FPGA BASED RANDOM NUMBER GENERATION FOR CRYPTOGRAPHIC APPLICATIONS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology

In

Electrical Engineering

By

SHATADAL MISHRA
ROLL NO -10602041
&
MRUTYUNJAY DAS
ROLL NO -10602063

Department of Electrical Engineering

National Institute of Technology

Rourkela

2010

FPGA BASED RANDOM NUMBER GENERATION FOR CRYPTOGRAPHIC APPLICATIONS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology

In

Electrical Engineering

By

SHATADAL MISHRA
ROLL NO -10602041
&
MRUTYUNJAY DAS
ROLL NO -10602063

Under the Guidance of
Prof. D. Patra

Department of Electrical Engineering

National Institute of Technology

Rourkela

2010

National Institute of Technology

Rourkela

CERTIFICATE

This is to certify that the thesis entitled **"FPGA based random number generation for cryptographic applications''** Submitted by **Shatadal Mishra, Roll No:10602041,** and **Mrutyunjay Das, Roll No. 10602063,** in the partial fulfillment of the requirement for the degree of **Bachelor of Technology** in **Electrical Engineering**, National Institute of Technology, Rourkela , is being carried out under my supervision.

To the best of my knowledge the matter embodied in the thesis has not been submitted to any other university/institute for the award of any degree or diploma.

Date:

Prof. D. Patra
Dept. of Electrical Engg.
National Institute of Technology
Rourkela - 769008

Acknowledgment

We avail this opportunity to extend our hearty indebtedness to our guide **Prof. D. Patra**, Electrical Engineering Department, for her valuable guidance, constant encouragement and kind help at different stages for the execution of this dissertation work.

Submitted by:

Shatadal Mishra
Roll No: 10602041
Electrical Engineering
National Institute of Technology
Rourkela

Mrutyunjay Das
Roll No: 10602063
Electrical Engineering
National Institute of Technology
Rourkela

CONTENTS

ABSTRACT

Random numbers are useful for a variety of purposes, such as generating data encryption keys, simulating and modeling complex phenomena and for selecting random samples from larger data sets. They have also been used aesthetically, for example in literature and music, and are of course ever popular for games and gambling. When discussing single numbers, a random number is one that is drawn from a set of possible values, each of which is equally probable, i.e., a uniform distribution. When discussing a sequence of random numbers, each number drawn must be statistically independent of the others.

Random numbers are generated by various methods. The two types of generators used for random number generation are pseudo random number generator (PRNG) and true random number generator (TRNG). The numbers generated are **random** because no polynomial – time algorithm can describe the relation amongst the different numbers of the sequence.

Numbers generated by true random number generator (TRNG) or cryptographically secure pseudo random number generator (CSPRNG). The sources of randomness in TRNG are physical phenomena like lightning, radioactive decay, thermal noise etc. The source of randomness in CSPRNG is the algorithm on which it is based.

In this project, the random numbers generated for cryptographic applications were generated by using the Blum Blum Shub generator, the CSPRBG. It was implemented on a FPGA platform using VHDL programming language and the simulation was done and tested on the Xilinx ISE 10.1i.

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1. INTRODUCTION

1.1 Early Work:

A random number generator is a computational device devised to generate a sequence of numbers or that lack any pattern. Hardware-based systems for random number generation are widely used, but often fall short of this target, though they may meet some of the statistical tests for randomness to confirm that they do not have any easily decipherable patterns. Methods for generating random results have existed since ancient times, including dice, coin flipping, the shuffling of playing cards, the use of yarrow stalks, and many other techniques. [1]

The earliest methods for generating random numbers — dice, coin, flipping, roulette wheels — are still used today, mainly in games and gambling as they are too slow for most applications in cryptography. [1], [2]

Various imaginative ways of collecting the entropic information for true random number generator have been devised. One technique is to run a hash function against a frame of a video stream from an unknown source.Lavarand used this technique with images of many lava lamps. HotBits tracks radioactive decay with GM tubes, while Random.org uses variations in the amplitude of atmospheric noise taped with a normal radio.[1]

One archaic way of producing random numbers was by a variation of the same machines used to select lottery numbers. Basically, these mixed numbered ping-pong balls with blown air, combined with mechanical agitation, use some method to withdraw balls from the mixing chamber (U.S. Patent 4,786,056). This method gives plausible results, but the random numbers generated by this means proved to be a costly affair. The method is slow, and is not fit to be used in most situations. [2]

RAND Corporation generated random digits with an "electronic roulette wheel", which consisted of a random frequency pulse source of about 100,000 pulses per second gated once per second with a constant frequency pulse and fed into a 5-bit binary counter. Douglas Aircraft manufactured the equipment, incorporating Cecil Hasting's suggestion for a noise source (behavior of the 6D4 miniature gas thyratron tube, when placed in a magnetic field). [2]

The Intel 80802 Firmware Hub chip included hardware RNG using two free running oscillators, one fast and one slow. A thermal noise source (noise from two diodes) is used to modulate the frequency of the slow oscillator, which then triggers a measurement of the fast oscillator. That output is then de-biased using a von Neumann type de-correlation step. The output rate of this device is less than 100,000 bit/s. This chip was an optional component of the 840 chipset family. [2]

The work on devising generators to generate random numbers for various purposes continues based on the requirements.

1.2 Motivation:

In today's world security is of prime importance and hence cryptography plays an important role in computer and networking security. Cryptographically secure random number generators are essential for this purpose. The versatility of this project in many fields urged and motivated us to select the same.

.

CHAPTER 2

FIELD PROGRAMMABLE GATE ARRAY

2.1 Introduction:

A **field-programmable gate array** (**FPGA**) is an integrated circuit created to be configured by the customer after manufacturing—hence "field-programmable". The FPGA configuration is generally defined using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) (circuit diagrams were previously used to specify the configuration, as they were for ASICs, but this is increasingly rare). FPGAs can be used to implement any logical function that an ASIC can perform. The ability to update the functionality after shipping, partial re-configuration of the portion of the design and the low non-recurring engineering costs relative to an ASIC design, offer advantages for many applications. [16]

FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "connected together"—somewhat like a one-chip programmable breadboard. Logic blocks can be configured to perform complex combinational functions, or merely simple logic like AND and NAND. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. [16]

2.2 History:

The FPGA industry burgeoned from programmable read-only memory (PROM) and programmable logic devices (PLDs). PROMs and PLDs both had the option of being programmed in batches in a factory or in the field (field programmable), however programmable logic was hard-wired between logic gates. In the late 1980s the Naval Surface Warfare Department sponsored an experiment proposed by Steve Casselman to develop a computer that would implement 600,000 reprogrammable gates. Casselman was successful and a patent related to the system was issued in 1992. Xilinx Co-Founders, Ross Freeman and Bernard Vonderschmitt, invented the first commercially viable field programmable gate array in 1985 – the XC2064. The XC2064 had programmable gates and programmable interconnections between gates, the beginnings of a new technology and market. The XC2064 boasted of mere 64 configurable logic blocks (CLBs), with two 3-input lookup tables (LUTs). [16]

2.3 Modern Progress:

A recent trend has been to take the architectural approach a step further by combining the logic blocks and interconnections of traditional FPGAs with embedded microprocessors and related peripherals to form a complete "system on a programmable chip". This work reflects the architecture by Ron Perlof and Hana Potash of Burroughs Advanced Systems Group which combined a reconfigurable CPU architecture on a single chip called the SB24. That work was done in 1982. Examples of such hybrid technologies can be found in the Xilinx Virtex-II PRO and Virtex-4 devices, which include one or more PowerPC processors embedded within the FPGA's logic fabric. The Atmel FPSLIC is another such device, which uses an AVR processor in combination with Atmel's programmable logic architecture. The Actel SmartFusion devices incorporate an ARM_architecture Cortex-M3 hard processor core (with up to 512kB of flash and 64kB of RAM) and analog peripherals such as a multi-channel ADC and DACs to their flash-based FPGA fabric. [16]

As previously mentioned, many modern FPGAs have the ability to be reprogrammed at "run time," and this is leading to the idea of reconfigurable computing or reconfigurable systems — CPUs that reconfigure themselves to suit the task at hand. The Mitrion Virtual Processor from Mitrionics is an example of a reconfigurable soft processor, implemented on FPGAs. However, it does not support dynamic reconfiguration at runtime, but instead adapts itself to a specific program. [16]

