Linear-feedback shift register

In computing, a **linear-feedback shift register** (**LFSR**) is a shift register whose input bit is a linear function of its previous state.

The most commonly used linear function of single bits is exclusive-or (XOR). Thus, an LFSR is most often a shift register whose input bit is driven by the XOR of some bits of the overall shift register value.

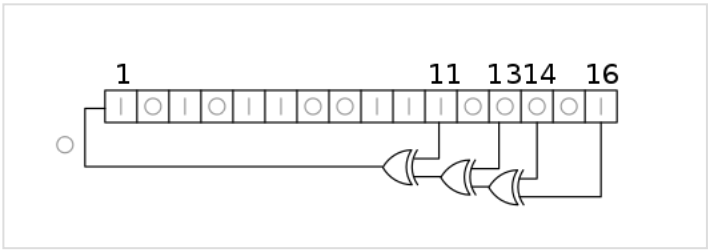
The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits that appears random and has a very long cycle.

Applications of LFSRs include generating pseudo-random numbers, pseudo-noise sequences, fast digital counters, and whitening sequences. Both hardware and software implementations of LFSRs are common.

The mathematics of a cyclic redundancy check, used to provide a quick check against transmission errors, are closely related to those of an LFSR.[1] In general, the arithmetics behind LFSRs makes them very elegant as an object to study and implement. One can produce relatively complex logics with simple building blocks. However, other methods, that are less elegant but perform better, should be considered as well.

Fibonacci LFSRs

The bit positions that affect the next state are called the *taps*. In the diagram the taps are [16,14,13,11]. The rightmost bit of the LFSR is called the output bit, which is always also a tap. To obtain the next state, the tap bits are XOR-ed sequentially; then, all bits are shifted one place to the right, with the rightmost bit being discarded, and that result of XOR-ing the tap bits is fed back into the now-vacant leftmost bit. To obtain the pseudorandom output stream, read the rightmost bit after each state transition.



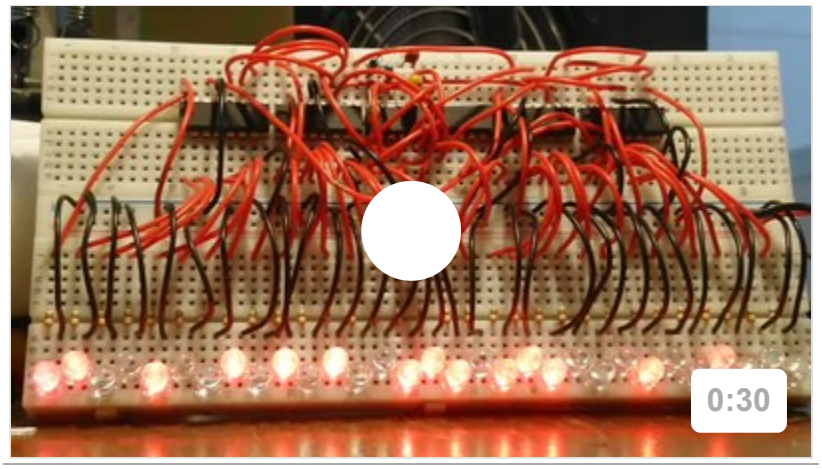
A 16-bit Fibonacci LFSR. The feedback tap numbers shown correspond to a primitive polynomial in the table, so the register cycles through the maximum number of 65535 states excluding the all-zeroes state. The state shown, 0xACE1 (hexadecimal) will be followed by 0x5670.

- A maximum-length LFSR produces an m-sequence (i.e., it cycles through all possible $2^m - 1$ states within the shift register except the state where all bits are zero), unless it contains all zeros, in which case it will never change.
- As an alternative to the XOR-based feedback in an LFSR, one can also use XNOR.[2] This function is an affine map, not strictly a linear map, but it results in an equivalent polynomial counter whose state is the complement of the state of an LFSR. A state with all ones is illegal when using an XNOR feedback, in the same way as a state with all zeroes is illegal when using XOR. This state is considered illegal because the counter would remain "locked-up" in this state. This method can be advantageous in hardware LFSRs using flip-flops that start in a zero state, as it does not start in a lockup state, meaning that the register does not need to be seeded in order to begin operation.

The sequence of numbers generated by an LFSR or its XNOR counterpart can be considered a binary numeral system just as valid as Gray code or the natural binary code.

