IMPLEMENTATION OF PATTERN GENERATION AND PATTERN MATCHING IN SERDES AND INTRODUCE NEW SET OF READABLE INSTRUCTIONS SET

A thesis submitted in partial fulfillment of the requirements for the award of the degree of

B. Tech

in

Electronics and Communication Engineering

By

P R MADHU VARSHAN (108121067)

ELECTRONICS AND COMMUNICATION ENGINEERING

NATIONAL INSTITUTEOF TECHNOLOGY

TIRUCHIRAPALLI-620015

MAY 2025

IMPLEMENTATION OF PATTERN GENERATION AND PATTERN MATCHING IN SERDES AND INTRODUCE NEW SET OF READABLE INSTRUCTIONS SET

A thesis submitted in partial fulfillment of the requirements for the award of the degree of

B. Tech

in

Electronics and Communication Engineering

By

P R MADHU VARSHAN (108121067)

ELECTRONICS AND COMMUNICATION ENGINEERING

NATIONAL INSTITUTEOF TECHNOLOGY

TIRUCHIRAPALLI-620015

MAY 2025

BONAFIDE CERTIFICATE

This is to certify that the project titled **'Implementation Of Pattern Generation And Pattern Matching In Serdes And Introduce New Set Of Readable Instructions Set'** is a bonafide record of the work done by

Madhu Varshan P R (108121067)

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Electronics and Communication Engineering** of the **NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI,** during the year 2021-2025.

Dr.Bukke Chandrababu Naik
Guide

Dr. M.Bhaskar
Head of the Department

Project Viva-voce held on ________________________________

Internal Examiner

External Examiner

ABSTRACT

The primary objective of this project is to design and implement a robust system for generating pseudo-random binary sequences (PRBS) using a linear feedback shift register (LFSR) and to evaluate its performance through SONET testing, error testing, and stress testing. The project focuses on optimizing power consumption through the introduction of clock gating[2] modules and ensures proper synchronization between the transmitter and receiver using synchronizers and delay elements.

The PRBS generator employs LFSRs of different polynomial sizes and operates in multiple modes, including DC-balanced patterns and extended user-defined modes. Clock gating ensures minimal power wastage, and synchronizers prevent metastability issues during clock domain crossing. Thorough testing using Spyglass lint checks verified the design integrity.

In addition to the hardware implementation, a supplementary scripting framework was developed using Perl to improve the readability and maintainability of firmware instruction sets. This script automates the conversion of repetitive instruction sequences—such as ten-line multiplication routines in assembly—into single high-level commands like MUL(reg_a, reg_b, reg_c). This significantly reduces manual coding errors, improves firmware clarity, and accelerates development and debugging, especially for calibration and adaptation modules.

Results show efficient error detection, low power consumption, and reliable data pattern generation and matching, making the system suitable for high-speed optical communication and VLSI verification environments.

Keywords: PRBS, LFSR, Clock Gating, Synchronization, SONET Testing, Firmware Optimization, Instruction Compression

ACKNOWLEDGEMENTS

I would like to express our sincere gratitude to our project guide **Dr.Bukke Chandrababu Naik**, Assistant Professor in the Department of Electronics and Communication Engineering, National Institute of Technology, Tiruchirappalli, for providing me an opportunity to work with him. While providing constant support throughout the project. He greatly helped us understand the concepts involved and pushed us to think more critically. Without his guidance, we would not have been able to complete the project successfully. His valuable feedback helped us a lot in avoiding potential pitfalls and dealing with errors that did occur. His patience and genial attitude will always be a source of inspiration to us.

I wish to express my sincere thanks to **Mr.Ramakrishna Uppalapati** for allowing me to work on this project and for his constant support and guidance throughout the project work.

My hearty thanks to all the members of the **Project Coordination Committee**, Department of Electronics and Communication Engineering, National Institute of Technology, Tiruchirappalli, for their precious suggestions and comments during the project reviews.

We would also like to thank SYNOPSYS for providing the right environment and our individual parents and friends for their constant support and help.

Madhu Varshan P R(108121067)

TABLE OF CONTENTS

List of Tables

List of Figures

ABBREVIATIONS

PRBS	Pseudo-Random Binary Sequence
LFSR	Linear Feedback Shift Register
SERDES	Serializer/Deserializer
FSM	Finite State Machine
ASIC	Application-Specific Integrated Circuit
SoC	System on Chip
CDC	Clock Domain Crossing
JSON	JavaScript Object Notation
DUT	Device Under Test

CHAPTER 1

INTRODUCTION

1.1 General Motivation

The advancement of digital communication systems has significantly raised the need for reliable, efficient, and high-performance data transmission mechanisms. In high-speed serial links and system-on-chip (SoC) environments, the serialization and deserialization (SerDes) of data are fundamental processes. These systems demand not only accurate data transfer but also robustness against timing errors, metastability, and noise introduced due to high-frequency operations. As such, testing and validation of SerDes modules have become critical tasks in modern VLSI and communication system design.

One of the most widely adopted techniques to validate these systems is pattern-based testing using Pseudo-Random Binary Sequences (PRBS). PRBS[1] provides a controllable yet randomized stream of bits that simulates real-world traffic and can be used to stress-test both hardware and firmware components. Implementing PRBS using Linear Feedback Shift Registers (LFSRs) enables efficient hardware realization of such sequences across different polynomial configurations, supporting various applications, including DC-balanced data testing and user-defined test modes.

While the hardware components ensure functional correctness and timing compliance, the firmware side presents challenges of its own. Writing repetitive low-level assembly instructions for tasks like calibration and adaptation can be time-consuming and error-prone. For example, a simple arithmetic operation like multiplication might span several lines of hand-written code, increasing the risk of syntactic and logical errors, especially in large-scale firmware projects.

To address this, a key motivation of the project is to improve firmware development workflows through automation. A custom Perl script has been designed that parses and transforms verbose instruction sets into single-line, high-level operations. This drastically improves the readability and maintainability of firmware code. For instance, a ten-line assembly routine for multiplication can be simplified to a single instruction MUL(reg_a, reg_b, reg_c). This enhancement not only reduces manual error but also accelerates development and debugging in time-critical SoC environments.

Combining these hardware and firmware optimizations, the project aims to deliver a unified testing and verification platform that is power-efficient, reliable, and developer-friendly — characteristics crucial for future communication and VLSI systems.

Our Contributions

1) Developed a PRBS generator in Verilog using multiple LFSR modes (LFSR7, LFSR9, LFSR13, up to LFSR31) to support various pattern generation configurations.

2) Implemented the error count module and integrated it with the pattern matcher to detect and log bit-level mismatches.

3) Learned and applied Perl and Verilog to work across both firmware and hardware aspects of the project.

4) Wrote a Perl optimization script that converts repetitive assembly sequences into high-level macro instructions, improving firmware readability and reducing errors.

1.2 Introduction to SERDES

In high-speed VLSI communication systems, **SERDES (Serializer/Deserializer)** blocks are essential components that help reduce the number of data lines by converting parallel data into serial format for transmission and then converting it back to parallel at the receiving end. This approach significantly saves routing resources and supports higher data throughput, especially in SoC (System on Chip) and high-speed interface designs such as PCIe, USB, SATA, and Ethernet. SERDES blocks are commonly integrated into PHYs and transceivers in modern chipsets and are known for their ability to maintain signal integrity across long distances and multiple clock domains.

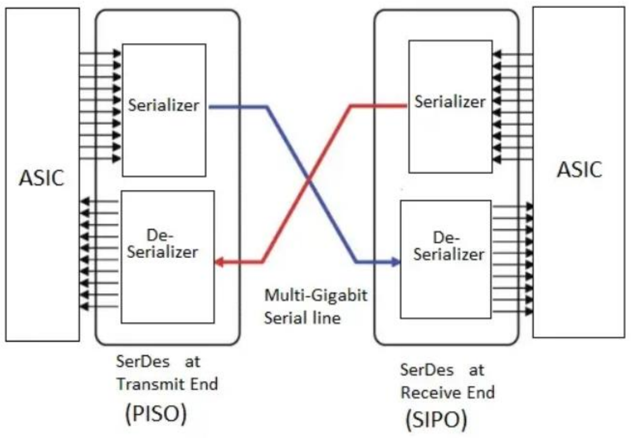


Figure 1.1. Serializing data in transmitter and transmitting to a desserializer in the receiver side

Need for Synchronizers

In systems using multiple clock domains (a common scenario in designs with SERDES), **clock domain crossing (CDC)** becomes a critical issue. When data is transferred between two domains that do not share a common clock or have asynchronous relationships, **metastability** may occur — a condition where flip-flops fail to resolve to a valid logic level, leading to erroneous behavior. To address this, synchronizers are used to safely transfer signals from one clock domain to another, minimizing the risk of metastability and ensuring robust data communication.

Types of Synchronizers

Synchronizers are broadly classified into two types:

1. **Single-bit Synchronizers:** Used for control signals or flags, these typically consist of a chain of flip-flops (usually two or more) clocked by the destination clock to absorb any metastability.
2. **Multi-bit Synchronizers or FIFO-based Synchronizers:** Used for bulk data transfer, especially when dealing with buses or parallel data. These utilize handshake-based schemes or asynchronous FIFO buffers to ensure data integrity.

Each type is chosen based on data width, clock relationship (asynchronous or mesochronous), and timing constraints.

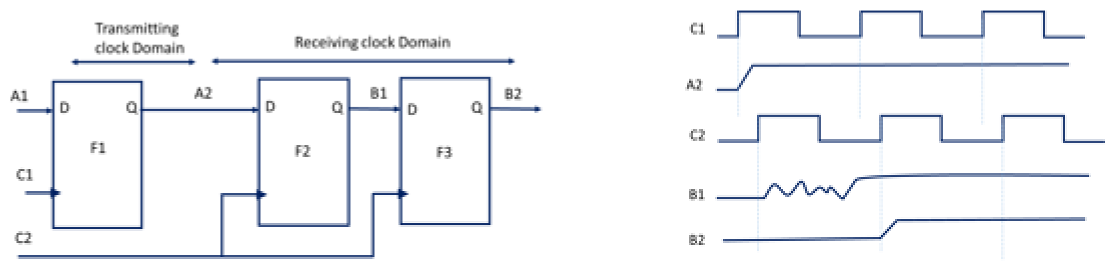


Figure 1.2. Synchronizing the transmitted data to receiver clock domain using flops

3-Stage Synchronizers

While the standard synchronization mechanism uses **two flip-flops**, a **3-stage synchronizer** may be introduced in high-reliability systems where signal stability is critical. The three-stage structure further reduces the probability of metastability propagating into functional logic. It consists of three sequential flip-flops clocked by the receiving domain. The first stage captures the incoming asynchronous signal, the second stage helps resolve metastability if it occurs, and the third stage ensures stable output for downstream logic.

The trade-off is slightly increased latency, but it ensures significantly improved reliability, especially for critical control paths in SERDES interfaces where even a single incorrect transition can lead to major failures in communication.

Overview of Pseudo-Random Generators and LFSR Selection

In digital design and testing, **pseudo-random pattern generators (PRPGs)** play a crucial role in providing test vectors to stimulate logic in an efficient, repeatable, and varied manner. These generators produce sequences that appear random but are deterministically generated.

