Instituto Tecnológico y de Estudios Superiores de Occidente

Repositorio Institucional del ITESO **rei.iteso.mx**

Departamento de Electrónica, Sistemas e Informática

DESI - Trabajos de fin de Especialidad en Diseño de Sistemas en Chip

2016-07

Test Modules Design for a SerDes Chip in 130 nm CMOS technology

Limones-Mora, César F.

Limones-Mora, C. F. (2016). Test Modules Design for a SerDes Chip in 130 nm CMOS technology. Trabajo de obtención de grado, Especialidad en Diseño de Sistemas de Chip. Tlaquepaque, Jalisco: ITESO

Enlace directo al documento: http://hdl.handle.net/11117/3892

Este documento obtenido del Repositorio Institucional del Instituto Tecnológico y de Estudios Superiores de Occidente se pone a disposición general bajo los términos y condiciones de la siguiente licencia: http://quijote.biblio.iteso.mx/licencias/CC-BY-NC-2.5-MX.pdf

(El documento empieza en la siguiente página)

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Especialidad en Diseño de Sistemas en Chip

Reconocimiento de Validez Oficial de Estudios de nivel superior según Acuerdo Secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976

DEPARTAMENTO DE ELECTRÓNICA, SISTEMAS E INFORMÁTICA

Test Modules Design for a SerDes Chip in 130 nm CMOS technology

Tesina para obtener el grado de:

Especialista en Diseño de Sistemas en Chip

Presenta:

César Fernando Limones Mora

Asesores:

Víctor Avendaño Fernández, Alexandro Girón Allende

Guadalajara, Jalisco, Julio 2016

Acknowledgements

This thesis is dedicated to the memory of Socorro Mora. You shan't be forgotten.

I would like to express my gratitude and appreciation to my tutors A. Giron and V. Avendaño. They were a source of support, guidance and encouragement.

My appreciation to all my colleagues in the specialty program who were of great help and support. In particular I would like to thank R. Rivas, E. Conde, E. Arrambide and J. Nuñez.

My deep gratitude to my parents and to my brother for their much needed support, patience, understading and encouragement. No words of appreciation could ever rewards them for all they have done for me.

Finally, I would like to thank CONACYT for the financial support given to this project. Without CONACYT contribution this wouldn't be possible.

Abstract

Over the last decades, the semiconductor technology has been advancing exponentially, creating many challenges in circuit testing. As a result, design-for-testability (DFT) and built-in-self-test (BIST) techniques are becoming essential parts of any high-speed VLSI design.

This thesis presents a DFT architecture for testing a high-speed *SerDes* circuit. In addition to the default functional mode of the *SerDes*, such architecture proposes the use of eight different operating modes specifically designed for testing the circuit. Three test modules were designed to make this possible: a comparator, a linear-feedback shift register (LFSR) and a signal driver.

The physical and logic synthesis for these modules were performed for a 130nm CMOS implementation using ARM standard cells library for **IBM**´s cmrf8sf Design Kit Process. Simulation results show the correct behavior of all the operating modes. Synthesis results show timing compliance for a maximum frequency of 111 MHz and no DRC, geometry or connectivity violations.

Table of Contents

Acknowledgements I
Abstract II
Table of Contents III
Table of Figures VI
List of Tables VIII
Introduction 1
Chapter 1: Background of a *SerDes* system and testing techniques 3
1.1. *SerDes* Overview 3
1.2. PCI Express communication protocol 5
1.2.1. PCI Express Features 5
1.2.1.1. The Link - A Point-to-Point Interconnect 5
1.2.1.2. Differential Signaling 6
1.2.1.3. Bandwidth and Clocking 6
1.3. 8B/10B Coding Scheme 6
1.4. VLSI Testing 8
1.5. Automatic Test Pattern Generation 9
1.6. Built-In Self-Test 9
1.7. Linear Feedback Shift Register 10
1.8. Comparator 11
1.9. *SerDes* system specifications 13
1.10. *SerDes* architecture 13
Chapter 2: Test architecture for a *SerDes* system 17
2.1. Background 17
2.2. Test modules 18
2.2.1. The comparator 21
2.2.2. The LFSR 27
2.2.3. The signal driver 30
2.3. Operating modes 32
2.3.1. Operation mode 0 A - Functional mode 34
2.3.2. Operation mode 0 B – Functional mode – Test enabled 37
2.3.3. Operation mode 1 – Parallel loopback 38
2.3.4. Operation mode 2 – Serial loopback 39

2.3.5. Operation mode 3 – RXA bypass 40
2.3.6. Operation mode 4 – BIST with serial loopback 41
2.3.7. Operation mode 5 – RXA bypass with parallel loopback 42
2.3.8. Operation mode 6 – Open BIST 44
2.3.9. Operation mode 7 – RXA output with analog loopback 45
Chapter 3: Logic synthesis and Gate Level Simulation 47
3.1. Logic Synthesis 47
3.1.1. Gates summary report 52
3.1.2. Power report 52
3.1.3. Gate Level Simulation 53
3.1.3.1. GLS - Operation Mode 0 A - Functional mode 54
3.1.3.2. GLS - Operation Mode 0 B – Functional mode – Test enabled 55
3.1.3.3. GLS - Operation Mode 1 – Parallel loopback 55
3.1.3.4. GLS - Operation Mode 2 – Serial loopback 56
3.1.3.5. GLS - Operation Mode 3 – RXA bypass 56
3.1.3.6. GLS - Operation Mode 4 – BIST with serial loopback 57
3.1.3.7. GLS - Operation Mode 5 – RXA bypass with parallel loopback 57
3.1.3.8. GLS - Operation Mode 6 – Open BIST 58
3.1.3.9. GLS - Operation Mode 7 – RXA Output with analog loopback 59
Chapter 4: Physical synthesis and Layout Verification 60
4.1. Physical synthesis 60
4.1.1. Connectivity, geometry and DRC report. 61
4.1.2. Timing analysis 63
4.1.3. Layout generation 65
4.1.4. GDS stream import to Virtuoso 66
4.2. Layout Verification (DRC) 67
Conclusions 73
References 75
Appendix 77
7.1. Table: archives of test module 77
7.2. Glossary 78
7.3. RTL Codes 79
7.3.1. SerDes.sv 79
7.3.2. analog_receiver.sv 82

7.3.3. analog_transmitter_wrap.sv 83
7.3.4. analog_transmitter.sv 85
7.3.5. clock_divider.sv 86
7.3.6. digital_receiver.sv 86
7.3.7. decode.sv 88
7.3.8. deserializer.sv 92
7.3.9. digital_transmitter.sv 95
7.3.10. encode.sv 96
7.3.11. serializer.sv 99
7.3.12. test_modules.sv 101
7.3.13. comparator.sv 104
7.3.14. lsfr.sv 108
7.3.15. signal_driver.sv 110
7.1. Tesbench Codes 113
7.1.1. SerDes_m0_tb.sv 113
7.1.2. SerDes_m0B_tb.sv 115
7.1.3. SerDes_m1_tb.sv 117
7.1.4. SerDes_m2_tb.sv 119
7.1.5. SerDes_m3_tb.sv 122
7.1.6. SerDes_m4_tb.sv 124
7.1.7. SerDes_m5_tb.sv 126
7.1.8. SerDes_m6_tb.sv 128
7.1.9. SerDes_m7_tb.sv 130
7.2. Synthesis files 132
7.2.1. Logic_synthesis.tcl 132
7.2.2. Full_Synthesis_EDI_test_modules.tcl 139

Table of Figures

Figure 1 - Serial vs Parallel Data Transmission [1] ... 3
Figure 2 - The Serialization process [1] ... 4
Figure 3 - PCI Express Link [3] ... 6
Figure 4 - PCI Express Differential Signal [3] ... 6
Figure 5 - The 8b/10b Encoder/Decoder in a System [5] ... 8
Figure 6 - A typical BIST architecture [9] ... 10
Figure 7 - Generic Standard LFSR [9] ... 11
Figure 8 - Generic Standard Comparator [9] ... 12
Figure 9 - SerDes structure ... 15
Figure 10 - SerDes Top Level ... 15
Figure 11 - Test_modules top view ... 19
Figure 12 - Comparator top module ... 22
Figure 13 - Comparator schematic ... 23
Figure 14 - Comparator FSM diagram ... 24
Figure 15 - Comparator Simulation Start ... 26
Figure 16 - Comparator Simulation End ... 27
Figure 17 - LFSR top module ... 28
Figure 18 - LFSR Simulation ... 29
Figure 19 - Signal_driver top module ... 31
Figure 20 - SerDes top Module ... 33
Figure 21 - Mode 0 - Functional mode ... 36
Figure 22 - Mode 0 - Functional mode – Simulation ... 36
Figure 23 - Mode 0 B – Functional mode – Test enabled ... 37
Figure 24 - Mode 0 B - Functional mode – Test enabled - simulation ... 38
Figure 25 - Mode 1 - Parallel Loopback ... 38
Figure 26 - Mode 1 - Parallel Loopback - Simulation ... 39
Figure 27 - Mode 2 - Serial Loopback ... 39
Figure 28 - Mode 2 - Serial Loopback - Simulation ... 40
Figure 29 - Mode 3 - RXA Bypass ... 40
Figure 30 - Mode 3 - RXA Bypass - Simulation ... 41
Figure 31 - Mode 4 -BIST with serial loopback ... 42
Figure 32 - Mode 4 -BIST with serial loopback - Simulation ... 42
Figure 33 - Mode 5 - RXA bypass with parallel loopback ... 43
Figure 34 - Mode 5 - RXA bypass with parallel loopback - Simulation ... 44
Figure 35 - Mode 6 - Open BIST ... 44
Figure 36 - Mode 6 - Open BIST - Simulation ... 45
Figure 37 - Mode 7 - RXA Output with analog loopback ... 46
Figure 38 - Mode 7 - RXA Output with analog loopback - Simulation ... 46
Figure 39 - Logic Synthesis flow ... 48
Figure 40 - Test_modules RTL Compiler schematic ... 49
Figure 41 - Comparator RTL Compiler Schematic ... 50
Figure 42 - LFSR RTL Compiler Schematic ... 50
Figure 43 - Signal Driver RTL Compiler Schematic ... 51
Figure 44 - Gate-level simulation flow ... 53
Figure 45 - GLS resetting sequence ... 54
Figure 46 - GLS - Mode 0 A - Functional mode ... 55
Figure 47 - GLS - Operation Mode 0 B – Functional mode – Test enabled ... 55
Figure 48 - GLS - Operation Mode 1 – Parallel loopback ... 56
Figure 49 - GLS - Operation Mode 2 – Serial loopback ... 56
Figure 50 - GLS - Operation Mode 3 – RXA bypass ... 57

Figure 51 - GLS -Operation Mode 4 – BIST with serial loopback ... 57
Figure 52 - GLS - Operation Mode 5 – RXA bypass with parallel loopback ... 58
Figure 53 - GLS - Operation Mode 6 – Open BIST - Unconnected ... 58
Figure 54- GLS - Operation Mode 6 – Open BIST - Analog receiver set to 1 ... 59
Figure 55 - GLS - Operation Mode 7 – RXA Output with analog loopback ... 59
Figure 56 - Connectivity Verification ... 62
Figure 57 - Geometry Verification ... 62
Figure 58 - DRC verification ... 63
Figure 59 - Setup timing ... 64
Figure 60 - Example of timing violating path ... 65
Figure 61 - Setup timing 111 MHz ... 65
Figure 62 - Hold Timing 111 MHz ... 65
Figure 63 - test_modules preliminary layout ... 66
Figure 64 - GDS Virtuoso Stream ... 67
Figure 65 - Calibre Errors ... 68
Figure 66 - Diode to fix DRC antenna errors ... 68
Figure 67 - NW spacing DRC errors ... 69
Figure 68 - Filled space of NW ... 69
Figure 69 - Latch-Up DRC errors ... 69
Figure 70 - Latch-Up Error Fix Example ... 70
Figure 71 - M3 - NW Ratio DRC errors ... 71
Figure 72 - Fixing M3- NW Ratio errors Example ... 71

List of Tables

TABLE 1 - SERDES SPECIFICATIONS 13
TABLE 2 - TOOLS AVAILABLE 13
TABLE 3 - EXTERNAL INPUTS AND OUTPUTS OF THE SERDES 16
TABLE 4 - INPUTS AND OUTPUTS OF THE TEST_MODULES BLOCK 20
TABLE 5 - TEST_EN MODIFIER 21
TABLE 6 - ERRORS_EN MODIFIER 21
TABLE 7 - INPUTS AND OUTPUTS OF THE COMPARATOR 24
TABLE 8 - COMPARATOR FSM TRANSITION TABLE 25
TABLE 9 - LFSR INPUTS AND OUTPUTS 29
TABLE 10 - INPUTS AND OUTPUTS OF THE SIGNAL_DRIVER 32
TABLE 11 - MODES DESCRIPTION 34
TABLE 12 - GATES SUMMARY 52
TABLE 13 - POWER REPORT 52

Introduction

SerDes circuits are widely spread across high speed communication systems used in today’s industry. The *SerDes* term comes from Serializer/Deserializer and the main function of this device is to receive serial binary data and convert it into parallel and vice versa.

The current project is the design, testing and manufacturing of a *SerDes* system for PCI express generation 1 communications protocol. This *SerDes* implementation will use a CMOS 130nm technology with ARM standard cells in **IBM**´s cmrf8sf Design Kit.

The final *SerDes* circuit should be able to be tested once it’s manufactured, for achieving this, the system has to be designed using design-for-testability (DFT) techniques such as built-in-self-test (BIST).

In this thesis, a DFT architecture is proposed for testing this high-speed *SerDes* circuit. This architecture suggests the implementation of three testing modules: a comparator, a linear-feedback shift register (LFSR) and a signal driver. By embedding these test modules to the *SerDes*, the circuit will be able to perform eight different operating modes for testing in addition to the functional mode. With these operating modes, the functionality of all the modules individually and collectively can be tested by the use of multiplexers and BIST.

Throughout this thesis, figures containing diagrams or waveforms will be shown to explain how a functional block works and its verification results. To understand these figures better, the signals of the circuits will be wrapped between brackets like: “{*signal_name*}”. This will help to facilitate the location of the signals that the text is referring to.

The remainder of this thesis is organized as follows:

Chapter 1 provides the background of a SerDes system and testing techniques. This chapter gives details on the available technology, an overview the *SerDes* concepts and the description of the basic SerDes architecture designed. It also gives a concise review of VLSI testing, as well as some testing concepts and methodologies that were applied for designing the test modules.

Chapter 2 introduces the proposed test architecture for the *SerDes* and its general specifications. It illustrates how the test modules were implemented giving a detailed description and simulation results for each one of them. It also contains on the description and simulation results of all the test operating modes that are proposed in this thesis.

Chapter 3 provides details on the process of performing the logic synthesis of the test modules as well as some results obtained during this process. It also presents the simulation results using the Gate level models of the test modules for all the test operating modes.

Chapter 4 contains the details of the physical synthesis and layout verification. It contains details of the steps to follow to complete the physical synthesis flow of the test modules. It also includes the DRC verification results providing details of how the DRC errors were fixed.

This work concludes with the conclusions of the research described in the thesis. The aim and objectives of the research are reviewed and their achievement addressed. Proposals for future work indicated by the research are suggested.

Chapter 1: Background of a *SerDes* system and testing techniques

The current chapter will provide a general background of the key concepts that are relevant for this work as well as a description of the project architecture and specification.

1.1. *SerDes* Overview

The term *SerDes* comes from Serializer/Deserializer, it is a transceiver device whose main function is to convert parallel data to serial data and vice versa (Figure 1). The transmitter section is a serial-to-parallel converter, and the receiver section is a parallel-to-serial converter. The *SerDes* can be either a separate device or, in most cases, an IP core embedded in a serial bus controller or an ASIC [1].

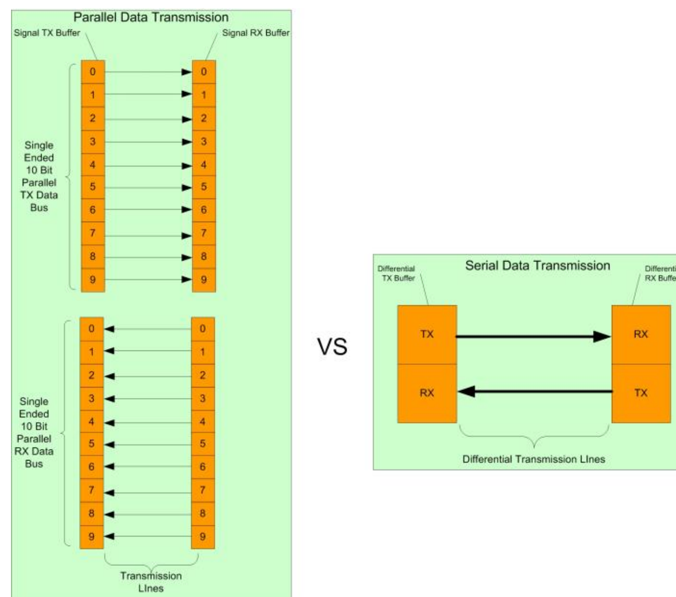

Figure 1 - Serial vs Parallel Data Transmission [1]

Most of *SerDes* devices are capable of operating in full duplex, meaning that data conversion may take place in both directions simultaneously. The *SerDes* systems are widely used in today's industry; they are used for instance in Gigabit Ethernet systems, routers, wireless networking systems, optical fiber communications, and storage applications. Specifications and speeds vary depending on the needs of the user and the application.

The use of *SerDes* devices facilitate transmission of a large amount of data between two points, while reducing the complexity, cost, energy, and the use of board space associated when having to implement wide parallel data buses.

The basic operation of a *SerDes* is relatively simple. The following figure (Figure 2) is a High level description of the signal flow in a *SerDes* device. A parallel data bus, switched to a particular frequency, is driven to the parallel interface of the *SerDes*, synchronizing the data with the rising or falling edge of the input clock, even though there are some *SerDes* modules that get the input clock signal form the data managed. This design is considered to have a dedicated terminal for both clocks at the input of the serializer and output of the de-serializer.

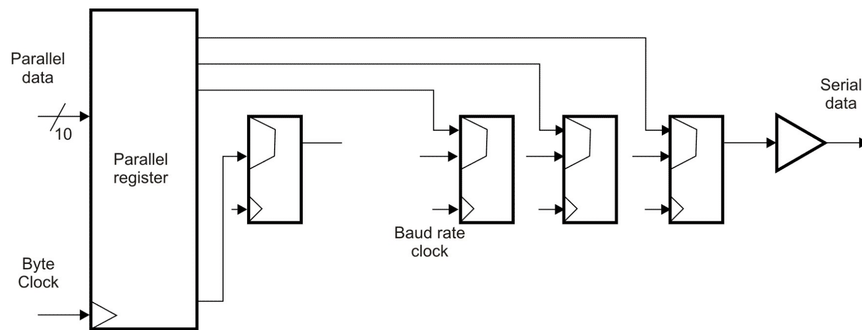

Figure 2 - The Serialization process [1]

Once the data has been loaded into the registers of the serializer, the bits are typically coded using standard coding schemes such as coders known as 8b10b.

The bus of coded data bits is then serialized and converted from a parallel bus to a serial bus. The function of serialization of a *SerDes* takes a parallel set of bits and serializes it for efficient transmission over a single differential transmission channel.

The serialized bus is then fed to a differential line driver, also known as differential signal buffer. The signal buffer drives the serialized bit stream out in a media such as copper wire.

1.2. PCI Express communication protocol

The *SerDes* system implemented in this thesis will use the *Peripheral Component Interconnect Express* protocol (PCI Express), generation 1, which is one of the standard protocols for serial communications between electronic devices. The basic concepts and specifications of PCI Express is introduced here, especially that related to the protocol applied to this design.

PCI Express, officially abbreviated as PCIe, is a computer expansion card standard designed to replace the older PCI, PCI-X, and AGP standards. Introduced by Intel in 2004, PCIe (or PCI-E, as it is commonly called) is the latest standard for expansion cards that is available on mainstream personal computers.

PCI Express is used in consumer, server, and industrial applications, both as a motherboard level interconnect (to link motherboard-mounted peripherals) and as an expansion card interface for add-in boards [2]. A key difference between PCIe and earlier PC buses is a topology based on point-to-point serial links, rather than a shared parallel bus architecture.

1.2.1. PCI Express Features

PCI Express provides a high-speed, high-performance, point-to-point, dual simplex, differential signaling Link for interconnecting devices. Data is transmitted from a device on one set of signals, and received on another set of signals [3].

1.2.1.1. The Link - A Point-to-Point Interconnect

As shown in Figure 3, a PCI Express interconnect consists of either a x1, x2, x4, x8, x12, x16 or x32 point-to-point Link [3]. A PCI Express Link is the physical connection between two devices. A Lane consists of signal pairs in each direction. An x1 Link consists of 1 Lane or 1 differential signal pair in each direction for a total of 4 signals. An x32 Link consists of 32 Lanes or 32 signal pairs for each direction for a total of 128 signals. The Link supports a symmetric number of Lanes in each direction. During hardware initialization, the Link is initialized for Link width and frequency of operation automatically by the devices on opposite ends of the Link. No OS or firmware is involved during Link level initialization.

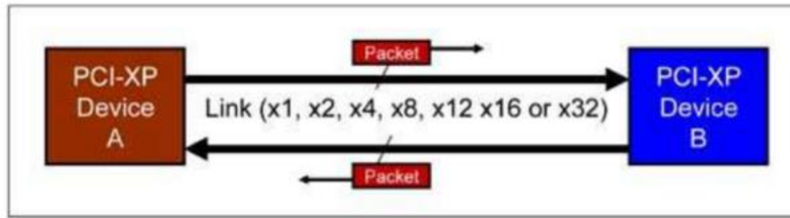

Figure 3 - PCI Express Link [3]

1.2.1.2. Differential Signaling

PCI Express devices employ differential drivers and receivers at each port. Figure 4 shows the electrical characteristics of a PCI Express signal. A positive voltage difference between the D+ and D- terminals implies Logical 1. A negative voltage difference between D+ and D- implies a Logical 0. No voltage difference between D+ and D- means that the driver is in the high-impedance tristate condition, which is referred to as the electrical-idle and low-power state of the Link.

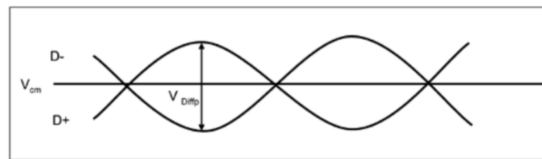

Figure 4 - PCI Express Differential Signal [3]

1.2.1.3. Bandwidth and Clocking

The aggregate bandwidth achievable with PCI Express is significantly higher than any bus available today. The PCI Express 1.0 specification supports 2.5 Gbits/sec/lane/direction transfer rate [3].

No clock signal exists on the Link. Each packet to be transmitted over the Link consists of bytes of information. Each byte is encoded into a 10-bit symbol. All symbols are guaranteed to have one-zero transitions. The receiver uses a PLL to recover a clock from the 0-to-1 and 1-to-0 transitions of the incoming bit stream.

1.3. 8B/10B Coding Scheme

The 8B/10B coding scheme is frequently used in communication systems to ensure sufficient data transitions for clock recovery. It finds its applications in PCI express, Serial ATA, USB 3.0, Fiber Channel, 33A and many more [4]. It was initially proposed by Albert X. Widmer and Peter A. Franaszek and published in IBM Journal of research and development

in the year 1983. The 8B/10B encoder is used to generate sample data transition for facilitating a clock recovery function on the various networks; also, it provides a DC balance by trying to equalize the number of “0” and “1” in the data stream.

The primary purpose of this scheme is to embed a clock into the serial bit stream transmitted on all Lanes. No clock is therefore transmitted along with the serial data bit stream. This eliminates the need for a high frequency clock signal which would generate significant noise and would be a challenge to route on a standard board. Link wire routing between two ports is much easier given that there is no clock to route, removing the need to match clock length to Lane signal trace lengths. Two devices are connected by simply wiring their lanes together.

The encoder on the transmitter side maps the 8-bit parallel data input to 10-bit output. This 10-bit output is then loaded in and shifted out through a high-speed Serializer (Parallel-in Serial-out 10-bit Shift Register). The serial data stream will be transmitted through the transmission media to the receiver. The high-speed Deserializer (Serial-in Parallel-out 10-bit Shift Register) on the receiver side converts the received serial data stream from serial to parallel. The decoder will then remap the 10-bit data back to the original 8-bit data. When the 8b/10b coding scheme is employed, the serial data stream is DC-balanced and has a maximum run-length without transitions of 5. These characteristics aid in the recovery of the clock and data at the receiver [5]. Figure 5 shows the 8b/10b encoder/decoder usage in a communication system.

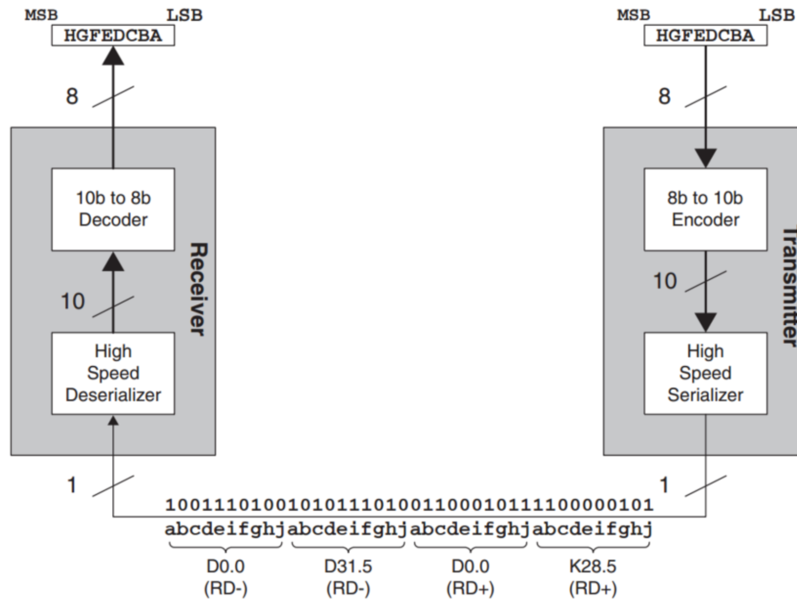

Figure 5 - The 8b/10b Encoder/Decoder in a System [5]

A DC-balanced serial data stream means that it has the same number of 0s and 1s for a given length of data stream. DC-balance is important for certain media as it avoids a charge being built up in the media [5].

The 8b/10b encoder submits an 8-bit character along with a signal to indicate whether the character is a Data (D) or Control (K) character. The PCI Express specification defines Control characters that encode into the following Control symbols: STP, SDP, END, EDB, COM, PAD, SKP, FTS, and IDL [6]. These Control symbols can easily be detected by the receiver logic in an incoming symbol stream.

On this thesis, only the COM, “comma” symbol is implemented. The COM character is used as the first character of any transmission. The 10-bit encoding of the COM character contains two bits of one polarity followed by five bits of the opposite polarity (001111 1010 or 110000 0101), thereby making it easy to detect at the receiver's point. A receiver detects the COM pattern to detect the start of stream transmission.

1.4. VLSI Testing

The purpose of testing a VLSI device is to ensure, with reasonable confidence that the device functions according to the design specifications. This testing must be achievable within certain economic constraints to keep the cost per device as low as possible [7].

In the past two decades, the cost of testing integrated circuits in high-volume manufacturing has been steadily increasing. It is predicted that the cost of testing transistors may actually surpass the cost of fabricating them within the next two decades [8]. As ICs become more highly integrated, the job of diagnosing failures becomes increasingly difficult. This is why it is very essential to use advanced test techniques that enables the testing process to cope with the advances in semiconductor technology.

1.5. Automatic Test Pattern Generation

Digital systems are tested by applying appropriate stimuli and checking the responses. Generation of such stimuli together with calculation of their expected responses is called test pattern generation. Test patterns are typically generated by an automatic test pattern generator and applied to the circuit externally. Due to several limitations of this method, there exist approaches where the main functions of the external tester have been moved onto the chip. Such DFT practice is generally known as Built-In-Self-Test (BIST) [9].

1.6. Built-In Self-Test

The main idea behind a BIST approach is to eliminate or reduce the need for an external tester by integrating active test infrastructure onto the chip. The test patterns are not any more generated externally, but internally, using special BIST circuitry. BIST techniques can be divided into offline and on-line techniques [9]. On-line BIST is performed during normal functional operation of the chip, either when the system is in idle state or not. Off-line BIST is performed when the system is not in its normal operational mode but in special test mode.

A typical BIST architecture consists of a test pattern generator (TPG), a test response analyzer (TRA), and a BIST control unit (BCU), all implemented on the chip (Figure 6). Examples of TPG are a ROM with stored patterns, a counter, or a LFSR. A typical TRA is a comparator with stored responses or an LFSR used as a signature analyzer. A BCU is needed to activate the test and analyze the responses. This approach eliminates virtually the need for an external tester. Equipping the cores with BIST features is especially preferable if the modules are not easily accessible externally, and it helps to protect intellectual property (IP) as less information about the core has to be disclosed [9].

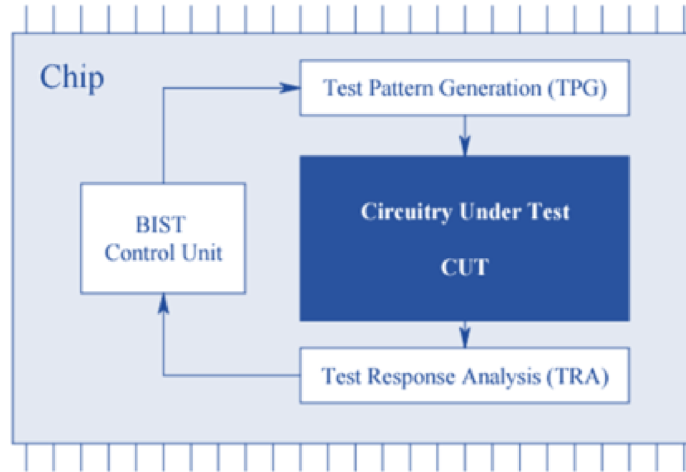

Figure 6 - A typical BIST architecture [9]

In the architecture presented in this thesis for the *SerDes* system. Modules for TPG, TRA and BCU were developed. As a test pattern generator, a LFSR was implemented. As a test response analyzer, a comparator of the sent and received data was designed. And finally, a signal driver module performs the work of a BIST Control Unit.

1.7. Linear Feedback Shift Register

Typical BIST schemes rely on either exhaustive, pseudo exhaustive, or pseudorandom testing and the most relevant approaches use LFSRs for test pattern [9]. This is mainly due to the simple and fairly regular structure of the LFSR. The LFSR generated tests are much easier to generate and have good pseudorandom properties.

Figure 7 shows the generic structure of the n-stage standard LFSR. An LFSR is a shift register, composed from memory elements (latches or flip-flops) and exclusive OR (XOR) gates, with feedback from different stages. It is fully autonomous, i.e. it does not have any input beside the clock. C_i in Figure 7 denotes a binary constant and if C_i = 1 then there is a feedback from/to the ith D flip-flop; otherwise, the output of this flip-flop is not tapped and the corresponding XOR gate can be removed. The outputs of the flip-flops ($Y_1, Y_2 \dots Y_N$) form the test pattern. The number of unique test patterns is equal to the number of states of the circuit, which is determined, by the number and locations of the individual feedback tabs.
The configuration of the feedback tabs can be expressed with a polynomial, called characteristic or feedback polynomial. For an LFSR in Figure 6 the characteristic polynomial is:

$$P_{(x)} = 1 + c_1 x + c_2 x + \cdots + c_n x^n.$$

An LFSR goes through a cyclic or periodic sequence of states and produces periodic output. The maximum length of this period is 2^n -1, where *n* is the number of stages, and the characteristic polynomials that cause an LFSR to generate maximum-length sequences are called primitive polynomials. In the LFSR implemented in this thesis, a total of 62 states are generated before restarting the sequence.

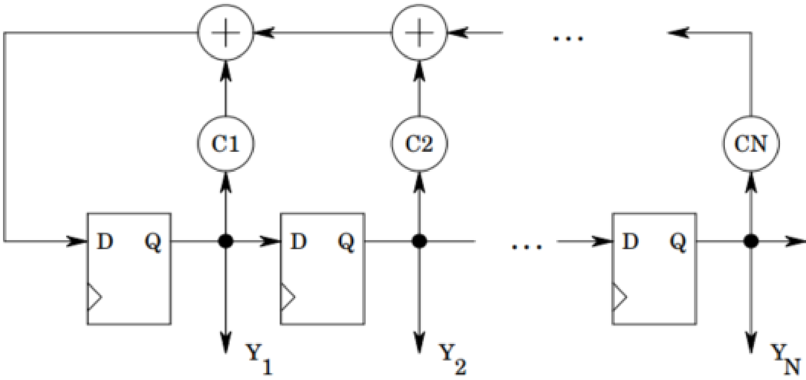

Figure 7 - Generic Standard LFSR [9]

In serial BIST, deterministic patterns are encoded into smaller vectors (aka seeds) that are loaded into the LFSR. These seeds can be used to initialize the circuit to a certain pattern [10]. In the LSFR module presented in this thesis, the initial seed is set to a "comma" symbol in a way that the transmission of the data can start on every reset of the BIST sequence.

The test vectors generated by an LFSR appear to be randomly ordered. They satisfy most of the properties of random numbers even though we can predict them deterministically from the LFSR's present state. Thus, these vectors are called pseudorandom vectors and such LFSRs can be called pseudorandom pattern generator (PRPG).

1.8. Comparator

A comparator is basically comprised of 2 memory blocks that are compared against each other and output if their data is equal or not. The first memory block will normally be loaded with all the values that this digital block is expected to have [11]. Figure 8, shows represents the structure of a basic comparator.

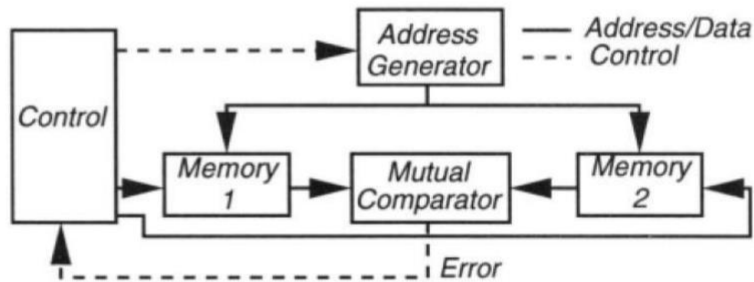

Figure 8 - Generic Standard Comparator [9]

The *comparator* proposed in this thesis will perform as a Test Response Analyzer (TRA) module for the BIST. It will store the input parallel data in a memory block and, with the aid of a controller FSM, synchronize this data with the output data of the deserializer and compare it on the fly. This implementation, at difference than the one presented in Figure 8, will only use one memory block for storing the data, reducing the area and logic for this module.

In offline and in non-concurrent online BIST, the normal operation of the CUT is stalled in order to perform the test. However if the failures cannot be detected during the normal operation of the circuit, the circuit performance will be degraded [12].

The test modules presented in this thesis are working in such way that they can be used for both off-line and on-line testing. By carrying out testing concurrently with the normal operation of the *SerDes* we can provide the circuit with enhanced diagnostic capability. Note that the off-line testing capability of the BIST resources are still maintained in the *SerDes*.

During both the online mode and offline modes of the *SerDes*, the parallel inputs that are driven into the serializer, are also driven into the comparator. When a “comma” symbol is transmitted, the *comparator* will start flagging errors between the transmitted and the sent data until a second “comma” symbol is transmitted. This way, even in a functional mode, the *SerDes* will be flagging mismatches between the data transmitted in parallel against the received de-serialized data.

1.9. *SerDes* system specifications

The specifications for the *SerDes* system developed are shown in the following Table 1.

Specifications	
Manufacturing technology	CMOS 130nm
Communication protocol	PCIE 1.0
VDD	1.2V
VSS	0V
Area	1.5 mm X 1.5 mm
Clock frequency	1.25 GHz
Packaging	DIP40 chip

Table 1 - SerDes Specifications

The tools available for designing the chip are given in the following Table 2.

Description	Tool
Schematic and Layout edition, analysis and simulation Tool.	Cadence Virtuoso
RTL simulation and debugging tool.	Quartus Prime 15.1 Lite Edition and Model Sim
Standard Cells.	ARM standard cells library for 130 nm IBM´s cmrf8sf Design Kit.
RTL debugging and Logic synthesis generation tool.	Cadence RTL Compiler
Physical synthesis generation tool.	Cadence Encounter

Table 2 - Tools Available

1.10. *SerDes* architecture

This *SerDes* system has three main stages, transmission, reception and testing. Both the transmission and the reception stages are composed by an analog and a digital block, while the testing stage is entirely digital.

The transmission stage function is to receive a 9-bit parallel data, encode it to 8b/10b and transmit a serialized data for the PCI express protocol.

The digital block of the transmission is capable of converting a parallel data bus into serial

data format. It encodes the data using 8b/10b codification, meets the specifications of speed and provides a transmission clock signal to synchronize the circuit on when to send the data. It is composed by two modules: a serializer and an encoder.

The analog block of the transmission includes a driver strength selection and an equalization circuit to implement pre-emphasis and de-emphasis to account for channel loss.

The reception stage receives a serial differential input data encoded in 8b/10b and will provide an output of a parallel digital decoded 9 bit data.