The arrangement of taps for feedback in an LFSR can be expressed in finite field arithmetic as a polynomial mod 2. This means that the coefficients of the polynomial must be 1s or 0s. This is called the feedback polynomial or reciprocal characteristic polynomial. For example, if the taps are at the 16th, 14th, 13th and 11th bits (as shown), the feedback polynomial is

$$x^{16} + x^{14} + x^{13} + x^{11} + 1.$$



A Fibonacci 31 bit linear feedback shift register with taps at positions 28 and 31, giving it a maximum cycle and period at this speed of nearly 6.7 years.

The "one" in the polynomial does not correspond to a tap – it corresponds to the input to the first bit (i.e. x^0, which is equivalent to 1). The powers of the terms represent the tapped bits, counting from the left. The first and last bits are always connected as an input and output tap respectively.

The LFSR is maximal-length if and only if the corresponding feedback polynomial is primitive over the Galois field GF(2).[3][4] This means that the following conditions are necessary (but not sufficient):

- The number of taps is even.
- The set of taps is setwise co-prime; i.e., there must be no divisor other than 1 common to all taps.

Tables of primitive polynomials from which maximum-length LFSRs can be constructed are given below and in the references.

There can be more than one maximum-length tap sequence for a given LFSR length. Also, once one maximum-length tap sequence has been found, another automatically follows. If the tap sequence in an n-bit LFSR is [n, A, B, C, 0], where the 0 corresponds to the $x^0 = 1$ term, then the corresponding "mirror" sequence is [n, $n - C$, $n - B$, $n - A$, 0]. So the tap sequence [32, 22, 2, 1, 0] has as its counterpart [32, 31, 30, 10, 0]. Both give a maximum-length sequence.

An example in C is below:

```c
#include <stdint.h>
unsigned lfsr_fib(void)
{
    uint16_t start_state = 0xACE1u;  /* Any nonzero start state will work. */
    uint16_t lfsr = start_state;
    uint16_t bit;                    /* Must be 16-bit to allow bit<<15 later in the code */
    unsigned period = 0;

    do
    {   /* taps: 16 14 13 11; feedback polynomial: x^16 + x^14 + x^13 + x^11 + 1 */
        bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5)) & 1u;
        lfsr = (lfsr >> 1) | (bit << 15);
        ++period;
    }
    while (lfsr != start_state);

    return period;
}
```

If a fast parity or popcount operation is available, the feedback bit can be computed more efficiently as the dot product of the register with the characteristic polynomial:

- `bit = parity(lfsr & 0x002Du);`, or equivalently
- `bit = popcnt(lfsr & 0x002Du) /* & 1u */;`. (The `& 1u` turns the popcnt into a true parity function, but the bitshift later `bit << 15` makes higher bits irrelevant.)

If a rotation operation is available, the new state can be computed as

- `lfsr = rotateright((lfsr & ~1u) | (bit & 1u), 1);`, or equivalently
- `lfsr = rotateright(((bit ^ lfsr) & 1u) ^ lfsr, 1);`

This LFSR configuration is also known as **standard**, **many-to-one** or **external XOR gates**. The alternative Galois configuration is described in the next section.

Example in Python

A sample python implementation of a similar (16 bit taps at [16,15,13,4]) Fibonacci LFSR would be

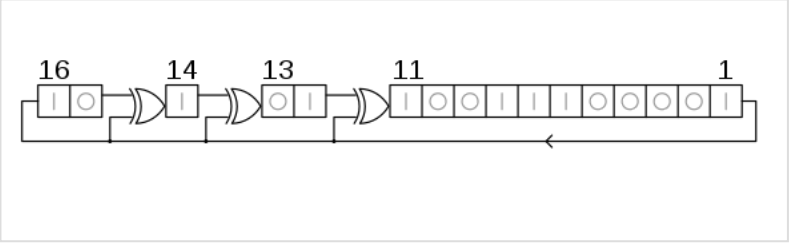
```
start_state = 1 << 15 | 1
lfsr = start_state
period = 0

while True:
    # taps: 16 15 13 4; feedback polynomial: x^16 + x^15 + x^13 + x^4 + 1
    bit = (lfsr ^ (lfsr >> 1) ^ (lfsr >> 3) ^ (lfsr >> 12)) & 1
    lfsr = (lfsr >> 1) | (bit << 15)
    period += 1
    if lfsr == start_state:
        print(period)
        break
```

Where a register of 16 bits is used and the xor tap at the fourth, 13th, 15th and sixteenth bit establishes a maximum sequence length.

