XAPP220 (v1.1) January 11, 2001

LFSRs as Functional Blocks in Wireless Applications

Author: Stephen Lim and Andy Miller

Summary

Linear Feedback Shift Registers (LFSRs) are commonly used in applications where pseudo-random bit streams are required. LFSRs are the functional building blocks of circuits like the pseudo-random noise (PN) code generator (XAPP211) and Gold code generators (XAPP217) commonly used in Code Division Multiple Access (CDMA) systems. This application note describes two implementations of an LFSR using the SRL16 (Shift Register Look-Up Table) primitive for area-efficient designs. The first LFSR implementation describes the parallel output access and parity calculation; the second describes the multi-cycle output access and sequential parity calculation. This application note covers the Virtex™ series, the Virtex-II series and the Spartan™-II family of devices.

Introduction

LFSRs can be used for performance-critical binary counters used to generate sequences of random numbers. LFSRs will often satisfy this requirement, although the generated sequence is pseudo-random in nature. Pseudo-random patterns repeat over time; the longer the LFSR, however, the longer the sequence of random numbers before pattern repetition occurs.

When longer sequences are desired, the physical size of the hardware is increased. Conventionally, in the older FPGA architectures, flip-flops would be used. With two flip-flops in each of the older-architecture CLBs, an n-bit LFSR will take up at least n/2 CLBs. In the Virtex, Virtex-E, Virtex-EM, Virtex-II, and Spartan-II architectures, a four-input LUT can also function as a 16-bit shift register with a single output accessed by the LUT's address lines. This 16-bit shift register function can be accessed using the SRL16 primitive. An LFSR implemented using these SRL16 primitives reduces FPGA resource utilization compared to implementations using flip-flops.

Linear Feedback Shift Registers

LFSRs sequence through $2^N - 1$ states, where N is the number of flip-flops in the LFSR. At each clock edge, the contents of the flip-flops are shifted right by one position. There is a feedback path from predefined flip-flops to the leftmost flip-flop through an exclusive-NOR (XNOR) or an exclusive-OR (XOR) gate. A value of all "1's" is illegal in the case of an XNOR feedback, and a value of all "0's" is illegal for XOR feedback. The illegal state causes the counter to remain in its present state, locking out any further new values from being registered.

Because of the pseudo-random nature of LFSRs, they are used as basic functional blocks in Gold code generators (Figure 1). The feedback taps are predefined mathematically to assure that the LFSR shifts through the maximum number of possible values. The taps for up to 168 bits are detailed in XAPP210 (Table 1). The following section discusses some of these basic concepts in detail.

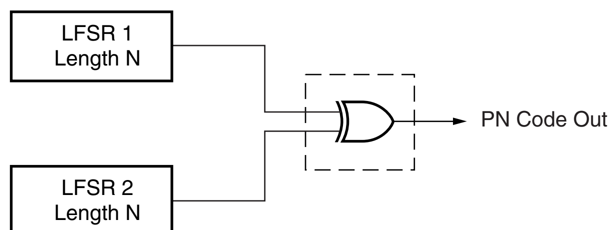

Figure 1: **Gold Code Generator Using LFSRs**

© 2000 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and disclaimers are as listed at http://www.xilinx.com/legal.htm.
All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

LFSR Terminology

LFSRs sequence through ($2^N - 1$) states, where N is the number of registers in the LFSR. The contents of the registers are shifted right by one position at each clock cycle. The feedback from predefined registers or taps to the leftmost register are XORed together.

LFSRs have several variables:

- The number of stages in the shift register
- The number of taps in the feedback path
- The position of each tap in the shift register stage
- The initial starting condition of the shift register, often referred to as the FILL state

  NOTE: In the case of LFSRs with an XOR feedback, the FILL value must be non-zero to avoid the LFSR locking up in the next state.

Shift Register Length (*N*)

The shift register length is often referred to as the *degree*, and the longer the shift register, the longer the duration of the PN sequence before it repeats. For a shift register of fixed length N, the number and duration of the sequences it can generate are determined by the number and position of taps used to generate the parity feedback bit.

Shift Register Taps

The combination of taps and their location is often referred to as a polynomial, and expressed as:

$$P(x) = X^7 + X^3 + 1$$

Various conventions are used to map the polynomial terms to register stages in the shift register implementation. The convention used in this application note is consistent with the convention used in the CDMA UMTS specification.

In the polynomial $P(x) = X^7 + X^3 + 1$, the trailing "1" represents X^0, which is the output of the last stage of the shift register. X^3 is the output of register stage 3 and X^7 the output of the XOR.

A few points to note about LFSRs and the polynomial used to describe them:

- The last tap of the shift register is the leading "1" and is always used in the shift register feedback path.
- The length of the shift register can be deduced from the exponent of the highest order term in the polynomial.
- The highest order term of the polynomial is the signal connecting the final XOR output to the shift register input. It does not feed back into the parity calculation along with the other taps identified in the polynomial.

LFSR Implementation

There are two implementation styles of LFSRs, Galois implementation and Fibonacci implementation.

