Editors Note: This is the second in a three-part 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 3.)

Seeding an LFSR
One quirk with an XOR-based LFSR is that, if it happens to find itself in the all 0s value, it will happily continue to shift all 0s indefinitely (similarly for an XNOR-based LFSR and the all 1s value). This is of particular concern when power is first applied to the circuit. Each register bit can randomly initialize containing either a logic 0 or a logic 1, and the LFSR can therefore "wake up" containing its "forbidden" value. For this reason, it is necessary to initialize an LFSR with a seed value.

One method for loading a seed value is to use registers with reset or set inputs. A single control signal can be connected to the reset inputs on some of the registers and the set inputs on others. When this control signal is placed in its active state, the LFSR will load with a hard-wired seed value. In certain applications, however, it is desirable to be able to vary the seed value. One technique for achieving this is to include a multiplexer at the input to the LFSR (Fig 1).

When the multiplexer's data input is selected, the device functions as a standard shift register and any desired seed value may be loaded. After loading the seed value, the feedback path is selected and the device returns to its LFSR mode of operation.

FIFO Applications
The fact that an LFSR generates an unusual sequence of values is irrelevant in many applications. Consider a 4-bit × 16-word first-in first-out (FIFO) memory device (Fig 2).
2. First-in first-out (FIFO) memory.

A brief summary of the FIFO's operation is as follows. The write and read pointers are essentially 4-bit registers whose outputs are processed by 4:16 decoders to select one of the sixteen words in the memory array. The reset input is used to initialize the device, primarily by clearing the write and read pointers such that they both point to the same memory word. The initialization also causes the empty output to be placed in its active state and the full output to be placed in its inactive state.

The write and read pointers chase each other around the memory array in an endless loop. An active edge on the write input causes any data on the data-in[3:0] bus to be written into the word pointed to by the write pointer; the empty output is placed in its inactive state (because the device is no longer empty) and the write pointer is incremented to point to the next empty word.

Data can be written into the FIFO until all the words in the array contain values. When the write pointer catches up to the read pointer, the full output is placed in its active state (indicating that the device is full) and no more data can be written into the device.

An active edge on the read input causes the data in the word pointed to by the read pointer to be copied into the output register; the full output is placed in its inactive state and the read pointer is incremented to point to the next word containing data. (These discussions assume write-and-increment and read-and-increment techniques, but it should be noted that some FIFOs employ an increment-and-write and increment-and-read approach.)

Data can be read out of the FIFO until the array is empty. When the read pointer catches up to the write pointer, the empty output is placed in its active state, and no more data can be read out of the device.

The write and read pointers for a 16-word FIFO are often implemented using 4-bit binary counters. However, a moment's reflection reveals that there is no intrinsic advantage to a binary sequence for this particular application, and the sequence generated by a 4-bit LFSR would serve equally well. In fact, the two functions operate in a very similar manner as is illustrated by their block diagrams (Fig 3).
However, the combinational logic for the 4-bit LFSR consists of a single, two-input XOR gate, while the combinational logic for the 4-bit binary counter requires a number of AND and OR gates. This means that the LFSR requires fewer tracks and is significantly more efficient in terms of silicon real estate. Additionally, the LFSR's feedback only passes through a single level of logic, while the binary counter's feedback passes through multiple levels of logic. This means that the new data value is available sooner for the LFSR, which can therefore be clocked at a higher frequency. These differentiations become even more pronounced for FIFOs with more words requiring pointers with more bits. Thus, LFSR's are an option that should at least be considered for the discerning designer of FIFOs.

**Editor's Note:** As opposed to using LFSRs for this type of application, Reader Jay Dowling, co-owner of StereoImaging Corporation, uses Gray code counters in many of his designs. Jay has compiled a Very Interesting Report (presented in the form of an Excel spreadsheet) that compares a variety of counter implementations in this context, including binary, Gray code, and two different types of LFSR counter. (Jay also has a Yahoo Group from whence he provides Verilog design files and links.)

Furthermore, if you visit Jay's yahoo group, you'll find information to help you determine appropriate taps for your LFSR applications. Jay points out that although 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.

**Modifying LFSRs to sequence through \(2^n\) values**

The sole downside to the 4-bit LFSRs in the FIFO scenario above is that they will sequence through only \((2^4 - 1) = 15\), as compared to a binary counter's sequence of \(2^4 = 16\) values. Designers may not regard this to be a problem, especially in the case of larger FIFOs. However, if it is required for an LFSR to sequence through every possible value, there is a simple solution (Fig 4).
4. LFSR modified to sequence $2^n$ values.

For the value where all of the bits are 0 to appear, the preceding value must have comprised a logic 1 in the most-significant bit (MSB) and logic 0s in the remaining bit positions. (As is often the case with any form of shift register, the MSB in these examples is taken to be on the right-hand side of the register, and the LSB is taken to be on the left-hand side, which is opposite to the way we usually do things.) In an unmodified LFSR, the next clock would result in a logic 1 in the least-significant bit (LSB) and logic 0s in the remaining bit positions.

In the case of the modified LFSR, the output from the NOR is logic 0 for every case but two: the value preceding the one where all the bits are 0 and the value where all the bits are 0. These two values force the NOR's output to a logic 1, which inverts the usual output from the XOR. This in turn causes the sequence to first enter the all-0s value and then resume its normal course. (In the case of LFSRs with XNOR feedback paths, the NOR can be replaced with an AND, which causes the sequence to cycle through the value where all of the bits are 1.)

Accessing the previous value

In some applications it is required to make use of a register's previous value. For example, in certain FIFO implementations, the "full" condition is detected when the write pointer is pointing to the location preceding the location pointed to by the read pointer (try saying that quickly!). This implies that a comparator must be used to
compare the current value in the write pointer with the previous value in the read pointer. Similarly, the "empty" condition may be detected when the read pointer is pointing to the location preceding the location pointed to by the write pointer. This implies that a second comparator must be used to compare the current value in the read pointer with the previous value in the write pointer.

In the case of binary counters (assuming that, for some reason, we decided to use them for a FIFO application), there are two techniques by which the previous value in the sequence may be accessed. The first requires the provision of an additional set of so-called shadow registers. At the same time the counter is incremented, its current contents are also copied into the shadow registers. Alternatively, a block of combinational logic can be used to decode the previous value from the current value. Unfortunately, both of these techniques involve a substantial overhead in terms of additional logic. By comparison, LFSRs of the "many-to-one" flavor (as discussed in Part 1) inherently remember their previous value. All that is required is the addition of a single register bit appended to the MSB (Fig 5).

5. Accessing an LFSR's previous value.

But wait, there's more
In Part 3 of these series we shall consider a variety of applications for which LFSRs may of interest, including encryption and decryption, cyclic redundancy checks (CRCs), data compression, built-in self-test (BIST), and pseudo-random number generation.

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.