Linear Feedback Shift Registers Theory and Applications

Kewal K. Saluja

Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Madison, Wisconsin 53706

November 1987
Revised October 1988
Updated 1991

A version of these notes was originally prepared by Prof. James E. Smith of the Univesity of Wisconsin - Madison.

1 Introduction

Use of *Linear Feedback Shift Registers* (LFSR) is being studied extensively by engineers, designers and researchers working in testing, design for testability and built-in self-test environments. LFSRs are rather attractive structures for use in these environments for some of the following reasons:

1) LFSRs have a simple and fairly regular structure,

2) their shift property is easily integratable in the scan design environment,

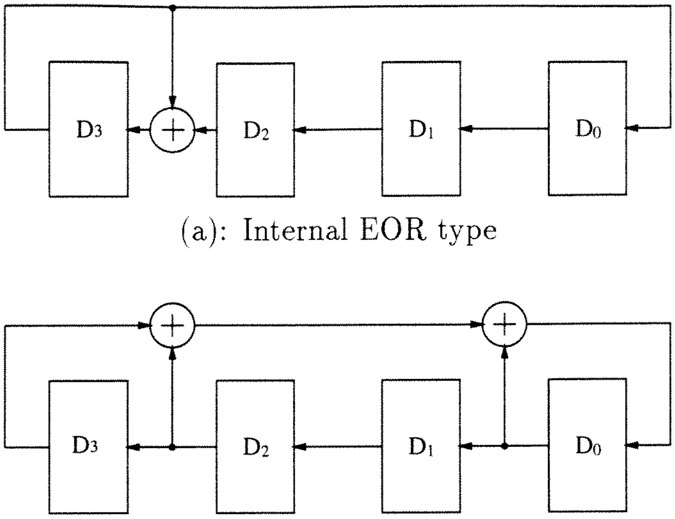
3) they are capable of generating exhaustive and/or random vectors, and

4) their error detection and error correction properties make them a prime candidates for signature analysis applications.

Typically the components used to construct them are D-type flip-flops and *Exclusive-OR* (EOR) gates. Despite their simple appearance, LFSRs are based on rather complex mathematical theory and have several interesting applications, particularly in the area of digital system testing and fault-tolerant computing. Examples of some of the applications are the random testing of logic circuits, fault signature analysis and error detecting/correcting codes.

Two example LFSRs are shown in Figure 1. Note that both these structures use D-type flip-flops and linear logic elements (EOR gates) to realize LFSRs. The basic difference in these two structures being the circuit of Figure 1(a) uses linear elements interspersed between the flip-flops whereas the circuit of Figure 1(b) has no linear element appearing between the flip-flops instead the linear elements appear only in the feedback path. It is for this reason the realization of Figure 1(a) is called *internal-EOR* LFSR and the realization of Figure 1(b) is called *external-EOR* LFSR. An equivalence exists between the two structures in the sense that knowing the properties of the first structure one can deduce the properties of the second structure. We shall essentially study the structures of the first type in this note. Although we shall point out the properties of the other type through examples.

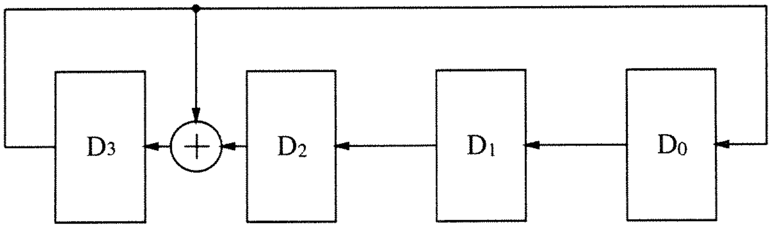
Let us consider the example LFSR shown in Figure 1(a). If we initialize the shift register to a nonzero state (the state of all the D-type flip-flops represented as a vector is called the state of the LFSR), say 0110, and shift the register a number of times, the LFSR goes through a number of states and we generate a sequence of binary



(a): Internal EOR type

(b): External EOR type

Figure 1: Example LFSRs



D_3	D_2	D_1	D_0
0	1	1	0
1	1	0	0
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0
1	0	0	1
1	0	1	1
1	1	1	1
0	1	1	1

Figure 2: A simulation of 8 shifts of the LFSR of Figure 1(a) with initial contents 0110.

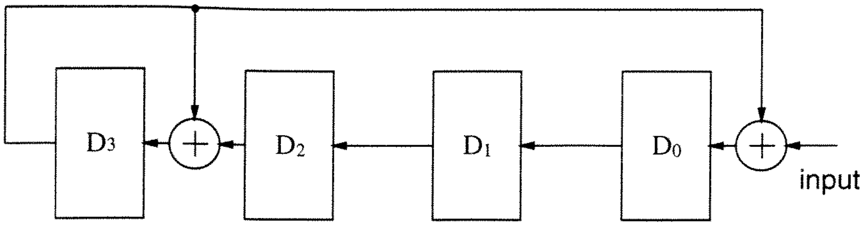


Figure 3: An LFSR with an input.

vectors shown in Figure 2. For certain configurations of the LFSRs, these sequences have an interesting property, to be more precise they are *pseudo-random.* That is, they possess many of the statistical properties of sequence of random vectors [4], although in a strict sense they are not random. Nevertheless, for generating random test vectors, pseudo-randomness is quite sufficient, and LFSRs provide an efficient and economical means for random logic testing.

We can also add an input to an LFSR (Figure 3) and shift in a serial stream of inputs. This, in effect compresses the stream of inputs to the length of the LFSR. The contents of the LFSR are said to form the *signature* of the serial input stream. It is this data compression capability that is used in fault signature analysis.

Binary vectors can be represented as polynomials with binary coefficients. LFSRs can be configured so that they can multiply and divide such polynomials. This ability to perform arithmetic operations on polynomials can be used to encode and decode error detecting and correcting codes. The popular *cyclic redundancy check* (CRC) codes are among the large class of codes that can be encoded and decoded in this way.

In this note, a brief discussion of the mathematical structure of the LFSRs is presented. The behavior of LFSRs can be studied by either making use of linear recurrence relations to describe them or through the study of polynomial fields. In a given situation one method may be preferable over the other. Therefore, we will describe both methods to represent LFSRs. The underlying theory of finite fields makes up an important part of modern algebra, and as such, entire courses are often devoted entirely to the study of finite fields. Because of the size and the complexity of the theory only those results that are pertinent to our applications are discussed. Theorems are given without proof. For more details, one can refer to nearly any textbook in modern algebra. For a treatment of finite fields geared towards computer applications, a number of references from the list of refences at the end of this note will be useful [5, 8].

Following the sections on the mathematics of LFSRs there are two sections devoted to the applications of the LFSRs. In these sections references are given to allow further exploration of the applications of the LFSRs.

2 The Mathematics of LFSRs

2.1 Groups

Definition 1: A *group*, <G, ·>, consists of a set G and a binary operator, ·, and has the following properties:

1) (*closure*) $g \cdot h \in G$ for all $g, h \in G$.
2) (*associativity*) $g \cdot (h \cdot k) = (g \cdot h) \cdot k$ for all $g, h, k \in G$.
3) (*identity*) There is an $e \in G$ such that $e \cdot g = g \cdot e = g$ for all $g \in G$.
4) (*inverse*) For each $g \in G$ there exists $g^{-1} \in G$ such that $g \cdot g^{-1} = g^{-1} \cdot g = e$.