The digital block of the reception is in charge of converting the received serial data to parallel data and align it with a system clock. It proposes a recovery scheme based on the clock and sampling data. As the received data is encoded, it also performs the 8b/10b decoding process while meeting the specifications of speed. It is composed by three modules: a deserializer, an encoder and a clock divider.

The analog block of the reception aims to compensate for the attenuation in amplitude experienced by the transmission of the serial data through the communication channel. It also has a bias circuit to achieve compensation of voltage and temperature.

The testing stage adds verification capabilities to the chip´s architecture by using control signals and design-for-testability (DFT) techniques that will allow to test the functionality of the final *SerDes* circuit once it’s manufactured. It is composed by three modules: a comparator, a linear-feedback shift register (LFSR) and a signal driver. These modules will be explained in detail in Chapter 4.4.

The *SerDes* structure is presented in Figure 9.

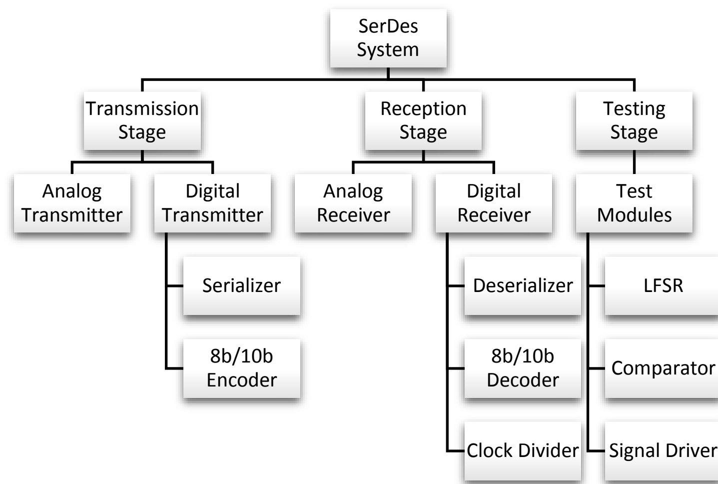

Figure 9 - SerDes structure

The following diagram on Figure 10 shows the modules that conform the architecture for the *SerDes* circuit as well as the *SerDes* inputs and outputs. In total we are allowed to have a maximum of 40 pins in the final circuit, so multiplexers were used in order to reduce the number of pins. How these selectors work will be detailed in 2.2.3.

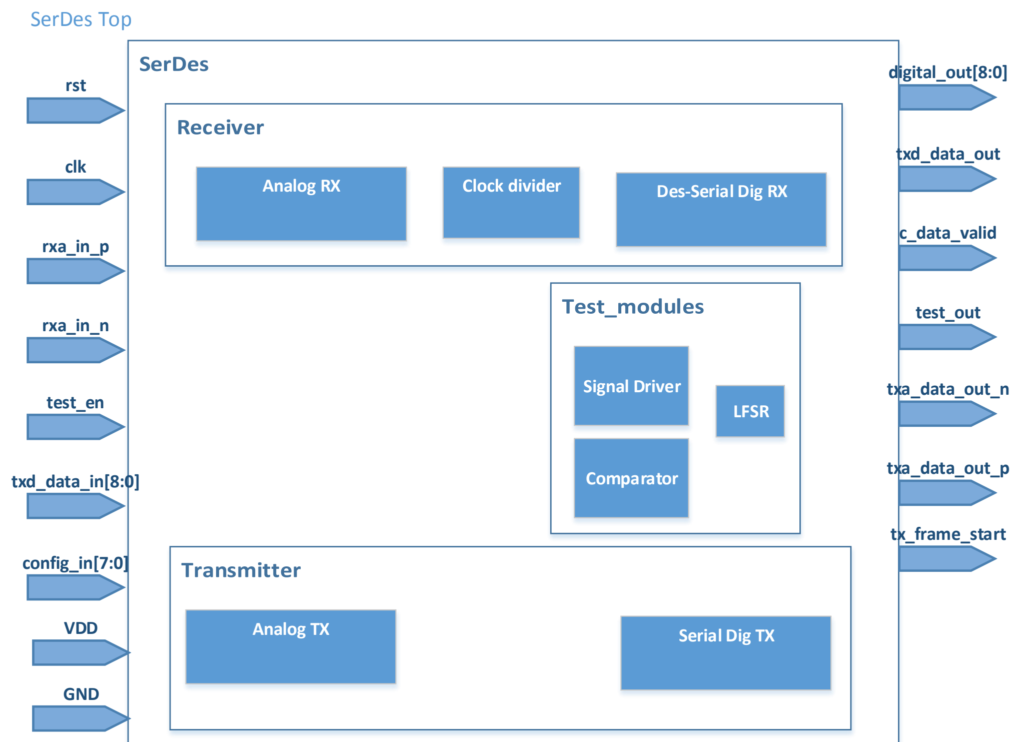

Figure 10 - SerDes Top Level

The following table (Table 3) shows all the external inputs and outputs of the *SerDes* circuit.

***SerDes* top**	**Signal name**	**Description**
Inputs	VDD	Power supply
	GND	Ground
	rst	Global reset active high
	clk	Global clock
	rxa_in_p	Positive input for the analog receiver
	rxa_in_n	Negative input for the analog receiver
	txd_data_in [8:0]	Parallel data in for the digital transmitter
	test_en	Signal that enables the test inputs for different testing modes instead of the analog transmitter configuration pins.
	config_in[7:0]	This is the input configuration for either the test modules or the analog transmitter.
Outputs	digital_out[8:0]	This can be either the error number or the digital receiver output depending on bit [4] of the config_in signal.
	txd_data_out	Digital transmitter output
	tx_frame_start	Sync signal of the digital serializer
	c_data_valid	Data valid of the digital receiver
	test_out	Output than can either be the BIST end or the output of the analog receiver depending on the test mode.
	txa_data_out_n	Positive output of the analog transmitter.
	txa_data_out_p	Negative output of the analog transmitter.

Table 3 - External inputs and outputs of the SerDes

Chapter 2: Test architecture for a *SerDes* system

In this thesis we propose a new architecture for testing the *SerDes* system using three test modules such as a signal driver, a LFSR pseudo random pattern generator and a comparator. The implementation of these testing modules will allow the circuit to have eight test modes and flag any mismatch of the data sent and received between two “comma” symbols. These modules and operating modes will be explained in this chapter.

2.1. Background.

The original plan for this part of the project was to improve the DFT modules proposed in [11]. Such architecture was analyzed and studied in depth and some improvements were identified. For example, it required to have another serializer and encoder for sending encoded serial data through the *LFSR*. This would mean adding more logic for testing the circuit than the one used by the circuit itself. The loss of this capability is something that doesn’t compromise the testing capabilities of the circuit so it was decided to remove this feature entirely.

Also, for driving the data, the previous design proposed the use of a series of multiplexers and decoders to interconnect the signals between the internal modules. This type of design increased the complexity and debug capability of the code. Looking forward to avoid this, we opted for a modular implementation, creating a signal driver module which code is simpler, easier to maintain and less complex.

It was decided that the best approach for the current project was to completely redefine the DFT modules from scratch and just reuse some elements used in [12]. This way, the final circuit will be a more focused, clear and reduced module that fulfills with the test

requirements of the *SerDes* system.

The design proposed in this thesis, is a new implementation of the test modules for a *SerDes* system. It proposes the use three test modules and eight different operating modes for testing to completely fulfill with the test requirements of the circuit. The next two chapters will focus on explaining in detail the new test modules and operating modes for testing that were developed.

2.2. Test modules

There are multiple ways for making sure that a circuit works correctly, for this *SerDes* system it was opted to apply DFT (design for testability) methodologies like BIST (Built in Self-Test) to develop different operating modes for testing. As it was mentioned in the previous section 2.1, in addition to the functional mode, eight operating modes specifically designed for testing were embedded on the chip´s architecture, each of these operating modes is explained in detail in Chapter 2.3.

One example of these operating modes is the ability to "bypass" a certain module of the *SerDes* in order to test the system without it. This is particularly useful in case of possible malfunctioning of individual modules. Another example of the testing capability granted by these modes is being able of selecting between a serial or parallel loopback of the data to connect the Digital Receiver Block and the Digital Transmitter blocks internally across the chip to do a full test of the design. For making this possible, a digital module called *test_modules* was developed. This module englobes all the submodules involved in the testing modes. It is composed with three submodules: a signal driver, a LFSR pseudo random pattern generator and a comparator.

The top level diagram of the *test_modules* is shown on Figure 11. This module is connected to both the transmitter and receiver blocks. It contains three modules: the LFSR, the comparator, and the signal driver. It will use the *signal_driver* module to drive the signals appropriately depending on the test mode selected. The *LFSR* module will be used to generate a pseudo random parallel data sequence to be used in BIST mode. Finally, the comparator will be actively looking for mismatches in most of the operating modes.

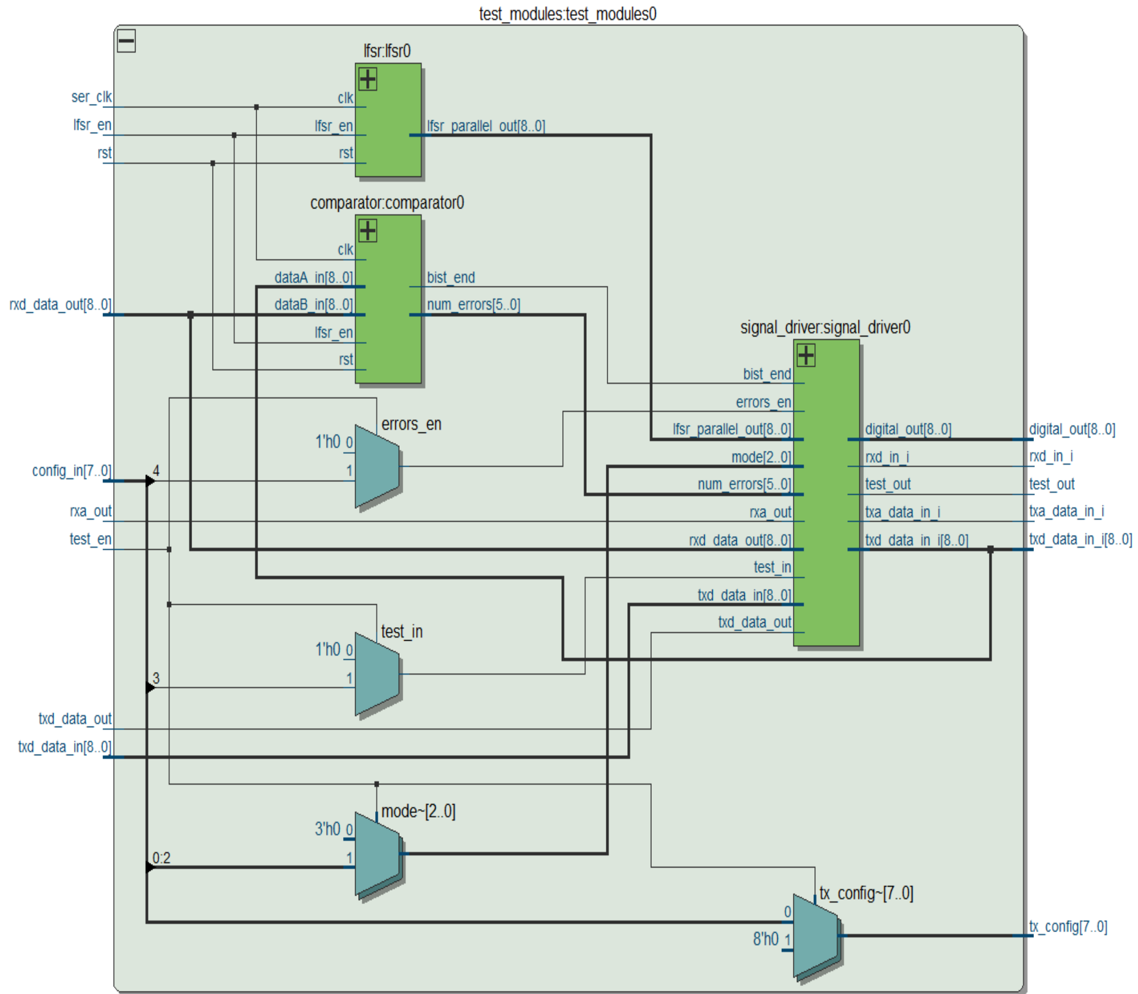

Figure 11 - Test_modules top view

All the sub modules that compose the *test_modules* block will be explained in the coming chapters. The inputs and outputs of this module are also explained in the next table (Table 4).

Test_modules	Signal name	Description
Inputs	ser_clk	A serial clock coming from one of the output clocks of the clock divider.
	lfsr_en	Signal to sync with the transmitter data, tied to the {tx_frame_start} of the transmitter.
	rst	Global reset for the system.
	rxd_data_out[8:0]	Output data of the digital receiver.
	config_in[7:0]	External configuration input that will be used as input of the analog transmitter or test mode selector depending if {test_en} is high.
	rxa_out	Analog receiver output data.
	test_en	Modifies the {config_in [7:0]} signal for test modes entries.
	txd_data_out	Digital transmitter data output.
	txd_data_in [8:0]	Parallel data input for the transmitter.
Outputs	rxd_in_i	Internal wire that connects to the serial input of the digital receiver.
	txd_data_in_i [8:0]	Internal wire that connects to the parallel input of the digital transmitter.
	tx_config[7:0]	Input for the configurations pins of the Analog transmitter.
	digital_out[8:0]	External output that can either reflect the output of the digital receiver or the errors of the system.
	txa_data_in_i	Internal wire for the analog transmitter input.
	test_out	External output that can be either the {bist_en} or the analog receiver output depending on the operation mode.

Table 4 - Inputs and Outputs of the test_modules block

Due to the pin limitation to only 40 external pins we had to add extra logic that allow a multifunction of the input and output pins. It was stablished that the analog transmitter configurations inputs will only be enabled in the functional mode of the *SerDes*, allowing those input pins to be used for the different tests modes available. This is done by setting high or low the test enable signal {test_en} in the circuit. When {test_en} is set low, the *SerDes* input {config_in}, will drive the analog transmitter configurations. When {test_en} is

set high, {config_in} lower bits will instead drive the *test_modules* inputs signals {mode}, {test_in} and {errors_en}. A detailed definition of how setting {test_en} affects the *SerDes* input pins is shown in the following table (Table 5).

test_en	Internal signal	driver
High	mode	config_in[2:0]
	test_in	config_in[3]
	errors_en	config_in[4]
	tx_config[7:0]	set to zero
Low	mode	set to zero
	errors_en	set to zero
	test_in	set to zero
	tx_config[7:0]	config_in[7:0]

Table 5 - test_en modifier

Another signal that allow us to gain some pins is {errors_en}, this signal modify the pins for the digital output {digital_out} between the error counter {num_errors} and the parallel output of the digital receiver {rxd_data_out} (see Table 6).

errors_en	Output signal	driver
High	digital_out[8:6]	set to zero
	digital_out[5:0]	num_errors
Low	digital_out	rxd_data_out

Table 6 - errors_en modifier

2.2.1. The comparator

The comparator module works as a scoreboard that compares two different signals which aren't synchronized, this module will also flag any mismatch error that could exist between the two compared data. For achieving this, a synchronization mechanism was developed that initiates the comparison when a "comma" symbol is detected on the transmitter input data {txd_data_in}. Upon detecting the first "comma" all the next coming data will be continuously stored in registers to be compared against the output data of the digital receiver.

As soon as the receiver has no longer a decoded comma in their input data, this module will start comparing the registered data against the data that is being received. This way only one of the compared data is being stored in registers while the second one is compared on the fly, by doing this we reduce the amount of logic registers used by the comparator.

A six bit counter will be keeping record of the mismatches found between the two data and show the amount of errors in the {num_errors} signal. This will continue to stay true until a second "comma" is sent by the transmitter, ending the comparison.

It's important to state that the data comparison is done only for the first eight bits [8:0] disregarding the ninth bit [9] (bit K of the transmitter) in all cases except when detecting a comma in which all the nine bits are required.

It was decided that all the operating modes would have the comparator enabled when sending data between two commas, comparing the parallel input of the digital serializer and the output of the digital deserializer. This provides the circuit an enhanced diagnostic capability in which this feature is enabled for both offline and online testing.

To reduce the total number of pins, the {digital_out} output signal could be set to either show the number of errors {num_errors} or the parallel output of the digital receiver {rxd_data_out} by having the {errors_en} bit in zero or one.

Figure 12 shows the top module of the comparator. The *comparator* lies inside the *test_modules* block and is connected to the digital serializer and deserializer. Its inputs and outputs are also connected to the signal driver module that will drive the signals appropriately.

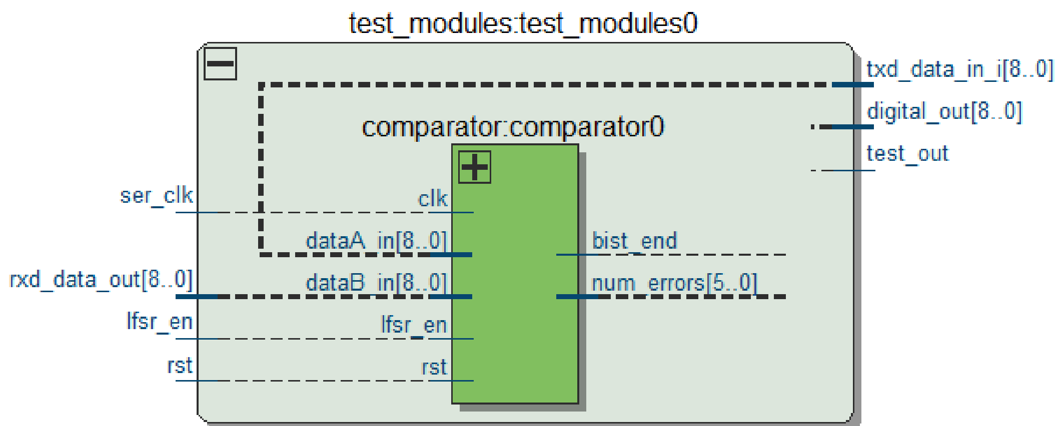

Figure 12 - Comparator top module

The following diagram (Figure 13) shows the schematic of the *comparator* module. It consists of a register array that will capture the transmitted data, synchronize it with the received parallel data and flag any mismatches that may be found.

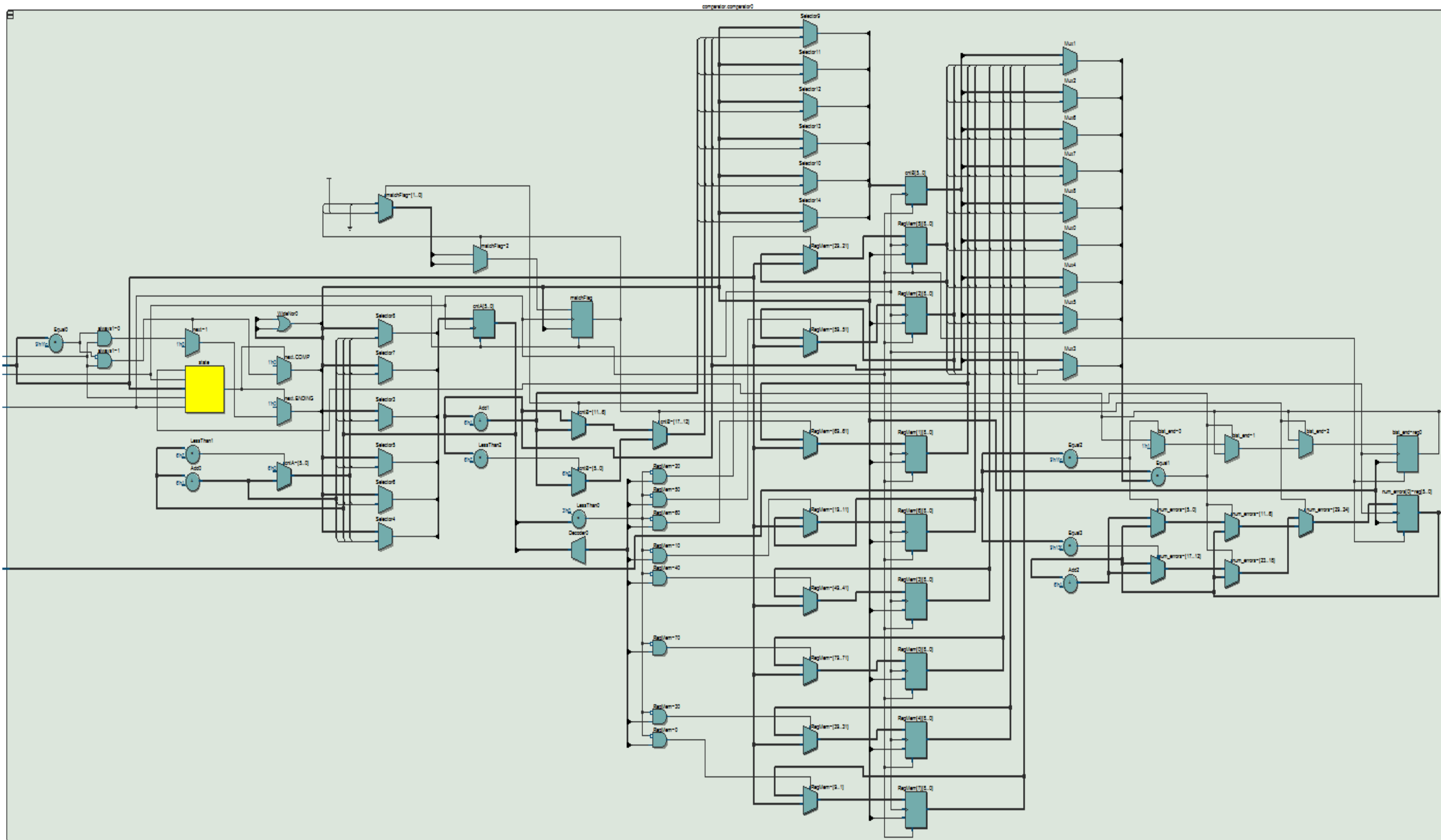

Figure 13 - Comparator schematic

The first data to compare {dataA_in} will always come from the input of the digital transmitter {txd_data_in_i}, however if this data comes from either the pins or the *LFSR* will depend based on the test mode selected. The second data {dataB_in} will remain to be the output of the digital receiver {rxd_data_out} for all the cases. An explanation of the rest of the inputs and outputs of this module can be found on Table 7.

Comparator	**Signal name**	**Description**
Inputs	clk	A serial clock coming from one of the output clocks of the clock divider.
	dataA_in[8:0]	Data A to be compared. Meaning the input data of the digital transmitter. {txd_data_in_i}
	dataB_in[8:0]	Data B to be compared. Meaning the output data of the digital receiver. {rxd_data_out}
	lfsr_en	Signal to sync with the transmitter data, tied to the {tx_frame_start} of the transmitter.
	rst	Global reset for the system.
Outputs	bist_end	Set when the BIST mode has ended its cycle.
	num_errors	Show the counter of errors found during comparison.

Table 7 - Inputs and Outputs of the comparator

To reduce the combinational logic used for the *comparator* a Finite State Machine (FSM) was implemented (see Figure 14).

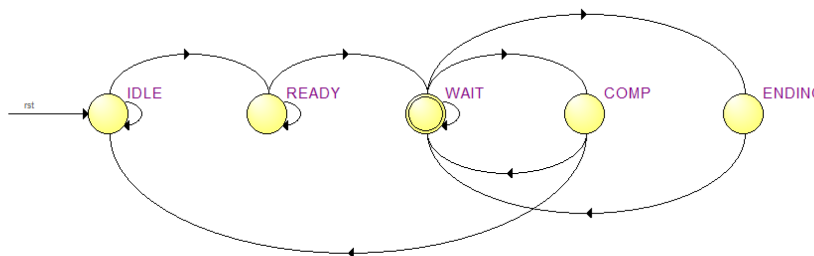

Figure 14 - Comparator FSM diagram

The FSM implemented consist of five states of the *comparator*. These are the following:

- IDLE: The *comparator* will remain in this state as long as {bist_end} signal is set. This is to stop the comparison from restarting when receiving the second "comma"

symbol. Once the {bist_end} signal is asserted, only a reset of the *SerDes* will get the *comparator* out of this state and send it to READY state.

- READY: In this state, the *comparator* is ready to start the comparison. It will monitor the incoming data {data_Ain} and as soon it detects that a "comma" symbol is being transmitted ({lfsr_en} high), it will change to WAIT state.
- WAIT: In this state, on every data transmission (detected by the pulses of {lfsr_en}), the *comparator* will be monitoring the incoming data {data_Ain} and as if it detects that a "comma" symbol is not being transmitted it will move to the comparison state COMP. Then, if a second "comma" symbol is being transmitted it will go the ENDING state, meaning that the comparison is about to end.
- COMP: This state is where the data is being registered and compared. A flag {matchFlag}, determines is a first match has already occurred. This is useful for ending the comparison when detecting a second "comma" symbol on the receiver.
- ENDING: When the *comparator*'s FSM reaches this state, it means that a second "comma" symbol has been transmitted and the comparison will now soon reach to an end. To avoid *comparator* errors on mismatches due to this second "comma" symbol, this state will increase both counters {cntA} and {cntB}, skipping the comparison of the second "comma".

The state transitions can be more easily explained by looking into the transition table below:

	Source State	Destination State	Condition
1	COMP	WAIT	!bist_end
2	COMP	IDLE	bist_end
3	ENDING	WAIT	
4	IDLE	READY	!bist_end
5	IDLE	IDLE	bist_end
6	READY	WAIT	(lfsr_en && dataA_in == 9'h1FC)
7	READY	READY	!(lfsr_en && dataA_in == 9'h1FC)
8	WAIT	WAIT	!lfsr_en
9	WAIT	ENDING	(lfsr_en && dataA_in == 9'h1FC)
10	WAIT	COMP	(lfsr_en && dataA_in != 9'h1FC)

Table 8 - Comparator FSM Transition Table

The following diagram (Figure 15) shows the behavior of the *comparator* module when the functional mode is enabled and the transmitter's output and the transmitter's input are tied in the test bench. If there is no "comma" symbol being transmitted through {dataA_in}, the data will be stored in a register {RegMem} that can contains up to 8 data. On every upcoming data if the data matches the stored data, a counter will either increase on {num_matches} or {num_errors} accordingly.

The module uses two internal flags to determine the status of the comparison: {matchFlag} and {comp_busy}. The first one, {matchFlag}, is used to determine if a match has been detected between {data_Ain} and {dataB_in} on the previous data valid. This is especially helpful the case in which the sequence is restarting and there is expectation that the data won't match on receiving the second "comma" symbol. The second flag, {comp_busy}, will be set high after when a "comma" is sent by the transmitter, indicating that the *comparator* is now busy and actively comparing the data. This signal will go down when detecting the comparison comes to an end.

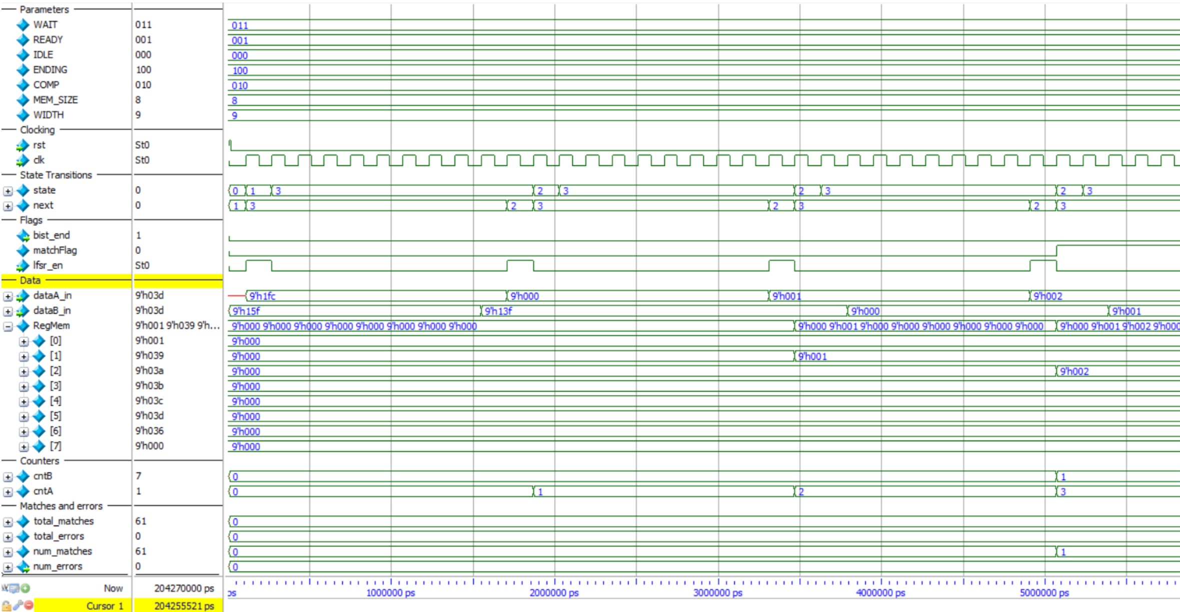

Figure 15 - Comparator Simulation Start

The comparison will be ended when a second "comma" symbol (encoded) and at least one match has been detected. Upon receiving this "comma", the *comparator* may receive wrong data on {dataB_in} as the sequence will be restarting. This module is smart enough to detect this and increment the counters twice to skip this data from the comparison so it won't flag errors in this case. Then after this second "comma" is received, the *comparator* will set

{bist_end}, reset the flags and stop the comparison. In this way, we can count how many mismatches are there between two “comma” symbols. This can be appreciated in Figure 16.

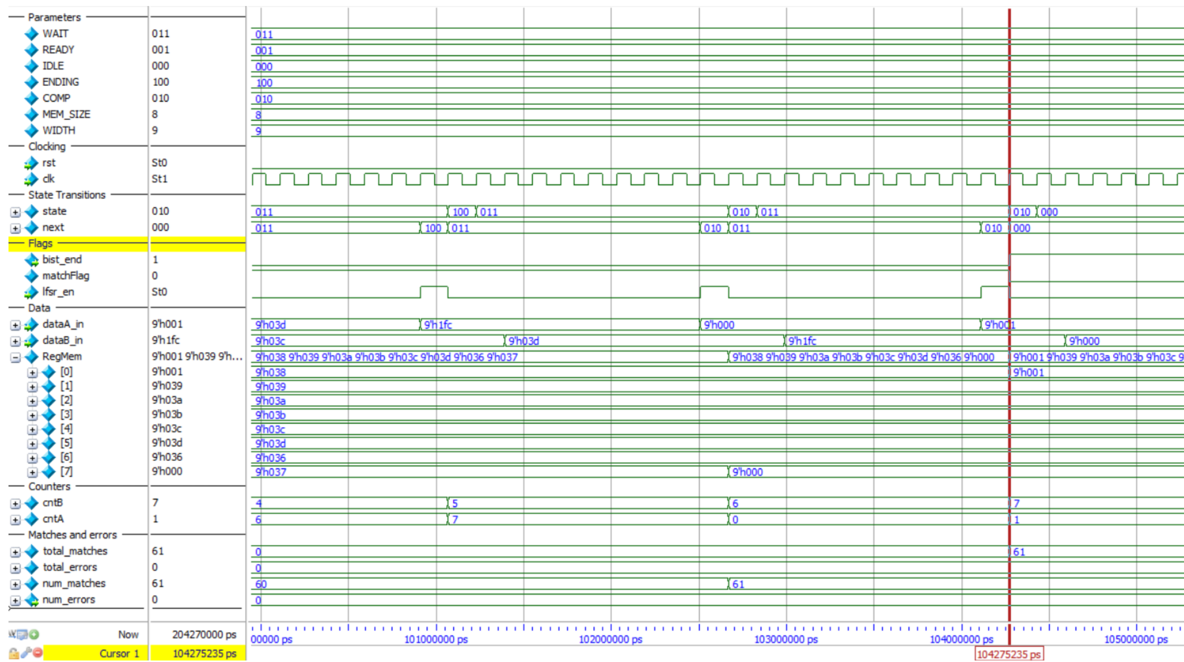

Figure 16 - Comparator Simulation End

2.2.2. The LFSR

The LFSR, is in charge of creating a pseudo random pattern signal that is injected to the digital transmitter parallel input. This will allow the *SerDes* to create parallel data by itself with the purpose of enabling BIST mode. After the first data is created, the *LFSR* will wait for the conversion from parallel to serial to conclude in order to generate the next data. This is achieved by connecting the transmitter {tx_frame_start} signal to the enable signal of the *LFSR* {lfsr_en} this will trigger a data shift on the *LFSR* register and generate the next pseudo random pattern value of the sequence.

The initial value of the output registers {lfsr_parallel_out} is determined by fixed seed. The *LFSR* seed is set to start on reset to the initial value of a “comma” symbol of 10'b1111111100, then on every clock of the enable signal {lfsr_en} it generates the next data. The *LFSR* module can generate a total of 61 random patterns from “comma” to “comma” before restarting the sequence.

The data generated by the *LFSR* will not take into account the ninth bit, (bit K of the transmitter), for the generated data and set {lfsr_parallel_out [9]} to zero for every random

data, except when transmitting the “comma” symbol which makes use of this bit. This is done in order to guarantee that the data will match the same values in the encoder -> decoder and vice versa. As if the bit K is randomized, the output data will match the different special symbols of the 8b/10b encoding table and will not match the sent data after being decoded.

The *LFSR* module will only generate un-encoded data in parallel, so the output of the *LFSR* will only be connected to the digital transmitter input. This is done in this way because implementing a *LFSR* data for the serial input needs the generated data to be encoded and achieving this will make use of a second serializer and 8b/10b encoder. Figure 17 shows the top diagram of the *LFSR* module.

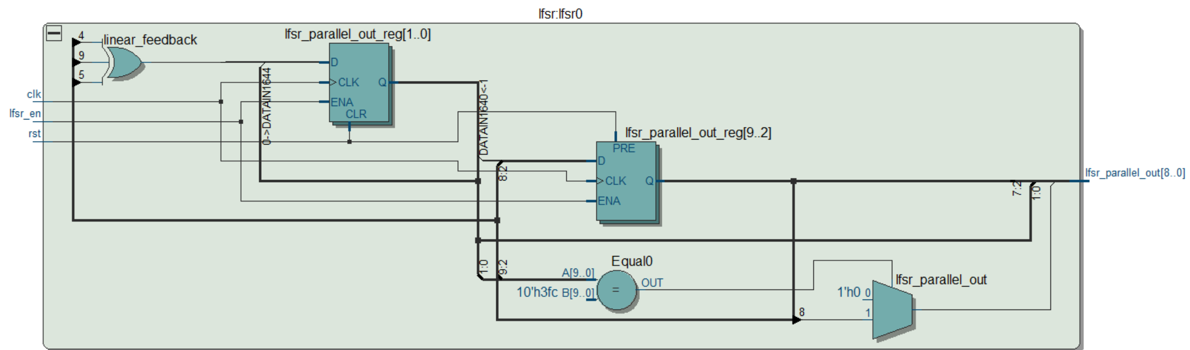

Figure 17 - LFSR top module

This module consists of a series of multiplexers and registers that are connected to a XOR gate to generate the pseudo random pattern. It will change to the next data on every clock if the *LFSR* enable {lfsr_en} is set. The way this works in the *SerDes* system is by setting this signal {lfsr_en} on every {tx_frame_start} of the transmitter in order to synchronize the generated data with the transmitter.

The following table (Table 9) shows the description of the inputs and outputs of the *LFSR* module.

LFSR	**Signal name**	**Description**
Inputs	clk	A serial clock coming from one of the output clocks of the clock divider.
	lfsr_en	Enable bit for the *LFSR* that will shit the data sequence.
	rst	Asynchronous global reset shared among all the modules of the chip.
Outputs	lfsr_parallel_out[8:0]	Parallel output generated by the *LFSR*.

Table 9 - LFSR Inputs and Outputs

The following figure (Figure 18) shows the simulation of the *LFSR* module. In the simulation it is shown that the seed value {seed} is fixed to a 10h'3fc which the value for the comma symbol. On reset {rst}, the *LFSR* will be set to the seed value, so the first output value from the *LFSR* {lfsr_parallel_out} will be the comma upon receiving the enable signal.

On every enable pulse {lfsr_en}, the output value will shift to the next value on the sequence. This will continue to be true for 61 more values until the sequence is restarted and the output of the *LFSR* {lfsr_parallel_out} get again a "comma" value of 9h'1fc.

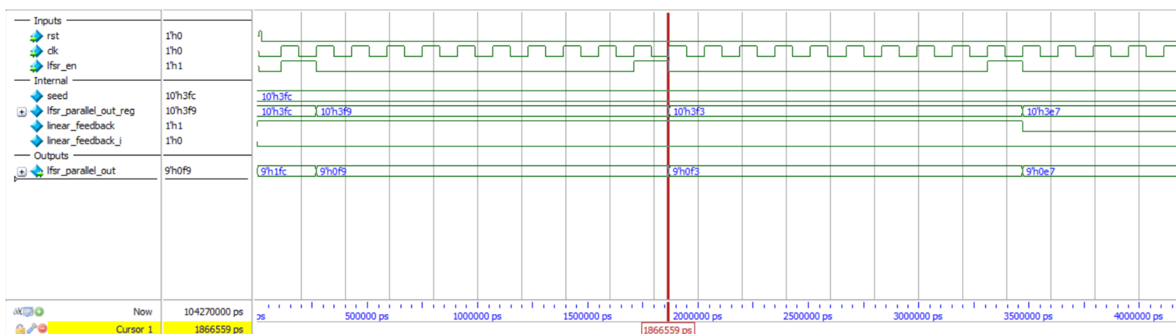

Figure 18 - LFSR Simulation

2.2.3. The signal driver

The signal driver module works as a driver for all the internal signals of the circuit when a different mode of operation is applied on the *SerDes*. This module will basically add a series of multiplexers between the signals of the *SerDes* to connect them according to what operating mode is desired. All the multiplexers of this module can be appreciated in Figure 19.

