



Edgar H. Sibley
Panel Editor

Cyclic Redundancy Check (CRC) codes provide a simple yet powerful method of error detection during digital data transmission. Use of a table look-up in computing the CRC bits will efficiently implement these codes in software.

COMPUTATION OF CYCLIC REDUNDANCY CHECKS VIA TABLE LOOK-UP

DILIP V. SARWATE

A Cyclic Redundancy Check (CRC) code provides a simple, yet powerful method for the detection of errors during digital data transmission. The transmitter, in a system employing CRC coding, transmits data bits followed by parity check bits that are usually called CRC bits. The CRC bits are related to the data bits as follows. The data bits are treated as the coefficients of a data polynomial. The CRC bits are the coefficients of the remainder when the data polynomial is divided by a fixed polynomial known as the CRC polynomial. Because of the presence of noise or other interference on the data link, the transmitted bits may not be received correctly, that is, the received data bits and CRC bits need not be the same as those transmitted.

The receiver detects the occurrence of such errors during transmission in the following ways. The received data bits are treated as the coefficients of a received data polynomial and the corresponding CRC bits are computed as the remainder when the received data polynomial is divided by the CRC polynomial. If the *computed* CRC bits are not the same as the *received* CRC bits, then the receiver knows that errors have occurred during the transmission. Usually a re-

transmission is requested in such cases. If the computed CRC bits are identical to the received CRC bits, the receiver assumes that the received data bits are error-free. This assumption is not always correct, and the received data bits may contain undetected errors. Fortunately, the probability of such undetected errors is quite small. Further discussion of CRC codes and retransmission schemes may be found in [2] and [5].

Typical CRC code implementations in hardware use feedback shift registers to implement polynomial division via the familiar long division method. Examples of such circuits for commonly used CRC codes are given in [1]. These circuits process one bit per clock cycle. Since much data transmission is also bit-by-bit or serial transmission, this approach can be quite satisfactory. This approach, however, requires the transmitter to determine the CRC bits after the data has been converted from parallel format (as in most digital systems, framed in bytes and words) to serial format. At the same time, the receiver must compute CRC bits and decide whether errors have occurred before the data has been converted from serial format to parallel format. On the other hand, in many communication networks, the data link layer [2] (which is responsible for error control and thus handles the CRC computations) is implemented in software. The development of effi-

cient methods for the implementation of CRC codes in software is of particular interest since a program that simply mimics the hardware method does far too many bit manipulations and runs far too slowly on most processors. An efficient software implementation must take advantage of the fact that computers handle bits in groups of bytes and words, and not as a serial bit stream.

We will focus on one efficient method for polynomial division and CRC code implementation. Instead of working on one bit of the dividend at a time, this method uses table look-ups to handle eight bits (one byte) at a time. A detailed description of the computation of CRC bits using a CRC polynomial of degree 16 is given. Depending on the capabilities of the processor, either two tables of 256 8-bit bytes (each accessed once for each byte of the dividend) or one table of 256 16-bit words (accessed once for each byte of the dividend) can be used. With appropriate changes, the same technique can be applied to the computation of CRCs using CRC polynomials of degree 32 in which case four tables of 256 8-bit bytes are needed. These tables can also be organized as two tables of 256 16-bit words or as one table of 256 32-bit double words. In all cases, each table is accessed once for each byte of the dividend. The method described in this paper requires fewer tables and fewer instructions, and thus is more efficient than the tea-leaf reader algorithm proposed by Griffiths and Stones [3].

CRC CODES AND ERROR DETECTION

Let $g(x)$ denote a CRC polynomial of degree 16. Three commonly used polynomials are $x^{16} + x^{15} + x^{13} + x^8 + x^6 + x^3 + x + 1$ used by Griffiths and Stones [3] in their examples; $x^{16} + x^{12} + x^5 + 1$, used in the X.25 standard (this polynomial is also called the CRC-CCITT or the ADCCP polynomial); and the CRC-16 polynomial $x^{16} + x^{15} + x^2 + 1$ [1, 2, 5]. Let $d_{N-1}, d_{N-2}, d_{N-3}, \dots, d_2, d_1, d_0$ denote the N data bits to be transmitted in descending order of subscripts. The data polynomial $d(x)$ is defined to be $d_{N-1}x^{N+15} + d_{N-2}x^{N+14} + d_{N-3}x^{N+13} + \dots + d_1x^{17} + d_0x^{16}$. On dividing $d(x)$ by $g(x)$, a remainder $b(x) = b_{15}x^{15} + b_{14}x^{14} + \dots + b_1x + b_0$ of degree at most 15 is obtained. Note that when performing the polynomial division, the various terms are added modulo 2. The CRC bits are the coefficients $b_{15}, b_{14}, \dots, b_1, b_0$ of $b(x)$. The $N + 16$ bits transmitted are $d_{N-1}, d_{N-2}, d_{N-3}, \dots, d_2, d_1, d_0, b_{15}, b_{14}, \dots, b_1, b_0$. The transmitted codeword polynomial is $c(x)$ where