A group is *abelian* if the group operation is also commutative, i. e., if for every $g, h \in G$ $g \cdot h = h \cdot g$.

As an example of a group, we consider the set of integers, $Z = \{0, \pm 1, \pm 2, \pm 3, \ldots\}$, and the operator addition, +, as the group operator. Then $G =< Z, + >$ is a group. This can be easily verified; for example the identity is 0 and the inverse of a is -a. On the other hand, if · is integer multiplication than $G =< Z, \cdot >$ is not a group because 1 is the only member of Z that has an inverse in Z.

$< Z, + >$ is an example of an infinite group, but we are actually more interested here in finite groups. We let $Z_n = \{0, 1, 2, ..., n-1\}$, and instead of using conventional integer arithmetic, we use *modular* arithmetic.

Two integers I and J are *congruent* modulo n, written $I \equiv J \ (mod\ n)$ if there is another integer K such that $I = J + nK$. J is said to be a *residue* of I. The *least positive residue* can be found by dividing I by n and taking the positive remainder. For example, the least positive residue of 21 *mod* 4 is 1. Modulus addition, $+_n$, and modulus multiplication, $\cdot_n$, are arithmetic operations that give the least positive residue *mod n* of normal addition and multiplication on integers.

Using modular arithmetic, $< Z_n, +_n >$ forms a finite group for any positive n, Furthermore, $< Z_n, +_n >$ is abelian.

Definition 2: A subset of elements of a finite group is said to be a set of generators of the group if every element of G can be expressed only in terms of the group operation and the generators.

For example, in $< Z_6, +_6 >$ the set $\{2, 3\}$ is a set of generators because

$$\begin{array}{rcl} 0 & = & 2 +_6 2 +_6 2; \\ 1 & = & 2 +_6 2 +_6 3; \\ 2 & = & 2; \\ 3 & = & 3; \\ 4 & = & 2 +_6 2; \\ 5 & = & 2 +_6 3. \end{array}$$

Definition 3: A group is *cyclic* if it has a generator with one element.

2.2 Fields

We now turn to an algebraic structure more complex than a group, one which has two operations.

Definition 4: A field, $< F, +, \cdot >$, is an algebraic structure where F is a set and + and $\cdot$ are binary operations, and the following axioms are satisfied.

1) $< F, + >$ is an abelian group.

2) If 0 is the identity with respect to +, $< F - \{0\}, \cdot >$ is an abelian group.

3) $\cdot$ distributes over +, i.e., $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ and $(y + z) \cdot x = (y \cdot x) + (z \cdot x)$ for all $x, y, z \in F$.

Theorem 1: $< Z_n, +_n, \cdot_n >$ is a field if and only if n is prime.

2.3 Polynomial Fields

Theorem 1 points out a set of finite fields that are based on prime numbers. We now develop a generalization of such fields based on polynomials.

We can represent vectors as polynomial coefficients. For example, $< 3, 2, 5, 0, 1 >$ could be represented as $3x^4 + 2x^3 + 5x^2 + 1$, and $< 1, 0, 1, 0, 1 >$ could be represented as $x^4 + x^2 + 1$.

We can perform arithmetic on polynomials, just as we can with integers. Further, we can choose coefficients from a field $< Z_n, +_n, \cdot_n >$ and perform coefficient arithmetic within this field. For example, consider two polynomials $q(x)$ and $p(x)$ with coefficients in $< Z_3, +_3, \cdot_3 >$:

$$
\begin{aligned}
q(x) &= x^3 + x^2 + 1 \\
p(x) &= 2x^2 + x + 1 \\
p(x) + q(x) &= x^3 + 3x^2 + x + 2 \\
&= x^3 + x + 2 \text{ (note that coefficient are added } \mathit{mod} \text{ 3)}
\end{aligned}
$$

Similarly,

$$p(x) \cdot q(x) = 2x^5 + 2x^3 + x + 1$$

In addition, we can express polynomials *mod* a polynomial. That is, two polynomials $r(x)$ and $s(x)$ are *congruent* modulo $q(x)$, written as $r(x) = s(x) \ \mathit{mod} \ q(x)$ if there is a polynomial $n(x)$ such that $r(x) = s(x) + q(x) \cdot n(x)$. As with integers, the *least positive residue* can be found by dividing $r(x)$ by $q(x)$ and taking the positive remainder. Then, we can perform modulus arithmetic on polynomials as we do with integers.

For example, if coefficients are in the field $< Z_2, +_2, \cdot_2 >$, i.e., we are representing binary vectors, then

$$
\begin{aligned}
& (x^2 + x + 1) \cdot (x^3 + 1) \ \mathit{mod} \ (x^4 + x) \\
= \ & (x^5 + x^4 + x^3 + x^2 + x + 1) \ \mathit{mod} \ (x^4 + x)
\end{aligned}
$$

Performing the required division, we get:

$$
\begin{array}{r|rrrrrrrrrrr}
\multicolumn{1}{r}{} & & & & & & & & & x & + & 1 \\ \cline{2-12}
x^4 + x & x^5 & + & x^4 & + & x^3 & + & x^2 & + & x & + & 1 \\
\multicolumn{1}{r}{} & x^5 & & & & & + & x^2 & & & & \\ \cline{2-10}
\multicolumn{1}{r}{} & & & x^4 & + & x^3 & & & + & x & & \\
\multicolumn{1}{r}{} & & & x^4 & & & & & + & x & & \\ \cline{4-12}
\multicolumn{1}{r}{} & & & & & x^3 & & & & & + & 1
\end{array}
$$

Therefore, $(x^2 + x + 1) \cdot (x^3 + 1) \ \mathit{mod} \ (x^4 + x) = x^3 + 1$. We can then consider an algebraic structure containing the set of all polynomials of degree less than n defined on the coefficient field $< Z_p, +_p, \cdot_p >$. In this field the addition and the multiplication operations are on polynomials and are *mod* $q(x)$ where $q(x)$ is a degree n polynomial. This structure is an *algebra of polynomials mod* $q(x)$. We now show an analogy to prime integers for polynomials.

Definition 5: A polynomial $q(x)$ of degree $n > 0$ defined in a coefficient field is *irreducible* if there do not exist polynomials $f(x)$ and $g(x)$ of degree > 0 defined in the same coefficient field such that $f(x) \cdot g(x) = q(x)$ where multiplication is ordinary polynomial multiplication with coefficient operations in the coefficient field.

For example, we know from an earlier example that $2x^5 + 2x^3 + x + 1$ is not irreducible over $< Z_3, +_3, \cdot_3 >$, since $2x^5 + 2x^3 + x + 1 = (x^3 + x^2 + 1) \cdot (2x^2 + x + 1)$. On the other hand, $x^3 + x + 1$ is irreducible over the same coefficient field.

