DESIGN AND IMPLEMENTATION OF PRBS GENERATOR USING VHDL

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

Bachelor of Technology

In

Electronics & Instrumentation Engineering

By

SANDEEP MUKHERJEE
ROLL NO -10307017
&
RUCHIR PANDEY
ROLL NO -10307019

Department of Electronics & Communication Engineering

National Institute of Technology

Rourkela

2007

DESIGN AND IMPLEMENTATION OF PRBS GENERATOR USING VHDL

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

Bachelor of Technology

In

Electronics & Instrumentation Engineering

By

SANDEEP MUKHERJEE
ROLL NO -10307017
&
RUCHIR PANDEY
ROLL NO -10307019

Under the Guidance of
Prof. K.K. MAHAPATRA

Department of Electronics & Communication Engineering

National Institute of Technology

Rourkela

2007

National Institute of Technology

Rourkela

CERTIFICATE

This is to certify that the thesis entitled **"Design and Implementation of PRBS Generator using VHDL''** Submitted by **Sandeep Mukherjee, Roll No:10307017,** and **Ruchir Pandey, Roll No. 10307019,** in the partial fulfillment of the requirement for the degree of **Bachelor of Technology** in **Electronics & Instrumentation 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. K.K. Mahapatra
Dept. of ECE (E&IE)
National Institute of Technology
Rourkela - 769008

Acknowledgment

We avail this opportunity to extend our hearty indebtedness to our guide **Prof. K.K. Mahapatra**, Electronics & Communication Engineering Department, for his valuable guidance, constant encouragement and kind help at different stages for the execution of this dissertation work.

We also express our sincere gratitude to **Prof G. Panda**, Head of the Department, Electronics & Communication Engineering Department, for providing valuable departmental facilities.

Submitted by:

Sandeep Mukherjee
Roll No: 10307017
Electronics & Communication Engineering
National Institute of Technology Rourkela

Ruchir Pandey
Roll No: 10307019
Electronics & Communication Engineering
National Institute of Technology Rourkela

CONTENTS

ABSTRACT

Pseudo random binary sequence is essentially a random sequence of binary numbers. So PRBS generator is nothing but random binary number generator. It is '**random**' in a sense that the value of an element of the sequence is independent of the values of any of the other elements. It is '**pseudo**' because it is deterministic and after N elements it starts to repeat itself, unlike real random sequences.

The implementation of PRBS generator is based on the linear feedback shift register (LFSR). The PRBS generator produces a predefined sequence of 1's and 0's, with 1 and 0 occurring with the same probability. A sequence of consecutive **n*(2^n -1)** bits comprise one data pattern, and this pattern will repeat itself over time.

In this project, the entire design of the PRBS generator was implemented using VHDL programming language and the simulation were done and tested on the **XILINX ISE 9.1i** simulator. A separate program module for D-Flip-Flop was written and this module was called 16 times in the main program to get the 16-bit shift register. Now the taps 1, 2, 4 and 15 were taken out and XORed together and then was fed back to the first bit as an input to the shift register. The output to the PRBS generator was taken from all the 16-bits of the shift register. Thus the output of the PRBS generator cycles between 0 to 65535.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1. INTRODUCTION

1.1 Resilience :

The word “resilience” means the ability to adapt well to stress. It means that, overall you remain stable and maintain healthy levels of physical functioning in the face of disruption or chaos.

A resilient network is a network, which does not fail under any circumstances. Failure refers to a situation where the observed behaviour of a system differs from its specified behaviour. A failure occurs due to an error, caused by a fault. Faults can be hard or soft. For example a cable break is a hard failure whereas an intermittent noise in the network is a soft failure.

Resilience in the context of resilient network is the ability of the network,a device on the network, or a path on the network to respond to failure, resist failure, handle flux in demand and easily shift and configure – with little or no impact on service delivery. A resilient network is the agent that can help to diminish the loss of employee productivity in the event of a major disaster.

1.2 Need for Resilient Network:

Businesses in all the industries are becoming dependent on Information Technology (IT) and the intra- and inter- organizational online communication and collaboration it enables. Digitization and workforce mobilization, automation and embedded computing have changed the way enterprises do business and interact with their customers, employees and business partners. The requirements for business infrastructure have also changed. Business infrastructure must provide a stable IT foundation for the internal organization as well as allow integration with a virtual value chain of suppliers and customers. To effectively support the needs of today’s businesses, business infrastructure must, in effect, be RESILIENT. Resilient implies flexible and adaptive yet at the same

time fortified against all types of threats. Resilient network design is the key component of Resilience.

Resilient networks incorporate many of the elements of a highly available network. The resilient network architecture should include redundant (multiple) components that can take over the function of one another if one should fail. How the network, device or path reacts to failure should be determined before hand so that predictable network, device or path are present after response to failure.

1.3 Types of Failures:

Single point failure: It indicates that a system or a network can be rendered inoperable, or significantly impaired in operation, by the failure of one single component. For example, a single hard disk failure could bring down a server; a single router failure could break all connectivity for a network.

Multiple points of failure: It indicates that a system or a network can be rendered inoperable through a chain or combination of failures. For example, failure of a single router plus failure of a backup modem link cou
ld mean that all the connectivity is lost for a network. In general it is much more expensive to cope with multiple points of failure and often financially impractical.