$$\begin{aligned} c(x) &= c_{N+15}x^{N+15} + c_{N+14}x^{N+14} + c_{N+13}x^{N+13} \\ &\quad + \dots + c_{17}x^{17} + c_{16}x^{16} + c_{15}x^{15} + c_{14}x^{14} \\ &\quad + c_{13}x^{13} + \dots + c_2x^2 + c_1x + c_0 \\ &= d_{N-1}x^{N+15} + d_{N-2}x^{N+14} + d_{N-3}x^{N+13} \\ &\quad + \dots + d_1x^{17} + d_0x^{16} + b_{15}x^{15} + b_{14}x^{14} \\ &\quad + b_{13}x^{13} + \dots + b_2x^2 + b_1x + b_0 \\ &= d(x) + b(x). \end{aligned}$$

Since $d(x) = h(x)g(x) + b(x)$ implies that $d(x) + b(x) = h(x)g(x)$ modulo 2, the transmitted codeword polynomial $c(x)$ is a multiple of $g(x)$, the CRC polynomial.

Let $r_{N+15}, r_{N+14}, r_{N+13}, \dots, r_2, r_1, r_0$ denote the bits received when the codeword bits $c_{N+15}, c_{N+14}, c_{N+13}, \dots, c_2, c_1, c_0$ are transmitted. Let the received word polynomial be $r(x)$ where

$$\begin{aligned} r(x) &= r_{N+15}x^{N+15} + r_{N+14}x^{N+14} + r_{N+13}x^{N+13} \\ &\quad + \dots + r_{16}x^{16} + r_{15}x^{15} + r_{14}x^{14} + r_{13}x^{13} \\ &\quad + \dots + r_2x^2 + r_1x + r_0 \\ &= \hat{d}(x) + r_{15}x^{15} + r_{14}x^{14} + r_{13}x^{13} \\ &\quad + \dots + r_2x^2 + r_1x + r_0 \end{aligned}$$

where

$$\hat{d}(x) = \hat{d}_{N-1}x^{N+15} + \hat{d}_{N-2}x^{N+14} + \dots + \hat{d}_1x^{17} + \hat{d}_0x^{16}$$

is the received data polynomial. In the absence of errors, $r(x)$ equals $c(x)$ and is therefore a multiple of $g(x)$. If $r(x)$ is not a multiple of $g(x)$, then it is certain that errors must have occurred during transmission. On the other hand, if $r(x)$ is a multiple of $g(x)$, the receiver cannot be sure that errors did not occur (because errors could have changed the transmitted multiple $c(x) = h(x)g(x)$ into some *other* multiple $\hat{h}(x)g(x)$). Fortunately, such a transmutation is quite unlikely, and the receiver accepts $\hat{d}(x)$ as an error-free data polynomial whenever $r(x)$ is a multiple of $g(x)$.

The receiver checks whether $r(x)$ is a multiple of $g(x)$ by the obvious method of dividing $r(x)$ by $g(x)$ to see if a zero remainder is obtained. This can also be viewed as computing the CRC bits that correspond to the received data polynomial $\hat{d}(x)$ and checking to see whether they are the same as the received CRC bits $r_{15}, r_{14}, \dots, r_1, r_0$. To see this, note that if $\hat{d}(x) = \hat{a}(x)g(x) + \hat{b}(x)$ where $\hat{b}(x)$ is the remainder (the computed CRC bits), then $\hat{d}(x) + \hat{b}(x)$ modulo 2 is a multiple of $g(x)$. Thus, $r(x)$ is a multiple of $g(x)$ if and only if $r(x) + \hat{d}(x) + \hat{b}(x)$ is a multiple of $g(x)$. Nevertheless, $r(x) + \hat{d}(x) + \hat{b}(x) = r_{15}x^{15} + r_{14}x^{14} + \dots + r_1x + r_0 + \hat{b}(x)$ modulo 2. Since the right hand side is of degree at most 15, while $g(x)$ is of degree 16, it follows that $r(x)$ is a multiple of $g(x)$ if and only if the right hand side is identically zero. That is, if and only if the *computed* CRC bits $\hat{b}_{15}, \hat{b}_{14}, \dots, \hat{b}_1, \hat{b}_0$ are identical to the *received* CRC bits $r_{15}, r_{14}, \dots, r_1, r_0$.