Theorem 2: In the algebra of polynomials modulo $q(x)$, where $q(x)$ is a polynomial of degree n over a field $< z_p, +_p, \cdot_p >$, the polynomials form a field with respect to polynomial multiplication and addition if and only if $q(x)$ is irreducible.

Definition 6: An irreducible polynomial $q(x)$ is *primitive* if the polynomial x is a generator of the multiplicative group of the field of polynomials *mod* $q(x)$.

This definition of primitive polynomial differs from the standard one although it is equivalent; this particular definition is chosen because of the particular application we have for primitive polynomials. Table 1 contains a selected set of primitive polynomials with binary coefficients.

Example: Since $x^4 + x^3 + 1$ is primitive, x must generate the nonzero members of the algebra of polynomials *mod* $(x^4 + x^3 + 1)$. By performing repeated multiplications by x, the sequence of polynomials generated is:

> x, x^2, x^3, $x^4 = x^3 + 1$ (*mod* $x^4 + x^3 + 1$), $x^3 + x + 1$, $x^3 + x^2 + x + 1$, $x^2 + x + 1$, $x^3 + x^2 + x$, $x^2 + 1$, $x^3 + x$, $x^3 + x^2 + 1$, $x + 1$, $x^2 + x$, $x^3 + x^2$, 1, then the sequence begins again with x.

$x^2 + x + 1$
$x^3 + x^2 + 1$
$x^4 + x^3 + 1$
$x^8 + x^4 + x^3 + x^2 + 1$
$x^{12} + x^6 + x^4 + x + 1$
$x^{16} + x^5 + x^3 + x^2 + 1$
$x^{16} + x^9 + x^7 + x^4 + 1$
$x^{32} + x^7 + x^5 + x^3 + x^2 + x + 1$
$x^{64} + x^4 + x^3 + x + 1$

Table 1: A selected set of primitive polynomials.

2.4 Arithmetic with LFSRs

In the following discussion, binary vectors are of primary interest, so we use polynomials with binary coefficients. We now show how arithmetic in a polynomial field can be performed using LFSRs.

LFSRs have two basic building blocks:

1) a storage device (we use D-type flip-flops) and

2) a *mod* 2 adder/subtractor, i.e., an EOR gate.

Note that *mod* 2 addition and subtraction are the same functions.

First we consider multiplication of a polynomial, $q(x)$, by x *mod* $g(x)$. Say $q(x) = q_r x^r + q_{r-1}x^{r-1} + ... + q_0 x^0$ and $g(x) = g_n x^n + g_{n-1}x^{n-1} + ... + g_0 x^0$ where $n > r$. We want to compute $xq(x)$ *mod* $g(x)$. Now, $xq(x) = q_r x^{r+1} + q_{r-1}x^r + ... + q_0 x^1$; if $q(x)$ were stored as a binary vector in a shift register, $xq(x)$ would correspond to a left shift of the shift register by one position. If we put $q(x)$ in a shift register of length $n-1$ and a 1 is to be shifted out of the high order position, then $r+1 = n$ and to get $xq(x)$ *mod* $g(x)$ we need to subtract $g(x)$ from $xq(x)$; on the other hand if a 0 is to be shifted out, we do nothing to $xq(x)$. Recalling that *mod* 2 coefficient subtraction can be done with an EOR, Figure 4 shows an LFSR that computes $xq(x)$ *mod* $g(x)$. the meaning of g_i is that if the coefficient g_i of x^i in $g(x)$ is 1 then connection is made and if $g_i = 0$ then there is no connection. Figure 1(a) shows the LFSR for computing $xq(x)$ *mod* $(x^4 + x^3 + 1)$.

Figure 5 shows an LFSR for multiplying an arbitrary polynomial by a given poly-

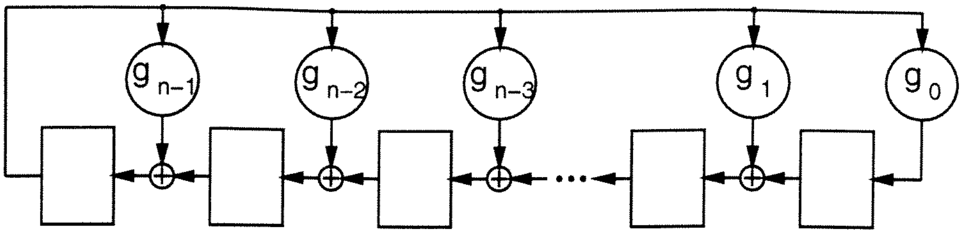


Figure 4: A shift register for computing $xq(x)$ *mod* $g(x)$, where $q(x)$ is the initial content of the shift register.

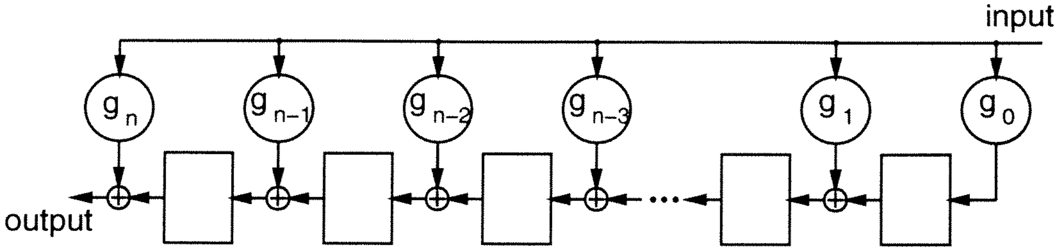
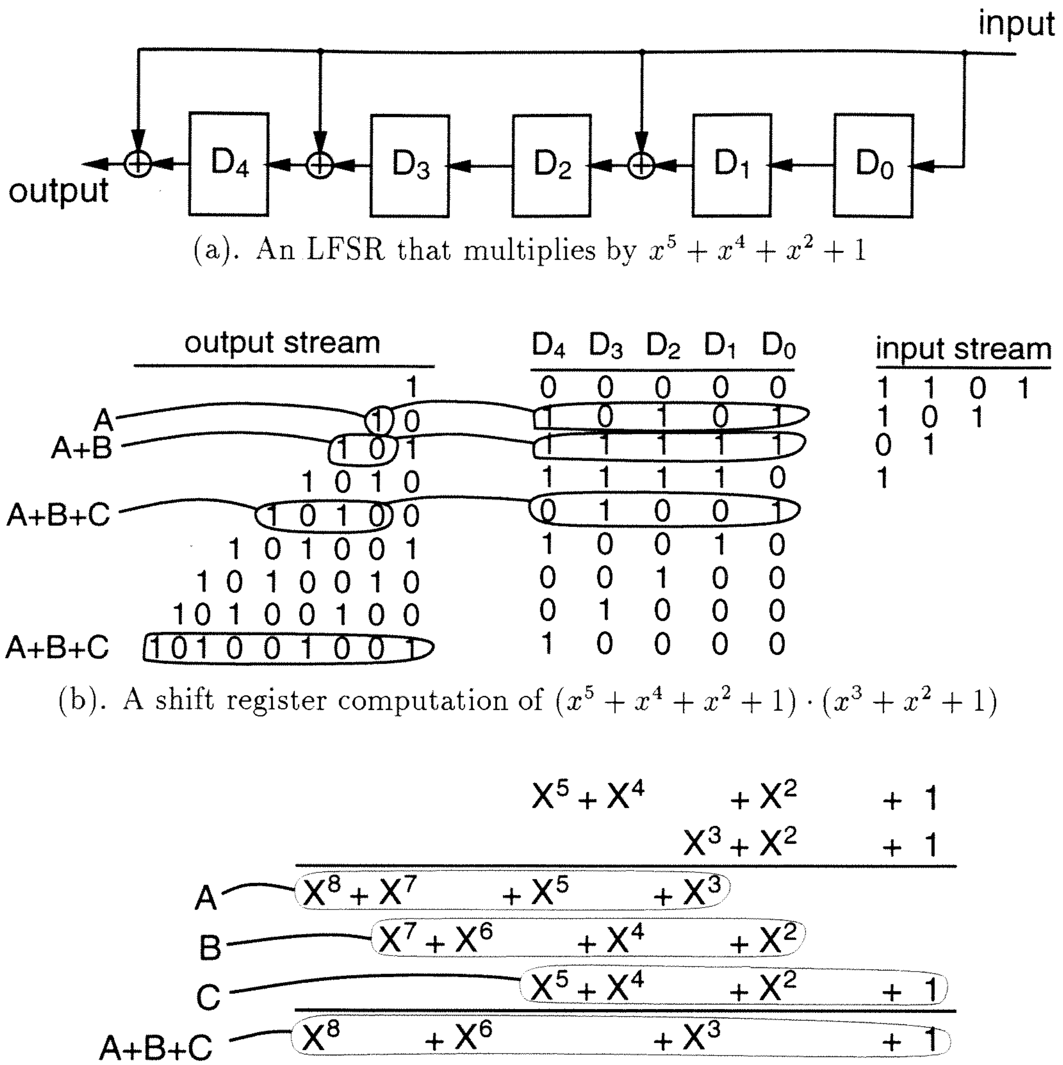