Disaster recovery is the process of identifying all potential failures, their impact on the network as a whole, and planning the means to recover from such failures.

In our project we have implemented two types of failures:

- ***Link failure****:* In case of link failure if one link between two nodes fails then only that link gets failed. It won't affect any other nodes in the network.
- ***Node failure*** *:* In case of node failure if any node fails, then all the links connected to it also fail

CHAPTER 2

VHDL – THE LANGUAGE OF HARDWARE

2.1 Introduction

The **VHSIC Hardware Description Language** (VHDL) is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware.

2.2 Design Entities and Configurations

The ***design entity*** is the primary hardware abstraction in VHDL. It represents a portion of a hardware design that has well defined inputs and outputs and performs a well defined function. A design entity may represent an entire system, a subsystem, a board, a chip, a macro-cell, a logic gate, or any level of abstraction in between. A ***configuration*** can be used to describe how design entities are put together to form a complete design.

A design entity may be described in terms of a hierarchy of blocks, each of which represents a portion of the whole design. The top level block on such a hierarchy is the design entity itself; such a block is an external block that resides in a library and may be used as a component of other designs. Nested blocks in hierarchy are internal blocks, defined by block statements.

A design entity may also be described in terms of interconnected components. Each component of a design entity may be bound to a lower-level design entity in order to

define the structure or behavior of that component. Successive decomposition of a design entity into components, and binding those components to other design entities that may be decomposed in like manner, result in a hierarchy of design entities representing complete design.

2. 3 Entity Declaration

An entity declaration defines the interface between a given design entity and the environment in which it is used. It may also specify declarations and statements that are part of the design entity. A given entity declaration may be shared by many design entities, each of which has a different architecture. Thus, an entity declaration can potentially represent a class of design entities, each with the same interface.

entity_declaration::=
entity identifier is
entity_header
entity_declarative_part
[**begin**
entity_statement_part]
end [**entity**][*entity*_simple_name];

The entity header and entity declarative part consist of declarative items that pertain to each design entity whose interface is defined by the entity declaration. The entity statement part, if present, consists of concurrent statements that are present in each such design entity.

If a simple name appears at the end of an entity declaration, it must repeat the identifier of the entity declaration.

2.4 Entity Header

The entity header declares objects used for communication between a design entity and its environment.

entity_header::=
    [*formal*_generic_clause]
    [*formal*_port_clause]
generic_clause::=
    generic (generic_list);

port_clause::=
    port (port_list);

The generic list in the formal generic clause defines generic constants whose values may be determined by the environment. The port list in the formal port clause defines the input and output ports of the design entity.
In certain circumstances, the names of generic constants and ports declared in the entity header become visible outside of the design entity.

Examples:
— An entity declaration with port declarations only:

```
entity  Full_Adder is
        Port (X, Y, Cin: in Bit;
        Cout, Sum: out Bit);
end Full_Adder;
```

— An entity declaration with generic declarations also:

```
entity AndGate is
        Generic
            (N: Natural: = 2);
```

```
        Port
            (Inputs: in Bit_Vector (1 to N) ;
             Result: out Bit);
    end entity AndGate;
```

— An entity declaration with neither:

```
    entity TestBench is
    end TestBench;
```

2.5 Generics

Generics provide a channel for static information to be communicated to a block from its environment. The following applies to both external blocks defined by design entities and to internal blocks defined by block statements.

generic_list::= generic_interface_list

The value of a generic constant may be specified by the corresponding actual in a generic association list. If no such actual is specified for a given formal generic (either because the formal generic is unassociated or because the actual is **open),** and if a default expression is specified for that generic, the value of this expression is the value of the generic. It is an error if no actual is specified for a given formal generic and no default expression is present in the corresponding interface element. It is an error if some of the sub elements of a composite formal generic are connected and others are either unconnected or unassociated.

2.6 Ports

Ports provide channels for dynamic communication between a block and its environment. The following applies to both external blocks defined by design entities and to internal blocks defined by block statements, including those equivalents to component instantiation statements and generate statements.

port_list::= port_interface_list

To communicate with other blocks, the ports of a block can be associated with signals in the environment in which the block is used. Moreover, the ports of a block may be associated with an expression in order to provide these ports with constant driving values; such ports must be of mode **in**. A port is itself a signal thus, a formal port of a block may be associated as an actual with a formal port of an inner block. The port, signal, or expression associated with a given formal port is called the actual corresponding to the formal port. The actual, if a port or signal, must be denoted by a static name. The actual, if an expression, must be a globally static expression.
After a given description is completely elaborated, if a formal port is associated with an actual that is itself a port, then the following restrictions apply depending upon the mode of the formal port:

a) For a formal port of mode **in**, the associated actual may only be a port of mode **in, inout**, or **buffer**.
b) For a formal port of mode **out**, the associated actual may only be a port of mode **out** or **inout**.
c) For a formal port of mode **inout,** the associated actual may only be a port of mode **inout.**
d) For a formal port of mode **buffer,** the associated actual may only be a port of mode **buffer.**
e) For a formal port of mode **linkage,** the associated actual may be a port of any mode.