For implementation purposes, a third approach is useful. The polynomial $r(x)$ is a multiple of $g(x)$ if and only if $x^{16}r(x)$ is a multiple of $g(x)$. Thus, one can divide $x^{16}r(x)$ by $g(x)$ instead of dividing $r(x)$ by $g(x)$. The advantage to this approach is that the algorithms for computing CRC bits at the transmitter, and error detection at the receiver, become nearly identical, and can be implemented via a single procedure. This procedure processes its input until the last bit (d_0 at the transmitter, r_0 at the receiver) has been used. Following the processing of the last input bit, the remainder (which can be thought of as the value returned by the procedure) can be interpreted at the transmitter as the set of

CRC bits to be transmitted, while the receiver accepts the received data only if the remainder is zero. Another advantage of the procedure is in packet communications with variable-length packets: the CRC procedure does not need to know the number of bits to be processed but simply operates until an end-of-file mark is read, or until the last bit (or byte in an array) has been used.

CRC ALGORITHMS USING TABLE LOOKUP

Let $N = 8n$, and suppose that the N data bits $d_{N-1}, d_{N-2}, d_{N-3}, \dots, d_2, d_1, d_0$ form n eight-bit data bytes $D_{n-1}, D_{n-2}, D_{n-3}, \dots, D_2, D_1, D_0$ to be transmitted in descending order of subscripts. The CRC bits $b_{15}, b_{14}, \dots, b_1, b_0$ form two CRC bytes B_1 and B_0 , and are transmitted following the last data byte D_0 . The $n + 2$ bytes transmitted are called the codeword bytes $C_{n+1}, C_n, C_{n-1}, \dots, C_2, C_1, C_0$, where $C_i = D_{i-2}$ for $n + 1 \geq i \geq 2$, and $C_1 = B_1$ and $C_0 = B_0$ are the CRC bytes. Corresponding to these $n + 2$ transmitted code bytes, the $n + 2$ received bytes are $R_{n+1}, R_n, R_{n-1}, \dots, R_2, R_1, R_0$, of which $R_{n+1}, R_n, R_{n-1}, \dots, R_2$ are the received data bytes $\hat{D}_{n-1}, \hat{D}_{n-2}, \hat{D}_{n-3}, \dots, \hat{D}_0$, and R_1 and R_0 are the received CRC bytes.

The program fragment that follows computes two CRC bytes C_1 and C_0 from m data bytes stored in an array A . Two arrays (tables) f_1 and f_0 of 256 bytes each are used. The byte T is used to compute the relative address of the table elements to be looked up at each step. Note that the elements of the array A are not modified at all. A detailed discussion of the derivation of this algorithm and the construction of the tables f_1 and f_0 is given in the appendix.

```

 $C_1 \leftarrow 0; \quad C_0 \leftarrow 0;$ 
for  $i \leftarrow m - 1$  step  $-1$  until  $0$  do
  begin
    comment  $\oplus$  denotes the
      Exclusive-OR (XOR) operation;
     $T \leftarrow C_1 \oplus A[i];$ 
     $C_1 \leftarrow C_0 \oplus f_1[T];$ 
     $C_0 \leftarrow f_0[T]$ 
  end

```

The program fragment can be used as part of a procedure whose arguments include the array A and its size m , and which returns C_1 and C_0 as the value computed by the procedure. In fact, if the procedure arguments include the arrays f_0 and f_1 , then the same procedure can be used to compute CRC bytes for different CRC polynomials of degree 16. If $m = n$ and the array A contains the transmitted data bytes $D_{n-1}, D_{n-2}, D_{n-3}, \dots, D_2, D_1, D_0$, then $C_1 = B_1$ and $C_0 = B_0$ are the CRC bytes to be transmitted. If $m = n + 2$ and the array A contains the received bytes $R_{n+1}, R_n, R_{n-1}, \dots, R_2, R_1, R_0$, then the received data bytes $R_{n+1}, R_n, R_{n-1}, \dots, R_2$ are accepted as error-free if and only if C_1 and C_0 are zero. Thus, the same procedure can be used at the transmitter and the receiver provided that the results

returned by the procedure are interpreted appropriately.