Figure 5: A shift register for multiplying an input polynomial by $g(x)$.

nomial $g(x)$.

The shift register for multiplication by $g(x) = x^5 + x^4 + x^2 + 1$ is shown in Figure 6(a). The set of n D-flip-flops store a polynomial. Initially all the flip-flops are set to zeros. The presence of the coefficient q_k at the input adds $q_k g(x)$ to the register. Shifting multiplies by x and delivers the first coefficient whose calculation is complete to the output. Then, the appearance of q_{k-1} at the input adds $q_{k-1}g(x)$ into the register, and shifting again multiplies by x and delivers the second coefficient to the output, etc. We see that a total of $r+n$ shifts are needed to produce the product. Compare the shift register operation in Figure 6(b) with the long multiplication shown in Figure 6(c); in both cases $g(x)$ is multiplied by $x^3 + x^2 + 1$.

A circuit for dividing polynomials is shown in Figure 7. Here, we divide an input polynomial by $g(x)$. Figure 8(a) shows an LFSR that divides by $x^5 + x^4 + x^2 + 1$. The flip-flops are set to 0 initially. The output is 0 for the first n shifts, until the first input symbol reaches the end of the register. The first nonzero output is $q_r {g_n}^{-1} = q_r$, when we are computing $q(x)/g(x)$. For each quotient coefficient h_j, the polynomial $h_j g(x)$ must be subtracted from the dividend. The feedback connections accomplish this subtraction. After a total of r shifts the entire quotient has been shifted out, and the remainder is held in the shift register. Figure 8(b) shows a shift register computation of $x^7 + x^6 + x^5 + x^4 + x^2 + 1/x^5 + x^4 + x + 1$ and a hand computation



(a). An LFSR that multiplies by $x^5 + x^4 + x^2 + 1$

(b). A shift register computation of $(x^5 + x^4 + x^2 + 1) \cdot (x^3 + x^2 + 1)$

(c). A hand computation of the same multiplication.

Figure 6: Example of LFSR multiplication

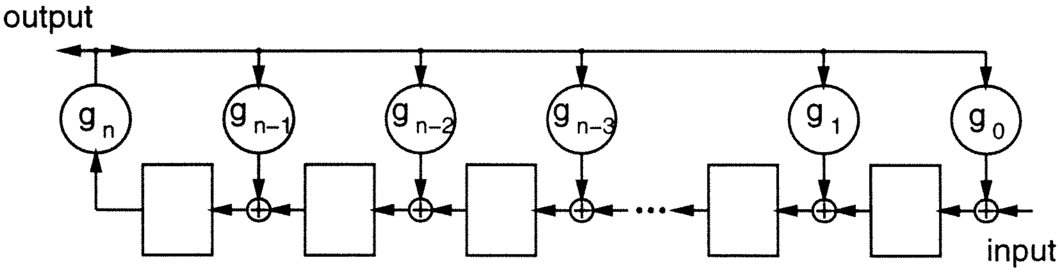
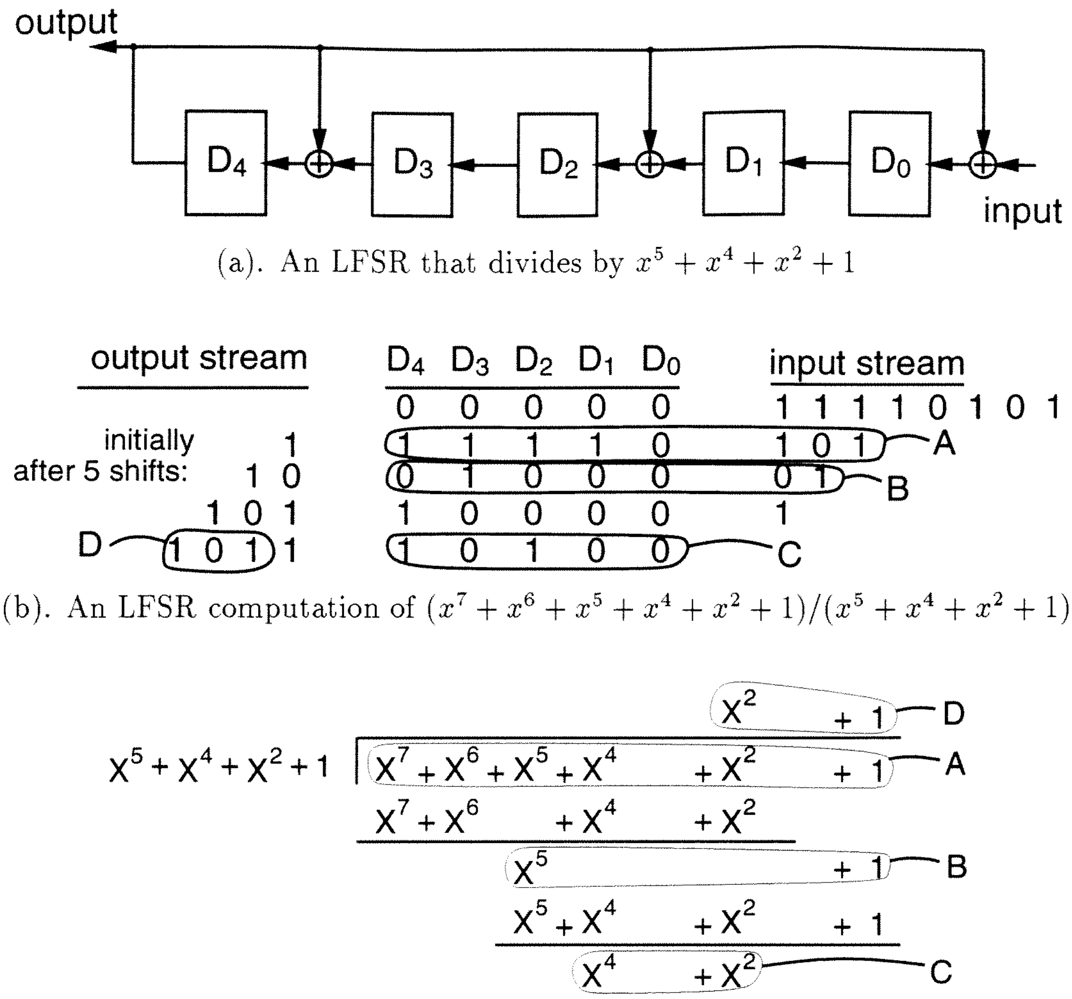