2.4 Comparisons:

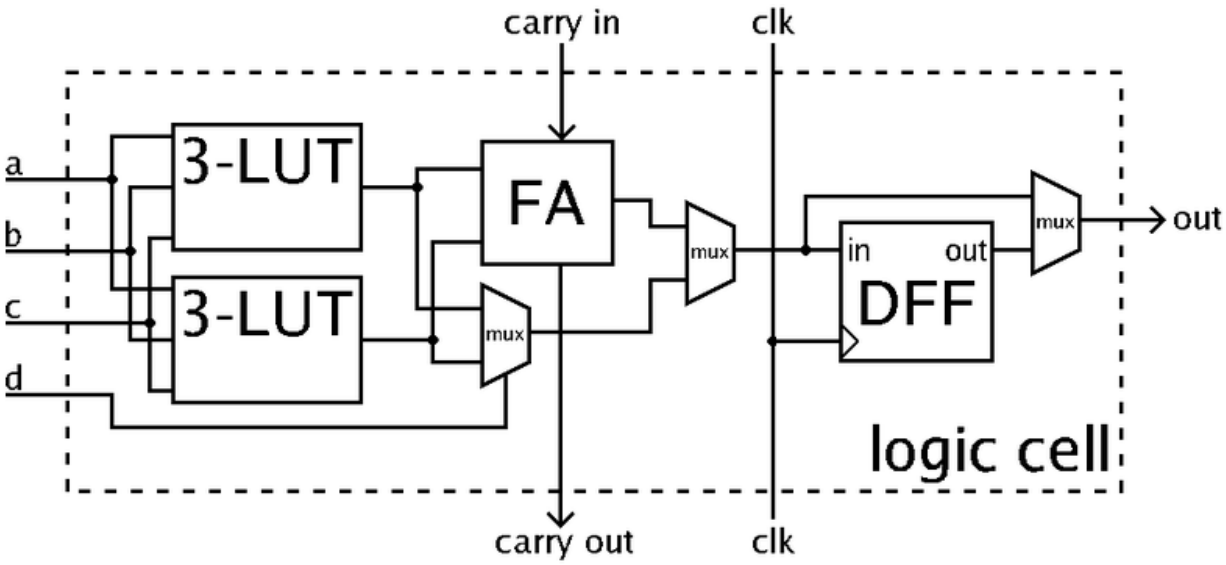
Historically, FPGAs have been slower, less energy efficient and generally achieved less functionality than their fixed ASIC counterparts. A combination of volume, fabrication improvements, research and development, and the I/O capabilities of new supercomputers have largely reduced the performance gap between ASICs and FPGAs. The primary differences between CPLDs and FPGAs are of the architecture level. A CPLD has a restrictive structure consisting of one or more programmable sum-of-products logic arrays feeding a relatively small number of clocked registers. With respect to security, FPGAs have both advantages and disadvantages as compared to ASICs or secure microprocessors. FPGAs' flexibility makes malicious changes during fabrication a lower risk. For many FPGAs, the loaded design is exposed while it is loaded (typically on every power-on). To address this issue, some FPGAs support bit stream encryption. [16]

2.5 Applications:

Applications of FPGAs include digital signal processing, software-defined radio, aerospace and defense systems,ASIC prototyping, medical imaging, cryptography, bioinformatics computer hardware emulation, radio astronomy, metal detection etc. The inherent parallelism of the logic resources on an FPGA allows for considerable computational throughput even at a low MHz clock rates. The flexibility of the FPGA allows for even higher performance by trading off precision and range in the number format for an increased number of parallel arithmetic units. This has driven a new type of processing called reconfigurable computing, where time intensive tasks are offloaded from software to FPGAs. FPGAs have been reserved for specific vertical applications where the volume of production is small. For these low-volume applications, the premium that companies pay in hardware costs per unit for a programmable chip is more affordable than the development resources spent on creating an ASIC for a low-volume application. Today, new cost and performance dynamics have broadened the range of viable applications. [16]

2.6 Architecture:

The most common FPGA architecture consists of an array of logic blocks (called Configurable Logic Block, CLB, or Logic Array Block, LAB, depending on vendor), I/O pads, and routing channels. Generally, all the routing channels have the same width (number of wires). Multiple I/O pads may fit into the height of one row or the width of one column in the array.



In general, a logic block (CLB or LAB) consists of a few logical cells. A typical cell consists of a 4-input Lookup table (LUT), a Full adder (FA) and a D-type flip-flop, as shown. The LUT are in this figure split into two 3-input LUTs. In normal mode those are combined into a 4-input LUT through the left mux. In arithmetic mode, their outputs are fed to the FA. The selection of mode is programmed into the middle mux. The output can be either synchronous or asynchronous, depending on the programming of the mux to the right, in the figure above. In practice, entire or parts of the FA are put as functions into the LUTs in order to save space. [16]

2.7 FPGA Design and Programming:

To specify the behavior of the FPGA, the user provides a hardware description language (HDL) or a schematic design. The HDL form is more suited to work with large structures because it's possible to just specify them numerically rather than having to draw every piece. However, schematic entry can allow for easier imagination of a design. Then, using an electronic design automation tool, a technology-mapped netlist is generated. The netlist can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software. The user will validate the map, place and route results via timing analysis, simulation, and other verification methods. Once the design and validation process is complete, the binary file generated is used to reconfigure the FPGA. The most common HDLs are VHDL and Verilog. Though these two languages are similar but we prefer VHDL for programming because of its widely in use. [16]

2.8 Major Manufacturers:

Xilinx and Altera are the current FPGA market leaders and long-time industry rivals. Together, they control over 80 percent of the market, with Xilinx alone representing over 50 percent. Other competitors include Lattice Semiconductor (SRAM based with integrated configuration Flash, instant-on, low power, live reconfiguration), Actel (antifuse, flash-based, mixed-signal), SiliconBlue Technologies(low power), Achronix (RAM based, 1.5 GHz fabric speed), and QuickLogic (handheld focused CSSP, no general purpose FPGAs). [16]

CHAPTER 3

VHDL- The Language of Hardware

3.1 Introduction:

VHDL is an acronym for VHSlC Hardware Description Language (VHSIC is an acronym for Very High Speed Integrated Circuits). It is a hardware description language that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level. The complexity of the digital system being modeled could vary from that of a simple gate to a complete digital electronic system, or anything in between. The digital system can also be described hierarchically. Timing can also be explicitly modeled in the same description. The language not only defines the syntax but also defines very clear simulation semantics for each language construct. Therefore, models written in this language can be verified using a VHDL simulator. It is a strongly typed language and is often verbose to write. It inherits many of its features, especially the sequential language part, from the ADA programming language. Because VHDL provides an extensive range of modeling capabilities, it is often difficult to understand. [17]

3.2 History:

The requirements for the language were first generated in 1981 under the VHSIC program. In this program, a number of U.S. companies were involved in designing VHSIC chips for the Department of Defense (DoD). At that time, most of the companies were using different hardware description languages to describe and develop their integrated circuits. The initial version of VHDL, designed to IEEE standard 1076-1987, included a wide range of data types, including numerical (integer and real), logical (bit and boolean), character and time, plus arrays of bit called bit_vector and of character called string. A problem not solved by this edition, however, was "multi-valued logic", where a signal's drive strength (none, weak or strong) and unknown values are also considered. This required IEEE standard 1164, which defined the 9-value logic types: scalar std_ulogic and its vector version std_ulogic_vector. In June 2006, VHDL Technical Committee of Accellera (delegated by IEEE to work on next update of the standard) approved so called Draft 3.0 of VHDL-2006.

While maintaining full compatibility with older versions, this proposed standard provides numerous extensions that make writing and managing VHDL code easier. Key changes include incorporation of child standards (1164, 1076.2, 1076.3) into the main 1076 standard, an extended set of operators, more flexible syntax of 'case' and 'generate' statements, incorporation of VHPI (interface to C/C++ languages) and a subset of PSL (Property Specification Language). These changes should improve quality of synthesizable VHDL code, make testbenches more flexible, and allow wider use of VHDL for system-level descriptions. [17]

In February 2008, Accellera approved VHDL 4.0 also informally known as VHDL 2008, which addressed more than 90 issues discovered during the trial period for version 3.0 and includes enhanced generic types. In 2008, Accellera released VHDL 4.0 to the IEEE for balloting for inclusion in IEEE 1076-2008. The VHDL standard IEEE 1076-2008 was approved by REVCOM in September 2008. [17]

3.3 Capabilities:

• The language can be used as an exchange medium between chip vendors and CAD tool users. Different chip vendors can provide VHDL descriptions of their components to system designers. CAD tool users can use it to capture the behavior of the design at a high level of abstraction for functional simulation.

• The language supports hierarchy, that is, a digital system can be modeled as a set of interconnected components; each component, in turn, can be modeled as a set of interconnected subcomponents.

• The language supports flexible design methodologies: top-down, bottom-up, or mixed.

• It supports both synchronous and asynchronous timing models.

• It is an IEEE and ANSI standard, and therefore, models described using this language is portable. The government also has a strong interest in maintaining this as a standard so that re-procurement and second-sourcing may become easier.

