Maximal and Near-Maximal Shift Register Sequences: Efficient Event Counters and Easy Discrete Logarithms

Douglas W. Clark
Digital Equipment Corp.
77 Reed Rd. (HLO2-3/J3)
Hudson, MA 01749
doug@ad.enet.dec.com

Lih-Jyh Weng
Digital Equipment Corp.
333 South Street (SHR1-3/E29)
Shrewsbury, MA 01545
weng@cache.enet.dec.com

Abstract

A Linear Feedback Shift Register, or LFSR, can implement an event counter by shifting whenever an event occurs. A single two-input exclusive-OR gate is often the only additional hardware necessary to allow a shift register to generate, by successive shifts, all of its possible nonzero values. The counting application requires that the number of shifts be recoverable from the LFSR contents so that further processing and analysis may be done. Recovering this number from the shift register value corresponds to a problem from number theory and cryptography known as the discrete logarithm. For some sizes of shift register, the maximal-length LFSR implementation requires more than a single gate, and for some the discrete logarithm calculation is hard. This paper proposes for such sizes the use of certain one-gate LFSRs whose sequence lengths are nearly maximal, and which support easy discrete logarithms. These LFSRs have a concise mathematical characterization, and are quite common. The paper concludes by describing an application of these ideas in a computer hardware monitor, and by presenting a table that describes efficient LFSRs of size up to 64 bits.

appears in *IEEE Transaction on Computers 43*, **5 (May 1994), pp. 560-568**

1 Introduction

Using a linear feedback shift register, or LFSR, is an extremely attractive way to generate a sequence of binary words: a single two-input exclusive-OR gate is often the only extra logic needed to make a shift register generate, by successive shifts, all of its possible nonzero values. Applications of LFSRs include error-correcting codes [1, 21, 22], pseudorandom sequence generation for ranging and synchronization [11], test-pattern generation and signature analysis in VLSI circuits [16], and program counters in simple computers [7, 19].

In this paper we study the use of LFSRs as event counters, in which the increment function is implemented by a shift of the register. Compared with the usual logic for a binary increment, an LFSR is wonderfully small and fast. The price for this is the problem of figuring out, after the fact, just how many times the register has been shifted. Recovering the actual number of shifts from the shift register contents corresponds to a problem in number theory and cryptography known as the *discrete logarithm* problem [17, 18]. Crytographic applications of the discrete logarithm favor structures in which the calculation, because it is part of decrypting, is difficult [6]. We, on the other hand, are interested in systems in which the discrete logarithm is easy.

The key insight behind this application is that in a counting instrument, only the increment function needs to be fast. There is no need for any other on-line operations, such as comparison or general addition. Manipulation of the counts is done off-line, and hence can use calculations more expensive than the fast increment.

In the next section of this paper we first briefly review the basic ideas behind LFSRs and then go on to consider the problem of awkward sizes: those sizes of counter in which the maximum-period LFSR implementation requires more than a single gate, or the discrete logarithm problem is hard. (The otherwise appealing size of 32 bits unfortunately has *both* problems!) We propose for such sizes the use of LFSRs whose periods are *very nearly* maximal, and we explore the properties of such registers. Section 3 proves a theorem that precisely characterizes shift registers whose period is greater than one-half the maximum for their size.

Then in Section 4 we consider the discrete logarithm problem, and present a version of the Pohlig-Hellman-Silver algorithm [23]. We show how the use of near-maximal-period LFSRs can lead to faster discrete logarithm calculations and more efficient use of storage for the necessary tables. In

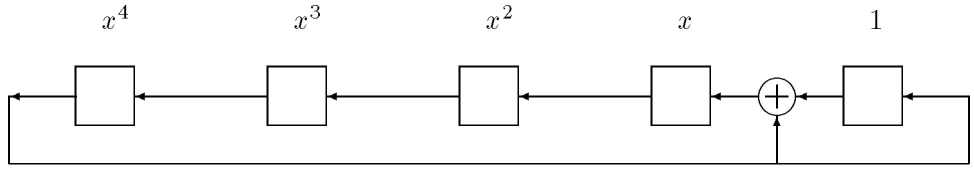

Figure 1: A linear feedback shift register with characteristic polynomial $x^5 + x + 1$. Each stage holds the (binary) coefficient of one term of a degree-4 polynomial. A 2-input exclusive-OR gate, or modulo-2 adder, combines the feedback with the low-order bit. One shift multiplies the polynomial contents by x, modulo the characteristic polynomial.

Section 5 we discuss applications of this method, including our use of it in a hardware monitor for a VAX computer, and also present a table specifying useful long-period shift registers for sizes up to 64 bits. Section 6 contains concluding remarks and discusses possible extensions of the results.

2 LFSRs as Counters

Figure 1 shows a 5-bit example of the simplest type of autonomous (input-free) LFSR: one having a single 2-input exclusive-OR gate between two of its stages, with the end-to-end feedback connected to one input of the gate. For purposes of algebraic manipulation, the contents of an LFSR are commonly treated as the binary coefficients of a polynomial in x. Thus in Figure 1, the value 01010 would correspond to the polynomial $x^3 + x$. The shift register is wired to shift in the direction of increasing exponent, so a single shift is like multiplying by x, neglecting the feedback for the moment. One shift would therefore change the value 01010 into 10100, or the polynomial $(x^3 + x)x = x^4 + x^2$.

The next shift will involve the feedback circuitry and require arithmetic on the coefficients. The polynomial interpretation of the function of this circuitry is as follows. The exclusive-OR gate performs coefficient arithmetic modulo 2 (that is, in the Galois field GF(2)), so addition is the same as subtraction. The feedback connections in Figure 1 therefore mean that whenever a 1 is shifted out of the x^4 position, the polynomial $x^5 + x + 1$ is effectively subtracted from what would otherwise have been the shifted contents of the register. Thus a shift may be thought of as a polynomial multiplication by x (over GF(2)), modulo the shift register's *characteristic* *polynomial* x^5+x+1. In Figure 1, a shift of the polynomial contents x^4+x^2 would yield the polynomial

$$(x^4+x^2)x \bmod x^5+x+1 \;=\; x^3+x+1 \, .$$

Suppose we initialize the LFSR to the (polynomial) 1, and then shift it L times, thereby recording L occurences of some experimental event. This amounts to L multiplications by x, subject to the feedback circuitry, and the polynomial result is therefore the remainder of $x^L/(x^5+x+1)$, or $x^L \bmod x^5+x+1$.