Figure 7: An LFSR for dividing a polynomial $g(x)$



(a). An LFSR that divides by $x^5 + x^4 + x^2 + 1$

(b). An LFSR computation of $(x^7 + x^6 + x^5 + x^4 + x^2 + 1)/(x^5 + x^4 + x^2 + 1)$

(c). A hand computation of the same division

Figure 8: Example of LFSR division

of the same division is given in Figure 8(c). An alternative divider circuit appears in Figure 9; the same division as above is performed in Figure 10. We observe that the quotient produced by the LFSR in Figure 10 is correct, but the content of the register after the division are not the remainder as the case with the divider shown in Figure 8. That is, the input/output behavior is same, but the internal states are different.

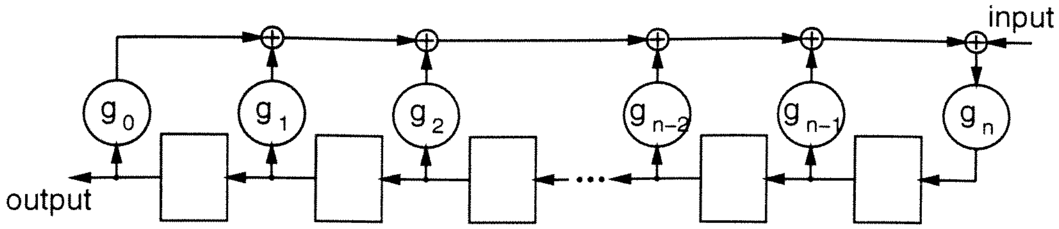
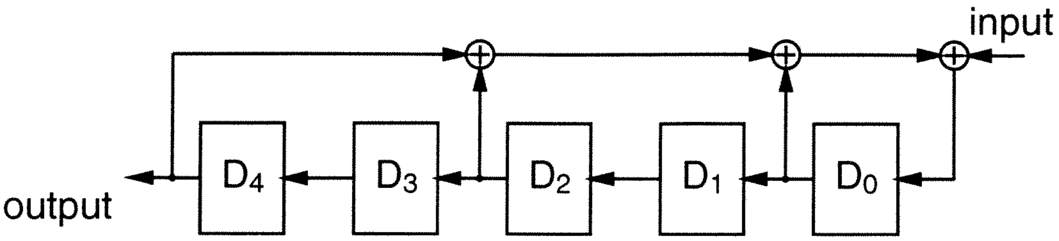


Figure 9: An alternate LFSR for dividing by polynomial $g(x)$



(a). An LFSR for dividing by $x^5 + x^4 + x^2 + 1$

output stream	D_4	D_3	D_2	D_1	D_0	input stream
initially	0	0	0	0	0	1 1 1 1 0 1 0 1
after 5 shifts:	0	0	0	0	1	1 1 1 0 1 0 1
1	0	0	0	1	0	1 1 0 1 0 1
1 0	0	0	1	0	1	1 0 1 0 1
D 1 0 1	0	1	0	1	1	0 1 0 1
	1	0	1	1	1	1 0 1
	0	1	1	1	0	0 1
	1	1	1	0	1	1
	1	1	0	1	0	

(b). An LFSR computation of $(x^7 + x^6 + x^5 + x^4 + x^2 + 1)/(x^5 + x^4 + x^2 + 1)$
the final register content is **not** the remainder

Figure 10: Example of LFSR division

2.5 Linear Recurrence Relations

The circuits of Figure 1 can be viewed as an autonomous circuit producing a binary sequence at the D_3 output of the LFSR. A sequence produced by a shift register at its output is termed a *shift register sequence.* A sequence of symbols in which i^{th} symbol can be expressed as a function of some preceding n symbols can be written as a recurrence relation. If the function used to represent the i^{th} symbol is a linear function then the resulting relation is a linear recurrence relation. Clearly knowing

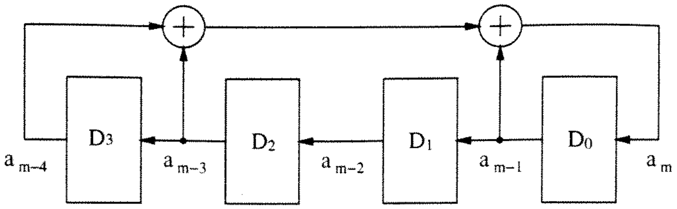


Figure 11: Example LFSRs of Figure 1(b)

the initial state of the LFSR and the recurrence relation defining the output of such an LFSR, complete output sequence for the LFSR can be deduced. If we denote the output sequence as a_0, a_1, a_2, ... , a_n, ... , then the example circuit of Figure 1(b) satisfies

$$\begin{aligned} a_4 &= a_3 + a_1 + a_0 \\ a_5 &= a_4 + a_2 + a_1 \\ a_6 &= a_5 + a_3 + a_2 \\ &\vdots \end{aligned}$$

These equations satisfy the recurrence relation: (as shown in Figure 11)

$$a_m = a_{m-1} + a_{m-3} + a_{m-4}$$

We can use the method of generating functions to study shift register sequences as follows. A *generating function* $G(x)$ can be associated with a shift register sequence, a_0, a_1, a_2, ... , a_i, ... , as follows:

$$G(x) = a_0 + a_1x + a_2x^2 + ... + a_ix^i + ...$$

Thus,

$$G(x) = \sum_{i=0}^{\infty} a_ix^i$$

represents the history of the output stage of the LFSR. Similarly the recurrence relation defining a_m can be expressed as:

$$a_m = \sum_{i=1}^{n} c_ia_{m-i}$$

where c_i = 0 or 1. Note that the recurrence relation which defines a_m depends on the feedback connections in the LFSR and hence the values of c_i depends on the feedback connections. On substituting the value of a_m in the expression for $G(x)$ and rearranging the terms in it we find

$$G(x) = \frac{\sum_{i=1}^{n} c_i x^i (a_{-i}x^{-i} + ... + a_{-1}x^{-1})}{1 + \sum_{i=1}^{n} c_i x^i}$$

In this equation a_{-1}, a_{-2}, ... , a_{-i}, represent the initial state of the shift register and it has been assumed that the coefficients of the polynomial are from the field of two elements often referred as GF(2). The denominator in the above expression is called the *characteristic polynomial* of the above sequence or of the LFSR which produces this sequence. It is evident from this discussion that the output sequence of an LFSR can be determined by knowing the initial state of the LFSR and it's characteristic polynomial.

