A Tutorial on CRC Computations

Data can lose integrity in storage and transmission. Here are a few software algorithms to protect your data.

Tenkasi V. Ramabadran and Sunil S. Gaitonde
Iowa State University

Cyclic redundancy codes, or CRCs, preserve the integrity of data in storage and transmission applications. CRC implementations can use either hardware or software methods. In the traditional hardware implementation, a simple shift register circuit performs the computations by handling the data a bit at a time. In software implementations, however, handling the data as bytes or even words becomes more convenient and faster. Several software algorithms for performing CRC computations have been reported in the literature.[1-3] Here, after briefly reviewing the theory behind CRC, we describe these algorithms from a mathematical viewpoint. We then compare the different algorithms in terms of their speeds and storage requirements.

Mathematical background

We shall first introduce the binary field and the binary polynomials that facilitate the definition of cyclic redundancy codes. In simple terms, a *field* is an algebraic system in which the operations of addition, subtraction, multiplication, and division can be performed. The set of real numbers, for example, forms a field. Fields can be *finite* or *infinite.* The smallest finite field is the binary field that has just two elements denoted usually by 0 and 1. The tables in Figure 1 define the addition and multiplication operations in this field.

From the addition table, we see that an EXOR gate is all that we need to perform the addition operation in the binary field. Moreover, we see 0 and 1 to be their own additive inverses, and so subtraction in the binary field is the same as addition. Multiplication in the binary field can be performed simply by means of an AND gate. We must define division in this field only for the single nonzero element 1, and we do this trivially by noting that division by 1 leaves both 0 and 1 unchanged.

A *binary polynomial* is a polynomial with coefficients from the binary field. For example, $0, 1, x, 1 + x, x^2, 1 + x + x^2$ are all binary polynomials in the dummy variable x. Given any sequence of bits, we can associate a binary polynomial with it by regarding the different bits as representing the coefficients of the polynomial. For instance, with the sequence 1 0 1 0 1 1, we can associate the fifth degree polynomial $1 \cdot x^0 + 0 \cdot x^1 + 1 \cdot x^2 + 0 \cdot x^3 + 1 \cdot x^4 + 1 \cdot x^5 = 1 + x^2 + x^4 + x^5$.

0272-1732/88/0800-0062$01.00 © 1988 IEEE

According to the convention used here, the rightmost bit of a sequence represents the coefficient of the highest degree term of the associated polynomial. A left-to-right shift of a sequence of bits by i positions (with the vacated positions filled with 0's), therefore, corresponds to multiplying the associated polynomial by x^i. We perform operations involving binary polynomials in exactly the same manner as we do with ordinary polynomials, that is, polynomials with real number coefficients. However, we manipulate the coefficients using the rules of the binary field. Figure 2 contains some examples.

Cyclic redundancy codes

Error control coding provides the means to protect data from errors and involves essentially adding a certain amount of redundancy to the data in a controlled fashion. In a typical (block) coding scheme, the data to be protected is first divided into k-bit *message* blocks. Each block is then encoded into an n-bit ($n > k$) *codeword*. The redundancy added therefore amounts to $n - k$ bits per message block. We refer to these bits as *check* (or *parity*) bits.

We collectively refer to the set of codewords as a *code* and select it such that it has good error correction/detection capabilities and also some algebraic structure to facilitate implementation. A simple example of a code is the even-parity code used in many microcomputer systems to protect data in memory. For this code, $k = 8$, $n = 9$, and $n - k = 1$. All codewords have even parity, and the code is capable of detecting all odd numbers (1, 3, 5, 7, and 9) of errors in a codeword.

+	0	1
0	0	1
1	1	0

(a)

·	0	1
0	0	0
1	0	1

(b)

Figure 1. Addition table (a) and multiplication table (b).

The selection of a code for a specific application depends on a number of factors including the amount of protection required, the overhead involved, the cost of implementation, the error control strategy employed, and the nature of errors. The $n - k$ check bits of a code forming the overhead directly affect the error control capability of the code and thereby the amount of protection provided. In general, the more check bits in a code, the greater is its power of error correction/detection. For a given number of check bits, the relative overhead of a code can be kept low by using a large message block size k; however, this action tends to increase the implementation cost of the code.

Two error control strategies have been popular in practice. They are the FEC, or forward error correction, strategy, which uses error correction alone, and the ARQ, or automatic repeat request, strategy, which uses error detection combined with retransmission of corrupted data. The ARQ strategy is generally preferred for several reasons, including the fact that the number of check bits required to provide a certain amount of error protection is smaller for error detection than

Addition
(or subtraction)

$$\begin{array}{lllllll} 1 & +x & & +x^3 & & & +x^6 \\ & x & +x^2 & & +x^4 & & +x^6 \\ \hline 1 & & +x^2 & +x^3 & +x^4 & & \end{array}$$

Multiplication

$$(1 + x^2 + x^4) \cdot (x + x^3) = x + x^7$$

Division

$$\begin{array}{r|l} & x^2 + x + 1 \quad \text{—Quotient} \\ \hline x^4 + x^2 + 1 & x^6 + x^5 + x + 1 \\ & x^6 + x^4 + x^2 \\ \hline & x^5 + x^4 + x^2 + x + 1 \\ & x^5 + x^3 + x \\ \hline & x^4 + x^3 + x^2 + 1 \\ & x^4 + x^2 + 1 \\ \hline & x^3 \quad \text{—Remainder} \end{array}$$

Figure 2. Examples of operations involving binary polynomials.

for error correction. The FEC strategy is typically used when retransmission is impossible or impractical. Depending on the nature of the error-generating mechanism in the storage or transmission medium, errors can be *random* (isolated) or *bursty* (clustered) and correspondingly influence the choice of a code.

Cyclic redundancy codes (also known as *polynomial* codes) form a powerful class of codes suited especially for the detection of burst errors. In such a code, we select the codewords such that the associated polynomials are multiples of a certain polynomial $g(x)$ called the *generator polynomial*. The generator polynomial therefore decides the error control properties of a CRC. Later, we discuss how we can choose the generator polynomial of a CRC in such a way that we induce certain desirable properties in the code. Some commonly used generator polynomials are listed in Table 1.

We now present a simple encoding procedure for a CRC given its generator polynomial $g(x)$ of degree $n - k$. Let $u(x)$ and $v(x)$ represent the polynomials associated with a k-bit message and the corresponding n-bit codeword respectively. We refer to $u(x)$ and $v(x)$, with respective (maximum) degrees of $k - 1$ and $n - 1$, as a *message polynomial* and a *codeword polynomial* respectively. The relationship between $u(x)$ and $v(x)$ is then expressed as

$$v(x) = u(x)\ g(x).$$

Notice that the number of added redundant bits equals the degree of the generator polynomial $g(x)$. The above encoding procedure, which involves the multiplication of two binary polynomials, can easily be implemented in hardware by means of shift registers and EXOR gates.