Figure 1 is an instance of the general form in which an LFSR can have a two-input exclusive-OR gate for each shift-register stage except the low-order one. There are several other circuit arrangements with equivalent behavior, and richer hardware structures can perform other polynomial operations (see, e.g., [12, 22, 25]). The characteristic polynomial in the general case of an n-bit LFSR has the form $p(x) = x^n + c_{n-1}x^{n-1} + \cdots + c_1x^1 + 1$. The c_i are either 1 or 0, corresponding to the presence or absence of an exclusive-OR gate; and because c_0 is always 1, $p(x)$ is never divisible by x.

Of great interest are the *periods* of such circuits: when initialized to 1, after how many shifts will 1 reappear? From the polynomial point of view, so to speak, this is the same as asking: what is the smallest m such that

$$x^m \equiv 1 \pmod{p(x)}$$

or, equivalently, what is the smallest m such that $p(x)$ evenly divides x^m-1 over GF(2)? Thus we may speak of the period of a characteristic polynomial just as we do the period of an LFSR. In the counting application, the period represents the value at which the counter overflows, so the period must be greater than the maximum expected count.

The period of the shift register of Figure 1 is 21. The maximum period for a 5-bit LFSR is clearly $2^5-1=31$ (the all-zero value never appears), and is achievable with a different characteristic polynomial: x^5+x^2+1. The maximum possible period for an n-bit LFSR is 2^n-1, and it is well known that characteristic polynomials with maximum period, called *primitive polynomials*, exist for all n. Maximum-period LFSRs, primitive polynomials, and their corresponding algebras GF(2^n), are the focus of most work with shift registers. Many applications need the maximum period or the Galois field, but the counting application does not. The idea of using an LFSR to count in a Galois field appears in Peterson [21].

All primitive polynomials are *irreducible*—not divisible by any other polynomial over GF(2)—but some irreducible polynomials are not primitive, that is, some have periods less than the maximum. In fact, it is known that the period of an irreducible but nonprimitive characteristic polynomial of degree n is a proper divisor of $2^n - 1$, and hence is at most one-third of the maximum. Such a polynomial partitions all of a shift register's $2^n - 1$ nonzero states into a number of equal-sized cycles, one of which contains the polynomial value 1.

Thus it would seem that in our application we would always choose an LFSR whose characteristic polynomial was primitive. Certainly when a characteristic polynomial of the desired degree is a *trinomial*, implying a one-gate LFSR implementation, it would have great appeal. Unfortunately, however, for many sizes of shift register, no primitive trinomial exists. There are 30 sizes between 1 and 64 that have this problem [26]; in particular, there are no primitive trinomials—indeed, no irreducible trinomials—for any size that is a multiple of 8 [24], sizes that might otherwise be favored by current computer architectures.

The alternative of using, for some desired shift register size, an irreducible but nonprimitive trinomial is unattractive because its period can be at most one-third of the maximum for that size. This would seem an inefficient use of the hardware, since an LFSR of fewer bits could potentially do as well. Primitive polynomials with more than three terms are another possibility, but unfortunately four-term polynomials are excluded because they are all divisible by $x + 1$ and hence not primitive. Primitive polynomials of five terms may be unattractive due to their extra hardware cost (although Wang and McCluskey [25] show that an alternative LFSR wiring pattern for some of these polynomials can use two, not three, exclusive-OR gates, at the cost of increased combinational delay). And for some values of n, as we shall see in Section 4, the discrete logarithm calculation for the maximum period is hard, and this consideration may rule out using a primitive polynomial of any number of terms.

There remains the option of using a *reducible* trinomial: one that is the product (over GF(2)) of two or more irreducible factors. Reducible polynomials have been studied as a way to produce shift registers with prescribed periods [9]. The periods of such polynomials can be extremely close to the maximum and the discrete logarithm calculation can be easier than it is for the maximum period. Consider, for example, the useful size of 32 bits, which has no primitive trinomial. If we make an LFSR using the reducible trinomial $x^{32} + x^{15} + 1$, we get a period length that is over 99.95 percent of the maximum $2^{32}-1$, and we also get a discrete logarithm procedure that is faster and much more space-efficient than the maximum-period one (details forthcoming in Section 5.1).

Figure 1 illustrated a reducible characteristic polynomial,

$$x^5 + x + 1 = (x^3 + x^2 + 1)(x^2 + x + 1) ,$$

whose period of 21 is 68 percent of the maximum period 31. As we will see in Section 5, reducible polynomials with long periods are quite common. Reducible polynomials can also have *short* periods, however: $x^{32}+x+1$, for example, is reducible and has period only 1023. It is interesting, therefore, to ask what properties of these polynomials might give them long periods. We turn next to this question.

3 Guaranteeing Long Periods

In this section we will characterize in a precise way those polynomials whose periods are greater than half the maximum for their degrees. A characteristic polynomial of smaller period would be inefficient in the sense that a different polynomial of lesser degree could have a longer period.

The period of a reducible polynomial is a function of the periods of its polynomial factors. Let the polynomial $p(x)$ factor (over GF(2), as usual) this way:

$$p(x) = p_1(x)^{b_1} p_2(x)^{b_2} \cdots p_h(x)^{b_h} ,$$

where the $p_i(x)$ are distinct and irreducible. Then the period of $p(x)$ may be calculated as follows [10]:

$$\text{Period}\,[p(x)] = \text{lcm}\,(\text{Period}\,[p_1(x)^{b_1}], \ldots, \text{Period}\,[p_h(x)^{b_h}]) \tag{1}$$

and for each factor $p_j(x)^{b_j}$,

$$\text{Period}\,[p_j(x)^{b_j}] = 2^{\lceil \log_2 b_j \rceil}\,\text{Period}\,[p_j(x)] . \tag{2}$$

Equation (1) suggests that for period maximization, the polynomial factors' periods ought not to have any integer factors in common. Equation (2) suggests that repeated polynomial factors might be bad. We will now formalize these observations.