One of the most widely used PRPGs in VLSI systems is the **Linear Feedback Shift Register (LFSR)**. LFSRs use a series of flip-flops with XOR feedback logic to create a sequence of bits that cycle through a long period before repeating. The length of the sequence depends on the tap polynomial used and the number of bits in the register. LFSRs[11] are especially valuable in built-in self-test (BIST) mechanisms, communication systems, due to hardware simplicity

Several methods exist for generating pseudo-random sequences. These include **M-sequences**, **cellular automata**, **XOR-shift algorithms**, and **cryptographic pseudo-random generators**. Each method has trade-offs in terms of randomness quality, hardware complexity, and application fit. The table below offers a quick comparison of these techniques:

- **LFSR:** Offers low hardware complexity and is highly efficient in VLSI applications. It produces deterministic, repeatable sequences ideal for hardware test patterns.
- **M-Sequence:** Similar to LFSRs but derived from maximal-length feedback configurations. It offers high randomness but requires slightly more logic.
- **Cellular Automata:** Utilizes rules-based parallel logic to evolve sequences but is less common in traditional VLSI designs.
- **XOR-Shift:** Extremely simple and fast, but offers lower entropy, making it more suitable for software than hardware.

1.3 Objectives, Scope, and Thesis Organization

Objectives

The primary objective of this thesis is to explore, implement, and optimize efficient testing and debugging techniques in high-speed VLSI designs using a combination of pseudo-random pattern generation, synchronizer design[12], and firmware scripting methodologies. Specifically, the goals are:

- To design and implement a robust Pseudo-Random Bit Sequence (PRBS) generator and pattern matcher[13] based on Linear Feedback Shift Registers (LFSRs), suitable for high-speed digital testing environments.
- To utilize SERDES (Serializer/Deserializer) interfaces and synchronizer circuits for clock domain crossing and signal reliability in multi-clock systems.

Table 1. Comparison of different types of Pseudo-Random Generators

Generator Type	Hardware Complexity	Randomness Quality	Reproducibility	Use in Hardware	Remarks
LFSR	Low	High (deterministic)	Yes	Excellent	Simple XOR & Shift operations
M-Sequence	Medium	Very High	Yes	Moderate	Maximal-length sequence, similar to LFSR
Cellular Automata	Medium to High	High	Yes	Limited	Parallel logic, less common in VLSI
XOR-Shift	Low	Moderate	Yes	Limited	Mostly used in software, fast but lower entropy
Cryptographic PRNG	Very High	Very High	Yes	Poor	Not used in hardware due to area and latency

- To implement a multi-stage synchronizer mechanism and analyze its stability in avoiding metastability errors.
- To perform functional verification and lint analysis using industry-standard tools such as Spyglass for Lint checks and Verdi for signal tracing and waveform debugging.
- To simplify and optimize repetitive assembly-level firmware scripts by converting them into readable macro-instruction sets, thus improving maintainability and reducing error-prone manual coding.

Scope

This thesis covers several domains within digital design and verification:

- The design and simulation of LFSR-based PRBS[5] generators for creating deterministic yet pseudo-random patterns used in built-in self-test (BIST[5]) and communication link validation.
- Synchronization techniques using 2-stage and 3-stage synchronizers for safe signal transfer across asynchronous clock domains, critical for high-speed and low-latency systems.
- Evaluation of multiple pseudo-random sequence generators (e.g., M-sequences, XOR-shift, cellular automata) and a justification for choosing LFSRs based on hardware efficiency and simplicity.
- Scripting enhancements using Perl or equivalent scripting tools to convert low-level assembly code into macro-instruction sets that increase readability and debugging speed.
- Debugging and linting using Verdi and Spyglass, respectively, to validate design correctness and eliminate syntax errors early in the development cycle.

CHAPTER 2

LITERATURE REVIEW

2.1 Fundamentals of PRBS, LFSRs, and Synchronization

The design of efficient pseudo-random binary sequence (PRBS) generators for high-speed communication systems requires careful consideration of power consumption, timing synchronization, and throughput optimization. Several seminal works in the field of VLSI design and integrated circuits provide valuable insights that directly informed the architectural decisions in this project.

Chen and Yang (2012) presented a **low-power, highly multiplexed parallel PRBS generator** at the IEEE Custom Integrated Circuits Conference, which demonstrated the advantages of using parallel LFSR structures to achieve high-speed pattern generation while minimizing power consumption. Their key innovation was the use of a multiplexed architecture that allows multiple LFSR outputs to be combined, significantly increasing throughput without a proportional increase in power usage. This work inspired the multi-mode LFSR implementation in our system, where configurable polynomial sizes (e.g., 31-, 23-, and 16-bit) enable flexible operation across different testing scenarios. By adopting a similar parallel approach, our design achieves high-speed PRBS generation suitable for SONET and other optical communication standards, while maintaining low power overhead.

Power efficiency is a critical concern in modern VLSI systems, and the work by Wu, Pedram, and Wu on **clock-gating techniques** provides a robust framework for reducing dynamic power consumption. Published in the *IEEE Transactions on Circuits and Systems I*, their research highlights how selectively disabling clock signals to inactive circuit modules can lead to significant energy savings. Their methodology was particularly influential in guiding the clock-gating implementation for our LFSR and serializer modules. By incorporating clock-gating controls, our system dynamically shuts off clock signals to idle components, reducing power wastage without affecting performance. This approach aligns with the broader goal of energy-efficient design, ensuring that the PRBS generator meets stringent power constraints in embedded and high-performance applications.

The challenge of maintaining signal integrity across clock domains is addressed in the work by Fisher and Kung, "Synchronizing Large VLSI Processor Arrays," published in the *IEEE Transactions on Computers*. Their study underscores the importance of robust synchronization mechanisms to prevent metastability in systems with multiple clock domains. They advocate for multi-stage synchronizers to mitigate the risks of timing violations, a recommendation that directly influenced our adoption of a **3-stage synchronizer** for clock domain crossing (CDC) in the PRBS system. This design choice ensures reliable data transmission between the transmitter and receiver, even under high-speed operating conditions. The synchronizer effectively handles potential metastability issues, enhancing system stability and reducing error rates during pattern generation and matching.

Together, these works form the theoretical and practical foundation for our PRBS generator, addressing the tripartite goals of speed, power efficiency, and reliability. Chen and Yang's parallel architecture enables high-throughput pattern generation, Wu et al.'s clock-gating techniques minimize energy consumption, and Fisher and Kung's synchronization strategies ensure robust operation across clock domains. By integrating these insights, our system achieves a balanced optimization of performance and power, making it suitable for a wide range of applications, from optical communication testing to VLSI verification.

The synthesis of these studies not only validates the design choices made in this project but also highlights the importance of interdisciplinary research in advancing VLSI systems. Future work could explore further optimizations, such as adaptive clock-gating policies or advanced error-correction techniques, to push the boundaries of PRBS generator performance.

Pseudo-Random Binary Sequences (PRBS) are essential in the testing of high-speed digital systems due to their ability to simulate real-world data with high entropy and repeatability. These sequences, while appearing random, are deterministically generated using simple mathematical rules, enabling consistent validation and debug of integrated circuits. The most commonly used hardware technique to generate PRBS is the Linear Feedback Shift Register (LFSR), which is a sequence of flip-flops combined with feedback logic that cycles through a pseudo-random sequence of bits.

LFSRs operate on a defined seed and polynomial function. For instance, a polynomial like x^7 + x^6 + 1 indicates that the 6th and 7th bits are XORed and fed back to the first flip-flop. This mechanism ensures the generation of long and repeatable sequences. Various polynomials such as those of degrees 7, 9, 11, 13, 15, 16, 23, and 31 are used depending on the application. For example, the 11, 13, and 15 polynomials are typically used in jitter measurements, whereas the 31-bit polynomial is suitable for stress testing.

Pattern Generators are designed using modular blocks like the LFSR generator, control register, and error injection units. A typical design includes:

- **LFSR Gen**: Converts seed to output bits based on polynomial
- **Error Write**: Intentionally injects errors (e.g., flipping LSB)
- **Control Register**: Stores pattern settings and modes
- **Clk Gating and Synchronization**: Used for efficient clock handling

Fixed and DC balanced patterns are also used, with 10-bit reserved fixed words for transmission. The use of DC balanced words (~PAT0) helps ensure equal distribution of 1s and 0s, critical in maintaining signal integrity during high-speed transmission.

The generated pattern is transmitted and later checked using a Pattern Matcher that compares the incoming data with previously known PRBS[10] values. The matcher supports both bit-wise and byte-wise error detection, allowing identification of corrupted bits during transmission. Modes such as 10/8 or full/half-bit checking are supported.

Synchronization, particularly in multi-clock systems, is crucial. Signals moving from one clock domain to another can encounter metastability if not managed properly. This is typically handled using 2-stage or 3-stage synchronizers. In a 2-stage synchronizer, the first stage may go metastable, but the second stage stabilizes the signal. Multiple stage synchronizers increase the Mean Time Between Failures (MTBF) and reduce metastability risks. Quasi-static level signals are another concept, where signals remain at a certain level long enough to be safely captured.

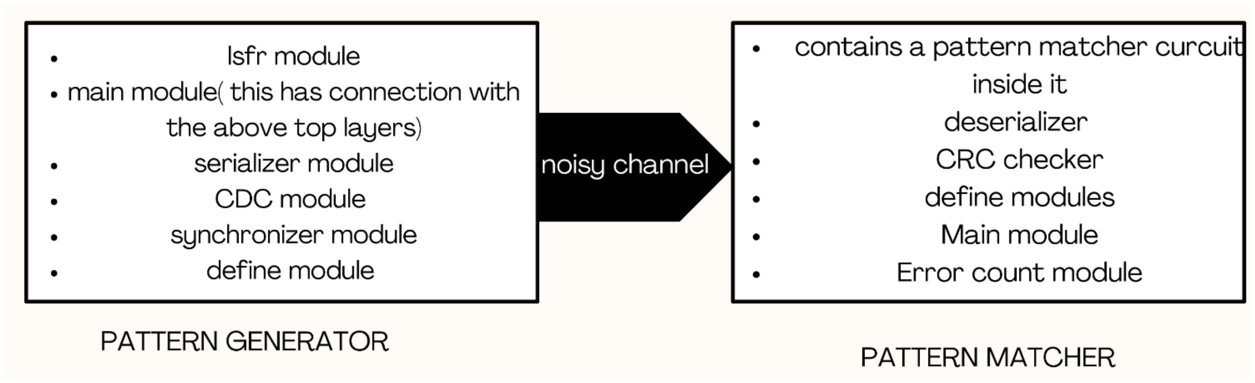


Figure 2.1 . This shows the modules present in Pattern Generator and Pattern Matcher

2.2 SERDES Testing, Clock Gating[16], and CDC Handling

Serializer/Deserializer (SERDES) systems are foundational for high-speed serial communication in modern VLSI systems. They convert parallel data to serial form for transmission and revert it to parallel upon reception. Ensuring reliability in such systems requires rigorous testing, which involves PRBS patterns, error detection logic, and synchronization mechanisms.

Scan chains are a crucial Design-for-Test (DFT) technique used in internal testing. They involve connecting flip-flops in a shift-register fashion, enabling the insertion of test vectors and the observation of output values. Flip-flop-based techniques like muxed-D scan and clocked scan are common, while latch-based approaches include scan shift devices (SSD). DFT insertion introduces scan inputs (SI) and scan outputs (SO) for this purpose. In normal mode, scan flip-flops behave like regular flip-flops, but in test mode, they form a shift chain.