We describe another encoding procedure for a CRC by the equation

$$v(x) = s(x)\ +\ x^{n-k}\, u(x)$$

where $s(x) = R_{g(x)}\,[x^{n-k}\, u(x)]$ is the remainder resulting from the division of $x^{n-k}u(x)$ by $g(x)$. That is, for some unique polynomials $a(x)$ and $s(x)$, we can write

$$x^{n-k}u(x) = a(x)g(x) + s(x)$$

so that $v(x) = a(x)g(x)$ is a multiple of $g(x)$.

In this encoding procedure the rightmost k bits of a codeword, which correspond to the coefficients of $x^{n-k}\ u(x)$, are simply the message bits being encoded. The leftmost $n - k$ bits, which correspond to the coefficients of $s(x)$ (which has a maximum degree of $n - k - 1$), form the check bits. Later, we discuss the implementation of this encoding procedure using hardware and software approaches.

Both of the encoding procedures we described generate the same set of codewords [multiples of $g(x)$], but the correspondence between a message and a codeword is altered depending on the procedure used. The second procedure, which maps a message into a codeword containing the message itself as a constituent part, is invariably used because the encoding and decoding algorithms are essentially identical when using this procedure.

Let us look at an example of this procedure. Let the message to be encoded be the sequence 1 0 1 0 1 1 represented by the polynomial $u(x) = 1 + x^2 + x^4 + x^5$. Let $g(x)$ be the CRC-16 polynomial $1\ +\ x^2\ +\ x^{15}\ +\ x^{16}$. Then

$$x^{n-k}u(x) = x^{16}(x^5 + x^4 + x^2 + 1)$$

$$= x^{21} + x^{20} + x^{18} + x^{16}.$$

Dividing $x^{n-k}u(x)$ by $g(x)$, we obtain the remainder $s(x)$ seen in Figure 3.

The check bits are therefore 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0, and the message 1 0 1 0 1 1 is encoded into the codeword 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 1 0 1 0 1 1.

We accomplish error detection using a CRC as follows. Let a message polynomial $u(x)$ be encoded into

Table 1.
Commonly used generator polynomials.

CRC-16	$x^{16} + x^{15} + x^2 + 1$
SDLC (IBM, CCITT)	$x^{16} + x^{12} + x^5 + 1$
CRC-16 REVERSE	$x^{16} + x^{14} + x + 1$
SDLC REVERSE	$x^{16} + x^{11} + x^4 + 1$
LRCC-16	$x^{16} + 1$
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + x + 1$
LRCC-8	$x^8 + 1$
ETHERNET	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11}$ $+ x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

$$
\begin{array}{r|l}
 & x^5 + x^2 + x \\
\hline
x^{16} + x^{15} + x^2 + 1 & x^{21} + x^{20} + x^{18} + x^{16} \\
 & x^{21} + x^{20} + x^7 + x^5 \\
\hline
 & x^{18} + x^{16} + x^7 + x^5 \\
 & x^{18} + x^{17} + x^4 + x^2 \\
\hline
 & x^{17} + x^{16} + x^7 + x^5 + x^4 + x^2 \\
 & x^{17} + x^{16} + x^3 + x \\
\hline
 & x^7 + x^5 + x^4 + x^3 + x^2 + x \\
\hline
\end{array}
$$

$$s(x) = x + x^2 + x^3 + x^4 + x^5 + x^7$$

Figure 3. An example of message encoding.

the codeword polynomial $v(x)$ before transmission or storage. Let the errors affecting the codeword be represented by the *error polynomial* $e(x)$. The maximum degree of $e(x)$ is $n - 1$, and its nonzero terms indicate the positions where errors have occurred.

The polynomial $r(x) = v(x) + e(x)$ represents the *corrupted* codeword received at the decoder and has a maximum degree also of $n - 1$. The decoder checks whether $r(x)$ represents a codeword, that is, whether it is a multiple of $g(x)$. We do this simply by dividing $r(x)$ (or equivalently $[x^{n-k}r(x)]$ by $g(x)$ and checking for a zero remainder. If the remainder is zero, we assume $r(x)$ to be error free. That is, $e(x) = 0$ so that $r(x) = v(x)$, and the original message $u(x)$ may be recovered from it.

If the remainder is not zero, we detect the presence of one or more errors. However, the possibility exists that the remainder is zero; that is, $r(x)$ is a multiple of $g(x)$, even though errors have occurred. When can this happen? Since both $v(x)$ and $r(x)$ are multiples of $g(x)$, such a situation arises when $e(x)$ itself is a nonzero multiple of $g(x)$; that is, when $e(x)$ represents a nonzero codeword. The probability of such *undetected* errors depends on the nature and distribution of errors in the storage or transmission medium and the generator polynomial $g(x)$ of the CRC.

We can control the error detection capability of a CRC by suitable choice of its generator polynomial $g(x)$.[4] Consider a *single* error pattern represented by $e(x) = x^i$ for some i, $0 \le i \le n-1$. If $g(x)$ has more than one nonzero term, it does not divide x^i evenly and therefore can detect all single errors.

Suppose $g(x)$ has $(1 + x)$ as one of its factors. Then the codeword polynomials, which are multiples of $g(x)$, also have $(1 + x)$ as a factor. Any polynomial with $(1 + x)$ as a factor has an even number of terms. We show this by considering some polynomial $v(x) = (1 + x)w(x)$. Upon substituting $x = 1$, we have $v(1) = (1 + 1)w(1) = 0$, implying that $v(x)$ has an even number of terms. Therefore, if $g(x)$ has $(1 + x)$ as a factor, all the codewords have even parity and any *odd* number of errors can be detected.

Consider now a *double* error pattern $e(x) = x^i + x^j = x^i(1 + x^{j-i})$ for some i, $0 \le i \le n-2$ and j, $i+1 \le j \le n-1$. If $g(x)$ does not have x as a factor and if it does not evenly divide $[1 + x^{j-i}]$ for $1 \le j - i \le n - 1$, we can detect all double errors.