A buffer port may have at most one source. Furthermore, after a description is completely elaborated, any actual associated with a formal buffer port may have at most one source.
If a formal port is associated with an actual port, signal, or expression, then the formal port is said to be *connected.* If a formal port is instead associated with the reserved word **open** then the formal is said to be unconnected. A port of mode **in** may be unconnected or unassociated only if its declaration includes a default expression. A port of any mode other than **in** may be unconnected or unassociated as long as its type is not an unconstrained array type. It is an error if some of the sub elements of a composite formal port are connected and others are either unconnected or unassociated.

2.7 Entity Declarative Part

The entity declarative part of a given entity declaration declares items that are common to all design entities whose interfaces are defined by the given entity declaration.

entity_declarative_part::=
{entity_declarative_item}

entity_declarative_item ::=
subprogram_declaration
|subprogram_body
|type_declaration
|subtype_declaration
|constant_declaration
|signal_declaration
|shared_variable_declaration
|file_declaration

|alias_declaration
|attribute_declaration
|attribute_specification
|disconnection_specification
|use_clause
|group_template_declaration
|group_declaration

Names declared by declarative items in the entity declarative part of a given entity declaration are visible within the bodies of corresponding design entities, as well as within certain portions of a corresponding configuration declaration.

Example:

— An entity declaration with entity declarative items:

```
entity ROM is
        Port (Addr: in Word;
              Data: out Word;
              Sel: in Bit);
        type Instruction is array (1 to 5) of Natural;
        type Program is array (Natural range <>) of Instruction;
        use Work.OpCodes.all, Work.RegisterNames.all;
        constant ROM_Code: Program: =
        (
          (STM, R14, R12, 12, R13),
          (LD, R7, 32, 0, R1),
          (BAL, R14, 0, 0, R7),
                •
                • -- etc.
                •
      ) ;
end ROM;
```

2.8 Entity Statement Part

The entity statement part contains concurrent statements that are common to each design entity with this interface.

entity_statement_part ::=

{entity_statement}

entity_statement ::=

concurrent_assertion_statement

|passive_concurrent_procedure_call

|passive_process_statement

Only concurrent assertion statements, concurrent procedure call statements, or process statements may appear in the entity statement part. All such statements must be passive. Such statements may be used to monitor the operating conditions or characteristics of a design entity.

Example:

— An entity declaration with statements:

```
entity Latch is
        Port (Din: in Word;
              Dout: out Word;
              Load: in Bit;
              Clk: in Bit);
        constant Setup: Time := 12 ns;
        constant PulseWidth: Time := 50 ns;
        use Work.TimingMonitors.all;
begin
        assert Clk='1' or Clk'Delayed'Stable (PulseWidth);
        CheckTiming (Setup, Din, Load, Clk);
end;
```

2.9 Architecture Bodies

An architecture body defines the body of a design entity. It specifies the relationships between the inputs and outputs of a design entity and may be expressed in terms of structure, data ow, or behavior. Such specifications may be partial or complete.

architecture_body::=

architecture identifier of entity_name is

architecture_declarative_part

begin

architecture_statement _part

end [architecture] [architecture_simple_name];

The identifier defines the simple name of the architecture body **,** this simple name distinguishes the architecture bodies with the same entity declaration.
The entity name identifies the name of the entity declaration that defines the interface of this design entity. For a given design entity , both the entity declaration and the associated architecture body must reside in the same library.

If a architecture name appears at the end of the architecture name body , it must repeat the identifier of the architecture body.

More than one architecture body may exist corresponding to a given entity declaration. Each declares a different body with the same identifier; thus, each together with the entity declaration represents a different design entity with the same interface.
design entity with the same interface.

2.9.1 Architecture declarative part

The architecture declarative part contains declarations of items that are available for use within the block defined by the design entity.

architecture _declrative _part::=

{block _declarative_item}

block_declarative_item::=

subprogram_declartation

subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| shared_variable_declaration
| file_declaration
| alias_declaration
| component_declaration
| attribute_declaration
| attribute_specification
| configuration_specification
| disconnection_specification
| use_clause
| group_template_declaration
| group_declaration

2.9.2 Architecture statement part

The architecture statement part contains the statements that contain the internal organization and/or operation of the block defined by the design entity.

architecture_statement _part::;
{cocurrent_part}

All of the statements in the architecture statement part are concurrent statements which execute asynchronously with respect to one another.

Example ::

----A body of entity **Full adder** :

```
architecture data flow of  entity Full Adder is
      signal A,B:Bit;
begin
      A <= X xor Y;
      B <= A and Cin;
      Sum <= A xor Cin;
      Cout <= B or ( X and Y );
end architecture Data flow;
```

----A body of entity Testbench ::

```
library test;
use test.components.all;
architecture Structure of testbench is
        component Full Adder
                port(X, Y, Cin: Cout,  Sum: out Bit);
        end component;
signal  A,B,C,D,E,F,G: Bit;
signal  OK: Boolean:
begin
        UUT: Full_Adder port map (A,B,C,D,E);
        Generator: AdderTest port map (A,B,C,F,G);
        Comparator: AdderCheck port map (D,E,F,G,OK);
End Structure;
```

-----A body of entity AndGate::

```
architecture Behavior of AndGate is
begin
        process (Inputs)
            variables  Temp: Bit;
             begin
                    Temp = '1';
                    for i in  inputs'Range loop
                            if input(i) = '0' then
                                Temp = '0';
                                 exit;
                            end if;
                    end loop;
                    Result <= after 10 ns;
          end process;
    end Behaviour;
```

