Stream Cipher Design Using LFSRs

Sergio Brito

February 20, 2025

Abstract

This document explains the design and implementation of a stream cipher using a Linear Feedback Shift Register (LFSR). We'll cover the basic concepts of XOR operations, LFSR functionality, encryption principles, and the importance of synchronism in digital systems. The goal is to provide both theoretical background and practical VHDL examples for educational purposes.

1 Introduction

In this document, we explore the fundamentals of stream ciphers with an emphasis on LFSR-based designs. We briefly review how pseudo-random bit generators, like LFSRs, and the XOR operation can be combined to create effective encryption mechanisms. The focus will be on understanding the core concepts and the practical aspects of implementing these ideas using VHDL.

The subsequent sections will delve into detailed explanations of the XOR operation, LFSRs, encryption and decryption processes, timing considerations, and practical VHDL implementations complete with testbenches.

2 The XOR Operation

The XOR (exclusive OR) operation is a fundamental building block in digital logic and cryptography. It outputs a '1' when the number of '1' inputs is odd, and '0' otherwise. This property makes XOR perfect for encryption since applying it twice with the same key restores the original data.

XOR Truth Table

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Example: 1-Byte Encryption and Parity Generation

Consider a single byte of data and an 8-bit key:

$$\text{Data} = 10101100$$
$$\text{Key} = 11001010$$

Encryption: We can illustrate the XOR operation bit-by-bit in a table:

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Data	1	0	1	0	1	1	0	0
Key	1	1	0	0	1	0	1	0
XOR	0	1	1	0	0	1	1	0

Thus, the **encrypted byte** is `01100110`.

Decryption: We apply XOR again with the same key:

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Encrypted	0	1	1	0	0	1	1	0
Key	1	1	0	0	1	0	1	0
XOR	1	0	1	0	1	1	0	0

Hence, we recover the original **Data** = `10101100`.

Additionally, XOR is often used for generating a *parity bit* in error detection schemes. By XORing all bits of a data byte, you produce a single bit that ensures the total count of 1s is even (for even parity) or odd (for odd parity), enabling basic detection of single-bit errors.

3 Linear Feedback Shift Registers

3.1 Polynomials and Tap Selection

An **LFSR** is often described by a characteristic polynomial over the finite field $GF(2)$. For an n-bit register, the polynomial has the form:

$$x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + 1,$$

where each a_i is either 0 or 1. A coefficient of 1 indicates a *tap* at that bit position. For example, the polynomial

$$x^8 + x^6 + x^5 + x^4 + 1$$

implies taps at bits 7, 5, 4, and 3 (counting from 0 as the least significant bit).

If this polynomial is *primitive*, the LFSR will produce a **maximum-length sequence** of $2^n - 1$ non-repeating states. Otherwise, it may cycle prematurely. Thus, selecting the correct taps (i.e., a primitive polynomial) is crucial for achieving the longest possible pseudo-random sequence.

3.2 What Are LFSRs?

A **Linear Feedback Shift Register (LFSR)** is a sequence of flip-flops (bits) that shifts its contents every clock cycle. The new bit entering the register is determined by XORing specific *tap* bits of the current state. Because the next state depends linearly (via XOR) on the current state, these registers are easy to implement in hardware yet can generate long pseudo-random sequences if the tap polynomial is chosen wisely.

3.3 Pseudo-Randomness

Although an LFSR is entirely deterministic, it can still generate sequences that *appear* random under many circumstances. The term **pseudo-randomness** refers to this illusion of randomness arising from a predictable, yet complex, pattern of bits. Key points include:

- **Deterministic Nature:** Once the initial state (seed) and tap configuration are known, the entire sequence is fixed. There is nothing inherently random about an LFSR.

- **Statistical Properties:** Well-chosen tap polynomials can produce sequences that pass many statistical randomness tests (e.g., distribution of 0s and 1s, autocorrelation properties).
- **Periodicity:** If the polynomial is primitive, the LFSR will cycle through $2^n - 1$ distinct states before repeating. However, once it repeats, the sequence is fully predictable.
- **Cryptographic Concerns:** While LFSRs can be used in stream ciphers, a single LFSR is not *cryptographically secure* by modern standards. Additional techniques (like combining multiple LFSRs or using nonlinear filters) are often employed to enhance security.

Pseudo-random sequences generated by LFSRs are useful in applications like *scrambling*, *error checking*, and simple *encryption* demonstrations. However, for robust cryptographic purposes, more advanced methods are recommended.