Let us now consider the detection of burst errors. A *burst* error of length b is any error pattern for which the number of bits between the first and last errors, including these errors, is b. For example, the error pattern 0 0 0 1 0 1 0 1 1 0 0, ..., 0 represented by $e(x) = x^3 + x^5 + x^7 + x^8$ is a burst error of length 6. Let the generator polynomial of a CRC be of the form $g(x) = 1 + g_1 x + \cdots + g_{n-k-1}x^{n-k-1} + x^{n-k}$ where $g_1, g_2, \ldots, g_{n-k-1}$ can be either 0 or 1. In other words, $g(x)$ has a degree of $n - k$ and is not divisible by x. Any burst error of length $(n - k)$ or less can be represented as $e(x) = x^i(1 + e_1 x + \cdots + e_{n-k-1}x^{n-k-1})$ for some i, $0 \le i \le k$ and where $e_1, e_2, \ldots, e_{n-k-1}$ can be either 0 or 1. Clearly, such a polynomial is not evenly divisible by $g(x)$, and therefore we can detect the corresponding burst error.

Consider now a burst error of length $n - k + 1$ represented by $e(x) = x^i(1 + e_1 x + \cdots + e_{n-k-1}x^{n-k-1} + x^{n-k})$. Of the 2^{n-k-1} possible error patterns of this form for each i, $0 \le i \le k - 1$, only one error pattern, namely, $e(x) = x^i g(x)$, is undetectable. The fraction of undetected burst errors of length $n - k + 1$ is therefore $2^{-(n-k-1)}$. Similar consideration shows that the fraction of undetected burst errors of length greater than $n - k + 1$ is $2^{-(n-k)}$. Notice the fundamental role played by the number of check bits $n - k$ in the detection of burst errors.

We now quantify the error detection capability of a specific code, the one generated by the CRC-16 polynomial. For this code, $g(x) = 1 + x^2 + x^{15} + x^{16} = (1 + x)(1 + x + x^{15})$ and the smallest integer m for which $g(x)$ divides $[1 + x^m]$ is 32,767. So, if the codeword length $n \le 32{,}767$ (or equivalently, if the message block size $k \le 32{,}751$), the CRC-16 polynomial can detect all single, double, triple, and odd numbers of errors. Furthermore, it can detect all burst errors of length 16 or less, 99.997 percent of 17-bit error bursts, and 99.998 percent of 18-bit or longer error bursts.

Among the different generator polynomials shown earlier in Table 1, the CRC-16 polynomial is commonly used worldwide; for instance, the Bisync (binary synchronous) protocol uses it. IBM's synchronous data link control protocol uses the SDLC polynomial, and CCITT (the International Consultative Committee for Telegrapy and Telephony) has standardized it. The "reverse" polynomials are the same as the "forward" polynomials except that the codewords are in reverse order. The CRC-12 polynomial is used with 6-bit characters. The LRC (Longitudinal Redundancy Code) polynomials are used for simple longitudinal parity calculations, that is, a mod-2 sum of bytes (LRCC-8) or words (LRCC-16). The Ethernet polynomial is used in local area networks.[5]

Hardware implementation

The second CRC encoding procedure described earlier involves the computation of $s(x)$, which is the remainder resulting from the division of $x^{n-k}u(x)$ by $g(x)$. The decoding operation also involves the division of $r(x)$ [or $x^{n-k}r(x)$] by $g(x)$ for computing the remainder. A hardware circuit that simulates the division operation is a linear feedback shift register.[4] Figure 4 shows the LFSR circuit for the CRC-16 polynomial. Notice that the message polynomial $u(x)$ [$r(x)$ for decoding] feeds in at the right end of the shift register that corresponds to multiplication of $u(x)$ [or $r(x)$] by x^{n-k}.

We perform CRC encoding using a LFSR circuit as follows. We feed in the entire message $u(x)$ (with the rightmost bit first) at the right end of the LFSR circuit. The circuit performs the division operation and stores the check bits corresponding to $s(x)$ in the different shift register stages as indicated. Then we append these bits to the message to form the codeword. The decoding operation is quite similar. We feed the received polynomial $r(x)$ into the LFSR circuit to compute the remainder. If the remainder is zero, we simply drop the check bits from the codeword to recover the message. If the remainder is not zero, we have detected the presence of error(s) and must take suitable action to recover from the error condition.

Software implementations

In the hardware implementation we just described, we process data bit by bit. It is possible to duplicate the operation of the hardware circuit in software, giving rise to what we refer to as the bitwise, or CRCB, algorithm. However, we can achieve faster implementations in software by handling the data as bytes or even words. We derive the necessary relationship for such implementations by considering the effect on the check bits of a message when the message is augmented by a byte. We assume a generator polynomial $g(x)$ of degree 16 (for example, CRC-16) in the following derivation.

Let $u(x)$ and $s(x)$ represent a message and the corresponding check bits respectively. These binary polynomials are of the form,

$$u(x) = u_0 + u_1x + u_2x^2 + \cdots, \text{ and}$$

$$s(x) = s_0 + s_1x + s_2x^2 + \cdots + s_{15}x^{15}.$$

Since $s(x)$ is the remainder resulting from the division of $x^{16}u(x)$ by $g(x)$, we can write for some polynomial $a(x)$,

$$x^{16}u(x) = a(x)g(x) + s(x).$$

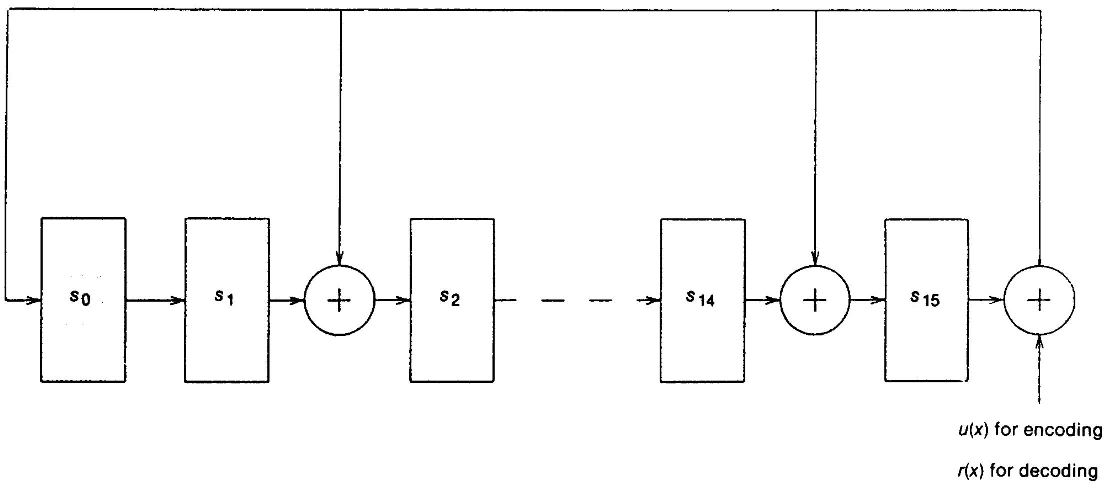

Figure 4. LFSR circuit for the CRC-16 polynomial $1 + x^2 + x^{15} + x^{16}$.