In this module there are two selectors for the multiplexers, the test mode selector signal {mode [2:0]} and the signal used to see the errors on the output pins {errors_en}. For example if {errors_en} is set, the *signal_driver* will then change the output signal {digital_out} to displays the errors signals: {num_errors[5:0]}, {code_err}, {disp_err} and {dispout} instead of the output of the digital receiver {rxd_data_out }.

The other selector signal {mode [2:0]} will be used to modify the internal signals: {rxd_in_i}, {txd_data_in_i}, {test_out} and {txa_data_in_i} to its appropriate driver for the selected operating mode.

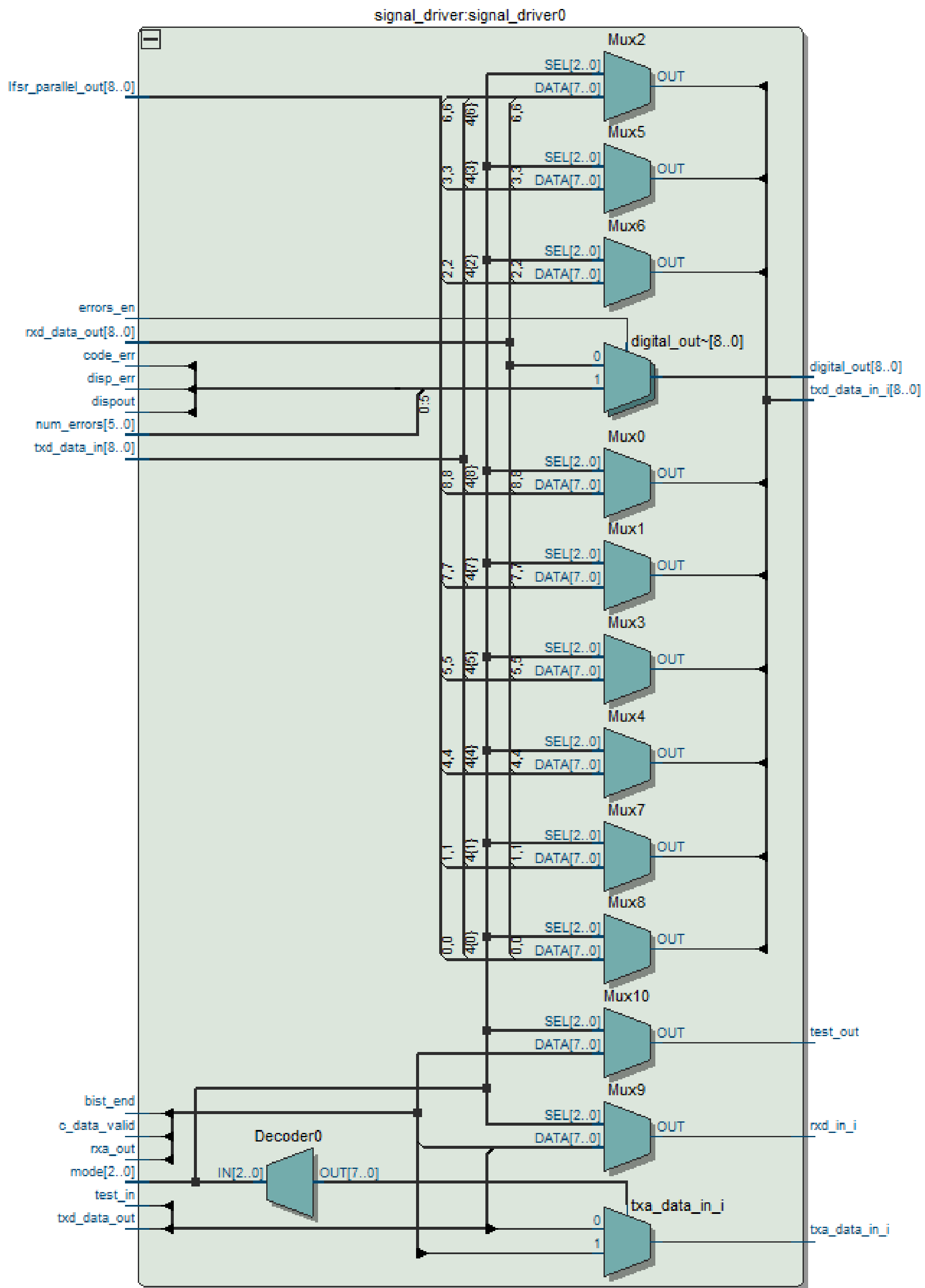

Figure 19 - Signal_driver top module

The following table (Table 10), shows the description of the inputs and outputs of the *signal_driver* module.

Signal driver	Signal name	Description
Inputs	mode[2:0]	Signal that shows the current operation mode of the circuit.
	rxa_out	Analog receiver output data.
	txd_data_in[8:0]	Parallel data input for the transmitter.
	lfsr_parallel_out[8:0]	Internal wire that connects to the parallel output of the digital transmitter.
	rxd_data_out[8:0]	Parallel data output for the transmitter.
	txd_data_out	Digital transmitter data output.
	bist_end	Set when the BIST mode has ended its cycle.
	test_in	Used as an input for the digital receiver {rxd_in} when mode 3 – RXA bypass is selected.
	errors_en	Signal that allows the errors to be shown on {digital_out}.
	num_errors[5:0]	Counter of the mismatches between {txd_data_in} and {rxd_data_out}.
	c_data_valid	Data valid of the digital receiver.
	dispout	Error signal from the decoder.
	code_err	Error signal from the decoder.
	disp_err	Disparity error signal from the decoder.
Outputs	test_out	External output that can be either the {bist_en} or the analog receiver output depending on the operating mode.
	rxd_in_i	Internal wire that connects to the input of the digital receiver.
	txd_data_in_i[8:0]	Internal wire that connects to the input of the digital transmitter.
	txa_data_in_i	Internal wire that connects to the input of the analog transmitter.
	digital_out[8:0]	External output that can either reflect the output of the digital receiver or the errors of the system.

Table 10 - Inputs and Outputs of the signal_driver

2.3. Operating modes

In order to make sure that our *SerDes* circuit works correctly, eight different testing modes were created. These modes cover both bypass and testing modes. The selection is possible due to the *signal_driver* module which by adding a series of multiplexers allows bypass and looping of the modules to perform different operations modes.

To verify the correct behavior of these operating modes, simulation for all of them was performed. For doing this, a test bench file was created for each of them. These testbenches (found in Appendix 7.1) are very similar and have only few variants between each other.

It is not possible to test the correct functionality of these operating modes without all the modules of the *SerDes*, therefore, a top module of the full *SerDes* system described in Chapter 1.10 was used as CUT (Circuit Under Test). For the digital modules; the serializer, deserializer and clock divider modules from previous work [13] were added to the circuit. For the analog modules; dummy modules that behave just as pass-through buffers were created and added to the circuit. The final *SerDes* circuit implementation is shown on Figure 20. This is the architecture of the circuit used for the simulation of all the test Operating modes.

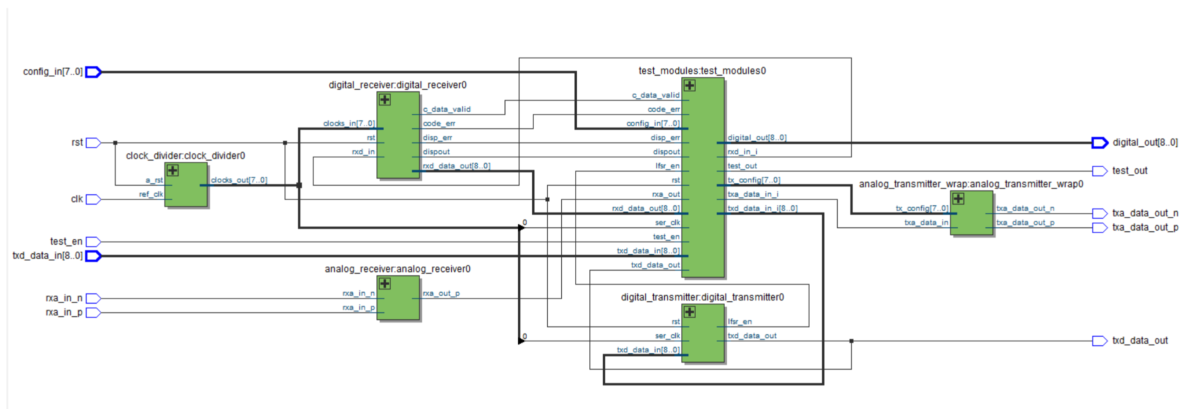

Figure 20 - SerDes top Module

These operating modes are activated by setting the {test_en} signal. Once this signal is set high, the operating modes can be accessed through {config_in [2:0]} otherwise when {test_en} is low, the inputs pins of {config_in [7:0]} will be used to control the configuration setups of the analog transmitter {tx_config [7:0]}. Depending of the operating mode the *signal_driver* will add multiplexers to route the internal signals of the *SerDes* in order to perform the tests mode described in Table 11.

Mode #	Mode[2:0]	test_en	Mode Name	Description
0	000	0	Functional mode	The default operation mode of the *SerDes*. The receiver and transmitter modules work independently. The analog transmitter can receive configuration inputs through {config_in [7:0]}.
1	000	1	Functional mode – Test enabled	By setting the {test_en} signal it enables the testing inputs. When the {test_en} signal is set. The mode can be set through {config_in [2:0]}. {test_in} through {config_in [3]}, {errors_en} through {config_in [4]} and {tx_config} will be tied off to zero.
2	001	1	Parallel loopback	Creates an internal parallel loopback.
3	010	1	Serial loopback	Creates an internal serial loopback
4	011	1	RXA bypass	Bypasses the Analog receiver then observe the signal outside the chip.
5	100	1	BIST with serial loopback	Generates the Data through the LFSR and create an internal serial loopback.
6	101	1	RXA bypass with parallel loopback	Bypass the Analog Receiver and the Analog Transmitter to test the Digital modules. Inject a signal to the Digital Receiver, pass it over to the Digital Transmitter and observe the result.
7	110	1	Open BIST	Bypass the Analog Receiver and the Analog Transmitter to test the Digital modules. Inject a signal to the Digital Transmitter, pass it over to the Digital Receiver and observe the result.
8	111	1	RXA Output with analog loopback	Bypass the Digital Receiver and the Digital Transmitter to test the Analog modules. Inject a signal to the Analog Receiver, pass it over to the Analog Transmitter and observe the result.

Table 11 - Modes description

Detailed descriptions and simulation results of all the operating modes using the complete *SerDes* module are presented in the following chapters.

2.3.1. Operation mode 0 A - Functional mode

This is the typical mode of operation of the *SerDes*. This mode consists of 2 functions, Reception and Transmission. The diagram that show the top level behavior of this mode is shown on Figure 21.

For the Reception function, a differential signal is injected to the Analog RX block {rxa_in_p} and {rxa_in_n}. For simulation purposes, the analog receiver will be seen only as a buffer that will output the positive signal {rxa_in_o} in the output {rxa_out}. In the final circuit, once the *SerDes* system is fully integrated with the analog parts, the Analog Rx block will amplify the signal, eliminates noise, and transforms the differential signal into a single ended signal {rxa_out}. The output of the analog receiver will then be passed over to the input, {rxd_in}, of the digital receiver block. This block, Des-Serial Dig Rx, will de-serialize the upcoming data and decode it from an 8b/10b encoding to a 9 bit parallel output signal {rxd_data_out}.

For the transmission function, a 9 bit parallel signal {txd_data_in} is injected to the digital transmitter block, Serial Digital TX, to be encoded in 8b/10b and serialize it. The output signal {txd_data_out} will be now a serialized encoded signal that will be passed over to the input {txa_data_in} of the analog transmitter, Analog RX, and it will also be driven as an output of the *SerDes* chip. Again for our simulation purposes, the analog block, will be working as a buffer but on the final chip it will allow the signal to be amplified, modulated and equalized depending on the configuration pins {tx_config[7:0]}. Then the transmitter will send out the data as a differential signal outside the chip: {txa_data_out_p} and {txa_data_out_n}.

Mode 0 A – Functional Mode

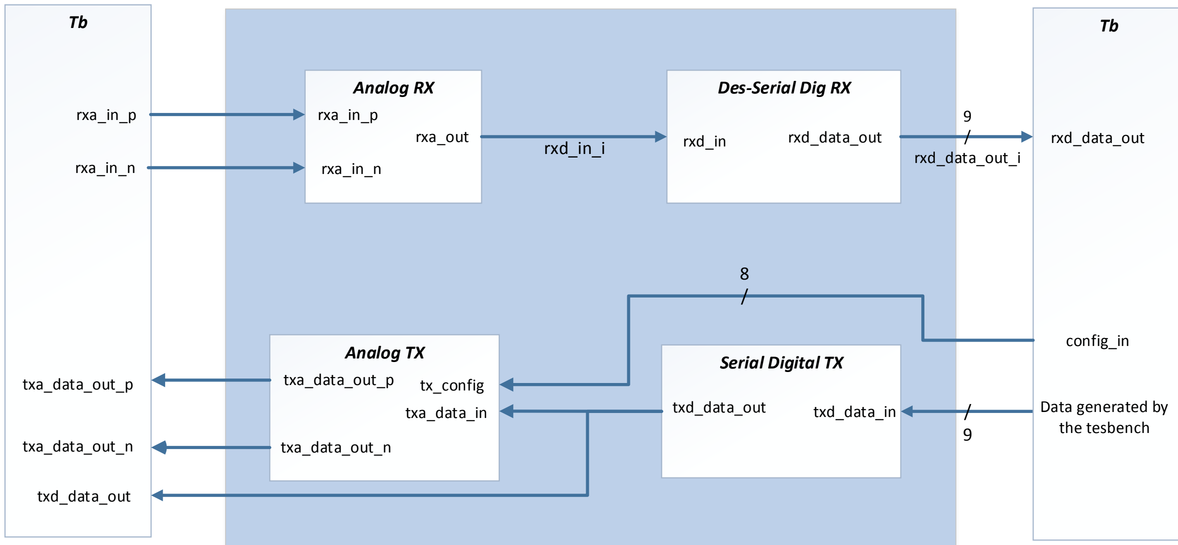

Figure 21 - Mode 0 - Functional mode

The simulation of this operation mode is displayed on Figure 22. The waveform shows how the parallel data {txd_data_in} generated in the testbench is encoded and serialized. Then we can see the differential output of the encoded serialized data in {txa_data_out_p} and {txa_data_out_n}. In this simulation, this output is tied in the testbench and injected back to the analog receiver inputs {rxa_in_p} and {rxa_in_n}. The digital receiver module will now decode and deserializer this signal and send the parallel output to {digital_out}.

The transmitted data have to match with the final received data or the errors counter {num_errors} will increase.

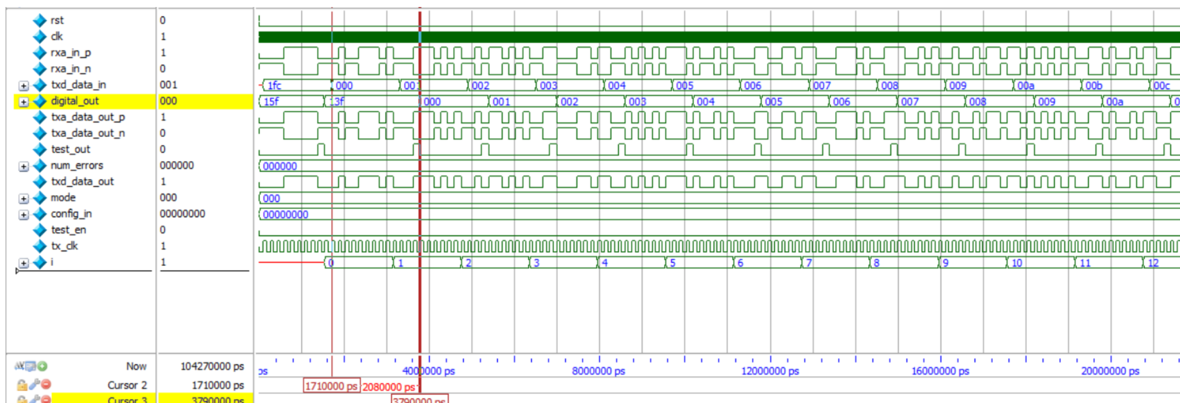

Figure 22 - Mode 0 - Functional mode – Simulation

2.3.2. Operation mode 0 B – Functional mode – Test enabled

This operation mode is the pretty much the same as mode 0 – Functional mode, with the difference that by setting high the test enable signal {test_en}, it will now change the configuration input signal {config_in} to receive the testing inputs. The diagram that show the top level behavior of this mode is shown on Figure 23.

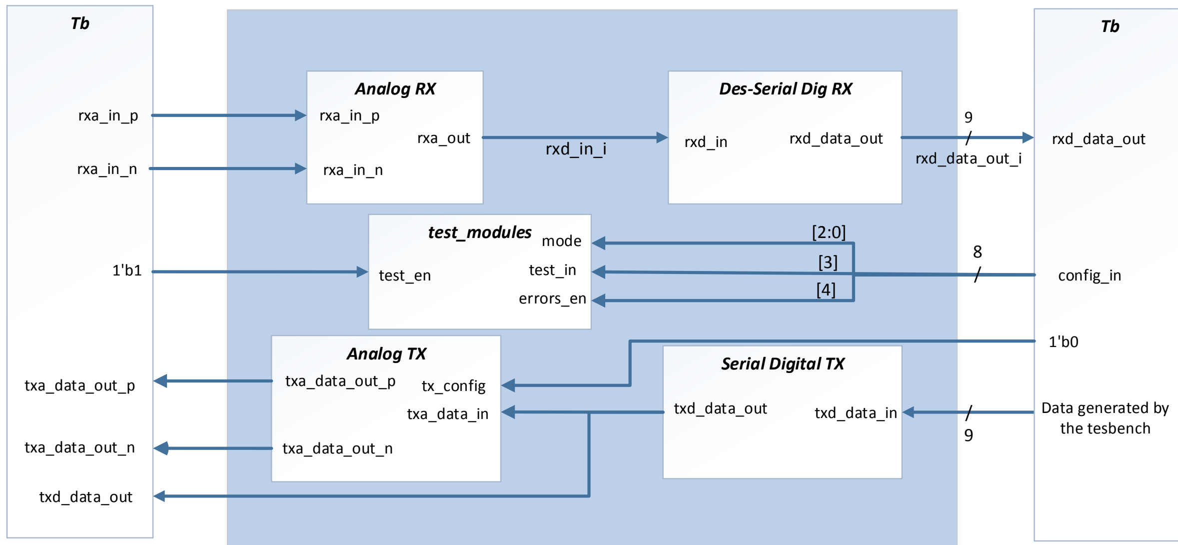

Figure 23 - Mode 0 B – Functional mode – Test enabled

Ideally, this mode wouldn't be necessary and all these inputs would have their own assigned input pin, but due to the limited amount of available pins this action was necessary.

Figure 24 shows the simulation of this mode. The waveforms shows how when {test_en} is set it multiplexes the other signals. {tx_config} is set to zero and {mode}, {test_in} and {errors_en} will now get whatever value is set on {config_in}. Also as {errors_en} is set, we can see that the output {digital_out} will no longer show {rxd_data_out} but rather the errors found in the circuit.

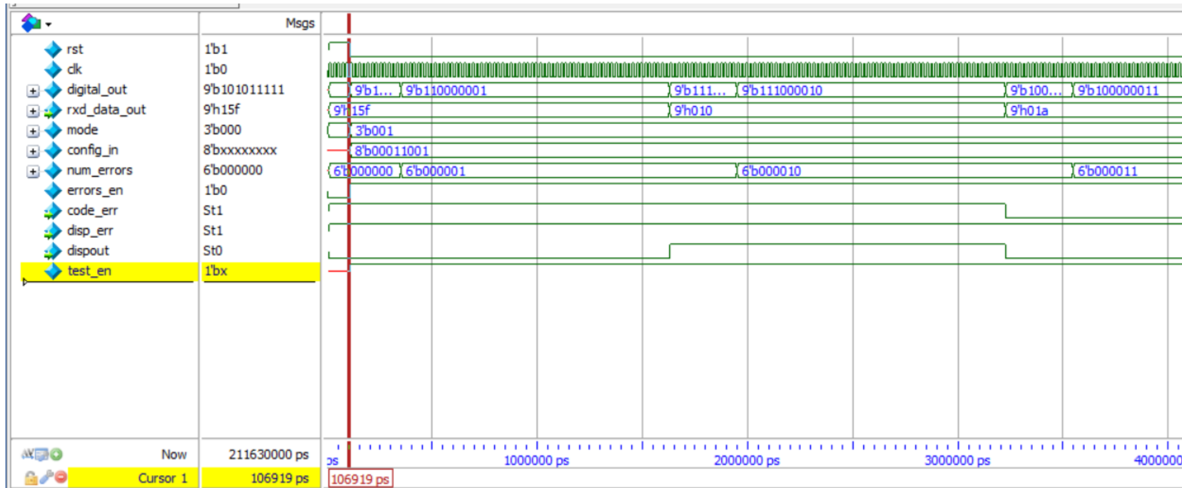

Figure 24 - Mode 0 B - Functional mode – Test enabled - simulation

2.3.3. Operation mode 1 – Parallel loopback

The objective of this operation mode is to create an internal loopback of the parallel data. In this operation mode the data input is done by the differential signals of the analog receiver {rxa_in_p} and {rxa_in_n}. This received serial data needs to be encoded as it is going to pass through the digital decoder and deserializer. The output data, will then be parallel and decoded {rxd_data_out} and will be connected to the input of the digital transmitter {txd_data_in} to create an internal loopback of the data. Finally the transmitter will sent out encoded the differential encoded serial data through {txa_data_out_p} and {txa_data_out_n}. The diagram of this operation mode can be found in Figure 25.

Mode 1 – Parallel Loopback

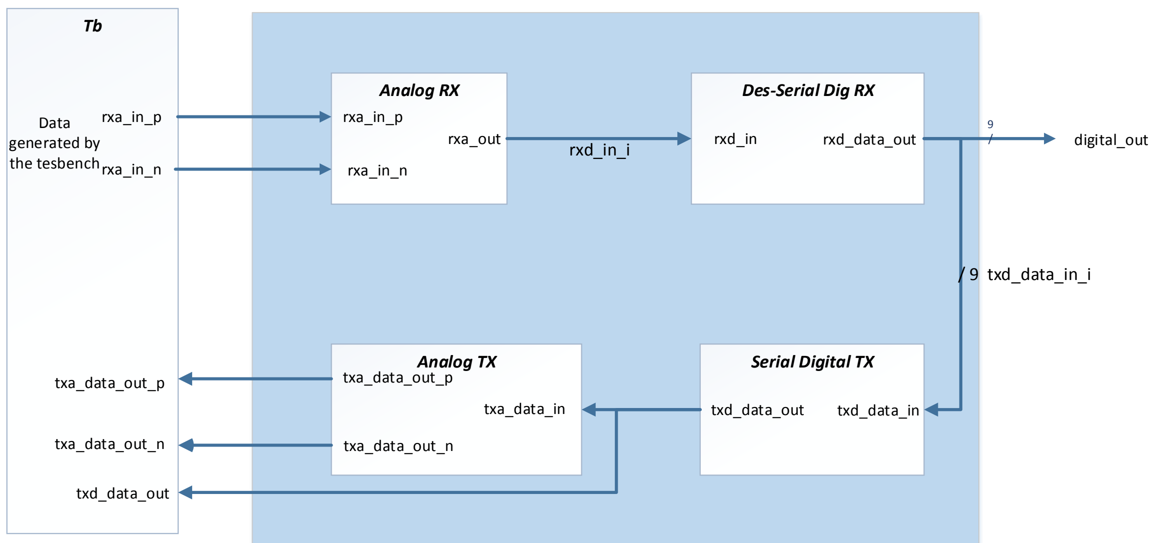

Figure 25 - Mode 1 - Parallel Loopback

The functional simulation of this module is shown on Figure 26. For this simulation, encoded data had to be injected through a serial function. First an encoded “comma” symbol was received (10h’1fc), then the encoded values of 0(10'h0b9), 1 (10'h0ae), and 2 (10'h0ad). In the waveform it is appreciated that {digital_out} is showing the decoded data correctly, and that {txd_data_out} is correctly sending the encoded data serially.
In this mode the error count {num_errors} will not be valid, as only parallel data can be checked this way.

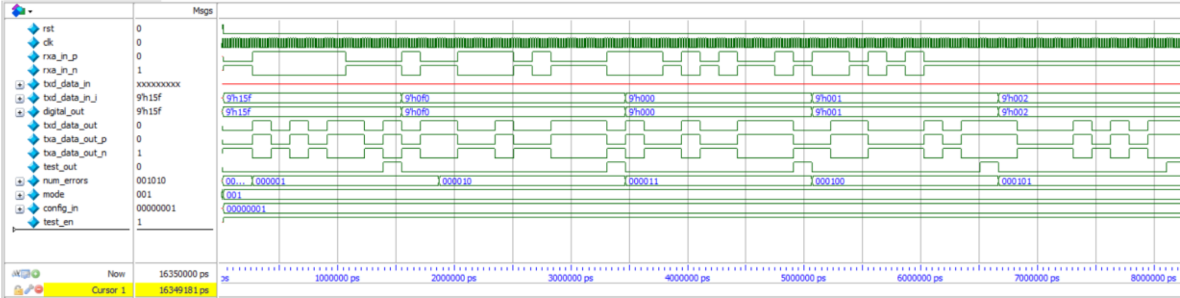

Figure 26 - Mode 1 - Parallel Loopback - Simulation

2.3.4. Operation mode 2 – Serial loopback

The objective of this operation mode is to test the digital modules without intervention of the analog modules. This mode bypasses the analog receiver and creates a loopback of the serial data {txd_data_out} and connects it to the input of the digital receiver {rxd_in} (see Figure 27).

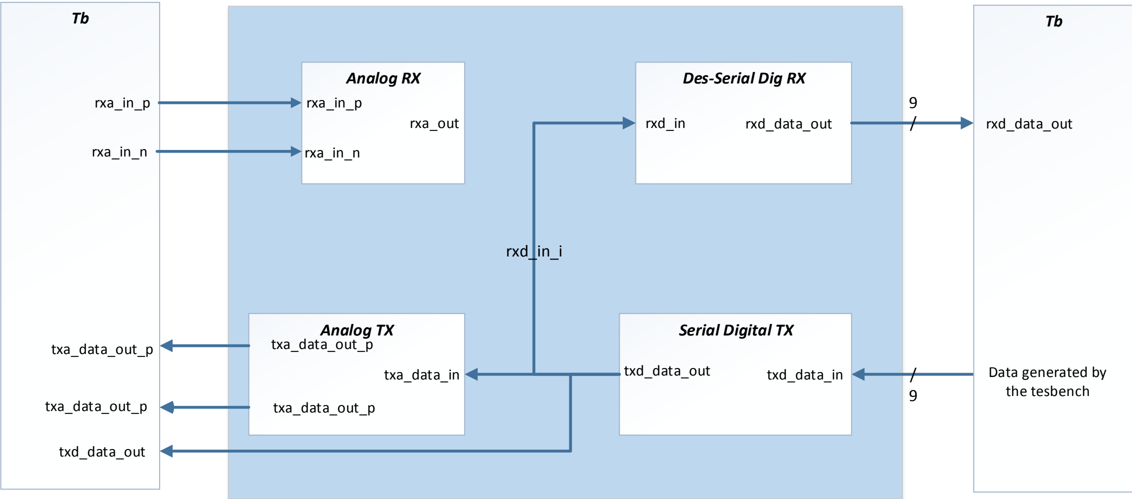

Figure 27 - Mode 2 - Serial Loopback

The simulation for this module is shown on Figure 28. It can be appreciated that even if the analog inputs {rxa_in_p} and {rxa_in_n} have undefined values, the digital receiver input {rxd_in} is still connected to the output of the digital transmitter {txd_data_out} creating the loopback of the serialized and encoded data.

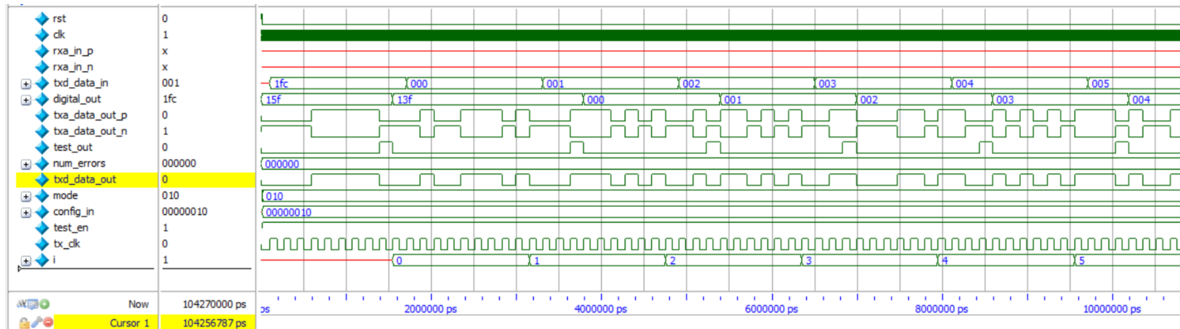

Figure 28 - Mode 2 - Serial Loopback - Simulation

2.3.5. Operation mode 3 – RXA bypass

This mode of operation will make use of the {test_in} signal (enabled through the third bit of the configuration input {config_in[3]} when {test_en} is set) to bypass the analog receiver and connect this signal directly to the digital receiver input signal {rxd_in}.

It was decided to use a different input that the analog ones, to do not mess with the charges of the analog pins. The diagram that describes this operation mode is shown on Figure 29.

Mode 3 – RXA Bypass

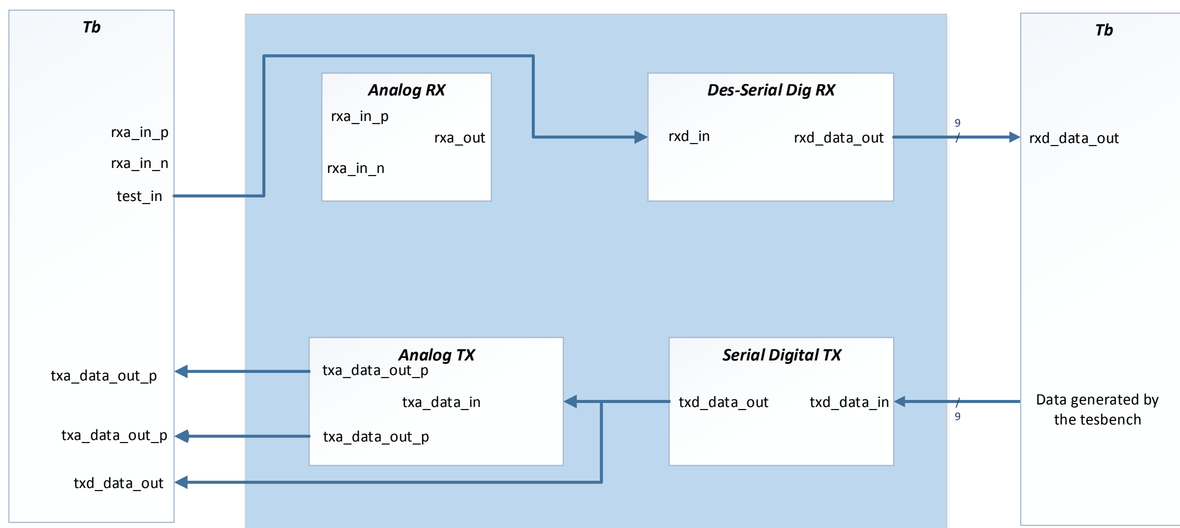

Figure 29 - Mode 3 - RXA Bypass

For the simulation of this test, the {test_in} signal was tied to the positive output of the analog transmitter {txa_data_out_p} in the testbench. Such simulation can be shown on Figure 30. As appreciated in the waveform this operation mode works as expected, as the serial input is coming through {test_in} instead of {rxa_in_p} and {rxa_in_n}.

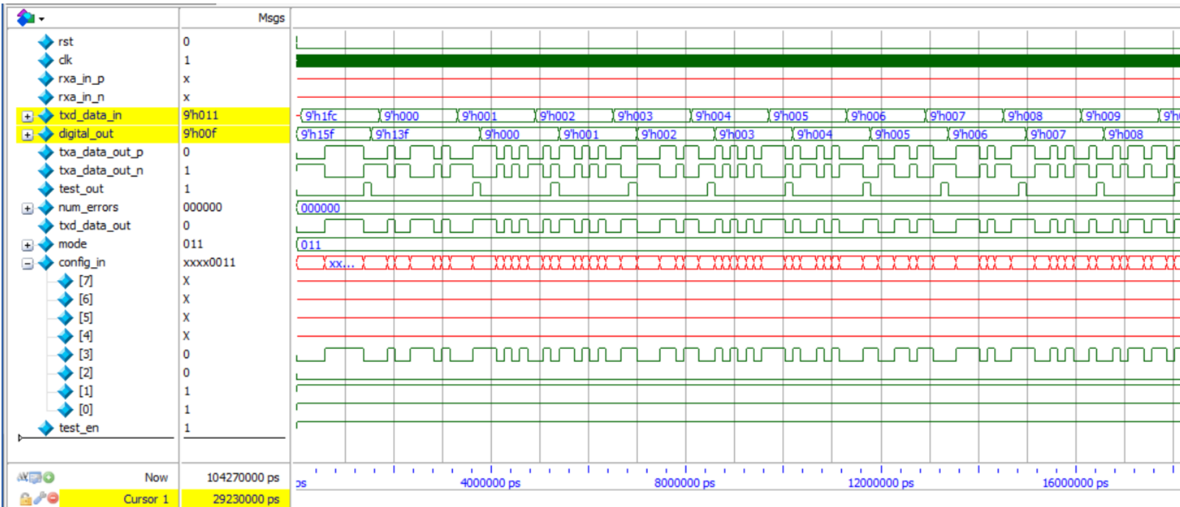

Figure 30 - Mode 3 - RXA Bypass - Simulation

2.3.6. Operation mode 4 – BIST with serial loopback

In this operation mode, the data to be transmitted will be auto generated by the internal module of the *LFSR*. This allows will allow the chip to test itself without any external data generator.

The *LFSR* will generate the parallel data for the digital transmitter input {txd_data_in}, the first data generated after reset will be a “comma” symbol, and then the *LFSR* will start sending a sequence of 61 pseudo random data. The diagram that shows the behavior of this operation mode is shown in Figure 31.

Mode 4 – BIST With Serial Loopback

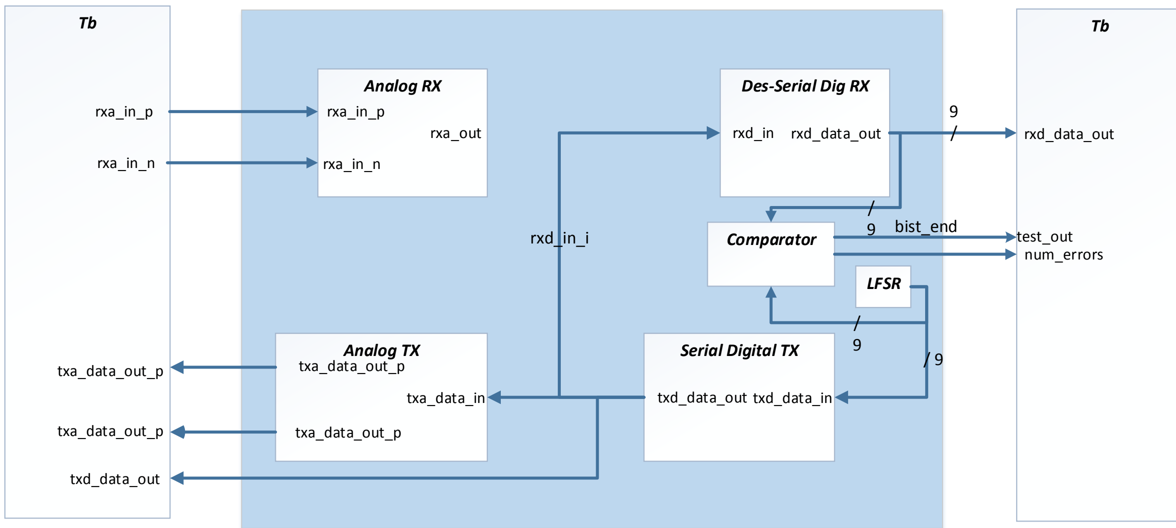

Figure 31 - Mode 4 -BIST with serial loopback

The simulation for this module is shown on Figure 32. It can be seen how the data generated by the LFSR {lfsr_parallel_out} is used as an input to the serial transmitter {txd_data_in_i} instead of the one coming from the pins {txd_data_in} which is X for this mode. We can also notice that the analog receiver inputs {rxa_in_p} and {rxa_in_n} are not driving any value and digital receiver is then fed by {txd_data_out}.

