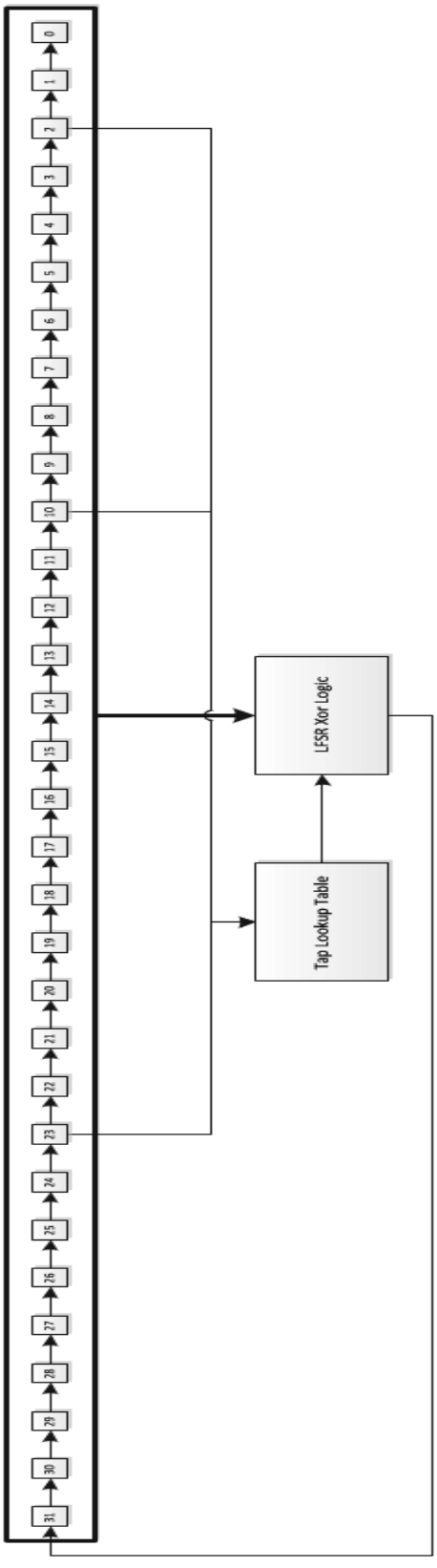

FIGURE 3.4. Design of the LFSR with lookup table and XOR logic.

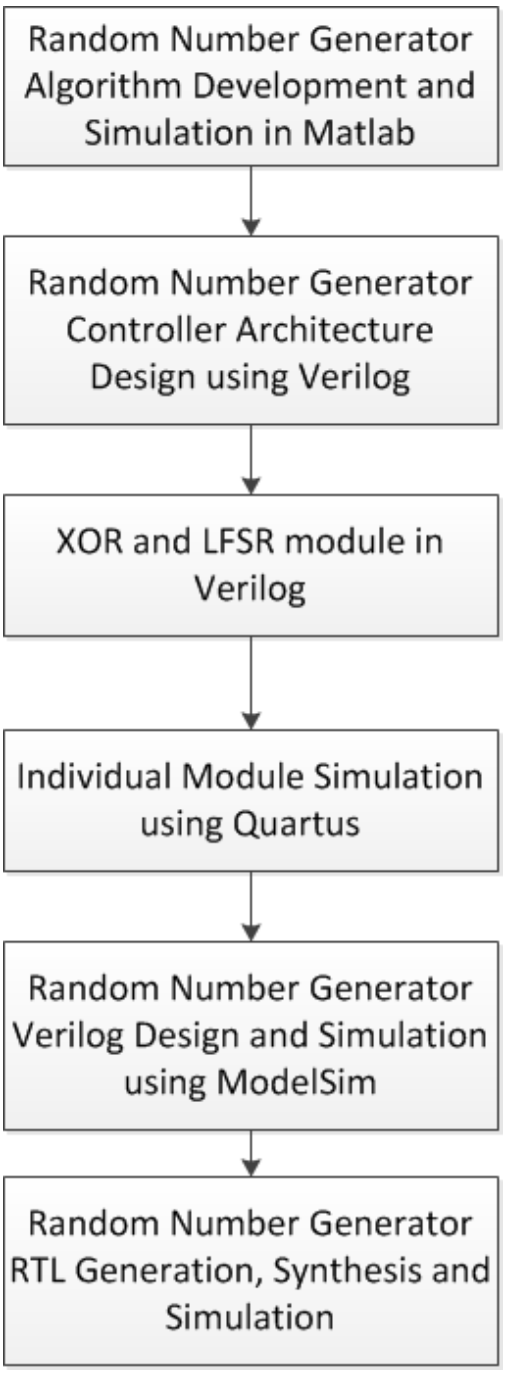

FIGURE 3.5. Design flow of FPGA RNG implementation

will change the tap and the seed of the LFSR and start producing values. The third input is the init-tap. This input is what specifies the tap when the module is reset. The fourth input is init (or seed). This is what contains the seed when the module is reset. The output of the LFSR is just a 32-bit word that is updated every clock cycle.

The controller module, as the name implies, controls the XOR handler. The LFSR produces the random bits. This module has 4 inputs and 5 outputs as shown in Table 3.4.

TABLE 3.4. Pin assignment of the controller module of the design.

Pin	Direction	Description
clock	Input	This is the 50 MHz global clock.
rdy_snd	Input	This is the signal that lets the controller know if XOR Handler is ready to send a new value.
bitsin	Input	This is the 32-bit word that is sent by XOR Handler.
LFSR_in	Input	This is the 32-bit word that is sent by the LFSR.
LFSR_reset	Output	This is the pin to Reset the LFSR to change the TAP and the SEED.
got_it	Output	This is the pin to tell XOR Handle that Controller received the 32-bit word.
LFSR_seed	Output	This is the seed that is sent to the LFSR when the LFSR is reset.
LFST_Tap	Output	This is the TAP that is sent to the LFSR when the LFSR is reset.
bitsout	Output	This is the final bits that will be output.

3.3. Simulation of the FPGA Prototype

After the implementation of the FPGA design, it is tested to ensure the design was correct. The design was tested by simulating the FPGA design. The results are compared with the results of the MATLAB implementation. The MATLAB implementation's results were tested by using common statistical testing suite.

The same inputs were given to both the MATLAB implementation and the FPGA design. It is expected to get the same values in both provided the same inputs are given. From Table 3.5 it can be seen that the first 20 outputs of the two design are the same. The comparison test was performed over 1 million 32 bit values. However, for brevity, only 20 of them are shown. These numbers are decimal representation of the 32 bit values calculated in the design.

3.4. RTL Synthesis from FPGA

After writing the synthesizable Verilog code in Quartus, an RTL version of the design is obtained. The top view shows the three modules connected as explained in the last section. In Figure 3.6, the RTL schematic view of the FPGA implementation is presented. In Figure 3.7, the RTL view of the XOR module is presented. In Figure 3.8, the RTL view of the LFSR module is shown. In Figure 3.9, the RTL view of the controller module is provided.

From FPGA synthesis results the resource utilization data are obtained. The resource usage of the FPGA implementation is given in Table 3.6. It is evident from the table that the design does not require too many resource and will be simple to add to existing designs.