Theorem (Long Periods): Let $p(x)$ be an arbitrary polynomial over GF(2) of degree n, not divisible by x. Let m be the smallest integer such that $p(x)$ evenly divides $x^m - 1$ over GF(2). Then m, the period of $p(x)$, exceeds 2^{n-1} if and only if all four of the following conditions are true:

A. $p(x)$ has no factor of degree 1;

B. $p(x)$ has no repeated factors;

C. the irreducible factors of $p(x)$ are all primitive; and

D. the degrees of the irreducible factors of $p(x)$ are pairwise relatively prime.

Proof: We first show that if all four conditions hold then $m > 2^{n-1}$. Let

$$p(x) = p_1(x)p_2(x)\cdots p_h(x)$$

be the factorization of $p(x)$ into its irreducible factors and let n_i be the degree of factor $p_i(x)$, $1 \leq i \leq h$. Condition B allows us to omit exponents on the $p_i(x)$. By condition C, all the factors of $p(x)$ are primitive, so the period of each $p_i(x)$ is $2^{n_i}-1$. Condition D says the n_i are pairwise relatively prime; this implies that the periods $2^{n_i}-1$ are also pairwise relatively prime [14, p. 272]. Thus m, the period of $p(x)$, which is the least common multiple of the periods of the $p_i(x)$, is just their product:

$$m = \prod_{i=1}^{h}(2^{n_i} - 1)\ .$$

We pull 2^n out of the product and write

$$m = 2^n \prod_{i=1}^{h}(1 - \frac{1}{2^{n_i}})\ .$$

Now $m > 2^{n-1}$ if

$$\prod_{i=1}^{h}(1 - \frac{1}{2^{n_i}}) > \frac{1}{2}\ . \tag{3}$$

This we can show by first observing [13] that

$$\prod_{i=1}^{h}(1 - \frac{1}{2^{n_i}}) \geq 1 - \sum_{i=1}^{h}\frac{1}{2^{n_i}}\ .$$

Because the n_i are distinct (condition D) and are all greater than 1 (condition A),

$$\sum_{i=1}^{h}\frac{1}{2^{n_i}} < (\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots) = \frac{1}{2}$$

and (3) immediately follows. (A similar line of argument was used by Parhami in his analysis of certain residue number systems [20].) This completes the first part of the proof.

We will now prove that if $m > 2^{n-1}$ then all four conditions hold. We will argue by contradiction, demonstrating that the violation of any condition would imply that $m \leq 2^{n-1}$.

For notational simplicity, let $q(x)^b$ denote an arbitrary factor of $p(x)$, where $q(x)$ is irreducible and has degree d. Let s be the period of $q(x)$ and let $\tilde{m}$ be the period of $q(x)^b$. Then from Equation (2) we know that

$$\tilde{m} = 2^{\lceil \log_2 b \rceil} s \ . \tag{4}$$

Since $q(x)^b$ has degree bd, the period of the product of all the other factors of $p(x)$ can be at most $2^{n-bd} - 1$. Of course there may not be any other factors, but if there are, then the least common multiple formula for m (Equation (1)) is bounded by the product of the factors' periods:

$$m \leq \tilde{m}(2^{n-bd} - 1) \ .$$

If the violation of some condition leads to

$$\tilde{m} \leq 2^{bd-1} \ , \tag{5}$$

then we will immediately have $m \leq 2^{n-1}$. In the case that there are no polynomial factors of $p(x)$ other than $q(x)^b$ we have $m = \tilde{m}$ and $n = bd$, so (5) is the same as $m \leq 2^{n-1}$. Thus, showing that (5) follows from a violation of some condition will establish, by contradiction, that condition's validity. We will look at the first three conditions in this way, and finish with a separate argument for condition D.

First, suppose that $q(x)^b$ violates condition A. Because x cannot be a factor of $p(x)$, the only possibility of degree $d = 1$ is $q(x) = x + 1$, which has period 1. (We note in passing that if $p(x)$ is itself $x + 1$, then the theorem holds vacuously.) Then (4) simplifies to

$$\tilde{m} = 2^{\lceil \log_2 b \rceil},$$

and to show (5) we need only show that

$$2^{\lceil \log_2 b \rceil} \leq 2^{b-1}, \tag{6}$$

which is clearly true for all positive b. Thus no power of $x + 1$ can be a factor of $p(x)$, and condition A is established. We may therefore assume that $d > 1$ in what follows.

Now suppose that $q(x)^b$ violates condition B, so $b > 1$. The period of $q(x)$ is at most $2^d - 1$. Using this fact together with (6) we can bound $\tilde{m}$ as follows:

$$\tilde{m} \leq 2^{\lceil \log_2 b \rceil}(2^d - 1) \leq 2^{b-1}(2^d - 1) < 2^{b+d-1} .$$

We complete this chain of inequalities by observing that $b + d \leq bd$ because both $b > 1$ and $d > 1$, and therefore

$$\tilde{m} < 2^{b+d-1} \leq 2^{bd-1} ,$$

which satisfies (5). Condition B is thus established: $p(x)$ can have no repeated factors.

Condition C asserts that $q(x)$ must be primitive. Were it not primitive, $\tilde{m}$ would properly divide $2^d - 1$. This number is odd, so the biggest that $\tilde{m}$ could be is one-third of $2^d - 1$. Thus using $d > 1$ we may write

$$\tilde{m} \leq \frac{1}{3}(2^d - 1) < 2^{d-1},$$

which satisfies (5) because we now know that $b = 1$.