Galois Implementation

As shown in Figure 2, the data flow is from left to right and the feedback path is from right to left. The polynomial increments from left to right with X^0 term (the "1" in the polynomial) as the first term. This is referred to as a Tap polynomial, as it indicates which taps are to be fed back from the shift register. Since the XOR gate is in the shift register path, the Galois implementation is also known as an *in-line* or modular type (M-type) LFSR.

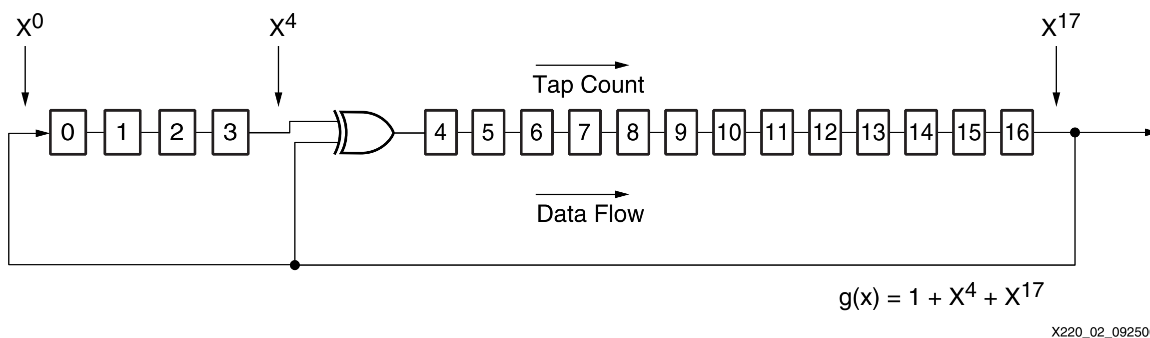

Figure 2: **Galois Implementation**

Fibonacci Implementation

In Figure 3, the data flow is from left to right and the feedback path is from right to left, similar to the Galois implementation. However, the Fibonacci implementation polynomial decrements from left to right with X^0 as the last term in the polynomial. This polynomial is referred to as a Reciprocal Tap polynomial and the feedback taps are incrementally annotated from right to left along the shift register. Since the XOR gate is in the feedback path, the Fibonacci implementation is also known as an *out-of-line* or simple type (S-type) LFSR.

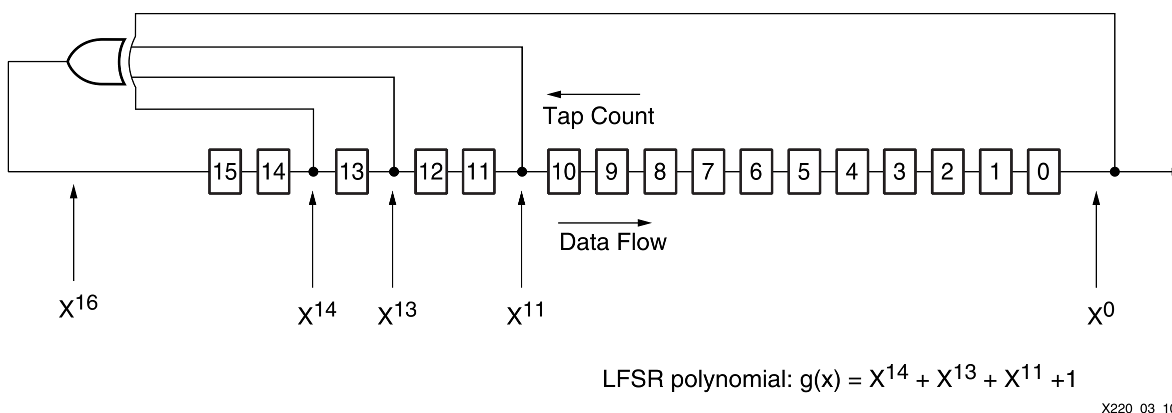

Figure 3: **Fibonacci Implementation**

Maximal Length Sequences (*L*)

A maximal length sequence for a shift register of length *N* is referred to as an m-sequence, and is defined as:

$$L = 2^N - 1$$

An eight-stage LFSR, for example, will have a set of m-sequences of length 255.

Shift Register LUT Mode for Area-Efficient LFSRs

With the SRL16 primitive, it is possible to implement an *n*-bit LFSR in a fraction of the space used by a flip-flop design. A 16-bit LFSR would take up at least eight slices using flip-flops, since there are just two flip-flops per slice. The same 16-bit LFSR can be implemented in just four slices when using the SRL16s. Virtex-II devices have a new macro, SRLC16 in addition to the SRL16/E. Two outputs of the SRLC16 can be accessed simultaneously. One output is determined by the value of the 4-bit address line (i.e., A[3] – A[0]) and the other output is the cascadable output (the 16th bit of the shift register.)

In Virtex devices, only a single tap or output of the SRL16s can be accessed at a time. The SRL16 output to be accessed is determined by the value of the 4-bit address line A[3] – A[0]. The SRL16 primitive will shift data on every clock cycle. A second primitive (SRL16E) provides the same shift register functionality, but adds a shift register Clock Enable.

Both the SRL16 and SRL16E implement area-efficient shift registers in one LUT. It is worth noting, however, that parallel access to multiple taps is not possible, as the primitives have only one data output pin. Thus, for every single output that must be accessed, another SRL16 must be created if that output is in the same 16-bit set as the other. Because the number of taps rarely exceeds four, creating multiple instances of the SRL16 primitive is not a concern.