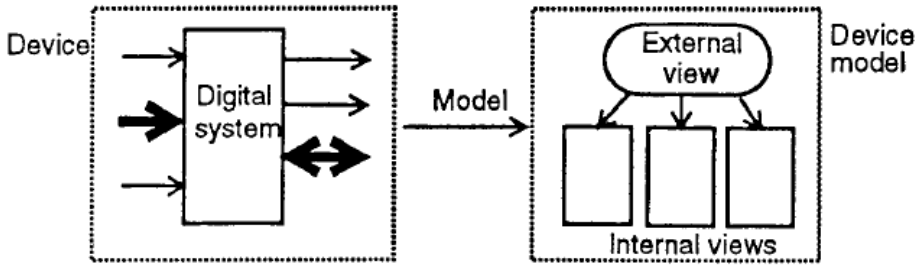
• The capability of defining new data types provides the power to describe and simulate a new design technique at a very high level of abstraction without any concern about the implementation details.

• Behavioral models that conform to a certain synthesis description style are capable of being synthesized to gate-level descriptions.

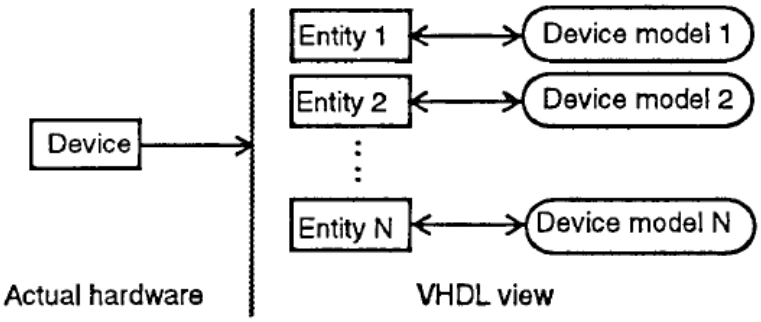
• A common language can be used to describe library components from different vendors. Tools that understand VHDL models will have no difficulty in reading models from a variety of vendors since the language is a standard. [17]

3.4 Hardware Abstraction:

VHDL is used to describe a model for a digital hardware device. This model specifies the external view of the device and one or more internal views. The internal view of the device specifies the functionality or structure, while the external view specifies the interface of the device through which it communicates with the other models in its environment. Figure shows the hardware device and the corresponding software model. [17]



Device versus device model.

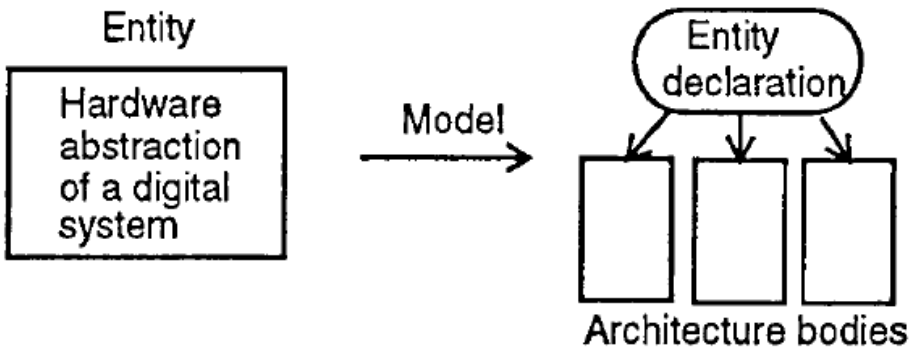


3.5 Language Concepts:

VHDL is a hardware description language that can be used to model a digital system. The digital system can be as simple as a logic gate or as complex as a complete electronic system. A hardware abstraction of this digital system is called an *entity* in this text. An entity X, when used in another entity Y, becomes a component for the entity Y. [17]

VHDL provides five different types of primary constructs, called" design units:

- Entity declaration.
- Architecture body.
- Configuration declaration.
- Package declaration.
- Package body. [17]



An entity and its model.

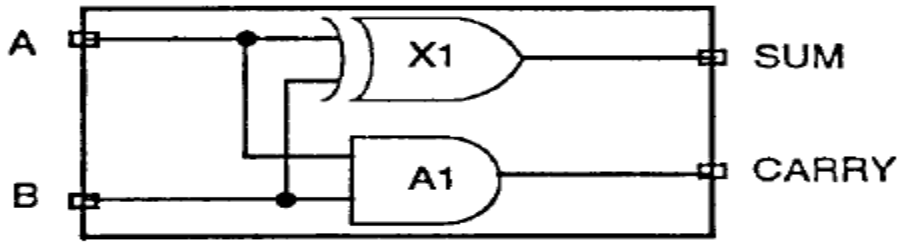
An entity is modeled using an entity declaration and at least one architecture body. The entity declaration describes the external view of the entity, for example, the input and output signal names. The architecture body contains the internal description of the entity, for example, as a set of interconnected components that represents the structure of the entity, or as a set of concurrent or sequential statements that represents the behavior of the entity. Each style of representation can be specified in a different architecture body or mixed within a single architecture body Figure 2.1 shows an entity and its model. [17]

A configuration declaration is used to create a configuration for an entity. It specifies the binding of one architecture body from the many architecture bodies that may be associated with the entity. It may also specify the bindings of components used in the selected architecture body to other entities. An entity may have any number of different configurations. [17]

A package declaration encapsulates a set of related declarations such as type declarations, subtype declarations, and subprogram declarations that can be shared across two or more design units. A package body contains the definitions of subprograms declared in a package declaration. [17]

3.6 Entity Declaration:

The entity' declaration specifies the name of the entity being modeled and lists the set of interface ports. Ports are signals through which the entity communicates with the other models in its external environment. [17]



A half-adder circuit.

3.7 Architecture Body:

The internal details of an entity are specified by an architecture body using any of the following modeling styles:

- As a set of interconnected components (to represent structure),
- As a set of concurrent assignment statements (to represent dataflow),
- As a set of sequential assignment statements (to represent be-hav.ior),
- Any combination of the above three. [17]

3.8 Structural style of Modeling:

The architecture body is composed of two parts: the declarative part (before the keyword begin) and the statement part (after the keyword begin). Two component declarations are present in the declarative part of the architecture body. The components XOR2 and AND2 may either be predefined components in a library, or if they do not exist, they may later be bound to other components in a library. The declared components are instantiated in the statement part of the architecture body using component instantiation statements. XI and A1 are the component labels for these component instantiations. [17]

```
architecture HA_STRUCTURE of HALF_ADDER is

        component XOR2

        port (X, Y: in BIT; Z: out BIT);

        end component;

        component AND2

        port (L, M: in BIT; N: out BIT);

        end component;

        begin

X1: XOR2 port map (A, B, SUM);

A1: AND2 port map (A, B, CARRY);


end HA_STRUCTURE;
```

3.9 Dataflow Style of Modeling:

In this modeling style, the flow of data through the entity is expressed primarily using concurrent signal assignment statements. The structure of the entity is not explicitly specified in this modeling style, but it can be implicitly deduced. Consider the following alternate architecture body for the HALF_ADDER entity that uses this style. [17]

```
architecture HA_CONCURRENTof HALF_ADDER is

        begin

        SUM <= A xor B after 8 ns;

        CARRY <= A and B after 4 ns;

end HA_CONCURRENT;
```

3.10 Behavorial Style of Modeling:

In contrast to the styles of modeling described earlier, the behavioral style of modeling specifies the behavior of an entity as a set of statements that are executed sequentially in the specified order. This set of sequential statements, that are specified inside a process statement, do not explicitly specify the structure of the entity but merely specifies its functionality. A process statement is a concurrent statement that can appear within an architecture body. For example, consider the following behavioral model for the DECODER2x4 entity. [17]

```
Architecture DEC_SEQUENTIAL of DECODER2x4 is

begin

process (A, B, ENABLE)

variable ABAR, BBAR: BIT;

begin

ABAR := not A;

BBAR := not B;

if (ENABLE = '1')

then

Z(3) <= not (A and B):
```

```
Z(0) <= not (ABAR and BBAR);

Z(2) <= not (A and BBAR);

Z(1 ) <= not (ABAR and B);

Else

Z<= "1111";
```

The variable declaration (starts with the keyword variable) declares two variables called ABAR and BBAR. A variable is different from a signal in that it is always assigned a value instantaneously and the assignment operator used is the: = compound symbol; contrast this with a signal that is assigned a value always. [17]

Signal assignment statements appearing within a process are called *sequential signal assignment statements.* Sequential signal assignment statements, including variable assignment statements, are executed sequentially independent of whether an event occurs on any signals in its right-hand-side expression or not; contrast this with the execution of concurrent signal assignment statements in the dataflow modeling style. [17]

