Tutorial: Linear Feedback Shift Registers (LFSRs) - Part 3

Clive (Max) Maxfield - January 03, 2007

Editors Note: This is the third and final portion of our introduction to Linear Feedback Shift Registers (LFSRs). These articles are abstracted from the book Bebop to the Boolean Boogie (An Unconventional Guide to Electronics) with the kind permission of the publisher. (See also Part 1 and Part 2.)

Encryption and decryption applications
The unusual sequence of values generated by an LFSR can be gainfully employed in the encryption (scrambling) and decryption (unscrambling) of data. A stream of data bits can be encrypted by XOR-ing them with the output from an LFSR (Fig 1).

![Diagram of encryption using an LFSR]

1. Data encryption using an LFSR.

The stream of encrypted data bits seen by a receiver can be decrypted by XOR-ing them with the output of an identical LFSR. This is obviously a very trivial form of encryption that’s not very secure, but it’s "cheap-and-cheerful" and may be useful in certain applications.

Cyclic redundancy check (CRC) applications
A traditional application for LFSRs is in cyclic redundancy check (CRC) calculations, which can be used to detect errors in data communications. The stream of data bits being transmitted is used to modify the values fed back into an LFSR (Fig 2).

2. Cyclic redundancy check (CRC) calculations.

The final CRC value stored in the LFSR is known as a checksum, and is dependent on every bit in the data stream. After all of the data bits have been transmitted, the transmitter sends its checksum value to the receiver. The receiver contains an identical CRC calculator and generates its own checksum value from the incoming data. Once all of the data bits have arrived, the receiver compares its internally generated checksum value with the checksum sent by the transmitter to determine whether any corruption occurred during the course of the transmission. This form of error detection is very efficient in terms of the small number of bits that have to be transmitted in addition to the data.

In the real world, a 4-bit CRC calculator would not be considered to provide sufficient confidence in the integrity of the transmitted data. This is due to the fact that a 4-bit LFSR can only represent 16 unique values, which means that there is a significant probability that multiple errors in the data stream could result in the two checksum values being identical. However, as the number of bits in a CRC calculator increases, the probability that multiple errors will cause identical checksum values approaches zero. For this reason, CRC calculators typically use a minimum of 16-bits providing 65,536 unique values.

There are a variety of standard communications protocols, each of which specifies the number of bits employed in their CRC calculations and the taps to be used. The taps are selected such that an error in a single data bit will cause the maximum possible disruption to the resulting checksum value. Thus, in addition to being referred to as maximal-length, these LFSRs may also be qualified as maximal-displacement.

Editor's Note: Reader Jay Dowling, co-owner of StereoImaging Corporation, has a Yahoo Group from whence he provides Verilog design files and links. Amongst many other things, here you'll find information to help you determine appropriate taps for your LFSR applications. Jay points out that even though some tap combinations will generate maximum sequences, they may not be "good" because they don't statistically appear random and/or have other undesirable characteristics. Jay's site provides tables of all the values you might want to use in the file called LFSR_table.pdf.

In addition to checking data integrity in communications systems, CRCs find a wide variety of other uses: for example, the detection of computer viruses. For the purposes of this discussion, a computer virus may be defined as a self-replicating program released into a computer system for a number of purposes. These purposes range from the simply mischievous (such as displaying humorous or annoying messages) to the downright nefarious (such as corrupting data or destroying – or
One mechanism by which a computer virus may both hide and propagate itself is to attach itself to an existing program. A cursory check of the system shows only the expected files to be present, but whenever the infected program is executed it will first trigger the virus to replicate itself. In order to combat this form of attack, a unique checksum can be generated for each file on the system, where the value of each checksum is based on the binary instructions forming the program with which it is associated. At some later date, an anti-virus program can be used to recalculate the checksum values for each program, and to compare them to the original values. A difference in the two values associated with a program may indicate that a virus has attached itself to that program.

**Editor's Note:** Unfortunately, the creators of computer viruses can be rather sophisticated, and some viruses are armed with the ability to perform their own CRC calculations. When a virus of this type attaches itself to a program, it can pad itself with dummy binary values, which are selected so as to cause an anti-virus program to return an identical checksum value to the original. And so it goes . . .