Let us now augment the message by one more byte. This corresponds to shifting the original message eight positions to the right and inserting the extra byte in the vacated positions. The augmented message $u'(x)$ can therefore be written as

$$u'(x) = b(x) + x^8 u(x) \text{ where,}$$
$$b(x) = b_0 + b_1 x + \cdots + b_7 x^7$$

corresponds to the newly added byte. Let $s'(x)$ represent the check bits corresponding to $u'(x)$. Then, $s'(x)$ is the remainder resulting from the division of $x^{16}u'(x)$ by $g(x)$. We will denote this symbolically as

$$s'(x) = R_{g(x)}[x^{16}u'(x)].$$

Expressing $u'(x)$ in terms of $u(x)$, we have

$$\begin{aligned} x^{16}u'(x) &= x^{16}[b(x) + x^8 u(x)] \\ &= x^{16}b(x) + x^8[x^{16}u(x)] \\ &= x^{16}b(x) + x^8[a(x)g(x) + s(x)] \\ &= x^{16}b(x) + x^8 a(x)g(x) + x^8 s(x) \end{aligned}$$

The remainder of a sum of polynomials under division by another polynomial is simply the sum of the remainders of the individual polynomials. So,

$$\begin{aligned} s'(x) &= R_{g(x)}[x^{16}u'(x)] \\ &= R_{g(x)}[x^{16}b(x) + x^8 a(x)g(x) + x^8 s(x)] \\ &= R_{g(x)}[x^{16}b(x)] + R_{g(x)}[x^8 a(x)g(x)] \\ &\quad + R_{g(x)}[x^8 s(x)] \\ &= R_{g(x)}[x^{16}b(x) + x^8 s(x)] \end{aligned}$$

since $g(x)$ evenly divides $x^8 a(x)g(x)$. Expressing $b(x)$ and $s(x)$ in expanded form,

$$\begin{aligned} x^{16}b(x) + x^8 s(x) &= b_0 x^{16} + b_1 x^{17} + \cdots + b_7 x^{23} \\ &+ s_0 x^8 + s_1 x^9 + \cdots + s_7 x^{15} \\ &+ s_8 x^{16} + s_9 x^{17} + \cdots + s_{15} x^{23} \\ &= (b_0 + s_8)x^{16} + (b_1 + s_9)x^{17} + \cdots + (b_7 + s_{15})x^{23} \\ &+ s_0 x^8 + s_1 x^9 + \cdots + s_7 x^{15}. \end{aligned}$$

Replacing $[b_i + s_{i+8}]$ by t_i for $i = 0, 1, \ldots, 7$,

$$\begin{aligned} x^{16}b(x) + x^8 s(x) &= t_0 x^{16} + t_1 x^{17} + \cdots + t_7 x^{23} \\ &+ s_0 x^8 + s_1 x^9 + \cdots + s_7 x^{15} \end{aligned}$$

and

$$\begin{aligned} s'(x) &= R_{g(x)}[t_0 x^{16} + t_1 x^{17} + \cdots + t_7 x^{23}] \\ &+ R_{g(x)}[s_0 x^8 + s_1 x^9 + \cdots + s_7 x^{15}] \\ &= R_{g(x)}[t_0 x^{16} + t_1 x^{17} + \cdots + t_7 x^{23}] \\ &+ (s_0 x^8 + s_1 x^9 + \cdots + s_7 x^{15}) \end{aligned}$$

since the degree of the second expression is smaller than that of $g(x)$.

The last equation relates the check bits of the augmented message with the check bits of the original message and the added byte. Notice that the expression $(s_0 x^8 + s_1 x^9 + \cdots + s_7 x^{15})$ represents the high-order eight check bits of $u(x)$ shifted to the right by eight positions. The bits $(t_0, t_1, \ldots, t_7)$ are simply the sum of the added byte $(b_0, b_1, \ldots, b_7)$ and the low-order eight check bits $(s_8, s_9, \ldots, s_{15})$ of $u(x)$.

Different algorithms for CRC computations may be viewed as methods to compute $s'(x)$ from $s(x)$ and $b(x)$ using the last equation. In the following material, we present a few bytewise algorithms that can be used directly with generator polynomials of degree 16. Modifications to these algorithms to handle generator polynomials of other degrees and to handle other units of data, for example, words, are fairly straightforward.

Table lookup algorithm

For different values of the byte $(t_0, t_1, \ldots, t_7)$, we can precompute the values of the remainder $R_{g(x)}[t_0 x^{16} + t_1 x^{17} + \cdots + t_7 x^{23}]$ and store them in a table. Such a table would have 256 entries, each two bytes long. For example, Table 2 stores the lookup values corresponding to the CRC-16 polynomial. We describe an algorithm for CRC computations using such a table as follows. Assume that the check bits are stored in a register referred to as the CRC register.

1) Initialize the CRC register to 0000 hex, that is, set the values of s_0 through s_{15} to 0.
2) EXOR the input byte $(b_0, b_1, \ldots, b_7)$ with $(s_8, s_9, \ldots, s_{15})$ to form $(t_0, t, \ldots, t_7)$.
3) Shift the CRC register eight positions to the right.
4) Look up the value corresponding to $(t_0, t_1, \cdots, t_7)$ from the table and EXOR the CRC register with it.
5) Repeat steps 2 to 4 until you reach the end of the message.

Compared with the bitwise algorithm, the table lookup, or CRCT, algorithm we described has a considerable speed advantage.

Reduced table lookup algorithm

In some applications the amount of storage required for the table lookup algorithm may be too high. We can use a reduced (smaller) table in such situations. The basic idea here is to split up the expression $R_{g(x)}[t_0 x^{16} + t_1 x^{17} + \cdots + t_7 x^{23}]$ into the sum $R_{g(x)}[t_0 x^{16}] + R_{g(x)}[t_1 x^{17}] + \cdots + R_{g(x)}[t_7 x^{23}]$. For $i = 0, 1, \ldots, 7$, t_i can be either 0 or 1. If $t_i = 0$, $R_{g(x)}[t_i x^{i+16}] = 0$. So, we must precompute only eight 16-bit values corresponding to $R_{g(x)}[x^{16}]$, $R_{g(x)}[x^{17}]$, $\cdots$, and $R_{g(x)}[x^{23}]$. These values can be stored in a smaller table just 16 bytes long as against the 512 bytes required for the previous algorithm. Table 3 shows such a reduced table with precomputed values for CRC-16. We describe the bytewise reduced table lookup, or CRCR(B), algorithm as follows:

Table 2.
Lookup table for CRC-16.