Finally, suppose condition D is violated: some two polynomial factors have degrees that are not relatively prime. Because their periods—we now know—are of the form $2^d - 1$, the periods are also not relatively prime [14, p. 272]. The periods' smallest common divisor must be at least 3, so the least-common-multiple formula for m guarantees that $m < 2^{n-1}$. This completes the proof. □

4 Count Recovery with Discrete Logarithms

We turn now to the problem of recovering the integer number of shifts from the polynomial contents of an LFSR. If the register is not too wide, this conversion could be done in several obvious ways, including simply creating a lookup table of all possible values. For wide registers we need a more sophisticated technique. In our application, we are willing to invest in a large amount of up-front precomputation in the count recovery algorithm because we will want to do many recoveries for the same shift register. Thus we are prepared to construct tables and calculate constants that will be re-used in every count recovery.

Suppose we have an n-bit LFSR with characteristic polynomial $p(x)$, of period m, and suppose it is initialized to 1. The set of values generated by such an LFSR is one representation of the cyclic group of order m. The group operation is polynomial multiplication modulo $p(x)$ over GF(2); the identity element is the polynomial 1; and the generator of the group is the polynomial x. The elements of the group are just $x^j \bmod p(x)$, $0 \leq j < m$. For notational convenience and because much of the following discussion applies to the abstract group independent of representation, we will denote the group by G and the generator by g, so the elements of G will be $g^0 = 1, g^1, g^2, \ldots, g^{m-1}$. G is isomorphic to the additive group of integers modulo m [3].

The *discrete logarithm* problem is: given an element $y \in G$, find the integer L in the range $[0 \ldots m-1]$ such that $g^L = y$. We may write $L = \log_g y$. (The discrete logarithm is sometimes called the *index* function; see the surveys by Odlyzko [18] and McCurley [17].) If an LFSR is initialized to 1, then L is the number of shifts that yield shift-register contents y, provided the number of shifts is less than m.

We will use the Pohlig-Hellman-Silver algorithm [23], originally developed for the multiplicative group of a Galois field (corresponding to a maximum-period, or primitive, characteristic polynomial), and generalized to any cyclic group by Massey [15]. The key to this algorithm is the Chinese remainder theorem: instead of calculating L directly, we will find the *residues* of L modulo certain factors of m and then use the theorem to compute L itself by combining those residues. This method is attractive when all these factors are small compared to the period. (This leads cryptographers to shun such groups and favor instead groups whose orders have large prime factors.)

Let the order m of the cyclic group G be expressed as the product $m = m_1 m_2 \cdots m_k$ of k factors that are pairwise relatively prime. We want the residues

$$r_i = L \bmod m_i \ , \quad 1 \leq i \leq k \ . \tag{7}$$

The Chinese remainder theorem says that there is exactly one value of L in $[0 \ldots m-1]$ that satisfies the simultaneous equations (7):

$$L = \sum_{i=1}^{k} r_i \frac{m}{m_i} v_i \mod m \ , \tag{8}$$

where each v_i is chosen to satisfy

$$\frac{m}{m_i} v_i \equiv 1 \pmod{m_i} \ .$$

Table 1: Polynomials, Group, Subgroups, and Logarithms for Figure 1

LFSR	polynomial	G	G_1	G_2	L	r_1	r_2
00001	1	g^0	g_1^0	g_2^0	0	0	0
00010	x	g^1			1	1	1
00100	x^2	g^2			2	2	2
01000	x^3	g^3		g_2^1	3	0	3
10000	x^4	g^4			4	1	4
00011	$x+1$	g^5			5	2	5
00110	x^2+x	g^6		g_2^2	6	0	6
01100	x^3+x^2	g^7	g_1^1		7	1	0
11000	x^4+x^3	g^8			8	2	1
10011	x^4+x+1	g^9		g_2^3	9	0	2
00101	x^2+1	g^{10}			10	1	3
01010	x^3+x	g^{11}			11	2	4
10100	x^4+x^2	g^{12}		g_2^4	12	0	5
01011	x^3+x+1	g^{13}			13	1	6
10110	x^4+x^2+x	g^{14}	g_1^2		14	2	0
01111	x^3+x^2+x+1	g^{15}		g_2^5	15	0	1
11110	$x^4+x^3+x^2+x$	g^{16}			16	1	2
11111	$x^4+x^3+x^2+x+1$	g^{17}			17	2	3
11101	$x^4+x^3+x^2+1$	g^{18}		g_2^6	18	0	4
11001	x^4+x^3+1	g^{19}			19	1	5
10001	x^4+1	g^{20}			20	2	6

The precomputed v_i can be found via the extended Euclidean algorithm [14].

Now we need a way to find the r_i. Recalling our original input $y = g^L$, we define, for $1 \leq i \leq k$, $g_i = g^{m/m_i}$ and $y_i = y^{m/m_i}$, where the exponentiations are done in G. Because m_i is relatively prime to m/m_i, y_i must match one of $g_i^0, g_i^1, g_i^2, \ldots, g_i^{m_i-1}$, and this match determines r_i:

$$y_i = y^{m/m_i} = g^{Lm/m_i} = g^{r_i m/m_i} = g_i^{r_i} .$$

In other words, r_i is itself the discrete logarithm of y_i in the cyclic subgroup of G generated by g_i—a formulation due to Massey [15]. This subgroup has order only m_i, so if all the factors of m are small, it is feasible to do the subgroup discrete logarithms by table lookup. (When m has a large factor that is a power of a prime, Pohlig and Helmann give a more complicated method that saves table space [23].) The main work in the calculation would then be the k exponentiations of y in G. In our setting these operations would be polynomial exponentiations over GF(2), modulo some characteristic polynomial $p(x)$ of period m. Each exponentiation takes at most $2\lceil\log_2 m/m_i\rceil$ polynomial multiplications [14].