BIST[23] (Built-In Self-Test) techniques enhance fault coverage using on-chip test pattern generators like LFSRs1, cellular automata, and accumulators driven by constant values. Weighted inputs are used to increase pattern diversity while minimizing hardware costs. A notable optimization is the use of Multiple Input, Single Change (MISC) sequences, where only one bit changes per transition, reducing power consumption and logic switching.

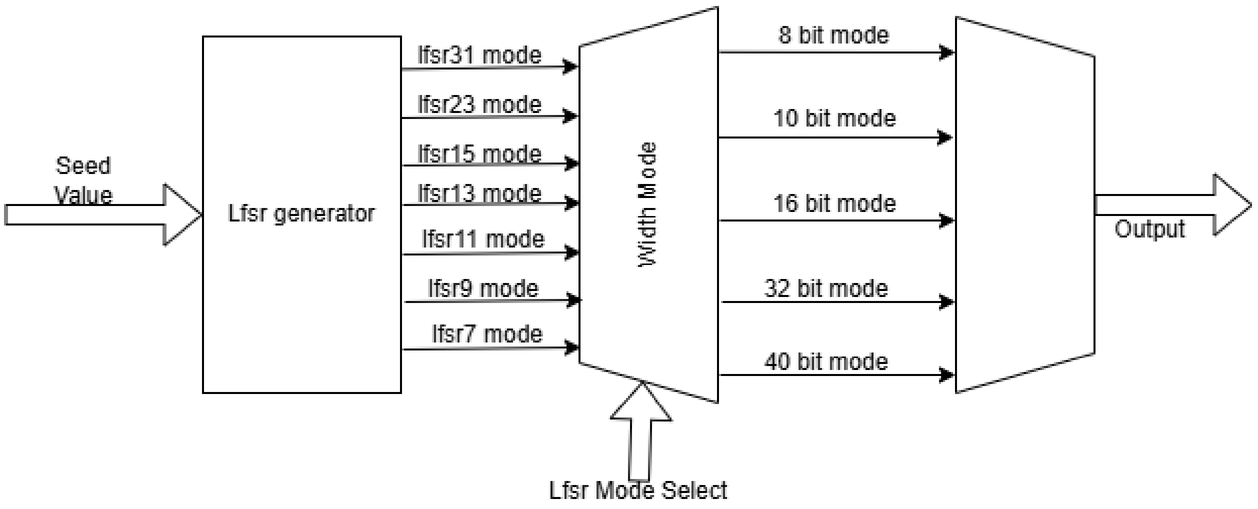


Figure 2.2 . Shows the working architecture of Pattern Generator using LFSR

- **Using Synchronizers**: 2-stage or 3-stage synchronizers are introduced at the boundary
- **Standard Cell Synchronizers**: Ensure placement proximity to minimize timing differences
- **Clock Gating**: Disables clocks when not needed to prevent unnecessary transitions, improving power efficiency

Byte Error Rate Testers (BERTs) such as L-BERT integrate LFSR-based PRBS generation and checking logic. They include mode selection (e.g., 8, 16, 20, 32, or 40-bit widths), fixed and user-defined pattern insertion, and options for DC-balanced word transmission. The seed used for LFSR is deterministic, ensuring repeatable PRBS generation.

Modes are crucial in determining pattern generation logic:

- **Mode 12**: 64-bit user-defined
- **Mode 13**: LFSR with 13-degree polynomial
- **PAT Selection Register**: Controls the mode and stores polynomial/fixed/DC-balanced patterns

Error checking uses matchers with configurations such as:

- **Pat_count**: Tracks number of generated patterns
- **Lbert_ec**: Error counter (bit-wise or byte-wise)
- **Clk_gate**: Maintains clock control
- **Link Register**: Indicates status of the link (ON/OFF)

Pattern matchers compare d[n] with d[n-10] and use tree-based XOR logic to highlight errors. Registers used are lane-independent, allowing parallel checks across different data channels.

In conclusion, integrating robust PRBS generation, advanced scan techniques, and precise CDC handling significantly enhances the fault detection capability and data integrity in modern VLSI designs. The use of LFSR-based designs, synchronized scan chains, and BIST frameworks ensures scalable, efficient testing frameworks.

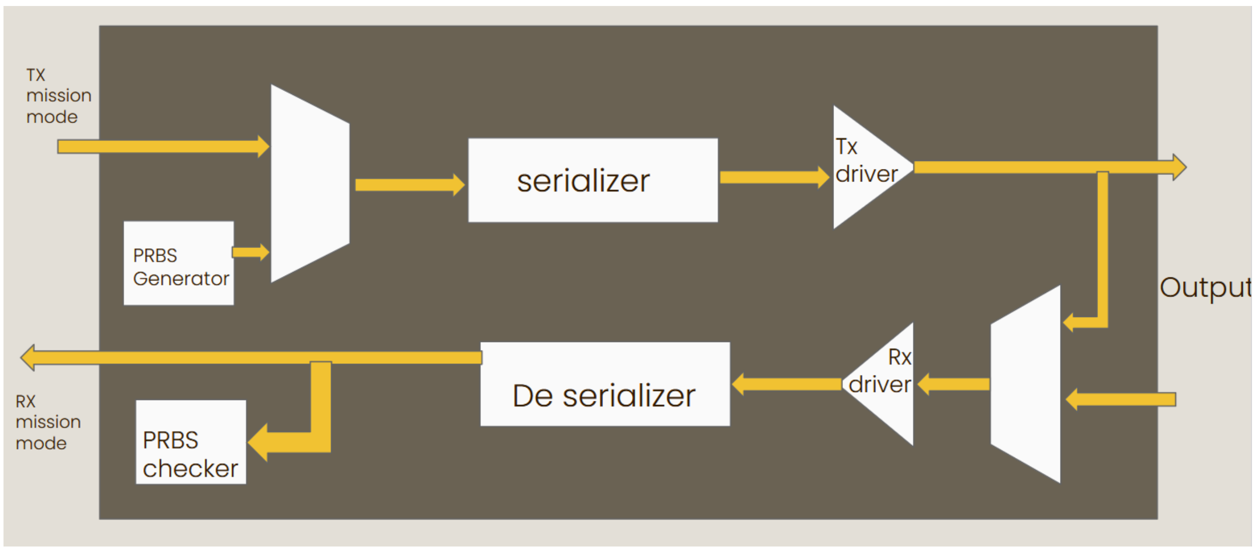


Figure 2.3. Overall function of SERDES with Pattern Generator

CHAPTER 3

METHODOLOGY

3.1 System Architecture with LFSR and Synchronizer Design

This thesis proposes a methodology to simulate and validate high-speed digital systems using an LFSR-driven architecture for pattern generation and serialization. The system architecture[6] is built on modular and deterministic components capable of emulating realistic data streams required in hardware verification. The design encompasses three primary stages: pattern generation using a Linear Feedback Shift Register (LFSR), data serialization for transmission, and pattern extraction for analysis.

LFSR-Based Pseudo-Random Pattern Generator

The heart of the architecture[15] is a 40-bit Linear Feedback Shift Register (LFSR) configured with a characteristic polynomial (e.g x^7 + x^6+1). This polynomial ensures that feedback is generated through XOR operations on the 6th and 7th bits of the shift register. The result of this feedback is shifted into the register to produce a pseudo-random binary sequence (PRBS).

To initialize the sequence, a fixed seed value is preloaded into the register. This allows deterministic pattern reproduction, which is essential for repeatable testing. The system uses a control mechanism to ensure the LFSR is seeded correctly before it begins cycling through the pseudo-random values. A trail counter governs the seeding and pattern propagation phases. This logic ensures that the PRBS starts from a known state and that the generated pattern covers sufficient entropy to simulate real-world high-speed[7] data behavior.

The LFSR can be reconfigured to accommodate different polynomials depending on the testing requirement, such as stress testing, jitter measurements, or SONET compliance. The use of a 40-bit width allows for a long periodic sequence, increasing the test coverage and reducing the likelihood of repeating patterns in short timeframes.

Table 2. Shows the different polynomials used in LFSR

Mode	Polynomial/Type	Description	Application
7	$x7+x6+1$	XOR 6th and 7th bits; seed starts with all 1s.	Error testing (low complexity)
9	$x9+x5+1$	Fixed 10-bit word transmission.	Error testing
11	$x11+x9+1$	Jitter measurement.	High-frequency jitter analysis
13	$x13+x4+x3+1$	Dedicated 13-bit polynomial.	Jitter/stress testing
15	$x15+x14+1$	Jitter measurement.	Signal integrity validation
16	$x16+x15+x4+1$	DC-balanced word generation (negate PAT value).	SONET/SDH compliance SONET/SDH compliance
23	$x23+x18+1$	Standard for SONET testing.	SONET frame synchronization
31	$x31+x28+1$	Long-sequence stress testing.	High-stress reliability testing

Clock-Gated Serialization Pipeline

After PRBS generation, the next stage involves serializing the 40-bit parallel data into a high-speed serial stream. This is critical in replicating the behavior of SERDES-based communication systems, where data is transmitted bit-by-bit over a shared medium.

The serialization process begins by generating a divided clock signal from the main system clock. This is achieved using a simple counter-based clock division technique. The divided clock ensures that the serialized data rate is well-controlled and matches the downstream system requirements. Once the clock signal is ready, the PRBS data is loaded into a shift register, and the bits are transmitted one at a time on each clock cycle.

The serializer operates in a loop, continuously updating the shift register with new pseudo-random words and streaming them serially. This method provides a consistent and controllable test stream that mimics real communication traffic. Moreover, the serialization logic is implemented in a manner that separates control and data logic, improving modularity and simplifying verification.

.