3.4 Fibonacci vs. Galois LFSRs

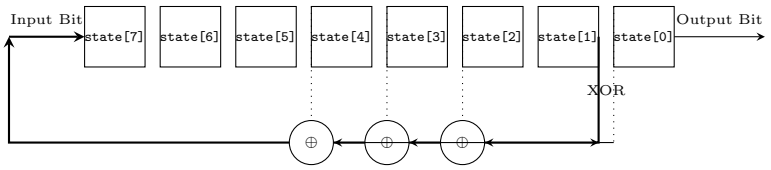
There are two common ways to arrange the XOR feedback in an LFSR:

- **Fibonacci LFSR**: A single XOR network combines the tap bits externally, then feeds that result back into the register's input (the most significant bit).
- **Galois LFSR**: The XOR gates are distributed throughout the register; each tapped bit directly modifies one of the register stages during shifting.

Both approaches can yield the same maximum-length sequences if the taps correspond to a primitive polynomial, but they differ in hardware efficiency and conceptual simplicity. Fibonacci LFSRs are straightforward to understand, while Galois LFSRs can be more efficient in certain hardware implementations.

3.5 Fibonacci LFSR Example

Below is a simplified diagram of an 8-bit *Fibonacci* LFSR with taps at bits 7, 5, 4, and 3 (one of the well-known maximum-length polynomials for 8-bit registers):



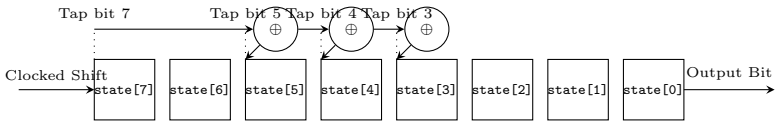
Operation:

1. On each clock, the register shifts to the right: `state[i]` $\rightarrow$ `state[i+1]`.
2. The old `state[0]` (LSB) becomes the *Output Bit*.
3. The XOR of the tap bits is fed back into `state[7]` (MSB).

If the taps correspond to a primitive polynomial (e.g., $x^8 + x^6 + x^5 + x^4 + 1$), this LFSR will generate a maximum-length sequence of $2^8 - 1 = 255$ states before repeating.

3.6 Galois LFSR Example

In the **Galois** form, each tapped bit modifies a register stage directly rather than feeding a single XOR at the input. Here's a conceptual diagram for the same tap positions (bits 7, 5, 4, 3):



Operation:

- The MSB (bit 7) shifts into bit 6, bit 6 shifts into bit 5, and so on.
- Each tapped stage (bits 5, 4, 3) is XORed with the old MSB before storing the new value.
- The LSB (bit 0) is ejected as the output bit on every clock cycle.

4 Encryption Using LFSRs

A simple way to build a **stream cipher** is to XOR each bit of the plaintext with a pseudo-random bit produced by the LFSR. Because XOR is its own inverse, the same operation decrypts the ciphertext when using the identical LFSR key stream.

4.1 Basic Principle

$$\text{Ciphertext} = \text{Plaintext} \oplus \text{KeyStream}$$

$$\text{Plaintext} = \text{Ciphertext} \oplus \text{KeyStream}$$

Here, KeyStream is generated by shifting the LFSR each clock cycle. The output bit (e.g., the LSB) serves as the *key bit* for encryption.

- **Initialization**: Load the LFSR with a seed (initial state).
- **Encryption**: For each incoming plaintext bit, XOR it with the LFSR output bit.
- **Update**: After each bit, shift the LFSR to produce the next key bit.

4.2 Decryption

Decryption follows the exact same process. The receiver's LFSR must be *identically initialized* and clocked in sync with the sender's. By XORing the ciphertext bit with the LFSR output bit, the original plaintext is recovered.

4.3 Example Flow

1. **Sender:**
   (a) Loads the LFSR with a known seed.
   (b) Generates one key bit per clock cycle.
   (c) XORs each key bit with the plaintext bit to form ciphertext.

2. **Receiver:**
    (a) Loads the same LFSR seed.
    (b) Generates the same key bits in sync.
    (c) XORs each key bit with the ciphertext bit to recover plaintext.

Note on Synchronization: If sender and receiver ever lose alignment (e.g., due to dropped bits or starting on different cycles), decryption fails until they resynchronize. This is why controlling and maintaining clock alignment is critical in hardware stream ciphers.