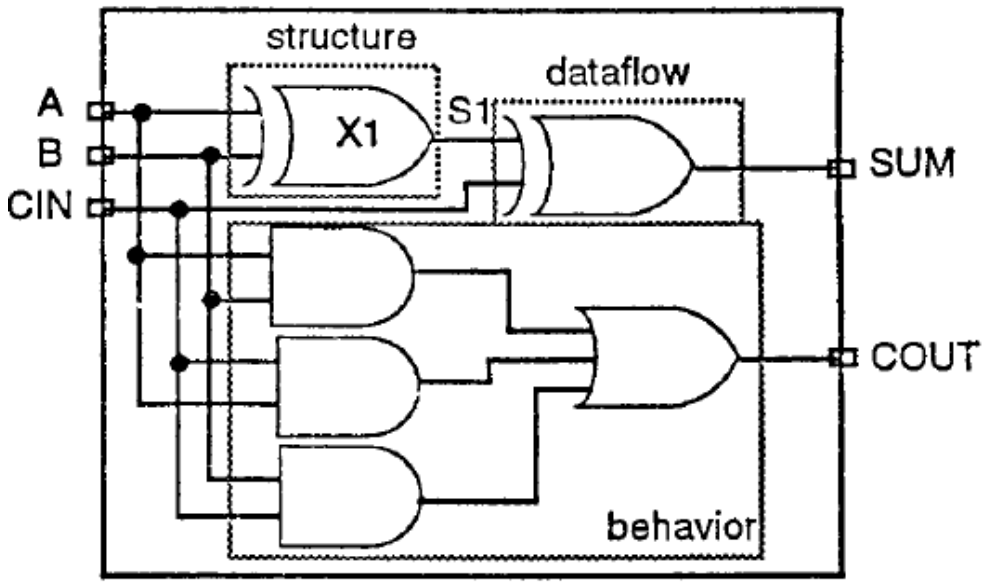
3.11 Mixed Style of Modeling:

It is possible to mix the three modeling styles that we have seen so far in a single architecture body. That is, within an architecture body, we could use component instantiation statements (that represent structure), concurrent signal assignment statements (that represent dataflow), and process statements (that represent behavior). [17]

```
entity FULL_ADDER is
      port (A, B, CIN: in BIT; SUM, COUT: out BIT);
end FULL_ADDER;

architecture FA_MIXED of FULL_ADDER is
component XOR2
port (A, B: in BIT; Z: out BIT);
end component;
signal S1: BIT;
begin
X1: XOR2 port map (A, B, S1 ); - structure.
process (A, B, CIN) - behavior.
variable T1, T2, T3: BIT;
begin
T1 :=A and B;
```

T2 := B and CIN;

T3:=A and CIN;
COUT <= T1 or T2 or T3;
end process;
SUM <= S1 xor CIN; - dataflow.
end FA_M!XED



A 1-bit full-adder.

3.12 Process Statement:

A process statement contains sequential statements that describe the functionality of a portion of an entity in sequential terms. The syntax of a process statement is [17]

```
[ process-label: ] process [ ( sensitivity-list ) ]
        [process-item-declarations]
begin
        sequential-statements; these are ->
        variable-assignment-statement
        signal-assignment-statement
        wait-statement
        loop-statement
        null-statement
exit-statement
```

CHAPTER 4

RANDOM NUMBER GENERATION

4.1 Random Number Generation:

A random number generator (RNG) is a device designed to generate a sequence of numbers or symbols that don't have any pattern. Hardware-based systems for random number generation are widely used, but often fall short of this goal, albeit they may meet some of the statistical tests for randomness for ensuring that they do not have any "de-cod able" patterns. Methods for generating random results have existed since ancient times, including dice, coin flipping, the shuffling of playing cards, the use of yarrow stalks and many other techniques. [1], [2]

The many applications of randomness have led to many different methods for generating random data. These methods may vary as to how unpredictable or random they are, and how quickly they can generate random numbers. [1], [2]

4.2 True Random Number Generators (TRNGs):

There are two principal methods used to generate random numbers. One measures some physical phenomenon that is expected to be random and then compensates for possible biases in the measurement process. The other uses mathematical algorithms that produce long sequences of apparently random numbers, which are in fact completely determined by an initial value, known as a seed. The former one is known as True Random Number Generator (TRNG). [1]

In comparison with PRNGs, TRNGs extract randomness from physical phenomena and introduce it into a computer. One can imagine this as a die connected to a computer. The physical phenomenon can be very simple, like the little variations in mouse movements or in the amount of time between keystrokes. In practice, however, one has to be careful about which source one chooses. For example, it can be tricky to use keystrokes in this fashion, because keystrokes are often buffered by the computer's operating system, meaning that several keystrokes are collected before they are sent to the program. To a program waiting for the keystrokes, it will seem as though the keys were pressed almost simultaneously, and there may not be a lot of randomness there after all. [3]

However, there are many other methods to get true randomness into your computer. A really good physical phenomenon to use is a radioactive source. The points in time at which a radioactive source decays are completely unpredictable, and they can easily be detected and fed into a computer, avoiding any buffering mechanisms in the operating system. The HotBits service at Fourmilab in Switzerland is an excellent example of a random number generator that uses this technique. Another suitable physical phenomenon is atmospheric noise, which is quite easy to pick up with a normal radio. This is the approach used by RANDOM.ORG. You could also use background noise from an office or disco, but you'll have to watch out for patterns. The fan from the computer can contribute to the noise, and since the fan is a rotating device, chances are the noise it produces won't be as random as atmospheric noise. [3]

Undoubtedly one of the effective approaches was the lavarand generator, which was built by Silicon Graphics and used snapshots of lava lamps to generate true random numbers. [3]

4.3 Pseudo Random Number Generators (PRNGs):

A pseudorandom number generator (PRNG),is an algorithm for generating a sequence of numbers that approximates the properties of random numbers. The sequence is not truly random. Although sequences that are closer to truly random can be generated using hardware random number generators, *pseudorandom* numbers are important in practice for simulations (e.g., of physical systems with the Monte Carlo method), and are important in the practice of cryptography . [4]

A PRNG can be started from an arbitrary starting state using a seed s. It will always produce the same sequence thereafter when initialized with that state. The maximum length of the sequence before it begins to repeat is determined by the size of the state. However, since the length of the maximum period doubles with each bit of 'state' added, it is easy to build PRNGs with periods long enough for many practical applications. [4]

Most pseudorandom generator algorithms produce sequences which are uniformly distributed by any of several tests. The security of most cryptographic algorithms and protocols using PRNGs is based on the assumption that it is infeasible to demarcate use of a suitable PRNG from the usage of a truly random sequence. The simplest examples of this dependency are stream ciphers, which work by exclusive or-ing the plaintext of a message with the output of a PRNG, producing cipher text. The design of cryptographically secure PRNGs is extremely difficult; because they must meet additional criteria .The size of its period is an important factor in the cryptographic suitability of a PRNG, but not the only one. [4]

The following algorithms are pseudorandom number generators: [5]

- **Blum Blum Shub**
- **Inversive congruential generator**
- **ISAAC (cipher)**
- **Lagged Fibonacci generator**
- **Linear congruential generator**
- **Linear feedback shift register**
- **Mersenne twister**
- **Multiply-with-carry**
- **Well Equidistributed Long-period Linear**
- **Xorshift**

Cipher algorithms and cryptographic hashes can also be used as pseudorandom number generators. These include: [5]

- **Block ciphers in counter mode**
- **Cryptographic hash function in counter mode**
- **Stream Ciphers**

4.4 Types of Pseudo Random Number Generators

Blum Blum Shub:

Blum Blum Shub (B.B.S.) is a pseudorandom number generator proposed in 1986 by Lenore Blum, Manuel Blum and Michael Shub (Blum et al., 1986).

Blum Blum Shub takes the form:

$$\mathbf{X_{n+1} = X_n^2 \bmod n}$$

Where $\mathbf{n=p \times q}$ is the product of two large primes p and q. At each step of the algorithm, some output is derived from x_{n+1}; the output is commonly the bit parity of X_{n+1} or one or more of the least significant bits of X_{n+1}.

The two primes, p and q, should both be congruent to 3 (mod 4) (this guarantees that each quadratic residue has one square root which is also a quadratic residue) and gcd(φ(p-1), φ(q-1)) should be small (this makes the cycle length large).[6]

Inversive Congruential Generator:

Inversive congruential generators are a type of nonlinear congruential pseudorandom number generator, which use the modular multiplicative inverse (if it exists) to generate the next number in a sequence. The standard formula for an inversive congruential generator is

$$\mathbf{X_{i+1} = (aX_i^{-1} + c) \pmod m}$$

Where $0 < x_i < m$. [7]

ISAAC (cipher):

ISAAC is a pseudorandom number generator and a stream cipher designed by Robert Jenkins (1996) to be cryptographically secure. The name is an acronym for Indirection, Shift, Accumulate, Add, and Count.