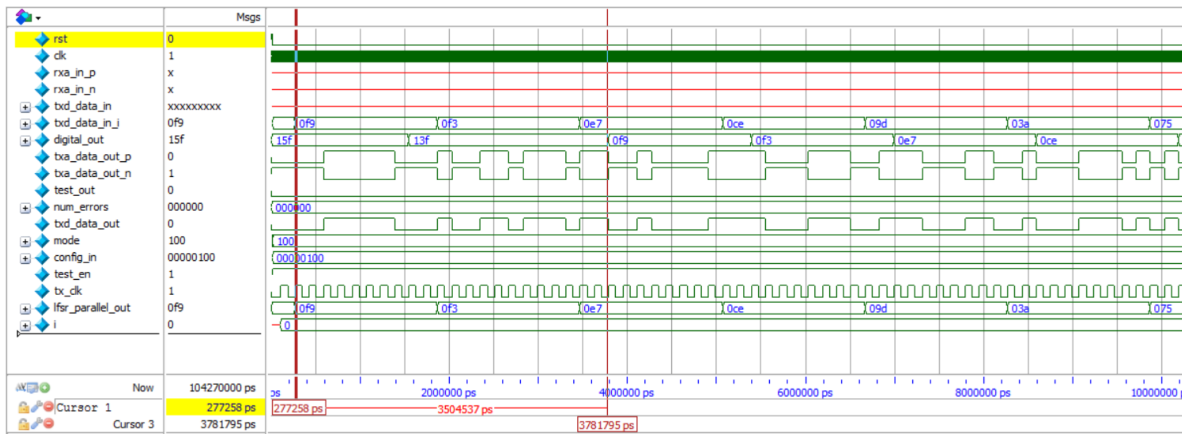

Figure 32 - Mode 4 -BIST with serial loopback - Simulation

2.3.7. Operation mode 5 – RXA bypass with parallel loopback

This mode is a combination of Operation mode 1 – Parallel loopback and Operation mode 3 – RXA bypass. In this operation mode, two things are happening, there is an internal loopback on the parallel data and the analog receiver is being bypassed entirely. Again the

signal used for input will become {test_in} signal that will be connected to the digital receiver input {rxd_in}.

The graphic description of this diagram can be found on Figure 33.

Mode 5 – RXA bypass with parallel loopback

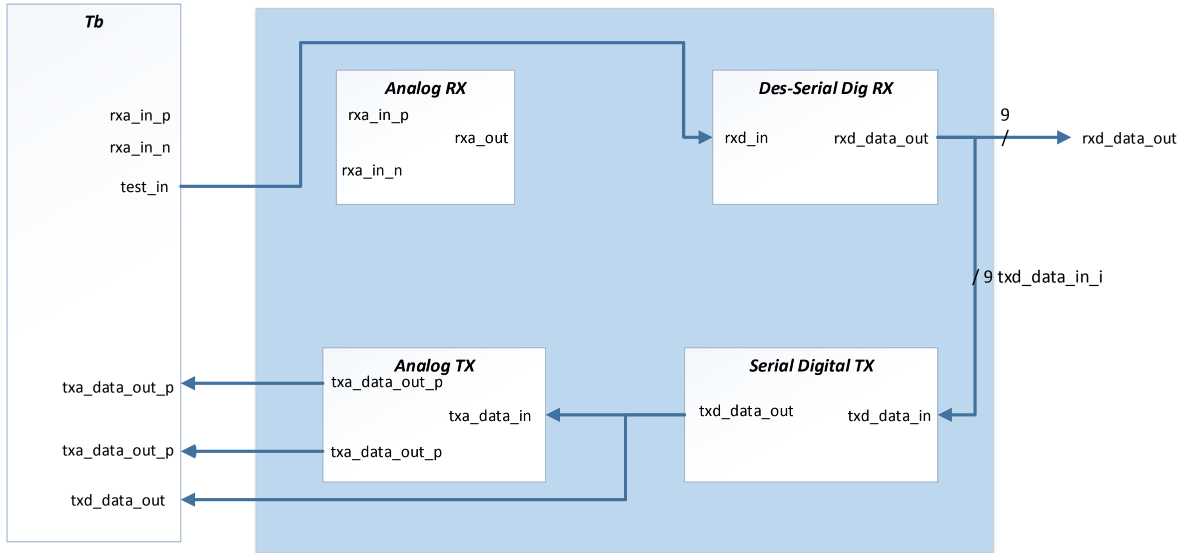

Figure 33 - Mode 5 - RXA bypass with parallel loopback

For the simulation below, Figure 34, as in the Operation mode 1 – Parallel loopback, encoded data had to be injected through a serial function. We're injecting the same values, first an encoded "comma" symbol (10h'1fc), followed by the encoded values of 0 (10'h0b9), 1 (10'h0ae), and 2 (10'h0ad). In the waveform it is appreciated that the upcoming data on digital receiver is coming directly from {test_in} which maps to the *test_modules* internal signal {test_in}. The parallel output {digital_out} is showing the decoded data (0, 1 and 2) correctly. It can also be seen that the output of the transmitter {txd_data_out} is working as expecting, sending the encoded data serially.

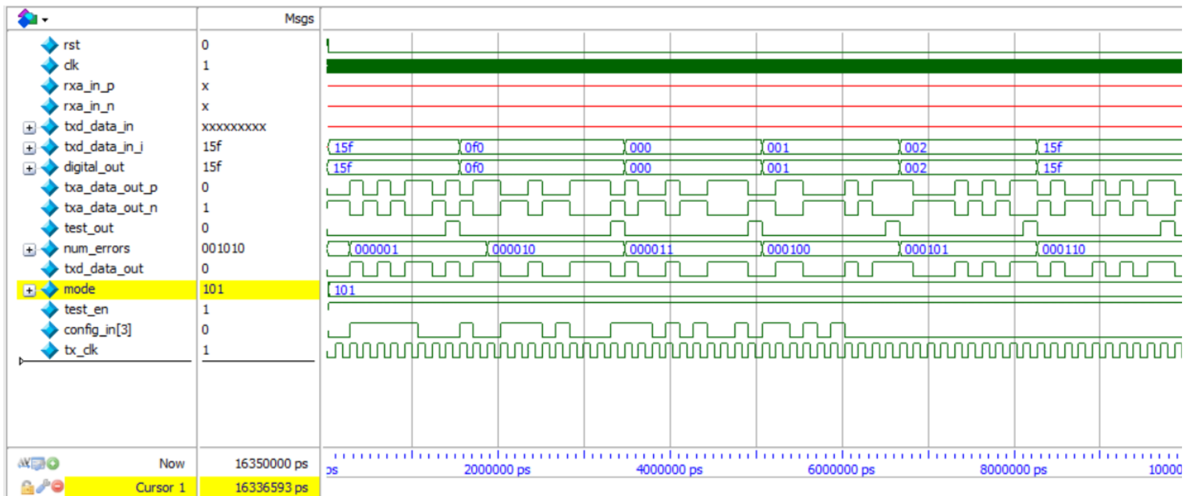

Figure 34 - Mode 5 - RXA bypass with parallel loopback - Simulation

2.3.8. Operation mode 6 – Open BIST

This mode of operation will perform the BIST mode with the difference that there isn't any internal loopback. The *LFSR* will still generate the parallel data to the digital transmitter. The diagram that shows the behavior of this circuit can be seen in Figure 35.

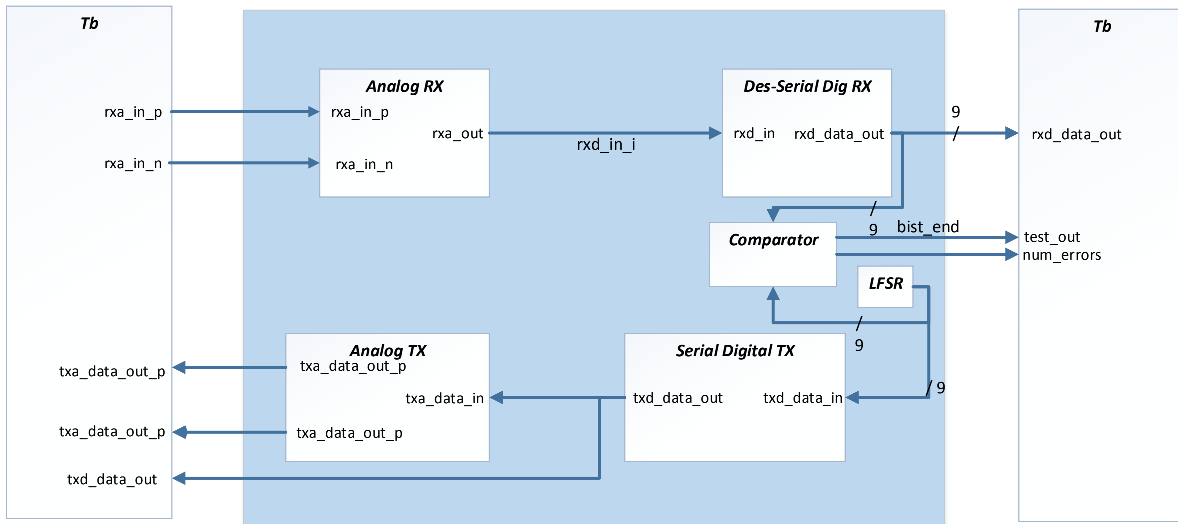

Figure 35 - Mode 6 - Open BIST

It is important to note that the *comparator* will use 4-state (0, 1, x, z) logical equality operators such as "===" and "!==". This way the comparison of data with x's bits will also be flagged as errors.

This kind of operations will be used only in the simulation as == can be synthesized into a

hardware (x-nor gate), but === can't be synthesized as x is not a valid logic level in digital, it is in fact having voltages in between 0 and 1. And z is not itself any logic, it shows disconnection of the circuit.

The simulation for this operating mode is shown on Figure 36. In this simulation the differential input of the receiver {rxa_in_p} and {rxa_in_n} is not connected to show that *LFSR* is generating data and on every clock that the input has x's and the errors counter {num_errors} is increasing every time in which the receiver is not getting any data.

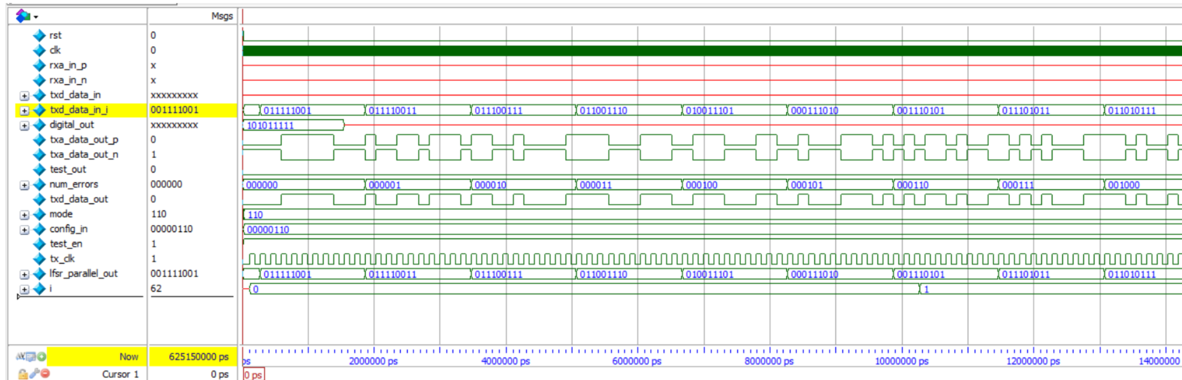

Figure 36 - Mode 6 - Open BIST - Simulation

2.3.9. Operation mode 7 – RXA output with analog loopback

This operating mode consist of way to see the output signal of the analog receiver {rxa_out} on an output pin. This signal is an internal signal that is visible only during this mode of operation.

A second function have been added to this mode, which is the analog loopback created by connecting the receiver output {rxa_out} also to the input of the analog transmitter {txa_data_in}. This operating mode is very useful to test the functionality of the analog blocks.

The diagram that describes this mode is shown on Figure 37.

Mode 7 – Analog Receiver Output and Analog Loopback

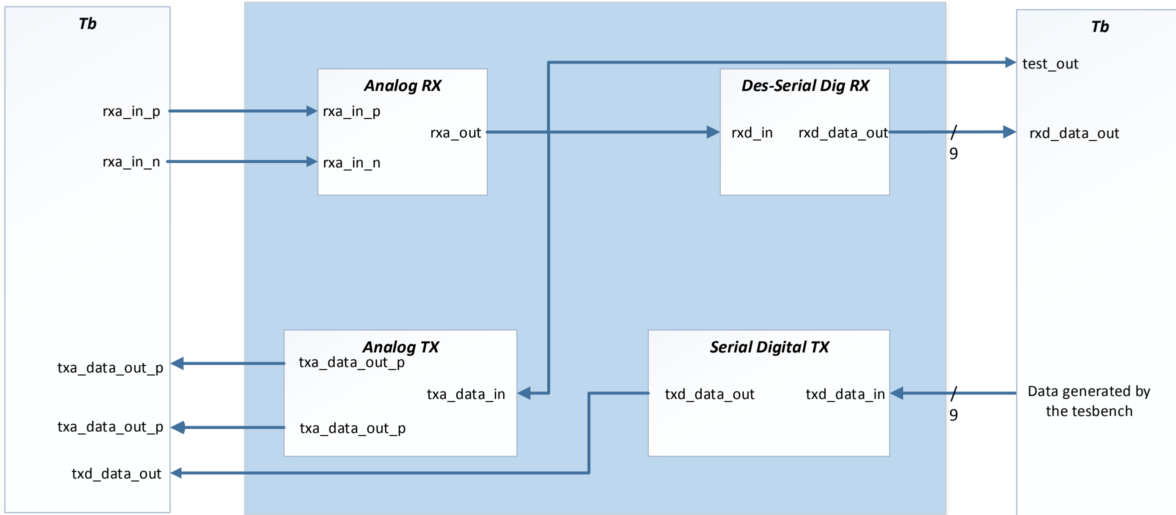

Figure 37 - Mode 7 - RXA Output with analog loopback

The following simulation (Figure 38) shows the behavior of this operation mode. In the testbench we are generating parallel data of a "comma" followed by a numeric sequence starting in zero and sending it through {txd_data_in}. The output of the digital transmitter {txd_data_out} is connected in the testbench to the input of the analog receiver {rxa_in_p}. We can now see that the analog transmitter input {txa_data_in} is getting this signal and outputting the same in {txa_data_out_p} and {txa_data_out_n}. The deserializer is also decoding this data and sending it on the output pin {digital_out}.

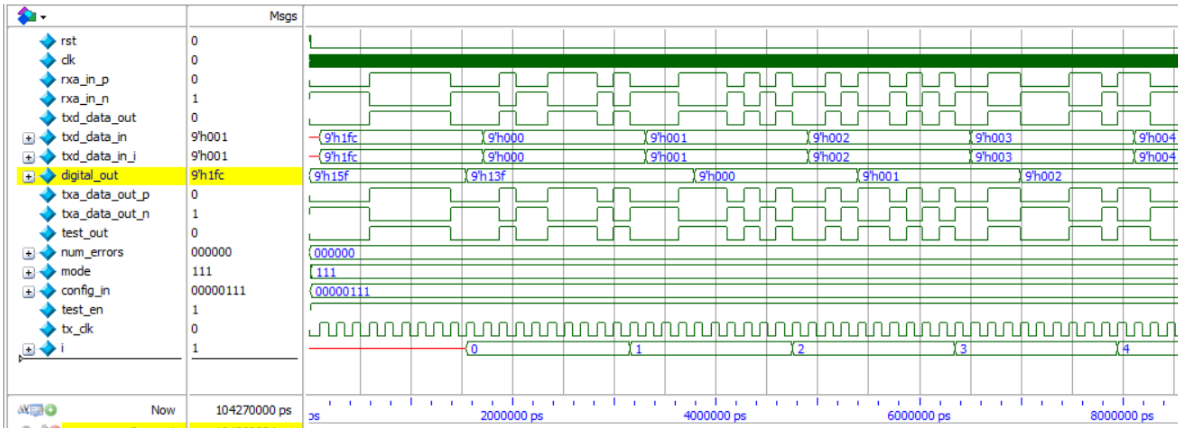

Figure 38 - Mode 7 - RXA Output with analog loopback - Simulation

Chapter 3: Logic synthesis and Gate Level Simulation

The current chapter presents the steps that were followed to perform the logic synthesis for the test modules of a SerDes system. It includes the results obtained with this synthesis and a GLS simulation of each operating modes implemented in the design.

3.1. Logic Synthesis

The logic synthesis is the translation of the RTL's behavioral model into logic gates. Then these gates will be mapped to the 130nm **IBM**´s cmrf8sf Design Kit standard cells library.

The **RTL Compiler** (RC) tool was used to obtain the logic synthesis. This tool is controlled by TCL commands. TCL is common scripting language used to interact with Electronic Design Automation tools such as ***RTL Compiler*** (RC) or ***Encounter Digital Implementation System*** (EDI). Each command performs a step of digital synthesis to elaborate using configuration files such as HDL, constraint, etc. The full set of commands to perform logic synthesis with timing constraints were added to the script *logic_synthesis.tcl* found on Appendix 7.2.

The logic synthesis also requires certain files or ingredients. The synthesis flow with all these files is shown on Figure 39.

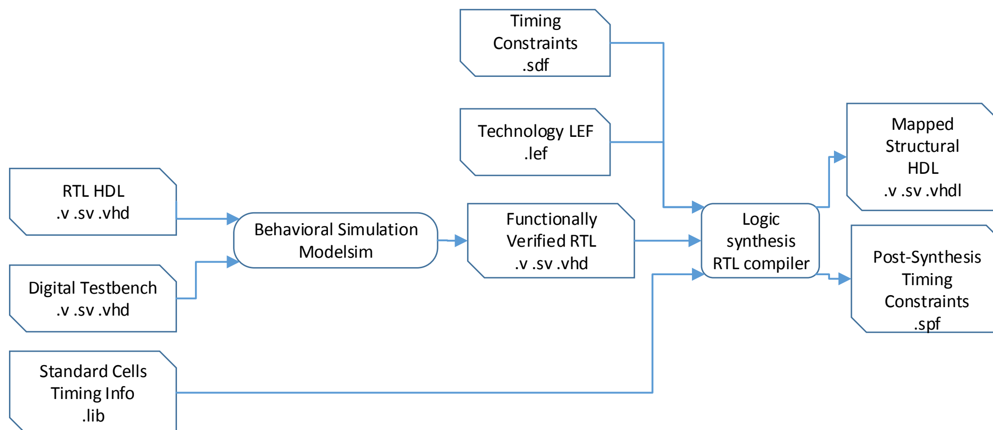

Figure 39 - Logic Synthesis flow

Some examples of the files or ingredients that were added to the script are:

- The path of a constraints file (*test_modules_m.sdc*). This file contains timing constraints in order to define the clocks of the system, the frequency of operation of the chip, hold and setup timing constraints, among others.
- The path of the all the HDL modules (*test_modules.sv*, *lfsr.sv*, *comparator.sv* and *signal_driver.sv*) files containing the behavioral *RTL*´s description of the chip.
- The path of the .lib files of the 130nm **IBM**´s cmrf8sf Design Kit.

Running **RTL Compiler** parsing this script will generate the logic synthesis of the circuit. The command used is the following:

```
rc -f logic_synthesis.tcl
```

The circuit generated by **RTL Compiler** can be appreciated when opening the GUI, using the option "`-gui`". Figure 40, Figure 41, Figure 42 and Figure 43 show the *test_modules*, *comparator*, *LFSR* and *signal_driver* diagram respectively.

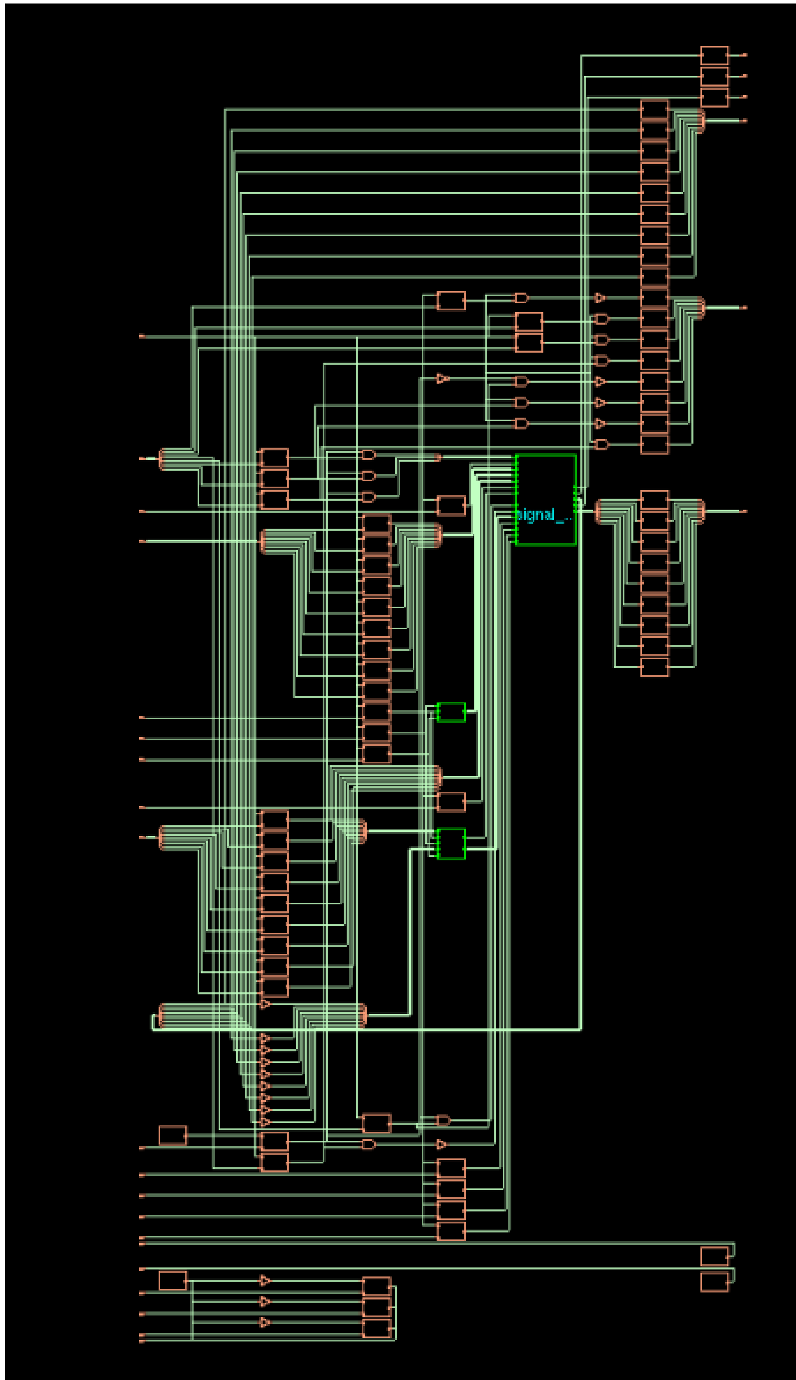

Figure 40 - Test_modules RTL Compiler schematic

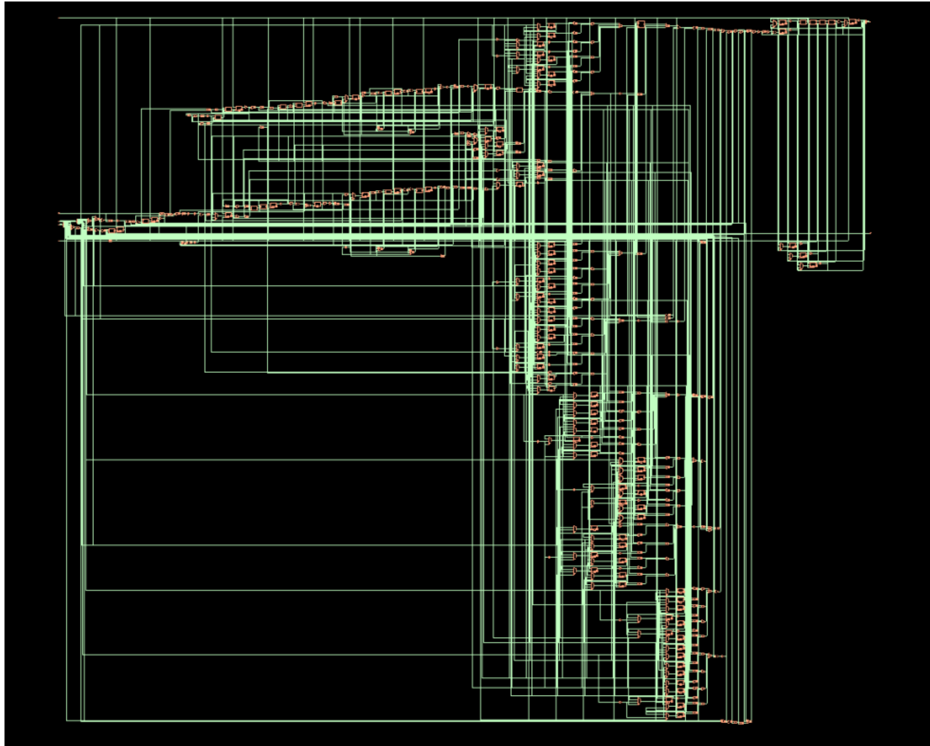

Figure 41 - Comparator RTL Compiler Schematic

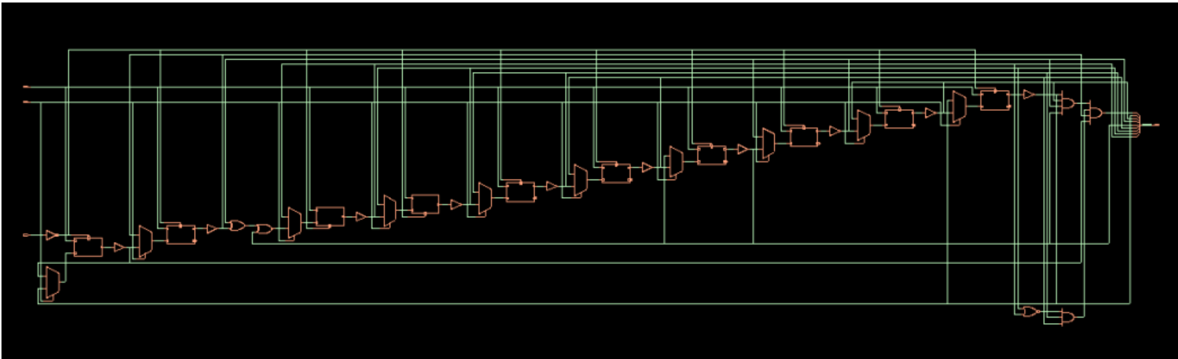

Figure 42 - LFSR RTL Compiler Schematic

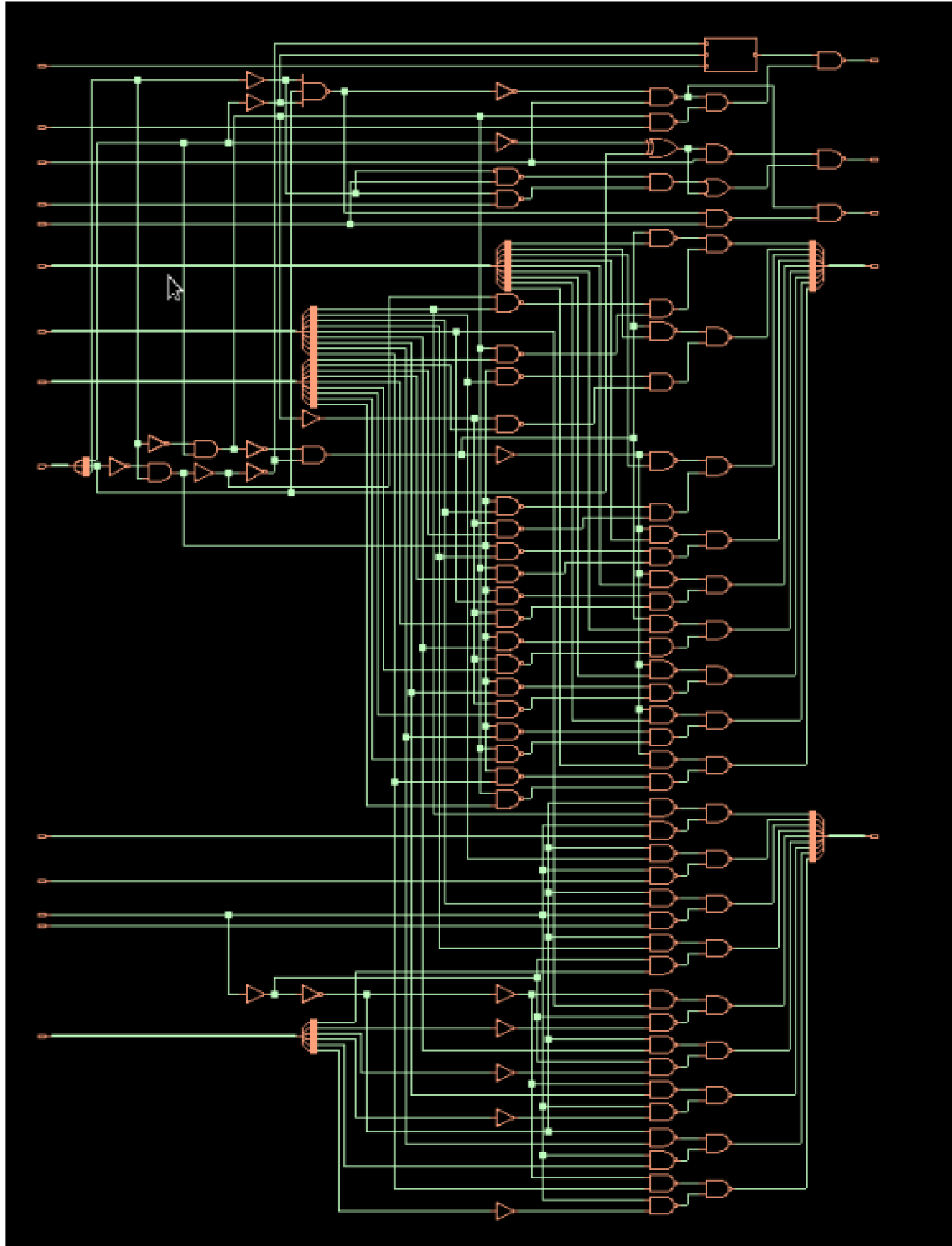

Figure 43 - Signal Driver RTL Compiler Schematic

After performing the logic synthesis, the tool generates output files that are needed for the physical synthesis, which is the next step of the VLSI design flow. There are two main output files generated by **RTL Compiler**. An HDL file containing the design mapped to the standard cells contained in the **IBM**´s cmrf8sf Design Kit. This HDL (*test_modules*_m.v) file will be used to perform a second round of simulations on the *SerDes* to make sure that the circuit behaves the same way before and after the logic synthesis.

There is important information that can be obtained from the output files of **RTL Compiler**.

The gates and power summary reports is shown on Table 12 and Table 13 respectively.

3.1.1. Gates summary report

Instance	Cells	Cell Area	Net Area	Total Area
test_modules	689	9798	9530	19328
--*comparator*0	519	7584	6423	14008
--*signal_driver*0	102	1220	964	2184
--*lfsr*0	38	654	280	934

Table 12 - Gates Summary

3.1.2. Power report

Instance	Cells	Leakage	Dynamic	Total
		Power (uW)	Power (uW)	Power (uW)
test_modules	725	0.012	1348.857	1348.870
--*comparator*0	523	0.009	899.645	899.655
--*signal_driver*0	121	0.002	190.680	190.682
--*lfsr*0	51	0.001	115.748	115.749

Table 13 - Power Report

3.1.3. Gate Level Simulation

Gate-level simulation (GLS) is used to boost the confidence regarding the implementation of a design and can help to verify the dynamic circuit behavior, which cannot be verified accurately by static methods. It is a significant step in the verification process.

GLS may take up to one-third of the simulation time and could potentially take most of the debugging time. It is run after RTL code is simulated and synthesized into a gate-level netlist.

Figure 44 illustrates the basic GLS flow. Performing gate-level simulation gives us the opportunity to check that our circuit still works properly after being synthesized and Placed and Routed. Additionally, we use the gate-level simulations to estimate energy consumption considering the switching activity for each gate in the design.

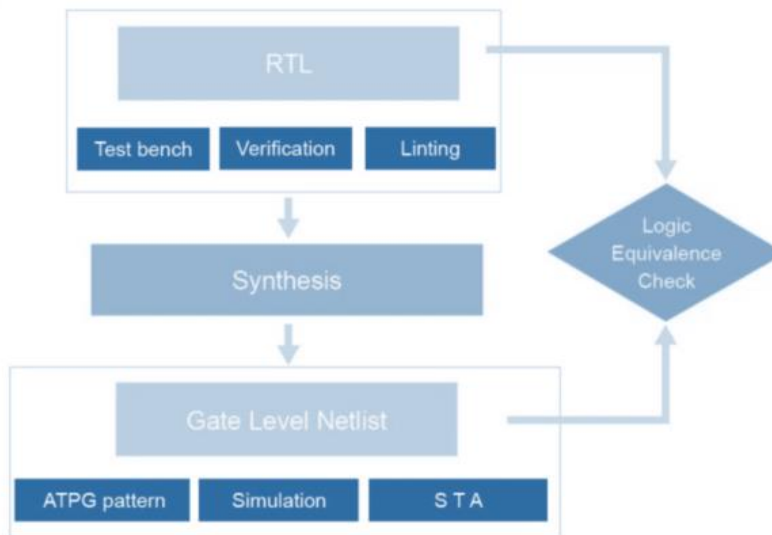

Figure 44 - Gate-level simulation flow

Using **RTL Compiler**, we obtained an HDL file named *test_modules_m*.v. This file contains a gate level model of the test modules that is synthetized and mapped to the gates of **IBM**´s cmrf8sf Design Kit standard cells library.

To verify that the logic synthesis of the test modules is still having the same functionality as the original RTL model, a second round of simulations was performed but now replacing the *test_modules* with the GLS model.

For these simulations, **Quartus prime** kept on crashing so **Modelsim** was used instead for

both compile and simulation. Also, the same testbenches used for the verifying the operating modes were reused.

As it is noted on the remainder of this chapter, the functionality of the *SerDes* system remained the same after replacing the *test_modules* block with its gate level counterpart. However, it is noticeable on Figure 45 that it takes to the circuit 17000 ps to stabilize after reset is asserted. This is expected on a gate level model due to the delays associated to the standard cells and doesn't affect the final functionality of the circuit.

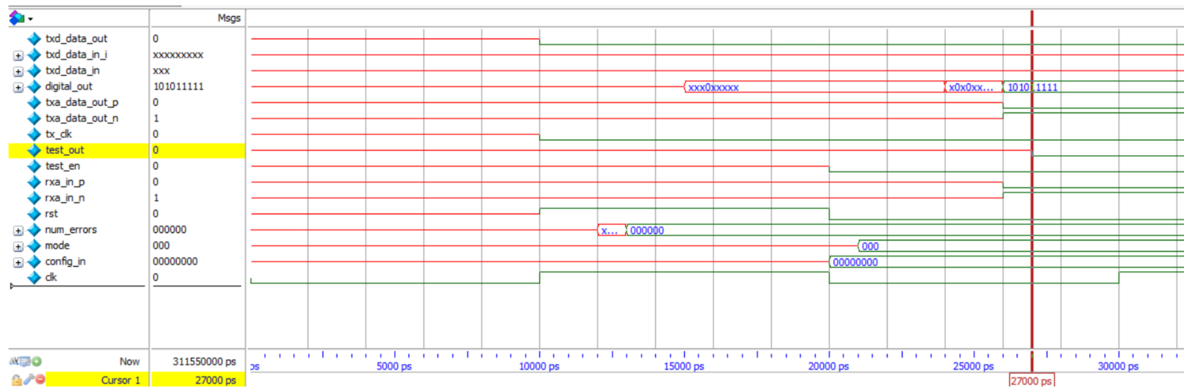

Figure 45 - GLS resetting sequence

All the operating modes were simulated using GLS. The objective of GLS is to verify that the behavior remains the same between the two models. The results from those simulations are presented in the coming sections.

3.1.3.1. GLS - Operation Mode 0 A - Functional mode

The following Figure 46 shows that the *SerDes* system functional mode is still working as described in Chapter 2.3.1. The most important to look at is that {txd_data_in} input to the system matches the output {digital_out} after the next positive edge flank from {test_out}. It is also important to verify that there aren't any errors flagged by the errors counter {num_errors}.

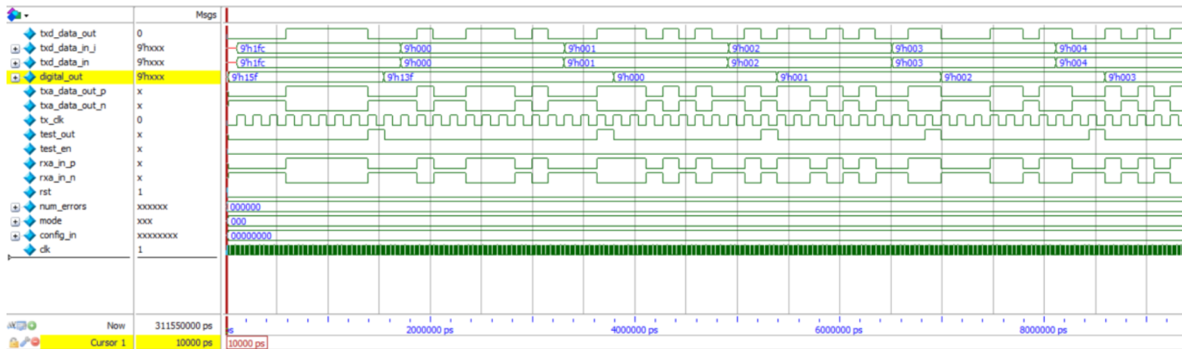

Figure 46 - GLS - Mode 0 A - Functional mode

3.1.3.2. GLS - Operation Mode 0 B – Functional mode – Test enabled

Figure 47 below, shows that the *SerDes* system test features are still being enabled when setting {test_en}. This is further explain in Chapter 2.3.2. Basically, by setting {test_en}, the input of the *SerDes* {config_in} will now be the test modules inputs instead of the analog transmitter's ones.