Consider an example using the counter of Figure 1, whose (reducible) characteristic polynomial is x^5+x+1, and whose period $m=21$. Table 1 illustrates the algebraic interpretation of this counter. Call the factors $m_1=3$ and $m_2=7$. Then the assignments $v_1=1$ and $v_2=5$ will work because

$$v_1\frac{m}{m_1}=1\cdot\frac{21}{3}=7\equiv 1 \pmod{m_1}$$

and

$$v_2\frac{m}{m_2}=5\cdot\frac{21}{7}=15\equiv 1 \pmod{m_2}\,.$$

Suppose our input $y=g^L$ is the 5-bit binary value 01010, representing the polynomial x^3+x. We compute the y_i by repeated squarings and multiplications in G:

$$\begin{aligned}
y_1 &= y^{m/m_1}=y^7=(y^2)^2y^2y\\
&= x^{21}+x^{19}+x^{17}+x^{15}+x^{13}+x^{11}+x^9+x^7 \mod x^5+x+1\\
&= x^4+x^2+x
\end{aligned}$$

$$\begin{aligned}
y_2 &= y^{m/m_2}=y^3=(y^2)y\\
&= x^9+x^7+x^5+x^3 \mod x^5+x+1\\
&= x^4+x^2\,.
\end{aligned}$$

We now look up the y_i in Table 1 by first comparing y_1 with the three elements of G_1: g_1^0, g_1^1, g_1^2; we find that $y_1=g_1^2$, so $r_1=2$. Similarly, if we compare y_2 with the seven elements of G_2, we find that $y_2=g_2^4$, so $r_2=4$. Then we roll up the r_i using equation (8):

$$\begin{aligned}
L &= (r_1\frac{m}{m_1}v_1+r_2\frac{m}{m_2}v_2) \bmod m\\
&= (2\cdot 7\cdot 1+4\cdot 3\cdot 5) \bmod 21\\
&= 11\,.
\end{aligned}$$

We can check that this is right by cheating: in Table 1, we can see that $y=x^3+x=g^{11}$ so y does indeed have discrete logarithm 11. The right-hand columns of the table show that row 11 is the only one with $r_1 = 2$ and $r_2 = 4$, as promised by the Chinese remainder theorem.

The method described above will work for any characteristic polynomial not divisible by x, but for those reducible polynomials that are covered by the long-period theorem, the method can be adapted to use narrower tables and smaller exponents on the input polynomial y. Suppose the characteristic polynomial $p(x)$ of degree n and period m has primitive factor $p_j(x)$ of degree n_j, and let its period be $\hat{m} = 2^{n_j} - 1$. Some of the factors of m will also be factors of $\hat{m}$. Let m_i be one such factor.

The straightforward application of the discrete logarithm algorithm would have us store, for factor m_i, a table of n-bit polynomial representations of the m_i elements of the subgroup G_i:

$$1, x^{m/m_i}, x^{2m/m_i}, \ldots, x^{(m_i-1)m/m_i} \mod p(x)\,.$$

For an input y we would then get the residue r_i by finding the table entry that matched $y^{m/m_i} \bmod p(x)$. But we could instead represent the subgroup G_i as a table of n_j-bit polynomials

$$1, x^{\hat{m}/m_i}, x^{2\hat{m}/m_i}, \ldots, x^{(m_i-1)\hat{m}/m_i} \mod p_j(x)\,,$$

and then find r_i by looking up the value $y^{\hat{m}/m_i} \bmod p_j(x)$.

It is not hard to see that this faster and more compact approach does in fact find the same r_i. We know that

$$y^{m/m_i} \equiv x^{r_i m/m_i} \pmod{p(x)}$$

and therefore that

$$y^{m/m_i} \equiv x^{r_i m/m_i} \pmod{p_j(x)}\,,$$

since $p_j(x)$ is a factor of $p(x)$. Now we divide each exponent by the integer $m/\hat{m}$, which is relatively prime to $\hat{m}$, the period of $p_j(x)$. This property guarantees that the resulting subgroup elements exist, and therefore

$$y^{\hat{m}/m_i} \equiv x^{r_i \hat{m}/m_i} \pmod{p_j(x)}\,,$$

which shows that the more efficient procedure does in fact find the same residue r_i as the original procedure. For each factor m_i, therefore, we need a table of m_i elements whose width in bits is equal to the degree of that polynomial factor whose own period has m_i as a factor.

Table 2: Period Factors and Discrete-Log Constants for $x^{32} + x^{15} + 1$

i	m_i	v_i
1	7^2	33
2	23	16
3	89	27
4	127	48
5	337	320

Compared with a primitive polynomial, then, a reducible trinomial offers four potential advantages in the counting application: first, fewer LFSR logic gates for degrees that have no primitive trinomial; second, smaller exponents on the discrete logarithm algorithm's input polynomial; third, shorter discrete log tables when the period has smaller factors; and finally, narrower log tables, according to the polynomial factors of the trinomial. The table space savings can be enormous, as we will shortly see.

5 Applications

In this section we look more closely at a particular 32-bit LFSR, report on our use of a 36-bit LFSR in a hardware monitor, and give a table of useful trinomials, their periods, and the sizes of the required discrete log tables.

5.1 Details of 32-bit LFSR

Like all multiples of 8, degree 32 has no irreducible, and hence no primitive, trinomial [24]; the simplest primitive polynomials, such as $x^{32}+x^{31}+x^5+x^4+1$, have five terms because all four-term polynomials are reducible. If we use one of these in an LFSR implementation, we can do the discrete logarithms with the method of Section 4 by committing a great deal of space for the tables; this follows from the prime factorization of the maximum period:

$$2^{32} - 1 = 3 \cdot 5 \cdot 17 \cdot 257 \cdot 65537$$

The five tables, each storing 32-bit values, would consume a total of 2,106,208 bits (exclusive of any lookup infrastructure).