CHAPTER 3

PRBS-BASIC IMPLEMENTATION TECHNIQUES

3.1 Introduction

PRBS or Pseudo Random Binary Sequence is essentially a random sequence of binary numbers. It is **random** in a sense that the value of an element of the sequence is independent of the values of any of the other elements. It is '**pseudo**' because it is deterministic and after N elements it starts to repeat itself, unlike real random sequences**..**

Examples of random sequences are **radioactive decay** and **white noise.**

A binary sequence (BS) is a sequence of N bits, a_j for j = 0, 1, ..., N − 1, i.e. m ones and N m zeros. A binary sequence is pseudo-random (PRBS) if its autocorrelation function,

$$C(v) = \sum_{j=0}^{N-1} (a_j\, a_{j+v})$$

has only two values:

$C(v) = m$ if $v = 0 \pmod N$

$C(v) = mc$ if $v \neq 0 \pmod N$

where

$c = (m - 1)/(N - 1)$

is called the ***duty cycle*** of the **PRBS**.

The implementation of PRBS generator is based on the linear feedback shift register, which consists of ‘n’ master slave flip-flops. The PRBS generator produces a predefined sequence of 1's and 0's, with 1 and 0 occurring with the same probability

3.2 Implementation

PRBS is implemented using **LFSR** or Linear Feedback Shift Register.

LFSR is an *n*-bit shift register which pseudo-randomly scrolls between 2^n-1 values, but does it *very quickly* because there is minimal combinational logic involved. Once it reaches its final state, it will traverse the sequence exactly as before.

SHIFT REGISTERS:

One of the two main parts of an LFSR is the shift register (the other being the feedback function). A shift register is a device whose identifying function is to shift its contents into adjacent positions within the register or, in the case of the position on the end, out of the register. The position on the other end is left empty unless some new content is shifted into the register.

The contents of a shift register are usually thought of as being binary, that is, ones and zeroes. If a shift register contains the bit pattern 1101, a shift (to the right in this case) would result in the contents being 0110; another shift yields 0011. After two more shifts, things tend to get boring since the shift register will never contain anything other than zeroes.

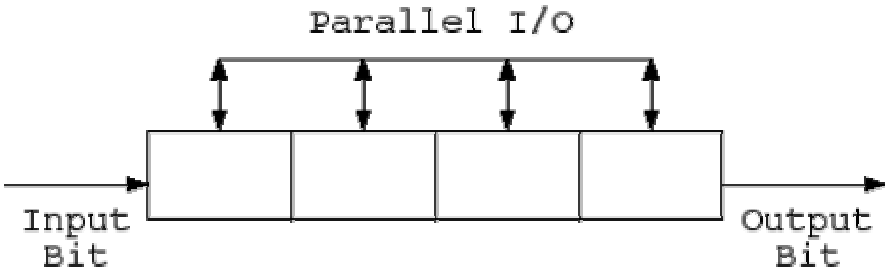
Two uses for a shift register are:

1) convert between parallel and serial data

2) delay a serial bit stream.

The conversion function can go either way -- fill the shift register positions all at once (parallel) and then shift them out (serial) or shift the contents into the register bit by bit (serial) and then read the contents after the register is full (parallel). The delay function simply shifts the bits from one end of the shift register to the other, providing a delay equal to the length of the shift register.

Figure.1



SHIFT REGISTER

SOME NOMENCLATURE:

CLOCKING: One of the inputs to a shift register is the clock; a shift occurs in the register when this clock input changes state from one to zero (or from zero to one, depending on the implementation). From this, the term "clocking" has arisen to mean activating a shift of the register. Sometimes the register is said to be "strobed" to cause the shift.

SHIFT DIRECTION: A shift register can shift its contents in either direction depending on how the device is designed. (Some registers have extra inputs that dictate the direction of the shift.) For the purposes of this discussion, the shift direction will always be from left to right.

OUTPUT: During a shift, the bit on the far right end of the shift register is moved out of the register. This end bit position is often referred to as the output bit. To confuse matters a bit, the bits that are shifted out of the register are also often referred to as output bits. To really muddy the waters, every bit in the shift register is considered to be output during a serial to parallel conversion. Happily, the context in which the term "output" is used generally clears things up.

INPUT: After a shift, the bit on the left end of the shift register is left empty unless a new bit (one not contained in the original contents) is put into it. This bit is sometimes referred to as the input bit. As with the output bit, there are several different references to input that are clarified by context.

3.2.1 FEEDBACK ACTION:

In an LFSR, the bits contained in selected positions in the shift register are combined in some sort of function and the result is fed back into the register's input bit. By definition, the selected bit values are collected before the register is clocked and the result of the feedback function is inserted into the shift register during the shift, filling the position that is emptied as a result of the shift.

Feedback around an LFSR's shift register comes from a selection of points (taps) in the register chain and constitutes XORing these taps to provide tap(s) back into the register. Register bits that do not need an input tap, operate as a standard shift register. It is this feedback that causes the register to loop through repetitive sequences of pseudo-random value. The choice of taps determines how many values there are in a given sequence before the sequence repeats. The implemented LFSR uses a one-to-many structure, rather than a many-to-one structure, since this structure always has the shortest clock-to-clock delay path.

The feedback is done so as to make the system more stable and free from errors. Specific taps are taken from the tapping points and then by using the XOR operation on them they are feedback into the registers.