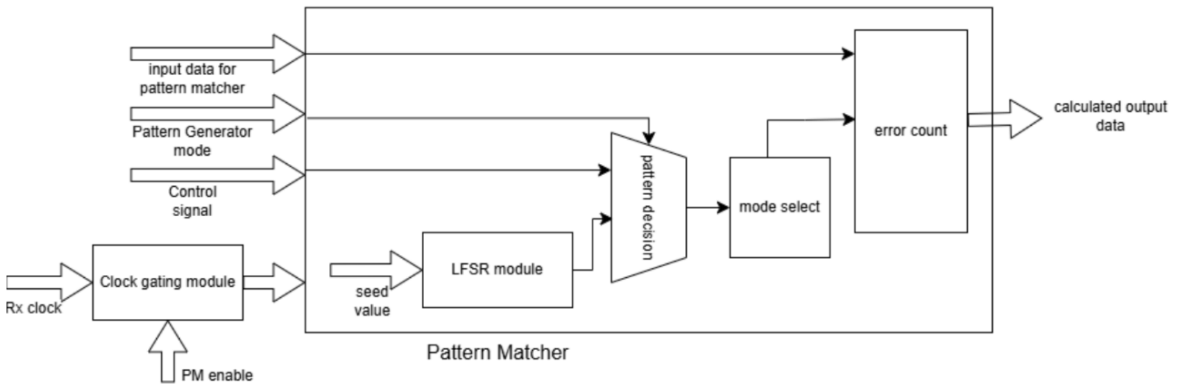


Figure 3.1. Architecture of pattern matcher

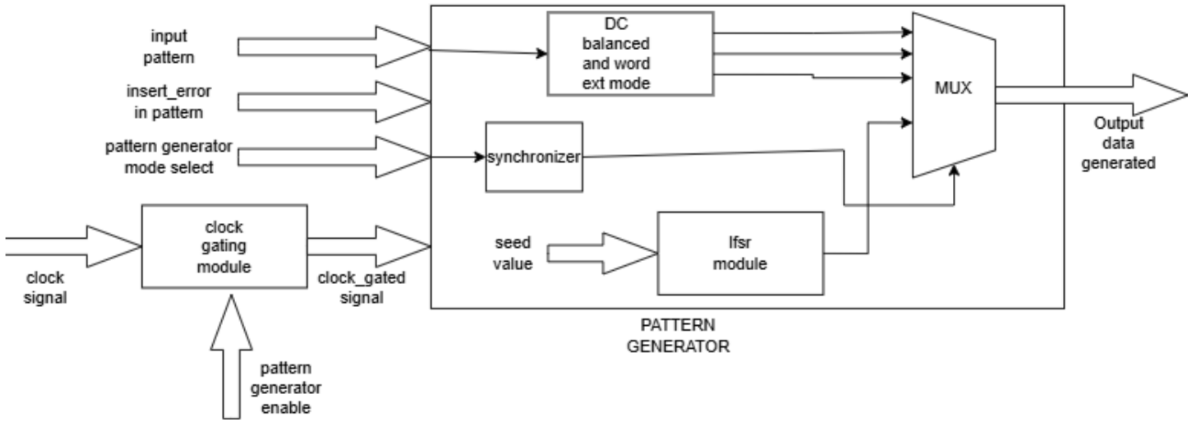


Figure 3.2. Architecture for pattern generator

Pattern Extraction and Word Selection

An additional feature of the system is the ability to extract and select specific segments of the PRBS stream. This is handled through a pattern extraction module, which holds a 64-bit PRBS word and segments it into multiple 16-bit sub-patterns. Depending on a control register (pattern count), one of the four sub-patterns is routed to the output.

This modular segmentation provides two distinct benefits:

- It facilitates easy inspection and debugging of shorter pattern sequences.
- It enables different modules or subsystems to receive tailored test patterns without rerunning the full PRBS cycle.

The selection logic is implemented using a multiplexer-like structure controlled by a two-bit pattern counter. As the system progresses, this counter cycles through the different segments, ensuring a dynamic and varying test stream across time. This enhances the comprehensiveness of the pattern coverage without increasing the memory footprint or the need for external data storage.

```
Initialize:
    toggle_pattern ← 16-bit constant
    pg_pattern ← 64-bit register
    pat_count ← 1

On every cr_clk edge:
    Depending on pat_count value:
        00 → toggle_pattern ← bits 63 to 48 of pg_pattern
        01 → toggle_pattern ← bits 47 to 32 of pg_pattern
        10 → toggle_pattern ← bits 31 to 16 of pg_pattern
        11 → toggle_pattern ← bits 15 to 0 of pg_pattern
```

Figure 3.3 . Pseudo Code for Pattern Extraction

This segmentation feature is particularly useful in simulation environments where full-width PRBS data may be excessive for localized testing. It also mirrors how real-world high-speed interfaces often operate with different bus widths, such as 8-bit or 16-bit channels.

Table 3. Shows the different Data width used in Serialization and Deserialization

Speed Mode	**Data Width (bits)**	**Description**	**Use Case**
8-bit	8	Processes 8-bit chunks; lowest throughput.	Low-speed interface testing
16-bit	16	Balances speed and resource usage.	Mid-range applications
20-bit	20	Optimized for specific protocol requirements.	Custom serial links
32-bit	32	High-throughput mode[14] for wide buses.	Parallel data verification
40-bit	40	Maximum width for stress testing (e.g., SONET).	High-speed optical communication

System-Level Integration and Modularity

The complete system is designed with modularity in mind. Each block — LFSR generation, serialization, and pattern extraction — can be developed, tested, and reused independently. This modular approach aligns with modern digital design practices and enables rapid integration into different simulation or verification environments.

By using deterministic seed values and known polynomial configurations, this architecture allows for reproducible results — a key requirement in any design verification process. Furthermore, the simplicity of the design enables low-resource implementation, making it suitable for both FPGA prototyping and ASIC testbench deployment.

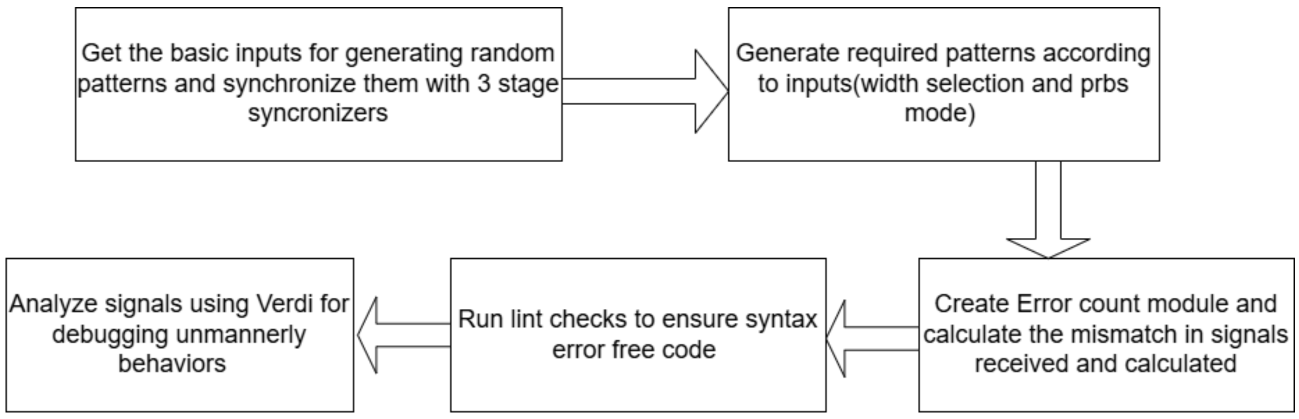
The methodology focuses on delivering testability without introducing excessive overhead or complexity. By emphasizing lightweight control mechanisms and well-structured datapaths, this architecture is scalable, flexible, and easy to adapt for extended verification frameworks or higher bit-width expansions.

3.2 PRBS-Based Pattern Generator and Matcher

This part of the project focuses on implementing a reliable pattern generation and matching system using a Pseudo-Random Binary Sequence (PRBS) based on Linear Feedback Shift Registers (LFSRs). The aim is to synchronize control signals from various modules, use LFSRs to generate predictable test patterns, and validate the correctness of data transmission using a pattern matcher.

Steps followed:

- Signals like clock, enable, and seed input were first synchronized using two-stage flip-flop synchronizers to avoid metastability during domain crossing.
- A configurable LFSR module was created to generate PRBS patterns of desired lengths (e.g., PRBS-7, PRBS-15) using polynomial logic and a dynamic seed.
- The pattern generator used the LFSR output to transmit test sequences.
- A pattern matcher module was developed, which recreated the expected sequence using the same LFSR logic and compared it with the incoming pattern to detect mismatches.
- Linting tools like Spyglass and Verilator were used to detect syntax errors, combinational loops, and width mismatches early in the flow.
- Verdi was used to visualize the behavior of the LFSR and other critical control signals. This helped in debugging the sequence and validating the synchronizer output and data alignment.


Get the basic inputs for generating random patterns and synchronize them with 3 stage syncronizers
Generate required patterns according to inputs(width selection and prbs mode)
Create Error count module and calculate the mismatch in signals received and calculated
Run lint checks to ensure syntax error free code
Analyze signals using Verdi for debugging unmannerly behaviors

.

Figure 3.4. Shows the methodology of implementing pattern generator and pattern matcher using linear feedback shift register

3.3 Instruction Set Optimization for Firmware Simplification

The second part of the project focused on identifying patterns in repetitive firmware instructions and optimizing[8] them using macros. This was done to reduce redundancy, improve readability, and minimize manual coding errors in pattern generation and matching scripts.

Steps followed:

- Repetitive instruction sequences in the firmware (like multiple lines to write to registers, perform conditional jumps, polling waits, and loops) were identified and grouped logically.
- A set of macro instructions were defined to represent these grouped operations, such as MACROS_WRITE, MACROS_READ, MACROS_JMP_NOT_ZERO, and MACROS_MUL, among others.
- These macro instructions significantly simplified the scripts by replacing 3–6 lines of assembly-level code with a single readable line.
- Code was cleaned up by introducing new functions that internally used these macros, resulting in modular and compact firmware scripts.
- Each macro was tested against the original instruction flow to verify that it retained the correct functionality and behavior.
- Verdi was again used to verify and debug signal flow during macro execution, ensuring that abstraction didn’t impact test logic.
- A comparison between original and optimized scripts showed significant line reduction and better maintainability.
- This optimization also improved productivity, as engineers could now use predefined macros instead of rewriting complex instruction sequences repeatedly.

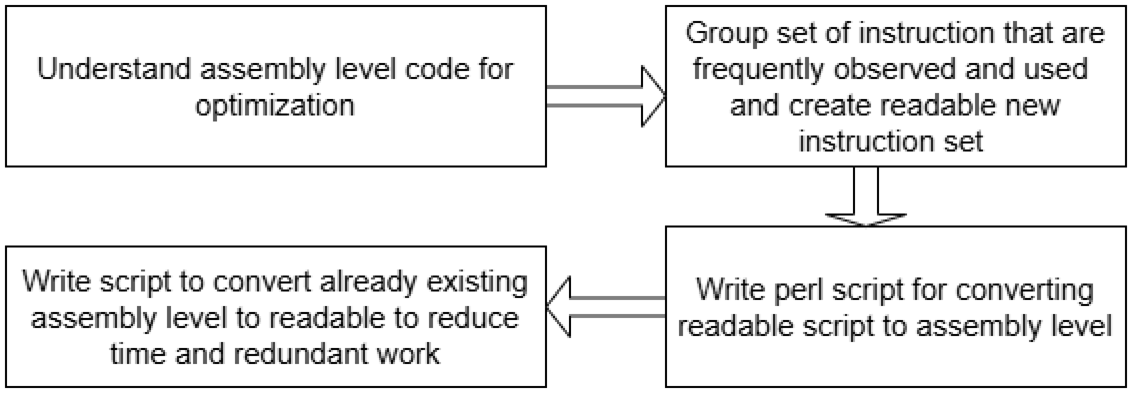
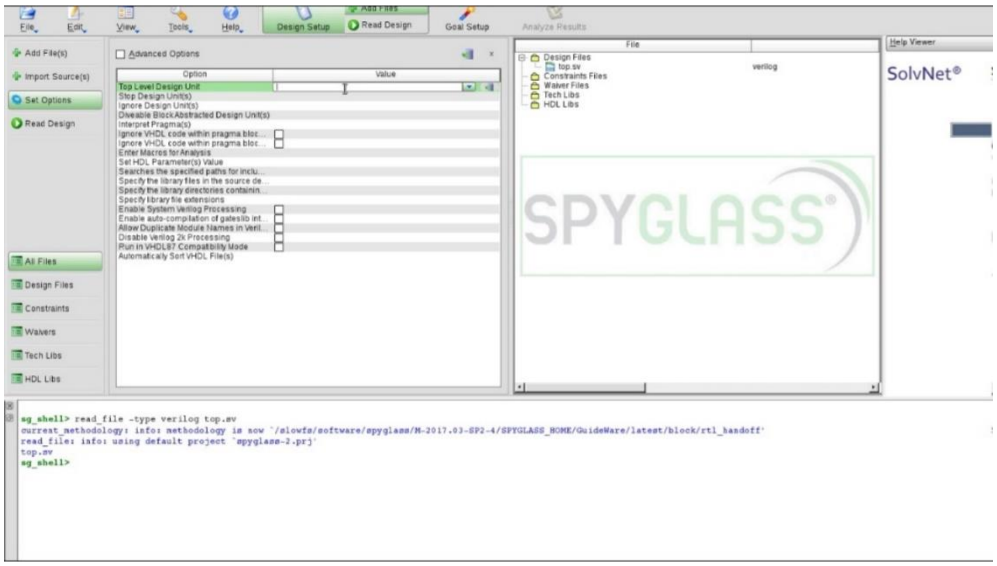

Understand assembly level code for optimization
Group set of instruction that are frequently observed and used and create readable new instruction set
Write perl script for converting readable script to assembly level
Write script to convert already existing assembly level to readable to reduce time and redundant work

Figure 3.5. Methadology for implementation of new instruction sets in assembly language

3.4 Signal Debugging with Verdi and Lint Check Integration

This section describes the methodology adopted for signal debugging and design verification through the integration of Verdi and static linting tools. In high-speed digital systems, accurate visibility into signal behavior is crucial for early-stage bug detection and functional verification. The methodology focuses on enhancing traceability and design correctness by leveraging automated debug environments and rule-based code analysis, independent of clock domain verifition.