Galois LFSRs

Named after the French mathematician Évariste Galois, an LFSR in Galois configuration, which is also known as **modular**, **internal XORs**, or **one-to-many LFSR**, is an alternate structure that can generate the same output stream as a conventional LFSR (but offset in time).[5] In the Galois configuration, when the system is clocked, bits that are not taps are shifted one position to the right unchanged. The taps, on the other hand, are XORed with the output bit before they are stored in the next position. The new output bit is the next input bit. The effect of this is that when the output bit is zero, all the bits in the register shift to the right unchanged, and the input bit becomes zero. When the output bit is one, the bits in the tap positions all flip (if they are 0, they become 1, and if they are 1, they become 0), and then the entire register is shifted to the right and the input bit becomes 1.



A 16-bit Galois LFSR. The register numbers above correspond to the same primitive polynomial as the Fibonacci example but are counted in reverse to the shifting direction. This register also cycles through the maximal number of 65535 states excluding the all-zeroes state. The state ACE1 hex shown will be followed by E270 hex.

To generate the same output stream, the order of the taps is the *counterpart* (see above) of the order for the conventional LFSR, otherwise the stream will be in reverse. Note that the internal state of the LFSR is not necessarily the same. The Galois register shown has the same output stream as the Fibonacci register in the first section. A time offset exists between the streams, so a different startpoint will be needed to get the same output each cycle.

- Galois LFSRs do not concatenate every tap to produce the new input (the XORing is done within the LFSR, and no XOR gates are run in serial, therefore the propagation times are reduced to that of one XOR rather than a whole chain), thus it is possible for each tap to be computed in parallel, increasing the speed of execution.
- In a software implementation of an LFSR, the Galois form is more efficient, as the XOR operations can be implemented a word at a time: only the output bit must be examined individually.

Below is a C code example for the 16-bit maximal-period Galois LFSR example in the figure:

```c
#include <stdint.h>
unsigned lfsr_galois(void)
{
    uint16_t start_state = 0xACE1u;  /* Any nonzero start state will work. */
    uint16_t lfsr = start_state;
    unsigned period = 0;

    do
    {
#ifndef LEFT
        unsigned lsb = lfsr & 1u;  /* Get LSB (i.e., the output bit). */
        lfsr >>= 1;                /* Shift register */
        if (lsb)                   /* If the output bit is 1, */
            lfsr ^= 0xB400u;       /*  apply toggle mask. */
#else
        unsigned msb = (int16_t) lfsr < 0;   /* Get MSB (i.e., the output bit). */
        lfsr <<= 1;                          /* Shift register */
        if (msb)                             /* If the output bit is 1, */
            lfsr ^= 0x002Du;                 /*  apply toggle mask. */
#endif
        ++period;
    }
    while (lfsr != start_state);

    return period;
}
```

The branch `if (lsb) lfsr ^= 0xB400u;` can also be written as `lfsr ^= (-lsb) & 0xB400u;` which may produce more efficient code on some compilers. In addition, the left-shifting variant may produce even better code, as the msb is the carry from the addition of `lfsr` to itself.

Galois LFSR parallel computation

State and resulting bits can also be combined and computed in parallel. The following function calculates the next 64 bits using 63-bit polynomial $x^{63} + x^{62} + 1$:

```c
#include <stdint.h>

uint64_t prsg63(uint64_t lfsr) {
    lfsr = lfsr << 32 | (lfsr<<1 ^ lfsr<<2) >> 32;
    lfsr = lfsr << 32 | (lfsr<<1 ^ lfsr<<2) >> 32;
    return lfsr;
}
```

Non-binary Galois LFSR

Binary Galois LFSRs like the ones shown above can be generalized to any q-ary alphabet {0, 1, ..., q − 1} (e.g., for binary, q = 2, and the alphabet is simply {0, 1}). In this case, the exclusive-or component is generalized to addition modulo-q (note that XOR is addition modulo 2), and the feedback bit (output bit) is multiplied (modulo-q) by a q-ary value, which is constant for each specific tap point. Note that this is also a generalization of the binary case, where the feedback is multiplied by either 0 (no feedback, i.e., no tap) or 1 (feedback is present). Given an appropriate tap configuration, such LFSRs can be used to generate Galois fields for arbitrary prime values of q.