In the program fragment just described, both table lookup operations use the same index T . Since many processors can fetch 16-bit operands in one memory cycle, it may be advantageous to combine the two byte tables f_1 and f_0 into a word table f containing 256 16-bit words. The high-order byte of the table entry $f[T]$ (the T -th word in the table f , where T is interpreted as an integer in the range 0 to 255) is $f_1[T]$, and the low-order byte is $f_0[T]$, so that a single table lookup of a 16-bit word fetches both $f_1[T]$ and $f_0[T]$. The reader is reminded that a minor complication arises if assembly language is used for the implementation. Generally, memory is organized so that each byte has a different address. Thus, the addresses of the two bytes fetched are offset by $2T$ and $2T + 1$ from the base address f of the table, and T may need to be multiplied by two (shifted left one bit) before being used as an index for the word table f . High-level languages, of course, take care of the address conversion if f is declared to be an array of 16-bit words.

One other speed-up relies on the fact that many processors can Exclusive OR the contents of a 16-bit word in memory into a 16-bit register or a 16-bit word. Thus, a single instruction can be used to change both C_1 and C_0 . The program fragment shown previously is easily modified to use a single table. In the modified version that follows, it is assumed that C_1 and C_0 are respectively the high-order and low-order bytes of a 16-bit register or word denoted by C .

```

 $C \leftarrow 0;$ 
for  $i \leftarrow m - 1$  step  $-1$  until  $0$  do
  begin
    comment  $\oplus$  denotes the
      Exclusive-OR (XOR) operation;
     $T \leftarrow C_1 \oplus A[i];$ 
     $C_1 \leftarrow C_0; \quad C_0 \leftarrow 0;$ 
    comment  $f[T]$  denotes the
       $T$ -th word in the table  $f$ ;
     $C \leftarrow C \oplus f[T]$ 
  end

```

Both program fragments process the data bytes in the array A one at a time, and can be readily modified to process the data on-line, for example, by replacing the loop controlled by the **for** statement with a loop controlled by a **while** statement which reads and processes bytes until an end-of-file is encountered.

Algorithms for the computation of 32 bit CRCs are very similar to the ones shown above. If the discussion in the appendix is carefully followed there will be no difficulty in understanding how the following program fragment was obtained or how the tables were designed. This program fragment uses tables f and \hat{f} with 256 16-bit words that are always accessed with a single address. (In fact, a single 32-bit operand fetch could also be used, or conversely, the two tables could be split into four tables each containing 256 8-bit bytes).

Note that X and Y are two 16-bit registers or words containing four CRC bytes. The high-order bytes in X and Y are denoted as X_1 and Y_1 respectively, while the low-order bytes are denoted respectively as X_0 and Y_0 . At the transmitter, the CRC bytes would be transmitted in the order X_1, X_0, Y_1, Y_0 .

```

 $X \leftarrow 0; \quad Y \leftarrow 0;$ 
for  $i \leftarrow m - 1$  step  $-1$  until  $0$  do
  begin
     $T \leftarrow X_1 \oplus A[i];$ 
     $X_1 \leftarrow X_0; \quad X_0 \leftarrow Y_1; \quad Y_1 \leftarrow Y_0; \quad Y_0 \leftarrow 0;$ 
    comment  $f[T]$  and  $\hat{f}[T]$  denote the
       $T$ -th words in the tables  $f$  and  $\hat{f}$ ;
     $X \leftarrow X \oplus f[T]; \quad Y \leftarrow Y \oplus \hat{f}[T]$ 
  end

```

If the word tables f and \hat{f} are split up into four 256-byte tables, then the last line in this procedure would read as

```

 $X_1 \leftarrow X_1 \oplus f_1[T]; \quad X_0 \leftarrow X_0 \oplus f_0[T];$ 
 $Y_1 \leftarrow Y_1 \oplus \hat{f}_1[T]; \quad Y_0 \leftarrow \hat{f}_0[T]$ 

```

In this form, the algorithm uses four shifts and four Exclusive ORs for each byte that is processed.