0000	D801	F001	2800	A001	7800	5000	8801
C0C1	18C0	30C0	E8C1	60C0	B8C1	90C1	48C0
C181	1980	3180	E981	6180	B981	9181	4980
0140	D941	F141	2940	A141	7940	5140	8941
C301	1B00	3300	EB01	6300	BB01	9301	4B00
03C0	DBC1	F3C1	2BC0	A3C1	7BC0	53C0	8BC1
0280	DA81	F281	2A80	A281	7A80	5280	8A81
C241	1A40	3240	EA41	6240	BA41	9241	4A40
C601	1E00	3600	EE01	6600	BE01	9601	4E00
06C0	DEC1	F6C1	2EC0	A6C1	7EC0	56C0	8EC1
0780	DF81	F781	2F80	A781	7F80	5780	8F81
C741	1F40	3740	EF41	6740	BF41	9741	4F40
0500	DD01	F501	2D00	A501	7D00	5500	8D01
C5C1	1DC0	35C0	EDC1	65C0	BDC1	95C1	4DC0
C481	1C80	3480	EC81	6480	BC81	9481	4C80
0440	DC41	F441	2C40	A441	7C40	5440	8C41
CC01	1400	3C00	E401	6C00	B401	9C01	4400
0CC0	D4C1	FCC1	24C0	ACC1	74C0	5CC0	84C1
0D80	D581	FD81	2580	AD81	7580	5D80	8581
CD41	1540	3D40	E541	6D40	B541	9D41	4540
0F00	D701	FF01	2700	AF01	7700	5F00	8701
CFC1	17C0	3FC0	E7C1	6FC0	B7C1	9FC1	47C0
CE81	1680	3E80	E681	6E80	B681	9E81	4680
0E40	D641	FE41	2640	AE41	7640	5E40	8641
0A00	D201	FA01	2200	AA01	7200	5A00	8201
CAC1	12C0	3AC0	E2C1	6AC0	B2C1	9AC1	42C0
CB81	1380	3B80	E381	6B80	B381	9B81	4380
0B40	D341	FB41	2340	AB41	7340	5B40	8341
C901	1100	3900	E101	6900	B101	9901	4100
09C0	D1C1	F9C1	21C0	A9C1	71C0	59C0	81C1
0880	D081	F881	2080	A881	7080	5880	8081
C841	1040	3840	E041	6840	B041	9841	4040

Note: The lookup values in hex notation correspond to $(t_0, t_1, \ldots, t_7)$, ranging from 0 to 255 and increasing first by column and then by row.

1) Initialize the CRC register to 0000 hex.

2) EXOR the input byte $(b_0, b_1, \ldots, b_7)$ with $(s_8, s_9, \ldots, s_{15})$ to form $(t_0, t_1, \ldots, t_7)$.

3) Shift the CRC register eight positions to the right.

4) For $i = 0, 1, \ldots, 7$ if $t_i = 1$, look up the corresponding 16-bit value $(R_{g(x)}[x^{i+16}])$ from the table and EXOR the CRC register with it.

5) Repeat steps 2 to 4 until you reach the end of the message.

Notice that only step 4 differs from the previous algorithm. Instead of a single EXOR operation, the number of EXOR operations corresponds to the number of nonzero bits in $(t_0, t_1, \ldots, t_7)$. An average value of 4 seems reasonable for random data. This increased number of operations naturally detracts from the speed performance of the algorithm.

It is possible to generate a regular lookup table from a reduced lookup table in a simple manner. For example, Table 2 can be generated from Table 3 as follows. For any given byte $(t_0, t_1, \ldots, t_7)$, generate the lookup value in Table 2 by EXORing the entries of Table 3 corresponding to the nonzero bits of the byte.

Table 3.
Reduced lookup table for CRC-16.

Power of x (i)	$R_{g(x)}$ [x^i] (in hex)
16	A001
17	F001
18	D801
19	CC01
20	C601
21	C301
22	C181
23	C0C1

On-the-fly algorithm

Another approach that keeps the storage requirement low is the on-the-fly, or CRCF, algorithm.[3] In this algorithm, we compute each modified check bit (s_i') as a function of the original check bits (s_i's) and the sum of the original check bits and the input bits (t_i's). We can derive the required functional relationships by using the reduced lookup table, as shown in Table 4 for the CRC-16 polynomial. The first row in Table 4 corresponds to the original check bits, that is, $s(x)$ shifted eight positions to the right. The next eight rows correspond to the entries of Table 3 expressed in binary with the nonzero bits replaced by the appropriate symbolic notation $t_i, i = 0, 1, \ldots, 7$. Table 5 is a re-

Table 4.
Relationships leading to the on-the-fly algorithm.[3]

s_0'	s_1'	s_2'	s_3'	s_4'	s_5'	s_6'	s_7'	s_8'	s_9'	s_{10}'	s_{11}'	s_{12}'	s_{13}'	s_{14}'	s_{15}'
0	0	0	0	0	0	0	0	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
t_0	0	t_0	0	0	0	0	0	0	0	0	0	0	0	0	t_0
t_1	t_1	t_1	t_1	0	0	0	0	0	0	0	0	0	0	0	t_1
t_2	t_2	0	t_2	t_2	0	0	0	0	0	0	0	0	0	0	t_2
t_3	t_3	0	0	t_3	t_3	0	0	0	0	0	0	0	0	0	t_3
t_4	t_4	0	0	0	t_4	t_4	0	0	0	0	0	0	0	0	t_4
t_5	t_5	0	0	0	0	t_5	t_5	0	0	0	0	0	0	0	t_5
t_6	t_6	0	0	0	0	0	t_6	t_6	0	0	0	0	0	0	t_6
t_7	t_7	0	0	0	0	0	0	t_7	t_7	0	0	0	0	0	t_7

Table 5.
Rearranged form of Table 4.

s_0'	s_1'	s_2'	s_3'	s_4'	s_5'	s_6'	s_7'	s_8'	s_9'	s_{10}'	s_{11}'	s_{12}'	s_{13}'	s_{14}'	s_{15}'
0	0	0	0	0	0	0	0	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
t_0	t_1	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7						t_0
t_1	t_2	t_1	t_2	t_3	t_4	t_5	t_6	t_7							t_1
t_2	t_3														t_2
t_3	t_4														t_3
t_4	t_5														t_4
t_5	t_6														t_5
t_6	t_7														t_6
t_7															t_7

arranged form of Table 4 and is identical to Perez's Table 5[3] except for notational differences. We describe the on-the-fly algorithm here:

1) Initialize the CRC register to 0000 hex.

2) EXOR the input byte ($b_0, b_1, \ldots, b_7$) with ($s_8, s_9, \ldots, s_{15}$) to form ($t_0, t_1, \ldots, t_7$).

3) Shift the CRC register eight positions to the right.