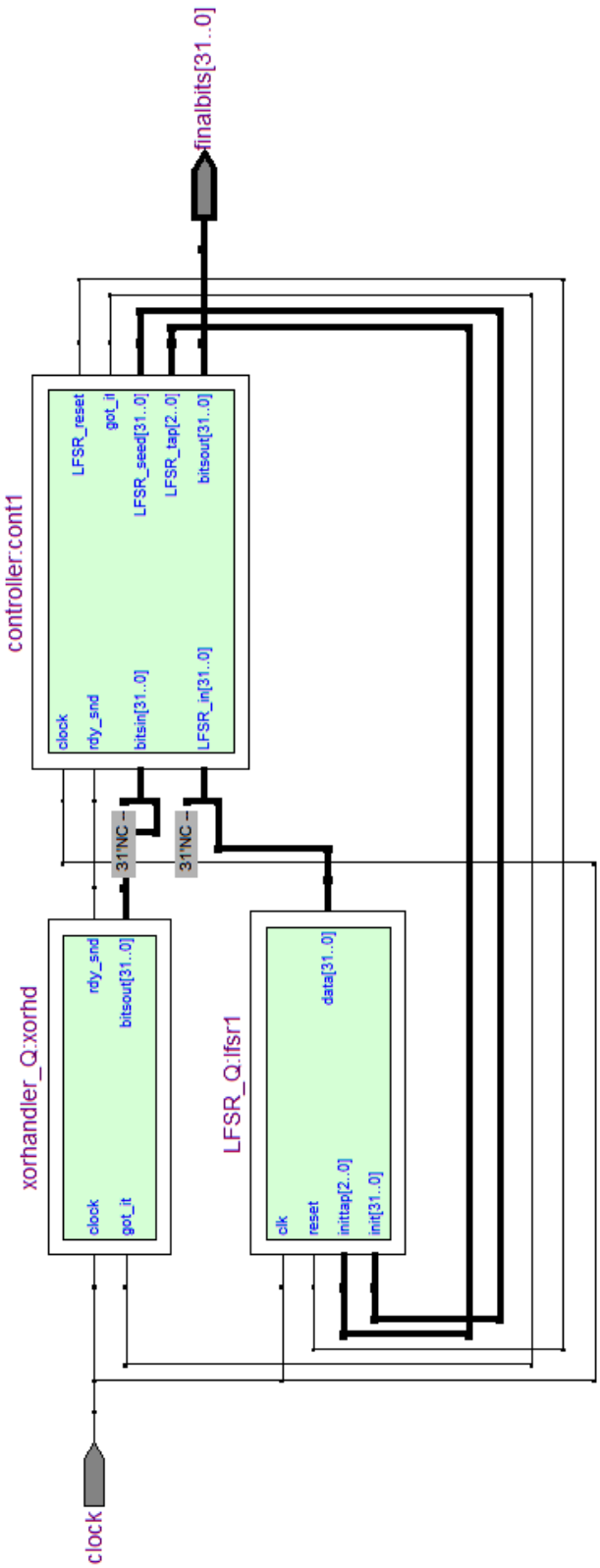

FIGURE 3.6. Top level view of design using Quartus II RTL viewer.

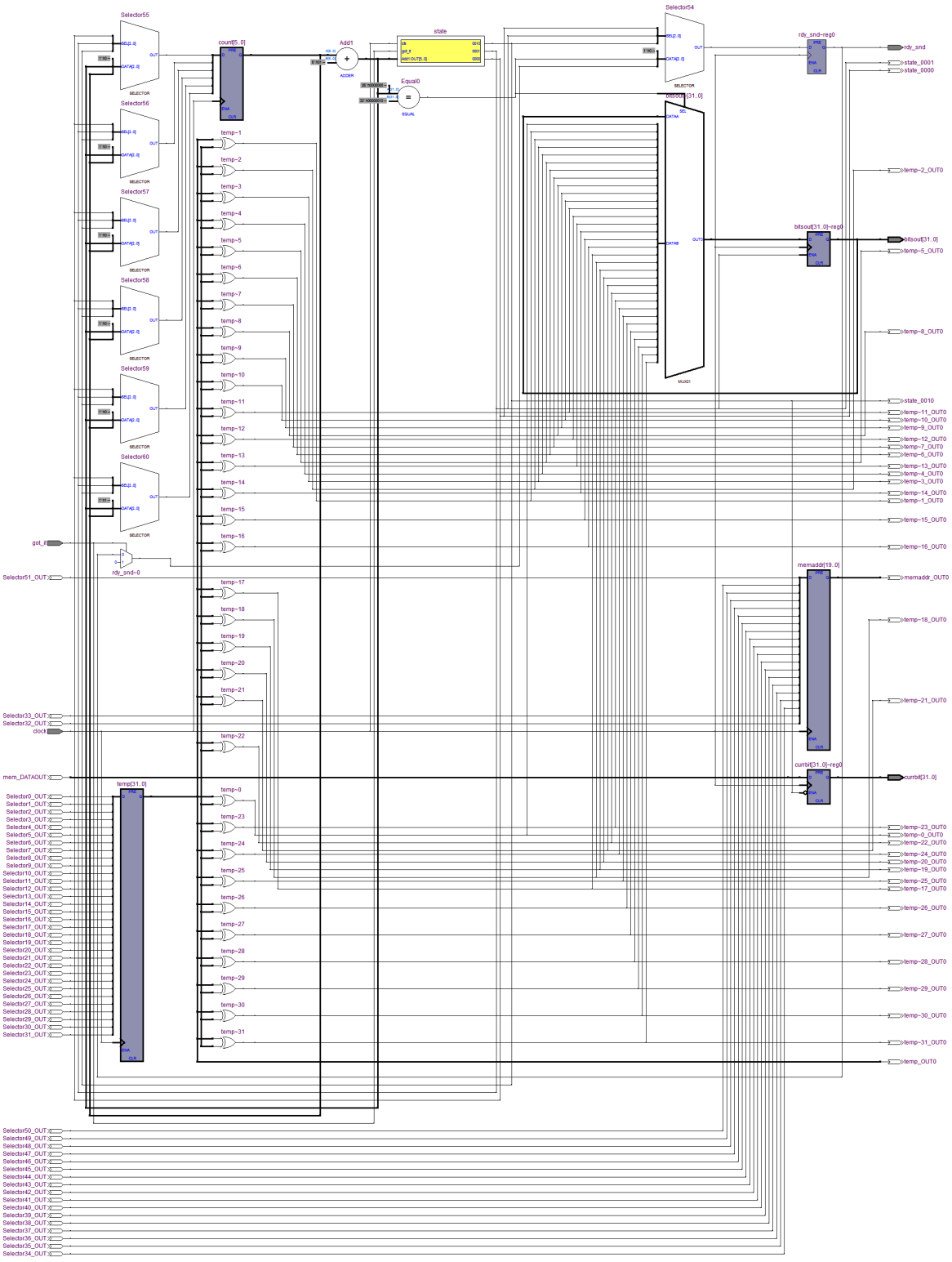

FIGURE 3.7. XOR handler of design using Quartus II RTL viewer.