It is interesting to compare this algorithm to the tea-leaf reader algorithm proposed by Griffiths and Stones [3]. The latter uses five 256-byte tables while the algorithm proposed here uses only four 256-byte tables. Four shifts and five Exclusive OR instructions are required by the tea-leaf algorithm compared to four shifts and only four Exclusive OR instructions for the algorithm proposed here. Thus, the algorithm described here is faster and requires less memory space for tables than the tea-leaf reader algorithm. Furthermore, the tables are all accessed with the same address (the tea-leaf reader algorithm does not enjoy this property) so that table lookups can be combined together. The ability to combine byte-to-byte Exclusive-OR instructions into word-to-word Exclusive OR instructions provides further speed advantages to the algorithm described here.

Thus the algorithms described in this article provide the most efficient means of computing CRCs of the table look-up in software.

Acknowledgment. The preparation of this paper was supported by the Joint Services Electronics Program under Contract N00014-84-C-0149.

APPENDIX

A brief review of polynomial division is helpful in understanding the development of the CRC algorithms described in this article. A CRC polynomial of degree 16 is used, and some of the details are explained using the CRC-CCITT polynomial $x^{16} + x^{12} + x^5 + 1$. The method can be applied to other CRC polynomials of degree 16, however, and is readily adapted for use with CRC polynomials of degrees 24 or 32 or more.

Polynomial Division

In the familiar process of polynomial division, multiples of the divisor $g(x)$ are subtracted from the dividend $d(x)$ until the remainder is of smaller degree than $g(x)$. The classical polynomial division algorithm has two important characteristics. First, the process is essentially iterative in that the highest degree term of $d(x)$ is removed by subtracting off an appropriate multiple of $g(x)$, and then the highest degree term in what is left is removed

by subtracting off some other appropriate multiple of $g(x)$, and so on. Since each subtraction changes the values of the remaining high-degree terms, the later subtractions must wait for the completion of the earlier ones. The second important characteristic is that the multiples, which are subtracted, are all of the form $\alpha x^i g(x)$ since the use of this form facilitates the computation of the quotient (indeed the i -th degree term of the quotient is just αx^i). In CRC computations, however, the quotient is of no interest at all: it is the remainder that is important. For any arbitrary polynomial $p(x)$, the remainder, when $d(x) - p(x)g(x)$ is divided by $g(x)$, is the same as the remainder when $d(x)$ is divided by $g(x)$. When only the remainder is of interest, polynomial division can be speeded up by careful choice of $p(x)$. In fact, by using different multiples of $g(x)$, several steps can be done simultaneously. A very useful set of multiples of $g(x)$ is exhibited as follows:

Let $g_0(x), g_1(x), g_2(x), \dots, g_7(x)$ denote polynomial multiples of $g(x)$. Each $g_i(x)$ is of the form

$$x^{16+i} + g_{i,15}x^{15} + g_{i,14}x^{14} + \dots + g_{i,13}x^{13} + \dots + g_{i,2}x^2 + g_{i,1}x + g_{i,0}.$$

These polynomials can be found easily as follows:

$$g_0(x) = g(x),$$

$$g_i(x) = xg_{i-1}(x) - g_{i-1,15}g(x), \quad 0 < i < 8.$$

For example, if $g(x)$ is the CRC-CCITT polynomial $x^{16} + x^{12} + x^5 + 1$, then

$$\begin{aligned} g_0(x) &= x^{16} + x^{12} + x^5 + 1 \\ g_1(x) &= x^{17} + x^{13} + x^6 + x \\ g_2(x) &= x^{18} + x^{14} + x^7 + x^2 + x \\ g_3(x) &= x^{19} + x^{15} + x^8 + x^3 + 1 \\ g_4(x) &= x^{20} + x^{16} + x^9 + x^4 + x \\ g_5(x) &= x^{21} + x^{17} + x^{10} + x^5 + x^2 + 1 \\ g_6(x) &= x^{22} + x^{18} + x^{11} + x^6 + x^3 + x \\ g_7(x) &= x^{23} + x^{19} + x^{12} + x^7 + x^4 + x^2 + 1 \end{aligned}$$

Note that $g_i(x) - x^{16+i}$ has degree less than 16, and its coefficients can be stored in two bytes $G_{i,1}$ and $G_{i,0}$. For example, $G_{5,1} = (0010010) \leftrightarrow x^{13} + x^{10}$, and $G_{5,0} = (01100010) \leftrightarrow x^6 + x^5 + x$. Linear combinations of the coefficients of these polynomials are stored in two 256-byte tables whose entries are obtained as follows.