The table for Xor is given below for various inputs:

Input A	Input B	Input C	XOR Output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Table.1 **XOR TRUTH TABLE**

The bit positions selected for use in the feedback function are called "taps". The list of the taps is known as the "tap sequence". By convention, the output bit of an LFSR that is n bits long is the nth bit; the input bit of an LFSR is bit 1

3.2.2TAPPING ACTION:

An LFSR is one of a class of devices known as state machines. The contents of the register, the bits tapped for the feedback function, and the output of the feedback function together describe the state of the LFSR. With each shift, the LFSR moves to a new state. (There is one exception to this -- when the contents of the register are all zeroes, the LFSR will never change state.) For any given state, there can be only one succeeding state. The reverse is also true: any given state can have only one preceding state. For the rest of this discussion, only the contents of the register will be used to describe the state of the LFSR.

A state space of an LFSR is the list of all the states the LFSR can be in for a particular tap sequence and a particular starting value. Any tap sequence will yield at least two state spaces for an LFSR. (One of these spaces will be the one that contains only one state -- the all zero one.) Tap sequences that yield only two state spaces are referred to as maximal length tap sequences.

The state of an LFSR that is n bits long can be any one of 2^n different values. The largest state space possible for such an LFSR will be 2^n - 1 (all possible values minus the zero state). Because each state can have only once succeeding state, an LFSR with a maximal length tap sequence will pass through every non-zero state once and only once before repeating a state.

One corollary to this behavior is the output bit stream. The period of an LFSR is defined as the length of the stream before it repeats. The period, like the state space, is tied to the tap sequence and the starting value. As a matter of fact, the period is equal to the size of the state space. The longest period possible corresponds to the largest possible state space, which is produced by a maximal length tap sequence. (Hence "maximal length")

Table.2 **4-Bit LFSR [4, 1] States and Output**

Register States				
Bit 1 (Tap)	Bit 2	Bit 3	Bit 4 (Tap)	**Output Stream**
1	1	0	1	
0	1	1	0	1
0	0	1	1	0
1	0	0	1	1
0	1	0	0	1
0	0	1	0	0
0	0	0	1	0
1	0	0	0	1
1	1	0	0	0
1	1	1	0	0
1	1	1	1	0
0	1	1	1	1
1	0	1	1	1
0	1	0	1	1
1	0	1	0	1
1	1	0	1	0

MAXIMAL LENGTH TAP SEQUENCES:

LFSR's can have multiple maximal length tap sequences. A maximal length tap sequence also describes the exponents in what is known as a primitive polynomial mod 2.

Example,

a tap sequence of 4, 1 describes the primitive polynomial

x^4 + x^1 + 1.

Finding a primitive polynomial mod 2 of degree n (the largest exponent in the polynomial) will yield a maximal length tap sequence for an LFSR that is n bits long.

There is no quick way to determine if a tap sequence is maximal length. However, there are some ways to tell if one is not maximal length:

1) Maximal length tap sequences always have an even number of taps.
2) The tap values in a maximal length tap sequence are all relatively prime.

A tap sequence like 12, 9, 6, 3 will not be maximal length because the tap values are all divisible by 3.

Discovering one maximal length tap sequence leads automatically to another. If a maximal length tap sequence is described by [n, A, B, C], another maximal length tap sequence will be described by [n, n-C, n-B, n-A]. Thus, if [32, 3, 2, 1] is a maximal length tap sequence, [32, 31, 30, 29] will also be a maximal length tap sequence. An interesting behavior of two such tap sequences is that the output bit streams are mirror images in time.

CHARACTERISTICS OF OUTPUT STREAM:

By definition, the period of an LFSR is the length of the output stream before it repeats. Besides being non-repetitive, a period of a maximal length stream has other features that are characteristic of random streams.

1) Sums of ones and zeroes.

In one period of a maximal length stream, the sum of all ones will be one greater than the sum of all zeroes. In a random stream, the difference between the two sums will tend to grow progressively smaller in proportion to the length of the stream as the stream gets longer. In an infinite random stream, the sums will be equal.

2) Runs of ones and zeroes.

A run is a pattern of equal values in the bit stream. A bit stream like 10110100 has six runs of the following lengths in order: 1, 1, 2, 1, 1, 2. One period of an n-bit LFSR with a maximal length tap sequence will have 2^(n-1) runs (e.g., a 5 bit device yields 16 runs in one period). 1/2 the runs will be one bit long, 1/4 the runs will be 2 bits long, 1/8 the runs will be 3 bits long, etc., up to a single run of zeroes that is n-1 bits long and a single run of ones that is n bits long. A random stream of sufficient length shows similar behavior statistically.

3) Shifted stream.

Take the stream of bits in one period of an LFSR with a maximal length tap sequence and circularly shift it any number of bits less than the total length. Do a bitwise XOR with the original stream. A random stream also shows this behavior.

One characteristic of the LFSR output not shared with a random stream is that the LFSR stream is deterministic. Given knowledge of the present state of the LFSR, the next state can always be predicted.

CHAPTER 4

TESTBENCH IMPLEMENTATION SIMULATION

4.1 INTRODUCTION

The code for implementing the required PRBS is realized by writing VHDL program.
In the program the logic implemented is very simple.
A 16-bit PRBS is realized by shifting the input through the D-flip flops and feed backing the outputs of some registers known as taps again into the first register after passing them through a XOR gate.