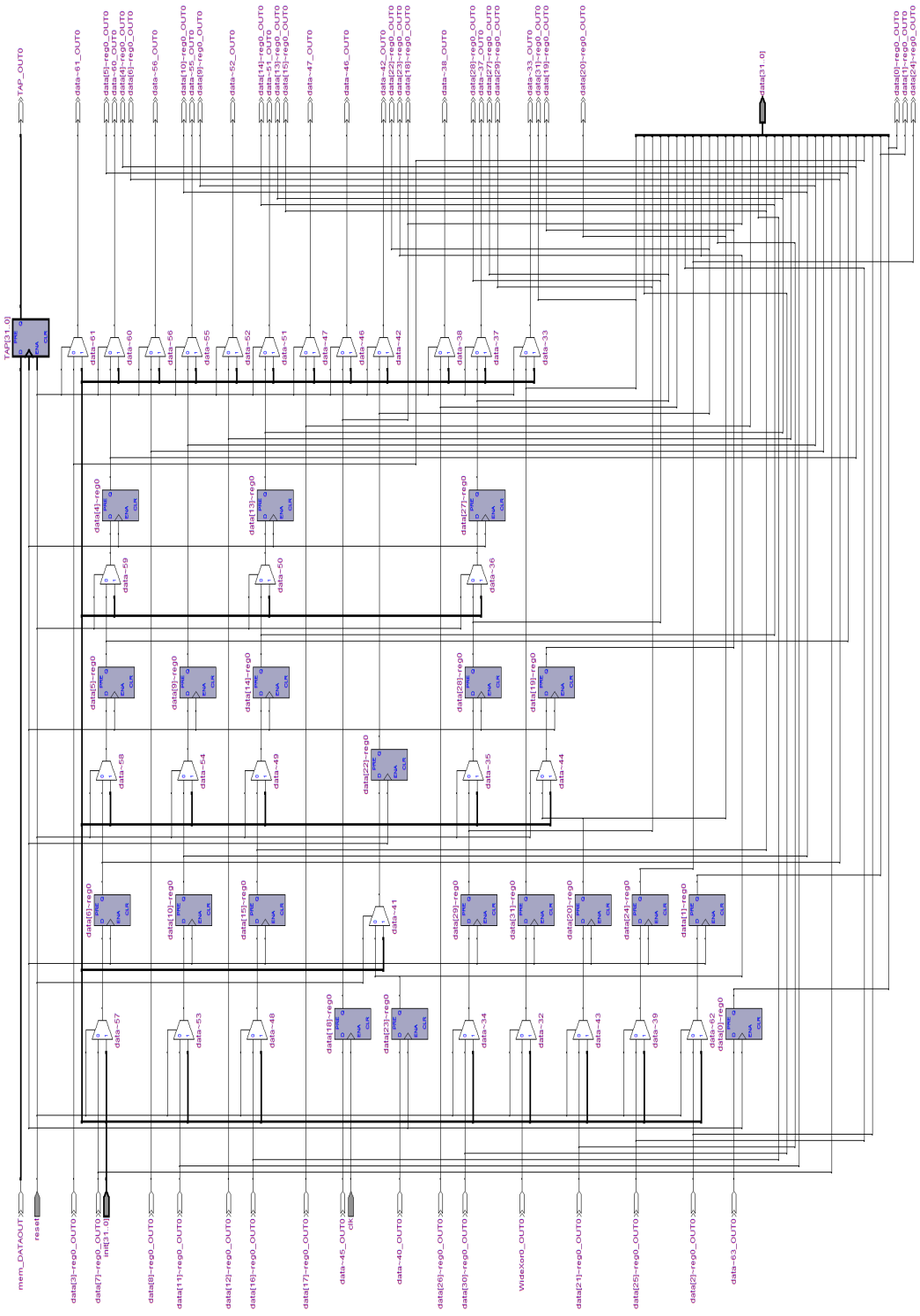

FIGURE 3.8. LFSR module of design using Quartus II RTL viewer.

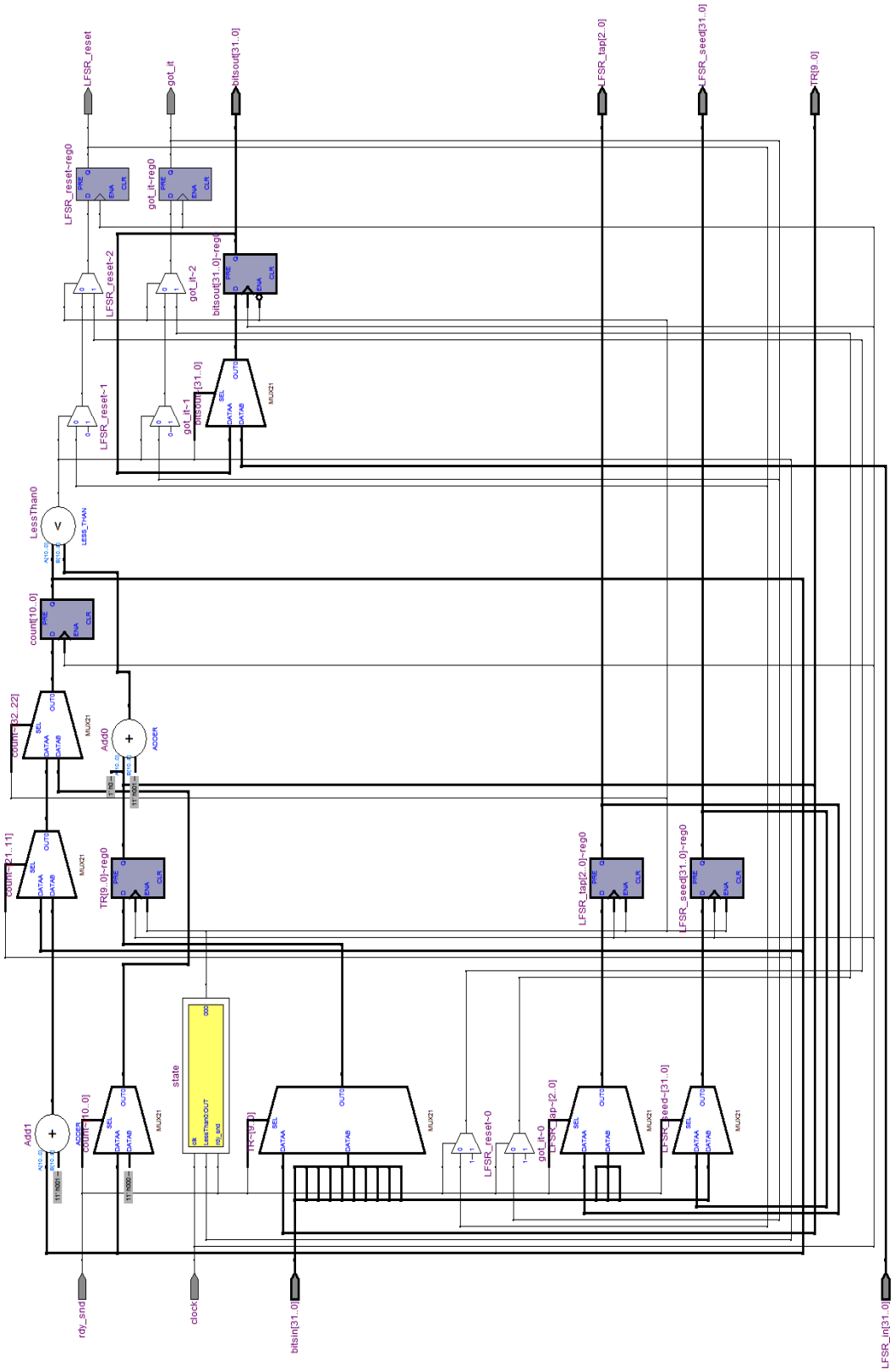

FIGURE 3.9. Controller module of design using Quartus II RTL viewer.

TABLE 3.5. Comparison of the MATLAB and FPGA results.

MATLAB (32 bit in decimal)	FPGA (32 bit in decimal)	Equal?
3869298507	3869298507	YES
1934649253	1934649253	YES
967324626	967324626	YES
2631145961	2631145961	YES
1315572980	1315572980	YES
657786490	657786490	YES
328893245	328893245	YES
2311930270	2311930270	YES
1155965135	1155965135	YES
2725466215	2725466215	YES
1362733107	1362733107	YES
681366553	681366553	YES
2488166924	2488166924	YES
1244083462	1244083462	YES
2769525379	2769525379	YES
3532246337	3532246337	YES
1766123168	1766123168	YES
3030545232	3030545232	YES
3662756264	3662756264	YES
1831378132	1831378132	YES

TABLE 3.6. Resource usage of the FPGA implementation

Element Name	Number Used
Total Logic Elements	83
Dedicated Logic Registers	64
Total Pins	33
Total PLLS	0

CHAPTER 4

CIRCUIT DESIGN OF THE RANDOM NUMBER GENERATOR

In this chapter the cell used to generate the seeds is explained. A layout of the cell is also shown using state-of-the art computer-aided design (CAD) tools. First, a transistor level design of the cell is performed using Cadence Virtuoso with the Schematic drawing tool. Then a layout level design of the cell performed using Virtuoso is also presented.

4.1. The Design Flow for the Random Number Generator Circuit

The custom integrated implementation of the random number generator is now discussed. A modular design approach is followed to generate the layout of the complete chip in which the logic design is top-down and the physical design is bottom-up. The overall design flow is presented in Fig. 4.1. By using a hierarchical approach, the layout of various resources is created. Finally, once the complete chip layout is generated, parasitic extraction and power, area, and performance analysis can be performed on the post-layout silicon design.

The Virtuoso tool from Cadence was used to generate a transistor-level schematic of the random number generator. This schematic is generated using a 45nm technology library from Cadence. This library included a PMOS and NMOS which are used for the transistor-level schematic design and layout design. For SPICE simulations, models included in the process design kit (PDK) are used. Another alternative is the use of the predictive technology model (PTM). The design first starts with the schematic design of the cell. The next step is to simulate the schematic so that a tes tbench could be created using a clock and a voltage source to power the cell. Using the electronic design automation (EDA) tool to perform the actual simulation, it is possible to probe the wires and get the results in a wave form.