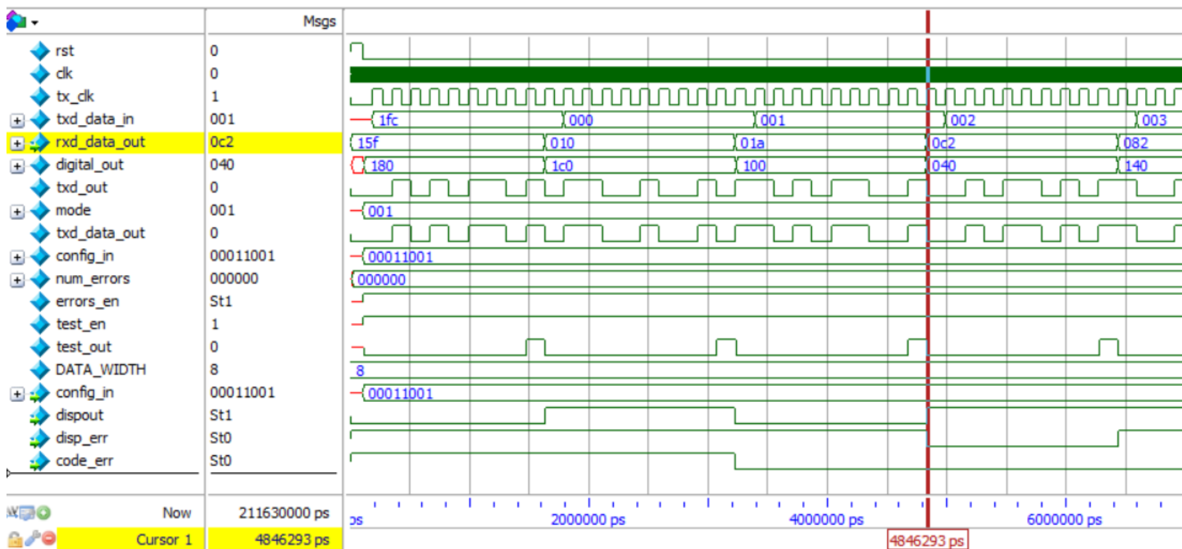

Figure 47 - GLS - Operation Mode 0 B – Functional mode – Test enabled

3.1.3.3. GLS - Operation Mode 1 – Parallel loopback

Figure 48 below, shows that the *SerDes* parallel loopback mode is having the same functionality as the one described in chapter 2.3.3. In this mode, a serial encoded data {rxa_in_p} and {rxa_in_n} in injected into the *SerDes* and the parallel output {digital_out} must match the injected data. Which in this case is 0, 1 and 2.

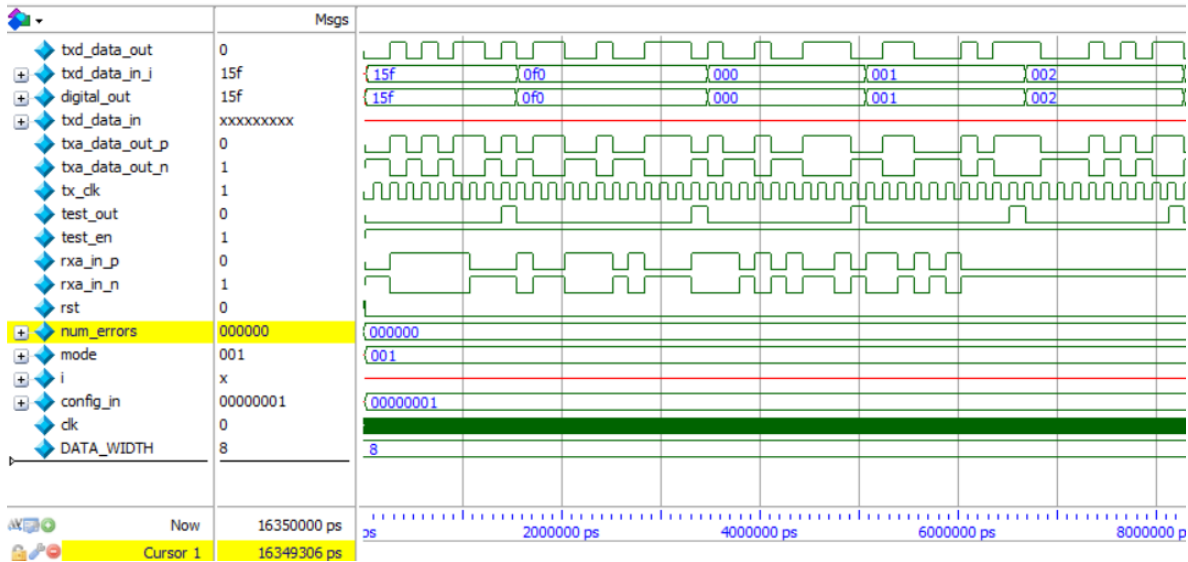

Figure 48 - GLS - Operation Mode 1 – Parallel loopback

3.1.3.4. GLS - Operation Mode 2 – Serial loopback

Figure 49 below, shows the GLS simulation for the serial loopback, this mode is described in detail in chapter 2.3.4. In this mode, a parallel data {txd_data_in} in injected into the *SerDes* and internally the serial data outputting this module is fed to the digital receiver.

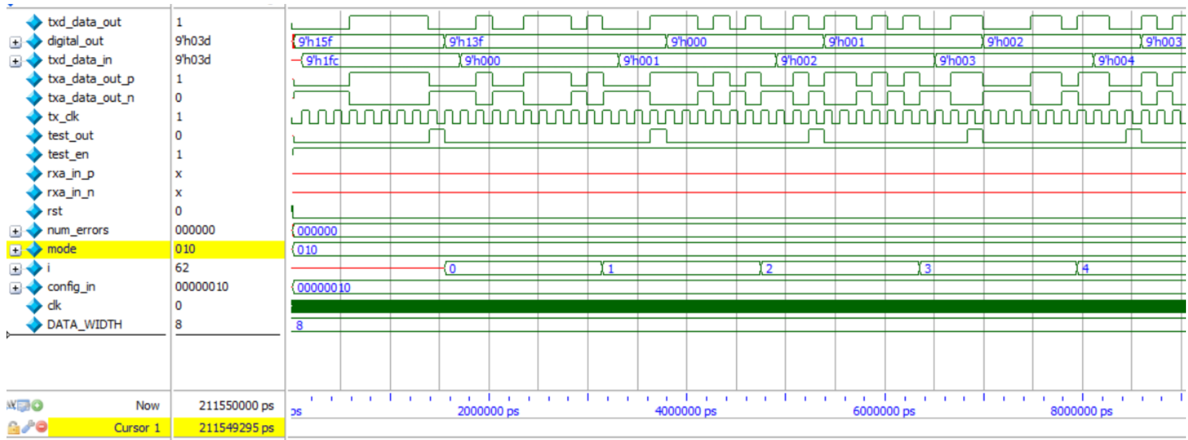

Figure 49 - GLS - Operation Mode 2 – Serial loopback

3.1.3.5. GLS - Operation Mode 3 – RXA bypass

In Figure 50 it is shown the GLS simulation of the analog receiver bypass mode. This mode consist of injecting the serial data to the digital receiver directly through the fourth bit of the configuration input {config_in}. These simulation results are similar to the RTL one described in chapter 2.3.5.

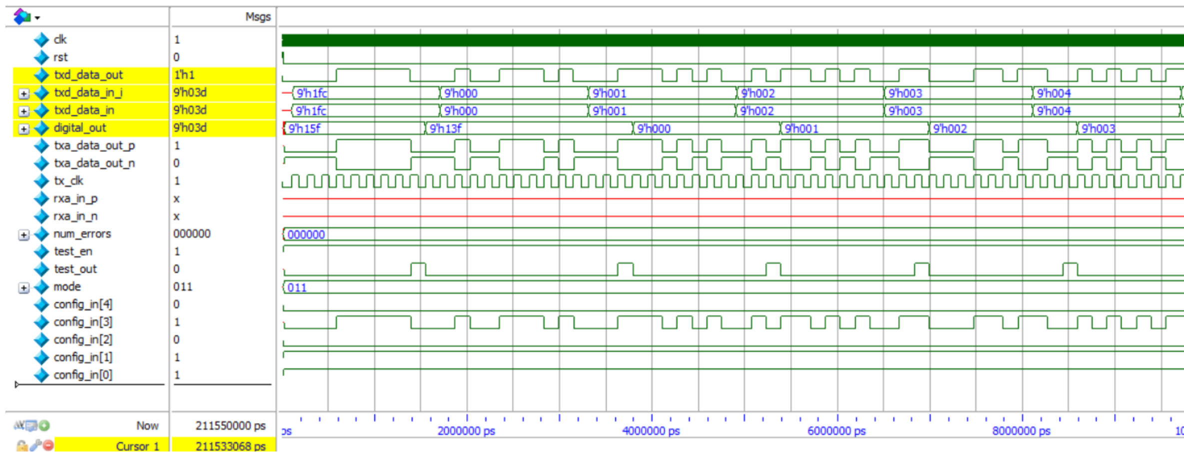

Figure 50 - GLS - Operation Mode 3 – RXA bypass

3.1.3.6. GLS - Operation Mode 4 – BIST with serial loopback

Figure 51 shows that the BIST mode with serial loopback is also having the same results for the GLS simulation as the one described in chapter 2.3.6. In this mode the LSFR is generating a parallel pattern {lsfr_parallel_out} which is later injected to the digital receiver creating an internal loopback of the serial data from the transmitter.

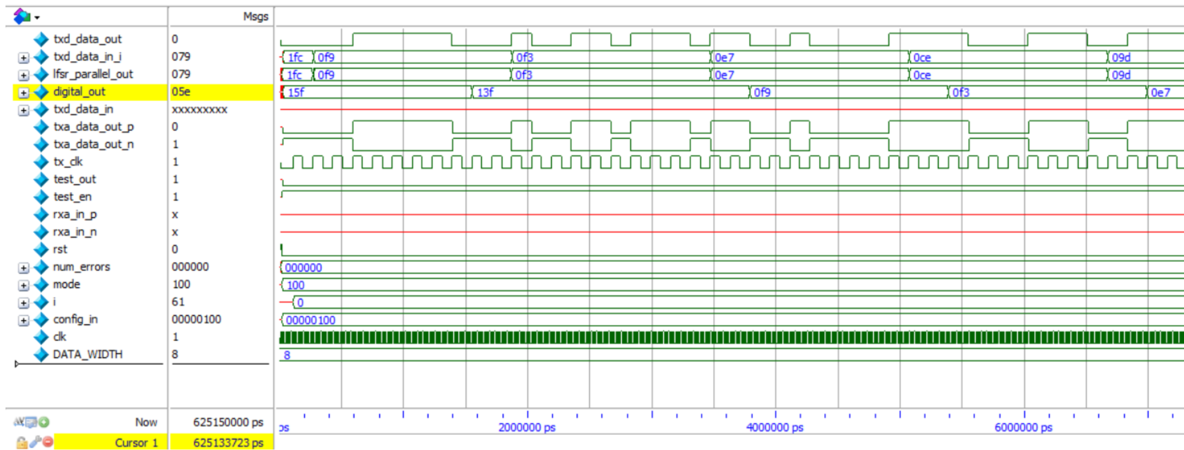

Figure 51 - GLS -Operation Mode 4 – BIST with serial loopback

3.1.3.7. GLS - Operation Mode 5 – RXA bypass with parallel loopback

Figure 52 shows the simulation of the operating mode when we are bypassing the analog receiver and at the same time creating an internal parallel loopback connecting the data coming out of the digital receiver {rxd_data_out} to the digital transmitter's input {txd_data_in}. This mode is described in detail in chapter 2.3.7.

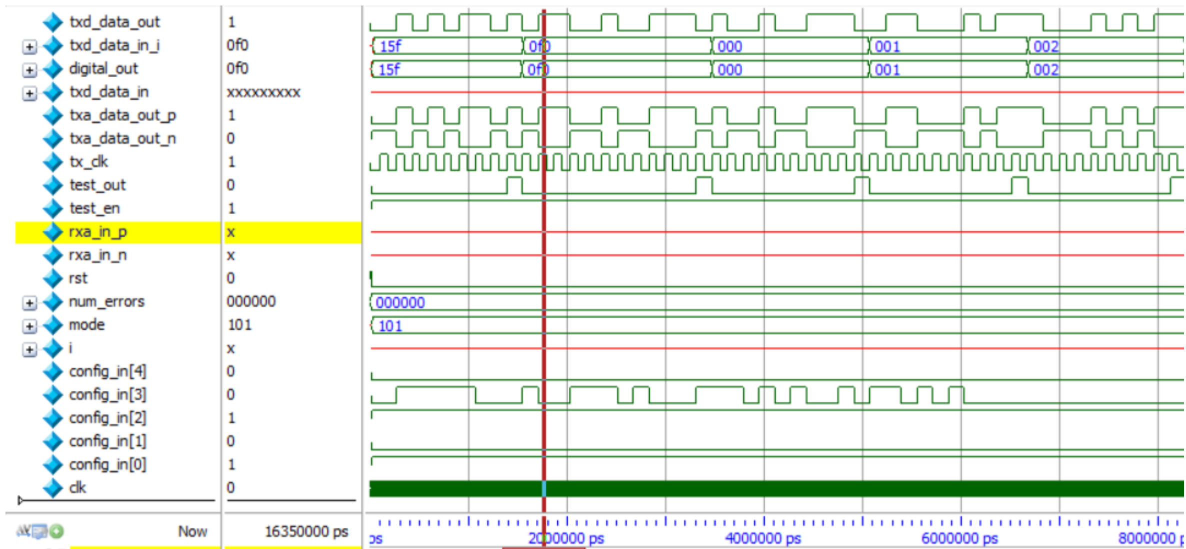

Figure 52 - GLS - Operation Mode 5 – RXA bypass with parallel loopback

3.1.3.8. GLS - Operation Mode 6 – Open BIST

Open BIST mode is intended to flag errors when nothing is connected. A serial loopback can also be created outside externally. In the following simulations, the intention is to notice how the errors increase when the circuits is open.

Figure 53 shows the open BIST mode when the inputs of the serializer aren't connected. In such simulation we can see that {test_out} has X's as the same as {num_errors}. In the simulation described in chapter 2.3.8, this behavior is not happening; this is because in a non GLS model the X's are being correctly compared with the non-synthesizable operand of ===.

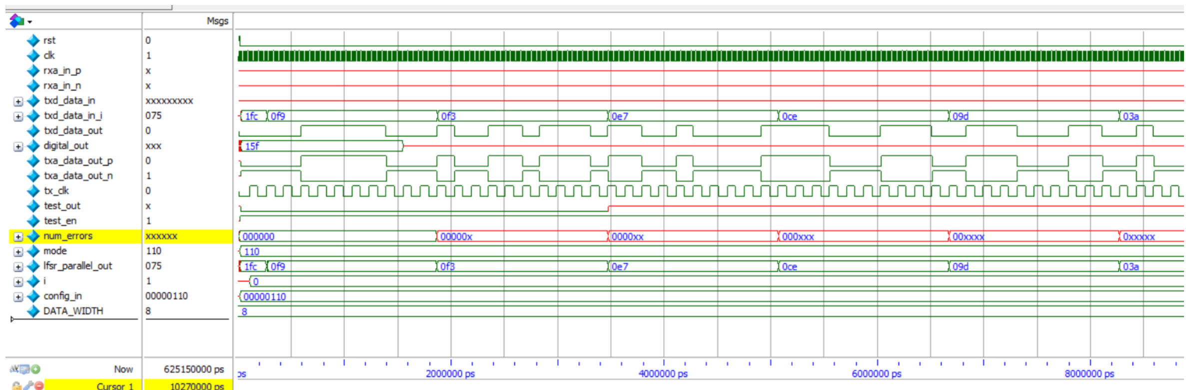

Figure 53 - GLS - Operation Mode 6 – Open BIST - Unconnected

For avoiding this, it is recommended to connect the analog input {rxa_in_p} to a VCC and

{rxa_in_n} to GND. Figure 54, shows the difference when the serial input has a fixed value. In this case the errors counter {num_errors} is correctly increasing on every mismatch.

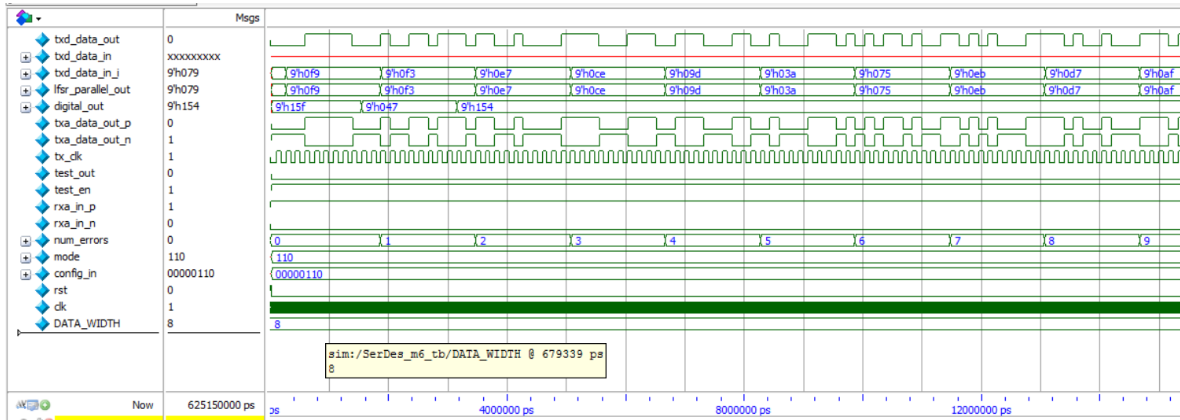

Figure 54- GLS - Operation Mode 6 – Open BIST - Analog receiver set to 1

3.1.3.9. GLS - Operation Mode 7 – RXA Output with analog loopback

Figure 55 shows the GLS simulation of the operating mode when the receiver output is set to be shown on the output pin {test_out}. In this mode, described in detail in chapter 2.3.9, it is also intended to create an analog loopback of the serial data coming from the analog receiver and connect it to the analog transmitter internally.

The simulation in GLS is having the same results as the one pictured in Figure 38.

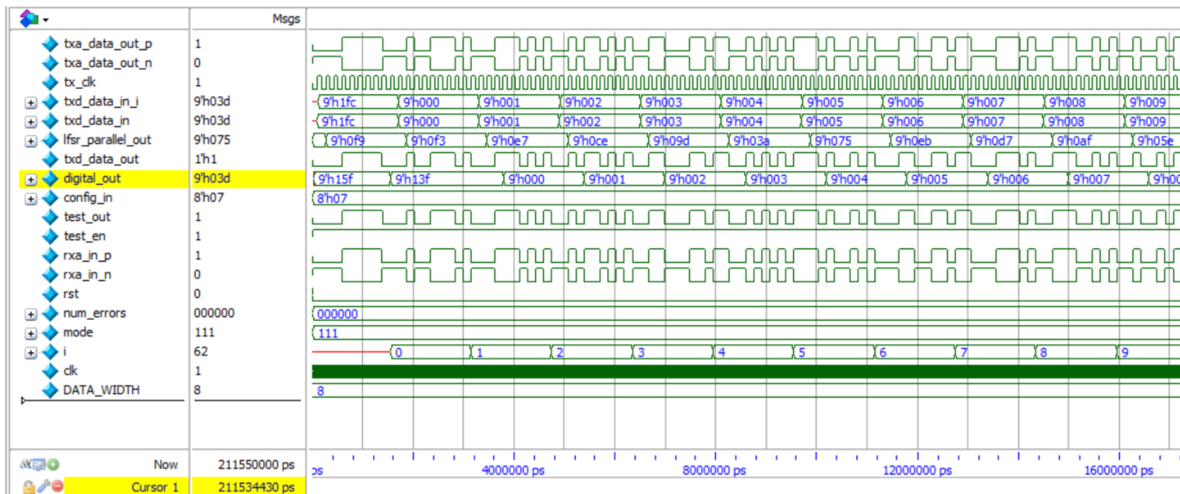

Figure 55 - GLS - Operation Mode 7 – RXA Output with analog loopback

Chapter 4: Physical synthesis and Layout Verification

This chapter covers the physical synthesis of the *test_modules* circuit as if it was going to be a final individual circuit. It describes the steps performed for the complete flow of the physical synthesis of this module. This chapter also contains the DRC verification results obtained in this thesis.

4.1. Physical synthesis

After running **RTL Compiler** for the logic synthesis, the resulting files (*test_modules*_m.sdc and *test_modules*_m.v) were used as inputs to be used by **Encounter** tool which will perform the physical Synthesis.

The full process to get the physical synthesis can be hard if done manually, we are reducing this process by using of a script that will help us replicate the repetitive work faster. This script has all the setup, analysis and optimization needed for the full synthesis of a digital module and just by changing a few parameters it can be adjustable to any circuit using this same technology. All the steps have been added to the script *Full_Synthesis_EDI_test_modules.tcl* so that a full synthesis flow can be performed for any circuit in this technology just by changing a few setups configurations: for instance, some of the setups needed for the circuit is to add power and ground pins to the logic gate HDL file. In our case, the pads were added to circuit, as well as the input ports in the top module for VDD, VSS, DVDD and DVSS:

```
PDVDD pad_DVDD(.DVDD (DVDD));
PDVSS pad_DVSS(.DVSS (DVSS));
PVDD pad_VDD(.VDD (VDD));
PVSS pad_VSS(.VSS (VSS));
```

Then standard pads were added to connect all the ports of the circuit. To do this, all the inputs and outputs were declared as a wires with and changed their name to <*signal_name*>_w. This is to connect wire to the input of the pad and the output of the pad

component to the real physical output port of the circuit.

For the input ports the following format was used:

```
  PIC pad_<signal_name>(.P (<signal_name>), .IE (VDD), .Y
(<signal_name>_w));
```

For the output ports the following format was used:

```
  POC4C pad_<signal_name>(.A (<signal_name>_w), .P
(<signal_name>));
```

The number of pads used needs to be divisible by 4 in order to make the circuit a symmetric square. In total, the *test_modules* circuit has 69 inputs and outputs ports. In order to make this number divisible by 4, 3 dummy pads were added to the circuit. The format used for these dummy pads is the following:

```
  PIC pad_dummy1(.P (dummy_in1), .IE (n_15), .Y (VSS));
```

In order to close the pad frame, corners also had to be added to the bottom left, bottom right, top left and top right of the circuit to this file, that is:

```
  PCORNER se_pcorner();
  PCORNER sw_pcorner();
  PCORNER ne_pcorner();
  PCORNER nw_pcorner();
```

Then all the inputs, outputs and corner pads have to be added to the input and output file (*test_modules*_arm.ioc). These is where we select the distribution of the pads in the circuits, so 18 ports were distributed between top, bottom, left and right side of the circuit.
An example of how this file is modified can be seen in the appendix.

After doing all these modifications we will run **Encounter** with the script as an input file. **Encounter** will then automatically generate a floorplan, a power grid and do the place and route of the circuit.

4.1.1. Connectivity, geometry and DRC report.

The script used for the full Synthesis flow, *Full_Synthesis_EDI_test_modules.tcl,* will also run connectivity, geometry and DRC violation checks. The results of these checks are shown in Figure 56, Figure 57 and Figure 58 respectively.
The circuit didn’t have any violation of this type.

```
VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Jul  5 22:25:34 2016

Design Name: test_modules
Database Units: 1000
Design Boundary: (0.0000, 0.0000) (2108.0000, 2108.0000)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Jul  5 22:25:35 2016
Time Elapsed: 0:00:01.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.2  MEM: 0.000M)
```

Figure 56 - Connectivity Verification

```
<CMD> verifyGeometry
 *** Starting Verify Geometry (MEM: 868.9) ***

  VERIFY GEOMETRY ...... Starting Verification
  VERIFY GEOMETRY ...... Initializing
  VERIFY GEOMETRY ...... Deleting Existing Violations
  VERIFY GEOMETRY ...... Creating Sub-Areas
                  ...... bin size: 2560
VG: elapsed time: 3.00
Begin Summary ...
  Cells       : 0
  SameNet     : 0
  Wiring      : 0
  Antenna     : 0
  Short       : 0
  Overlap     : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:03.0  MEM: 108.7M)
```

Figure 57 - Geometry Verification

```
*** Starting Verify DRC (MEM: 977.6) ***

 VERIFY DRC ...... Starting Verification
 VERIFY DRC ...... Initializing
 VERIFY DRC ...... Deleting Existing Violations
 VERIFY DRC ...... Creating Sub-Areas
 VERIFY DRC ...... Using new threading
 VERIFY DRC ...... Sub-Area : 1 of 16
 VERIFY DRC ...... Sub-Area : 1 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 2 of 16
 VERIFY DRC ...... Sub-Area : 2 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 3 of 16
 VERIFY DRC ...... Sub-Area : 3 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 4 of 16
 VERIFY DRC ...... Sub-Area : 4 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 5 of 16
 VERIFY DRC ...... Sub-Area : 5 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 6 of 16
 VERIFY DRC ...... Sub-Area : 6 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 7 of 16
 VERIFY DRC ...... Sub-Area : 7 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 8 of 16
 VERIFY DRC ...... Sub-Area : 8 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 9 of 16
 VERIFY DRC ...... Sub-Area : 9 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 10 of 16
 VERIFY DRC ...... Sub-Area : 10 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 11 of 16
 VERIFY DRC ...... Sub-Area : 11 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 12 of 16
 VERIFY DRC ...... Sub-Area : 12 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 13 of 16
 VERIFY DRC ...... Sub-Area : 13 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 14 of 16
 VERIFY DRC ...... Sub-Area : 14 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 15 of 16
 VERIFY DRC ...... Sub-Area : 15 complete 0 Viols.
 VERIFY DRC ...... Sub-Area : 16 of 16
 VERIFY DRC ...... Sub-Area : 16 complete 0 Viols.

 Verification Complete : 0 Viols.

*** End Verify DRC (CPU: 0:00:02.1  ELAPSED TIME: 2.00  MEM: -12.4M) ***
```

Figure 58 - DRC verification

4.1.2. Timing analysis

Static timing analysis (STA) is a method of validating the timing performance of a design by checking all possible paths for timing violations in their worst-case conditions. It considers the worst possible delay through each logic element, but not the logical operation of the circuits.

In STA, the word static alludes to the fact that this timing analysis is carried out in an input-independent manner. It locates the worst-case delay of the circuit over all possible input combinations. There are huge numbers of logic paths inside a chip of complex design. The advantage of STA is that it performs timing analysis on all possible paths (whether they are real or potential false paths).

However, it is worth noting that STA is not suitable for all design styles. It has proven efficient only for fully synchronous designs. Since the majority of chip design is synchronous, it has become a mainstay of chip design over the last few decades.

Encounter tool will perform a STA and present the results in the reports. However, to improve this results we have added to the script commands to run timing optimization methods to try to improve the timing results of the final circuits and try to don't violate the timing constraints.

After running the STA (Static Timing Analysis), it was found that with the expected frequency of 125 MHz, the circuit was not complying with the setup timing constraints as shown on Figure 59.

timeDesign Summary

Setup mode	all	reg2reg	reg2cgate	default
WNS (ns):	-0.473	-0.473	N/A	-0.041
TNS (ns):	-3.421	-3.421	N/A	-0.041
Violating Paths:	19	19	N/A	1
All Paths:	239	105	N/A	239

Figure 59 - Setup timing

It was analyzed that the violating paths resided inside the comparator as it is shown on Figure 60. These violations were root caused to appear due to the combinational logic involved in comparison on the fly. So it was shown that by using only one memory block, even when we gained some logic area, the maximum setup timing was affected.

```
Path 1: VIOLATED Setup Check with Pin comparator0/cntB_reg[2]/CK
Endpoint:   comparator0/cntB_reg[2]/D (^) checked with  leading edge of '125MHz_
CLK'
Beginpoint: comparator0/cntB_reg[1]/Q (v) triggered by  leading edge of '125MHz_
CLK'
Path Groups: {reg2reg}
Analysis View: WC_Analysis_View
Other End Arrival Time          0.130
- Setup                         1.202
+ Phase Shift                   8.000
= Required Time                 6.928
- Arrival Time                  7.402
= Slack Time                   -0.473
     Clock Rise Edge                      0.000
     + Source Insertion Delay             0.050
     + Network Insertion Delay            0.100
     = Beginpoint Arrival Time            0.150
```

Figure 60 - Example of timing violating path

As all the violating paths were inside the comparator, which is mostly used for testing. A reduction of the maximum frequency for testing was accepted. This compromise on frequency was of 12 %, giving a maximum frequency for testing of 111 MHz. All the following results were obtained using this frequency.

```
------------------------------------------------------------
     timeDesign Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg |reg2cgate| default |
+--------------------+---------+---------+---------+---------+
|           WNS (ns):|  0.054  |  0.054  |   N/A   |  0.118  |
|           TNS (ns):|  0.000  |  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |    0    |   N/A   |    0    |
|          All Paths:|   239   |   105   |   N/A   |   239   |
+--------------------+---------+---------+---------+---------+
```

Figure 61 - Setup timing 111 MHz

On Figure 62 it is seen that the Hold time constraint is positive, meaning that is complies with the constraints but only by a small margin.

```
------------------------------------------------------------
     timeDesign Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg |reg2cgate| default |
+--------------------+---------+---------+---------+---------+
|           WNS (ns):|  0.727  |  0.727  |   N/A   |  1.425  |
|           TNS (ns):|  0.000  |  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |    0    |   N/A   |    0    |
|          All Paths:|   239   |   105   |   N/A   |   239   |
+--------------------+---------+---------+---------+---------+
```

Figure 62 - Hold Timing 111 MHz

4.1.3. Layout generation

The preliminary layout design before exporting it to **Virtuoso** as a *gds* file was generated by **Encounter** and is shown on Figure 63.

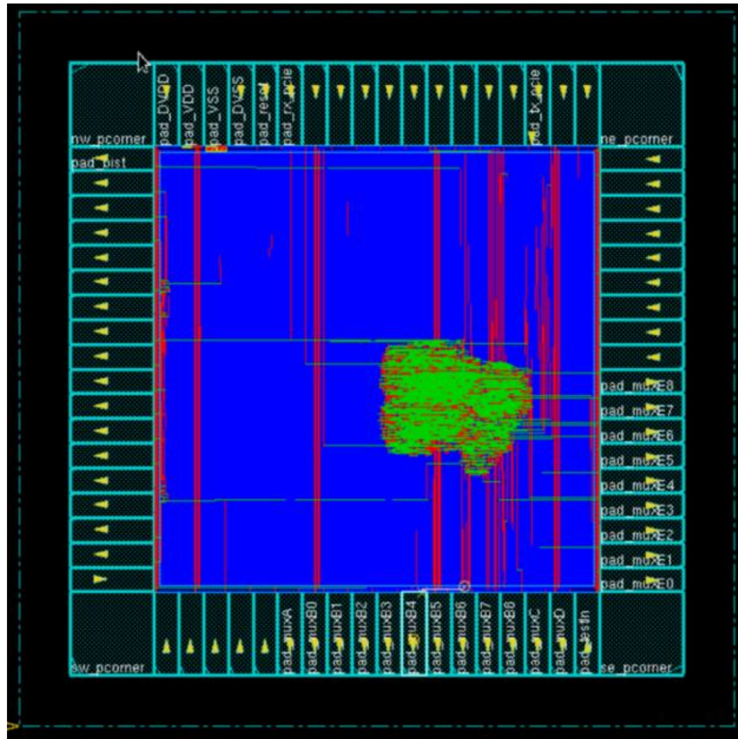

Figure 63 - test_modules preliminary layout

4.1.4. GDS stream import to Virtuoso

Graphic Database System (GDS) stream format is a database file format which is the de facto industry standard for data exchange of integrated circuit or IC layout artwork. It is a binary file format representing planar geometric shapes, text labels, and other information about the layout in hierarchical form. The data can be used to reconstruct all or part of the artwork to be used in sharing layouts, transferring artwork between different tools, or creating photomasks.

After the preliminary layout from encounter was generated, the next step is to convert this file into a GDS stream file for **Virtuoso**. Figure 64, presents the final GDS file obtained this way.

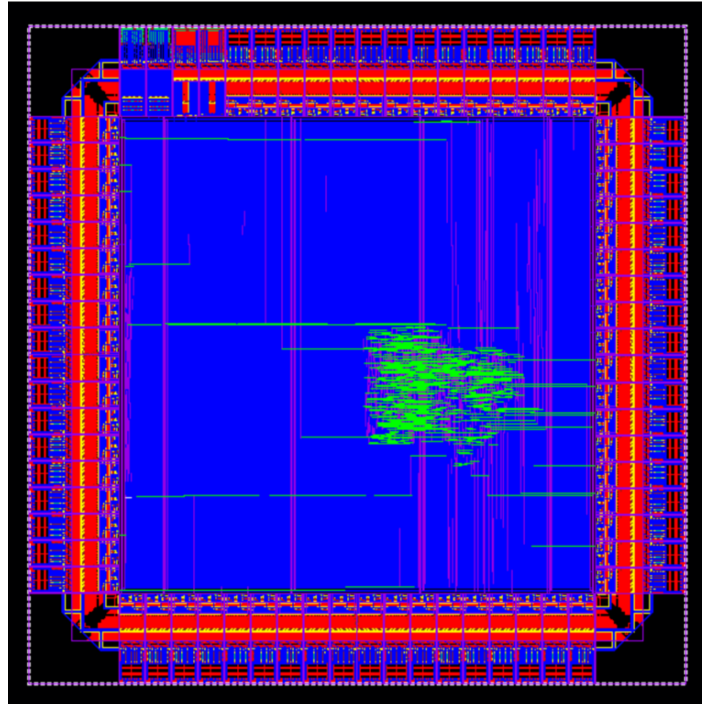

Figure 64 - GDS Virtuoso Stream

4.2. Layout Verification (DRC)

Design Rules are a series of parameters provided by semiconductor manufacturers that enable the designer to verify the correctness of a mask set. Design rules are specific to a particular semiconductor manufacturing process. A design rule set specifies certain geometric and connectivity restrictions to ensure sufficient margins to account for variability in semiconductor manufacturing processes, so as to ensure that most of the parts work correctly.

The next step in the design flow is to clean all the Design Rules Checking (DRC). For this, we ran **Calibre** checks and we found several errors in the design. For this module, the layout verification tool found 2322 errors as it is shown on Figure 65.

Figure 65 - Calibre Errors

All these errors had to be cleaned for this circuit. Most of the errors found were caused due to antenna errors originated by connections in the top metal layer. The first approach to clean these errors was to add diodes to the areas which had problems as shown in Figure 66. However, even though this approach cleaned this kind of errors, the amount of diodes necessaries to clean the circuit entirely was too much. It was then decided to go by a second approach of limiting the layers of metal generated by encounter, giving a max top routing layer level of 3. This was possible by changing the following parameters in the synthesis script *Full_Synthesis_EDI_test_modules.tcl*:

```
setNanoRouteMode -routeBottomRoutingLayer 0
setNanoRouteMode -routeTopRoutingLayer 3
```

By doing this, all these errors were fixed in the design.

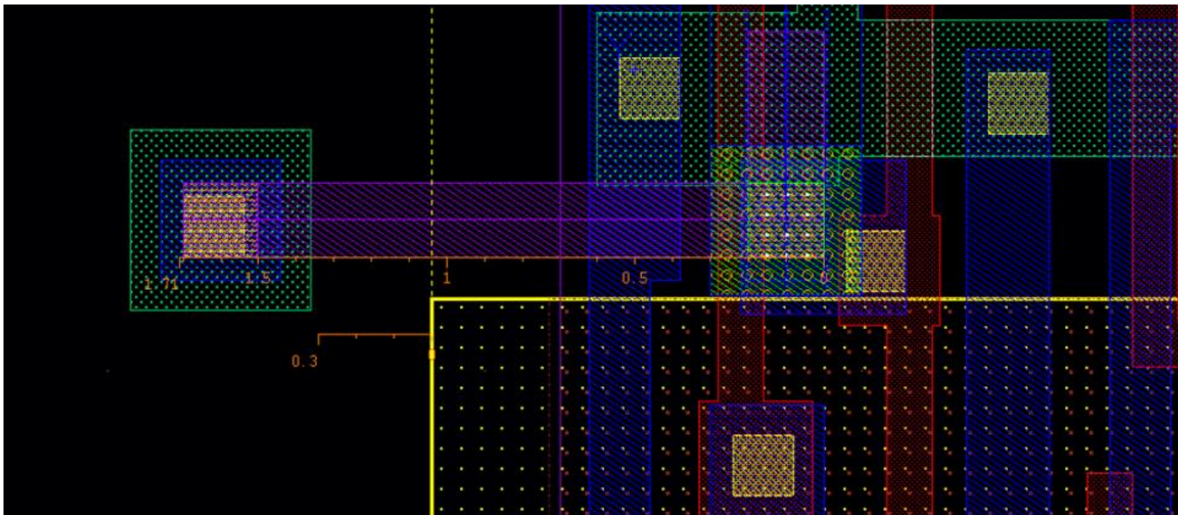

Figure 66 - Diode to fix DRC antenna errors

The other kind of errors that appeared the most in the circuit were fixed by adding a layer of GRLOGIC to all the circuit. This is for all the circuit to be treated as standard cells.