**Data compression applications**
The CRC calculators discussed above can also be used in a data compression role. One such application is found in the circuit board test strategy known as functional test. In this case, the board is plugged into a functional tester by means of its edge connector. The tester applies a pattern of signals to the board's inputs, allows sufficient time for any effects to propagate around the board, and then compares the actual values seen on the outputs with a set of expected values stored in the system. This process is repeated for a series of input patterns, which may number in the tens or hundreds of thousands.

The illustration above is simplified for reasons of clarity. In practice, the edge connector may contain hundreds of pins while the board may contain thousands of components and tens of thousands of tracks. If the board fails the preliminary tests, a more sophisticated form of analysis known as guided-probe may be employed to identify the cause of the failure.

In this case, the tester instructs the operator to place the probe at a particular location on the board, and then the entire sequence of test patterns is rerun. The tester compares the actual sequence of values seen by the probe with a sequence of expected values that are stored in the system. This process (placing the probe and running the tests) is repeated until the tester has isolated the faulty component or track.

A major consideration when supporting a guided probe strategy is the amount of expected data that must be stored. Consider a test sequence comprising 10,000 patterns driving a board containing 10,000 tracks. If the data were not compressed, the system would have to store 10,000 bits of expected data per track, which amounts to 100,000,000 bits of data for the board. Additionally, for each application of the guided probe, the tester would have to compare the 10,000 data bits observed by the probe with the 10,000 bits of expected data stored in the system. Thus, using data in an uncompressed form is an expensive option in terms of storage and processing requirements.

One solution to these problems is to employ LFSR-based CRC calculators. The sequence of expected values for each track can be passed through a 16-bit CRC calculator implemented in software. Similarly, the sequence of actual values seen by the guided probe can be passed through an identical CRC calculator implemented in hardware. In this case, the calculated checksum values are also known as signatures, and a guided probe process based on this technique is known as signature analysis. Irrespective of the number of test patterns used, the system has to store only two bytes of
data for each track. Additionally, for each application of the guided probe, the tester has to compare only the two bytes of data gathered by the probe with two bytes of expected data stored in the system. Thus, compressing the data results in storage requirements that are orders of magnitude smaller and comparison times that are orders of magnitude faster than the uncompressed data approach. **Built-in self-test (BIST) applications**

One test strategy which may be employed in complex integrated circuits is that of **built-in self-test (BIST)**. Devices using BIST contain special test generation and result gathering circuits. A multiplexer is used to select between the standard inputs and those from the *test generator*. A second multiplexer selects between the standard outputs and those from the *results gatherer*.

Both the test generator and results gatherer can be implemented using LFSRs (Fig 3). (The standard inputs and outputs along with *their* multiplexers have been omitted from this illustration for reasons of clarity.)

![BIST implemented using LFSRs](image)

The LFSR forming the test generator is used to create a sequence of test patterns, while the LFSR forming the results gatherer is used to capture the results. The results-gathering LFSR features modifications that allow it to accept parallel data.

Additional circuitry would be required to provide a way to load new seed values into the test generator and to access the final values in the results gatherer. This logic is not shown here purposes of simplicity.

Note that the two LFSRs are not obliged to contain the same number of bits, because the number of inputs to the logic being tested may be different to the number of outputs from the logic.

Also note that all of the flip-flops in the test generator would share a common clock. Similarly, all of the flip-flops would also share a common clock. These two clocks might be common or they might be distinct (in the latter case they would be synchronized in some way). The *clock* signals have been
omitted from *Fig 3* so as to keep things simple.

**Pseudo-random number applications**

Many computer programs rely on an element of randomness. Computer games such as "Space Invaders" employ random events to increase the player's enjoyment. Graphics programs may exploit random numbers to generate intricate patterns. All forms of computer simulation may utilize random numbers to more accurately represent the real world. For example, digital simulations may benefit from the portrayal of random stimulus such as external interrupts. Random stimulus can result in more realistic design verification, which can uncover problems that may not be revealed by more structured tests.