The physical design (layout or true representative of true silicon) is performed using the Cadence layout editor, Virtuoso. The layout editor allows to perform mapping of the design. After the layout is performed, then a Design Rule Check (DRC) is conducted to ensure the manufacturability of the design. The design is compared against the rules of

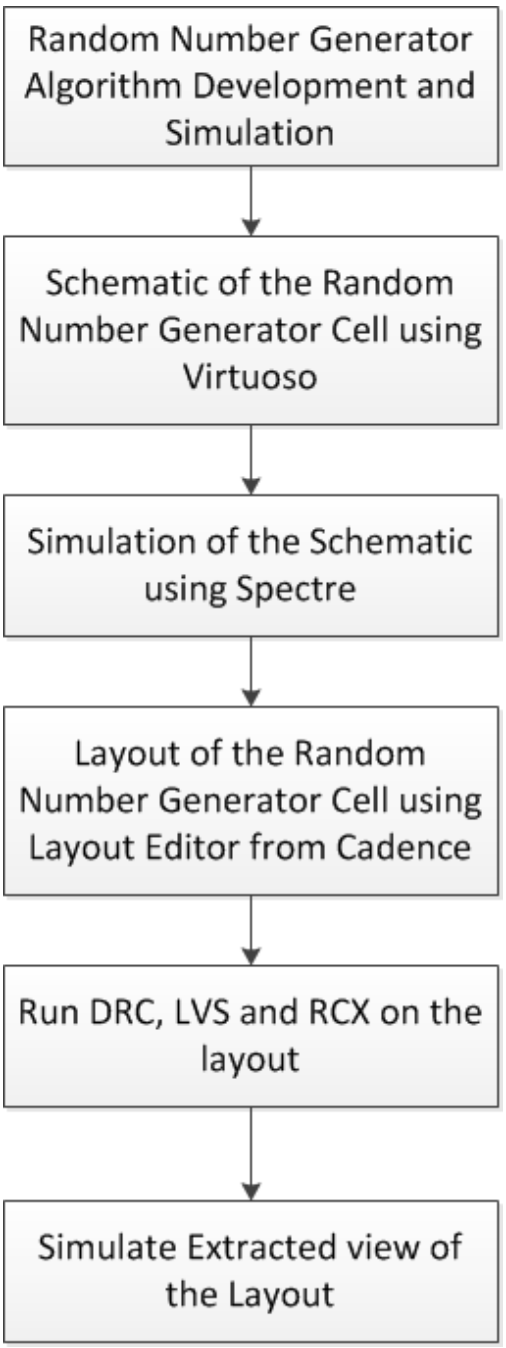

FIGURE 4.1. The design flow used for the physical design of the random number generator circuits.

the specific process technology to make sure there are not violations. Then the Layout vs. Schematic check is done. This phase compares the schematic design with the layout design to make sure they are alike. Then an RCX extraction is performed for the full-flown (RCLK) parasitic extraction of the circuit. This step changes the netlist by adding the followings

parasitic elements:

(1) parasitic resistors (R),

(2) parasitic capacitors (C),

(3) self-inductors (L),

(4) mutual inductance (K).

This ensures silicon accurate simulation of the design with all parasitic.

The last phase of the custom design process is to simulate the extracted view of the physical design. This is performed using EDA tools from Cadence as well. This simulation is performed in the same simulator as the schematic. This simulation took much longer to finish than the schematic simulation since the netlist is much bigger containing the following elements:

(1) active devices (i.e. transistors),

(2) parasitics from the active devices, and

(3) the parasitics from the interconnects (i.e. metal wires).

The parasitics due to active devices consist of RC elements. The parasitics due to the interconnect consist of RCLK elements. Thus the parasitics from interconnects are more severe than the active devices themselves for nanoscale circuits. The full-blown parasitic-aware netlist is therefore much more complex compared to schematic netlist.

4.2. Transistor Level Design

A single cell random generator cell circuit which was introduced by Intel was used [5]. The idea behind the design is to use something that is normally avoided, race conditions. Race conditions are defined as a flaw of electronic systems where the results may be altered or changed by a timing miscalculation. In other words, race conditions are unknown states of digital circuits. Race conditions cannot be predicted easily [5].

The way this circuit works is by connecting 2 inverters in a serial and circular connection. The two wires that connect the inverters are connected to a PMOS each on one side and V_{DD} on the other side of the PMOS. When the voltage in the gates goes to ground

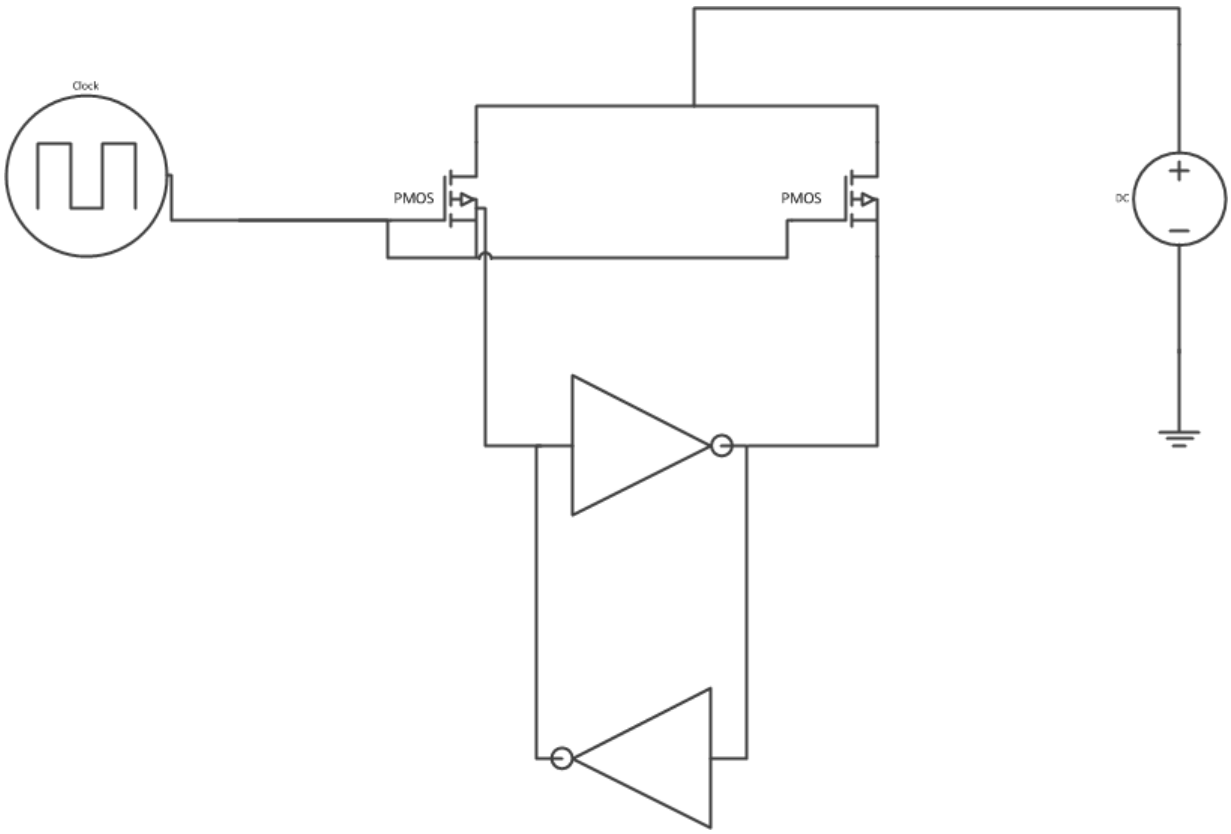

FIGURE 4.2. The mixed transistor and logic level view of the single cell random number generator design [8].

both sides of the inverters will become high or 1. Then when the values in the gate of the PMOS go high the inverter system will become unstable. Systems stabilize in nature and this system is not an exemption so one side will become logic 1 and the other logic 0. This is where the randomness of this design is introduced.

4.3. Layout Level Design

In this section the detailed discussion of the physical design of the random number generator circuit is provided. The design is implemented using a 45 nm CMOS technology process design kit (PDK) provided by Cadence.