It is instructive to note that the output sequence of the LFSR can be obtained by dividing the numerator in the expression of $G(x)$ by the characteristic polynomial of the LFSR. In the division process the quotient represents the output sequence a_0, a_1, a_2, ... at any stage of the division process. Clearly this method of analysis is useful to study LFSRs of the type shown in Figure 1(b) or more generally LFSRs which are structurally represented by Figure 12(a). For the example circuit of Figure 1(b) the characteristic polynomial is $f(x) = 1 + x + x^3 + x^4$. Details of this method can be found in [4,5].

3 LFSR as a Test Pattern Generator

Algorithmically generated test patterns for logic networks, while generally giving good fault coverage, have three disadvantages.

1) Test pattern generation can be extremely time-consuming, particularly for sequential circuits.

2) A good deal of storage is required by the tester to hold the test patterns.

3) The speed at which tests can be applied is limited.

The limitation on testing speed mentioned above is caused by the necessity of storing test patterns in some secondary storage and retrieving them as tests are applied. Testing speed is important not only because testing is a necessary step in a manufacturing line, but because it is often desirable to apply tests at operational speed. The problem of testing at operational speeds can be partially overcome by putting a high-speed memory in the tester so that test patterns can be applied for short, high-speed bursts. Nevertheless, these bursts are relatively short since blocks of patterns must still be retrieved from a secondary store.

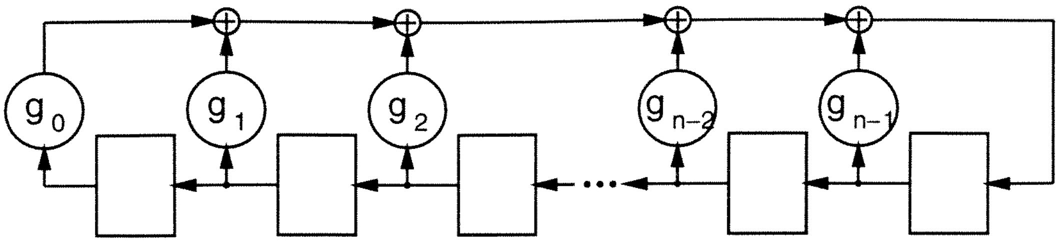
An alternative to algorithmically generated tests which overcomes these disadvantages is the use of pseudo-random or pseudo-exhaustive tests generated by an LFSR.

3.1 LFSR as an Exhaustive Sequence Generator

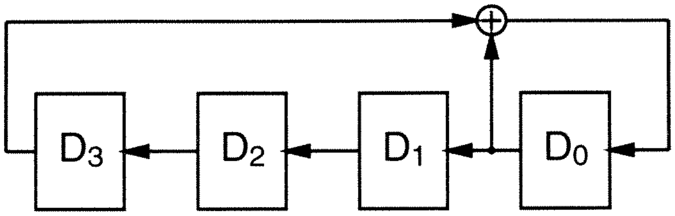
We recall that in a field of polynomials modulo a primitive polynomial, the polynomial x is a generator for the multiplicative group (all the polynomials except 0). This means that if $p(x)$ is a primitive polynomial of degree n then the sequence $x, x^2 \; mod \; p(x), x^3 \; mod \; p(x), ..., x^{2^n-1} \; mod \; p(x)$ generates all $2^n - 1$ nonzero polynomials of degree less than n. The shift register of Figure 4 essentially generates this sequence if the connections correspond to $p(x)$, since it multiplies the initial contents of the register by x modulo $p(x)$. Of course, the register must be initialized to some nonzero polynomial. The register of Figure 1(a) is connected according to the primitive polynomial $x^4 + x^3 + 1$, and one can verify that it cycles through all nonzero polynomials (vectors).

An alternative shift register which also generates all the nonzero binary vectors is shown in Figure 12(a). The register corresponding to the primitive polynomial $x^4 + x^3 + 1$ is shown in Figure 12(b) and the sequence it generates is shown in Figure 12(c). Note that the characteristic polynomial corresponding the LFSR of Figure 12(b) is $x^4 + x + 1$. For more details on this alternative sequence generator refer to [4, 5].

An LFSR of length n that generates all binary n-tuples (except all zero n-tuple) before repeating is said to generate the maximum-length sequence (or m-sequence) of n-tuples. For this reason, such sequence generators are often called maximum-length sequence generators or MLSGs for short. If one were to test combinational logic networks only, MLSGs could be used directly to generate test patterns. That is, for an n-input combinational circuit one could select a primitive polynomial of degree n, and construct the corresponding MLSG. To test the circuit, one could use a known-good circuit and compare test results. It is apparent that the MLSG could be



(a). An alternate MLSG



(b). The MLSG for a primitive polynomial $x^4 + x^3 + 1$

D_3	D_2	D_1	D_0
0	0	0	1
0	0	1	1
0	1	1	1
1	1	1	1
1	1	1	0
1	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0
0	1	0	0
1	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0

(c). The sequence generated by the alternate MLSG.

Figure 12: LFSR as an exhaustive sequence generator

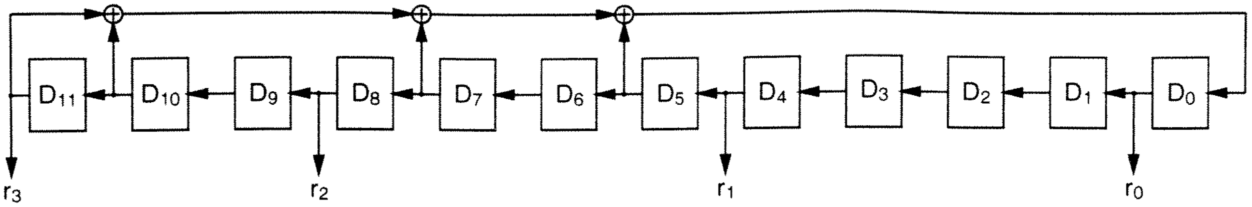
clocked at a very high rate, and no storage of patterns or results is required. Since the test patterns are reproducible, one could use fault simulation to determine the initial contents of the sequence generator and/or the number of shifts that give a sufficient test coverage. If it is considered important to apply all zero pattern to the circuit under test, such a pattern can be applied independently or alternatively the LFSR can be modified to a simple non-linear feedback shift register by including additional gates in the feedback path such that the all zero vector is also generated.