Xorshift LFSRs

As shown by George Marsaglia[6] and further analysed by Richard P. Brent,[7] linear feedback shift registers can be implemented using XOR and Shift operations. This approach lends itself to fast execution in software because these operations typically map efficiently into modern processor instructions.

Below is a C code example for a 16-bit maximal-period Xorshift LFSR using the 7,9,13 triplet from John Metcalf:[8]

```c
#include <stdint.h>
unsigned lfsr_xorshift(void)
{
    uint16_t start_state = 0xACE1u;  /* Any nonzero start state will work. */
    uint16_t lfsr = start_state;
    unsigned period = 0;

    do
    {   // 7,9,13 triplet from http://www.retroprogramming.com/2017/07/xorshift-pseudorandom-numbers-in-z80.html
        lfsr ^= lfsr >> 7;
        lfsr ^= lfsr << 9;
        lfsr ^= lfsr >> 13;
        ++period;
    }
    while (lfsr != start_state);

    return period;
}
```

Matrix forms

Binary LFSRs of both Fibonacci and Galois configurations can be expressed as linear functions using matrices in $\mathbb{F}_2$ (see GF(2)).[9] Using the companion matrix of the characteristic polynomial of the LFSR and denoting the seed as a column vector $(a_0, a_1, \ldots, a_{n-1})^{\mathrm{T}}$, the state of the register in Fibonacci configuration after k steps is given by

$$\begin{pmatrix} a_k \\ a_{k+1} \\ a_{k+2} \\ \vdots \\ a_{k+n-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & 1 \\ c_0 & c_1 & \cdots & \cdots & c_{n-1} \end{pmatrix} \begin{pmatrix} a_{k-1} \\ a_k \\ a_{k+1} \\ \vdots \\ a_{k+n-2} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & 1 \\ c_0 & c_1 & \cdots & \cdots & c_{n-1} \end{pmatrix}^k \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Matrix for the corresponding Galois form is :

$$\begin{pmatrix} c_0 & 1 & 0 & \cdots & 0 \\ c_1 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ c_{n-2} & 0 & \cdots & 0 & 1 \\ c_{n-1} & 0 & \cdots & \cdots & 0 \end{pmatrix}$$

For a suitable initialisation,

$$a'_i = \sum_{i=0}^{j} a_{i-j} c_{n-j},\ 0 \leq i < n$$

the top coefficient of the column vector :

$$\begin{pmatrix} c_0 & 1 & 0 & \cdots & 0 \\ c_1 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ c_{n-2} & 0 & \cdots & 0 & 1 \\ c_{n-1} & 0 & \cdots & \cdots & 0 \end{pmatrix}^k \begin{pmatrix} a'_0 \\ a'_1 \\ a'_2 \\ \vdots \\ a'_{n-1} \end{pmatrix}$$

gives the term a_k of the original sequence.

These forms generalize naturally to arbitrary fields.

Example polynomials for maximal LFSRs

The following table lists examples of maximal-length feedback polynomials (primitive polynomials) for shift-register lengths up to 24. The formalism for maximum-length LFSRs was developed by Solomon W. Golomb in his 1967 book.[10] The number of different primitive polynomials grows exponentially with shift-register length and can be calculated exactly using Euler's totient function[11] (sequence A011260 in the OEIS).