There were also errors caused by of spaces between the layers of NW (See Figure 67), these errors were fixed by completing the holes with NW-layer extension (See Figure 68).

Rule File Pathname: /home/aarias/climones/_cmrf8sf.drc.cal_
NW(generated) space (at different potentials) >= 0.92

Rule File Pathname: /home/aarias/climones/_cmrf8sf.drc.cal_
Layout will cause: BH to adjacent NW >= 0.40
home : bash

Figure 67 - NW spacing DRC errors

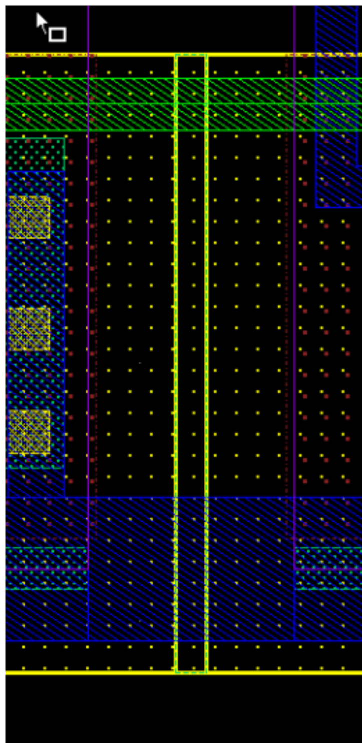

Figure 68 - Filled space of NW

After cleaning all the previous errors, 17 errors caused by latch-up (Figure 69) remained. These errors were fixed by adding nwCont instances to the affected areas (Figure 70).

Show Unresolved | test_modules, 553 Results (in 2 of 1657 Checks)

Check / Cell	Results
Check GRLUP10	17
Check GR594_M3	536

1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17

Rule File Pathname: /home/aarias/climones/_cmrf8sf.drc.cal_
(Total area of [(RX intersect NW) touching CA] outside BP)/
(Total(PC intersect RX) inside NW) over local (2.1*LUP1)
area stepped in ((2.1*LUP01)/2) >= 0.01

Figure 69 - Latch-Up DRC errors

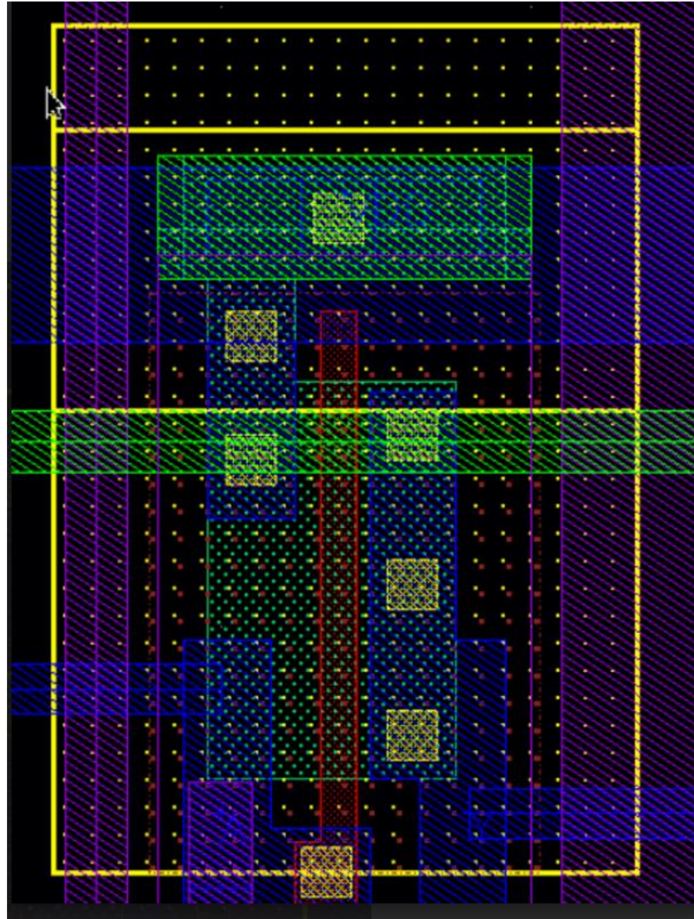

Figure 70 - Latch-Up Error Fix Example

The remaining 490 errors were caused by a ratio violation of M3 and NW (Figure 71). For fixing these errors, we increased the density of metal M3 over the layer of NW by adding connectors (M1-M2-M3) and a layer of M3 over the VDD bus (Figure 72).

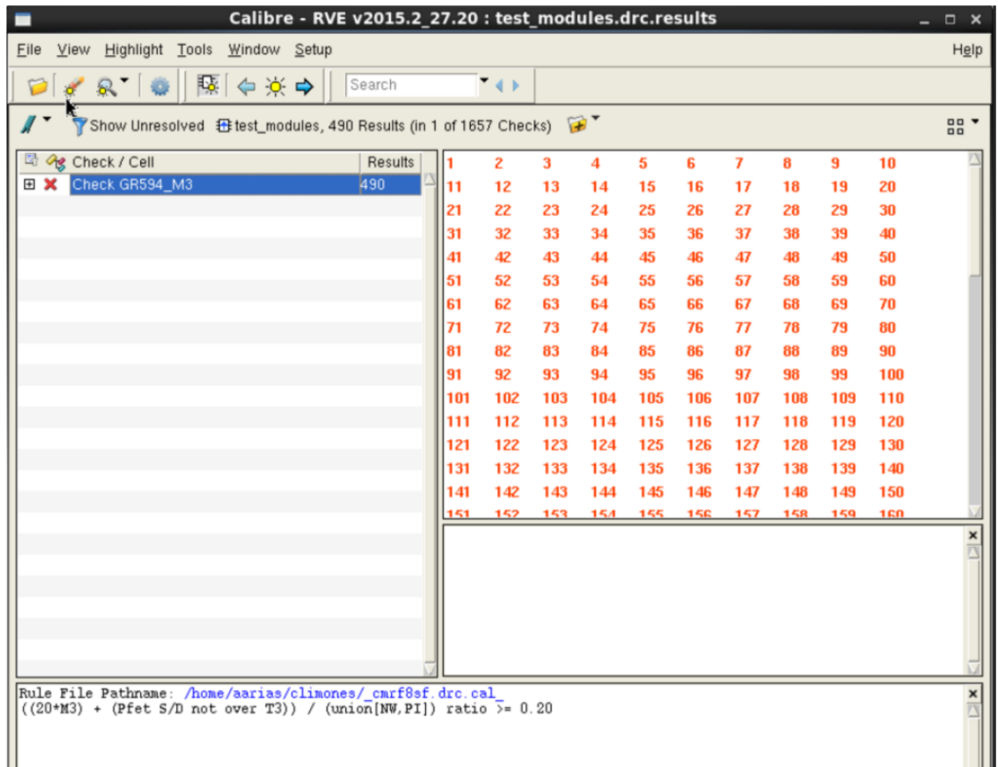

Figure 71 - M3 - NW Ratio DRC errors

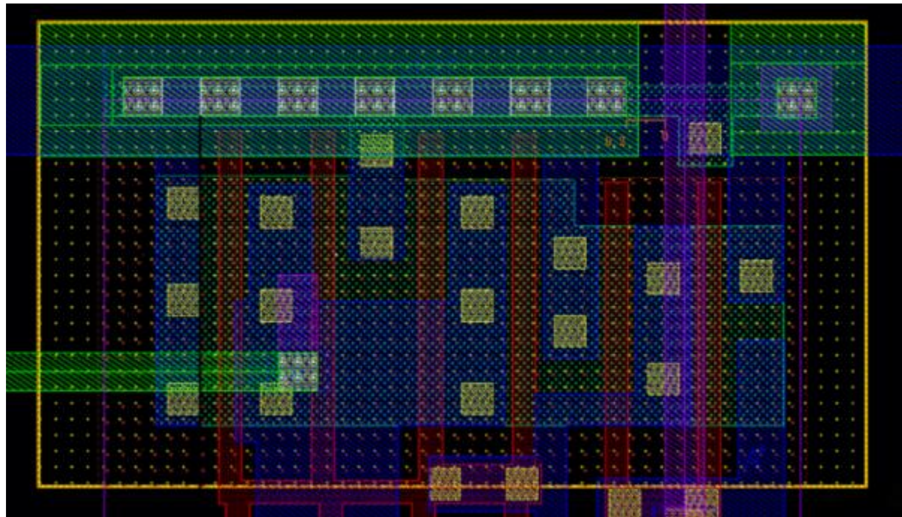

Figure 72 - Fixing M3- NW Ratio errors Example

With this, all the DRC errors were cleaned (see Figure 73).

```
RULECHECK GR595_M2 ................ TOTAL Result Count = 0          (0)
RULECHECK GRT3W595_M2 ............. TOTAL Result Count = 0          (0)
RULECHECK GR594_M3 ................ TOTAL Result Count = 0          (0)
RULECHECK GRT3W594a_M3 ............ TOTAL Result Count = 0          (0)
RULECHECK GRT3W594b_M3 ............ TOTAL Result Count = 0          (0)
RULECHECK GR595_M3 ................ TOTAL Result Count = 0          (0)
RULECHECK GRT3W595_M3 ............. TOTAL Result Count = 0          (0)
--------------------------------------------------------------------------------
--- RULECHECK RESULTS STATISTICS (BY CELL)
---
--------------------------------------------------------------------------------
--- SUMMARY
---
TOTAL CPU Time:                  121
TOTAL REAL Time:                 128
TOTAL Original Layer Geometries: 77937 (4106060)
TOTAL DRC RuleChecks Executed:   1655
TOTAL DRC Results Generated:     0 (0)
```

Figure 73 - DRC Final Results

Conclusions

This thesis proposes a new architecture for testing the *SerDes* system designed in 130 nm CMOS technology by the team formed by the second cohorts of specialty in system on chip design. This architecture will aid to perform the verification of the Chip before it is manufactured. It will also help to detect malfunctions in any of the blocks of the *SerDes*.

This architecture proposes the use of a test module composed by a signal driver, a LFSR pseudo random pattern generator and a comparator.

The signal driver module was implemented to interconnect the signals between the internal modules. It makes use of multiplexers between the internal signals of the SerDes to fulfill its purpose. This design is clear and can be easily modified to add or remove driven signals.

The comparator module will check sent and received data and flag mismatches between them. By the use of a Finite State Machine (FSM), the final design area and timing were optimized compared against earlier drafts. However, the current design is still not complying with the timing requirements proposed for this project. The design of this module can still be improved but this is left as future project work.

The LSFR module generates pseudo-random pattern data for testing that is injected to the digital transmitter parallel input. It uses a fixed seed set to a "comma" symbol value to start the transmission of the BIST mode on reset. The initial value of the output registers is determined by fixed seed. The *LFSR* seed is set to start on reset to the initial value of a "comma" symbol, then on every clock will generate the next data. The *LFSR* module can generate a total of 61 random patterns from "comma" to "comma" before restarting the sequence.

The circuit is able to perform on functional mode as well as eight different operating modes for testing. Some of these testing modes can perform a double function, hence reducing by half the number of test operating modes from previous proposal.

As for the functional verification, the behavioral model's validation was achieved for the test modules to be integrated to full *SerDes*. Also, all the operating modes have been simulated

and verified in both GLS and RTL.

Physical synthesis and timing analysis were performed for the test modules block. The results show that the operation frequency of the *SerDes* has to be relaxed when testing is enabled. The maximum frequency accomplished when the comparator is enabled was of 110 MHz, i.e., 88.8% of the expected 125 MHz frequency of PCI-e 1 protocol.

DRC verification was performed on the layout of this module and all the errors were identified and fixed. Most of them were cleaned by small changes the synthesis driven by a tcl script, however the remaining errors had to be cleaned manually in Virtuoso. The final layout of test module was clean of any DRC errors.

Unfortunately LVS checking and cleanup was not completed for this thesis. This check would guarantee that the final layout really represents the circuits we want to fabricate. This check will remain as future project work.

The synthesis of the standalone test modules block requires 72 PADs. The final *SerDes* system uses only 40 PADs, 32 less than the test module, this is because many of the inputs and outputs of the test modules block are internal signals of the *SerDes* and do not require an external pin when integrating the rest of the modules. Because the size of the PADs is fixed in this technology, the final layout for the test modules has noticeable unused area. The expectation for the synthesis of the full *SerDes* system is that the total area used will be less that the one presented in this project.

Testing results were obtained using the serializer and deserializer modules from previous works in [13]. The next phase of this project is the integration of an optimized serializer and deserializer and make sure that the functional simulation behaves as expected.

As future work, one optimization that can be performed is enabling the reseeding of the LFSR module. Also the setup timing results for the comparator could be improved by reducing the combinational logic of this module.

References

[1] A. Patel, "The basics of SerDes (serializers/deserializers) for interfacing," 16 09 2010. [Online]. Available: http://www.planetanalog.com/document.asp?doc_id=528099#msgs.

[2] M. D. Kumar, Implementation of PCS of physical layer for PCI Express, Rourkela: M.S. Thesis, National Institute of Technology, 2007-2009.

[3] R. Budruk, D. Anderson and T. Shanley, PCI Express System Architecture, Addison-Wesley, 2003, pp. 41-54.

[4] K. Yong-woo and K. Jin-ku, "An 8B/10B encoder with a modified coding table," *Proc. Asia Pacific Conference on Circuits and Systems,* pp. 1522-1525, 2008.

[5] Lattice Semiconductor, "8b/10b Encoder/Decoder - Documentation," 2015.

[6] National Instruments, "PCI Express – An Overview of the PCI Express Standard," 05 Nov 2014. [Online]. Available: http://www.ni.com/white-paper/3767/en/.

[7] M. A. Nummer, A DFT Technique for Testing High-Speed Circuits with Arbitrarily Slow Testers, Waterloo, Ontario: M.S. Thesis, University of Waterloo, 2001.

[8] S. Sengupta, S. Kundu, S. Chakravarty, P. Parvathala, R. Galivanche, G. Kosonocky, M. Rodgers and T. Mak, "Defect-Based Test: A Key Enabler for Successful Migration to Structural Test," *Intel Technology Journal,* Q1 1999.

[9] G. Jervan, Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, Linköping: Department of Computer and Information Science Linköpings universitet, 2005.

[10] A. A. Al-Yamani and E. J. McCluskey, Built-In Reseeding For Serial Bist, S. University, Ed., Stanford, California: Center for Reliable Computing, 2003.

[11] R. Godínez Maldonado, Test Module Design for ITESO TV1 SerDes, Guadalajara, Jalisco, Mexico: Project Report ESC.-ITESO, Dic. 2015.

[12] Y. Yu, Z. Yang, X. Peng and D. Xu, "Efficient Concurrent BIST with Comparator-based Response Analyzer," *2013 IEEE International Instrumentation and Measurement Technology Conference (I2MTC),* pp. 1115-1119, 6-9 May 2013.

[13] J. M. Centeno Quiñonez, C. Gonzalez Morales, V. A. Cuauhtémoc Aguilera and A. Giron, ITESO TV1. RTL Design for the PCIe deserializer module, Project Report ESC.-ITESO, Dic 2015.

[14] P. A. Franaszek and A. X. Widmer, "A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code," *IBM Journal of Research and Development,* 1983.

[15] J. C. Chen, "Multi-Gigabit SerDes: The Cornerstone of High Speed Serial Interconnects," *Genesys Logic America, Inc.,* 15 Sept. 2003.

[16] D. Lewis, "SerDes Architectures and Applications," *National Semiconductor Corporation,* 2004.

[17] R. J. Feugate and S. M. McIntyre, "Introduction to VLSI Testing," Prentice Hall, 1988.

[18] E. B. Eichelberge and E. Lidbloom, "Random Pattern Coverage Enhancement and

Diagnosis for LSSD Logic Self-Test," *IBM Journal of Research and Development,* vol. 27, no. 3, pp. 265-272, May 1983.

[19] R. Sharma and K. K. Saluja, Theory, Analysis and Implementation of an On-Line BIST Technique, 1 ed., vol. 1, Gordon and Breach Science Publishers S.A., 1993, pp. 99-22.

[20] S. Sameer, Complete Computer Hardware Only, PediaPress, 2015, pp. 367-381.

[21] A. Athavale and C. Christensen, "High-Speed Serial I/O Made Simple. A Designer's Guide, with FPGA Applications," *Connecticity Solutions,* 2005.

[22] Intel Corporation, "PHY Interface For the PCI Express, SATA, and USB 3.1 Architectures," Intel Corporation, 2014. [Online]. Available: http://www.intel.com/content/www/us/en/io/pci-express/phy-interface-pci-express-sata-usb31-architectures.html.

[23] M. L. Agrawal and D. Vishwani, "BIST Pattern Generation," in *Essentials of Electronic Testing For Digital, Memory & Mixed-Signal VLSI Circuits*, New York, Boston, Dordrecht, London, Moscow, Kluwer Academic Publishers, 2002, pp. 498-512.

[24] I. Voyiatzis, D. Gizopoulos and F. Makri, An Input Vector Monitoring Concurrent BIST Architecture Based on a Precomputed Test Set, Athens: Technological Educational Institute of Athens, 2008.

Appendix

7.1. Table: archives of test module

Name of archive	Library/Path	Extension (or format)	Brief description
test_modules.sv	/home/aarias/climones/EDI_PROYECT/RC	System Verilog	Top level RTL module
LFSR.sv	/home/aarias/climones/EDI_PROYECT/RC	System Verilog	RTL module
comparator.sv	/home/aarias/climones/EDI_PROYECT/RC	System Verilog	RTL module
Signal_driver.sv	/home/aarias/climones/EDI_PROYECT/RC	System Verilog	RTL module
logic_synthesis.tcl	/home/aarias/climones/EDI_PROYECT/RC	tcl	Script to perform the logic synthesis in RTL Compiler
test_modules_m.v	/home/aarias/climones/EDI_PROYECT/EDI/	verilog	Verilog file of the synthesized module
test_modules_Typical_WC_Analysis.view	/home/aarias/climones/EDI_PROYECT/EDI/	view	View file for the worst case analysis
test_modules_m.sdc	/home/aarias/climones/EDI_PROYECT/EDI/	sdc	Contraint file generated by RTL Compiler
test_modules.globals	/home/aarias/climones/EDI_PROYECT/EDI/	globals	File that have all the global contraints for encounter
test_modules_frame_arm.ioc	/home/aarias/climones/EDI_PROYECT/EDI/	ioc	Pad file for encounter
Test_modules.ctstch	/home/aarias/climones/EDI_PROYECT/EDI/	ctstch	Timing constraint file to be used by Encounter
Full_Synthesis_EDI_test_modules.tcl	/home/aarias/climones/EDI_PROYECT/EDI/	tcl	Script to perform the physical synthesis in Encounter
Test_modules_opt_script.enc	/home/aarias/climones/EDI_PROYECT/EDI/EDI_IMPLEMENTATION_limit_metals	enc	Layout file for Encounter
Test_modules_metal_limit.gds	/home/aarias/climones/EDI_PROYECT/EDI/EDI_IMPLEMENTATION_limit_metals	gds	File generated by Encounter to be exported to Virtuoso.
test_modules	test_modules2	gds	This is the final stream file used in Virtuoso

7.2. Glossary

8b/10b encoding – A scheme for encoding signals with an embedded clock. The encoding serves two purposes. First, it ensures that there are enough transitions in the data stream for clock recovery and, second, that the number of 0s and 1s is matched, maintaining DC balance in AC-coupled systems.

Peripheral component interconnect (PCI) – A high-speed parallel bus originally designed by Intel to connect I/O peripherals to a CPU.

PCI Express – An evolutionary version of PCI that maintains the PCI software usage model and replaces the physical bus with a high-speed (2.5 Gbps) serial bus serving multiple lanes.

Built-in self-test (BIST) – Is a mechanism that permits a machine to test itself. The main purpose of BIST is to reduce the complexity, and thereby decrease the cost and reduce reliance upon external (pattern-programmed) test equipment.

Design for Testability (DFT) – Stands for IC design techniques that add certain testability features to a hardware product design. The premise of the added features is that they make it easier to develop and apply manufacturing tests for the designed hardware. The purpose of manufacturing tests is to validate that the product hardware contains no manufacturing defects that could, otherwise, adversely affect the product's correct functioning.

Clock and Data Recovery (CDR) – Some digital data streams, especially high-speed serial data streams (such as the raw stream of data from the magnetic head of a disk drive) are sent without an accompanying clock signal. The receiver generates a clock from an approximate frequency reference, and then phase-aligns to the transitions in the data stream with a phase-locked loop (PLL). This process is commonly known as clock and data recovery (CDR).

Phase-locked loop (PLL) – Is a control system that generates an output signal whose phase is related to the phase of an input signal.

GUI – In computer science, a graphical user interface (GUI), is a type of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, instead of text-based user interfaces, typed command labels or text navigation. GUIs were introduced in reaction to the perceived steep learning curve of command-line interfaces (CLIs), which require commands to be typed on a computer keyboard.

ASIC – An ASIC (application-specific integrated circuit) is a microchip designed for a special application, such as a particular kind of transmission protocol or a hand-held computer. You might contrast it with general integrated circuits, such as the microprocessor and the random access memory chips in your PC. ASICs are used in a wide-range of applications, including auto emission control, environmental monitoring, and personal digital assistants (PDAs).

7.3. RTL Codes

7.3.1. SerDes.sv

```
/////////////////////////////////////////////////////////////////////
//
// Module: SerDes
//
//
//
// Description: This block is the top level wrapper of the
//
// SerDes system, it contains the digital and analog transmitter,
//
// the digital and analog receiver, the test modules and the
//
// clock divider.
//
/////////////////////////////////////////////////////////////////////
/

module  SerDes  #( parameter WIDTH = 9) (

//------------Inputs--------
  input                         rst, //reset
  input                         clk, //main system clock
  input                         rxa_in_p,//external Positive input
for Analog Receiver
  input                         rxa_in_n, //external Negative
```

```
input for Analog Receiver
  input       [ WIDTH - 1 : 0 ] txd_data_in, //external digital
transmission input
  input               [ 7 : 0 ] config_in,
  input                         test_en,

//------------Outputs--------
  output logic [ WIDTH - 1 : 0] digital_out,
  output logic                  txa_data_out_p,
  output logic                  txa_data_out_n,
  output logic                  txd_data_out,
  output logic                  test_out

  );

//------------Internal Variables--------
  logic rxa_out_p;
  logic rxa_out_n;
  logic [ 7 : 0 ] tx_config;
  logic [ 7 : 0 ] clock_div8_phases; //clock divided in 8 phases
  logic txa_data_in_i;
  logic [ WIDTH - 1 : 0] txd_data_in_i;
  logic rxd_in_i;
  logic lfsr_en;
  logic comma_detected;
  logic [ WIDTH - 1 : 0] rxd_data_out;
  logic c_data_valid;
  logic dispout;
  logic code_err;
  logic disp_err;

//------------Module instantiation --------

///////////////////////
//Transmitter modules//
///////////////////////

 //Digital Transmitter
digital_transmitter digital_transmitter0(
  .rst(rst),
  .txd_data_in(txd_data_in_i),
  .ser_clk(clock_div8_phases[0]),
  .lfsr_en(lfsr_en),
  .txd_data_out(txd_data_out)
);
```

```
 //Analog Transmitter
 analog_transmitter_wrap analog_transmitter_wrap0(
      .txa_data_in(txa_data_in_i),
      .tx_config(tx_config),
      .txa_data_out_p(txa_data_out_p),
      .txa_data_out_n(txa_data_out_n)
);

/////////////////////
// Clocking modules //
/////////////////////

// clocking section
clock_divider clock_divider0(
      .a_rst(rst),
      .ref_clk(clk),
      .clocks_out(clock_div8_phases)
);

/////////////////////
// Receiver modules //
/////////////////////

 //Analog receiver
analog_receiver analog_receiver0(
      .rxa_in_p(rxa_in_p),
      .rxa_in_n(rxa_in_n),
      .rxa_out_p(rxa_out_p),
      .rxa_out_n(rxa_out_n)
);

 //Digital receiver
digital_receiver digital_receiver0(
      .rst(rst),
      .clocks_in(clock_div8_phases),
      .rxd_in(rxd_in_i),
      .rxd_data_out(rxd_data_out),
      .c_data_valid(c_data_valid),
      .comma_detected(comma_detected),
      .dispout(dispout),
      .code_err(code_err),
      .disp_err(disp_err)
);

//////////////////
// TEST modules //
```

```
/////////////////

test_modules  #(9) test_modules0 (
      .rst(rst),
      .ser_clk(clock_div8_phases[0]),
      .lfsr_en(lfsr_en),
      .test_out(test_out),
      .rxd_data_out(rxd_data_out),
      .rxa_out(rxa_out_p),
      .txd_data_in(txd_data_in),
      .rxd_in_i(rxd_in_i),
      .txd_data_in_i(txd_data_in_i),
      .txd_data_out(txd_data_out),
      .txa_data_in_i(txa_data_in_i),
      .config_in(config_in),
      .test_en(test_en),
      .tx_config(tx_config),
      .digital_out(digital_out),
      .c_data_valid(c_data_valid),
      .dispout(dispout),
      .code_err(code_err),
      .disp_err(disp_err)
);

endmodule
```

7.3.2. analog_receiver.sv

```
//////////////////////////////////////////////////////////////////
/
// Module: analog_receiver
//
//
//
// Description: This block is a behavioural model for the analog
//
// front end of the Deserializer. The analog front end transforms
//
// a weak differential signal into a CMOS single ended signal.
//
//
//
//////////////////////////////////////////////////////////////////
/
```

```
module analog_receiver(

//------------Inputs--------
    input rxa_in_p,
    input rxa_in_n,

//------------Outputs--------
    output rxa_out_p,
      output rxa_out_n
    );


      assign rxa_out_n   = rxa_in_n;
    assign rxa_out_p   = rxa_in_p;

endmodule
```

7.3.3. analog_transmitter_wrap.sv

```
/////////////////////////////////////////////////////////////////
/
// Module: analog_transmitter_wrap
//
//
//
// Description: This block is a wrapper for the differential
//
// blocks of the analog transmitter. It contains both
//
// analog_transmitter_p and analog_transmitter_n
//
//
//
/////////////////////////////////////////////////////////////////
/


module analog_transmitter_wrap (

//------------Inputs--------
          input
txa_data_in,
        input            [ 7 : 0 ]  tx_config,

//------------Outputs--------
          output                                  txa_data_out_p,
```

```
          output                        txa_data_out_n
);


//------------Internal Variables--------

     logic                          txa_data_in_n;
     logic               [ 7 : 0 ]  tx_config_n;

//-------------Code Starts Here-------

     always_comb
   begin
     tx_config_n = ~tx_config;
     txa_data_in_n = ~txa_data_in;
   end


//------------- Module instantiation -------

         analog_transmitter analog_transmitter_P0(
        .Data(txa_data_in),
        .Eq_A(tx_config[0]),
        .Eq_B(tx_config[1]),
        .Imp_A(tx_config[2]),
        .Imp_B(tx_config[3]),
        .Imp_C(tx_config[4]),
        .Imp_D(tx_config[5]),
        .Amp_A(tx_config[6]),
        .Amp_B(tx_config[7]),
        .txa_data_out(txa_data_out_p)
     );

         analog_transmitter analog_transmitter_N0(
        .Data(txa_data_in_n),
        .Eq_A(tx_config_n[0]),
        .Eq_B(tx_config_n[1]),
        .Imp_A(tx_config_n[2]),
        .Imp_B(tx_config_n[3]),
        .Imp_C(tx_config_n[4]),
        .Imp_D(tx_config_n[5]),
        .Amp_A(tx_config_n[6]),
        .Amp_B(tx_config_n[7]),
        .txa_data_out(txa_data_out_n)
     );
```

```
endmodule
```

7.3.4. analog_transmitter.sv

```
//////////////////////////////////////////////////////////////////
//
// Module: analog_transmitter
//
//
//
// Description: This block is a behavioural model for the analog
//
// front end of the Serializer transforms a CMOS Single Ended
//
// signal into a differential signal with programmable amplitude,
//
// impedance and pre-emphasis.
//
//
//
// NOTE: Eq_A, Eq_b, Imp_A, Imp_B, Imp_C, Imp_D, Amp_A, Amp_B are
//
// not implemented in this module since they don't have a digital
//
// counterpart.
//
//////////////////////////////////////////////////////////////////
//

module analog_transmitter(

//------------Inputs--------
      input Data,
      input Eq_A,
      input Eq_B,
      input Imp_A,
      input Imp_B,
      input Imp_C,
      input Imp_D,
      input Amp_A,
      input Amp_B,

//------------Outputs--------
      output txa_data_out
      );
```

```
//-------------Code Starts Here-------
     assign txa_data_out = Data;

endmodule
```

7.3.5. clock_divider.sv

```
// Clock Divider
// This block takes a clock reference and divides it by 8.
// It will generate 8 clocks at equidistant phases.

module clock_divider(a_rst, ref_clk, clocks_out);
    input a_rst;
    input ref_clk;
    output reg [7:0] clocks_out;

    reg [3:0] div_by_8_q;
    wire ref_clk_div_by_8;

    // Use a chain of 4 flip flops to divide the clock by 8.
    // The flip flop chain with a not feedback follows this
    // sequence:
    always @(posedge ref_clk, posedge a_rst) begin
        if (a_rst)
            div_by_8_q <= 4'b0;
        else
            div_by_8_q <= { div_by_8_q[2:0], ~div_by_8_q[3]};
    end
    assign ref_clk_div_by_8 = div_by_8_q[3];

    // Delay the divided clock 7 times to generate the equidistant
phases
    // Use the divided clock as the first phase
    always @(posedge ref_clk, posedge a_rst) begin
        if (a_rst)
            clocks_out <= 8'd0;
        else
            clocks_out <= {clocks_out[6:0], ref_clk_div_by_8};
    end
endmodule
```

7.3.6. digital_receiver.sv

```
/////////////////////////////////////////////////////////////////
/
```

```
// Module: digital_receiver
//
//
//
// Description: This block is a wrapper for the digital modules
//
// that are involved in the reception of the data.
//
// It contains the deserializer and the decoder.
//
/////////////////////////////////////////////////////////////////
/

module digital_receiver (

//------------Inputs--------
                input rst,
                input [7:0] clocks_in,
                input rxd_in,

//------------Outputs--------
                output [8:0] rxd_data_out,
                output c_data_valid,
                output comma_detected,
                output dispout,
         output code_err,
         output disp_err
);

//------------Internal Variables--------

wire [9:0] encoded_rx_data; //RXD internal wire (encoded data)

//------------Module instantiation --------

    //Digital Receiver - Deserializer
    deserializer deserializer0(
        .a_rst(rst),
        .clocks_in(clocks_in),
        .a_rx(rxd_in),
        .c_parallel_out(encoded_rx_data),
        .clock_out(),
        .disparity_d(),
        .disparity_q(),
        .c_data_valid(c_data_valid),
        .comma_detected(comma_detected));
```

```
    //Digital Receiver - Decoder (8b/10b)
    decode decode0(
        .datain(encoded_rx_data),
        .dispin(1'b0),
        .dataout(rxd_data_out),
        .dispout(dispout),
        .code_err(code_err),
        .disp_err(disp_err)

             );


endmodule
```

7.3.7. decode.sv

```
// Chuck Benz, Hollis, NH   Copyright (c)2002
//
// The information and description contained herein is the
// property of Chuck Benz.
//
// Permission is granted for any reuse of this information
// and description as long as this copyright notice is
// preserved.  Modifications may be made as long as this
// notice is preserved.

// per Widmer and Franaszek

module decode (datain, dispin, dataout, dispout, code_err,
disp_err) ;
  input [9:0]   datain ;
  input         dispin ;
  output [8:0]  dataout ;
  output   dispout ;
  output   code_err ;
  output   disp_err ;

  wire ai = datain[0] ;
  wire bi = datain[1] ;
  wire ci = datain[2] ;
  wire di = datain[3] ;
  wire ei = datain[4] ;
  wire ii = datain[5] ;
```

```
  wire fi = datain[6] ;
  wire gi = datain[7] ;
  wire hi = datain[8] ;
  wire ji = datain[9] ;

  wire aeqb = (ai & bi) | (!ai & !bi) ;
  wire ceqd = (ci & di) | (!ci & !di) ;
  wire p22 = (ai & bi & !ci & !di) |
          (ci & di & !ai & !bi) |
          ( !aeqb & !ceqd) ;
  wire p13 = ( !aeqb & !ci & !di) |
          ( !ceqd & !ai & !bi) ;
  wire p31 = ( !aeqb & ci & di) |
          ( !ceqd & ai & bi) ;

  wire p40 = ai & bi & ci & di ;
  wire p04 = !ai & !bi & !ci & !di ;

  wire disp6a = p31 | (p22 & dispin) ; // pos disp if p22 and was
pos, or p31.
   wire disp6a2 = p31 & dispin ;  // disp is ++ after 4 bits
   wire disp6a0 = p13 & ! dispin ; // -- disp after 4 bits

  wire disp6b = (((ei & ii & ! disp6a0) | (disp6a & (ei | ii)) |
disp6a2 |
            (ei & ii & di)) & (ei | ii | di)) ;

  // The 5B/6B decoding special cases where ABCDE != abcde

  wire p22bceeqi = p22 & bi & ci & (ei == ii) ;
  wire p22bncneeqi = p22 & !bi & !ci & (ei == ii) ;
  wire p13in = p13 & !ii ;
  wire p31i = p31 & ii ;
  wire p13dei = p13 & di & ei & ii ;
  wire p22aceeqi = p22 & ai & ci & (ei == ii) ;
  wire p22ancneeqi = p22 & !ai & !ci & (ei == ii) ;
  wire p13en = p13 & !ei ;
  wire anbnenin = !ai & !bi & !ei & !ii ;
  wire abei = ai & bi & ei & ii ;
  wire cdei = ci & di & ei & ii ;
  wire cndnenin = !ci & !di & !ei & !ii ;

  // non-zero disparity cases:
  wire p22enin = p22 & !ei & !ii ;
  wire p22ei = p22 & ei & ii ;
  //wire p13in = p12 & !ii ;
```

```
  //wire p31i = p31 & ii ;
  wire p31dnenin = p31 & !di & !ei & !ii ;
  //wire p13dei = p13 & di & ei & ii ;
  wire p31e = p31 & ei ;

  wire compa = p22bncneeqi | p31i | p13dei | p22ancneeqi |
           p13en | abei | cndnenin ;
  wire compb = p22bceeqi | p31i | p13dei | p22aceeqi |
           p13en | abei | cndnenin ;
  wire compc = p22bceeqi | p31i | p13dei | p22ancneeqi |
           p13en | anbnenin | cndnenin ;
  wire compd = p22bncneeqi | p31i | p13dei | p22aceeqi |
           p13en | abei | cndnenin ;
  wire compe = p22bncneeqi | p13in | p13dei | p22ancneeqi |
           p13en | anbnenin | cndnenin ;

  wire ao = ai ^ compa ;
  wire bo = bi ^ compb ;
  wire co = ci ^ compc ;
  wire Do = di ^ compd ;
  wire eo = ei ^ compe ;

  wire feqg = (fi & gi) | (!fi & !gi) ;
  wire heqj = (hi & ji) | (!hi & !ji) ;
  wire fghj22 = (fi & gi & !hi & !ji) |
           (!fi & !gi & hi & ji) |
           ( !feqg & !heqj) ;
  wire fghjp13 = ( !feqg & !hi & !ji) |
            ( !heqj & !fi & !gi) ;
  wire fghjp31 = ( (!feqg) & hi & ji) |
            ( !heqj & fi & gi) ;

  wire dispout = (fghjp31 | (disp6b & fghj22) | (hi & ji)) & (hi |
ji) ;

  wire ko = ( (ci & di & ei & ii) | ( !ci & !di & !ei & !ii) |
           (p13 & !ei & ii & gi & hi & ji) |
           (p31 & ei & !ii & !gi & !hi & !ji)) ;

  wire alt7 =   (fi & !gi & !hi & // 1000 cases, where disp6b is 1
            ((dispin & ci & di & !ei & !ii) | ko |
             (dispin & !ci & di & !ei & !ii))) |
           (!fi & gi & hi & // 0111 cases, where disp6b is 0
            (( !dispin & !ci & !di & ei & ii) | ko |
             ( !dispin & ci & !di & ei & ii))) ;
```

```
  wire k28 = (ci & di & ei & ii) | ! (ci | di | ei | ii) ;
  // k28 with positive disp into fghi - .1, .2, .5, and .6 special
cases
  wire k28p = ! (ci | di | ei | ii) ;
  wire fo = (ji & !fi & (hi | !gi | k28p)) |
         (fi & !ji & (!hi | gi | !k28p)) |
         (k28p & gi & hi) |
         (!k28p & !gi & !hi) ;
  wire go = (ji & !fi & (hi | !gi | !k28p)) |
         (fi & !ji & (!hi | gi |k28p)) |
         (!k28p & gi & hi) |
         (k28p & !gi & !hi) ;
  wire ho = ((ji ^ hi) & ! ((!fi & gi & !hi & ji & !k28p) | (!fi &
gi & hi & !ji & k28p) |
                    (fi & !gi & !hi & ji & !k28p) | (fi & !gi & hi
& !ji & k28p))) |
         (!fi & gi & hi & ji) | (fi & !gi & !hi & !ji) ;

  wire disp6p = (p31 & (ei | ii)) | (p22 & ei & ii) ;
  wire disp6n = (p13 & ! (ei & ii)) | (p22 & !ei & !ii) ;
  wire disp4p = fghjp31 ;
  wire disp4n = fghjp13 ;

  assign code_err = p40 | p04 | (fi & gi & hi & ji) | (!fi & !gi &
!hi & !ji) |
              (p13 & !ei & !ii) | (p31 & ei & ii) |
              (ei & ii & fi & gi & hi) | (!ei & !ii & !fi & !gi &
!hi) |
              (ei & !ii & gi & hi & ji) | (!ei & ii & !gi & !hi &
!ji) |
              (!p31 & ei & !ii & !gi & !hi & !ji) |
              (!p13 & !ei & ii & gi & hi & ji) |
              (((ei & ii & !gi & !hi & !ji) |
                (!ei & !ii & gi & hi & ji)) &
               ! ((ci & di & ei) | (!ci & !di & !ei))) |
              (disp6p & disp4p) | (disp6n & disp4n) |
              (ai & bi & ci & !ei & !ii & ((!fi & !gi) | fghjp13))
|
              (!ai & !bi & !ci & ei & ii & ((fi & gi) | fghjp31))
|
              (fi & gi & !hi & !ji & disp6p) |
              (!fi & !gi & hi & ji & disp6n) |
              (ci & di & ei & ii & !fi & !gi & !hi) |
              (!ci & !di & !ei & !ii & fi & gi & hi) ;

  assign dataout = {ko, ho, go, fo, eo, Do, co, bo, ao} ;
```

```
  // my disp err fires for any legal codes that violate disparity,
may fire for illegal codes
   assign disp_err = ((dispin & disp6p) | (disp6n & !dispin) |
                      (dispin & !disp6n & fi & gi) |
                      (dispin & ai & bi & ci) |
                      (dispin & !disp6n & disp4p) |
                      (!dispin & !disp6p & !fi & !gi) |
                      (!dispin & !ai & !bi & !ci) |
                      (!dispin & !disp6p & disp4n) |
                      (disp6p & disp4p) | (disp6n & disp4n)) ;

endmodule
```