The attractive alternative is the reducible trinomial mentioned in Section 2, whose irreducible factors are as follows:

$$
\begin{aligned}
x^{32} + x^{15} + 1 \;&=\; (x^{21} + x^{19} + x^{15} + x^{13} + x^{12} + x^{10} + x^{9} + x^{8} + x^{7} + x^{6} + x^{4} + x^{2} + 1) \\
&\quad \cdot (x^{11} + x^{9} + x^{7} + x^{2} + 1)\,.
\end{aligned}
$$

(The large number of terms in these factors is of little concern since the factors themselves have no effect on the LFSR hardware.) The period of this trinomial is greater than 99.95 percent of the maximum—implying, according to the long-period theorem, that both polynomial factors are primitive. It is also *smoother*—has a smaller largest factor—than the maximum period:

$$4292868097 = 7^2 \cdot 23 \cdot 89 \cdot 127 \cdot 337\,.$$

The factors 7^2, 127, and 337 are themselves the factors of the period of the degree-21 polynomial factor. Factors 23 and 89 are owned by the degree-11 polynomial factor.

The minimum space requirements for the discrete logarithm are therefore: three tables of lengths 49, 127, and 337, each 21 bits wide; and two tables of 23 and 89 entries, each 11 bits wide. The grand total is 12,005 bits, or less than 0.57 percent of the requirement for a maximum-period LFSR.

Table 2 gives the constants v_i for this polynomial. There is one other trinomial of degree 32 whose period is greater than 2^{31}, but its period is less than the one discussed above, and its log tables are bigger.

5.2 VAX hardware monitor

With several colleagues at Digital we designed and built a VAX hardware monitor that uses an LFSR for counting [5]. This application was the original motivation for this paper. The monitor implements Emer's micro-Program Counter histogram technique [8]: it maintains a count for every control store address and increments the count each time the corresponding microinstruction is executed. The counts are kept in a static RAM addressed by the micro-PC. In every machine cycle a RAM location must be read, its contents incremented, and the updated count written back. Interpretation of the counts and any necessary arithmetic using them is done off-line, after a measurement experiment is complete. Thus while the increment must be extremely fast, a slow off-line conversion is quite tolerable, so an LFSR counter is ideal. Its tiny size is a boon too. (A pipelined implementation is feasible, of course. It would allow a standard binary increment, but would cost extra hardware: the incrementer itself, pipeline latches, multiplexors, and bypassing logic for the case in which the same location is incremented in successive cycles.)

For the VAX application, we wanted to measure about an hour's worth of 45-nanosecond cycles, so the count width needed to be in the vicinity of 36 bits. Happily, there is a primitive trinomial of degree 36, $x^{36} + x^{11} + 1$, whose period is smooth: its biggest factor is 109. Thus the discrete logarithm method of Section 4 was well suited to our needs. (At the time we designed this monitor, in fact, we believed that a primitive polynomial was *required*. For 36 bits, in any case, there is no reducible trinomial whose log tables are smaller than those of the primitive one.)

A measurement experiment starts with the initialization of all of the histogram counts to 1. Then programs of interest are run on the computer, without interference from the monitor. At the end of the measurement (and before any count has overflowed), further counting is disabled, the RAM contents read out, and the discrete logarithms calculated. The resulting integer counts are matched against the microcode listing file. Information from the listing file, such as labels and comments, guides the subsequent manipulation of these counts to yield performance statistics of interest to the experimenter. For example, one could calculate the total time spent executing a particular VAX opcode by summing the histogram counts for all of that opcode's microinstructions. Many other useful statistics can be imagined.

This monitor has been used for a host of measurements at Digital, some of which are reported in [2, 5], some of which have been used in the development of subsequent VAX processors, and some of which have been used to evaluate and tune software of various kinds.

5.3 Table of practical trinomials

Table 3 describes trinomials useful for counters up to 64 bits in width. We think 64 bits is a big enough counter for the foreseeable future: such a counter could count 10-picosecond events for five years before overflowing. We constructed the table by consulting Golumb [12] for trinomials up to degree 36, Zierler and Brillhart [26] for primitive trinomials up to degree 64, and using the Maple system [4] to investigate reducible trinomials between degrees 36 and 64. The table includes only trinomials of period greater than half the maximum for their degrees. (For each entry $x^n + x^a + 1$ there is a corresponding trinomial $x^n + x^{n-a} + 1$ whose period is the same, and whose polynomial factors have the same degrees as the listed one.)

Table 3: Trinomial Periods and Logarithm-Table Sizes

$x^n + x^a + 1$		polynomial		$\log_2$ table
n	a	factors' degrees	$\frac{\text{period}}{2^n-1}$	size (bits)
2	1	primitive	1.000	2.6
3	1	primitive	1.000	4.4
4	1	primitive	1.000	5.0
5	2	primitive	1.000	7.3
	1	3, 2	.677	4.8
6	1	primitive	1.000	6.6
7	1, 3	primitive	1.000	9.8
	2	5, 2	.732	7.3
8	3	5, 3	.851	7.5
9	4	primitive	1.000	9.5
	2	5, 4	.910	7.5
10	3	primitive	1.000	8.8
11	2	primitive	1.000	10.3
	3	6, 5	.954	8.0
12	1	5, 4, 3	.795	7.7
13	3	7, 6	.977	9.9
	1	8, 5	.965	8.5
14	1	7, 5, 2	.721	10.0
15	1, 4, 7	primitive	1.000	11.5
16	7	11, 5	0.968	10.4
⋮	⋮	⋮	⋮	⋮

For each degree, the table gives the trinomial of longest period—often a primitive one—and the logarithm of the size of the required decoding tables. It then lists, in order of decreasing period, any reducible trinomials whose decoding tables are smaller than any already on the list. Periods are reported as fractions of the maximum. Table 3 therefore illustrates the tradeoff between period and decoding table size: for a particular degree, a sacrifice in period is acceptable only if the resulting tables are smaller. For some degrees, a primitive trinomial has the smallest tables or is the only trinomial of long period, and for these degrees no reducible trinomial is shown (degree 36 is an example).