4.2 FEATURES

The process of realizing LFSR is carried out by first developing the VHDL code for a D-flip flop. The same D- flip flop code is then called 16 times in the main program code to realize the required LFSR.
In the code for the PRBS tapings are taken so as to get the maximum range of the binary numbers generated.
In the developed code tapings are taken from 1^{st} ,2^{nd} ,4^{th} and 15^{th} taps so as to obtain the maximum length of binary digits produced.
Initially when the reset is kept at zero the outputs of each of the registers is uninitialized and hence the output is uninitialized as well.
However as soon as the reset is made high the output of all the registers start coming out.

A dead lock condition arises in the case when the initial input into the first register as output of the XOR gate are all 0's.Under this condition the output of all the register of the PRBS Generator remains as 0 at all instants of time.
Therefore it is necessary that the initial input to the PRBS Generator be equal to 1, the output of the XOR gate.

4.3 VHDL CODE FOR D-FLIP FLOP

```
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
--library UNISIM;
--use UNISIM.VComponents.all;

entity dff is
   Port ( CLK : in std_logic;
          RSTn : in std_logic;
          D : in std_logic;
          Q : out std_logic);
end dff;

architecture Behavioral of dff is
begin
          process(CLK)
          begin
                if CLK'event and CLK='1' then
                      if RSTn='1' then
                            Q <= '1';
                      else
                            Q <= D;
                      end if;
                end if;
          end process;
end Behavioral;
```

4.4VHDL CODE FOR PRBS

```
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
--library UNISIM;
--use UNISIM.VComponents.all;

entity lfsr is
   Port ( CLK : in std_logic;
          RSTn : in std_logic;
          data_out : out std_logic_vector(15 downto 0));
end lfsr;

architecture Behavioral of lfsr is

component dff
Port ( CLK : in std_logic;
          RSTn : in std_logic;
          D : in std_logic;
          Q : out std_logic);
end component;
signal data_reg : std_logic_vector(15 downto 0);
signal tap_data : std_logic;

begin
          process(CLK)
```

```
        begin
                tap_data <= (data_reg(1) xor data_reg(2)) xor (data_reg(4) xor
data_reg(15));
        end process;
        stage0: dff
                port map(CLK, RSTn, tap_data, data_reg(0));
g0:for i in 0 to 14 generate
                stageN: dff
                        port map(CLK, RSTn, data_reg(i), data_reg(i+1));
        end generate;
        data_out <= data_reg after 3 ns;
end Behavioral;
```