*Editor's Note:* Digital simulation is based on a program called a *simulator*, which is used to build a virtual representation of an electronic design in the computer's memory. The simulator then applies stimulus to the design's virtual inputs, simulates the effect of these signals as they propagate through the design, and checks the responses at the design's virtual outputs.

Random number generators can be constructed in both hardware and software. The majority of these generators are not truly random, but they give the appearance of being random and are therefore said to be *pseudo-random*. In fact, pseudo-random numbers have an advantage over truly random numbers, because the majority of computer applications typically require repeatability. For example, a designer repeating a digital simulation would expect to receive identical answers to those from the previous run. However, designers also need the ability to modify the seed value of the pseudo-random number generator so as to spawn different sequences of values as required.

There are a variety of methods available for generating pseudo-random numbers. A popular "cheap-and-cheerful" technique uses the remainder from a division operation as the next value in a pseudo-random sequence. For example, a C function which returns a pseudo-random number in the range 0 to 32767 could be written as follows [This example is from *The C Programming Language (Second Edition)* by Brian W. Kernighan and Dennis M. Ritchie]:

```
int rand ()
{
    next = next * 1103515245 + 12345;
    return (unsigned int) (next/65535) % 32768;
}
```

Note that the variable `next` would be declared as global and initialized with some seed value. Also, an additional function would be provided to allow the programmer to load `next` with a new seed value if required. Every time this function is accessed, it returns a new pseudo-random value. Unfortunately, pseudo-random implementations based on division do not always produce a "white spectrum". The series of generated values may be composed of a collection of sub-series, in which case the results will not be of maximal length. By comparison, the sequence of values generated by a software implementation of a maximal-length LFSR provides a reasonably good pseudo-random source, but is somewhat more expensive in terms of processing requirements.

A simple technique for checking the "randomness" of a pseudo-random number generator is as follows. Assume the existence of a function called `rand()` that returns a pseudo-random number as a 16-bit value. The most significant and least significant bytes may be considered to represent the y-axis and x-axis of a graph respectively. To test the randomness of the function, a simple C program segment may be created to plot a reasonable sample of random numbers:
int x_axis, y_axis, i, my_number; /* Declare the variables. */

for (i = 1; i <= 1000; i++) /* perform a loop 1,000 times. */
{
    my_number = rand(); /* Get a pseudo-random number. */
    x_axis = my_number & 0x00FF; /* Extract the least-significant byte. */
    y_axis = (my_number & 0xFF00) >> 8; /* Extract the most-significant byte. */
    plot(x_axis, y_axis); /* Plot the value. */
}

This example assumes the existence of a function plot() that will cause the point represented by the variables x_axis and y_axis to be displayed on a computer screen.

If the resulting points are uniformly distributed across the plotting area as illustrated in Fig 4(a), then the pseudo-random generator is acceptable. If, however, the points form clusters or are grouped around one of the diagonal axes as illustrated in Fig 4(b), then the generator is less than optimal.
4. Checking for "randomness".

Last but not least . . .

LFSRs are simple to construct and are useful for a wide variety of applications, but be warned that choosing the optimal polynomial (which ultimately boils down to selecting the tap points) for a particular application is a task that is usually reserved for a master of the mystic arts, not to mention that the math involved can be "hairy" enough to make a grown man break down and cry. And don’t even get the author started on the subject of cyclotomic polynomials, which are key to the tap-selection process (the reason for saying this is that the author hasn't got the faintest clue what a cyclotomic polynomial is!).

Clive "Max" Maxfield is president of TechBites Interactive, a marketing consultancy firm specializing in high technology. Max is the author and co-author of a number of books,
including Bebop to the Boolean Boogie (An Unconventional Guide to Electronics), The Design Warrior's Guide to FPGAs (Devices, Tools, and Flows), and How Computers Do Math featuring the pedagogical and phantasmagorical virtual DIY Calculator.

Widely regarded as being an expert in all aspects of computing and electronics (at least by his mother), Max was once referred to as "an industry notable" and a "semiconductor design expert" by someone famous who wasn't prompted, coerced, or remunerated in any way. Max can be reached at max@techbites.com.