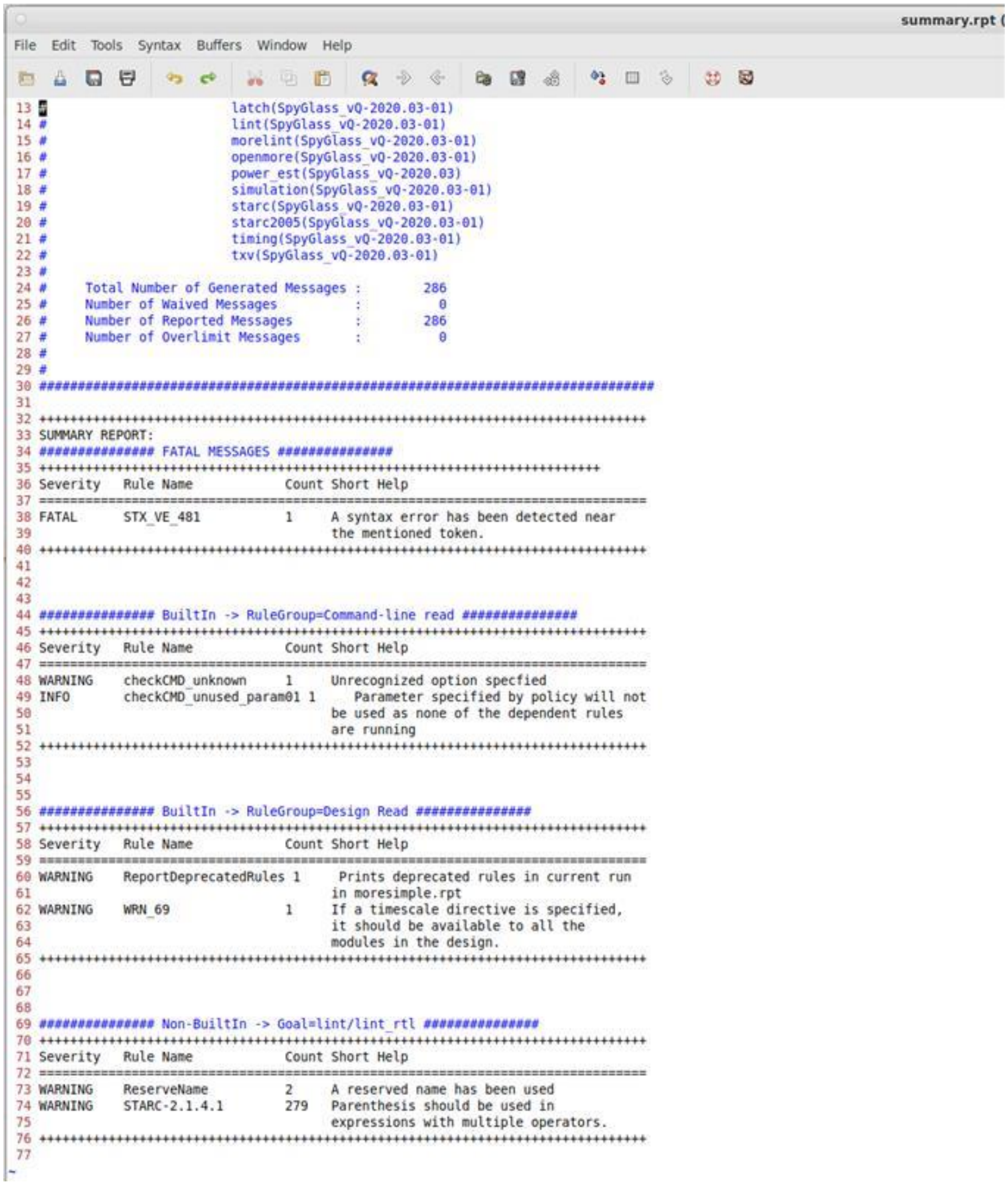
The Verdi debug tool is employed to perform waveform analysis and track signal transitions during simulation. It enables hierarchical exploration of the design, helping the developer pinpoint unexpected behaviors and root causes. During simulation, signals of interest—such as data paths from the LFSR, serialization outputs, and toggle patterns—are traced using Verdi's signal annotation and navigation features. Bookmarks and signal grouping allow for efficient correlation of waveforms across modules, improving the debugging workflow.



```
sg_shell> read_file -type verilog top.sv
current_methodology: info: methodology is now '/slowfs/software/spyglass/M-2017.03-SP2-4/SPYGLASS_HOME/GuideWare/latest/block/rtl_handoff'
read_file: info: using default project `spyglass-2.prj'
top.sv
sg_shell>
```

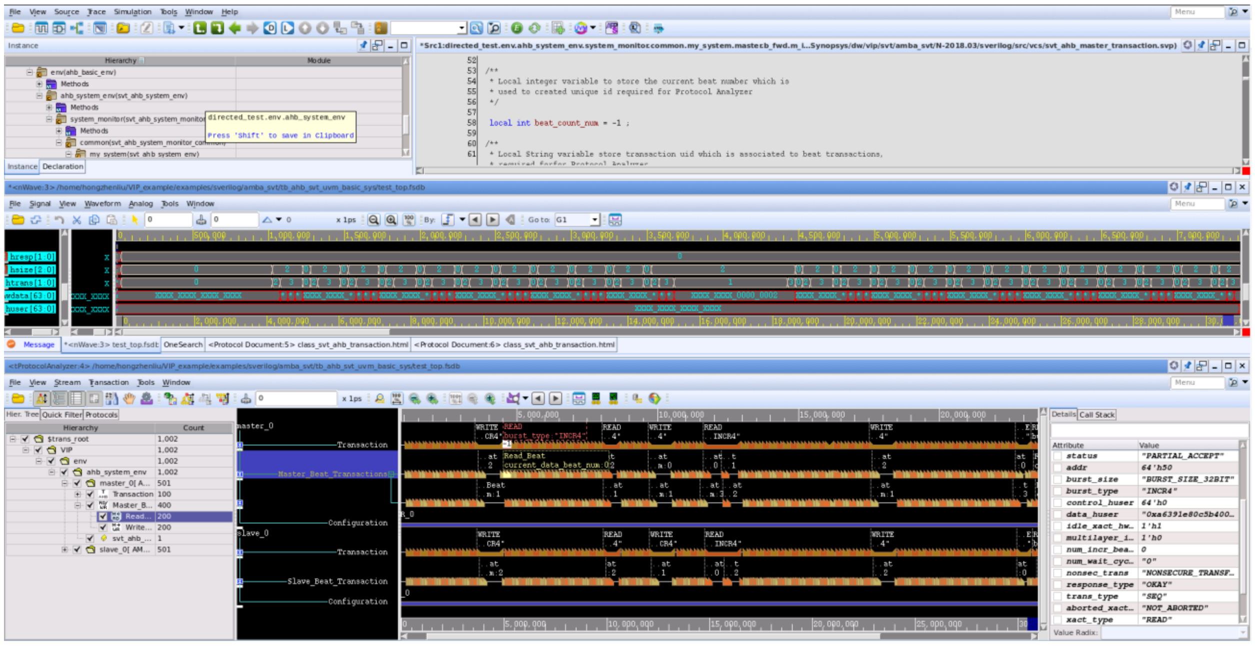
Figure 3.6. User Interface of Spyglass linting

To ensure the design adheres to best coding practices, a static linting tool is incorporated into the design flow. The lint tool runs before RTL simulation, scanning the code for syntactic and stylistic inconsistencies. Typical checks include unused signals, incorrect blocking vs. non blocking assignments, incomplete case statements, and race condition risks. By resolving these warnings early, the design is maintained in a cleaner and more maintainable state. This lint-verdi integration helps in establishing a verification-ready environment by minimizing human errors and enhancing observability. The process begins with a full RTL parse using the lint tool, followed by simulation using testbenches. As errors or mismatches are detected in the waveform, Verdi is used to visually trace the source module, signal behavior, and timing violations. Fixes are iteratively applied and re-validated both via lint passes and waveform re-analysis.



```
#                       latch(SpyGlass_vQ-2020.03-01)
#                       lint(SpyGlass_vQ-2020.03-01)
#                       morelint(SpyGlass_vQ-2020.03-01)
#                       openmore(SpyGlass_vQ-2020.03-01)
#                       power_est(SpyGlass_vQ-2020.03)
#                       simulation(SpyGlass_vQ-2020.03-01)
#                       starc(SpyGlass_vQ-2020.03-01)
#                       starc2005(SpyGlass_vQ-2020.03-01)
#                       timing(SpyGlass_vQ-2020.03-01)
#                       txv(SpyGlass_vQ-2020.03-01)
#
#     Total Number of Generated Messages :       286
#     Number of Waived Messages          :         0
#     Number of Reported Messages        :       286
#     Number of Overlimit Messages       :         0
#
#
################################################################################

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
SUMMARY REPORT:
############### FATAL MESSAGES ###############
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Severity   Rule Name            Count Short Help
===============================================================================
FATAL      STX_VE_481           1     A syntax error has been detected near
                                      the mentioned token.
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++



############### BuiltIn -> RuleGroup=Command-line read ###############
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Severity   Rule Name            Count Short Help
===============================================================================
WARNING    checkCMD_unknown     1     Unrecognized option specfied
INFO       checkCMD_unused_param01 1     Parameter specified by policy will not
                                      be used as none of the dependent rules
                                      are running
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++



############### BuiltIn -> RuleGroup=Design Read ###############
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Severity   Rule Name            Count Short Help
===============================================================================
WARNING    ReportDeprecatedRules 1     Prints deprecated rules in current run
                                      in moresimple.rpt
WARNING    WRN_69               1     If a timescale directive is specified,
                                      it should be available to all the
                                      modules in the design.
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++



############### Non-BuiltIn -> Goal=lint/lint_rtl ###############
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Severity   Rule Name            Count Short Help
===============================================================================
WARNING    ReserveName          2     A reserved name has been used
WARNING    STARC-2.1.4.1        279   Parenthesis should be used in
                                      expressions with multiple operators.
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

~
```

Figure 3.7. Generated error log file using Spyglass



*Src1:directed_test.env.ahb_system_env.system_monitor.common.my_system.master.b_fwd.m_i...Synopsys/dw/vip/svt/amba_svt/N-2018.03/sverilog/src/vcs/svt_ahb_master_transaction.svp)

```

/**
 * Local integer variable to store the current beat number which is
 * used to created unique id required for Protocol Analyzer
 */

 local int beat_count_num = -1 ;

/**
 * Local String variable store transaction uid which is associated to beat transactions,
```

Hierarchy	Count
$trans_root	1,002
VIP	1,002
env	1,002
ahb_system_env	1,002
master_0[A...	501
Transaction	100
Master_B...	400
Read...	200
Write...	200
svt_ahb_...	1
slave_0[AM...	501

Attribute	Value
status	"PARTIAL_ACCEPT"
addr	64'h50
burst_size	"BURST_SIZE_32BIT"
burst_type	"INCR4"
control_huser	64'h0
data_huser	"0xa6391e80c5b400...
idle_xact_hw...	1'h1
multilayer_i...	1'h0
num_incr_bea...	0
num_wait_cyc...	"0"
nonsec_trans	"NONSECURE_TRANSF...
response_type	"OKAY"
trans_type	"SEQ"
aborted_xact...	"NOT_ABORTED"
xact_type	"READ"

Figure 3.8. Shows the user interface of Verdi tool

In this methodology, focus is placed on the visualization and structural correctness of the design, ensuring that all internal signals behave as expected under various input conditions. This integration supports efficient root cause analysis, significantly reducing debug time and improving overall design confidence.