Bits (n)	Feedback polynomial	Taps	Taps (hex)	Period ($2^n - 1$)
2	$x^2 + x + 1$	11	0x3	3
3	$x^3 + x^2 + 1$	110	0x6	7
4	$x^4 + x^3 + 1$	1100	0xC	15
5	$x^5 + x^3 + 1$	10100	0x14	31
6	$x^6 + x^5 + 1$	110000	0x30	63
7	$x^7 + x^6 + 1$	1100000	0x60	127
8	$x^8 + x^6 + x^5 + x^4 + 1$	10111000	0xB8	255
9	$x^9 + x^5 + 1$	100010000	0x110	511
10	$x^{10} + x^7 + 1$	1001000000	0x240	1,023
11	$x^{11} + x^9 + 1$	10100000000	0x500	2,047
12	$x^{12} + x^{11} + x^{10} + x^4 + 1$	111000001000	0xE08	4,095
13	$x^{13} + x^{12} + x^{11} + x^8 + 1$	1110010000000	0x1C80	8,191
14	$x^{14} + x^{13} + x^{12} + x^2 + 1$	11100000000010	0x3802	16,383
15	$x^{15} + x^{14} + 1$	110000000000000	0x6000	32,767
16	$x^{16} + x^{15} + x^{13} + x^4 + 1$	1101000000001000	0xD008	65,535
17	$x^{17} + x^{14} + 1$	10010000000000000	0x12000	131,071
18	$x^{18} + x^{11} + 1$	100000010000000000	0x20400	262,143
19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	1110010000000000000	0x72000	524,287
20	$x^{20} + x^{17} + 1$	10010000000000000000	0x90000	1,048,575
21	$x^{21} + x^{19} + 1$	10100000000000000000	0x140000	2,097,151
22	$x^{22} + x^{21} + 1$	11000000000000000000	0x300000	4,194,303
23	$x^{23} + x^{18} + 1$	10000100000000000000000	0x420000	8,388,607
24	$x^{24} + x^{23} + x^{22} + x^{17} + 1$	11100001000000000000000	0xE10000	16,777,215

Xilinx published (https://docs.xilinx.com/v/u/en-US/xapp052) an extended list of tap counters up to 168 bit. Tables of maximum length polynomials are available from http://users.ece.cmu.edu/~koopman/lfsr/ and can be generated by the https://github.com/hayguen/mlpolygen project.

Output-stream properties

- Ones and zeroes occur in "runs". The output stream 1110010, for example, consists of four runs of lengths 3, 2, 1, 1, in order. In one period of a maximal LFSR, 2^{n-1} runs occur (in the example above, the

3-bit LFSR has 4 runs). Exactly half of these runs are one bit long, a quarter are two bits long, up to a single run of zeroes $n - 1$ bits long, and a single run of ones n bits long. This distribution almost equals the statistical expectation value for a truly random sequence. However, the probability of finding exactly this distribution in a sample of a truly random sequence is rather low.

- LFSR output streams are deterministic. If the present state and the positions of the XOR gates in the LFSR are known, the next state can be predicted.[12] This is not possible with truly random events. With maximal-length LFSRs, it is much easier to compute the next state, as there are only an easily limited number of them for each length.
- The output stream is reversible; an LFSR with mirrored taps will cycle through the output sequence in reverse order.
- The value consisting of all zeros cannot appear. Thus an LFSR of length n cannot be used to generate all 2^n values.

Applications

LFSRs can be implemented in hardware, and this makes them useful in applications that require very fast generation of a pseudo-random sequence, such as direct-sequence spread spectrum radio. LFSRs have also been used for generating an approximation of white noise in various programmable sound generators.

Uses as counters

The repeating sequence of states of an LFSR allows it to be used as a clock divider or as a counter when a non-binary sequence is acceptable, as is often the case where computer index or framing locations need to be machine-readable.[12] LFSR counters have simpler feedback logic than natural binary counters or Gray-code counters, and therefore can operate at higher clock rates. However, it is necessary to ensure that the LFSR never enters an all-zeros state, for example by presetting it at start-up to any other state in the sequence. The table of primitive polynomials shows how LFSRs can be arranged in Fibonacci or Galois form to give maximal periods. One can obtain any other period by adding to an LFSR that has a longer period some logic that shortens the sequence by skipping some states.

Uses in cryptography

LFSRs have long been used as pseudo-random number generators for use in stream ciphers, due to the ease of construction from simple electromechanical or electronic circuits, long periods, and very uniformly distributed output streams. However, an LFSR is a linear system, leading to fairly easy cryptanalysis. For example, given a stretch of known plaintext and corresponding ciphertext, an attacker can intercept and recover a stretch of LFSR output stream used in the system described, and from that stretch of the output stream can construct an LFSR of minimal size that simulates the intended receiver by using the Berlekamp-Massey algorithm. This LFSR can then be fed the intercepted stretch of output stream to recover the remaining plaintext.

Three general methods are employed to reduce this problem in LFSR-based stream ciphers:

- Non-linear combination of several bits from the LFSR state;
- Non-linear combination of the output bits of two or more LFSRs (see also: shrinking generator); or using Evolutionary algorithm to introduce non-linearity.[13]
- Irregular clocking of the LFSR, as in the alternating step generator.