For reducible trinomials, Table 3 gives the degrees of the polynomial factors. From these the exact period of the trinomial can be calculated by

Table 3 (continued): Trinomial Periods and Logarithm-Table Sizes

$x^n + x^a + 1$		polynomial		$\log_2$ table
n	a	factors' degrees	$\frac{\text{period}}{2^n-1}$	size (bits)
17	3, 5, 6	primitive	1.000	21.1
	2	14, 3	0.875	11.3
18	7	primitive	1.000	10.0
19	6, 7	12, 7	0.992	10.3
	1	7, 5, 4, 3	0.788	10.1
20	3	primitive	1.000	11.1
	7	11, 7, 2	0.744	11.1
21	2	primitive	1.000	13.4
22	1	primitive	1.000	14.1
	9	11, 7, 4	0.930	11.1
23	5, 9	primitive	1.000	22.0
24	5	13, 11	0.999	16.7
25	3, 7	primitive	1.000	15.5
26	5	15, 11	0.999	12.0
27	8	19, 8	0.996	23.2
	10, 13	22, 5	0.969	14.1
28	3, 9, 13	primitive	1.000	13.1
29	2	primitive	1.000	16.6
	9	21, 8	0.996	13.4
	4	11, 9, 7, 2	0.742	11.5
30	7	19, 11	> 0.999	23.2
	13	22, 5, 3	0.848	14.1
31	3, 6, 7, 13	primitive	1.000	36.0
32	15	21, 11	> 0.999	13.6
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

appeal to the long-period theorem of Section 3: it is just the product of the $2^d - 1$ for degrees d.

The size of the discrete logarithm tables is calculated according to the method of Section 4. There is a separate decoding table for each integer factor of the trinomial's period, including powers of primes. The width of each table in bits is the degree of the polynomial factor whose own period includes that integer factor. Of course any efficient lookup procedure would need extra bits for various reasons; we count only the minimum required to

Table 3 (continued): Trinomial Periods and Logarithm-Table Sizes

$x^n + x^a + 1$		polynomial		$\log_2$ table
n	a	factors' degrees	$\frac{\text{period}}{2^n-1}$	size (bits)
33	13	primitive	1.000	24.2
34	13	19, 15	$>$ 0.999	23.2
	11	17, 7, 5, 3, 2	0.631	21.1
35	2	primitive	1.000	22.0
	8	18, 17	$>$ 0.999	21.1
36	11	primitive	1.000	13.3
37	1	19, 18	$>$ 0.999	23.2
	7	25, 12	$>$ 0.999	15.9
	15	16, 9, 7, 5	0.959	12.6
38	15	35, 3	0.875	22.0
39	4, 8, 14	primitive	1.000	22.3
	11	23, 11, 5	0.968	22.0
	19	13, 11, 7, 5, 3	0.841	16.7
40	19	33, 7	.992	24.2
41	3, 20	primitive	1.000	32.7
	8	25, 9, 7	0.990	15.9
42	11	23, 15, 4	0.937	22.0
43	9	34, 9	0.998	22.5
	7	39, 4	0.938	22.3
	3, 19	40, 3	0.875	21.2
44	21	23, 13, 8	0.996	22.0
45	17	23, 22	$>$ 0.999	22.0
	7	29, 16	$>$ 0.999	16.7
46	15	29, 17	$>$ 0.999	21.2
47	5, 14, 20, 21	primitive	1.000	29.2
48	5	29, 19	$>$ 0.999	23.3
⋮	⋮	⋮	⋮	⋮

hold the discrete log tables themselves.

Practical trinomials of near-maximum period are quite common. Of the 30 degrees covered in Table 3 that have no primitive trinomial, 22 have a reducible trinomial of period exceeding 99 percent of the maximum. This is clearly less likely to be true of small degrees, and of the remaining 8 degrees, 5 have value less than 17. There are 33 degrees covered by the table that

Table 3 (continued): Trinomial Periods and Logarithm-Table Sizes

$x^n + x^a + 1$		polynomial		$\log_2$ table
n	a	factors' degrees	$\frac{\text{period}}{2^n-1}$	size (bits)
49	9, 12, 15, 22	primitive	1.000	47.6
50	7	29, 19, 2	0.750	23.3
51	22	28, 23	> 0.999	22.0
	16	29, 22	> 0.999	16.8
52	3, 19, 21	primitive	1.000	19.3
	15	44, 5, 3	0.848	17.2
53	11	27, 26	> 0.999	22.8
	4	51, 2	0.750	22.8
54	5	29, 18, 7	0.992	16.6
55	24	primitive	1.000	23.4
56	17	31, 25	> 0.999	36.0
	3	33, 23	> 0.999	24.5
	9	51, 5	0.969	22.8
57	7, 22	primitive	1.000	26.6
	28	43, 9, 5	0.967	26.4
58	19	primitive	1.000	27.4
59	21	30, 29	> 0.999	16.8
60	1, 11	primitive	1.000	16.9
61	30	37, 24	> 0.999	34.4
	6	43, 18	> 0.999	26.4
	21	52, 9	0.998	19.3
62	3	37, 25	> 0.999	34.4
	27	55, 7	0.992	23.4
63	1, 5, 31	primitive	1.000	25.5
	17	23, 21, 19	> 0.999	23.7
	20	51, 7, 5	0.961	22.8
64	15	39, 25	> 0.999	22.3

do have a primitive trinomial, and of these, 14 have at least one reducible trinomial with smaller log tables.

6 Conclusion

In this paper we have combined some old results from the theory of linear feedback shift registers with a computational method from cryptography to show how to make very big event counters that are very cheap and very fast. While earlier work has focused on maximal-period shift registers and primitive polynomials, we have shown that reducible polynomials are often the best choice for this application, with respect to both hardware economy and computational efficiency. We proved that such polynomials can be precisely characterized, and showed that useful trinomials are quite common.

While we have emphasized the instrumentation application, in which efficient hardware and easy discrete logarithms are both important, other applications and extensions of our results are apparent:

- Because the long-period theorem and the discrete logarithm algorithm apply to all polynomials, not just to trinomials, applications not requiring an LFSR of absolutely minimal logic (one gate) could use a long-period polynomial with more terms. Such polynomials could have longer periods and/or smaller logarithm tables than the best trinomial alternative of the same degree.