For $0 \leq i < 256$, let $I = (i_7, i_6, \dots, i_1, i_0)$ be the binary representation of i such that $i = i_7 2^7 + i_6 2^6 + \dots + i_1 2 + i_0$. The i -th entry in the byte table f_1 is referred to as $f_1[I]$ and is given by $i_7 G_{7,1} \oplus i_6 G_{6,1} \oplus \dots \oplus i_0 G_{0,1}$. Here \oplus denotes the bit-by-bit Exclusive OR of the bytes. Similarly, the i -th entry in the byte table f_0 is referred to as $f_0[I]$ and is given by $i_7 G_{7,0} \oplus i_6 G_{6,0} \oplus \dots \oplus i_0 G_{0,0}$. Note that the three bytes $I, f_1[I]$ and $f_0[I]$ together specify the coefficients of the polynomial $i_7 g_7(x) + i_6 g_6(x) + \dots + i_0 g_0(x)$. This fact is crucial to understanding the speeded-up division algorithm.

Consider the process of dividing $x^{16}a(x)$ by $g(x)$ where

$$a(x) = a_{8m-1}x^{8m-1} + a_{8m-2}x^{8m-2} + \dots + a_1x + a_0.$$

The coefficients of $a(x)$ are stored in m bytes $A_{m-1}, A_{m-2}, A_{m-3}, \dots, A_1, A_0$, where $A_i = (a_{8i+7}, a_{8i+6}, a_{8i+5}, \dots, a_{8i})$. In particular, note that

$$\begin{aligned} A_{m-1} &= (a_{8(m-1)+7}, a_{8(m-1)+6}, a_{8(m-1)+5}, \dots, a_{8(m-1)}) \\ &= (a_{8m-1}, a_{8m-2}, a_{8m-3}, \dots, a_{8(m-1)}) \end{aligned}$$

It is assumed that two zero bytes A_{-1} and A_{-2} follow A_0 and are used to store the remainder. The highest degree term in $x^{16}a(x)$, namely $a_{8m-1}x^{8(m+1)+7}$, can be removed by subtracting $a_{8m-1}x^{8(m-1)}g_7(x)$ from $x^{16}a(x)$. Since $g_7(x)$ has no terms of degrees 22, 21, \dots , 16, $x^{8(m-1)}g_7(x)$ has no terms of degree $8(m+1)+6, 8(m+1)+5, \dots, 8(m+1)$. Thus the highest degree term in the remainder after subtracting off $a_{8m-1}x^{8(m-1)}g_7(x)$

from $x^{16}a(x)$ is clearly $a_{8m-2}x^{8(m+1)+6}$, and this can be removed by subtracting $a_{8m-2}x^{8(m-1)}g_6(x)$ from the remainder. Since $x^{8(m-1)}g_6(x)$ has no terms of degree $8(m+1)+5, 8(m+1)+4, \dots, 8(m+1)$, the highest degree term in the remainder is clearly $a_{8m-3}x^{8(m+1)+5}$ which can be removed by subtracting $a_{8m-3}x^{8(m-1)}g_5(x)$, and so on. In short, $x^{8(m-1)}[a_{8m-1}g_7(x) + a_{8m-2}g_6(x) + \dots + a_{8(m-1)}g_0(x)]$ can be subtracted from $x^{16}a(x)$, thus completing eight iterations of the division algorithm simultaneously.

All this looks very complicated but is actually quite simple. The coefficients of the polynomial $a_{8m-1}g_7(x) + a_{8m-2}g_6(x) + \dots + a_{8(m-1)}g_0(x)$ are the three bytes $A_{m-1}, f_1[A_{m-1}]$, and $f_0[A_{m-1}]$. Consequently, all that actually needs to be done is to replace A_{m-2} and A_{m-3} by $A_{m-2} \oplus f_1[A_{m-1}]$ and $A_{m-3} \oplus f_0[A_{m-1}]$, respectively, and then discard A_{m-1} . This leaves a polynomial of degree $8(m+1)-8$, and the process can be repeated. At each iteration, the table entries in f_1 and f_0 addressed by the high-order byte are added (Exclusive OR addition) into the next two lower order bytes, and the high order byte is discarded. For the last two iterations during which A_1 and A_0 are being processed, the lower order bytes A_{-1} and A_{-2} (which are initially zero) are used, and at the end of the division process, these contain the remainder. Note that two table lookups and two Exclusive OR operations are used to process each byte in $a(x)$.