Important LFSR-based stream ciphers include A5/1 and A5/2, used in GSM cell phones, E0, used in Bluetooth, and the shrinking generator. The A5/2 cipher has been broken and both A5/1 and E0 have serious weaknesses.[14][15]

The linear feedback shift register has a strong relationship to linear congruential generators.[16]

Uses in circuit testing

LFSRs are used in circuit testing for test-pattern generation (for exhaustive testing, pseudo-random testing or pseudo-exhaustive testing) and for signature analysis.

Test-pattern generation

Complete LFSR are commonly used as pattern generators for exhaustive testing, since they cover all possible inputs for an *n*-input circuit. Maximal-length LFSRs and weighted LFSRs are widely used as pseudo-random test-pattern generators for pseudo-random test applications.

Signature analysis

In built-in self-test (BIST) techniques, storing all the circuit outputs on chip is not possible, but the circuit output can be compressed to form a signature that will later be compared to the golden signature (of the good circuit) to detect faults. Since this compression is lossy, there is always a possibility that a faulty output also generates the same signature as the golden signature and the faults cannot be detected. This condition is called error masking or aliasing. BIST is accomplished with a multiple-input signature register (MISR or MSR), which is a type of LFSR. A standard LFSR has a single XOR or XNOR gate, where the input of the gate is connected to several "taps" and the output is connected to the input of the first flip-flop. A MISR has the same structure, but the input to every flip-flop is fed through an XOR/XNOR gate. For example, a 4-bit MISR has a 4-bit parallel output and a 4-bit parallel input. The input of the first flip-flop is XOR/XNORd with parallel input bit zero and the "taps". Every other flip-flop input is XOR/XNORd with the preceding flip-flop output and the corresponding parallel input bit. Consequently, the next state of the MISR depends on the last several states opposed to just the current state. Therefore, a MISR will always generate the same golden signature given that the input sequence is the same every time. Recent applications[17] are proposing set-reset flip-flops as "taps" of the LFSR. This allows the BIST system to optimise storage, since set-reset flip-flops can save the initial seed to generate the whole stream of bits from the LFSR. Nevertheless, this requires changes in the architecture of BIST, is an option for specific applications.

Uses in digital broadcasting and communications

Scrambling

To prevent short repeating sequences (e.g., runs of 0s or 1s) from forming spectral lines that may complicate symbol tracking at the receiver or interfere with other transmissions, the data bit sequence is combined with the output of a linear-feedback register before modulation and transmission. This scrambling is removed at the receiver after demodulation. When the LFSR runs at the same bit rate as the transmitted symbol stream, this technique is referred to as scrambling. When the LFSR runs considerably faster than the symbol stream, the LFSR-generated bit sequence is called *chipping code*. The chipping code is combined with the data using exclusive or before transmitting using binary phase-shift keying or a similar modulation method. The resulting signal has a higher bandwidth than the data, and therefore this is a method of spread-spectrum communication. When used only for the spread-spectrum property, this technique is called direct-sequence spread spectrum; when used to distinguish several signals transmitted in the same channel at the same time and frequency, it is called code-division multiple access.

Neither scheme should be confused with encryption or encipherment; scrambling and spreading with LFSRs do *not* protect the information from eavesdropping. They are instead used to produce equivalent streams that possess convenient engineering properties to allow robust and efficient modulation and demodulation.

Digital broadcasting systems that use linear-feedback registers:

- ATSC Standards (digital TV transmission system – North America)
- DAB (Digital Audio Broadcasting system – for radio)
- DVB-T (digital TV transmission system – Europe, Australia, parts of Asia)
- NICAM (digital audio system for television)

Other digital communications systems using LFSRs:

- Intelsat business service (IBS)
- Intermediate data rate (IDR)
- HDMI 2.0
- SDI (Serial Digital Interface transmission)
- Data transfer over PSTN (according to the ITU-T V-series recommendations)
- CDMA (Code Division Multiple Access) cellular telephony
- 100BASE-T2 "fast" Ethernet scrambles bits using an LFSR
- 1000BASE-T Ethernet, the most common form of Gigabit Ethernet, scrambles bits using an LFSR
- PCI Express
- SATA[18]
- Serial Attached SCSI (SAS/SPL)
- USB 3.0
- IEEE 802.11a scrambles bits using an LFSR
- Bluetooth Low Energy Link Layer is making use of LFSR (referred to as whitening)
- Satellite navigation systems such as GPS and GLONASS. All current systems use LFSR outputs to generate some or all of their ranging codes (as the chipping code for CDMA or DSSS) or to modulate the carrier without data (like GPS L2 CL ranging code). GLONASS also uses frequency-division multiple access combined with DSSS.