The ISAAC algorithm has similarities with RC4. It uses an array of 256 4-byte integers (called mm) as the internal state, writing the results to another 256-integer array, from which they are read one at a time until empty, at which point they are recomputed. The computation consists of altering mm[i] with mm[i^128], two elements of mm found by indirection, an accumulator, and a counter, for all values

of i from 0 to 255. Since it only takes about 19 32-bit operations for each 32-bit output word, it is extremely fast on 32-bit computers. [8]

Lagged Fibonacci Generator:

A Lagged Fibonacci generator (LFG) is an example of a pseudorandom number generator. This class of random number generator is aimed at being an improvement on the 'standard' linear. These are based on a generalization of the Fibonacci sequence.

The Fibonacci sequence may be described by the recurrence relation:

$$\mathbf{S_n = S_{n-1} + S_{n-2}}$$

This can be generalized to the sequence:

$$\mathbf{S_n = S_{n-j} \star S_{n-k} \ (mod\ m)}$$

In which case, the new term is some combination of any two previous terms. m is usually a power of 2 ($m = 2^M$), often 2^{32} or 2^{64}. The $\star$ operator denotes a general binary operation. This may be either addition, subtraction, multiplication etc. The theory of this type of generator is rather complex, and it may not be sufficient simply to choose random values for j and k. These generators also tend to be very sensitive to initialization.

Generators of this type employ k words of state (they 'remember' the last k values).

If the operation used is addition, then the generator is described as an Additive Lagged Fibonacci Generator or ALFG, if multiplication is used, it is a Multiplicative Lagged Fibonacci Generator or MLFG. [9]

Linear Congruential Generator:

A Linear Congruential Generator (LCG) represents one of the oldest and best-known pseudorandom number generator algorithms. The theory behind them is easy to understand, and they are easily implemented and fast. [10], [18]

The generator is defined by the recurrence relation:

$$\mathbf{X_{n+1} = (aX_n + c) \ mod\ m}$$

Linear Feedback Shift Register:

A linear feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state. The only linear function of single bits is xor, thus it is a shift register whose input bit is driven by the exclusive-or (xor) of some bits of the overall shift register value. The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state.

Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits which appears random and which has a very long cycle. [11]

Mersenne Twister:

The Mersenne twister is a pseudorandom number generator developed in 1997 by Makoto Matsumoto and Takuji Nishimura that is based on a matrix linear recurrence over a finite binary field F_2. It provides for fast generation of very high-quality pseudorandom numbers, having been designed specifically to rectify many of the flaws found in older algorithms.

Its name derives from the fact that period length is chosen to be a Mersenne prime. There are at least two common variants of the algorithm, differing only in the size of the Mersenne primes used. The newer and more commonly used one is the Mersenne Twister MT19937, with 32-bit word length. There is also a variant with 64-bit word length, MT19937-64, which generates a different sequence.

For a k-bit word length, the Mersenne Twister generates numbers with a uniform distribution in the range $[0,2^k - 1]$. [12], [19]

Multiply-with-carry:

Multiply-with-carry (MWC) is a method invented by George Marsaglia for generating sequences of random integers based on an initial set of from two to many thousands of randomly chosen seed values. The main advantages of the MWC method are that it invokes simple computer integer arithmetic and leads to very fast generation of sequences of random numbers with immense periods, ranging from around 2^{60} to $2^{2000000}$. [13]

Xorshift:

Xorshift is a category of pseudorandom number generators designed by George Marsaglia. It repeatedly uses the transform of exclusive or on a number with a bit shifted version of it. [14]

Well Equidistributed Long-period Linear:

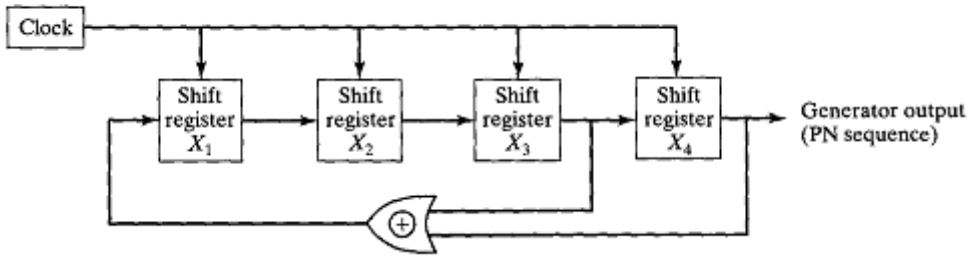
The Well Equidistributed Long-period Linear (WELL) is a pseudorandom number generator developed in 2006 by F. Panneton, P. L'Ecuyer, and M. Matsumoto that is based on linear recurrences modulo 2 over a finite binary field F_2. [15]

CHAPTER 5

PROGRAMMING IN VHDL FOR RANDOM GENERATORS

5.1 Implementation of PRNSG:

Pseudo random number sequence generator is generated in VHDL according to the following circuit based on the concept of shift register.



The sequence of numbers generated by this method is random. Because the period of the sequence is (2^n - 1).Where n is the number of shift registers used in the design. For 32 bit design the period is 4294967295.This is large enough for most of the practical applications.

5.2 Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity rng is
    generic ( wd : integer :=  32 );
port (
      clk : in std_logic;
      random_num : out std_logic_vector (wd-1 downto 0)   --output vector

    );
end rng;

architecture Behavioral of rng is
begin
process(clk)
variable rand_temp : std_logic_vector(wd-1 downto 0):=(wd-
1 => '1',others => '0');
variable temp : std_logic := '0';
begin
if(rising_edge(clk)) then
temp := rand_temp(wd-1) xor rand_temp(wd-2);
rand_temp(wd-1 downto 1) := rand_temp(wd-2 downto 0);
rand_temp(0) := temp;
end if;
```

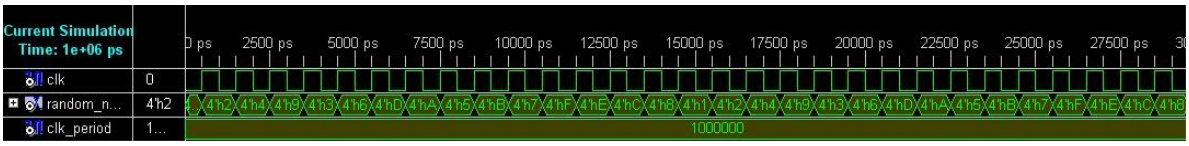
```
random_num <= rand_temp;
end process;

end rng;
```

5.3 Test bench:

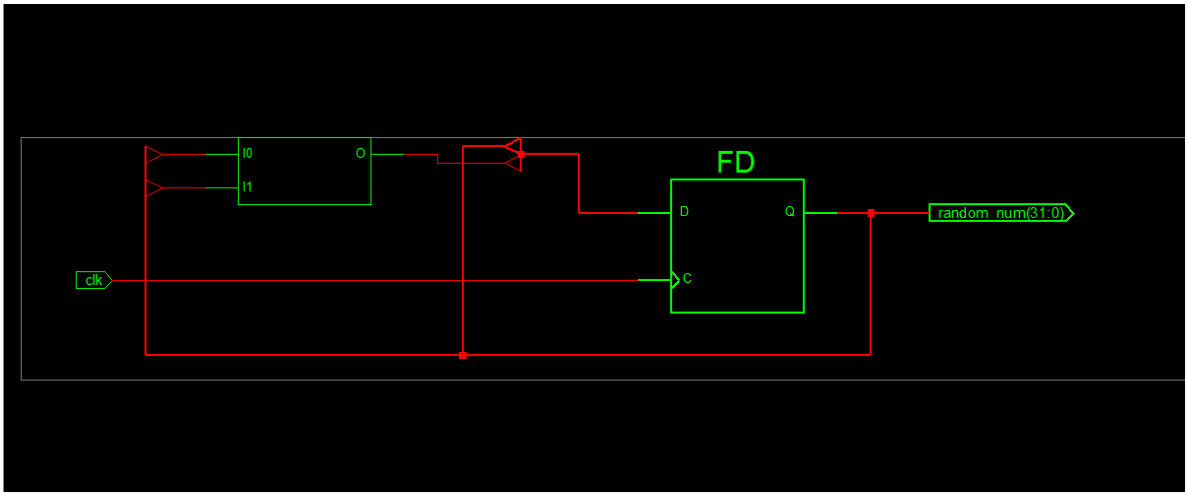
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY testb IS
END testb;

ARCHITECTURE behavior OF testb IS
   --Input and Output definitions.
   signal clk : std_logic := '0';
   signal random_num : std_logic_vector(3 downto 0);
   -- Clock period definitions
   constant clk_period : time := 1 ns;
BEGIN
        -- Instantiate the Unit Under Test (UUT)
   uut: entity work.random generic map (wd => 4) PORT MAP (
          clk => clk,
          random_num => random_num
        );
   -- Clock process definitions
   clk_process :process
   begin
                clk <= '0';
                wait for clk_period/2;
                clk <= '1';
                wait for clk_period/2;
   end process;
```