It should be noted that because flip-flop based LFSRs will only consume as many flip-flops as there are stages in the shift register, the size at which it becomes more area efficient to use flip-flops is less than or equal to eight.

To overcome the loss of parallel access, the following two approaches are reviewed.

Multiple Shift Registers with Parallel Tap Access and Parity Calculation

Figure 4 demonstrates a 16-stage LFSR with four selectable tap points designed with four SRL16 primitives. An additional 4-input LUT is used to implement a parallel XOR parity calculation (Figure 5) that is fed back into the shift register as the new bit in the sequence. Tap D is the last stage in the shift register and so represents the LFSR output. This circuit is clocked at a frequency known as the *chip rate*.

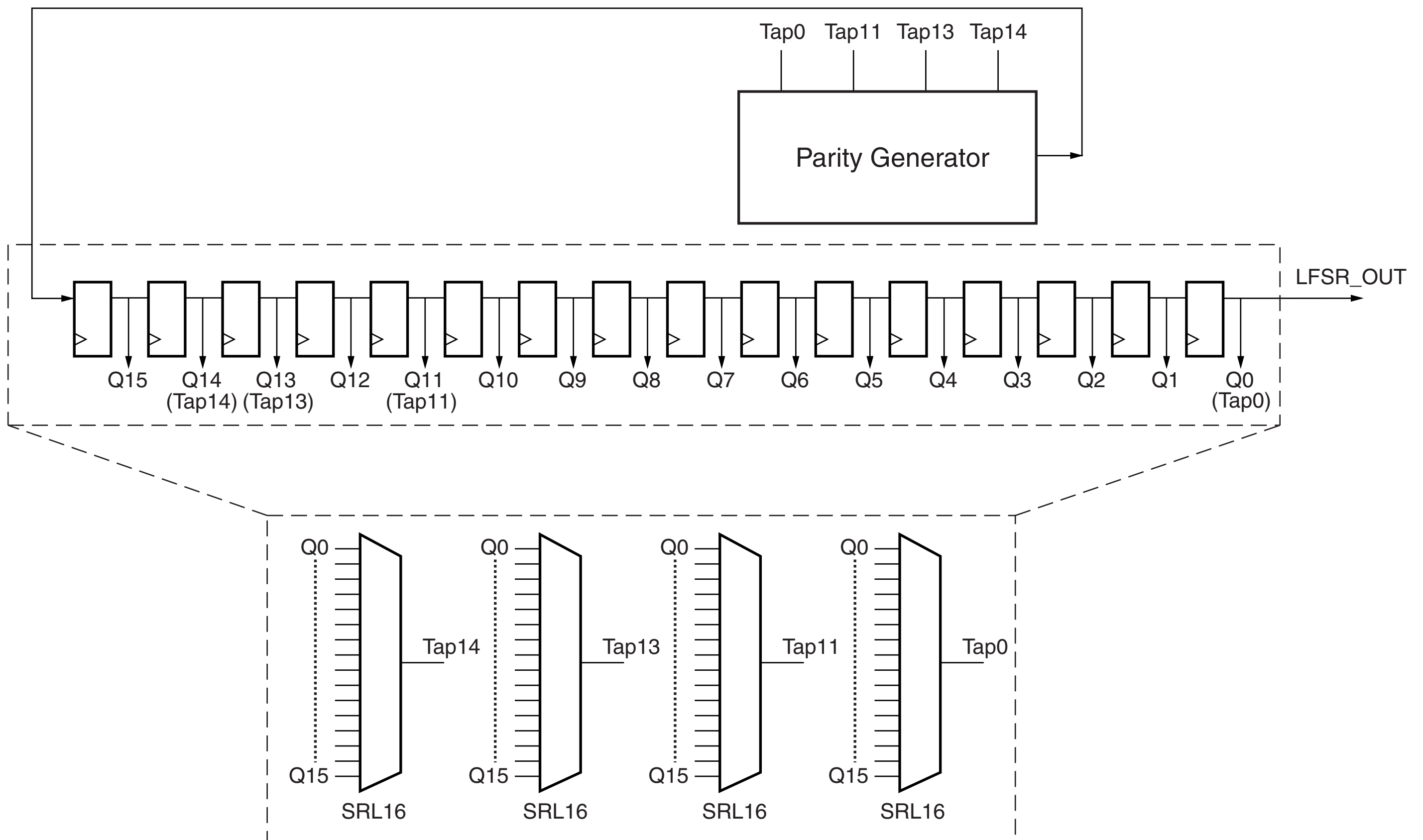

Figure 4: **Parity Calculation**

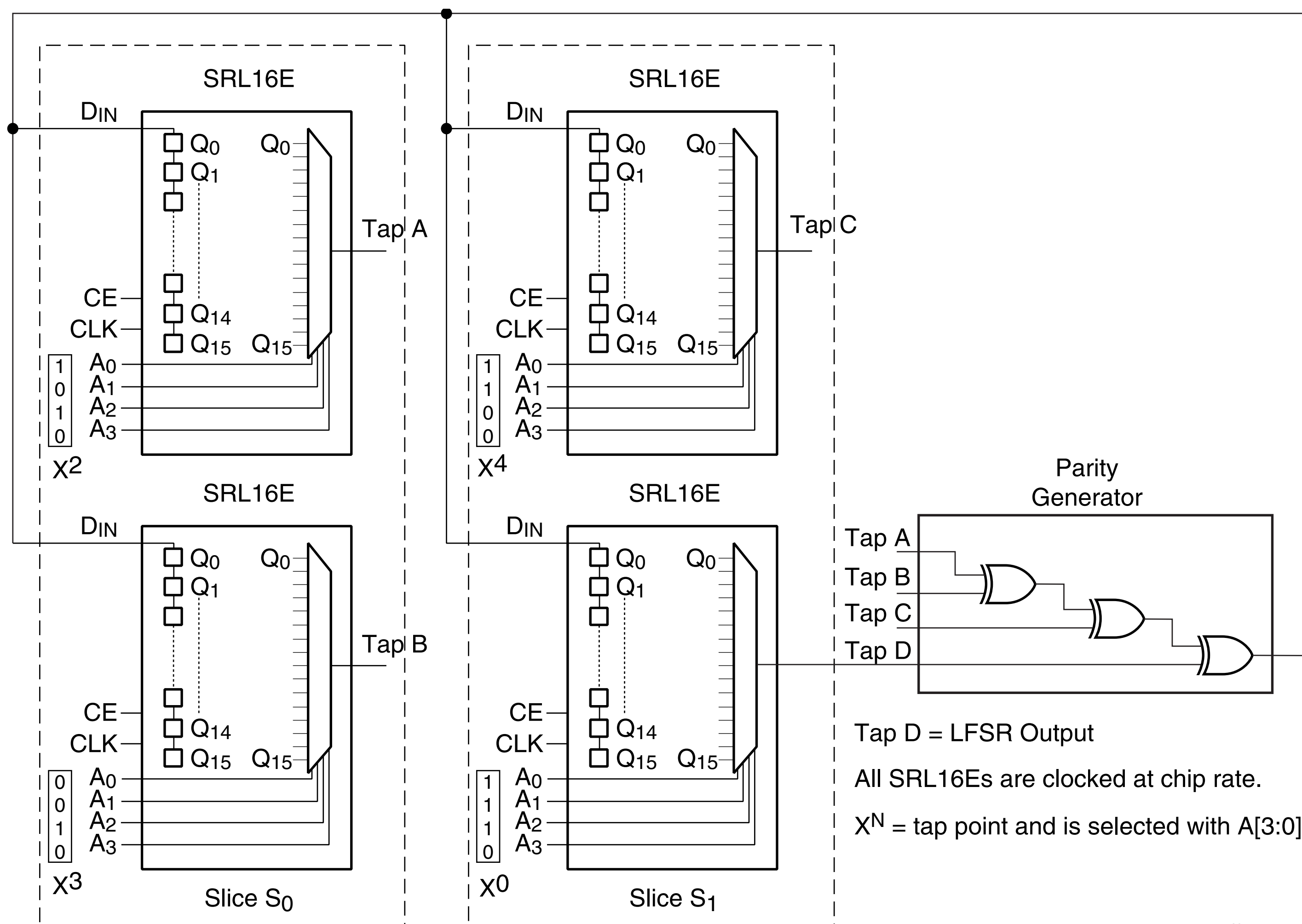

Figure 5: **Four Tap Parallel LFSR**

Single Shift Register with Multicycle Tap Access and Sequential Parity Calculation

Figure 6 demonstrates how a single SRL16E primitive, with some additional logic, implements a 16-stage shift register that is clocked at a frequency called the *chip rate*. The SRL16 primitive address lines are multiplexed at four times the chip rate allowing four of the 16 shift register taps to be accessed during one chip rate period.

The status of each accessed tap is input to a single XOR gate whose output is registered by a flip-flop also clocked at four times the chip rate. During one chip rate period, four taps are read and sequentially XORed to create the parity calculation. The final XOR state is available to the input of the shift register at the chip rate. This circuit enables any four of the sixteen shift register taps to be read, XORed together, and presented to the input of the shift register at the chip rate.

Accessing the shift register taps over multiple cycles enables parallel access to four of the shift register taps at the chip rate. The multicycle clock rate should be twice the chip rate if only two taps are required.

To access an odd number of taps during one chip rate period, the multicycle clock can be a binary-power-of-two multiple of the chip rate.

Multi-cycle clock frequency = chip rate $\times 2^N$ (where N is an integer)

The FDRE flip-flop can be clock enabled for only as many clock cycles as there are taps to be accessed during the chip rate period. For example, to implement a polynomial with three feedback taps, the FDRE could be clocked with a 4× chip rate clock, and only clock enabled for three of the cycles.