A More Useful Division Algorithm

The method for polynomial division just given, is very similar to the one given by Knuth [4]. The coefficients of the dividend $a(x)$ are modified at each step, giving this method an undesirable property. Both the transmitter and the receiver, however, need the unmodified coefficients of the dividend, since, at the transmitter,

the dividend consists of the data bytes to be transmitted, while at the receiver, the dividend consists of the data bytes that may (or may not) be accepted as error-free received data. Of course, the array can be copied so that the CRC algorithms can work on a private copy of the array. Copying arrays, however, is very time-consuming. Instead, the division algorithm can be modified so that the dividend is not changed at all. To understand how the modified division algorithm works, consider that in the straightforward division algorithm, the byte A_{i-2} is modified during exactly two iterations, namely, when the bytes A_i and A_{i-1} are the high order bytes in what is left. The modifications to be made to A_{i-2} during these iterations can be saved separately.

Also, since the only use of each modified A_i is as an address for a table lookup (followed by the byte discarded in the straightforward division algorithm), the address to be used can also be formed separately. The modified division algorithm follows. At the beginning of each iteration, the modifications to A_i (generated during the previous two iterations) are in byte C_1 while the modification to A_{i-1} (generated during the previous iteration)

is in byte C_0 . The address for table lookup is formed in byte T . The modifications to A_{i-1} and A_{i-2} generated during this iteration are $f_1[T]$ and $f_0[T]$, respectively. The former is stored in byte C_1 together with the contents of C_0 which contains the modification to A_{i-1} generated during the previous iteration. In this manner, C_0 is made available for storing $f_0[T]$. Although this modified algorithm does require additional storage as compared to the straightforward division algorithm, the amount is trivially small and easily justified in view of the savings in time achieved.

```

 $C_1 \leftarrow 0; C_0 \leftarrow 0;$ 
for  $i \leftarrow m - 1$  step  $-1$  until  $0$  do
  begin
    comment  $\oplus$  denotes the
      Exclusive-OR (XOR) operation;
     $T \leftarrow A_i \oplus C_1;$ 
     $C_1 \leftarrow C_0 \oplus f_1[T];$ 
     $C_0 \leftarrow f_0[T]$ 
  end

```

REFERENCES

1. Arazi, B. *A Commonsense Approach to the Theory of Error Correcting Codes*. MIT Press, Cambridge, Mass., 1988.
2. Bertsekas, D., and Gallager, R. *Data Networks*. Prentice-Hall, Englewood Cliffs, N.J., 1987.
3. Griffiths, G., and Stones, G.C. The tea-leaf reader algorithm: An efficient implementation of CRC-16 and CRC-32. *Commun. ACM* 30, 7 (July 1987), 617-620.
4. Knuth, D.E. *The Art of Computer Programming: Vol. 2—Seminumerical Algorithms*. Second edition. Addison-Wesley, Reading, Mass., 1981.
5. Tanenbaum, A.S. *Computer Networks*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

CR Categories and Subject Descriptors: D.1.m [Programming Techniques]: Miscellaneous; D.2.3 [Software Engineering]: Coding; D.2.m [Software Engineering]: Miscellaneous

General Terms: Algorithms, cyclic redundancy check codes, error-control coding, error detection, reliability

Additional Key Words and Phrases: CRC algorithms, cyclic codes

Author's Present Address: Dilip V. Sarwate, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1101 W. Springfield Avenue, Urbana, IL 61801.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

JOURNAL OF THE ASSOCIATION FOR COMPUTING MACHINERY

Subscriptions **\$15.00/year**
for ACM members;
\$75.00/year for nonmembers.

(Members please include
member #)

An excellent source to
information on computer theory
and research in...

- Algorithm & complexity theory
- Artificial intelligence
- Combinatorics & graph theory
- Computer organization & design
- Systems modeling & analysis
- Database theory & structures
- Distributed computing
- Formal languages
- Computational models
- Numerical analysis
- Operating systems and research
- Programming languages & related methodology
- Computational theory

Published four times a year
(ISSN 0004-5411)

Write for an order form and your
ACM Publications Catalog to:



Catherine Yunque,
ACM,
11 West 42nd Street,
New York, NY 10036