CHAPTER 4

SCRIPTING AND INSTRUCTION SET OPTIMIZATION

4.1 Analysis and Grouping of Firmware Instructions

This section outlines the methodology used to analyze and categorize firmware instructions into logical groups[19], forming the foundation for instruction set optimization in embedded systems. The primary objective is to streamline the instruction decoding and interpretation process, allowing for better control logic design, simplified[21] hardware-software interaction, and enhanced debugging efficiency.

Instruction Stream Analysis

The firmware under investigation consists of a mix of low-level assembly instructions that control memory operations, register manipulations, conditional branching, arithmetic operations, and control flow management. A com

prehensive review of the firmware instruction set was carried out by tracing the recurring usage patterns and identifying clusters of related operations. Each instruction was carefully analyzed to understand its purpose, operand dependencies, and side effects.

The instructions were then grouped into six high-level functional categories:

1. **Write Operations**
2. **Read Operations**
3. **Conditional Jump Operations**
4. **Wait Logic**
5. **Arithmetic Operations (Multiplication)**
6. **Control Flow Constructs (Case Statements)**

This categorization supports the development of a systematic instruction decoder and optimization logic, where related sequences can be bundled and optimized either in software execution or hardware control.

Instruction Group 1: Write Operations

This group includes all instructions that modify memory-mapped registers or internal states. Instructions such as store_addr write_mask value and store_address register_address value are used to configure target addresses and values. Additionally, operations like copy_register reset_mask write_mask help set internal masks or trigger specific write behaviors.

These are typically used in configuration or initialization stages and often follow a fixed pattern. Grouping them allows the creation of a reusable scripting block or hardware microcode template that can handle multiple write sequences efficiently.

Instruction Group 2: Read Operations

Read operations involve instructions that fetch register values for condition checking or data verification. Instructions like store_register read_mask <value> and copy_register reset_mask read_mask represent these operations. By grouping them, it becomes easier to standardize signal capture routines and set breakpoints for read verification during simulation.

This also aids in building readback validation tools and assertion-based testbenches, where the correct retrieval of data must be confirmed before proceeding.

Instruction Group 3: Jump if Not Zero (Conditional Branching)

Branching logic forms the basis of control flow in firmware. A common pattern observed was:

- store_register read_mask <value>
- copy_register reset_mask read_mask
- if_zero <jump_label>

This pattern is used for implementing conditional execution based on the evaluation of register content. Recognizing and isolating this structure simplifies the development of a unified jump/branch engine in the instruction decoder. These sequences can be reduced to micro-opcode equivalents, saving memory and improving execution speed in hardware-mapped interpreters.

Instruction Group 4: Wait and Synchronization Sequences

Wait logic is implemented through a combination of labels and condition checks. For example:

- :<wait_label>
- copy_address <register_address>
- if_zero <jump_label>
- jump <wait_label>

This logic forms polling-based synchronization between hardware states or processes. Grouping them helps in auto-generating wait-state handlers and simplifies waveform-based analysis where firmware execution is suspended until a certain condition is met. It also opens up the possibility of replacing software waits with event-driven triggers in future optimizations.

Instruction Group 5: Multiplication by Repeated Addition

The firmware implements multiplication using basic control flow and arithmetic primitives:

- Loop with if_not_zero, add, sub, and jump
- Intermediate accumulations using accumulator

This structure is critical for understanding how computational operations are emulated without native multiply instructions. Recognizing this sequence allows it to be abstracted into a macro or a single instruction in optimized versions, reducing the execution cycle count dramatically.

Instruction Group 6: Case Statement Logic

The case structure is formed using chained conditional checks:

- Multiple move register_address, sub, if_not_zero, and jump sequences

These are interpreted as cascading if-else blocks, common in decision-making routines. Grouping them makes it easier to implement a table-driven decision engine or finite-state machine in hardware, significantly reducing instruction footprint and improving control responsiveness.

Table 4. Shows the instructions sets that are converted into readable format

Category	**Original Assembly Instruction(s)**	**Converted Macro Instruction**
Write	store_addr write_mask value store_address register_address value copy_register reset_mask write_mask	WRITE(wr_mask_in_hex, wr_addr, wr_value);
Read	store_register read_mask <value> copy_address register_address copy_register reset_mask read_mask	READ(rd_mask, rd_addr);
Jump if Not Zero	store_register read_mask <value> copy_address register_address copy_register reset_mask read_mask if_zero <jump_label>	JMP_NOT_ZERO(rd_mask, register_address, jmp_label);
Wait	:<wait_label> copy_address <register_address> if_zero <jump_label> jump <wait_label>	WAIT(register_address, jmp_label, wait_label[optional]);
Multiplication	copy_value accumulator 0 :label_multiplication if_not_zero <label_multiplication_decrement> jmp <label_multiplication_done> add register_a accumulator sub 0x1 register_b jump <label_multiplication> :label_multiplication_done	MUL(operand1_register, operand2_register, result_register, label_name[optional]);
Case Statement	move register_address if_not_zero <try_label_1> jmp <label_0> :try_label_1 move register_address sub 0x1 accumulator if_not_zero <try_label_2> jump <label_1> ...	CASE(register_address) check value 1 in hex : jmp label1 ... default : default_label ENDCASE

4.2 Macro-Based Abstraction and Perl Automation Framework

In complex embedded firmware workflows, maintaining readability and consistency across low-level instruction sets becomes a growing challenge, especially as systems scale in complexity and functionality. This section introduces a macro-based abstraction mechanism integrated with a Perl automation framework that facilitates the preprocessing, transformation, and interpretation of assembly code. The goal of this approach is twofold: (1) improve human readability and debugging efficiency, and (2) enable seamless automation for regression and instruction analysis tasks.

The automation[20] framework primarily focuses on two Perl scripts:

- ❖ generate_asm_original.pl
- ❖ instr_conv.pl

Together, these scripts abstract raw .asm files into simplified, interpretable files suitable for firmware analysis and test planning, as shown in the figure below.

Workflow Description

The automation process is represented by a bidirectional transformation pipeline between the source assembly file and a cleaned, macro-processed output file. As depicted in the figure, the flow operates as follows:

1. **Input Assembly File**:
   The process begins with a raw .asm file containing low-level firmware instructions. These files often include comments, inconsistent indentation, labels, and instruction variants that make them difficult to analyze manually or pass into verification tools.

2. **Script 1: generate_asm_original.pl**:
   This Perl script serves as the first parser and cleaner in the pipeline. It takes the raw assembly input and removes extraneous whitespace, inline comments, and redundant labels. Its goal is to standardize the format of the assembly file and convert all instructions into a uniform structure. The output of this script is termed the **"readable file"**, a normalized version of the assembly code that is easier to process further.

3. **Readable File**:
   The readable file is a sanitized, compact version of the original code. It presents instructions in a canonical form, with each instruction occupying a single line, free from ambiguity or formatting noise. This makes it well-suited for both human reading and automated parsing.

4. **Script 2: instr_conv.pl**:
   Once the readable file is generated, it is passed into the second Perl script, instr_conv.pl. This script performs a macro-based translation of assembly instructions into high-level operations or pseudocode blocks. For instance, common patterns like conditional branches or wait loops are abstracted into labeled macros like WAIT_UNTIL_ZERO or JUMP_IF_NOT_ZERO, reflecting their intent rather than the literal syntax.

5. **Macro Enrichment**:
   The instr_conv.pl script uses regular expressions and parsing logic to identify these patterns and replace them with descriptive constructs. It enables modular reuse, simplifies verification, and supports the creation of a domain-specific language (DSL) for firmware test definition. In some cases, the macro layer also adds annotation headers or summaries for each logic block.

6. **Output Assembly File**:
   Finally, the abstracted file can be stored back as a modified .asm or further integrated into regression test frameworks, waveform debugging tools, or symbolic simulators. Since the entire flow is modular, additional transformation layers can be added without breaking the pipeline.

Advantages of Macro-Based Abstraction

The primary benefit of this methodology lies in its ability to distill complex assembly code into more meaningful representations. With the help of abstraction, developers and verification engineers can shift their focus from instruction-level execution to functional behavior.

- ❖ **Improved Debugging**: Macro translation allows easier tracking of logical blocks during signal-level analysis in Verdi, as one can correlate macro names to waveform regions or FSM transitions.

- **Repeatability**: Since all macros are derived through deterministic parsing, the output is consistent across multiple runs.
- **Simplified Documentation**: Macros double as documentation. Instead of needing comments, the structure of the abstracted code inherently conveys the operation.
- **Toolchain Integration**: The generated output can be reused across tools like waveform viewers, assertion monitors, or trace extractors.
- **Modularity**: New instruction formats or patterns can be added by extending the Perl script with additional pattern-matching logic.

Script Design and Customization

Both Perl scripts are designed to be modular and scalable. Configuration files or inline mappings can be used to extend the instruction sets or define new macros without altering the script's core logic. The automation also supports logging and diff comparisons, which allow for efficient regression checks during test updates.

The automation framework can also be extended to:

- Convert instruction sequences into FSM representations for easier state diagram generation.
- Generate bindable assertion templates based on macro types (e.g., a WAIT macro generating a corresponding wait_until assertion).
- Perform static analysis to detect dead code blocks or redundant operations.

CHAPTER 5

IMPLEMENTATION AND RESULTS

5.1 Pattern Generator, Matcher, and Error Detection Modules

The proposed system has been implemented using a modular approach that integrates pattern generation, matching, and error detection[9] mechanisms in a streamlined simulation environment. The core component, the Pattern Generator[22], is driven by an LFSR-based architecture that dynamically produces pseudo-random bit sequences for robust test case stimulation. These sequences are injected into the design under test (DUT), simulating realistic data traffic patterns for validation.