- One SRL16 implements 16-stage shift register
- Selected taps are multiplexed to the output at 4× shift register clock rate
- LUT-based XOR implements a time-shared parity generator

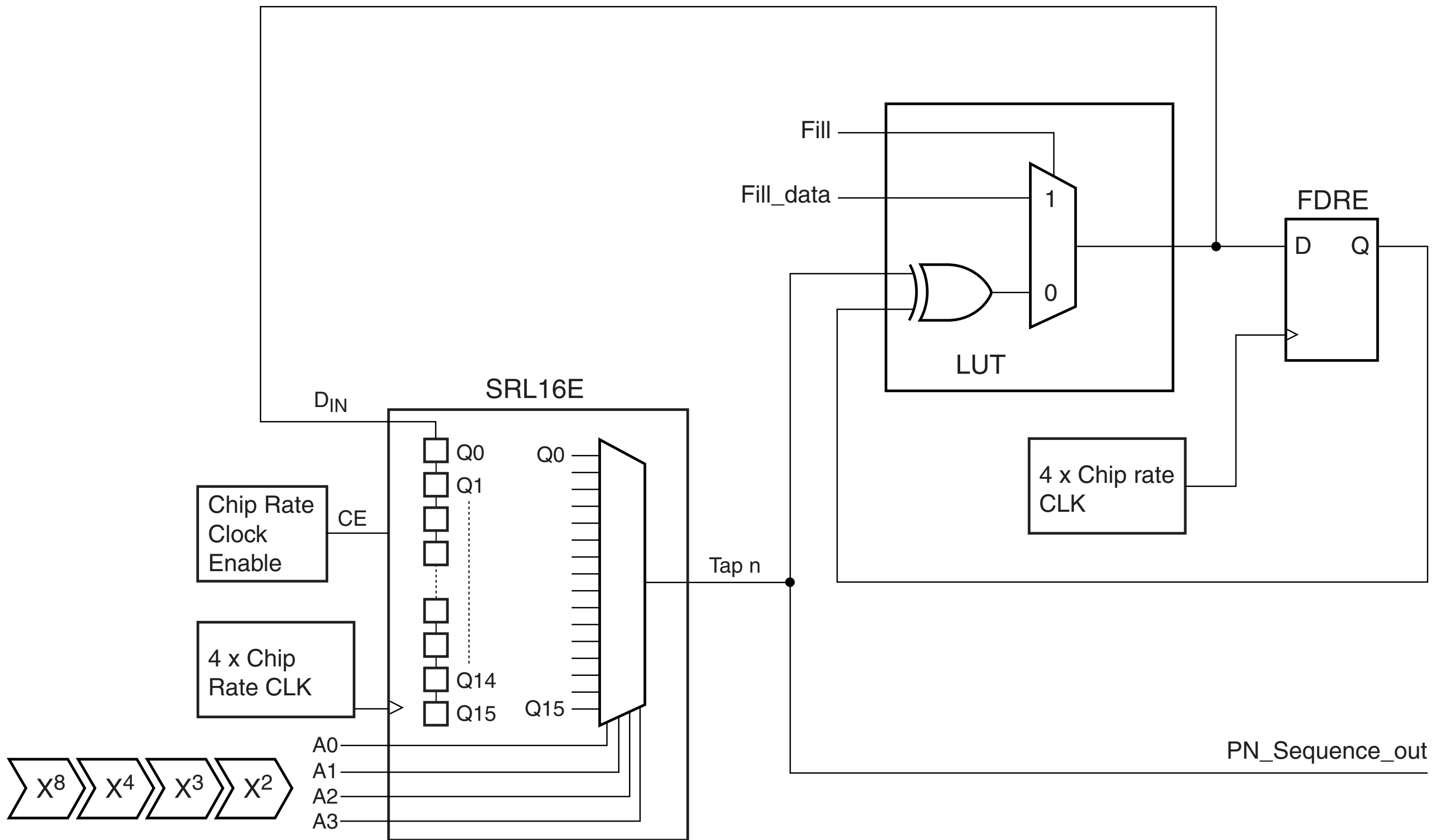

Figure 6: **Multicycle Tap Access and Sequential Parity Calculation**

Fill State

The *fill state* is defined as that point in a maximal length sequence at which the LFSR will start generating the subsequent states of that sequence. The fill is required to be completed in one cycle of the chip rate. With a parallel shift register, this requirement is easy to satisfy. It requires that each flip-flop in the chain be preceded by a multiplexer that can select between loading parallel FILL data or the shift-out data from the previous flip-flop. To implement an LFSR with this capability using flip-flops requires a 2:1 multiplexer at the input of every flip-flop. This means every shift register stage requires a LUT and flip-flop pair.

The SRL16 primitive is not a parallel-load shift register, but a solution to the parallel load problem comes through observing exactly what is happening during the last *N* chip periods of an *N*-stage LFSR prior to the FILL transition.

Consider an eight-stage LFSR that has just reached a condition where it is eight chip rate clock cycles away from a FILL transition. During these last eight clock cycles, the contents of the shift register are shifted out as the last eight states of the sequence prior to the FILL signal. Also during this eight-cycle period, the XOR feedback path will be generating eight new bits to inject into the shift register as the next eight bits of the sequence.

However, the eight feedback bits calculated during the last eight cycles of the current sequence will not be shifted out as part of the sequence because they will be overwritten by the FILL bits. So rather than shift these eight feedback bits into the shift register, the eight clock cycles preceding the FILL command can be used to serially shift in the eight bits of new FILL data.