7.3.8. deserializer.sv

```
// Deserializer
// This block samples an asynchronous signal. and transforms it
into
// a source synchronous parallel bus.
// It uses up-sampling data recovery and a shift register to
transform the
// serial input stream into a parallel bus.
//
// This block does not decode or encode any of the inputs.

module deserializer(a_rst, clocks_in, a_rx, c_parallel_out,
clock_out, disparity_d, disparity_q, c_data_valid,
comma_detected);
    input a_rst;
    input [7:0] clocks_in;
    input a_rx;
    input disparity_d ;
    output reg [9:0] c_parallel_out;
    output reg clock_out;
    output reg  disparity_q;
    output reg c_data_valid;
    output comma_detected;

    reg [7:0] c_rx_upsampled;
    reg [3:0] num_of_ones_in_rx;
    reg c_rx;
    wire clock;
    reg [9:0] shift_reg;
    reg [3:0] cycle_count;
```

```
    integer i;

    // Use the phase 0 clock as the system clock
    assign clock = clocks_in[0];


    // Use a flip flop to remember the running disparity in the
decoder
     always @(posedge clock, posedge a_rst) begin
        if (a_rst)
            disparity_q <= 1'b0;
        else
            disparity_q <= disparity_d;
    end

    // CDR
    // Up-sample de serial input with clocks_in at different
    // phases.
    genvar index;
    generate
        for (index=0; index < 8; index=index+1) begin :
gen_upsample
            always @(posedge clocks_in[index], posedge a_rst)
begin
                if (a_rst)
                     c_rx_upsampled[index] <= 1'b0;
                else
                     c_rx_upsampled[index] <= a_rx;

            end
        end
    endgenerate

    // Count the number of "1"s in the upsamples array
    always @(c_rx_upsampled) begin
        num_of_ones_in_rx = 4'h00;
        for(i=0; i < 8; i=i+1) begin
            num_of_ones_in_rx = num_of_ones_in_rx +
c_rx_upsampled[i];
        end
    end

    // Consider a_rx a 1, if a_rx stayed asserted on
    // most of the sampling phases.
    always @(posedge clock, posedge a_rst)
    begin
```

```
        if (a_rst)
            c_rx <= 1'b0;
        else
            c_rx <= (num_of_ones_in_rx > 4'd4)? 1'b1 : 1'b0;
    end

    // sipo 10-bit buffer
    // 10 bit shift register
    always@(posedge clock, posedge a_rst) begin
        if (a_rst)
            shift_reg <= 10'h000;
        else
            shift_reg <= {c_rx, shift_reg[9:1]};
    end

    // Look for comma symbol
    assign comma_detected = (shift_reg == 10'b0001111100) ||
(shift_reg == 10'b1110000011);

    // Use a 10 cycle counter to generate a data_valid signal.
    // Reset the counter if a special sync character, such as
    // a comma is identified
    always @(posedge clock, posedge a_rst) begin
        if (a_rst) begin
            cycle_count <= 4'd9;
            c_data_valid <= 1'b0;
            clock_out <= 1'b0;
        end
        else begin
            // 10 cycle counter
            if(comma_detected || (cycle_count == 4'd0))
                // Restart
                cycle_count <= 4'd9;
            else
                //Count down
                cycle_count <= cycle_count - 4'b0001;

            // Data is valid when the count down expires.
            if(comma_detected)
                c_data_valid <= 1'b0;
            else if (cycle_count == 4'd1)
                c_data_valid <= 1'b1;
            else
                c_data_valid <= 1'b0;

            // Assert clock_out one cycle after data_valid
```

```
            // De-assert clock_out in cycle 5
            if(cycle_count == 4'd5)
                clock_out <= 1'b0;
            else if (cycle_count == 4'd0 &&(!comma_detected))
                clock_out <= 1'b1;
        end
    end

    // reg outputs
    always @(posedge clock, posedge a_rst) begin
        if(a_rst)
            c_parallel_out <= 10'd0;
        else
            if (c_data_valid)
                c_parallel_out <= shift_reg;
    end

endmodule
```

7.3.9. digital_transmitter.sv

```
//////////////////////////////////////////////////////////////////
/
// Module: digital_transmitter
//
//
//
// Description:This block is a wrapper for the digital modules
//
// that are involved in the transmission of the data.
//
// It contains the encoder an the serializer.
//
//////////////////////////////////////////////////////////////////
/

module digital_transmitter (

//------------Inputs--------
          input rst,
          input [8:0] txd_data_in,
          input ser_clk,

//------------Outputs--------
          output lfsr_en,
          output txd_data_out
```

```
);

//------------Internal Variables--------

wire tx_disparityd;
wire tx_disparityq;
wire [9:0] encoded_tx_data;

//------------Module instantiation --------

    //Digital Transmitter - Encoder (8b/10b)
    encode encode0(
        .datain(txd_data_in),
        .dispin(tx_disparityq),
        .dataout(encoded_tx_data),
        .dispout(tx_disparityd)
    );
    //Digital Transmitter - Serializer
    serializer serializer0(
        .par_in(encoded_tx_data),
        .clk(ser_clk),
        .rst(rst),
        .disparity_d(tx_disparityd),
        .ser_out(txd_data_out),
        .disparity_q(tx_disparityq),
        .tx_frame_started(lfsr_en)
    );


endmodule
```

7.3.10. encode.sv

```
// Chuck Benz, Hollis, NH   Copyright (c)2002
//
// The information and description contained herein is the
// property of Chuck Benz.
//
// Permission is granted for any reuse of this information
// and description as long as this copyright notice is
// preserved.  Modifications may be made as long as this
// notice is preserved.
```

```
// per Widmer and Franaszek

module encode (datain, dispin, dataout, dispout) ;
  input [8:0]   datain ;
  input    dispin ;  // 0 = neg disp; 1 = pos disp
  output [9:0]  dataout ;
  output   dispout ;




  wire ai = datain[0] ;
  wire bi = datain[1] ;
  wire ci = datain[2] ;
  wire di = datain[3] ;
  wire ei = datain[4] ;
  wire fi = datain[5] ;
  wire gi = datain[6] ;
  wire hi = datain[7] ;
  wire ki = datain[8] ;

  wire aeqb = (ai & bi) | (!ai & !bi) ;
  wire ceqd = (ci & di) | (!ci & !di) ;
  wire l22 = (ai & bi & !ci & !di) |
          (ci & di & !ai & !bi) |
          ( !aeqb & !ceqd) ;
  wire l40 = ai & bi & ci & di ;
  wire l04 = !ai & !bi & !ci & !di ;
  wire l13 = ( !aeqb & !ci & !di) |
          ( !ceqd & !ai & !bi) ;
  wire l31 = ( !aeqb & ci & di) |
          ( !ceqd & ai & bi) ;

  // The 5B/6B encoding

  wire ao = ai ;
  wire bo = (bi & !l40) | l04 ;
  wire co = l04 | ci | (ei & di & !ci & !bi & !ai) ;
  wire Do= di & ! (ai & bi & ci) ;
  wire eo = (ei | l13) & ! (ei & di & !ci & !bi & !ai) ;
  wire io = (l22 & !ei) |
         (ei & !di & !ci & !(ai&bi)) |  // D16, D17, D18
         (ei & l40) |
         (ki & ei & di & ci & !bi & !ai) | // K.28
         (ei & !di & ci & !bi & !ai) ;
```

```
  // pds16 indicates cases where d-1 is assumed + to get our
encoded value
  wire pd1s6 = (ei & di & !ci & !bi & !ai) | (!ei & !l22 & !l31) ;
  // nds16 indicates cases where d-1 is assumed - to get our
encoded value
  wire nd1s6 = ki | (ei & !l22 & !l13) | (!ei & !di & ci & bi &
ai) ;

  // ndos6 is pds16 cases where d-1 is + yields - disp out - all
of them
  wire ndos6 = pd1s6 ;
  // pdos6 is nds16 cases where d-1 is - yields + disp out - all
but one
  wire pdos6 = ki | (ei & !l22 & !l13) ;


  // some Dx.7 and all Kx.7 cases result in run length of 5 case
unless
  // an alternate coding is used (referred to as Dx.A7, normal is
Dx.P7)
  // specifically, D11, D13, D14, D17, D18, D19.
  wire alt7 = fi & gi & hi & (ki |
                    (dispin ? (!ei & di & l31) : (ei & !di &
l13))) ;


  wire fo = fi & ! alt7 ;
  wire go = gi | (!fi & !gi & !hi) ;
  wire ho = hi ;
  wire jo = (!hi & (gi ^ fi)) | alt7 ;



  // nd1s4 is cases where d-1 is assumed - to get our encoded
value
  wire nd1s4 = fi & gi ;
  // pd1s4 is cases where d-1 is assumed + to get our encoded
value
  wire pd1s4 = (!fi & !gi) | (ki & ((fi & !gi) | (!fi & gi))) ;

  // ndos4 is pd1s4 cases where d-1 is + yields - disp out - just
some
  wire ndos4 = (!fi & !gi) ;
```

```
  // pdos4 is nd1s4 cases where d-1 is - yields + disp out
  wire pdos4 = fi & gi & hi ;

  // only legal K codes are K28.0->.7, K23/27/29/30.7
  //  K28.0->7 is ei=di=ci=1,bi=ai=0
  //  K23 is 10111
  //  K27 is 11011
  //  K29 is 11101
  //  K30 is 11110 - so K23/27/29/30 are ei & l31
  wire illegalk = ki &
           (ai | bi | !ci | !di | !ei) & // not K28.0->7
           (!fi | !gi | !hi | !ei | !l31) ; // not K23/27/29/30.7

  // now determine whether to do the complementing
  // complement if prev disp is - and pd1s6 is set, or + and nd1s6
is set
  wire compls6 = (pd1s6 & !dispin) | (nd1s6 & dispin) ;

  // disparity out of 5b6b is disp in with pdso6 and ndso6
  // pds16 indicates cases where d-1 is assumed + to get our
encoded value
  // ndos6 is cases where d-1 is + yields - disp out
  // nds16 indicates cases where d-1 is assumed - to get our
encoded value
  // pdos6 is cases where d-1 is - yields + disp out
  // disp toggles in all ndis16 cases, and all but that 1 nds16
case

  wire disp6 = dispin ^ (ndos6 | pdos6) ;

  wire compls4 = (pd1s4 & !disp6) | (nd1s4 & disp6) ;
  assign dispout = disp6 ^ (ndos4 | pdos4) ;

  assign dataout = {(jo ^ compls4), (ho ^ compls4),
             (go ^ compls4), (fo ^ compls4),
             (io ^ compls6), (eo ^ compls6),
             (Do^ compls6), (co ^ compls6),
             (bo ^ compls6), (ao ^ compls6)} ;


endmodule
```

7.3.11. serializer.sv

```
module serializer(par_in, clk, rst, disparity_d, ser_out,
disparity_q, tx_frame_started);

    input[9:0]  par_in;   // data from the encoder
    input       clk;     // shift clock
    input       rst;     // reset signal
    input       disparity_d ;

    output      ser_out;  // serial output data
    output reg  disparity_q;
// -------------Internal Registers-----------------------//
   reg [9:0]   par_reg;  // register to store data
   reg [3:0]   counter;
   output reg         tx_frame_started;

// -------------Control Path-----------------------------//



  always @(posedge clk, posedge rst ) begin
        if (rst) begin
              counter <= 4'b0000;
              tx_frame_started <= 1'b0;
        end

        //tx_frame_started should be 1 after counter is zero
        else if (counter == 4'b0)begin
                 tx_frame_started <= 1'b1;
                 counter <= counter + 1'b1;
        end

        //Count from 0 to 9
        else if (counter < 4'b1001) begin
                 tx_frame_started <= 1'b0;
                 counter <= counter + 1'b1;
        end

        // Restart counter
        else
        begin
                 counter <= 4'b0000;
                   tx_frame_started <= 1'b0;
          end
  end

// -------------Data Path-----------------------------//
```

```
always @(posedge clk, posedge rst) begin
    if (rst)
          par_reg <= 10'h00;

    else
       if (tx_frame_started)
           par_reg <= par_in;
       else
           par_reg <= {1'b0, par_reg[9:1]};
end

assign ser_out = par_reg[0];


always @(posedge clk, posedge rst) begin
                if (rst)
                     disparity_q <= 1'b0;

                else if (tx_frame_started)
                     disparity_q <= disparity_d;
     end

endmodule
```

7.3.12. test_modules.sv

```
//////////////////////////////////////////////////////////////////
// Module: test_modules                                         //
//                                                              //
// Author: Cesar Limones 2016                                   //
//                                                              //
// Description: This block contains all the testing modules     //
// for a SerDes. It contains the LFSR, a comparator             //
// and a signal driver.                                         //
//////////////////////////////////////////////////////////////////

module test_modules #( parameter WIDTH = 9 )(

//------------Inputs--------
  input                               rst,
  input                               ser_clk,
  input           [ WIDTH - 1 : 0 ]   rxd_data_out,
  input                               lfsr_en,
  input                               test_en,
  input                               rxa_out,
```

```
  input                              txd_data_out,
  input                  [ 7 : 0 ]   config_in,
  input          [ WIDTH - 1 : 0 ]   txd_data_in,
  input
c_data_valid,
  input                              dispout,
  input                              code_err,
  input                              disp_err,

//------------Outputs--------
  output logic                       rxd_in_i,
  output logic     [ WIDTH - 1 : 0 ] txd_data_in_i,
  output logic             [ 7 : 0 ] tx_config,
  output logic     [ WIDTH - 1 : 0 ] digital_out,
  output logic                       txa_data_in_i,
  output logic                       test_out

);

//------------Internal Variables--------

  logic            [ WIDTH - 1 : 0 ] lfsr_parallel_out;
  logic                    [ 2 : 0 ] mode;
  logic                              test_in;
  logic                              bist_end;
  logic                              errors_en;
  logic                    [ 5 : 0 ] num_errors;

//-------------Code Starts Here-------
always_comb
     begin
          if (test_en)
          begin
               mode = config_in[2:0];
               test_in = config_in[3];
               errors_en = config_in[4];
               tx_config = 7'b0;
          end
          else
          begin
               mode = 3'b000;
               tx_config = config_in;
               test_in = 1'b0;
               errors_en = 1'b0;
          end
```

```
    end

//------------- Module instantiation -------

//Signal driver
signal_driver #(9) signal_driver0(
    .mode(mode),
    .test_in(test_in),
    .rxa_out(rxa_out),
    .txd_data_in(txd_data_in),
    .lfsr_parallel_out(lfsr_parallel_out),
    .rxd_in_i(rxd_in_i),
    .c_data_valid(c_data_valid),
    .txd_data_in_i(txd_data_in_i),
    .rxd_data_out(rxd_data_out),
    .txd_data_out(txd_data_out),
    .bist_end(bist_end),
    .test_out(test_out),
    .txa_data_in_i(txa_data_in_i),
    .digital_out(digital_out),
    .errors_en(errors_en),
    .num_errors(num_errors),
    .dispout(dispout),
    .code_err(code_err),
    .disp_err(disp_err)
);


//Pseudo Alleatory pattern Generator LFSR
lfsr lfsr0(
    .clk(ser_clk),
    .rst(rst),
    .lfsr_parallel_out(lfsr_parallel_out),
    .lfsr_en(lfsr_en)
);

//Comparator
comparator #(9) comparator0(
    .dataB_in(rxd_data_out),
    .clk(ser_clk),
    .rst(rst),
    .dataA_in(txd_data_in_i),
    .lfsr_en(lfsr_en),
    .bist_end(bist_end),
    .num_errors(num_errors)
);
```

```
endmodule
```

7.3.13. comparator.sv

```
///////////////////////////////////////////////////////////////
// Module: comparator                                        //
//                                                           //
// Author: Cesar Limones 2016                                //
//                                                           //
// Description: This module will be contantly comparing the //
// transmitted parallel data against the final data in the  //
// receiver output and keep count of the number of errors   //
// and matches.                                              //
///////////////////////////////////////////////////////////////

module comparator #( parameter WIDTH = 9 )(

//------------Inputs--------

     input  [ WIDTH -1 : 0 ] dataA_in,
     input  [ WIDTH -1 : 0 ] dataB_in,
     input                        clk,
     input                        rst,
     input                    lfsr_en,

//------------Outputs--------
     output logic            bist_end,
     output logic [ 5 : 0 ] num_errors

);

//------------Parameters--------
parameter MEM_SIZE = 8;


//------------Internal Variables--------
 reg [2:0] state, next;


reg [5:0] total_errors;
reg [5:0] num_matches;
reg [5:0] total_matches;
reg [8:0] RegMem [ 0 : MEM_SIZE - 1]= '{ default: '0 };
```

```
reg [5:0]cntA;
reg [5:0]cntB;
reg matchFlag;

//-------------Code Starts Here-------

parameter IDLE = 3'b000,
          READY = 3'b001,
          COMP = 3'b010,
                 WAIT =    3'b011,
          ENDING = 3'b100;


always @(posedge clk, posedge rst)
     if (rst) state <= IDLE;
       else state <= next;

always_comb begin
 next = 'bx;
      case (state)
           //Bist sequence will not restart until reset.
                IDLE : if (!bist_end) next = READY;
                           else next = IDLE;

                //Comparator will start on sending the first comma
                READY : begin
                      if (lfsr_en & dataA_in == 9'b111111100)
                            next = WAIT;
                           else next = READY;
                           end
                COMP : begin
                      if (bist_end) next = IDLE;
                           else next = WAIT;
                           end
                WAIT:
                           if (lfsr_en & dataA_in != 9'b111111100)
next = COMP;
                           else if (lfsr_en & dataA_in ==
9'b111111100 ) next = ENDING;
                           else next = WAIT;
                ENDING : begin
                           next = WAIT;
                           end
      endcase
end
```

```
 always @(posedge clk, posedge rst)
      if (rst) begin
                     RegMem       <= '{ default: '0 };
                     cntA         <= 6'd0;
                     cntB         <= 6'd0;
                     matchFlag    <= 1'b0;
                     num_errors   <= 6'b0;
                     num_matches  <= 6'b0;
                     bist_end     <= 1'b0;
                     total_errors <= 6'b0;
                     total_matches<= 6'b0;
       end
      else begin
      case (next)

                //Entering comparison State
         COMP :
                begin
                      //Register the Data when there is no comma
on dataA_in
                           RegMem[cntA] <= dataA_in;
                           cntA <= cntA + 6'b000001;
                           //Increasing Counter A
                           if (cntA < MEM_SIZE - 1)
                           begin
                                cntA <= cntA + 6'b000001;
                           end
                           else
                           begin
                                cntA <= 6'b000000;
                           end
                                if (matchFlag)
                                     begin
                                          // Detecting a match
                                          if (RegMem[cntB] ==
dataB_in )
                                               begin
                                                    num_matches<=
num_matches + 6'b000001;
                                               end
                                          //detecting the comma
                                          else if (dataB_in ===
9'h1fc)
                                               begin
                                                    $display
("Comma found: RegMem[%d]=%h dataB_in%d= %h at time %t",cntB,
```

```
RegMem[cntB], cntA, dataB_in,  $time);
                                                      bist_end <=
1'b1;
                                                      matchFlag
<= 1'b0;
                                                      total_errors
<= num_errors;
                                                      total_matches
<= num_matches;
                                                      $display
("Ending bist: Total_errors=%d Total_matches=%d at time
%t",num_errors, num_matches,  $time);
                                                      $stop();
                                                 end
                                            else
                                                 begin
                                                      matchFlag <=
1'b0;
                                                      $display
("Error Not maching: RegMem[%d]=%h dataB_in%d= %h at time
%t",cntB, RegMem[cntB], cntA, dataB_in,  $time);
                                                      num_errors <=
num_errors + 6'b000001;
                                                 end

                                            //Increasing B counter

                                            if (cntB < MEM_SIZE - 1)
                                                 begin
                                                   cntB <= cntB +
6'b000001;
                                                 end
                                            else
                                                 begin
                                                   cntB <=
6'b000000;
                                                  end
                                  end
                                  else //!matchflag
                                       begin
                                             //Detecting the first
match of the registered data and the compared data
                                             if (RegMem[cntB] ==
dataB_in)
                                                   begin
```

```
     $display ("First Matching RegMem[%d]=%h dataB_in%d = %h at
time %t",cntB, RegMem[cntB], cntA, dataB_in,  $time);

num_matches<= num_matches + 6'b000001;
                                                        cntB <=
cntB + 6'b000001;
                                                        matchFlag
<= 1'b1;
                                                   end
                                          else if (dataB_in !==
9'h13f) //Received data isn't a decoded comma // Fixed to flag
errors on X
                                                 begin
                                                      $display
("Error Not maching: RegMem[%d]=%h dataB_in%d= %h at time
%t",cntB, RegMem[cntB], cntA, dataB_in,  $time);
                                                      num_errors <=
num_errors + 6'b000001;
                                                 end
                                         end
         end

                // Ignore the second comma on the LFSR_IN by
increasing the counters
                // This happens when the LFSR pattern is
restarting
                ENDING : begin
                                cntA <= cntA + 6'b000001;
                                cntB <= cntB + 6'b000001;
                                end
      endcase
 end

endmodule
```

7.3.14. lsfr.sv

```
//////////////////////////////////////////////////////////////
// Module: lsfr                                             //
//                                                          //
// Author: Cesar Limones 2016                               //
//                                                          //
// Description: This module is a linear feed back shift     //
// register that will generate parallel data to be sent by  //
// the transmitter when BIST mode is enabled. It has the    //
// initial value fixed to a comma symbol.                   //
```

```
//////////////////////////////////////////////////////////////

module lfsr (

//------------Inputs--------
input rst,
input clk,
input lfsr_en,

//------------Outputs--------

output logic [8:0] lfsr_parallel_out
);

//------------Parameters--------
//selected Seed is a comma
// it's used to start the transmission automatically after reset
if the correct mode is selected
parameter seed = 10'b1111111100;

//------------Internal Variables--------
wire        linear_feedback;
wire        linear_feedback_i;
reg  [9:0] lfsr_parallel_out_reg = seed; //initialize the register
to the comma

//-------------Code Starts Here-------
assign linear_feedback = lfsr_parallel_out_reg[4] ^
linear_feedback_i;
assign linear_feedback_i = lfsr_parallel_out_reg[5] ^
lfsr_parallel_out_reg[9];


always_ff @(posedge clk, posedge rst)
if (rst) begin // active high reset
  lfsr_parallel_out_reg <= seed ;
end else if (lfsr_en) begin

  lfsr_parallel_out_reg <= {lfsr_parallel_out_reg[8],
lfsr_parallel_out_reg[7],
          lfsr_parallel_out_reg[6],lfsr_parallel_out_reg[5],
          lfsr_parallel_out_reg[4],lfsr_parallel_out_reg[3],
          lfsr_parallel_out_reg[2],lfsr_parallel_out_reg[1],
          lfsr_parallel_out_reg[0], linear_feedback};
end
```

```
always_comb
begin
    if (lfsr_parallel_out_reg[9:0] == seed) //Sending all the 9
bits when there is a comma
       lfsr_parallel_out =  lfsr_parallel_out_reg[8:0];
    else //If there is no comma on the data send a zero on the k
bit.
       lfsr_parallel_out = {1'b0, lfsr_parallel_out_reg[7:0]};
end

endmodule // End Of Module LFSR
```

7.3.15. signal_driver.sv

```
//////////////////////////////////////////////////////////////
// Module: signal_driver                                    //
//                                                          //
// Author: Cesar Limones 2016                               //
//                                                          //
// Description: This module functionality is to gather all  //
// the signals from the other modules and drive them to     //
// their respective counterpart depending on the test mode  //
// that is being executed.                                  //
//////////////////////////////////////////////////////////////

module signal_driver # ( parameter WIDTH = 9)(

//------------Inputs--------
input                 [ 2 : 0 ]   mode,
input                             rxa_out,
input           [ WIDTH - 1 : 0 ] txd_data_in,
input           [ WIDTH - 1 : 0 ] lfsr_parallel_out,
input           [ WIDTH - 1 : 0 ] rxd_data_out,
input                             txd_data_out,
input                             bist_end,
input                             test_in,
input                             errors_en,
input                       [ 5 : 0 ] num_errors,
input                                                c_data_valid,
input                             dispout,
input                             code_err,
input                             disp_err,

//------------Outputs--------
output logic                      test_out,
```

```
output logic                        rxd_in_i,
output logic      [ WIDTH - 1 :0] txd_data_in_i,
output logic                        txa_data_in_i,
output logic      [ WIDTH - 1 :0] digital_out

);

//-------------Code Starts Here-------
always_comb
   if(errors_en)
     begin
       digital_out[5:0] = num_errors[5:0];
       digital_out[6] = dispout;
       digital_out[7] = code_err;
       digital_out[8] = disp_err;
      end
    else
       digital_out = rxd_data_out;

always_comb begin
    case (mode)
    3'h0: //Functional Mode
             begin
                     rxd_in_i = rxa_out;
                     txd_data_in_i = txd_data_in;
                     test_out = c_data_valid;
                     txa_data_in_i = txd_data_out;
             end //end of mode0

    3'h1://Parallel loopback
        begin
            rxd_in_i = rxa_out;
            txd_data_in_i = rxd_data_out;
           test_out = c_data_valid;
                txa_data_in_i = txd_data_out;
        end// end of mode1: Parallel Loopback

    3'h2: // Serial Loopback
        begin
            rxd_in_i = txd_data_out;
            txd_data_in_i = txd_data_in;
           test_out = c_data_valid;
                txa_data_in_i = txd_data_out;
        end// end of mode2: Serial loopback

    3'h3: // RXA bypass
```

```
        begin
            rxd_in_i = test_in;
            txd_data_in_i = txd_data_in;
           test_out = c_data_valid;
                txa_data_in_i = txd_data_out;
        end// end of mode3: RXA bypass

      4'h4: // BIST With Serial Loopback
        begin
            rxd_in_i = txd_data_out;
            txd_data_in_i = lfsr_parallel_out[ WIDTH - 1 : 0 ];
           test_out = bist_end;
                txa_data_in_i = txd_data_out;
        end// end of mode4: BIST With Serial Loopback

     4'h5: //RXA bypass with parallel loopback
          begin
            rxd_in_i = test_in;
            txd_data_in_i = rxd_data_out;
           test_out = c_data_valid;
                txa_data_in_i = txd_data_out;
        end // end of mode 5: RXA bypass with parallel loopback

   4'h6://Open BIST
        begin
            rxd_in_i = rxa_out;
            txd_data_in_i = lfsr_parallel_out [ WIDTH - 1 : 0 ];
              test_out = bist_end;
                txa_data_in_i = txd_data_out;
          end // end of mode 6 : Open BIST

     4'h7://Analog Receiver Output and Analog Loopback
          begin
            txd_data_in_i = txd_data_in;
              rxd_in_i = rxa_out;
                test_out = rxa_out;
                txa_data_in_i = rxa_out;
          end // end of mode 7: Analog Receiver Output and Analog
Loopback

    default://Default assignments
        begin
            rxd_in_i      = rxa_out;
            txd_data_in_i = txd_data_in;
              test_out      = bist_end;
              txa_data_in_i = txd_data_out;
```

```
        end

    endcase
end


endmodule //Signal_driver
```

7.1. Tesbench Codes

7.1.1. SerDes_m0_tb.sv

```
`timescale 1ns/100ps
`define DELAY 10

module SerDes_m0_tb();

localparam DATA_WIDTH = 8;

logic                             rst;
logic                             clk;
logic                     rxa_in_p;
logic                     rxa_in_n;
logic [DATA_WIDTH:0]          txd_data_in;
logic [DATA_WIDTH:0]          txd_data_in_i;
logic [DATA_WIDTH:0]          digital_out;
logic                       txa_data_out_p;
logic                       txa_data_out_n;
logic                          test_out;
logic [5:0]                num_errors;
logic txd_data_out;
logic                 [2:0]       mode;
logic                       txd_out;
logic        [ 7 : 0 ] config_in;
logic                                 test_en;
reg tx_clk;
integer i;


//Module instantiation.
SerDes  #(.WIDTH(9)) serdes0 (
      .rst(rst),
      .clk(clk),
      .rxa_in_p(rxa_in_p),
      .rxa_in_n(rxa_in_n),
      .txd_data_in(txd_data_in),
```

```
      .config_in(config_in),
      .test_en(test_en),
      .digital_out(digital_out),
      .test_out(test_out),
      .txa_data_out_p(txa_data_out_p),
      .txa_data_out_n(txa_data_out_n),
      .txd_data_out(txd_data_out)
);

// Clock generation
 initial begin
         clk <= 1'b0;
      forever
      begin
      #10   clk <= ~clk;
      end
 end

assign tx_clk = serdes0.clock_div8_phases[0];
assign txd_out =  serdes0.txd_data_out;
assign mode = serdes0.test_modules0.mode;
assign num_errors = serdes0.test_modules0.num_errors;
assign txd_data_in_i = serdes0.test_modules0.txd_data_in_i;


task parallel(input[DATA_WIDTH:0] data);
      begin
           @(posedge tx_clk);
                 txd_data_in = data;
                 repeat (DATA_WIDTH + 1 )
           @(posedge tx_clk);
      end
endtask

// input generation
initial
begin
    // reset
    #`DELAY  rst = 1'b1;
    #(`DELAY*10) rst = 1'b0;
      //Mode 0
       config_in=8'b0000_0000;
       test_en = 1'b0;
      //Send Comma
    @(posedge clk);
    // tx_data input generation
```

```
        parallel(9'b111111100); //Sending the first comma to start
        for (i = 0; i < 62; i = i + 1)
            parallel(i[ DATA_WIDTH :0]); //Send messages from 0 to
61
        parallel(9'b111111100); //Sending another comma to end
        for (i = 0; i < 62; i = i + 1)
            parallel(i[ DATA_WIDTH :0]); //Send messages from 0 to
61
    #(`DELAY*1000);
    $stop();
end

assign rxa_in_n = txa_data_out_n;
assign rxa_in_p = txa_data_out_p;

endmodule
```

7.1.2. SerDes_m0B_tb.sv

```
`timescale 1ns/100ps
`define DELAY 10

module SerDes_m05_tb();

localparam DATA_WIDTH = 8;

logic                             rst;
logic                             clk;
logic                     rxa_in_p;
logic                     rxa_in_n;
logic [DATA_WIDTH:0]          txd_data_in;
logic [DATA_WIDTH:0]          txd_data_in_i;
logic [DATA_WIDTH:0]          digital_out;
logic                       txa_data_out_p;
logic                       txa_data_out_n;
logic                          test_out;
logic [5:0]                num_errors;
logic txd_data_out;
logic                 [2:0]       mode;//Bist Configuration pins
logic                       txd_out;
logic         [ 7 : 0 ] config_in;
logic                                 test_en;
reg tx_clk;
integer i;
```

```
//Module instantiation.

SerDes  #(.WIDTH(9)) serdes0 (
      .rst(rst),
      .clk(clk),
      .rxa_in_p(rxa_in_p),
      .rxa_in_n(rxa_in_n),
      .txd_data_in(txd_data_in),
      .config_in(config_in),
      .test_en(test_en),
      .digital_out(digital_out),
      .test_out(test_out),
      .txa_data_out_p(txa_data_out_p),
      .txa_data_out_n(txa_data_out_n),
      .txd_data_out(txd_data_out)
);

// Clock generation
 initial begin
         clk <= 1'b0;
      forever
      begin
      #10   clk <= ~clk;
      end
 end

assign tx_clk = serdes0.clock_div8_phases[0];
assign txd_out =  serdes0.txd_data_out;
assign mode = serdes0.test_modules0.mode;
assign num_errors = serdes0.test_modules0.num_errors;
assign txd_data_in_i = serdes0.test_modules0.txd_data_in_i;


task parallel(input[DATA_WIDTH:0] data);
      begin
            @(posedge tx_clk);
                  txd_data_in = data;
                  repeat (DATA_WIDTH + 1 )
            @(posedge tx_clk);
      end
endtask


// input generation
initial
begin
```

```
    // reset
    #`DELAY  rst = 1'b1;
    #(`DELAY*10) rst = 1'b0;
      //Mode 1 - Functional mode - Test_en
       config_in=8'b0011_001;
       test_en = 1'b1;
      //Send Comma
    @(posedge clk);
    // tx_data input generation
        parallel(9'b111111100); //Sending the first comma to start
        for (i = 0; i < 62; i = i + 1)
            parallel(i[ DATA_WIDTH :0]); //Send messages from 0 to
61
        parallel(9'b111111100); //Sending another comma to end
        for (i = 0; i < 62; i = i + 1)
            parallel(i[ DATA_WIDTH :0]); //Send messages from 0 to
61
    #(`DELAY*1000);
    $stop();
       //END MODE 1
end

assign rxa_in_n = txa_data_out_n;
assign rxa_in_p = txa_data_out_p;


endmodule
```

7.1.3. SerDes_m1_tb.sv

```
`timescale 1ns/100ps
`define DELAY 10

module SerDes_m1_tb();

localparam DATA_WIDTH = 8;

logic                             rst;
logic                             clk;
logic                     rxa_in_p;
logic                     rxa_in_n;
logic [DATA_WIDTH:0]          txd_data_in;
logic [DATA_WIDTH:0]          txd_data_in_i;
logic [DATA_WIDTH:0]          digital_out;
logic                       txa_data_out_p;
logic                       txa_data_out_n;
```

```
logic                          test_out;
logic [5:0]                num_errors;
logic txd_data_out;
logic                 [2:0]       mode;//Bist Configuration pins
logic                      txd_out;
logic        [ 7 : 0 ] config_in;
logic                                 test_en;
reg tx_clk;
integer i;


//Module instantiation.
SerDes  #(.WIDTH(9)) serdes0 (
      .rst(rst),
      .clk(clk),
      .rxa_in_p(rxa_in_p),
      .rxa_in_n(rxa_in_n),
      .txd_data_in(txd_data_in),
      .config_in(config_in),
      .test_en(test_en),
      .digital_out(digital_out),
      .test_out(test_out),
      .txa_data_out_p(txa_data_out_p),
      .txa_data_out_n(txa_data_out_n),
      .txd_data_out(txd_data_out)
);

// Clock generation
 initial begin
         clk <= 1'b0;
      forever
      begin
      #10   clk <= ~clk;
      end
 end

assign tx_clk = serdes0.clock_div8_phases[0];
assign txd_out =  serdes0.txd_data_out;
assign mode = serdes0.test_modules0.mode;
assign num_errors = serdes0.test_modules0.num_errors;
assign txd_data_in_i = serdes0.test_modules0.txd_data_in_i;


task parallel(input[DATA_WIDTH:0] data);
      begin
           @(posedge tx_clk);
```

```
                txd_data_in = data;
                repeat (DATA_WIDTH + 1 )
            @(posedge tx_clk);
        end
endtask

task serial (input[9:0] data);
        for (integer i = 0; i <= 9 ; i = i + 1)
        begin
            rxa_in_p=data[i];//serial data
            @(posedge tx_clk);
        end
endtask

// input generation
initial
begin
    // reset
    #`DELAY  rst = 1'b1;
    #(`DELAY*10) rst = 1'b0;
      //Mode 01
       config_in=8'b0000_0001;
       test_en = 1'b1;
      //Send Comma
            serial(10'b0001111100);
              serial(10'h0b9);
              serial(10'h0ae);
              serial(10'h0ad);



    #(`DELAY*1000);
    $stop();
       //END MODE 1
end

assign rxa_in_n = ~rxa_in_p;





endmodule
```