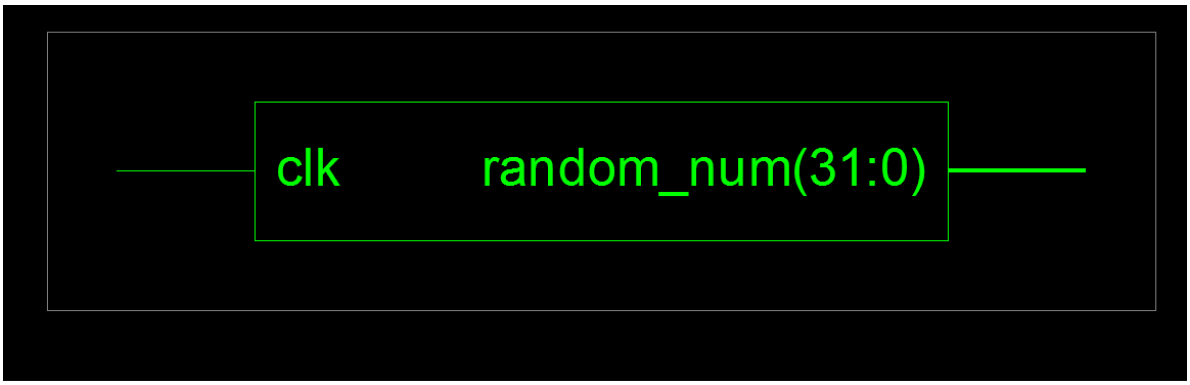
5.4 Simulation Diagram:

The simulation diagram is the following

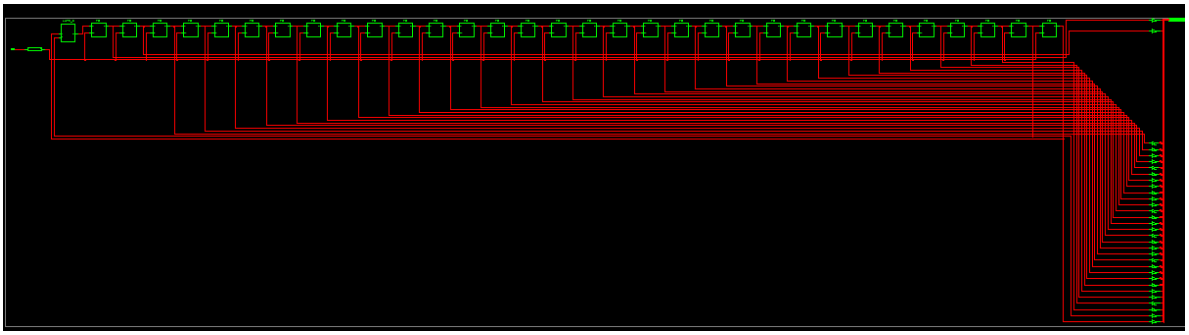


5.5 RTL Schematic:





5.6 Technology Schematic:



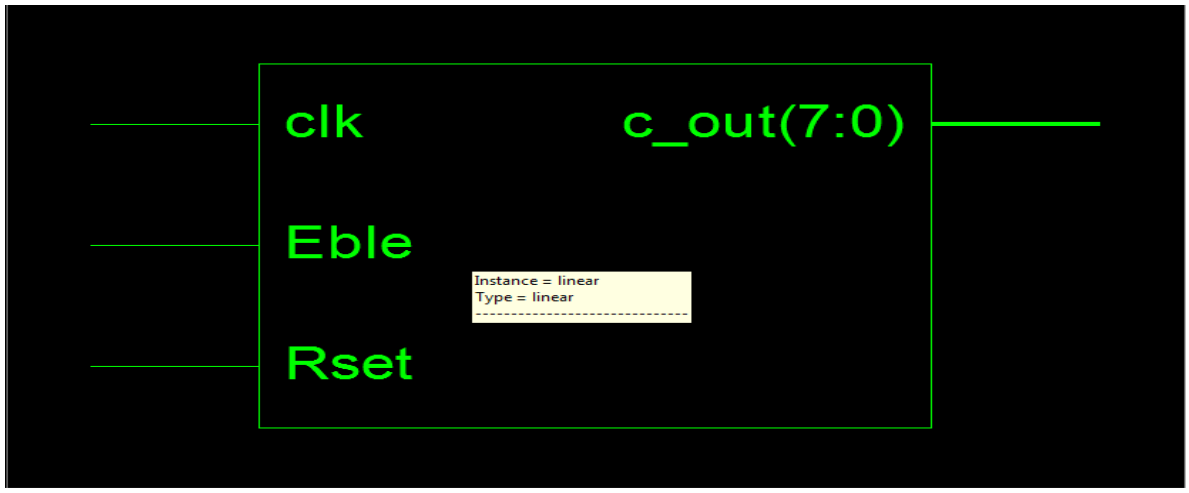
5.7 8-bit random generator using LFSR

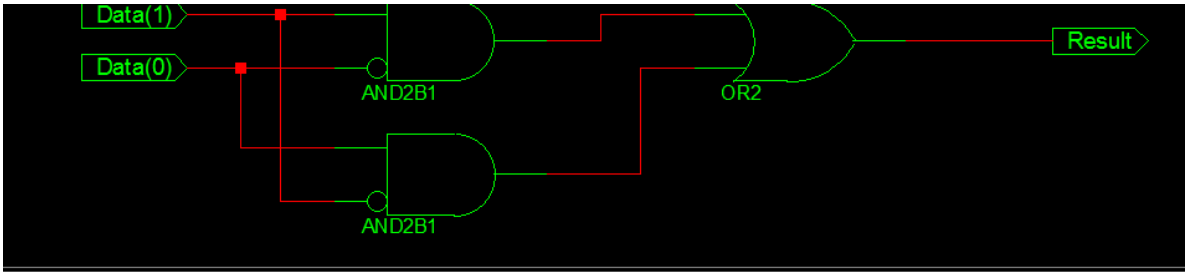
5.7.1 Code:

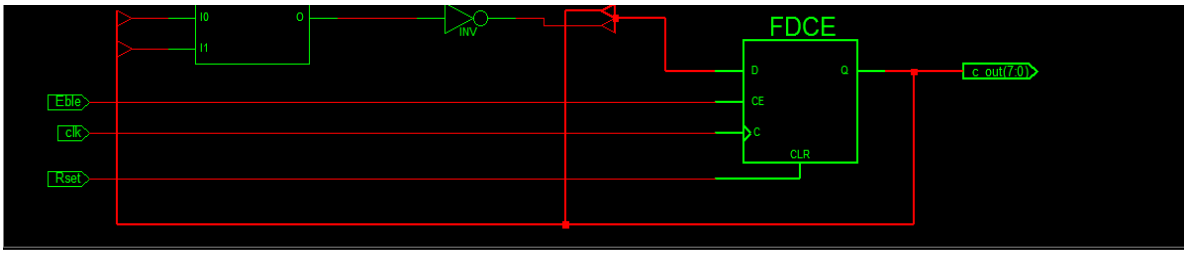
```
library ieee;

    use ieee.std_logic_1164.all;

entity linear is

  port (c_out   :out std_logic_vector (7 downto 0);--

 Eble :in  std_logic;                    --

 clk    :in  std_logic;                     --
Rset:in std_logic );

end linear;


architecture behavioral of linear is

    signal count :std_logic_vector (7 downto 0);

    signal linear_feedback :std_logic;
```

```
begin
    linear_feedback <= not(count(7) xor count(3));


    process (clk, Rset) begin
        if (Rset = '1') then
            count <= (others=>'0');
        elsif (rising_edge(clk)) then
            if (Eble = '1') then
                count <= (count(6) & count(5) & count(4) & count(3)
                          & count(2) & count(1) & count(0) &
                          linear_feedback);
            end if;
     end if;
    end process;
    c_out <= count;
end behavioral;
```

5.8 RTL Schematic:







5.9 16 BIT RANDOM NUMBER GENERATION WITH THE HELP OF XOR:

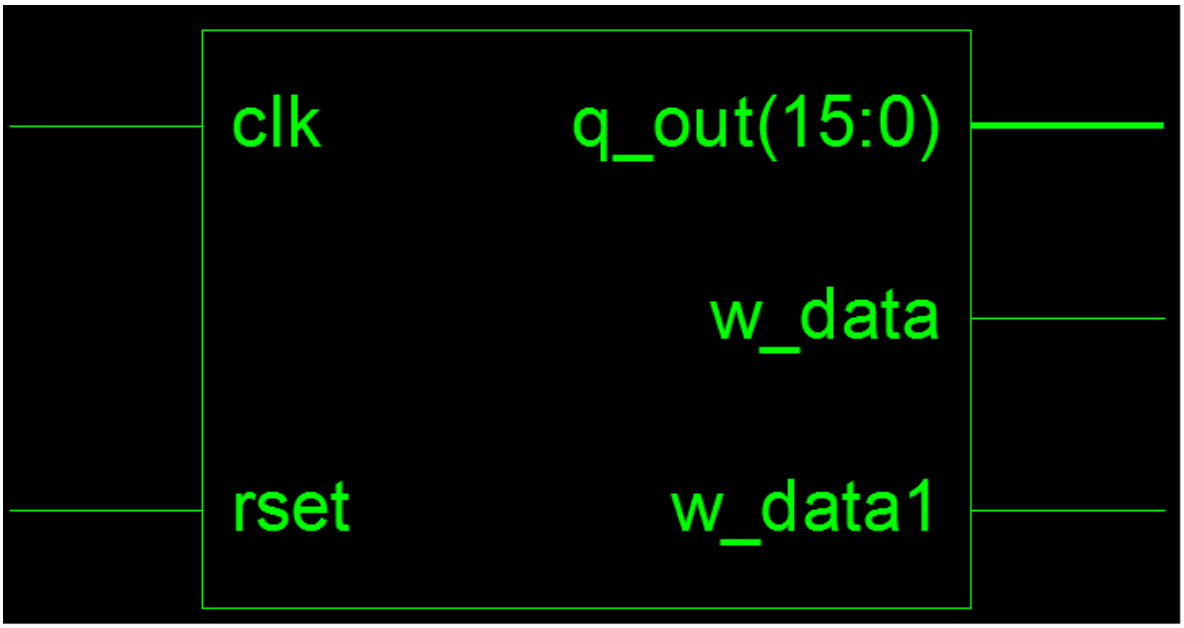
Here 16 bit random number is being generated with the help of XOR gate. The most significant digit and least significant digit both are passed through XOR gate and give the 1st bit of the random number and then this equation runs inside in the loop of 16 times to give and 16 bit random number.

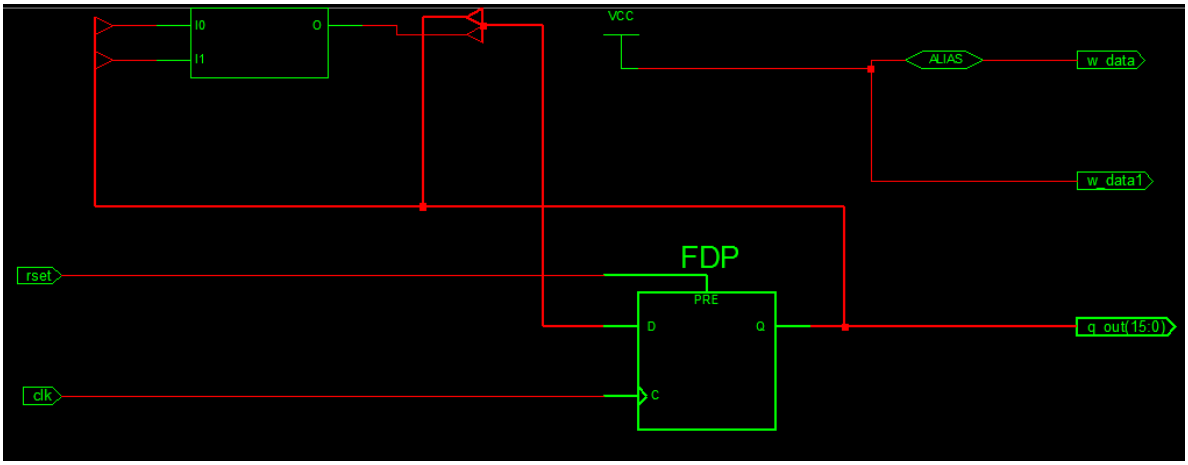
5.9.1 Code:

```
Library IEEE;

use IEEE.std_logic_1164.all;


Entity rng1 is

      port ( clk  : in   std_logic;

                rset : in  std_logic;
```

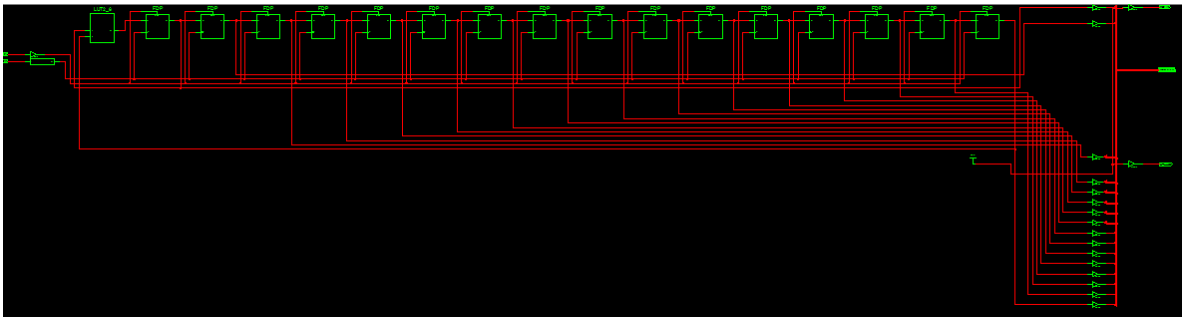
```
 q_out : out  std_logic_vector(15 downto 0);
             w_data : out std_logic;
             w_data1 : out std_logic
           );
end rng1;
Architecture behavioral of rng1 is
  signal qout_s : std_logic_vector(15 downto 0);
  signal tem_data : std_logic:='1';
begin
  q_out <= qout_s;
  random_gen : process(clk)
    variable  temp : std_logic;
  begin
    if (rset = '1') then
      qout_s <= "1111111111111111";
    elsif (clk'event and clk  = '1') then
      temp := tem_data;
      w_data <= temp;
      w_data1 <= tem_data;
      temp := tem_data;
      qout_s(0) <= qout_s(0) xor qout_s(15);

      for i in 15 downto 1 loop
            qout_s(i) <= qout_s(i - 1);
      end loop;
    end if;
  end process;
end behavioral;
```

5.10 RTL Schematic:





5.11 Technology Schematic:



5.12 Blum Blum Shub Generator:

The Blum Blum Shub Generator is known to be the cryptographically secure pseudo random number generator (CSPRNG).

The algorithm for BBS generator is as follows:

- Select two big prime numbers p and q, such that both the numbers leave a remainder of 3 when divided by 4.
- Choose n = p * q
- Choose seed s, such that s is relatively prime to n which means that neither p nor q is a factor of s.
- $X_o = s^2 \bmod n$
- The consequent values are generated according to the formula $X_i = (X_{i1})^2 \bmod n$
- A sequence of binary digits is produced according to the formula $B_i = X_i \bmod 2$

The output sequence is B_1, B_2, B_3, B_4…….

Now the BBS generator is first implemented in Turbo C++ as it is user friendly and easy to comprehend. The code in C++ is as follows:

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
void main()
{
      int B;
      long unsigned int X,p,q,s,n;
      cout<<endl<<"Enter the values of p and q: ";
      cin>>p>>q;
      n=p*q;
      cout<<endl<<n;
      cout<<endl<<"Enter the value of s: ";
      cin>>s;
      X=(s*s);
      cout<<X;
      X=X%n;
      for(int i=1;i<=50;i++)
      {
            X=(X*X)%n;
      cout<<endl<<"Value of X: "<<X;
            B=X%2;
            cout<<endl<<"Value of B: "<<B;
      }
      getch();
}
```

The output generated is as follows:

2895	1
7037	1
8929	1
6064	0
6010	0
11097	1
9111	1
3929	1

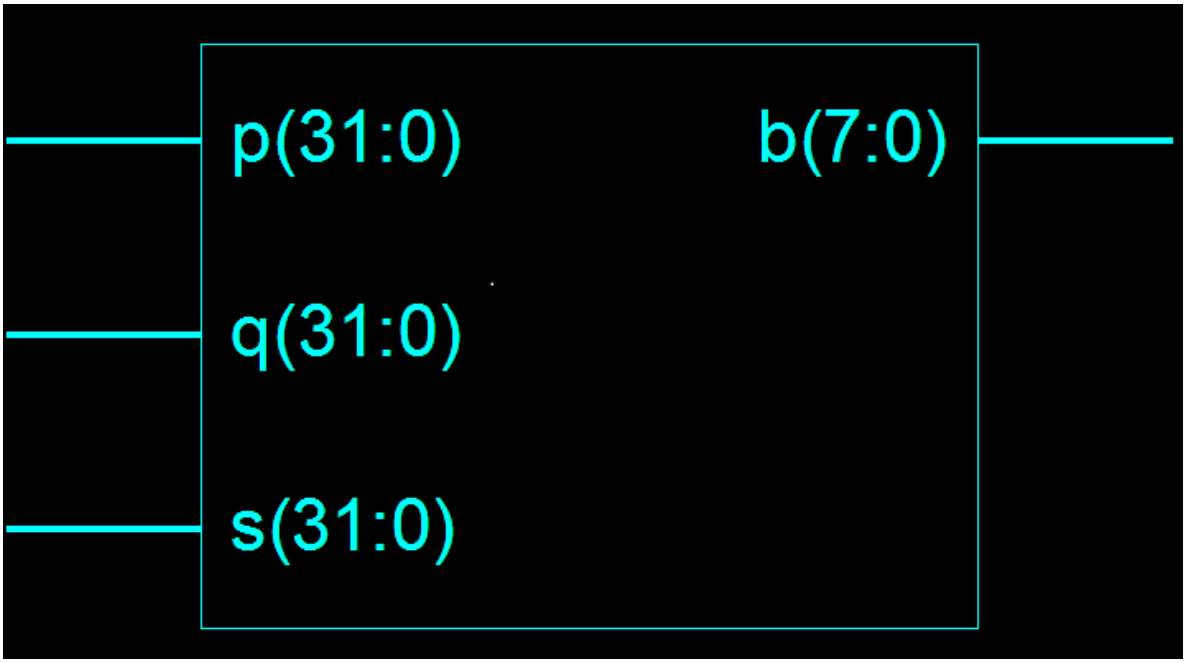
The first column refers to the values of X generated and second column refers to B_i.

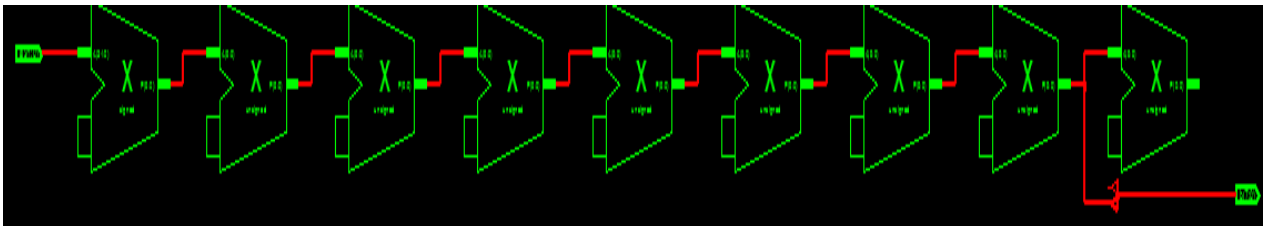
The VHDL code for the same is as follows:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity bbs is
Port ( p : in  integer;
       q : in  integer;
       s : in  integer;
       b : out  STD_LOGIC_VECTOR(7 downto 0) );
end bbs;

architecture Behavioral of bbs is
```

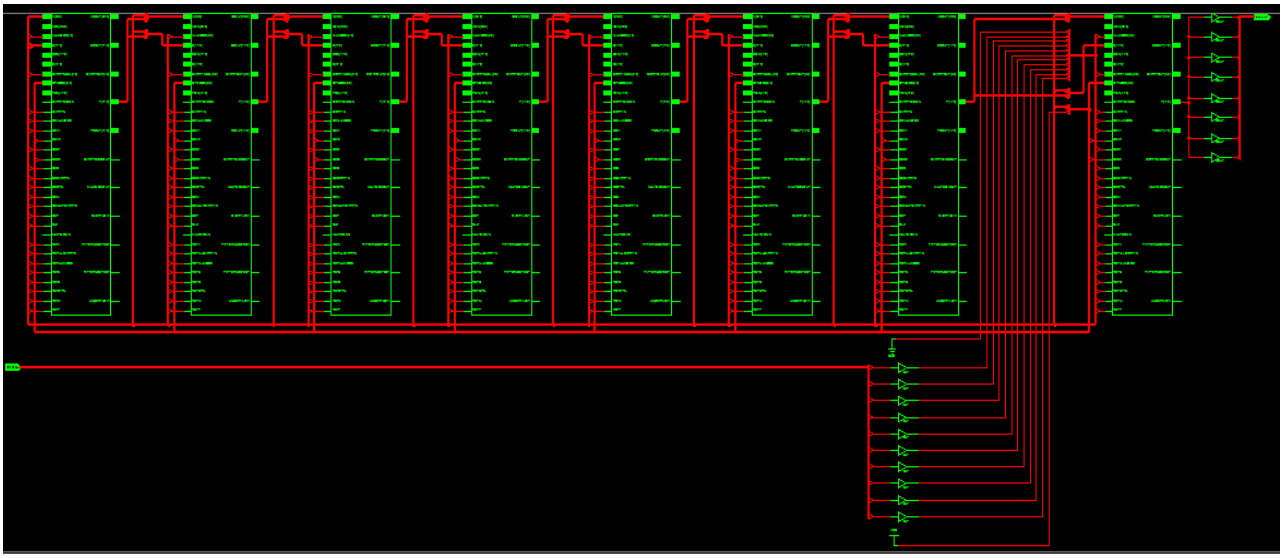
```
begin
process(p,q,s)
variable x0,x1,s1,x_0:integer;
constant p:integer:=11;
constant q:integer:=19;
constant n:integer:=551;
variable b1:integer ;
variable b2:std_logic;
variable r:std_logic_vector(7 downto 0);
  begin

                s1:=s*s;
                x0:=s1 mod 1024;
                for i in 0 to 7 loop
                x_0:=x0*x0;
                x1:=x_0 mod 1024;
    b1:=x1 mod 2;
                if b1=1 then
                b2:='1';
                else b2:='0';
                end if;
                r(6):=r(7);
                r(5):=r(6);
                r(4):=r(5);
                r(3):=r(4);
                r(2):=r(3);
                r(1):=r(2);
                r(0):=r(1);
    r(7):=b2;--so it will be serially transmitted but parallelly observed
                x0:=x1;
                b<=r;
                end loop;
        end process;
end Behavioral;
```

5.13 RTL Schematic:





5.14 Technology Schematic:



CONCLUSION

Various codes were written for generators like Pseudo Random Number Sequence Generator, 8 bit random generator using LFSR, 16 bit random generator and Blum Blum Shub generator. But Blum Blum Shub generator was found to be the most cryptographically secure generator. BBS generator was implemented in Turbo C++ and VHDL. Initially seed‘s’ and inputs p, q were taken. The inputs had to be two large prime numbers and relatively prime to s. Then mod function was used to calculate the remainder and the square of the same was used as the dividend in the next recursion. The syntax was found to be correct in case of VHDL but due to some hardware constraints it could not be implemented in FPGA.

REFERENCES

[1] http://en.wikipedia.org/wiki/Random_number_generation#cite_ref-0

[2] http://en.wikipedia.org/wiki/Hardware_random_number_generator

[3] http://www.random.org/randomness/

[4] http://en.wikipedia.org/wiki/Pseudorandom_number_generator#cite_note-0

[5] http://en.wikipedia.org/wiki/List_of_random_number_generators

[6] http://en.wikipedia.org/wiki/Blum_Blum_Shub

[7] http://en.wikipedia.org/wiki/Inversive_congruential_generator

[8] http://en.wikipedia.org/wiki/ISAAC_(cipher)

[9] http://en.wikipedia.org/wiki/Lagged_Fibonacci_generator

[10] http://en.wikipedia.org/wiki/Linear_congruential_generator

[11] http://en.wikipedia.org/wiki/Linear_feedback_shift_register

[12] http://en.wikipedia.org/wiki/Mersenne_twister

[13] http://en.wikipedia.org/wiki/Multiply-with-carry

[14] http://en.wikipedia.org/wiki/Xorshift

[15] http://en.wikipedia.org/wiki/Well_Equidistributed_Long-period_Linear

[16] http://en.wikipedia.org/wiki/Field-programmable_gate_array

[17] Bhasker J, A VHDL Primer, P T R Prentice Hall, Pages 1-2, 4-13, 28-30

[18] L'ECUYER PIERRE, TABLES OF LINEAR CONGRUENTIAL GENERATORS OF DIFFERENT SIZES AND GOOD LATTICE STRUCTURE, MATHEMATICS OF COMPUTATION Volume 68, Number 225, 1999, Pages 249-260

[19] Matsumoto, Makoto; Nishimura, Takuji; Hagita, Mariko; Saito, Mutsuo (2005), Cryptographic Mersenne Twister and Fubuki Stream/Block Cipher