The transistor-level schematic design of the random number generator cell is presented in Fig. 4.3. The physical design (or layout) of the random number generator cell is presented in Fig. 4.4. A comparison between the number of elements used in the schematic design and the number of elements used in the extracted view of the physical design is presented in

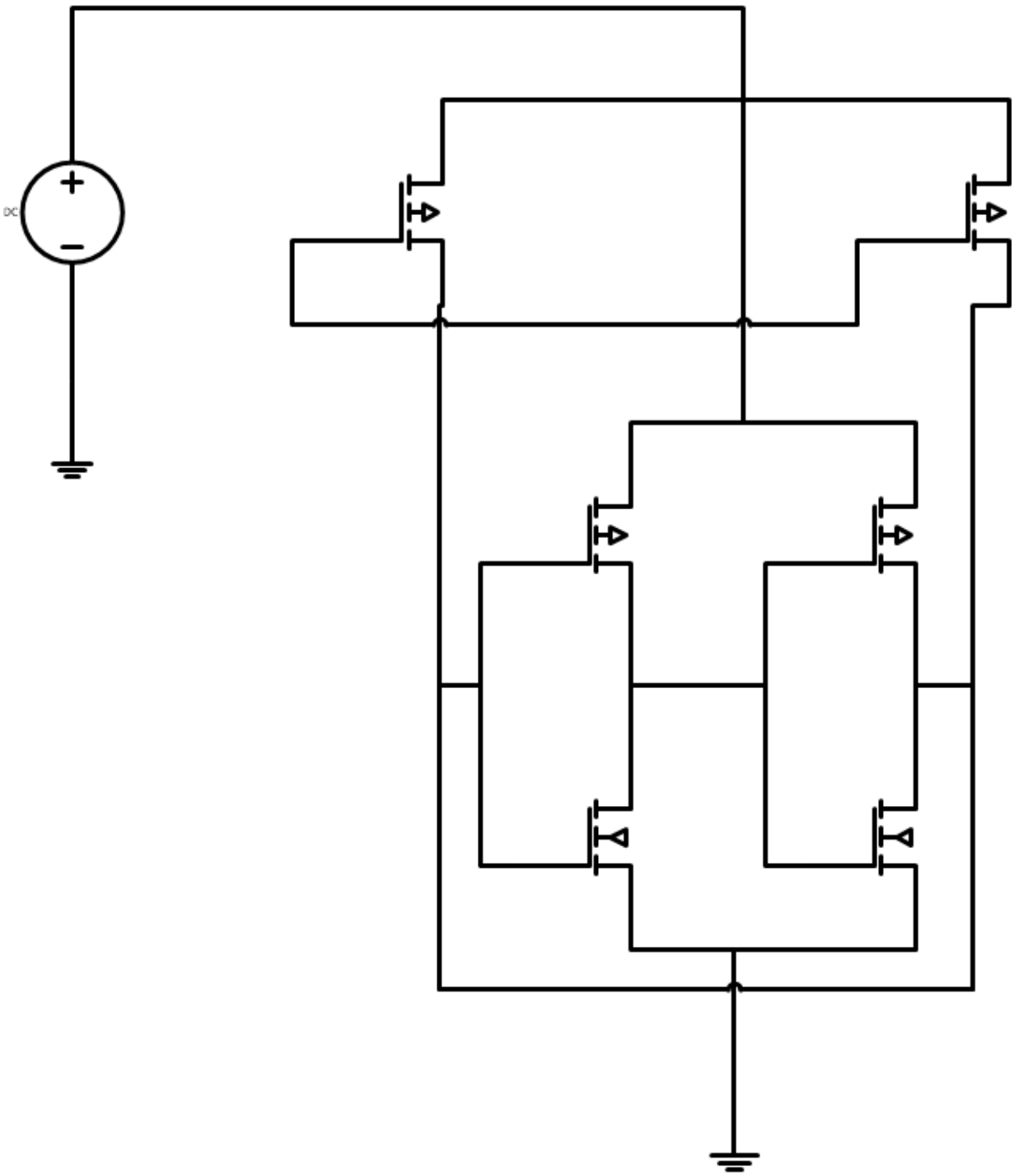

FIGURE 4.3. The transistor-level schematic design of the single cell random number generator design.

Table 4.1. The number of circuit elements give a broad perspective to compare the increase in the simulation time of the schematic and layout design of the circuit.

TABLE 4.1. Number of elements in the schematic design versus the physical design after RCX extraction.

Circuit Information	Schematic	Layout After RCX
Number Of Elements in Netlist	6	24
Number of Cell types used	5	13

Once the design is performed, rigorous tests are necessary to ensure that the design is robust and withstands all security related attacks. After running randomness tests such as die hard and runstest in MATLAB, it is observed that the RND cells alone do not pass the tests for their use with cryptographic applications. There is a need for using extra circuitry to make them pass the randomness tests [5]. The design with additional circuitry works because the numbers of ones and zeros being produce are generated in a random like sequence.

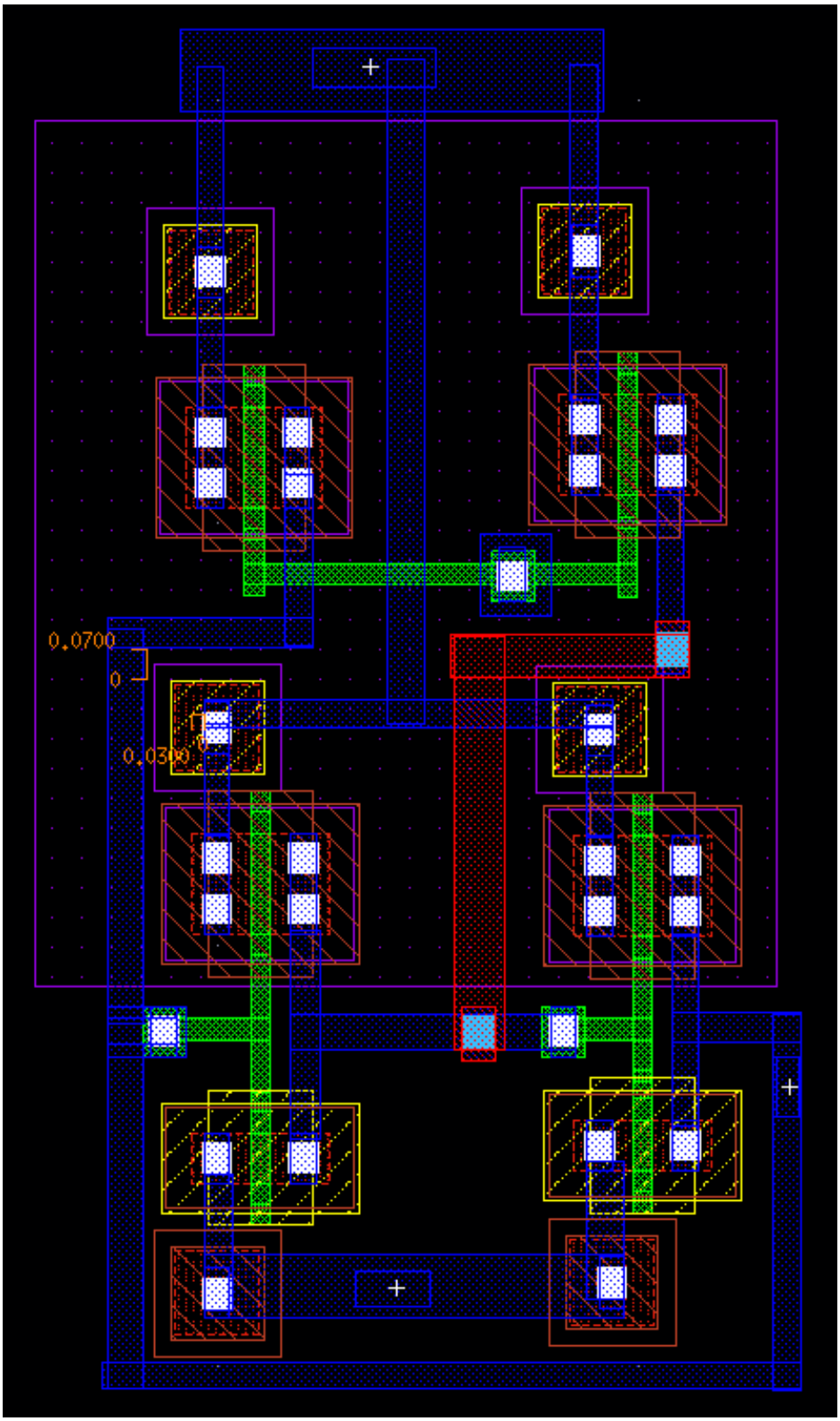

FIGURE 4.4. The physical design of the single cell random number generator circuit.

CHAPTER 5

EXPERIMENTAL EVALUATIONS

In this chapter the experiments and evaluation used to test the proposed random number generator design are explained. The results of these evaluations are presented along with their interpretation. First the experiments of the Random Number Generator Cell are introduced. Then the experiments of the MATLAB implementation of the design are discussed. Finally, the experiments and results of the FPGA design are presented.