4) Calculate the contribution to each s_i', $i = 0, 1, \ldots, 15$ from the t_i's using the relationships in Table 5.

Table 6.
Reduced lookup tables for some important CRC polynomials including Ethernet.

CRC-16		SDLC		CRC-16 Reverse		SDLC Reverse	
i	$R_{g(x)}[x^i]$	i	$R_{g(x)}[x^i]$	i	$R_{g(x)}[x^i]$	i	$R_{g(x)}[x^i]$
16	A001	16	8408	16	C002	16	8810
17	F001	17	4204	17	6001	17	4408
18	D801	18	2102	18	F002	18	2204
19	CC01	19	1081	19	7801	19	1102
20	C601	20	8C48	20	FC02	20	0881
21	C301	21	4624	21	7E01	21	8C50
22	C181	22	2312	22	FF02	22	4628
23	C0C1	23	1189	23	7F81	23	2314
24	C061	24	8CCC	24	FFC2	24	118A
25	C031	25	4666	25	7FE1	25	08C5
26	C019	26	2333	26	FFF2	26	8C72
27	C00D	27	9591	27	7FF9	27	4639
28	C007	28	CEC0	28	FFFE	28	AB0C
29	C002	29	6760	29	7FFF	29	5586
30	6001	30	33B0	30	FFFD	30	2AC3
31	9001	31	19D8	31	BFFC	31	9D71

Ethernet							
i	$R_{g(x)}[x^i]$	i	$R_{g(x)}[x^i]$	i	$R_{g(x)}[x^i]$	i	$R_{g(x)}[x^i]$
32	EDB88320	40	3B83984B	48	E1351B80	56	ED59B63B
33	76DC4190	41	F0794F05	49	709A8DC0	57	9B14583D
34	3B6E20C8	42	958424A2	50	384D46E0	58	A032AF3E
35	1DB71064	43	4AC21251	51	1C26A370	59	5019579F
36	0EDB8832	44	C8D98A08	52	0E1351B8	60	C5B428EF
37	076DC419	45	646CC504	53	0709A8DC	61	8F629757
38	EE0E612C	46	32366282	54	0384D46E	62	AA09C88B
39	77073096	47	191B3141	55	01C26A37	63	B8BC6765

EXOR the CRC register with the calculated 16-bit value.

5) Repeat steps 2 to 4 until you reach the end of the message.

Wordwise algorithms

The software algorithms we have discussed so far, except for the bitwise algorithm, handle data a byte at a time. These algorithms can easily be modified to handle larger units of data provided the size of such a unit does not exceed the degree of the generator polynomial. In Table 6 we list reduced lookup tables for some important generator polynomials. The program shown in the accompanying box generated these tables.

For each $g(x)$, we provide the values of $R_{g(x)}[x^i]$ for $n - k \leq i < 2(n - k)$. Such tables are useful in generating the necessary information for implementing the different software algorithms with any unit of data the size of which does not exceed the degree of the generator polynomial. For example, consider the CRC-16 polynomial with a word as the unit of data. The corresponding reduced lookup table can be used to generate a regular lookup table with 65,536 entries necessary to implement the table lookup algorithm. It can be used directly in a wordwise reduced table lookup, or CRCR(W), algorithm. It can also be used to generate the necessary functional relationships for implementing the on-the-fly algorithm. The descriptions of the different algorithms using a word as the unit of data undergo minor modifications. We describe the wordwise reduced table lookup algorithm, for example, as follows:

1) Initialize the CRC register to 0000 hex.

2) EXOR the input word $(b_0, b_1, \ldots, b_{15})$ with $(s_0, s_1, \ldots, s_{15})$ to form $(t_0, t_1, \ldots, t_{15})$.

3) Shift the CRC register 16 positions to the right.

4) For $i = 0, 1, \ldots, 15$ if $t_i = 1$, look up the corresponding 16-bit value from the table and EXOR the CRC register with it.

5) Repeat steps 2 to 4 until you reach the end of the message.

We compared the different software algorithms in terms of their speeds and storage requirements. Table 7 lists the results. These algorithms are the bitwise CRCB, the table lookup CRCT, the on-the-fly CRCF, and the bytewise and wordwise reduced table lookup CRCR(B) and CRCR(W). The generator polynomial used in the comparison is the CRC-16 polynomial. The accompanying box lists the five programs implementing the different algorithms. The programs are written in 8086 assembly language as routines to be called from a main C program. We used an Intel single-board computer (iSBC 86/12) operating at 5-MHz clock speed in executing the different programs.

Table generation

LISTING 1

```
/**************************************************************************
*                                                                         *
* This program interactively generates a reduced look-up table for any    *
* given generator polynomial g(x) of degree 16 or 32. The information     *
* about the polynomial is entered in two steps as follows:                *
*                                                                         *
*               1)  Degree of the polynomial (16 or 32).                  *
*               2)  Coefficients of the polynomial except that of         *
*                   the highest degree term in hex notation (4 or         *
*                   8 hex characters).                                    *
*                                                                         *
* For example, consider the CRC-16 polynomial with g(x) = 1 + x**2 +      *
* x**15 + x**16.  When prompted, information about CRC-16 is entered      *
* as follows:                                                             *
*               1)  Enter degree of the polynomial (16 or 32).            *
*                   16 <CR>                                               *
*               2)  Enter coefficients of the polynomial in hex.          *
*                   A001 <CR>                                             *
*                                                                         *
* The value A001 is obtained from the coefficients of the CRC-16          *
* polynomial (except the highest degree term) arranged as:               *
*                                                                         *
*               1.(x**0)+0.(x**1)+1.(x**2)+......+1.(x**15)               *
*                                                                         *
*                                                                         *
* NOTE:  This program assumes 32 bit integers                             *
*                                                                         *
**************************************************************************/

main()

{

 int      degree, coeff, remainder, temp, power;

/*  get information about the generator polynomial  */

 printf("Enter degree of the polynomial (16 or 32).\n");
 scanf("%d",&degree);
 while (degree != 32 && degree != 16)    {
         printf("Only 16 and 32 are allowed.\n");
         printf("Enter degree of the polynomial (16 or 32).\n");
         scanf("%d",&degree);
 }
 printf("Enter coefficients of the polynomial in hex.\n");
 scanf("%X", &coeff);

/*  compute and print the reduced look-up table  */

 printf("Power of x [=i]   R((x**i)/g(x))\n");
 printf("--------------    --------------\n");
 remainder = coeff;
 for (power = degree; power < 2*degree; power++) {
         printf("     %d                 %X\n", power, remainder);
         temp = remainder & 0x00000001;
         remainder = (remainder >> 1);
         if (temp == 1)
                 remainder = remainder ^ coeff;
 }

}                /*  end main  */
```