Other uses

LFSRs are also used in radio jamming systems to generate pseudo-random noise to raise the noise floor of a target communication system.

The German time signal DCF77, in addition to amplitude keying, employs phase-shift keying driven by a 9-stage LFSR to increase the accuracy of received time and the robustness of the data stream in the presence of noise.[19]

See also

- Pinwheel
- Mersenne twister
- Maximum length sequence
- Analog feedback shift register
- NLFSR, Non-Linear Feedback Shift Register
- Ring counter
- Pseudo-random binary sequence
- Gold sequence
- JPL sequence
- Kasami sequence
- Berlekamp–Massey algorithm

References

1. Geremia, Patrick. "Cyclic Redundancy Check Computation: An Implementation Using the TMS320C54x" (https://www.ti.com/lit/an/spra530/spra530.pdf) (PDF). Texas Instruments. p. 6. Retrieved October 16, 2016.

2. Linear Feedback Shift Registers in Virtex Devices (http://www.xilinx.com/support/documentation/application_notes/xapp210.pdf)
3. Gentle, James E. (2003). *Random number generation and Monte Carlo methods* (https://www.worldcat.org/oclc/51534945) (2nd ed.). New York: Springer. p. 38. ISBN 0-387-00178-6. OCLC 51534945 (https://www.worldcat.org/oclc/51534945).
4. Tausworthe, Robert C. (April 1965). "Random Numbers Generated by Linear Recurrence Modulo Two" (https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0184406-1/S0025-5718-1965-0184406-1.pdf) (PDF). *Mathematics of Computation*. **19** (90): 201–209. doi:10.1090/S0025-5718-1965-0184406-1 (https://doi.org/10.1090%2FS0025-5718-1965-0184406-1). S2CID 120804149 (https://api.semanticscholar.org/CorpusID:120804149).
5. Press, William; Teukolsky, Saul; Vetterling, William; Flannery, Brian (2007). *Numerical Recipes: The Art of Scientific Computing, Third Edition*. Cambridge University Press. p. 386. ISBN 978-0-521-88407-5.
6. Marsaglia, George (July 2003). "Xorshift RNGs" (https://www.jstatsoft.org/v08/i14/paper). *Journal of Statistical Software*. **8** (14). doi:10.18637/jss.v008.i14 (https://doi.org/10.18637%2Fjss.v008.i14).
7. Brent, Richard P. (August 2004). "Note on Marsaglia's Xorshift Random Number Generators" (https://www.jstatsoft.org/v11/i05/paper). *Journal of Statistical Software*. **11** (5). doi:10.18637/jss.v011.i05 (https://doi.org/10.18637%2Fjss.v011.i05). hdl:1885/34049 (https://hdl.handle.net/1885%2F34049).
8. Metcalf, John (22 July 2017). "16-Bit Xorshift Pseudorandom Numbers in Z80 Assembly" (http://www.retroprogramming.com/2017/07/xorshift-pseudorandom-numbers-in-z80.html). *Retro Programming*. Retrieved 5 January 2022.
9. Klein, A. (2013). "Linear Feedback Shift Registers". *Stream Ciphers*. London: Springer. pp. 17–18. doi:10.1007/978-1-4471-5079-4_2 (https://doi.org/10.1007%2F978-1-4471-5079-4_2). ISBN 978-1-4471-5079-4.
10. Golomb, Solomon W. (1967). *Shift register sequences*. Laguna Hills, Calif.: Aegean Park Press. ISBN 978-0894120480.
11. Weisstein, Eric W. "Primitive Polynomial" (https://mathworld.wolfram.com/PrimitivePolynomial.html). *mathworld.wolfram.com*. Retrieved 2021-04-27.
12. http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf
13. A. Poorghanad, A. Sadr, A. Kashanipour" Generating High Quality Pseudo Random Number Using Evolutionary Methods", IEEE Congress on Computational Intelligence and Security, vol. 9, pp. 331-335, May, 2008 [1] (http://www.computer.org/csdl/proceedings/cis/2008/3508/01/3508a331.pdf)
14. Barkam, Elad; Biham, Eli; Keller, Nathan (2008), "Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication" (https://web.archive.org/web/20200125081932/http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2006/CS/CS-2006-07.pdf) (PDF), *Journal of Cryptology*, **21** (3): 392–429, doi:10.1007/s00145-007-9001-y (https://doi.org/10.1007%2Fs00145-007-9001-y), S2CID 459117 (https://api.semanticscholar.org/CorpusID:459117), archived from the original (https://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2006/CS/CS-2006-07.pdf) (PDF) on 2020-01-25, retrieved 2019-09-15
15. Lu, Yi; Willi Meier; Serge Vaudenay (2005). "The Conditional Correlation Attack: A Practical Attack on Bluetooth Encryption". *Advances in Cryptology – CRYPTO 2005* (http://www.terminodes.org/micsPublicationsDetail.php?pubno=1216). Lecture Notes in Computer Science. Vol. 3621. Santa Barbara, California, USA. pp. 97–117. CiteSeerX 10.1.1.323.9416 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.323.9416). doi:10.1007/11535218_7 (https://doi.org/10.1007%2F11535218_7). ISBN 978-3-540-28114-6. {{cite book}}: |journal= ignored (help)
16. RFC 4086 section 6.1.3 "Traditional Pseudo-random Sequences"
17. Martínez LH, Khursheed S, Reddy SM. LFSR generation for high test coverage and low hardware overhead. IET Computers & Digital Techniques. 2019 Aug 21.UoL repository (https://livrepository.liverpool.ac.uk/3052312/)
18. Section 9.5 of the SATA Specification, revision 2.6
19. Hetzel, P. (16 March 1988). *Time dissemination via the LF transmitter DCF77 using a pseudo-random phase-shift keying of the carrier* (https://www.ptb.de/cms/fileadmin/internet/fachabteilungen/abteilung_4/4.4_zeit_und_frequenz/pdf/5_1988_Hetzel_-_Proc_EFTF_88.pdf) (PDF). 2nd European Frequency and Time Forum. Neuchâtel. pp. 351–364. Retrieved 11 October 2011.