5.1. Experiments of the Random Number Cell

After making the design of the random number generator cell in virtuoso, the Spectre simulator is used to perform the simulations. Spectre is an analog simulator that is used to simulate a large number of devices like transistors, resistors, capacitors and inductors. Spectre is used as the design of the actual cell was made in Virtuoso, another cadence tool to draw Schematic and layout of designs. The simulation waveforms for a single cell are presented in Fig. 5.1. The simulation waveforms are presented in enhanced fashion in Fig. 5.2. As can be observed from Fig. 5.1, when Vsource goes logic high the value from nodeA will randomly go to logic high or logic low. This demonstrates that the random number generator cell circuit works.

The simulation time for the cell was very complicated and not trivial at all. Since a large number of bits were needed to use the randomness benchmarks, the simulation time was very long. A total of 80 million bits were needed to use the DIE HARD benchmark. The design of one cell has a clock cycle time of 20 ns. So there is a needed for simulation covering $20 * 80 * 10^6$ ns. The time for one simulation is 1.6 seconds. To simulate this using only one cell will take about a week. This is despite of the availability of the fast processors including multi core processors. Some of the steps of the simulation, for example model evaluation are not simply parallelizable. To reduce the amount to time to simulate the design 16 different simulations were being executed at the same time on the server. This means that each simulation was using one core. Now each simulation only have to run for 100 ms which it is

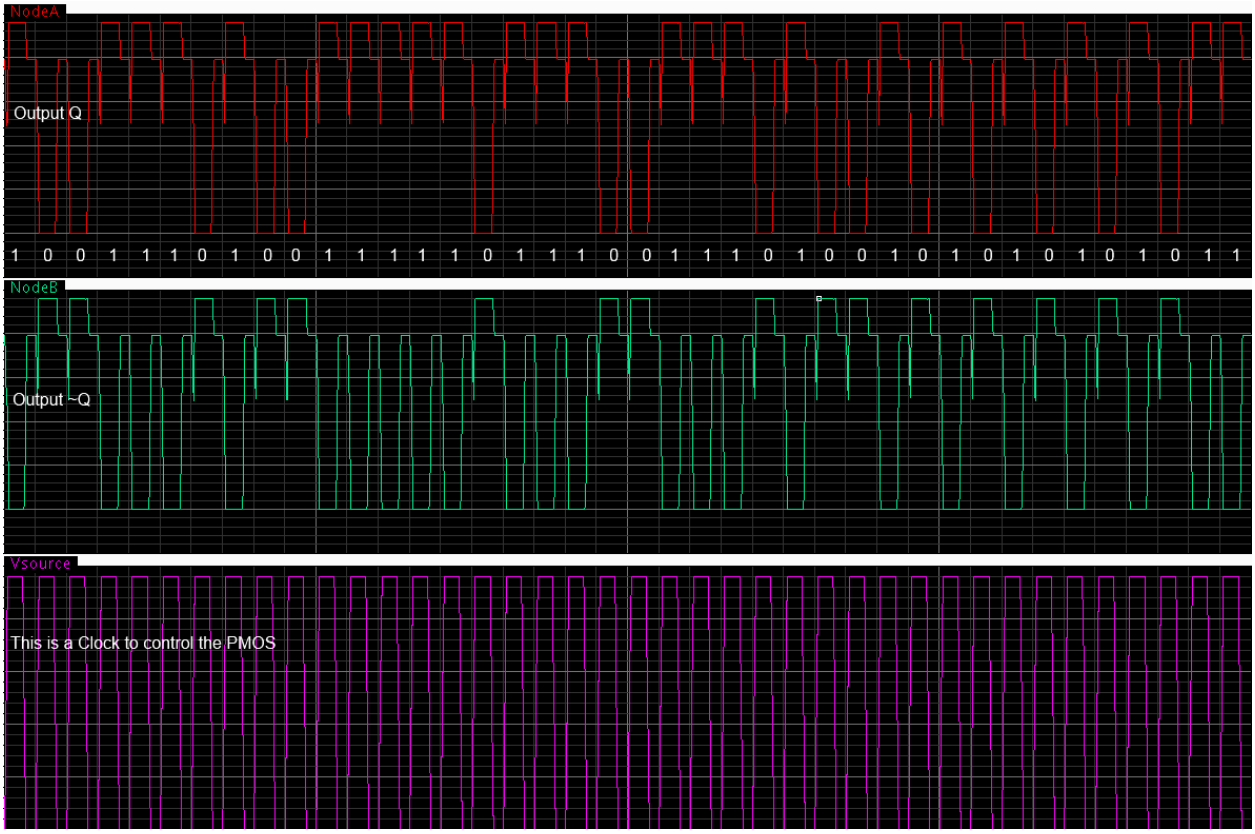

FIGURE 5.1. Simulation of a random number generator cell of the design using Spectre.

a big improvement. This reduces the time needed for the simulation to about 10 hours.

The '10011101001111101110011' sequence generated during the simulation is presented in Fig. 5.2. This is a random like sequence as can be seen from the waveforms. To prove statistically that the numbers are random like is achieved by generating many bits and checking how many ones and zeros are obtained. After running the simulation for 80 Mbits it is observed that there are about the same number of ones and zeros. This is a strong indicator that the random number generator design works. However, the problem is that this is not sufficient information to prove the numbers are robust. After running the results in the DIEHARD benchmark, it can be seen that the results are not really random.

This is evident from the discussion in [5] that the cells are not meant to be used on their own for random numbers. There was extra logic that needed to implemented. The cells are like unpredictable-hard-to-hack sources to use with known pseudorandom number

FIGURE 5.2. Zoomed view of the simulationof a random number generator cell of the design using Spectre.

generators. This is why the data obtained from the cells did not pass the tests.

5.2. Experiments of the 32-bit Random Number Generator using CMOS Circuit and MATLAB

After showing the design and implementation of the Random Number Generator, the experiments conducted on the data are elaborated. The state-of-the art approach to test if a set of given numbers or values is random is by using a randomness test benchmark. These benchmarks calculate the probability of the set of numbers to be random.

In this section the results of the random bit cell are discussed. The cells alone do not pass the tests. This is because the CMOS design and manufacturing reveals information

because of switching. This will make the probability of the output being one or zero not be 50-50. Also, even if the two transistors were in fact 'perfect' the natural characteristics of transistors in a circuit are not exactly equal. A good example of this is the difference of temperature in different parts of a chip.

The additional logic after the individual cells ensures to make the resulting data more random like. Several tests are performed using a tool from Computer Security Division in the Computer Security Resource Center of the National Institute of Standards and Technology (NIST). The tool is called Statistical Test Suite (STS). The main goals of this tool are the following:

(1) Develop a set of tools for statistical tests to detect non-randomness in a sequence of numbers and determine if these numbers could be used for cryptographic applications.
(2) Document and provide software tools for these tests [20].
(3) Provide sample data and tests using the tool [7].

STS will give results to all of the different randomness tests. A list of these tests are the following:

(1) Frequency: The emphasis of the test is the amount of zeroes and ones for the entire sequence. The purpose of this test is to determine whether the number of ones and zeros in a sequence are about the same as would be expected for a truly random sequence. The test assesses the closeness of the fraction of ones to 1/2. In other words, the number of ones and zeroes in a sequence should be about the same. All of the other tests depend on this tests passing.
(2) BlockFrequency: The emphasis of the test is the distribution of ones within M-bit blocks. The purpose of this test is to find out whether the frequency of ones in an M-bit block is about $M/2$, as would be anticipated under a hypothesis of randomness. For block size $M = 1$, this test deteriorates to test 1, the Frequency (Monobit) test.
(3) CumulativeSums: The emphasis of this test is the best excursion (from zero) of the

random walk that is defined by the increasing sum of adjusted (-1, +1) digits in the sequence. The resolution of the test is to determine whether the cumulative sum of the partial sequences occurring in the tested set of random numbers is too large or too small relative to the behavior supposed to be obtained from that cumulative sum of random sequences. This cumulative sum may be considered as a random walk. For a random sequence, the excursions of the random walk should be near zero. For some types of non-random sequences, the excursions of the random walk from zero will be large.

(4) Runs: The emphasis of this test is the total amount of runs in the sequence, where a run is an continuous sequence of equal bits. A run of size k consists of precisely k equal bits and is bounded before and after with a bit of the differing value. The purpose of the runs test is to find out whether the amount of runs of ones and zeros of several lengths is as likely for a random sequence. In other words, this test finds out whether the fluctuation between such zeros and ones is too fast or too slow.