The programs implementing the five CRC algorithms

```
;  This module contains 8086 assembly language routines implementing
;  different CRC algorithms for the CRC-16 polynomial. These routines
;  are to be invoked from a main C language program (not shown). This
;  module is to be assembled using the Aztec C86 software package.

        largecode
        assume  cs:codeseg, ds:dataseg

dataseg segment para    public  'data'

table1  dw      0000h                   ;the 256 entry look-up table
;                 .                     ;for CRC-16 goes here
;                 .

table2  dw      0A001h                  ;the 16 entry reduced look-up
;                 .                     ;table for CRC-16 goes here
;                 .

dataseg ends

codeseg segment para    public  'code'


;                          LISTING 2

;                      Bit-wise Algorithm
;************************************************************
;*                                                          *
;*  crcb (buff_addr,length)                                 *
;*                                                          *
;*  This routine computes the check bits for any given message *
;*  using the bit-wise algorithm, i.e., software simulation of *
;*  the LFSR circuit hardware implementation.               *
;*                                                          *
;*  Called from "C" program as :  crcb(buff_addr,length);   *
;*                                                          *
;*  Input     :     buff_addr  -  message buffer address    *
;*                  length     -  message length (words)    *
;*                                                          *
;*  Output    :     check bits -  returned in ax            *
;*                                                          *
;************************************************************

        public  crcb_
crcb_   proc    far

        push    bp              ;standard prolog
        mov     bp,sp
        push    es
        push    di
        push    si
        push    cx
        push    dx

        les     di,6[bp]        ;load buffer address in es:di
        mov     si,10[bp]       ;load message length in si

        xor     ax,ax           ;initialize CRC register
crcb0:  mov     dx,es:[di]      ;fetch a message word
        add     di,2            ;bump message pointer
        mov     cx,16           ;load word length (16 bits) in cx
crcb1:  clc                     ;prepare for a shift
        rcr     ax,1            ;shift CRC register to right
        jc      crcb2           ;check lsb of CRC register
        rcr     dx,1            ;check lsb of message word
        jnc     crcb4           ;0-0 - no action
        jmp     crcb3           ;0-1 - modify CRC register
crcb2:  rcr     dx,1            ;check lsb of message word
        jc      crcb4           ;1-1 - no action
crcb3:  xor     ax,0A001h       ;1-0 - modify CRC register
crcb4:  loop    crcb1           ;keep looping till end of word
        dec     si              ;
        jnz     crcb0           ;keep looping till end of message

        pop     dx              ;standard epilog
        pop     cx
        pop     si
        pop     di
        pop     es
        pop     bp
        ret
crcb_   endp


;                          LISTING 3

;                    Table Look-up Algorithm
;************************************************************
;*                                                          *
;*  crct (buff_addr,length)                                 *
;*                                                          *
;*  This routine computes the check bits for any given message *
;*  using the table look-up algorithm.                      *
;*                                                          *
;*  Called from "C" program as:   crct(buff_addr,length);   *
;*                                                          *
;*  Input     :     buff_addr  -  message buffer address    *
;*                  length     -  message length (bytes)    *
;*                                                          *
;*  Output    :     check bits -  returned in ax            *
;*                                                          *
;*  NOTE:  The 256 entry (512 bytes) look-up table starts at *
;*         label "table1" within the data segment.          *
;*                                                          *
;************************************************************

        public  crct_
crct_   proc    far

        push    bp              ;standard prolog
        mov     bp,sp
        push    es
        push    di
        push    bx
        push    cx

        les     di,6[bp]        ;load buffer address in es:di
        mov     cx,10[bp]       ;load message length in cx

        xor     ax,ax           ;initialize CRC register
crct0:  mov     bl,es:[di]      ;fetch a message byte
        inc     di              ;bump message pointer
        xor     bl,al           ;XOR message byte with low CRC byte
        mov     al,ah           ;shift CRC register 8 positions to
        xor     ah,ah           ; the right
        xor     bh,bh           ;prepare bx for indexing
        shl     bx,1            ;multiply bx by 2
        xor     ax,table1[bx]   ;XOR CRC register with table entry
        loop    crct0           ;keep looping till end of message

        pop     cx              ;standard epilog
        pop     bx
        pop     di
        pop     es
        pop     bp
        ret
crct_   endp
```

```
;                        LISTING 4

;                   On-the-fly Algorithm

;*********************************************************
;*                                                       *
;*  crcf (buff_addr,length)                              *
;*                                                       *
;*  This routine computes the check bits for any given message  *
;*  using the on-the-fly algorithm [3].                  *
;*                                                       *
;*  Called from "C" program as :  crcf(buff_addr,length);  *
;*                                                       *
;*  Input      :     buff_addr  -  message buffer address  *
;*                   length     -  message length (bytes)  *
;*                                                       *
;*  Output     :     check bits -  returned in ax        *
;*                                                       *
;*  NOTE:  See [3] for a description of the logic and the  *
;*         symbolic notations used in this routine.      *
;*                                                       *
;*********************************************************

         public  crcf_
crcf_    proc    far

         push    bp                  ;standard prolog
         mov     bp,sp
         push    es
         push    di
         push    si
         push    cx
         push    dx

         les     di,6[bp]            ;load buffer address in es:di
         mov     si,10[bp]           ;load message length in si

         xor     ax,ax               ;initialize CRC register
crcf0:   mov     dl,es:[di]          ;fetch a message byte
         inc     di                  ;bump message pointer
         xor     dl,al               ;xor message byte with low CRC byte
         mov     dh,dl               ; to form x and save it
         add     dh,dh               ;compute x8 & xx7 as carry and parity
         pushf                       ;save the flags
         xor     dl,dh               ;compute R14 through R7
         xor     dh,dh               ;make xx7 and xx8 equal to 0
         popf                        ;restore the flags
         jpe     crcf1               ;if xx7 actually is 1, make xx7 and
         mov     dh,011b             ; xx8 equal to 1
crcf1:   jnc     crcf2               ;if x8 equals 1, then
         xor     dh,010b             ; complement xx8
crcf2:   mov     al,dl               ;load R14 through R7 in low CRC byte
         mov     ch,ah               ;save high CRC byte
         mov     ah,dh               ;load R16 and R15 in high CRC byte
         shr     dh,1                ;make xx8 the lsb
         mov     cl,6                ;shift CRC register six positions
         shl     ax,cl               ; to left
         or      al,dh               ;CRC register has R16 through R1
         xor     al,ch               ;xor with high CRC byte
         dec     si                  ;
         jnz     crcf0               ;keep looping till end of message

         pop     dx                  ;standard epilog
         pop     cx
         pop     si
         pop     di
         pop     es
         pop     bp
         ret

crcf_    endp
```

```
;                        LISTING 5

;              Reduced Table Look-up Algorithm
;                        (Byte-wise)

;*********************************************************
;*                                                       *
;*  crcrb (buff_addr,length)                             *
;*                                                       *
;*  This routine computes the check bits for any given message  *
;*  using the byte-wise reduced table look-up algorithm.  *
;*                                                       *
;*  Called from "C" program as :  crcrb(buff_addr,length);  *
;*                                                       *
;*  Input      :     buff_addr  -  message buffer address  *
;*                   length     -  message length (bytes)  *
;*                                                       *
;*  Output     :     check bits -  returned in ax        *
;*                                                       *
;*  NOTE:  The 16 entry (32 bytes) reduced look-up table  *
;*         starts at label "table2" within the data segment.  *
;*         This routine uses only the first eight entries of  *
;*         the table.                                    *
;*                                                       *
;*********************************************************

         public  crcrb_
crcrb_   proc    far

         push    bp                  ;standard prolog
         mov     bp,sp
         push    es
         push    di
         push    si
         push    bx
         push    cx
         push    dx

         les     di,6[bp]            ;load buffer address in es:di
         mov     si,10[bp]           ;load message length in si

         xor     ax,ax               ;initialize CRC register
crcrb0:  mov     dl,es:[di]          ;fetch a message byte
         inc     di                  ;bump message pointer
         xor     dl,al               ;xor message byte with low CRC byte
         mov     al,ah               ;shift CRC register 8 positions to
         xor     ah,ah               ; the right
         mov     cx,8                ;load byte length (8 bits) in cx
         xor     bx,bx               ;load table index in bx
crcrb1:  rcl     dl,1                ;check the msb of dl
         jnc     crcrb2              ;0 - no action
         xor     ax,table2[bx]       ;1 - modify CRC register
crcrb2:  add     bx,2                ;bump table index
         loop    crcrb1              ;keep looping till end of byte
         dec     si                  ;
         jnz     crcrb0              ;keep looping till end of message

         pop     dx                  ;standard epilog
         pop     cx
         pop     bx
         pop     si
         pop     di
         pop     es
         pop     bp
         ret

crcrb_   endp
```

```
;                         LISTING 6

;               Reduced Table Look-up Algorithm
;                         (Word-wise)


;***********************************************************
;*                                                         *
;*  crcrw (buff_addr,length)                               *
;*                                                         *
;*  This routine computes the check bits for any given message *
;*  using the word-wise reduced table look-up algorithm.   *
;*                                                         *
;*  Called from "C" program as :  crcrw(buff_addr,length);  *
;*                                                         *
;*  Input      :      buff_addr  -  message buffer address  *
;*                    length     -  message length (words)  *
;*                                                         *
;*  Output     :      check bits -  returned in ax         *
;*                                                         *
;*  NOTE:  The 16 entry (32 bytes) reduced look-up table   *
;*         starts at label "table2" within the data segment. *
;*                                                         *
;***********************************************************

        public  crcrw_
crcrw_  proc    far

        push    bp              ;standard prolog
        mov     bp,sp
        push    es
        push    di
        push    si
        push    bx
        push    cx
        push    dx

        les     di,6[bp]        ;load buffer address in es:di
        mov     si,10[bp]       ;load message length in si

        xor     ax,ax           ;initialize CRC register
crcrw0: mov     dx,es:[di]      ;fetch a message word
        add     di,2            ;bump message pointer
        xor     dx,ax           ;xor message word with CRC register
        xor     ax,ax           ;shift CRC register 16 positions to
                                ; the right
        mov     cx,16           ;load word length (16 bits) in cx
        xor     bx,bx           ;load table index in bx
crcrw1: rcl     dx,1            ;check the msb of dx
        jnc     crcrw2          ;0 - no action
        xor     ax,table2[bx]   ;1 - modify CRC register
crcrw2: add     bx,2            ;bump table index
        loop    crcrw1          ;keep looping till end of word
        dec     si              ;
        jnz     crcrw0          ;keep looping till end of message

        pop     dx              ;standard epilog
        pop     cx
        pop     bx
        pop     si
        pop     di
        pop     es
        pop     bp
        ret

crcrw_  endp


codeseg ends
```