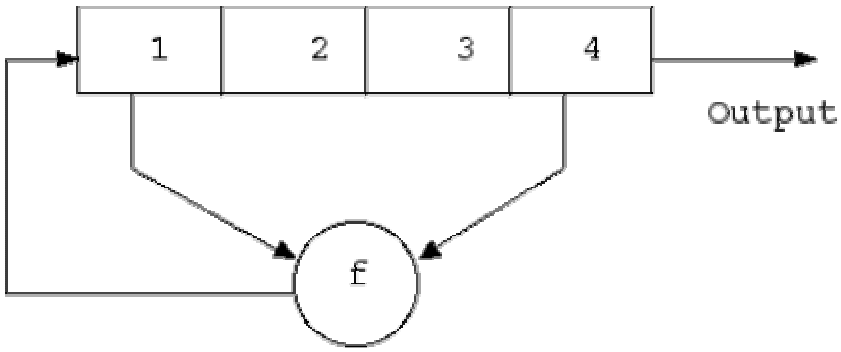


Figure.2 **4-bit PRBS realization with tapings**

4.5 SIMULATION RESULTS

Figure.3 **Schematic diagram of the implemented circuit**

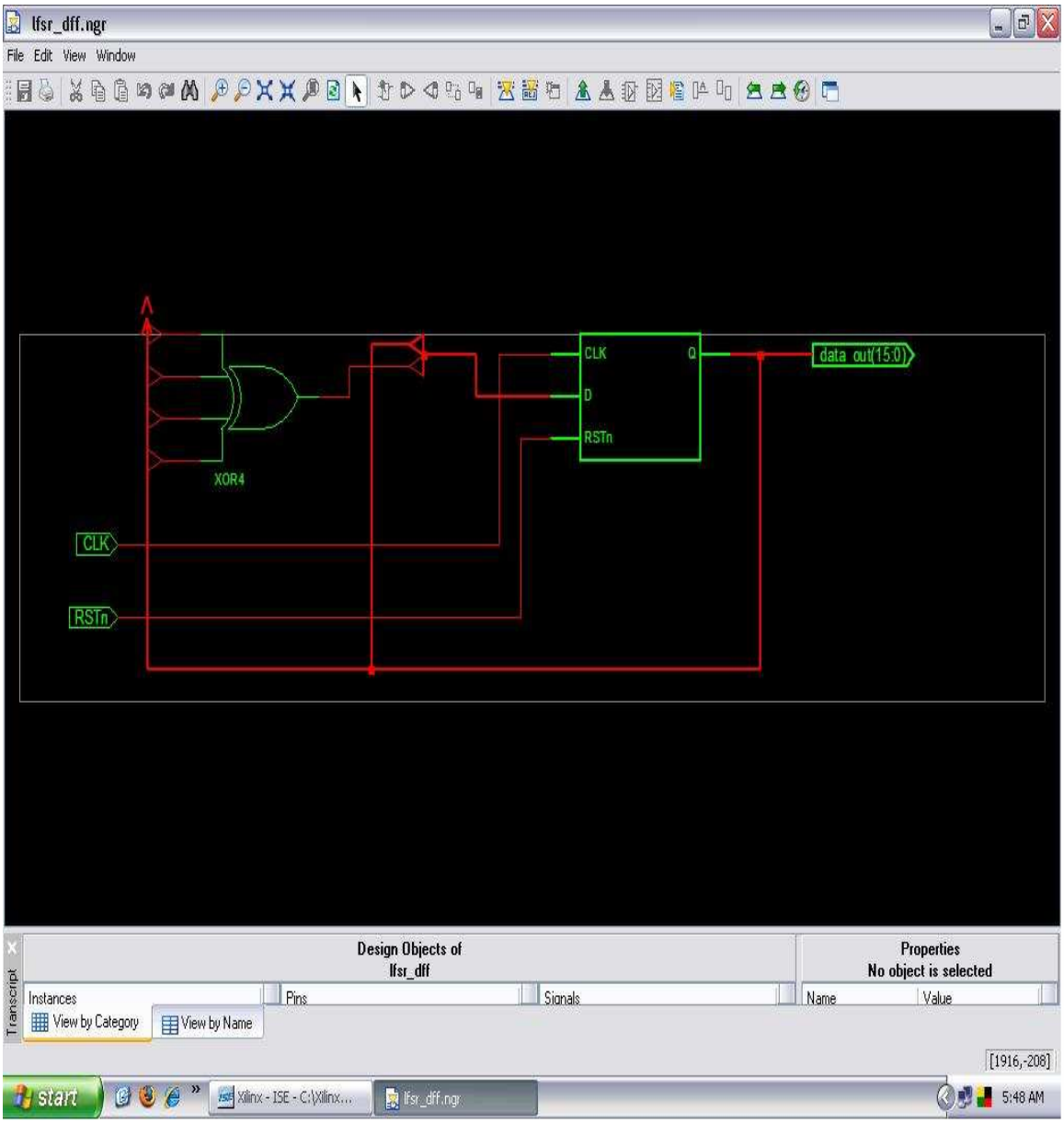
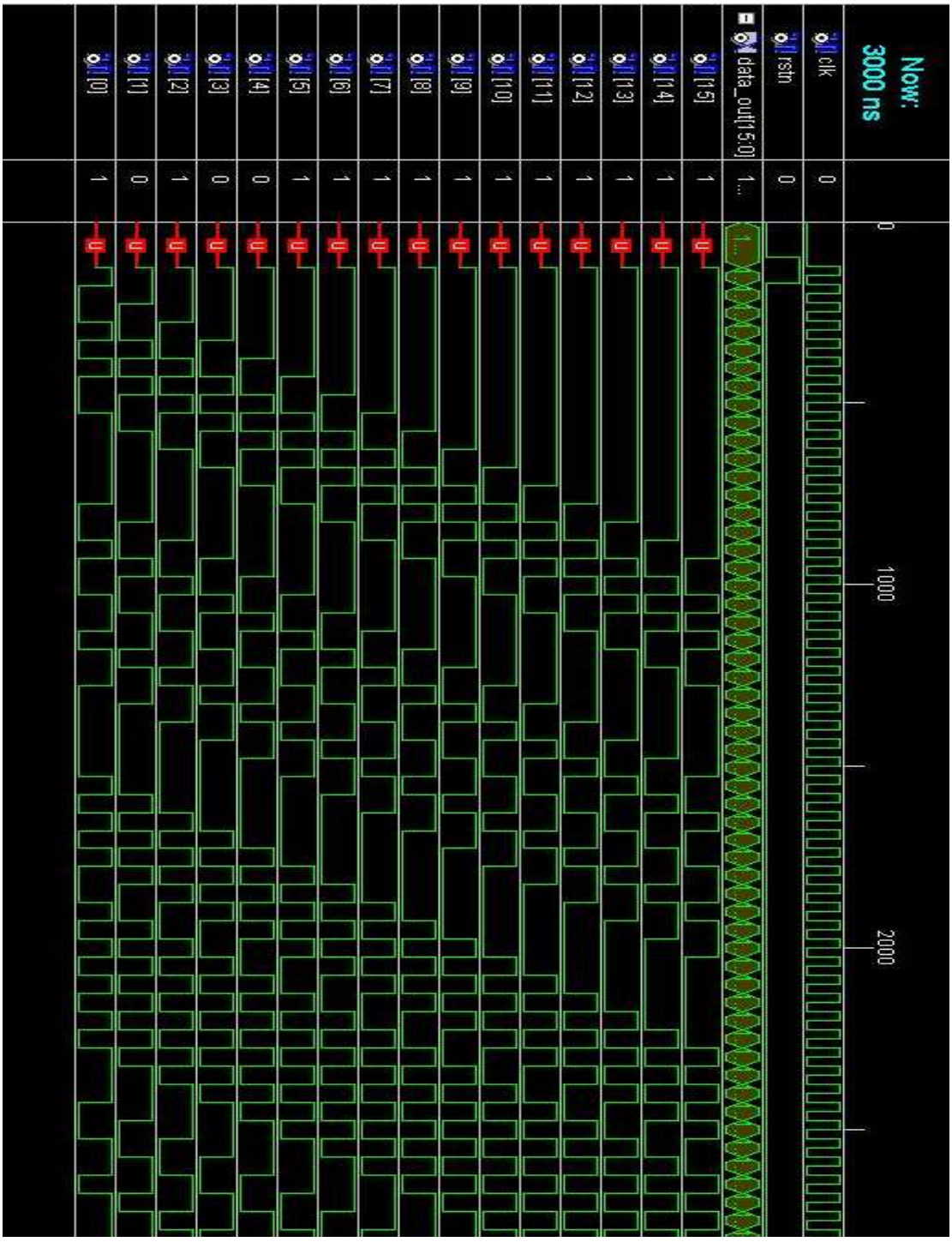