3.2 LFSR as a Random Pattern Generator

From the examples of Figure 2 and Figure 12 we observe that the binary vectors generated by these LFSRs appear to be rather "scrambled". However, they lack an important characteristic of random sequences. The sequences of vectors generated above do not repeat except at intervals of length $2^n - 1$. In a truly random sequence, vectors repeat at varying intervals, and the same vector can appear twice in a row. For testing combinational logic, there is little point in repeating a test vector, but for sequential logic a repetition can be necessary for proper exercising of the circuit.

To construct a pseudo-random test pattern generator (PRTPG) (we have been careful not to call the sequences generated above "pseudo-random"), one should only use a few selected stages of an MLSG for outputs. If we take an MLSG of length n and tap k stages as outputs, then each nonzero k-tuple appears 2^{n-k} times and the 0 vector appears $2^{n-k} - 1$ times as the MLSG cycles through its $2^n - 1$ states. (We note that the 0 vector will appear with this configuration, while it does not in an MLSG.) Further, the appearances of a given vector are separated by a variety of (nearly random) intervals.

Example: We wish to generate pseudo-random 4-tuples. One possibility is to choose primitive polynomial $x^{12} + x^6 + x^4 + x + 1$, construct an MLSG for the primitive polynomial, and tap 4 stages (see Figure 13). The choice of stages tapped is often left to intuition, but as a rule of thumb, the more separation between the taps, the less correlation there will be between input streams seen at individual network inputs. The PRTPG just discussed will cause each output bit position to be a 0 approximately half the time and 1 the other half. This may cause poor fault coverage for some networks. For example, consider a counter with a reset line. If it were tested with a PRTPG, it would be reset approximately half the time, and the counter would be poorly exercised.



(a). A Pseudo random test pattern generator PRTPG

r_3	r_2	r_1	r_0
1	0	0	1
0	1	0	0
1	0	1	1
0	1	0	1
1	0	1	0
0	0	0	0
1	1	1	1
0	0	1	0
0	1	0	0
1	0	0	0

⋮

(b). Sequence generated with initial contents 101010100101

Figure 13: LFSR as an exhaustive sequence generator

To overcome problems of this type, weighted PRTPGs can be employed [3]. A block diagram of a weighted PRTPG is shown in Figure 14. If we wish to increase the likelihood of a 1 output, then we can logically OR (NAND) several LFSR outputs together. If we wish to increase the likelihood of a 0 output, we can logically AND (NOR) several LFSR outputs together. Figure 15 shows a weighted PRTPG where the probability that $x^1 = 1$ is 1/2. The other probabilities are $p(x^2 = 1) \approx 3/4$ and $p(x^3 = 1) \approx 1/4$. Of course, we use more complex logic networks to get more refined probabilities. In some cases, we might be concerned with how often inputs change values (transitions on inputs). Figure 16 shows a weighted PRTPG that controls the transition on the inputs as opposed to signal values on the inputs [6]. In this scheme, the PRTPG feeds a decoder whose outputs are ORed in bunches. The outputs of the OR gates are connected to a T type flip-flop which causes the network input bits to change. In this way, one input at a time changes and the relative frequency of input changes are controlled by the number of lines ORed together.

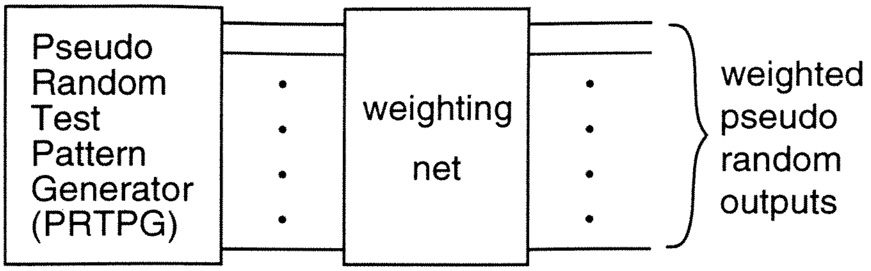

Pseudo Random Test Pattern Generator (PRTPG)
weighting net
weighted pseudo random outputs

Figure 14: A weighted PRTPG.

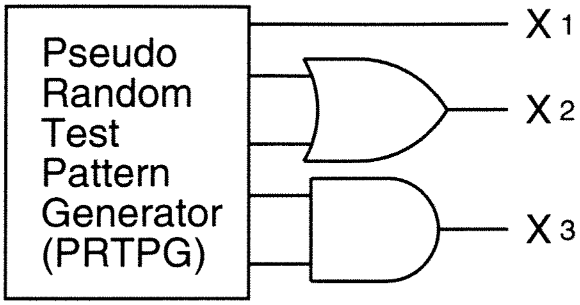


Figure 15: A weighted PRTPG, p(x^1 = 1) =1/2, p (x^2 = 1) = 3/4, p(x^3 = 1) = 1/4.

3.3 Test Coverage in Random Testing

There have been several studies in the test coverage of randomly generated test patterns (see [1–3] for references). Most of these studies deal with combinational circuits, although some work has dealt with sequential networks.

These studies show that for combinational circuits pseudorandom or weighted pseudorandom testing gives good coverage in many practical cases. For sequential testing its effectiveness is not as clear and at best questionable. In any case for sequential testing some weighted and/or adaptive (e.g varying the weights) procedures seem to be necessary to obtain a reasonable fault coverage.

As a further point of interest, a rather convincing argument has been made in [7] to the effect that verifying the coverage of a random test set is at least as difficult as deterministically generating a test set. Hence, if one wants to be sure that certain level of coverage is attained then the advantage offered by the random pattern testing, that is reduction in test pattern generation effort, is almost offset by fault coverage evaluation efforts.

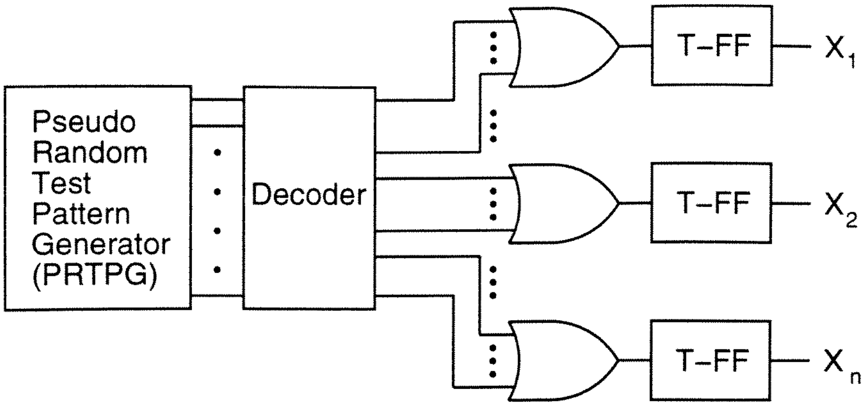


Figure 16: A PRTPG where outputs change with different frequencies.