Table 7.
Comparison of CRC algorithms.

Algorithm	Time (ms) (for 512 bytes)	Storage (bytes)
CRCB	61.2	56
CRCT	8.4	553
CRCF	18.8	67
CRCR(B)	53.3	72
CRCR(W)	50.9	88

In Table 7 we see the results of a comparison of the execution speeds of the different algorithms; we compared the times taken to encode a 512-byte long message. The bitwise algorithm is seen to be the slowest. In comparison, the reduced table lookup algorithms are about 20 percent faster. The on-the-fly and table lookup algorithms are faster by a factor of about 3 and 7. The table lookup algorithm requires the largest amount of storage, while the other algorithms have relatively moderate storage requirements. In terms of the flexibility to use different generator polynomials, all programs except the one implementing the on-the-fly algorithm require minimal recoding.

Acknowledgments

We wish to express our gratitude to Jim Davis of Iowa State University for his support and encouragement during the course of this work.

References

1. J.S. Whiting, "An Efficient Software Method for Implementing Polynomial Error Detection Codes," *Computer Design,* Mar. 1975, pp. 73-77.
2. R. Lee, "Cyclic Code Redundancy," *Digital Design,* July 1981, pp. 77-85.
3. A. Perez, "Byte-wise CRC Calculations," *IEEE Micro,* June 1983, pp. 40-50.
4. W.W. Peterson and D.T. Brown, "Cyclic Codes for Error Detection," *Proc. IRE,* Jan. 1961, pp. 228-235.
5. *ANSI/IEEE Std. 802.3-1985, "Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications,"* 1985.

Tenkasi V. Ramabadran is an assistant professor of electrical engineering and computer engineering at Iowa State University. His current research interests include data compression, error control coding, speech coding, and signal processing.

Ramabadran received his BE from Regional Engineering College, Trichy, India; his MTech from Indian Institute of Technology, Madras; and his MS and PhD from the University of Notre Dame. All were in electrical engineering.

Sunil S. Gaitonde is working for his PhD degree in electrical engineering and computer engineering at Iowa State University. His research interests include protocol design; local area network design and performance; and the integration of voice, data, and video on computer networks.

Gaitonde received a BTech degree in electrical engineering from the Indian Institute of Technology in Kharagpur and an MS in computer engineering from ISU.

Questions concerning this article can be addressed to Tenkasi Ramabadran, 332 Coover, Department of EE&CE, Iowa State University, Ames, Iowa 50011.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 165 **Medium** 166 **High** 167