Figure.4 **Simulation result for realized PRBS**



CHAPTER 5

APPLICATIONS

5.1 APPLICATIONS

A Pseudo Random Binary Sequence Generator actually consists of a Linear Feedback Shift Register which is a sequential shift register with combinational logic that causes it to pseudo-randomly cycle through a sequence of binary values. Linear feedback shift registers have multiple uses in digital systems design.

Applications Include:

- Data Encryption/Decryption
- Digital Signal Processing
- Wireless Communications
- Built-in Self Test (BIST)
- Data Integrity Checksums
- Data Compression
- Pseudo-random Number Generation (PN)
- Direct Sequence Spread Spectrum
- Scrambler/Descrambler
- Optimized Counters

A design modeled after LFSRs often has both speed and area advantages over a functionally equivialent design that does not use LFSRs.

5.2 Use as Built in self tester (BIST)

At the heart of this BIST approach, lie a pseudo-random binary sequence (PRBS) generator and a signature register. The PRBS generator is most easily implemented using a linear feedback shift register (LFSR). A PRBS generator allows us to generate all (well, almost all) of the required binary patterns for the circuit under test. The LFSR can be used to both generate the test sequence for the design that is to incorporate BIST and with slight modification can be used to capture the response of the design and generate a signature (the bit pattern held in the signature register).

The signature in the signature register can be compared to a known good signature. Within certain realms of mathematical probablity, if the signature for the circuit being tested is the same as the known good signature, then the tested circuit is deemed as being functionally correct. There is a little maths involved in discovering the known good value for the signature of the circuit being tested but more on that in Part Two. This month we are going to concentrate on the design of an LFSR and one kind of signature register.

The maximal length LFSR generates data that is almost random (hence the term ‘pseudo-random'). The output of the LFSR can be taken in parallel-out form or as a serial bit stream. The serial bit stream is usually taken from the MSB of the LFSR. Given taps 6 and 9, it turns out that the only pattern not generated is all zeroes. It is a fairly simple task to add a little extra circuitry to generate this pattern, but we won't tackle this just yet. Naturally this would give us a RBS generator, not a pseudo to be seen!

5.3 Use in Wireless Communication

One of the most important uses of PRBS comes in wireless communication using CDMA technology.

Here the input signal at the transmitter end is multiplied with a pseudo random binary number generated by PRBS to generate a unique code which identifies itself with that particular user. At the receiver end again the same process of multiplying the input signal with the pseudo random binary number takes place.

The user is identified by the fact that the correlation between the numbers generated for the same user is very high while in the case of other users the generated numbers are orthogonal to each other.

PRBS's application in generating a spread spectrum is also some what similar, where the obtained spectrum is multiplied with the generated pseudo random number.

In all other applications the PRBS generates binary numbers and provides all possible numbers within the given range and hence help in testing for all possibilities.

CONCLUSION

The code for implementing the required PRBS is realized by writing VHDL program.
In the program the logic implemented is very simple.
A 16-bit PRBS is realized by shifting the input through the D-flip flops and feed backing the outputs of some registers known as taps again into the first register after passing them through a XOR gate.

The process of realizing LFSR is carried out by first developing the VHDL code for a D-flip flop. The same D- flip flop code is then called 16 times in the main program code to realize the required LFSR.
In the code for the PRBS tapings are taken so as to get the maximum range of the binary numbers generated.
In the developed code tapings are taken from 1^{st} ,2^{nd} ,4^{th} and 15^{th} taps so as to obtain the maximum length of binary digits produced.
Initially when the reset is kept at zero the outputs of each of the registers is uninitialized and hence the output is uninitialized as well.
However as soon as the reset is made high the output of all the registers start coming out.

A dead lock condition arises in the case when the initial input into the first register as output of the XOR gate are all 0's.Under this condition the output of all the register of the PRBS Generator remains as 0 at all instants of time.
Therefore it is necessary that the initial input to the PRBS Generator be equal to 1, the output of the XOR gate.

The code for implementing the above circuit was written and hence the simulation results were generated and tested.

REFERENCES

1) **The Art of Electronics,** 2nd Edition , Horowitz and Hill, 1989, pp 665-667

2) **P.Alfke,** "Efficient Shift Registers, LFSR , Counters and Long Pseudo-Random Sequence Generators," XAPP 052, July 7, 1996(Version 1.1)

3) **HDL Chip Design,** Douglas J. Smith, Doone Publications, 1996

4) Woody Johnson, Freecore, **Linear Feedback Registers,** 1997

5) **www.Wikipedia.com**