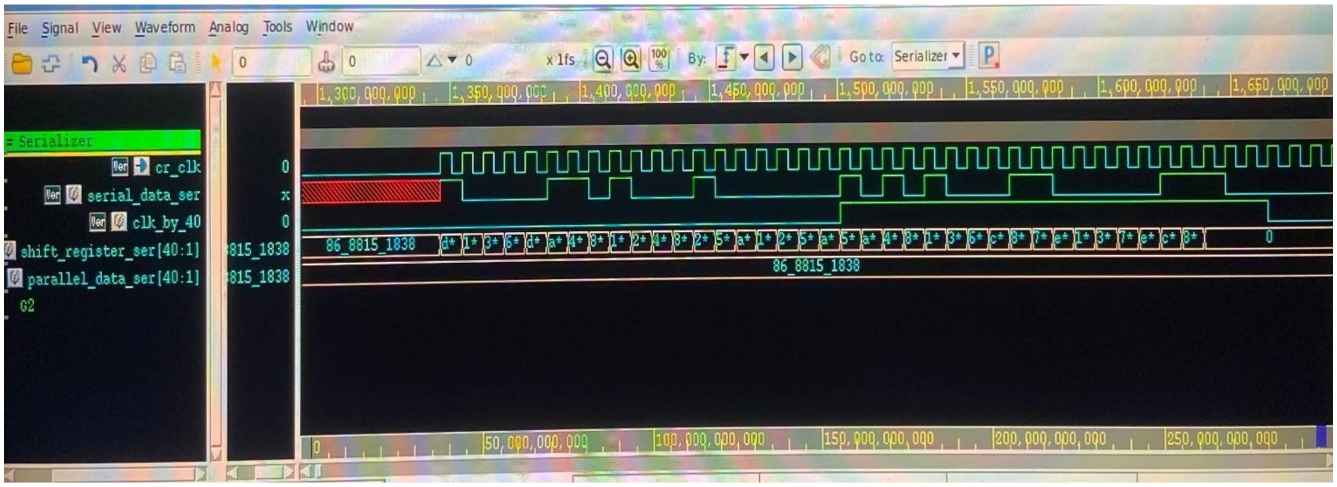


Figure 5.1. Serialized data out from serializer

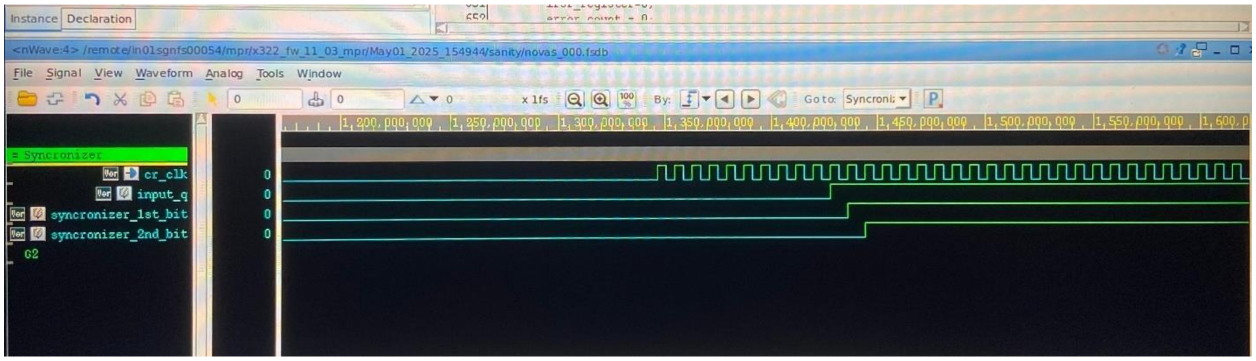


Figure 5.2. Synchronized data from 3 stage synchronizer

Following pattern generation, the Pattern Matcher module is tasked with continuously monitoring the output of the DUT. It uses a synchronized comparison logic that references expected outputs generated by a shadow LFSR or a predefined golden sequence. Any deviation is flagged by the Error Detection Module[18], which not only logs discrepancies but also isolates the failing bit index and timestamp for further waveform analysis in Verdi[4]. This modular structure ensures comprehensive coverage of edge cases, race conditions, and timing mismatches

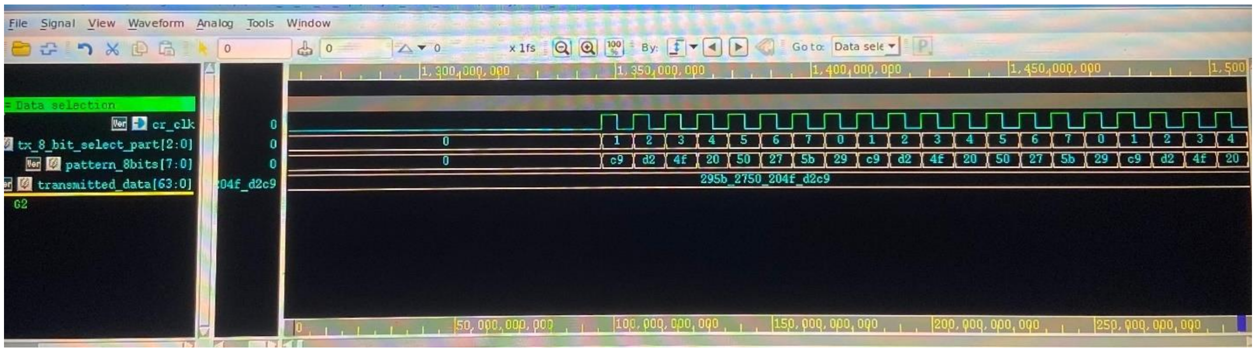


Figure 5.3. Data selection from 64 width depending on the transmission speed.

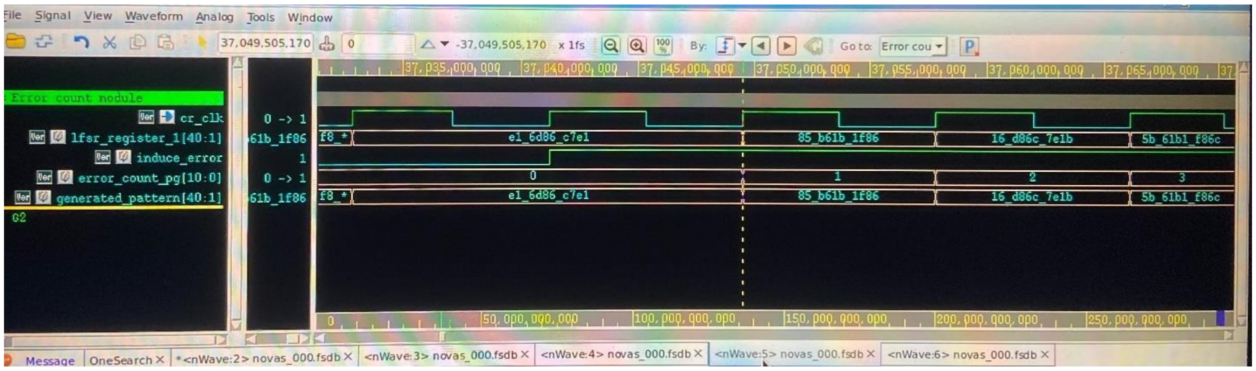


Figure 5.4. Error count module waveform

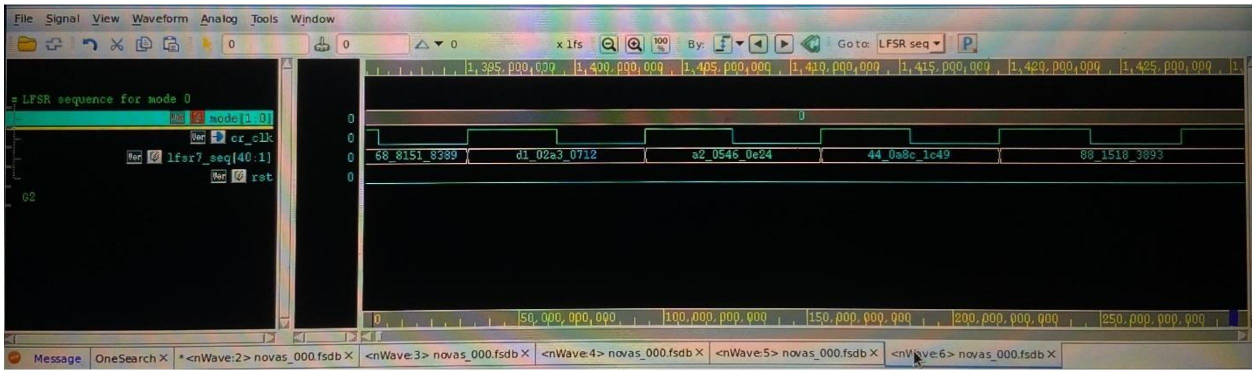


Figure 5.5. LFSR generation for mode 0 (polynomial 7)

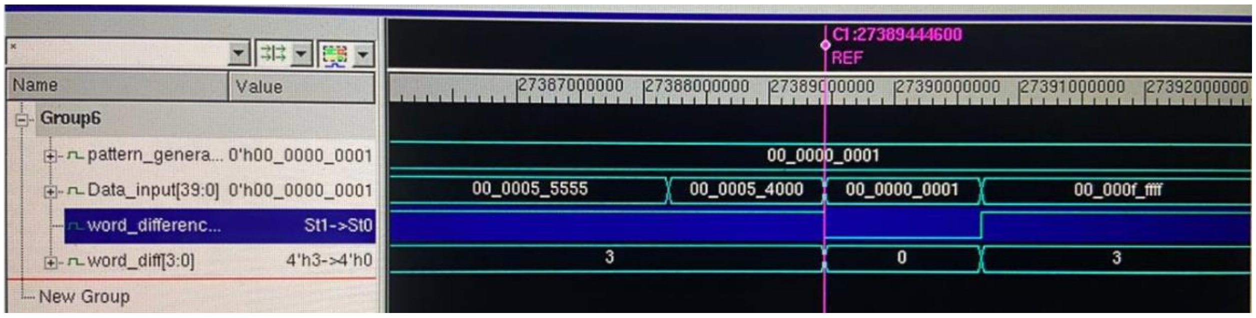


Figure 5.6. Waveform for word difference error count

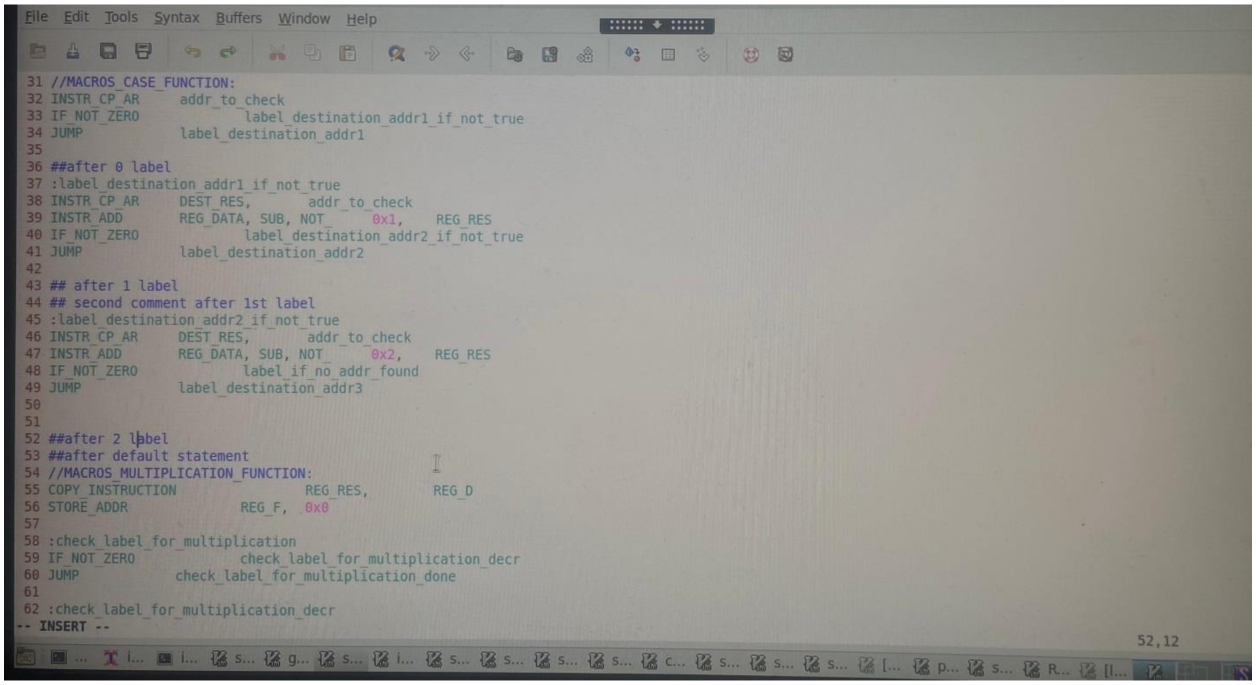
Waveform snapshots illustrating the exact matching, mismatch flagging, and sequence synchronization[3] will be appended to this section to provide a visual confirmation of functionality.