This enables the SRL16 primitive to be used in LFSR applications where the LFSR has to be parallel loaded in one cycle of the chip rate clock. Note that this implementation is only applicable to instances where an occurrence of the FILL command can be predicted.

Implementing the serial load is achieved with a 2:1 multiplexer that routes either feedback data or new FILL data into the shift register at the chip rate. The multiplexer select line should be pulled high *N* chip rate clock cycles before the last sequence. This multiplexer arrangement is shown in Figure 7.

HDL Code Implementation of LFSRs

LFSRs with Parallel Tap Access and Parity Calculation

One method of creating an LFSR is creating multiple parallel SRL16s for each tap. The different outputs are then XNORed simultaneously using a single LUT and the output is then fedback into each SRL16. Examples of a 16-bit LFSR implemented using the SRL16 primitives in both VHDL and Verilog are available (xapp220.zip). It is important to note that this code infers the SRL16 primitives, and thus is portable to any architecture. Because these SRL16s are inferred, they cannot be initialized to a known state on power-up (other than all zeros), nor can the shift registers be dynamically changed to a different length by altering the address lines. In this design, a single bit 2:1 multiplexer is inserted in order to enable the user to shift in the initial sequence.

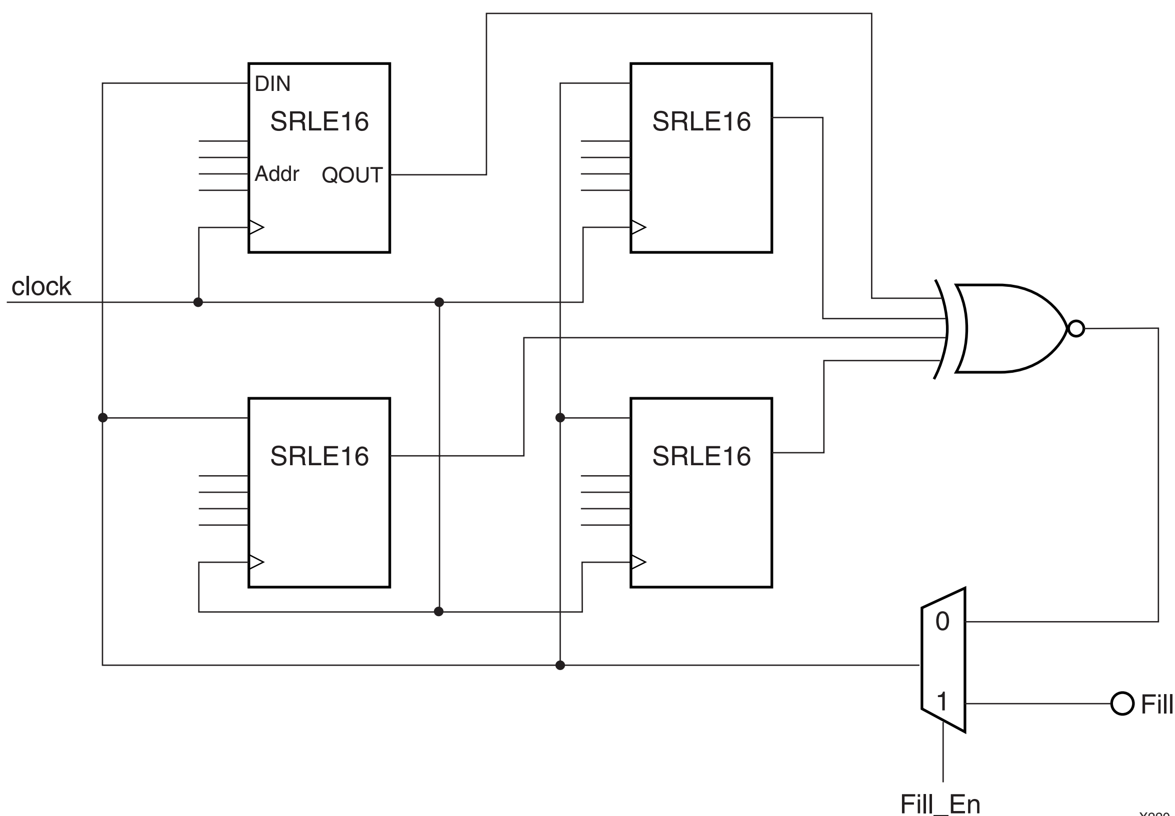

Figure 7: **16-bit, 4-tap Parallel LFSR**

This implementation can be cascaded in order to implement larger designs. It is important to note that a LFSR which is more than twice as long will not necessarily use twice as many SRL16s. For example, a 32-bit LFSR which has bus taps on bits 32, 22, 2, and 1 will not use

eight SRL16s even though the 16-bit implementation uses four as shown in Figure 7. In the 32-bit LFSR it will only use five SRL16s (Figure 8). Likewise a 64-bit LFSR will not use ten SRL16s, it will only use seven. Thus the marginal cost of using SRL16s to implement LFSRs decreases with the size of the LFSR and the location of the taps.