7.1.4. SerDes_m2_tb.sv

```
`timescale 1ns/100ps
`define DELAY 10

module SerDes_m2_tb();

localparam DATA_WIDTH = 8;

logic                             rst;
logic                             clk;
logic                     rxa_in_p;
logic                     rxa_in_n;
logic [DATA_WIDTH:0]          txd_data_in;
logic [DATA_WIDTH:0]          txd_data_in_i;
logic [DATA_WIDTH:0]          digital_out;
logic                       txa_data_out_p;
logic                       txa_data_out_n;
logic                          test_out;
logic [5:0]                num_errors;
logic txd_data_out;
logic                 [2:0]       mode;//Bist Configuration pins
logic                       txd_out;
logic        [ 7 : 0 ] config_in;
logic                                 test_en;
reg tx_clk;
integer i;


//Module instantiation.
SerDes  #(.WIDTH(9)) serdes0 (
      .rst(rst),
      .clk(clk),
      .rxa_in_p(rxa_in_p),
      .rxa_in_n(rxa_in_n),
      .txd_data_in(txd_data_in),
      .config_in(config_in),
      .test_en(test_en),
      .digital_out(digital_out),
      .test_out(test_out),
      .txa_data_out_p(txa_data_out_p),
      .txa_data_out_n(txa_data_out_n),
      .txd_data_out(txd_data_out)
);

// Clock generation
 initial begin
         clk <= 1'b0;
```

```
     forever
     begin
     #10   clk <= ~clk;
     end
 end

assign tx_clk = serdes0.clock_div8_phases[0];
assign txd_out =  serdes0.txd_data_out;
assign mode = serdes0.test_modules0.mode;
assign num_errors = serdes0.test_modules0.num_errors;
assign txd_data_in_i = serdes0.test_modules0.txd_data_in_i;


task parallel(input[DATA_WIDTH:0] data);
     begin
          @(posedge tx_clk);
               txd_data_in = data;
               repeat (DATA_WIDTH + 1 )
          @(posedge tx_clk);
     end
endtask

// input generation
initial
begin
    // reset
    #`DELAY  rst = 1'b1;
    #(`DELAY*10) rst = 1'b0;
     //Mode 02: Serial loopback
      config_in=8'b0000_0010;
      test_en = 1'b1;



     //Send Comma
    @(posedge clk);
    // tx_data input generation
        parallel(9'b111111100); //Sending the first comma to start
        for (i = 0; i < 62; i = i + 1)
            parallel(i[ DATA_WIDTH :0]); //Send messages from 0 to
61
        parallel(9'b111111100); //Sending another comma to end
        for (i = 0; i < 62; i = i + 1)
            parallel(i[ DATA_WIDTH :0]); //Send messages from 0 to
61
    #(`DELAY*1000);
```

```
    $stop();
end




endmodule
```

7.1.5. SerDes_m3_tb.sv

```
`timescale 1ns/100ps
`define DELAY 10

module SerDes_m3_tb();

localparam DATA_WIDTH = 8;

logic                            rst;
logic                            clk;
logic                    rxa_in_p;
logic                    rxa_in_n;
logic [DATA_WIDTH:0]         txd_data_in;
logic [DATA_WIDTH:0]         txd_data_in_i;
logic [DATA_WIDTH:0]         digital_out;
logic                      txa_data_out_p;
logic                      txa_data_out_n;
logic                         test_out;
logic [5:0]               num_errors;
logic txd_data_out;
logic                 [2:0]       mode;//Bist Configuration pins
logic                      txd_out;
logic        [ 7 : 0 ] config_in;
logic                                test_en;
reg tx_clk;
integer i;


//Module instantiation.
SerDes  #(.WIDTH(9)) serdes0 (
      .rst(rst),
      .clk(clk),
      .rxa_in_p(rxa_in_p),
      .rxa_in_n(rxa_in_n),
      .txd_data_in(txd_data_in),
      .config_in(config_in),
      .test_en(test_en),
```

```
      .digital_out(digital_out),
      .test_out(test_out),
      .txa_data_out_p(txa_data_out_p),
      .txa_data_out_n(txa_data_out_n),
      .txd_data_out(txd_data_out)
);

// Clock generation
 initial begin
          clk <= 1'b0;
      forever
      begin
      #10   clk <= ~clk;
      end
 end

assign tx_clk = serdes0.clock_div8_phases[0];
assign txd_out =  serdes0.txd_data_out;
assign mode = serdes0.test_modules0.mode;
assign num_errors = serdes0.test_modules0.num_errors;
assign txd_data_in_i = serdes0.test_modules0.txd_data_in_i;


task parallel(input[DATA_WIDTH:0] data);
      begin
           @(posedge tx_clk);
                 txd_data_in = data;
                 repeat (DATA_WIDTH + 1 )
           @(posedge tx_clk);
      end
endtask

// input generation
initial
begin
    // reset
    #`DELAY  rst = 1'b1;
    #(`DELAY*10) rst = 1'b0;
      //Mode 3
       config_in[2:0]=3'b011;
       config_in[4]=1'b0;
       test_en = 1'b1;
      //Send Comma
    @(posedge clk);
    // tx_data input generation
        parallel(9'b111111100); //Sending the first comma to start
```

```
        for (i = 0; i < 62; i = i + 1)
            parallel(i[ DATA_WIDTH :0]); //Send messages from 0 to
61
        parallel(9'b111111100); //Sending another comma to end
        for (i = 0; i < 62; i = i + 1)
            parallel(i[ DATA_WIDTH :0]); //Send messages from 0 to
61
    #(`DELAY*1000);
    $stop();
end

assign config_in[3] = txa_data_out_p;

endmodule
```

7.1.6. SerDes_m4_tb.sv

```
`timescale 1ns/100ps
`define DELAY 10

module SerDes_m4_tb();

localparam DATA_WIDTH = 8;

logic                            rst;
logic                            clk;
logic                     rxa_in_p;
logic                     rxa_in_n;
logic [DATA_WIDTH:0]          txd_data_in;
logic [DATA_WIDTH:0]          txd_data_in_i;
logic [DATA_WIDTH:0]          digital_out;
logic                       txa_data_out_p;
logic                       txa_data_out_n;
logic                          test_out;
logic [5:0]                num_errors;
logic txd_data_out;
logic                 [2:0]       mode;//Bist Configuration pins
logic                       txd_out;
logic        [ 7 : 0 ] config_in;
logic                                test_en;
reg tx_clk;
logic [DATA_WIDTH:0] lfsr_parallel_out;

integer i;
```

```
//Module instantiation.
SerDes  #(.WIDTH(9)) serdes0 (
      .rst(rst),
      .clk(clk),
      .rxa_in_p(rxa_in_p),
      .rxa_in_n(rxa_in_n),
      .txd_data_in(txd_data_in),
      .config_in(config_in),
      .test_en(test_en),
      .digital_out(digital_out),
      .test_out(test_out),
      .txa_data_out_p(txa_data_out_p),
      .txa_data_out_n(txa_data_out_n),
      .txd_data_out(txd_data_out)
);

// Clock generation
 initial begin
         clk <= 1'b0;
      forever
      begin
      #10   clk <= ~clk;
      end
 end

assign tx_clk = serdes0.clock_div8_phases[0];
assign txd_out =  serdes0.txd_data_out;
assign mode = serdes0.test_modules0.mode;
assign num_errors = serdes0.test_modules0.num_errors;
assign txd_data_in_i = serdes0.test_modules0.txd_data_in_i;
assign lfsr_parallel_out =
serdes0.test_modules0.lfsr_parallel_out;


task parallel(input[DATA_WIDTH:0] data);
      begin
            @(posedge tx_clk);
                  txd_data_in = data;
                  repeat (DATA_WIDTH + 1 )
            @(posedge tx_clk);
      end
endtask



// input generation
```

```
initial
begin
    // reset
    #`DELAY  rst = 1'b1;
    #`DELAY rst = 1'b0;
      //Mode 3
       config_in=8'b0000_0100;
       test_en = 1'b1;
      //Send Comma
    @(posedge tx_clk);
    // tx_data input generation
        for (i = 0; i < 62; i = i + 1)
             @(posedge tx_clk)
    #(`DELAY*1000);
    $stop();
end

endmodule
```

7.1.7. SerDes_m5_tb.sv

```
`timescale 1ns/100ps
`define DELAY 10

module SerDes_m5_tb();

localparam DATA_WIDTH = 8;

logic                             rst;
logic                             clk;
logic                     rxa_in_p;
logic                     rxa_in_n;
logic [DATA_WIDTH:0]          txd_data_in;
logic [DATA_WIDTH:0]          txd_data_in_i;
logic [DATA_WIDTH:0]          digital_out;
logic                       txa_data_out_p;
logic                       txa_data_out_n;
logic                          test_out;
logic [5:0]                num_errors;
logic txd_data_out;
logic                 [2:0]       mode;//Bist Configuration pins
logic                       txd_out;
logic        [ 7 : 0 ] config_in;
logic                                test_en;
reg tx_clk;
integer i;
```

```
//Module instantiation.

SerDes  #(.WIDTH(9)) serdes0 (
      .rst(rst),
      .clk(clk),
      .rxa_in_p(rxa_in_p),
      .rxa_in_n(rxa_in_n),
      .txd_data_in(txd_data_in),
      .config_in(config_in),
      .test_en(test_en),
      .digital_out(digital_out),
      .test_out(test_out),
      .txa_data_out_p(txa_data_out_p),
      .txa_data_out_n(txa_data_out_n),
      .txd_data_out(txd_data_out)
);

// Clock generation
 initial begin
          clk <= 1'b0;
      forever
      begin
      #10   clk <= ~clk;
      end
 end

assign tx_clk = serdes0.clock_div8_phases[0];
assign txd_out =  serdes0.txd_data_out;
assign mode = serdes0.test_modules0.mode;
assign num_errors = serdes0.test_modules0.num_errors;
assign txd_data_in_i = serdes0.test_modules0.txd_data_in_i;

task parallel(input[DATA_WIDTH:0] data);
      begin
           @(posedge tx_clk);
                 txd_data_in = data;
                 repeat (DATA_WIDTH + 1 )
           @(posedge tx_clk);
      end
endtask

task serial (input[9:0] data);
      for (integer i = 0; i <= 9 ; i = i + 1)
      begin
```

```
            config_in[3]=data[i];//serial data
            @(posedge tx_clk);
      end
endtask

// input generation
initial
begin
    // reset
    #`DELAY  rst = 1'b1;
    #`DELAY rst = 1'b0;
      //Mode 5
       config_in[2:0]=3'b101;
       config_in[4]=1'b0;
       test_en = 1'b1;
      //Send Comma
           serial(10'b0001111100);
              serial(10'h0b9);
              serial(10'h0ae);
              serial(10'h0ad);



    #(`DELAY*1000);
    $stop();
end
endmodule
```

7.1.8. SerDes_m6_tb.sv

```
`timescale 1ns/100ps
`define DELAY 10

module SerDes_m6_tb(); // Open Bist

localparam DATA_WIDTH = 8;

logic                             rst;
logic                             clk;
logic                     rxa_in_p;
logic                     rxa_in_n;
logic [DATA_WIDTH:0]          txd_data_in;
logic [DATA_WIDTH:0]          txd_data_in_i;
logic [DATA_WIDTH:0]          digital_out;
logic                       txa_data_out_p;
logic                       txa_data_out_n;
```

```
logic                          test_out;
logic [5:0]                num_errors;
logic txd_data_out;
logic                [2:0]       mode;//Bist Configuration pins
logic                       txd_out;
logic         [ 7 : 0 ] config_in;
logic                                 test_en;
reg tx_clk;
logic [DATA_WIDTH:0] lfsr_parallel_out;

integer i;


//Module instantiation.

SerDes  #(.WIDTH(9)) serdes0 (
      .rst(rst),
      .clk(clk),
      .rxa_in_p(rxa_in_p),
      .rxa_in_n(rxa_in_n),
      .txd_data_in(txd_data_in),
      .config_in(config_in),
      .test_en(test_en),
      .digital_out(digital_out),
      .test_out(test_out),
      .txa_data_out_p(txa_data_out_p),
      .txa_data_out_n(txa_data_out_n),
      .txd_data_out(txd_data_out)
);

// Clock generation
 initial begin
         clk <= 1'b0;
      forever
      begin
      #10   clk <= ~clk;
      end
 end

assign tx_clk = serdes0.clock_div8_phases[0];
assign txd_out =  serdes0.txd_data_out;
assign mode = serdes0.test_modules0.mode;
assign num_errors = serdes0.test_modules0.num_errors;
assign txd_data_in_i = serdes0.test_modules0.txd_data_in_i;
assign lfsr_parallel_out =
serdes0.test_modules0.lfsr_parallel_out;
```

```
task parallel(input[DATA_WIDTH:0] data);
     begin
          @(posedge tx_clk);
               txd_data_in = data;
               repeat (DATA_WIDTH + 1 )
          @(posedge tx_clk);
     end
endtask

// input generation
initial
begin
    // reset
    #`DELAY  rst = 1'b1;
    #`DELAY rst = 1'b0;
      config_in=8'b0000_0110;
      test_en = 1'b1;
     //Send Comma
    @(posedge tx_clk);
        for (i = 0; i < 62; i = i + 1)
             @(posedge tx_clk)
    #(`DELAY*1000);
    $stop();

end

endmodule
```

7.1.9. SerDes_m7_tb.sv

```
`timescale 1ns/100ps
`define DELAY 10

module SerDes_m7_tb(); // Analog Receiver Output and analog
Loopback

localparam DATA_WIDTH = 8;

logic                               rst;
logic                               clk;
logic                       rxa_in_p;
logic                       rxa_in_n;
logic [DATA_WIDTH:0]            txd_data_in;
logic [DATA_WIDTH:0]            txd_data_in_i;
logic [DATA_WIDTH:0]            digital_out;
logic                         txa_data_out_p;
```

```
logic                       txa_data_out_n;
logic                          test_out;
logic [5:0]                num_errors;
logic txd_data_out;
logic                 [2:0]       mode;
logic                       txd_out;
logic        [ 7 : 0 ] config_in;
logic                                test_en;
reg tx_clk;
logic [DATA_WIDTH:0] lfsr_parallel_out;

integer i;


//Module instantiation.

SerDes  #(.WIDTH(9)) serdes0 (
     .rst(rst),
     .clk(clk),
     .rxa_in_p(rxa_in_p),
     .rxa_in_n(rxa_in_n),
     .txd_data_in(txd_data_in),
     .config_in(config_in),
     .test_en(test_en),
     .digital_out(digital_out),
     .test_out(test_out),
     .txa_data_out_p(txa_data_out_p),
     .txa_data_out_n(txa_data_out_n),
     .txd_data_out(txd_data_out)
);

// Clock generation
 initial begin
         clk <= 1'b0;
     forever
     begin
     #10   clk <= ~clk;
     end
 end

assign tx_clk = serdes0.clock_div8_phases[0];
assign txd_out =  serdes0.txd_data_out;
assign mode = serdes0.test_modules0.mode;
assign num_errors = serdes0.test_modules0.num_errors;
assign txd_data_in_i = serdes0.test_modules0.txd_data_in_i;
assign lfsr_parallel_out =
```

```
serdes0.test_modules0.lfsr_parallel_out;
assign txd_data_out = serdes0.test_modules0.txd_data_out;

task parallel(input[DATA_WIDTH:0] data);
     begin
          @(posedge tx_clk);
               txd_data_in = data;
               repeat (DATA_WIDTH + 1 )
          @(posedge tx_clk);
     end
endtask

// input generation
initial
begin
    // reset
    #`DELAY  rst = 1'b1;
    #(`DELAY*10) rst = 1'b0;
      //Mode 7
       config_in=8'b0000_0111;
       test_en = 1'b1;
      //Send Comma
    @(posedge clk);
    // tx_data input generation
        parallel(9'b111111100); //Sending the first comma to start
        for (i = 0; i < 62; i = i + 1)
            parallel(i[ DATA_WIDTH :0]); //Send messages from 0 to
61
        parallel(9'b111111100); //Sending another comma to end
        for (i = 0; i < 62; i = i + 1)
            parallel(i[ DATA_WIDTH :0]); //Send messages from 0 to
61
    #(`DELAY*1000);
    $stop();
end

assign rxa_in_n = ~txd_data_out;
assign rxa_in_p = txd_data_out;



endmodule
```

7.2. Synthesis files

7.2.1. Logic_synthesis.tcl

```
#### Template Script for RTL->Gate-Level Flow (generated from RC RC14.26
- v14.20-s058_1)

if {[file exists /proc/cpuinfo]} {
  sh grep "model name" /proc/cpuinfo
  sh grep "cpu MHz"    /proc/cpuinfo
}

puts "Hostname : [info hostname]"

##########################################################################
#####
## Preset global variables and attributes
##########################################################################
#####


set DESIGN test_modules
set SYN_EFF high
set MAP_EFF high
set DATE [clock format [clock seconds] -format "%b%d-%T"]
set _OUTPUTS_PATH outputs_${DATE}
set _REPORTS_PATH reports_${DATE}
set _LOG_PATH logs_${DATE}

# Variable to specify the technology .lib file name
set           timing_library           {scx3_cmos8rf_lpvt_tt_1p2v_25c.lib
iogpil_cmrf8sf_rvt_tt_1p2v_2p5v_25c.lib}

set  my_lef_library  {/opt/libs/ARM/IB03LB501-FB-00000-r0p0-00rel0/aci/sc-
x/lef/ibm13_8lm_2thick_3rf_tech.lef     /opt/libs/ARM/IB03IG502-FB-00000-
r0p0-00rel0/aci/io/lef/iogpil_cmrf8sf_rvt_M2_3_3.lef
/opt/libs/ARM/IB03LB501-FB-00000-r0p0-00rel0/aci/sc-
x/lef/ibm13rflpvt_macros.lef}

set_attribute   lib_search_path   {/opt/libs/ARM/IB03LB501-FB-00000-r0p0-
00rel0/aci/sc-x/synopsys           /opt/libs/ARM/IB03IG502-FB-00000-r0p0-
00rel0/aci/io/synopsys/} /
set_attribute script_search_path {..} /
set_attribute hdl_search_path {..} /

##Uncomment and specify machine names to enable super-threading.
##set_attribute super_thread_servers {<machine names>} /

##Default undriven/unconnected setting is 'none'.
##set_attribute hdl_unconnected_input_port_value 0 | 1 | x | none /
##set_attribute hdl_undriven_output_port_value   0 | 1 | x | none /
##set_attribute hdl_undriven_signal_value        0 | 1 | x | none /
```

```
##set_attribute wireload_mode <value> /
set_attribute information_level 9 /

###############################################################
## Library setup
###############################################################

set_attribute library $timing_library
set_attribute lef_library $my_lef_library /
#set_attribute cap_table_file <file> /
set_attribute auto_ungroup none /


##set_attribute congestion_effort <low|medium|high> /
##generates <signal>_reg[<bit_width>] format
#set_attribute hdl_array_naming_style %s\[%d\] /
################


####################################################################
## Load Design
####################################################################


read_hdl  -sv  {  ../test_modules.sv  ../comparator.sv  ../signal_driver.sv
../lfsr.sv}
elaborate $DESIGN
puts "Runtime & Memory after 'read_hdl'"
timestat Elaboration


check_design -unresolved

####################################################################
## Constraints Setup
####################################################################

read_sdc { ../test_modules.sdc}
puts  "The  number  of  exceptions  is  [llength  [find  /designs/$DESIGN  -
exception *]]"


#set_attribute force_wireload <wireload name> "/designs/$DESIGN"

if {![file exists ${_LOG_PATH}]} {
  file mkdir ${_LOG_PATH}
  puts "Creating directory ${_LOG_PATH}"
}

if {![file exists ${_OUTPUTS_PATH}]} {
```

```
  file mkdir ${_OUTPUTS_PATH}
  puts "Creating directory ${_OUTPUTS_PATH}"
}

if {![file exists ${_REPORTS_PATH}]} {
  file mkdir ${_REPORTS_PATH}
  puts "Creating directory ${_REPORTS_PATH}"
}
report timing -lint


#########################################################################
##########
## Define  cost  groups  (clock-clock,  clock-output,  input-clock,  input-
output)
#########################################################################
##########

## Uncomment to remove already existing costgroups before creating new
ones.
## rm [find /designs/* -cost_group *]

if {[llength [all::all_seqs]] > 0} {
  define_cost_group -name I2C -design $DESIGN
  define_cost_group -name C2O -design $DESIGN
  define_cost_group -name C2C -design $DESIGN
  path_group -from [all::all_seqs] -to [all::all_seqs] -group C2C -name
C2C
  path_group -from [all::all_seqs] -to [all::all_outs] -group C2O -name
C2O
  path_group -from [all::all_inps]  -to [all::all_seqs] -group I2C -name
I2C
}

define_cost_group -name I2O -design $DESIGN
path_group -from [all::all_inps]  -to [all::all_outs] -group I2O -name I2O
foreach cg [find / -cost_group *] {
  report       timing       -cost_group       [list       $cg]       >>
$_REPORTS_PATH/${DESIGN}_pretim.rpt
}
#########################################################################
##############
## Leakage/Dynamic power/Clock Gating setup.
#########################################################################
##############


#set_attribute     lp_clock_gating_cell     [find     /lib*     -libcell
<cg_libcell_name>] "/designs/$DESIGN"
#set_attribute lp_power_unit {uW}
```

```
#set_attribute lp_pso_aware_estimation true
#set_attribute leakage_power_effort high
#set_attribute max_leakage_power 100 "/designs/$DESIGN"
#set_attribute lp_power_optimization_weight 0.5 "/designs/$DESIGN"
#set_attribute max_dynamic_power 100 "/designs/$DESIGN"
## read_tcf <TCF file name>
## read_saif <SAIF file name>
## read_vcd <VCD file name>



#### To turn off sequential merging on the design
#### uncomment & use the following attributes.
##set_attribute optimize_merge_flops false /
##set_attribute optimize_merge_latches false /
#### For a particular instance use attribute 'optimize_merge_seqs' to turn
off sequential merging.



########################################################################
###########################
## Synthesizing to generic
########################################################################
###########################


synthesize -to_generic -eff $SYN_EFF
puts "Runtime & Memory after 'synthesize -to_generic'"
timestat GENERIC

report power > $_REPORTS_PATH/${DESIGN}_generic_power.rpt
#write_design            -encounter            -gzip            -basename
${_OUTPUTS_PATH}/generic/${DESIGN}
generate_reports -outdir $_REPORTS_PATH -tag generic
## syntax : generate_reports -outdir <out dir> -tag <tag> [-encounter]
## syntax : summary_table -outdir <out dir>
#generate_reports -outdir $_REPORTS_PATH -tag generic
summary_table -outdir $_REPORTS_PATH



########################################################################
###########################
## Synthesizing to gates
########################################################################
###########################

synthesize -to_mapped -eff $MAP_EFF -no_incr
puts "Runtime & Memory after 'synthesize -to_map -no_incr'"
```

```
timestat MAPPED

report power > $_REPORTS_PATH/${DESIGN}_map_power.rpt
foreach cg [find / -cost_group *] {
  report       timing       -cost_group       [list       $cg]       >
$_REPORTS_PATH/${DESIGN}_[basename $cg]_post_map.rpt
}
generate_reports -outdir $_REPORTS_PATH -tag map
summary_table -outdir $_REPORTS_PATH
write_design -encounter -gzip -basename ${_OUTPUTS_PATH}/map/${DESIGN}


##Intermediate netlist for LEC verification..
write_hdl -lec > ${_OUTPUTS_PATH}/${DESIGN}_intermediate.v
write_do_lec -revised_design ${_OUTPUTS_PATH}/${DESIGN}_intermediate.v -
logfile              ${_LOG_PATH}/rtl2intermediate.lec.log              >
${_OUTPUTS_PATH}/rtl2intermediate.lec.do

## ungroup -threshold <value>

#########################################################################
##############################
## Incremental Synthesis
#########################################################################
##############################

## Uncomment to remove assigns & insert tiehilo cells during Incremental
synthesis
##set_attribute remove_assigns true /
##set_remove_assign_options   -buffer_or_inverter   <libcell>   -design
<design|subdesign>
##set_attribute use_tiehilo_for_const <none|duplicate|unique> /
synthesize -to_mapped -eff $MAP_EFF -incr
report power > $_REPORTS_PATH/${DESIGN}_incremental_power.rpt
generate_reports -outdir $_REPORTS_PATH -tag incremental
summary_table -outdir $_REPORTS_PATH

puts "Runtime & Memory after incremental synthesis"
timestat INCREMENTAL

foreach cg [find / -cost_group *] {
  report       timing       -cost_group       [list       $cg]       >
$_REPORTS_PATH/${DESIGN}_[basename $cg]_post_incr.rpt
}


###################################################
## Spatial mode optimization
###################################################
```

```
## Uncomment to enable spatial mode optimization
##synthesize -to_mapped -spatial


#########################################################################
#############################
## write Encounter file set (verilog, SDC, config, etc.)
#########################################################################
#############################


##write_encounter   design   -basename   <path   &   base   filename>   -lef
<lef_file(s)>

report clock_gating > $_REPORTS_PATH/${DESIGN}_clockgating.rpt
report power -depth 0 > $_REPORTS_PATH/${DESIGN}_power.rpt
report gates -power > $_REPORTS_PATH/${DESIGN}_gates_power.rpt

##report qor > $_REPORTS_PATH/${DESIGN}_qor.rpt
report area > $_REPORTS_PATH/${DESIGN}_area.rpt
report datapath > $_REPORTS_PATH/${DESIGN}_datapath_incr.rpt
report messages > $_REPORTS_PATH/${DESIGN}_messages.rpt
write_design -basename ${_OUTPUTS_PATH}/${DESIGN}_m
write_hdl  > ${_OUTPUTS_PATH}/${DESIGN}_m.v
## write_script > ${_OUTPUTS_PATH}/${DESIGN}_m.script
write_sdc > ${_OUTPUTS_PATH}/${DESIGN}_m.sdc


#################################
### write_do_lec
#################################


write_do_lec  -golden_design  ${_OUTPUTS_PATH}/${DESIGN}_intermediate.v  -
revised_design          ${_OUTPUTS_PATH}/${DESIGN}_m.v          -logfile
${_LOG_PATH}/intermediate2final.lec.log                               >
${_OUTPUTS_PATH}/intermediate2final.lec.do
##Uncomment if the RTL is to be compared with the final netlist..
##write_do_lec  -revised_design  ${_OUTPUTS_PATH}/${DESIGN}_m.v  -logfile
${_LOG_PATH}/rtl2final.lec.log > ${_OUTPUTS_PATH}/rtl2final.lec.do

puts "Final Runtime & Memory."
timestat FINAL
puts "============================"
puts "Synthesis Finished ........."
puts "============================"

file copy [get_attr stdout_log /] ${_LOG_PATH}/.

##quit
```

7.2.2. Full_Synthesis_EDI_test_modules.tcl

```
## script Full EDI Flow ##

## import design ##

set_global _enable_mmmc_by_default_flow      $CTE::mmmc_default
suppressMessage ENCEXT-2799
win
set ::TimeLib::tsgMarkCellLatchConstructFlag 1
set conf_qxconf_file NULL
set conf_qxlib_file NULL
set defHierChar /
set distributed_client_message_echo 1
set gpsPrivate::dpgNewAddBufsDBUpdate 1
set gpsPrivate::lsgEnableNewDbApiInRestruct 1
set init_gnd_net {VSS DVSS}
set init_io_file ../test_modules_arm.ioc
set init_lef_file {/opt/libs/ARM/IB03LB501-FB-00000-r0p0-00rel0/aci/sc-
x/lef/ibm13_8lm_2thick_3rf_tech.lef /opt/libs/ARM/IB03LB501-FB-00000-r0p0-
00rel0/aci/sc-x/lef/ibm13rflpvt_macros.lef /opt/libs/ARM/IB03IG502-FB-00000-r0p0-
00rel0/aci/io/lef/iogpil_cmrf8sf_rvt_M2_3_3.lef}
set init_mmmc_file ../Typ_WC.view
set init_pwr_net {VDD DVDD}
set init_verilog ../test_modules_m.v
set lsgOCPGainMult 1.000000
set pegDefaultResScaleFactor 1.000000
set pegDetailResScaleFactor 1.000000
set timing_library_float_precision_tol 0.000010
set timing_library_load_pin_cap_indices {}
set tso_post_client_restore_command {update_timing ; write_eco_opt_db ;}
init_design

# Defining process mode
setDesignMode -process 130

## Floor plan definition

getIoFlowFlag
setFPlanRowSpacingAndType 3.6 2
#3.6
setIoFlowFlag 0
#floorPlan -site IBM13SITE -s 1280.4 1280.4 16.8 16.8 16.8 16.8
#floorPlan -dieSizeByIoHeight max -site IBM13SITE -s 1280.4 1280.4 16.8 16.8 16.8 16.8
floorPlan -dieSizeByIoHeight max -site IBM13SITE -s 1280.4 1280.4 16.8 16.8 16.8 16.8
uiSetTool select
getIoFlowFlag
```

Defining Power Global Nets

```
clearGlobalNets
globalNetConnect VDD -type pgpin -pin VDD -inst * -module {} -verbose
globalNetConnect VSS -type pgpin -pin VSS -inst * -module {} -verbose
globalNetConnect VSS -type tielo -pin VSS -inst * -module {} -verbose
globalNetConnect VDD -type tiehi -pin VDD -inst * -module {} -verbose
```

Adding power Ring

```
addRing -skip_via_on_wire_shape Noshape -skip_via_on_pin Standardcell -
stacked_via_top_layer MA -type core_rings -jog_distance 0.2 -threshold 0.2 -nets {VDD
VSS} -follow io -stacked_via_bottom_layer M1 -layer {bottom M1 top M1 right M2 left
M2} -width 4 -spacing 5 -offset 2
 #jog_distance 0.2 threshold 0.2 spacing 5
```

Adding Horizontal lines

```
sroute -connect { blockPin padPin padRing corePin floatingStripe } -layerChangeRange {
M1 MA } -blockPinTarget { nearestTarget } -padPinPortConnect { allPort oneGeom } -
padPinTarget { nearestTarget } -corePinTarget { firstAfterRowEnd } -floatingStripeTarget {
blockring padring ring stripe ringpin blockpin followpin } -allowJogging 1 -
crossoverViaLayerRange { M1 MA } -allowLayerChange 1 -nets { VDD VSS } -blockPin
useLef -targetViaLayerRange { M1 MA }
```

Adding strip lines

```
#addStripe -skip_via_on_wire_shape Noshape -block_ring_top_layer_limit M3 -
max_same_layer_jog_length 8 -padcore_ring_bottom_layer_limit M1 -number_of_sets 4 -
skip_via_on_pin Standardcell -stacked_via_top_layer MA -padcore_ring_top_layer_limit
M3 -spacing 5 -xleft_offset 50 -xright_offset 50 -merge_stripes_value 0.2 -layer M2 -
block_ring_bottom_layer_limit M1 -width 4 -nets {VDD VSS} -stacked_via_bottom_layer
M1
addStripe -skip_via_on_wire_shape Noshape -block_ring_top_layer_limit M3 -
max_same_layer_jog_length 8 -padcore_ring_bottom_layer_limit M1 -number_of_sets 4 -
skip_via_on_pin Standardcell -stacked_via_top_layer MA -padcore_ring_top_layer_limit
M3 -spacing 5 -xleft_offset 50 -xright_offset 50 -merge_stripes_value 0.2 -layer M2 -
block_ring_bottom_layer_limit M1 -width 4 -nets {VDD VSS} -stacked_via_bottom_layer
M1
```

Place Standard Cells

```
setEndCapMode -reset
setEndCapMode -boundary_tap false
setPlaceMode -reset
setPlaceMode -congEffort auto -timingDriven 1 -modulePlan 1 -clkGateAware 1 -
```

```
powerDriven 0 -ignoreScan 0 -reorderScan 0 -ignoreSpare 0 -placeIOPins 1 -
moduleAwareSpare 0 -preserveRouting 0 -rmAffectedRouting 0 -checkRoute 0 -swapEEQ
0
setPlaceMode -fp false
placeDesign

## Clock synthesis CTS

# Use the FE-CTS
setCTSMode -engine ck

# Create clock tree using the clock buffers list:
createClockTreeSpec -bufferList {CLKBUFX2TS CLKBUFX3TS CLKBUFX4TS
CLKBUFX6TS CLKBUFX8TS CLKBUFX12TS CLKBUFX16TS CLKBUFX20TS} -file
../test_modules.ctstch

# Display Clock Tree
displayClockTree -skew -allLevel -preRoute

# Edit the .ctstch that was created to complete constraints...

## Route design with nano route

setNanoRouteMode -quiet -timingEngine {}
setNanoRouteMode -quiet -routeWithTimingDriven 1
setNanoRouteMode -quiet -routeWithSiDriven 1
setNanoRouteMode -quiet -routeWithSiPostRouteFix 0
setNanoRouteMode -quiet -routeTdrEffort 10
setNanoRouteMode -quiet -drouteStartIteration default
setNanoRouteMode -quiet -routeTopRoutingLayer default
setNanoRouteMode -quiet -routeBottomRoutingLayer default
setNanoRouteMode -quiet -drouteEndIteration default
setNanoRouteMode -quiet -routeWithTimingDriven true
setNanoRouteMode -quiet -routeWithSiDriven true
routeDesign -globalDetail


####### Run optimization script based on class script ###########

Puts "Timing the design before CTS"

# Calculates the delays for paths based on max. opertating conditions (op) and min. op.
setAnalysisMode -analysisType onChipVariation

timeDesign -preCTS -prefix preCTS_setup
timeDesign -preCTS -prefix preCTS_hold  -hold
```

```
Puts "Running CTS"
dbDeleteTrialRoute
clockDesign -specFile ../Clock_test_modules.ctstch -outDir clock_report -
fixedInstBeforeCTS
Puts "Finished running CTS"

Puts "Timing the design after CTS"
timeDesign -postCTS -prefix postCTS_setup
timeDesign -postCTS -prefix postCTS_hold  -hold

Puts "Setting Optimizaiton Mode Options for DRV fixes"
setOptMode -fixFanoutLoad true
setOptMode -fixDRC true
setOptMode -addInstancePrefix postCTSdrv
setOptMode -setupTargetSlack 0.05


Puts "Optimizing for DRV"
optDesign -postCTS -drv

Puts "Timing the design after DRV fixes"
timeDesign -postCTS -prefix postCTS_setup_DRVfix
timeDesign -postCTS -prefix postCTS_hold_DRVfix -hold

Puts "Setting Optimization Mode Options for Setup fixes"
setOptMode -addInstancePrefix postCTSsetup

Puts "Optimizing for Setup"
optDesign -postCTS

Puts  "Timing the design after Setup fixes"
timeDesign -postCTS -prefix postCTS_setup_Setupfix
timeDesign -postCTS -prefix postCTS_hold_Setupfix  -hold

setOptMode -addInstancePrefix postCTShold
optDesign -postCTS -hold
Puts  "Timing the design after Hold fixes"
timeDesign -postCTS -prefix postCTS_setup_Holdfix
timeDesign -postCTS -prefix postCTS_hold_Holdfix  -hold

Puts "Routing the Design"
setNanoRouteMode -quiet -timingEngine {}
setNanoRouteMode -quiet -routeWithSiPostRouteFix 0
setNanoRouteMode -quiet -routeTopRoutingLayer default
setNanoRouteMode -quiet -routeBottomRoutingLayer default
setNanoRouteMode -quiet -routeTdrEffort 10
setNanoRouteMode -quiet -drouteEndIteration default
```

setNanoRouteMode -quiet -routeWithTimingDriven True
setNanoRouteMode -quiet -routeWithSiDriven True
routeDesign -globalDetail

Puts "Timing the design after Route"
timeDesign -postRoute -prefix postRoute_setup
timeDesign -postRoute -prefix postRoute_hold -hold

Verify connectivity,DRC & Geometry

Geometry

setVerifyGeometryMode -area { 0 0 0 0 } -minWidth true -minSpacing true -minArea true -sameNet true -short true -overlap true -offRGrid false -offMGrid true -mergedMGridCheck true -minHole true -implantCheck true -minimumCut true -minStep true -viaEnclosure true -antenna false -insuffMetalOverlap true -pinInBlkg false -diffCellViol true -sameCellViol false -padFillerCellsOverlap true -routingBlkgPinOverlap true -routingCellBlkgOverlap true -regRoutingOnly false -stackedViasOnRegNet false -wireExt true -useNonDefaultSpacing false -maxWidth true -maxNonPrefLength -1 -error 1000
verifyGeometry
setVerifyGeometryMode -area { 0 0 0 0 }

DRC

setVerifyGeometryMode -area { 0 0 0 0 }
verify_drc -report test_modules.drc.rpt -limit 1000

Connectivity

verifyConnectivity -type all -error 1000 -warning 50

report power

set_power_analysis_mode -method static -analysis_view Typ_Analysis_View -corner max -create_binary_db true -write_static_currents true -honor_negative_energy true -ignore_control_signals true
set_power_output_dir -reset
set_power_output_dir power_report
set_default_switching_activity -reset
set_default_switching_activity -input_activity 0.2 -period 4.0 -seq_activity .5 -clock_gates_output 0
read_activity_file -reset
set_power -reset
set_powerup_analysis -reset
set_dynamic_power_simulation -reset
report_power -rail_analysis_format VS -outfile power_report/test_modules.rpt

```
# save design

saveDesign test_modules.enc
```