Further reading

- http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf
- https://web.archive.org/web/20161007061934/http://courses.cse.tamu.edu/csce680/walker/lfsr_table.pdf
- http://users.ece.cmu.edu/~koopman/lfsr/index.html — Tables of maximum length feedback polynomials for 2-64 bits.
- https://github.com/hayguen/mlpolygen — Code for generating maximal length feedback polynomials

External links

- Linear Feedback Shift Registers (https://web.archive.org/web/20181001062252/http://www.newwaveinstruments.com:80/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm) at the Wayback Machine (archived October 1, 2018) – LFSR theory and implementation, maximal length sequences, and comprehensive feedback tables for lengths from 7 to 16,777,215 (3 to 24 stages), and partial tables for lengths up to 4,294,967,295 (25 to 32 stages).
- International Telecommunication Union Recommendation O.151 (http://www.itu.int/rec/T-REC-O.151-199210-I/en) (August 1992)
- Maximal Length LFSR table (https://spreadsheets.google.com/ccc?key=0AvYtZsho-JTldFRYZnJLRFFaSWtUcVNXc1Y3M2VWd1E&hl=en) with length from 2 to 67.
- Pseudo-Random Number Generation Routine for the MAX765x Microprocessor (https://www.maximintegrated.com/en/design/technical-documents/app-notes/1/1743.html)
- http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/Drivers_Ed/lfsr.html
- http://www.quadibloc.com/crypto/co040801.htm
- Simple explanation of LFSRs for Engineers (https://web.archive.org/web/20060315203220/http://www.yikes.com/~ptolemy/lfsr_web/index.htm)
- Feedback terms (http://www.ece.cmu.edu/~koopman/lfsr/index.html)
- General LFSR Theory (https://web.archive.org/web/20060111183721/http://homepage.mac.com/afj/lfsr.html)
- An implementation of LFSR in VHDL. (http://opencores.org/project,lfsr_randgen)
- Simple VHDL coding for Galois and Fibonacci LFSR. (http://emmanuel.pouly.free.fr)
- mlpolygen: A Maximal Length polynomial generator (https://bitbucket.org/gallen/mlpolygen)
- LFSR and Intrinsic Generation of Randomness: Notes From NKS (https://www.wolframscience.com/nks/notes-7-5--random-number-generators/)