(5) LongestRun: The emphasis of the test is the longest run of ones in M-bit blocks. The reason for this test is to determine whether the size of the longest run of ones within the tried sequence is constant with the size of the longest run of ones that would be estimated in a random sequence. Remember that an irregularity in the expected size of the longest run of ones infers that there is also an irregularity in the expected size of the biggest run of zeroes. So only a test for ones is necessary.

(6) Rank: The emphasis of the test is the rank of disjoint sub-matrices of the whole sequence. The purpose of this test is to look for linear dependence between fixed size substrings of the original sequence.

(7) Fast Fourier Transform (FFT): The emphasis of this test is the top heights in the Discrete Fourier Transform (DFT) of the sequence. The purpose of this test is to find periodic structures (i.e., repetitive patterns that are close each other) in the verified sequence that would show a deviation from the hypothesis of randomness.

(8) NonOverlappingTemplate: The emphasis of this test is the number of incidences of

pre-specified target strings. The purpose of this test is to perceive generators that make too many incidences of a given non-periodic (aperiodic) pattern. For this test an m-bit window is used to examine for a exact m-bit pattern. If the pattern is not found, the window shifts one bit position. If the pattern is found, the window is set to the bit after the found pattern, and the search resumes.

(9) OverlappingTemplate: The emphasis of the Overlapping Template Matching test is the amount of incidences of pre-specified target strings. This test uses an m-bit spam to search for a specific m-bit pattern. If the pattern is not found, the window slides one bit location. The difference among this test and NonOverlapping Template is that when the pattern is found, the window shifts only one bit before resuming the search.

(10) Universal: The emphasis of this test is the number of bits among matching patterns. This is a measure that is related to the size of a compressed sequence. The purpose of the test is to sense whether or not the sequence can be compressed without loss of information. A compressible sequence is determined to be non-random.

(11) ApproximateEntropy: As with the serial test, the focus of this test is the rate of all possible overlapping m-bit patterns through the whole sequence. The purpose of the test is to compare the frequency of overlapping blocks of two sequential/together lengths (m and $m + 1$) against the expected result for a random sequence.

(12) RandomExcursions: The emphasis of this test is the amount of cycles taking exactly k visits in an increasing sum random walk. The increasing sum random walk is derived from part sums after the (0,1) sequence is transferred to the appropriate (-1, +1) sequence. A cycle of a random walk consists of a series of steps of unit length taken at random that start at and return to the origin. The purpose of this test is to find out if the number of calls to a specific state within a cycle deviates from what one would expect for a random sequence. This test is actually a series of eight tests (and conclusions), one test and conclusion for each of the states: -4, -3, -2, -1 and +1, +2, +3, +4.

(13) RandomExcursionsVariant: The emphasis of this test is the whole amount of times that a specific state is visited (i.e., occurs) in a increasing sum random walk. The purpose of this test is to notice deviations from the expected number of visits to various states in the random walk. This test is actually a series of eighteen tests (and conclusions), one test and conclusion for each of the states: -9, -8, -1 and +1, +2, +9.

(14) Serial: The emphasis of this test is the frequency of all likely overlapping m-bit patterns across the whole sequence. The purpose of this test is to find out whether the number of occurrences of the 2^m m-bit overlapping patterns is about the same as would be expected for a random sequence. Random sequences have homogeneousness. In other words, every m-bit pattern has the same chance of showing up as every other m-bit pattern.

(15) LinearComplexity: The emphasis of this test is the size of a linear feedback shift register (LFSR). The purpose of this test is to find out whether or not the sequence is complex enough to be considered random. Random sequences are categorized by longer LFSRs. An LFSR that is too small suggests non-randomness.

Input parameters from [9] were used to test the number sequence. The results of various tests conducted on the generated random numbers are presented in Table 5.1. The results are shown for various tests and different input sizes. The values of the parameters used during the experiments are also presented in the same Table.

Table 5.1: The test parameters, input sizes, and test names for studying the randomness of the number generated from the random number generator [9].

Test Name	**Input Size**	**Other Parameters**
Frequency	100	
BlockFrequency	2000	M=20, N=10

CumulativeSums	100	
Runs	100	
LongestRun	6272	M=128
Rank	38912	
FFT	1024	
NonOverlappingTemplate	1048576	m=9, B=111111111
OverlappingTemplate	998976	m=9, M=1032, N=96
Universal	387840	L=6, Q=64
ApproximateEntropy	500	m=5, n=500
RandomExcursions	1000000	
RandomExcursionsVariant	1000000	
Serial	500	m=5, n=500
LinearComplexity	1000000	M=20, N=100

To determine a passing score of a test, the P-value must be between 0.0001 and 0.9999 [28]. In that case, the random number data test passes with a 95% confidence. The test results are presented in Table 5.2. As can be observed from in the table, the random number generator passed all of the tests performed using the benchmarks from NIST. This shows that this random number generator technique can be used for encryption, watermarking, and other security application needs.

Table 5.2: The test results of the overall random number generator for a 32-bit design.

Test Name	**P-value**	**Pass-Fail**
Frequency	0.350485	Pass
BlockFrequency	0.213309	Pass

CumulativeSums	0.739918	Pass
Runs	0.534146	Pass
LongestRun	0.122325	Pass
Rank	0.288341	Pass
FFT	0.534146	Pass
NonOverlappingTemplate	0.213309	Pass
OverlappingTemplate	0.834499	Pass
Universal	0.739918	Pass
ApproximateEntropy	0.122325	Pass
RandomExcursions	0.714438	Pass
RandomExcursionsVariant	0.668627	Pass
Serial	0.739918	Pass
LinearComplexity	0.311800	Pass

5.3. Experiments of FPGA Implementation

In this section the experiments and results of the FPGA implementation of the random number generator are discussed.

The Verilog model of the random number generator architecture is the top-level design of the implementation. It is a test-bench made for ModelSim. The waveform of the simulation which was performed using ModelSim is presented in Fig. 5.3. The last two port maps are for saving the final bits to a file. This allows us to compare the output of the Verilog design with the output of the MATLAB design. The Verilog model was tested for correctness by comparing the result from MATLAB with the results from Verilog. As expected the two results were identical. This demonstrates that the Verilog design is identical to the MATLAB based behavioral simulation of the random number generator.