4 LFSR as a Response Compressor

After digital logic is placed on a chip or on a printed circuit board in a system, one would like to be able to conveniently and effectively test it. Hence, it is desirable to perform testing at normal operating speed using normal network stimulation. One way to do this would be to probe certain key points in a network while it operates on normal input streams. The serial data streams at the probed nodes could be stored and compared with known correct data streams. While this testing approach does little to disturb normal operation, there are some practical difficulties. First, it may be difficult to determine a correct data stream if data collection begins and ends at arbitrary times; second, the length of a data stream to be stored can be prohibitively long.

To solve the first problem, the circuit must be designed so that the same data stream can be repeated at a node, beginning with a special "start" signal and ending with a special "stop" signal. The stimulus resulting in the repeatable stream can be selected to exercise the circuit under test (CUT); the designer must provide for the generation of the start signal, the application of the repeatable test stimulus, and the generation of the stop signal. The second problem can be solved by storing only a *compressed*[1] representation of the observed data stream. For example one could count the number of 1's in the stream (its normalized form is termed as *syndrome*

[1] A word is becoming more prevalent is *compact* as opposed to compress. The difference between the two is that compression is lossless hence the information which is compressed can be fully recovered or regenerated from the compressed data. Whereas compaction entails some of loss of information but the data set after compaction is sufficiently small and can be used to draw some reasonable conclusions about the compacted information.

testing (see chapter 4 of [2] or chapter 10 of [1]) or the number of 0 to 1 or 1 to 0 transitions in the stream (termed as *transition count testing* (see chapter 4 of [2] or chapter 10 of [1]). While these techniques compress the data stream, yet another technique which has gained popularity in the industrial world is based on the use of LFSRs.

In the data compaction technique employing the use of LFSRs, a data stream of length m is considered to be a polynomial of degree $m-1$. This polynomial is shifted into a LFSR that acts as divisor of degree n polynomial (Figure 7). After the data is shifted in, the remainder of the division remains in the LFSR and forms a compressed representation or *signature* of the data stream.

We now analyze the error detecting capability of LFSR signature analysis. Let $M(x)$ be the degree $m-1$ polynomial representing the input data stream and let $P(x)$ be the degree n polynomial forming the divisor used in the LFSR of Figure 7 Then the signature, $S(x)$, is related to $P(x)$ and $M(x)$ in the following way.

$$M(x) = Q(x)P(x) + S(x), \text{where degree } S(x) < n.$$

We let $E(x)$ be an error polynomial, i.e. each nonzero coefficient represents an error in the corresponding bit position. Then for the correct data stream $M(x)$, the erroneous data stream is represented by $M(x) + E(x)$. An undetectable stream is one which satisfies:

$$M(x) + E(x) = Q'(x)P(x) + S(x)$$

That is, $M(x)$ and $M(x) + E(x)$ have the same signature. Formally stated:

Theorem 3: $M(x)$ and $M(x) + E(x)$ have the same signature $S(x)$ if and only if $E(x)$ is multiple of $P(x)$.

An alternative LFSR for computing a signature is the one shown in Figure 9. This implementation may be preferred because it does not require placing EOR gates between shift register stages; thus simplifying possible realization, although it is arguable. The LFSR of Figure 9 produces a different signature than the one shown in Figure 7, but the signatures can still be shown to satisfy Theorem 3.

Theorem 3 is still very useful for determining the error detecting capability of signature analysis. We now discuss some results in this area. First, in a sequence of length m there are $2^m - 1$ possible error polynomials, $E(x)$, which can occur (all polynomials of degree $m-1$ or less except of degree 0). Of all these polynomials, exactly $2^{m-n} - 1$ polynomials are nonzero multiples of $P(x)$, hence there are exactly $2^{m-n} - 1$ undetectable errors. If all the $2^m - 1$ possible errors are equally likely (not a very good assumption but an acceptable starting point), then the probability of failing to detect an error is $(2^{m-n} - 1)/(2^m - 1)$. As $m \rightarrow \infty$, this probability approaches $1/2^n$.

If we assume that a polynomial $P(x)$ has at least two nonzero coefficients, it can be shown that any multiple of $P(x)$ must also have at least two nonzero coefficients. Hence, all single bit errors are detected.

Following theorem can be used to address the issue of detectability of double bit errors.

Theorem 4: The polynomial $x^q + 1$ of least degree that is divisible by an irreducible polynomial of degree n satisfies q = $2^n - 1$.

A double error polynomial $E(x)$ is of the form $x^r(x^s + 1)$ and it follows that if a double error is undetectable then the errors must be separated by at least $2^n - 2$ bits when $P(x)$ is irreducible. This is the maximum separation of undetectable double errors that can be achieved with a polynomial of degree n.

Another kind of error that is important is a burst error. A *burst error of length t* is one where all the erroneous bits occur within an interval of t adjacent bit position. If a polynomial $P(x)$ of degree n has a nonzero coefficient of x^0, i.e. the polynomial is of the form $x^n + p_{n-1}x^{n-1} + ... + 1$, then any multiple of $P(x)$ can not have all its nonzero coefficients appearing in an interval of fewer than $n+1$ consecutive positions. Hence, all undetectable errors appear in burst of at least length $n+1$, and bursts of length n or less are all detected.

In literature the use of both primitive and non-primitive polynomials have been suggested for testing environment. For example Hewlett Packard uses the primitive polynomial $x^{16} + x^9 + x^7 + x^4 + 1$ in its 5004A signature analyzer. This polynomial can detect all single errors, all double errors separated by no more than $2^{16} - 2$ bits, and all bursts of length 16 or less. On the other hand the polynomial $x^n + 1$ have the same error detection capability as above for the primitive polynomial except for the double errors; that is $x^n + 1$ allows many more undetectable double bit errors. But $x^n + 1$ has much simpler implementation. Furthermore, there is no evidence that the double bit errors due to a fault in digital system are more common than the other error patterns. Therefore the choice of a polynomial can still be considered an unsolved problem. Although plenty of literature now exists which deals with the aliasing problem not only in the LFSRs but also with the aliasing in multiple input linear feedback shift registers known as MISRs and which have not been dealt with in this note.

References

[1] M. Abramovici, M. A. Breuer, and A. Friedman, *Digital systems testing and testable design*, Computer Science Press, New York, 1990.

[2] P. H. Bardell, W. H. McAnny, and J. Savir, *Built-in pseudorandom testing of digital circuits*, John Wiley and Sons, New York, 1988.

[3] E. B. Eichelberger, E. Lindbloom, J. A. Waicukauski, and T. W. Williams, *Structured logic testing*, Prentice Hall, Englewood Cliff, NJ, 1991.

[4] S. W. Golomb, *Shift register sequences*, Aegean Park Press, California, revised edition 1982.

[5] W. W. Peterson and F. J. Weldon, *Error correcting codes*, MIT Press, Cambridge, Mass., 2nd edition, 1972.

[6] H. D. Schnurman, E. Lindbloom, and R. G. Carpenter, "The weighted random test pattern generator," *IEEE Trans. Computers*, pp. 695–700, July 1975.

[7] J. J. Shedletsky, "Random testing: practicality vs verified effectiveness," *Proc. 7th Annual Conf. Fault-Tolerant Computing*, pp. 175–179, June 1977.

[8] H. S. Stone, *Discrete mathematical structures and their applications*, Science Research Associates, Chicago, 1973.