- Because the discrete logarithm algorithm will work for any polynomial not divisible by x, applications whose primary constraint is the size of the log tables rather than the size of the hardware could use a polynomial of at least the desired period but of higher than necessary degree, purely for the size of its log tables.

- Some applications have no use for the discrete logarithm at all, needing only an efficient generator with a long period. When a primitive trinomial does not exist for some degree, a reducible trinomial of nearly maximal period almost always does.

Acknowledgments. Lyle Ramshaw was a great help in the preparation of this paper: he gave us some early guidance on the discrete logarithm, sharpened several of the arguments in Section 3, suggested the format of Table 3, and carefully reviewed an early draft. Bob Stewart first suggested using an LFSR in the VAX hardware monitor described in Section 5.2, and also introduced us. Tryggve Fossum provided algebraic assistance at a critical time.

The hardware monitor and the associated software were developed by Pete Bannon, Walter Beach, Dave Laurello, Dave Vaughan, and Bei-Pong Wang. Some of our polynomial manipulations were done with the Maple system [4], which came to us from the Symbolic Computation Group, Department of Computer Science, University of Waterloo.

References

[1] E.R. Berlekamp, *Algebraic Coding Theory.* New York: McGraw-Hill, 1968.

[2] D. Bhandarkar and D.W. Clark, "Performance from architecture: comparing a RISC and a CISC with similar hardware organization," in *Proc. Fourth Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS IV)*, Santa Clara, CA, April 1991, pp. 310-319.

[3] G. Birkhoff and S. MacLane. *A Survey of Modern Algebra,* 3rd ed. New York: Macmillan, 1965.

[4] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt, *Maple V Language Reference Manual.* New York: Springer-Verlag, 1991.

[5] D.W. Clark, P.J. Bannon, and J.B. Keller, "Measuring VAX 8800 performance with a histogram hardware monitor," in *Proc. 15th Ann. Int. Symp. on Comp. Arch.*, Honolulu, May 1988, pp. 176-185.

[6] W. Diffie and M. Hellman. "New directions in cryptography," *IEEE Trans. on Inform. Theory,* vol. 22, pp. 472-479, 1976.

[7] C. Eldert, H.J. Gray, Jr., H.M. Gurk, and M. Rubinoff, "Shifting counters," *AIEE Trans. (Commun. Electron.)*, vol. 77, pp. 70-74, Mar. 1958.

[8] J.S. Emer and D.W. Clark, "A characterization of processor performance in the VAX-11/780," in *Proc. 11th Ann. Int. Symp. on Comp. Arch.*, Ann Arbor, MI, June 1984, pp. 301-310.

[9] G.B. Fitzpatrick, "Synthesis of Binary Ring Counters of Given Periods," *JACM*, vol. 7, pp. 287-297, July 1960.

[10] S.W. Golumb, *Sequences with Randomness Properties.* Baltimore, MD: Glenn L. Martin Co., Final report on contract no. W36-039SC-54-36611, 1955.

[11] S.W. Golomb, ed., *Digital Communications with Space Applications.* Englewood Cliffs, NJ: Prentice Hall, 1964.

[12] S.W. Golomb, *Shift Register Sequences.* San Francisco: Holden-Day, 1967.

[13] D.E. Knuth, *The Art of Computer Programming, vol. 1: Fundamental Algorithms,* 2nd ed. Reading, MA: Addison-Wesley, 1973, p. 34.

[14] D.E. Knuth, *The Art of Computer Programming, vol. 2: Seminumerical Algorithms,* 2nd ed. Reading, MA: Addison-Wesley, 1981.

[15] J.L. Massey, "Logarithms in finite cyclic groups–cryptographic issues," in *Proc. 4th Ann. Benelux Symp. on Inf. Theory*, Leuven, Belgium, May 1983, pp. 17-25.

[16] E.J. McCluskey, *Logic Design Principles with Emphasis on Testable Semicustom Circuits.* Englewood Cliffs, NJ: Prentice-Hall, 1986.

[17] K.S. McCurley, "The discrete logarithm problem," in *Cryptology and Computational Number Theory*, (C. Pomerance, ed.), Proc. of Symposia on Applied Math. Vol. 42, American Math. Soc., 1990, pp. 49-74.

[18] A.M. Odlyzko, "Discrete logarithms in finite fields and their cryptographic significance," in *Advances in Cryptology: Proc. of Eurocrypt 84,* Lecture Notes in Computer Science, vol. 209, New York: Springer-Verlag, 1985, pp. 224-314.

[19] A. Osborne and G. Kane, *Osborne 4 & 8-bit Microprocessor Handbook.* Berkeley, CA: Osborne/McGraw-Hill, 1981, pp. 1-4 to 1-5.

[20] B. Parhami, "Low-cost residue number systems for computer arithmetic," in *Proc. AFIPS NCC*, 1976, pp. 951-956.

[21] W.W. Peterson, "Encoding and error-correction procedures for the Bose-Chaudhuri codes," *IRE Trans. on Inf. Theory*, vol. IT-6, pp. 459-470, Sept. 1960.

[22] W.W. Peterson and E.J. Weldon, Jr., *Error-Correcting Codes*, 2nd ed. Cambridge, MA: MIT Press, 1972.

[23] S.C. Pohlig and M. Hellman, "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance," *IEEE Trans. on Inf. Theory*, vol. IT-24, pp. 106-110, 1978.

[24] R.G. Swan, "Factorization of polynomials over finite fields," *Pacific J. Math.*, vol. 12, pp. 1099-1106, 1962.

[25] L.-T. Wang and E.J. McCluskey, "Hybrid designs generating maximum-length sequences," *IEEE Trans. on Comp.-Aided Design*, vol. 7, no. 1, pp.91-99, Jan. 1988.

[26] N. Zierler and J. Brillhart, "On primitive trinomials (mod 2)," *Inf. and Control,* vol. 13, pp. 541-554, 1968, and vol. 14, pp. 566-569, 1969.