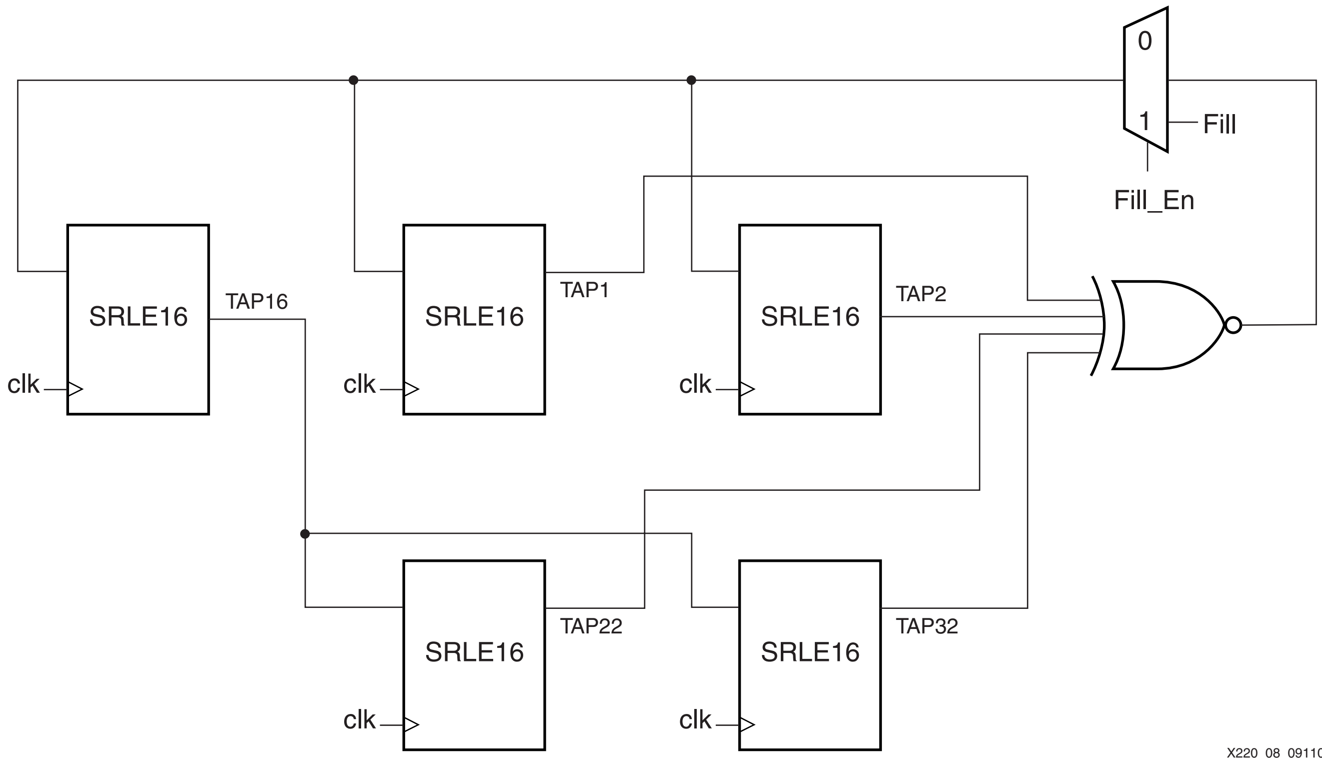

Figure 8: **32-bit, 4-tap Parallel LFSR**

The code has been tested on the following synthesis tools:

- FPGA Express 3.4
- Synplicity 6.0
- Leonardo Spectrum 2000a2.7s

FPGA Express

FPGA Express may not infer an SRL16 when the length of the shift register is two or less. Thus a tap on the first or second bit will be implemented using two flip-flops. When a tap on a shift register follows immediately after another tap, the tool will use a flip-flop instead of creating another SRL16 unless the **Preserve Hierarchy** option is checked.

Synplicity

As in FPGA Express, Synplicity may not infer an SRL16 when the length of the shift register is two or less.

Leonardo Spectrum

Leonardo Spectrum by default will not merge two SRL16s into a single slice, even if they share a common input and clock. In order to get this feature disabled, set the following environment variable to false:

set virtex_map_srl_pack FALSE

When a tap on a shift register follows immediately after another tap, the tool will use a flip-flop instead of creating another SRL16 unless the **Preserve Hierarchy** option is checked.

LFSRs with Multi-cycle Tap Access and Sequential Parity Calculation

A second method of creating an LFSR uses a single SRL16E, gaining access to the different taps by changing the address lines on the LUT that implements the SRL16E. An example of a 16-bit LFSR implemented in VHDL and Verilog is available (**xapp220.zip**). In this case, the output will be delayed by the number of taps needed to implement it. As shown in Figure 9, the output is produced on every rising edge of the chip rate clock, but the SRL16 and the output flip-flop are actually clocked at an increased clock rate—four times the chip rate for 4-tap LFSRs, and twice the chip rate for 2-tap LFSRs. The individual taps are then XNORed serially using the single output flip-flop. Note that during the last increased chip rate clock cycle, this flip-flop is synchronously reset to prepare it for the next increased clock cycle parity calculation.

This method of implementation may use fewer resources in certain extremely long LFSRs; however, for shorter length LFSRs, the parallel LFSR implementation requires fewer resources. In the multicycle implementation, the SRL16E primitives have to be instantiated, since the synthesis tools cannot infer a dynamically changing output on the SRL16E.