5.2 Efficiency Analysis: Power, Debug, and Instruction Reduction

In addition to functional accuracy, significant gains were achieved in debugging efficiency and script optimization. The integration of Perl-based automation for instruction parsing and pattern abstraction yielded tangible improvements in maintainability and script performance. Notably, the total line count across the Perl automation scripts was reduced from 39,314 lines to 35,807 lines, representing a 9% reduction[24]. This optimization was achieved by refactoring redundant blocks, merging overlapping condition checks, and using macro-based templates for repeated logic.

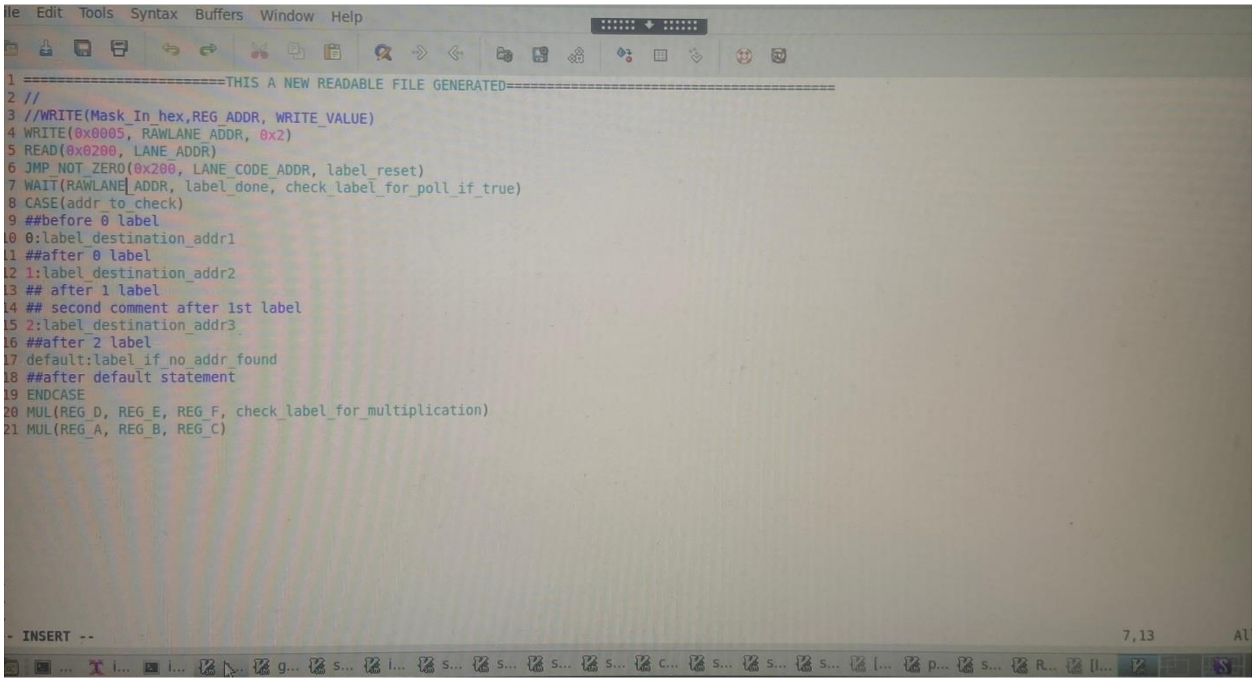
The resulting scripts not only occupy less memory footprint but also execute faster, particularly in batch regression environments where large volumes of test files are processed. Debug times were further reduced by linking waveform triggers with high-level macro labels, enabling engineers to jump directly to relevant portions in Verdi. Although direct power measurements were not captured due to the simulation environment, the reduction in instruction cycles and test iterations implies an indirect benefit in power estimation tools used downstream in ASIC design.

Collectively, the implementation demonstrates both architectural soundness and toolchain efficiency, setting a scalable foundation for further automation in instruction generation and verification environments.



```
//MACROS_CASE_FUNCTION:
INSTR_CP_AR     addr_to_check
IF_NOT_ZERO             label_destination_addr1_if_not_true
JUMP            label_destination_addr1

##after 0 label
:label_destination_addr1_if_not_true
INSTR_CP_AR     DEST_RES,       addr_to_check
INSTR_ADD       REG_DATA, SUB, NOT_      0x1,    REG_RES
IF_NOT_ZERO             label_destination_addr2_if_not_true
JUMP            label_destination_addr2

## after 1 label
## second comment after 1st label
:label_destination_addr2_if_not_true
INSTR_CP_AR     DEST_RES,       addr_to_check
INSTR_ADD       REG_DATA, SUB, NOT_      0x2,    REG_RES
IF_NOT_ZERO             label_if_no_addr_found
JUMP            label_destination_addr3


##after 2 label
##after default statement
//MACROS_MULTIPLICATION_FUNCTION:
COPY_INSTRUCTION                REG_RES,        REG_D
STORE_ADDR              REG_F,  0x0

:check_label_for_multiplication
IF_NOT_ZERO             check_label_for_multiplication_decr
JUMP            check_label_for_multiplication_done

:check_label_for_multiplication_decr
-- INSERT --                                                    52,12
```

Figure 5.7 . File with assembly language code



```
========================THIS A NEW READABLE FILE GENERATED========================================
//
//WRITE(Mask_In_hex,REG_ADDR, WRITE_VALUE)
WRITE(0x0005, RAWLANE_ADDR, 0x2)
READ(0x0200, LANE_ADDR)
JMP_NOT_ZERO(0x200, LANE_CODE_ADDR, label_reset)
WAIT(RAWLANE_ADDR, label_done, check_label_for_poll_if_true)
CASE(addr_to_check)
##before 0 label
0:label_destination_addr1
##after 0 label
1:label_destination_addr2
## after 1 label
## second comment after 1st label
2:label_destination_addr3
##after 2 label
default:label_if_no_addr_found
##after default statement
ENDCASE
MUL(REG_D, REG_E, REG_F, check_label_for_multiplication)
MUL(REG_A, REG_B, REG_C)

- INSERT --                                                     7,13
```

Figure 5.8. File after conversion in readable format

Table 5. Shows the overall gain using LFSR

Metric	LFSR	M-sequence
Hardware Gates (Est.)	~20	~40–60
Max Speed (Simulated)	High	Moderate–High

Table6. Shows the reduction in number of lines used

Metric	**Before Optimization**	**After Optimization**	**Improvement**
Total Perl Script Lines	39,314	35,807	3,507 lines reduced
Percentage Reduction	–	–	**~9% reduction**
Instruction Readability	Low (manual patterns)	High (macro-based)	Improved significantly
Maintenance Effort	High	Moderate	Easier to update
Debug Workflow Integration	Manual breakpoints	Linked macros + Verdi	Improved efficiency

CHAPTER 6

CONCLUSION AND FUTURE SCOPE

This thesis presents a dual-focused approach combining PRBS-based data generation and Perl-driven scripting automation to enhance digital system validation and firmware instruction management. The LFSR-based pseudo-random pattern generator, along with synchronizer modules, enabled robust, repeatable test streams for accurate functional verification. Simultaneously, the scripting framework introduced a macro-level abstraction and automated parsing of assembly instructions, significantly reducing manual overhead and achieving a 9% reduction in script size.The integration of these components within a modular verification flow has not only streamlined debugging and analysis using Verdi but also provided a scalable infrastructure for instruction profiling, test optimization, and modular testbench design. The improvements achieved in simulation clarity, maintainability, and runtime efficiency mark a considerable advancement in hardware-software co-verification.

Future work may explore extending the PRBS generator to support multiple polynomial configurations and seed randomization for broader coverage. On the scripting side, introducing GUI-based configuration tools or integrating with YAML/JSON for testplan-driven execution could further enhance usability. Additionally, transitioning to Python-based frameworks may offer improved cross-platform compatibility and richer libraries for pattern analysis and waveform correlation.

REFERENCES

[1] M.-S. Chen and C.-K. K. Yang, "A Low-Power Highly Multiplexed Parallel PRBS Generator"

[2] Qing Wu, M. Pedram, and Xunwei Wu, "Clock-Gating and Its Application to Low Power Design of Sequential Circuits"

[3] D. Fisher and H. T. Kung, "Synchronizing Large VLSI Processor Arrays"

[4] SYNOPSYS, Verdi Advanced Debug Platform – User Guide and Training Material, Synopsys

[5] K. K. Parhi, “VLSI Digital Signal Processing Systems”

[6] D. Gajski, “Principles of Digital Design”

[7] C. E. Shannon, “Communication in the presence of noise,”

[8] J. L. Hennessy and D. A. Patterson, “Computer Architecture: A Quantitative Approach”

[9] B. W. Johnson, “Design and Analysis of Fault-Tolerant Digital Systems”

[10] B. P. Lathi, “Modern Digital and Analog Communication Systems”

[11] J. S. Walling, “Design of LFSRs for high-speed BIST applications”

[12] J. Martin, “Synchronization and Arbitration in Digital Systems”

[13] M. Abramovici et al., “Digital Systems Testing and Testable Design”

[14] M. Bushnell and V. Agrawal, “Essentials of Electronic Testing”

[15] S. Kang and Y. Leblebici, “CMOS Digital Integrated Circuits”

[16] K. Roy and S. Prasad, “Low-Power CMOS VLSI Circuit Design”

[17] D. Rossi et al., “Flexible BIST architecture using multiple PRPGs”

[18] K. S. S. A. Kumar et al., “PRBS Verification using LFSR in BERT”

[19] N. Zeldovich et al., “Hardware Software Co-Design Automation”

[20] P. Mishra, “Automation in Firmware Debug Flows using Perl/Python”

[21] L. Cai and D. Gajski, “Instruction Abstraction for Embedded Architectures”

[22] A. Al-Yamani and T. M. Pinkston, “On-chip Network Testing using PRBS”

[23] F. Brglez, “BIST strategies for IP cores and subsystems”

[24] D. Brooks and M. Martonosi, “Dynamic instruction profiling for performance”