This simple XOR-based scheme demonstrates the core principle behind many stream ciphers. Of course, real-world ciphers often combine multiple LFSRs or add nonlinear elements to increase security.

5 Synchronization and Timing

In digital systems, **timing** is everything. When implementing an LFSR-based stream cipher, both encryption and decryption units must:

1. **Share the same clock frequency.**
2. **Start with identical seeds (initial states).**
3. **Advance the LFSR in perfect lockstep.**

5.1 Clock Domains and Alignment

If the sender and receiver operate on different clock domains or frequencies, the LFSR outputs will not match bit-for-bit over time. This leads to:

- **Bit Slips:** A single missed or extra clock causes a permanent shift in the key stream alignment.
- **Decryption Failure:** Once out of sync, XORing the wrong key bits with ciphertext yields garbage data.

5.2 Practical Considerations

- **Reset Synchronization:** Ensure that both sides apply reset at the same time, loading the same seed into the LFSR.
- **Enable Control:** If one side pauses the LFSR (e.g., via an enable signal) and the other side does not, they lose alignment.
- **Transmission Protocols:** Real systems often embed special markers or frame headers so the receiver can detect and correct alignment errors.

5.3 Timing Diagrams

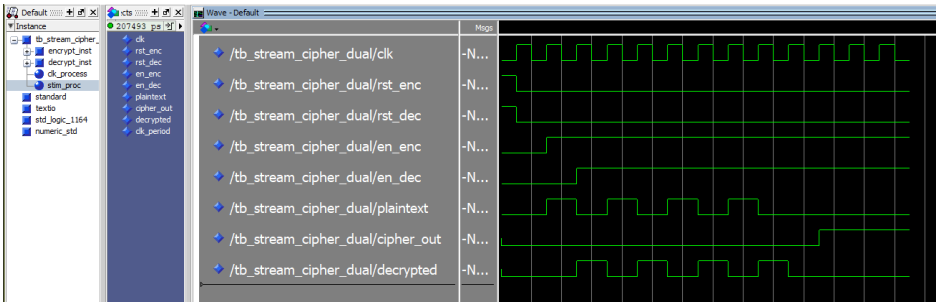


Figure 1: Timing diagram for synchronized encryption and decryption.

Maintaining synchronization is essential for correct decryption. A single cycle of misalignment is enough to corrupt the entire decrypted stream until the system is resynchronized.

6 Practical Example

To solidify these concepts, we present a simple VHDL implementation of a stream cipher using an LFSR. The design consists of:

- **LFSR Module:** Generates one pseudo-random bit per clock.
- **Stream Cipher Module:** XORs each incoming data bit with the LFSR output.
- **Testbench:** Demonstrates encryption and decryption using identical LFSRs in sync.

6.1 LFSR Entity

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lfsr_stream is
    port (
        clk     : in std_logic;
        rst     : in std_logic;
        en      : in std_logic;
        seed    : in std_logic_vector(7 downto 0);
        out_bit : out std_logic
    );
end lfsr_stream;
```

The `lfsr_stream` entity produces a single bit of pseudo-random output each clock cycle when enabled (`en`).

6.2 Stream Cipher Entity

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity stream_cipher is
    port (
        clk      : in std_logic;
        rst      : in std_logic;
        en       : in std_logic;
        data_in  : in std_logic;
        data_out : out std_logic
    );
end stream_cipher;
```

This module instantiates the LFSR, then XORs its output bit with `data_in` to produce the encrypted (or decrypted) `data_out`.

6.3 Dual Testbench

A typical testbench would instantiate two `stream_cipher` modules:

- **Encryption**: Feeds plaintext bits into one cipher instance.
- **Decryption**: Receives ciphertext from the first, XORing with the same LFSR sequence to recover the original data.

They must share the same seed and clock to remain synchronized.

6.4 Operation and Results

When you simulate, you'll observe:

1. Plaintext bits enter the encryption cipher, get XORed with the LFSR output, and emerge as ciphertext.
2. The second cipher uses the same seed, clock, and LFSR taps, XORing the ciphertext to retrieve the original plaintext.

This verifies that if both sides are synchronized, encryption and decryption match perfectly. Any loss of sync breaks the decryption until re-alignment.

This simple example showcases the fundamental principle of stream ciphers using LFSRs. In practice, more sophisticated methods (like multiple LFSRs combined or nonlinear filters) are often employed to enhance security.