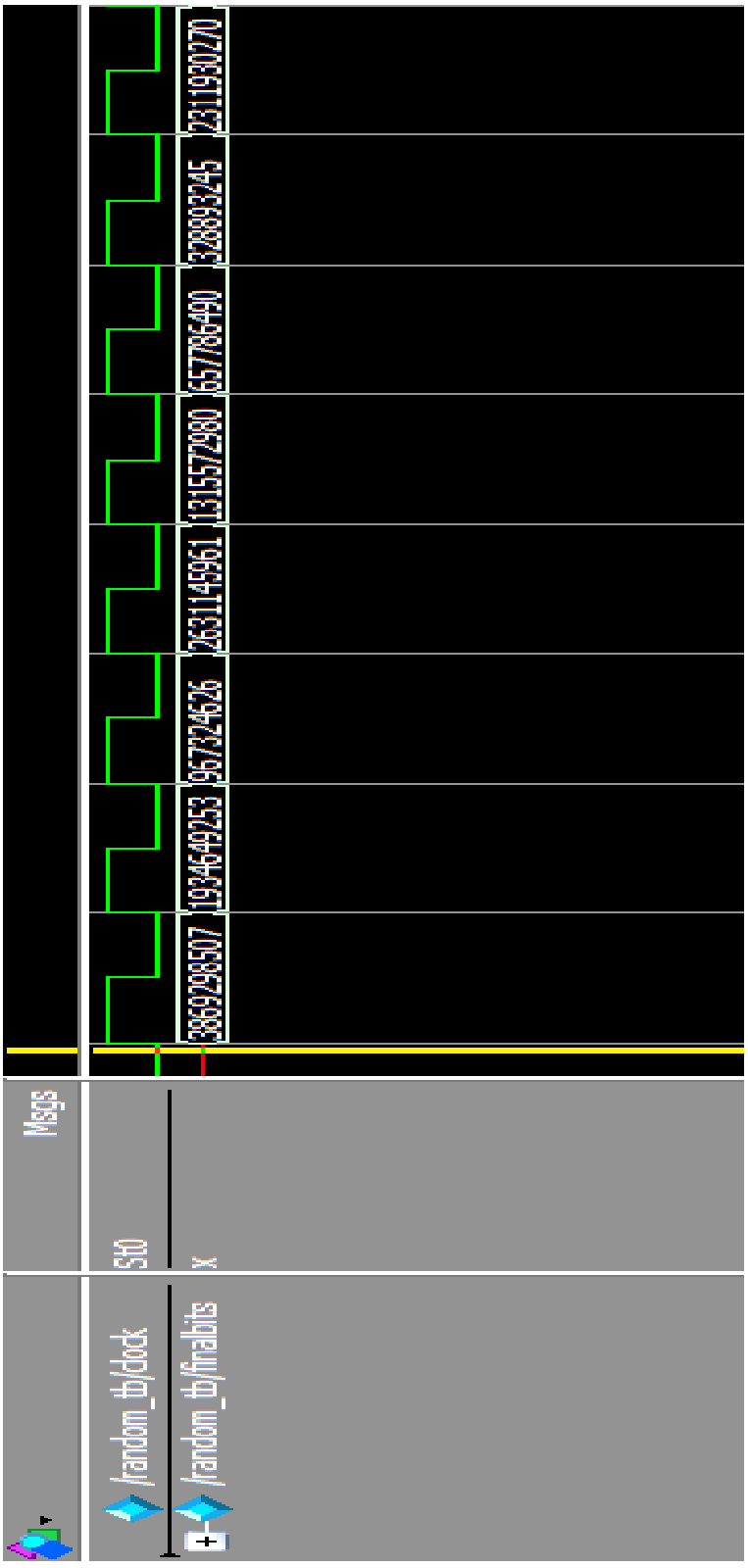

FIGURE 5.3. Simulation waveform from ModelSim

CHAPTER 6

CONCLUSION AND FUTURE RESEARCH

In this thesis, the schematic design and 45nm CMOS based layout design of a random number generator cell were performed. Unfortunately this design will not pass the randomness tests for cryptography. Additional logic is needed to increase distribution and randomness on the numbers. This circuit generates the initial sequence for the 32-bit random number generator.

The design of the random number generator using CMOS circuit and MATLAB was able the to pass the randomness tests. This was able to show that the logic done in MATLAB using the results from the random number generator cell was conditioning the bias bits and making them pass the randomness tests.

The design of the random number generator in Verilog was a necessary step to show that the implementation in MATLAB was feasible and practical. With the RTL design of the random number generator it was proved that the design is feasible. As the results from MATLAB and Verilog were identical given the same inputs, the thesis confirms that both of the implementations were identical.

In this current research the thesis showed the effectiveness of the design. A more involved approach is the overall layout implementation with tests. This will bring many challenges to the designers because of time and closure in real hardware. Also the combination of the cell with the layout of the implementation will allow to perform a more rigorous testing. An actual silicon chip will allow to fully test and confirm that this chip will produce strong, true random numbers.

BIBLIOGRAPHY

[1] A.A. Chowdhury, L. Bertling, B.P. Glover, and G.E. Haringa, *A monte carlo simulation model for multi-area generation reliability evaluation*, Probabilistic Methods Applied to Power Systems, 2006. PMAPS 2006. International Conference on, june 2006, pp. 1 –10.

[2] Viktor Fischer and Milos Drutarovsky, *True random number generator embedded in reconfigurable hardware*, Proc. of CHES 2002 (2003), 415–430.

[3] W.A. Fuller, *Sampling statistics*, Wiley series in survey methodology, Wiley, 2009.

[4] GaoFei, *Generalized chaotic binary sequence generation method*, Computer Engineering (2007).

[5] George Cox Greg Taylor, *Behind intel's new random-number generator*, IEEE Spectrum (2011).

[6] F. Hartung and F. Ramme, *Digital rights management and watermarking of multimedia content for m-commerce applications*, Communications Magazine, IEEE 38 (2000), no. 11, 78 – 84.

[7] Lawrence E Bassham II, *A statistical test for random and pseudorandom number generators for cryptographic applications*, National Institute of Standards and Technology, revision 1a ed., April 2010.

[8] Benjamin Jun and Paul Kocher, *The intel random number generator*, CRYPTOGRAPHY RESEARCH, INC. WHITE PAPER PREPARED FOR INTEL CORPORATION (1999).

[9] Charmaine Kenny, *Random number generators: An evaluation and comparison of random.org and some commonly used generators*, Tech. report, Computer Science Department, TCD, April 2005.

[10] Kougianos, *A Comparative Study on Gate Leakage and Performance of High-κ Nano-CMOS Logic Gates*, Taylor & Francis International Journal of Electronics (IJE) 97 (2010), 985–1005.

[11] Elias Kougianos, Saraju P. Mohanty, and Rabi N. Mahapatra, *Hardware assisted watermarking for multimedia*, Comput. Electr. Eng. 35 (2009), 339–358.

[12] Sammy H. M. Kwok, *Fpga-based high-speed true random number generator for cryptographic applications*, IEEE- Pokfulam Road (2006).

[13] S. P. Mohanty, D. Ghai, E. Kougianos, and P. Patra, *A Combined Packet Classifier and Scheduler Towards Net-Centric Multimedia Processor Design*, Proceedings of the 27th IEEE International Conference on Consumer Electronics (ICCE), 2009, pp. 11–12.

[14] S. P. Mohanty, K. R. Ramakrishnan, and M. S. Kanakanhalli, *An Adaptive DCT Domain Visible Watermarking Technique for Protection of Publicly Available Images*, Proceedings of the International Conference on Multimedia Processing and Systems (ICMPS), 2000, pp. 195–198.

[15] Saraju P. Mohanty, *A secure digital camera architecture for integrated real-time digital rights management*, J. Syst. Archit. 55 (2009), 468–480.

[16] Saraju P. Mohanty, *Uls: A dual-vth/high-k nano-cmos universal level shifter for system-level power management*, (2010).

[17] Saraju P. Mohanty, Renuka Kumara C., and Sridhara Nayak, *Fpga based implementation of an invisible-robust image watermarking encoder*, Lecture Notes in Computer Science, 2004, pp. 344–353.

[18] Saraju P. Mohanty, K. R. Ramakrishnan, and Mohan S. Kankanhalli, *A dct domain visible watermarking technique for images*, IEEE International Conference on Multimedia and Expo (II), 2000, pp. 1029–1032.

[19] S.P. Mohanty, R. Sheth, A. Pinto, and M. Chandy, *Cryptmark: A novel secure invisible watermarking technique for color images*, Consumer Electronics, 2007. ISCE 2007. IEEE International Symposium on, june 2007, pp. 1 –6.

[20] National Institute of Standards and Technology, *Random number generator*, "http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html", April 2008.

[21] J. Saarinen, *Vlsi implementation of tausworthe random number generator for parallel processing environment*, IEEE Proceedings-E 138 (1991), no. 3.

[22] N. Sklavos, P. Kitsos, K. Papadomanolakis, and O. Koufopavlou, *Random number generator architecture and vlsi implementation*, Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on, vol. 4, 2002, pp. IV–854 – IV–857 vol.4.

[23] Tatsuro Sugiura, *Demonstration of 30 gbit/s generation of superconductive true random number generator*, IEEE TRANSACTIONS ON APPLIED SUPERCONDUCTIVITY 21 (2011), no. 3, 843–846.

[24] N. M. Thamrin, *An enhanced hardware-based hybrid random number generator for cryptosystem*, International Conference on Information Management and Engineering (2009).

[25] Toshinari Takayanagi Vincent von Kaenel, *Dual true random number generators for cryptographic applications embedded on a 200 million device dual cpu soc*, Custom Intergrated Circuits Conference (CICC) (2007).

[26] GUAN Xiaohui, *The design of combined random number generator*, International Conference on Multimedia Information Networking and Security (2010).

[27] R.K. Youree, J.S. Yalowitz, A. Corder, and T.K. Ooi, *A multivariate statistical analysis technique for on-line fault prediction*, Prognostics and Health Management, 2008. PHM 2008. International Conference on, oct. 2008, pp. 1 –5.

[28] Guanglie Zhang, P.H.W. Leong, Dong-U Lee, J.D. Villasenor, R.C.C. Cheung, and W. Luk, *Ziggurat-based hardware gaussian random number generator*, Field Programmable Logic and Applications, 2005. International Conference on, aug. 2005, pp. 275 – 280.

[29] ZhangXuefeng, *Improved chaotic sequence generation method*, Computer Engineering and Design (2007).