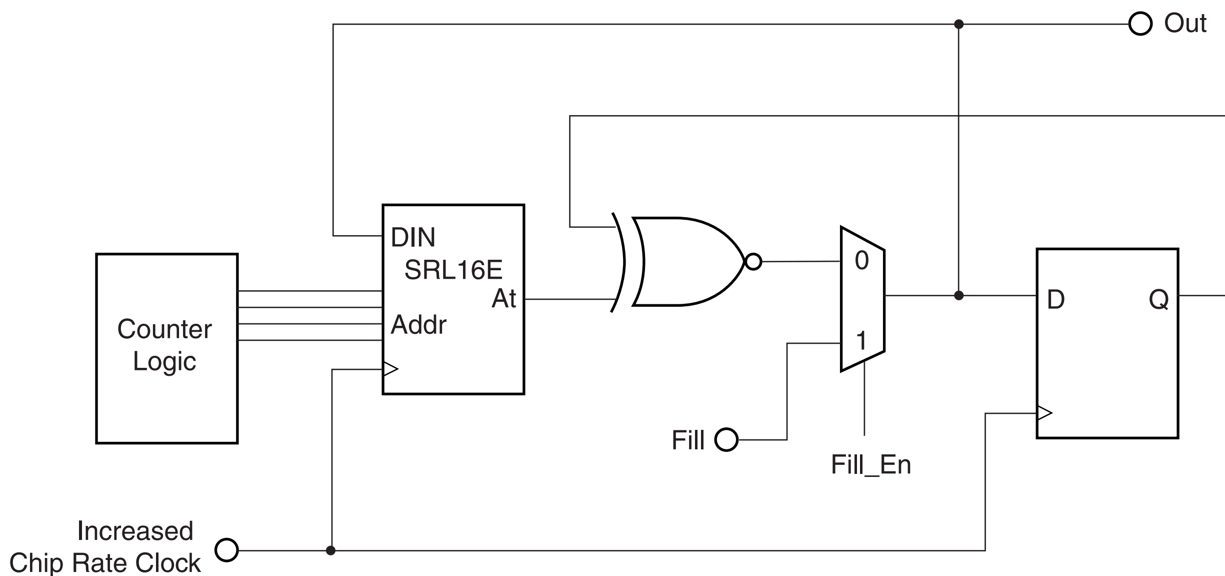

Figure 9: **Multicycle Tap Access LFSR**

HDL Code

The reference design was written in both VHDL and Verilog HDL. The files are available on the Xilinx web site at **xapp220.zip** or **xapp220.tar.gz**. The code was tested to work with current versions of Express, Exemplar, and Synplify. For both VHDL and Verilog code, the design is in two hierarchical levels. This makes the code readable and produces more efficient debugging and verification.

In the reference design, SRL16/E components were inferred to achieve the most efficient implementation results. The XCV50E-8 was the targeted device. The SRL16 is a shift register LUT with four inputs to select the length of the output signal. The SRL16 component can be instantiated in code, but doing so limits the ability to quickly modify the functionality.

To ensure that the SRL component is inferred, follow the required syntax. Synthesis tools recognize this syntax and infer the SRL16 component for Virtex series FPGAs. If for any reason the code is written differently, the output netlist (written by the Synthesis tool) should be checked to verify the presence of the SRL16 components. An example of the syntax that will correctly infer the SRL16 component is available on the Xilinx website at **http://www.xilinx.com/techdocs/7822.htm**. For readability, each LFSR implementation is in a separate block. If there are any problems after following the syntax in the above-listed solution, contact Xilinx Technical Support at **http://support.xilinx.com**.

The code was simulated on MTI's Modelsim simulator using the TCL interface; therefore, no testbench was used. On a simulator supporting stimulus using HDL code only, create the testbench (HDL) file to verify functionality.

The code has been tested on the following synthesis tools:

- FPGA Express 3.4
- Synplicity 6.0
- Leonardo Spectrum 2001a2.75

This implementation can also be cascaded to implement longer shift register sequences. Table 1 summarizes the utilization for each synthesis tool.

Table 1: **Utilization Summary (Appendix A Code 16 bit length LFSR)**

Tool	Synopsys FPGA Express v3.4	Synplicity Synplify 6.0	Exemplar Leonardo Spectrum 2001a2.75
Slices	4	4	3
LUTs	2	2	2
SRL16s	3	3	4
Flip-flops	2	2	0

Conclusion

Efficient LFSRs can be designed using the Virtex, Virtex-E, Virtex-EM, Virtex-II, and Spartan-II device architectural features like the Shift Register LUTs (SRL16). The FPGA resource saving becomes evident in applications like pseudo-random noise (PN) code generators and Gold code generators used in Code Division Multiple Access (CDMA) systems. The LFSRs are the basic functional blocks in these generators. For example, a 41-stage, two-tap Gold code generator can be implemented in just 5.5 Virtex slices.

The example given in this document creates a 41-stage Gold code generator. Each LFSR is a 41-stage, two-tap LFSR implemented using SRL16s.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/11/00	1.0	Initial Xilinx release.
01/11/01	1.1	Added Virtex-II information.