Cyclic redundancy checking by program

by P. E. BOUDREAU and R. F. STEEN

IBM Corporation
Research Triangle Park, N.C.

INTRODUCTION

Recent advances in the use of mini-computers as control elements of a computer complex and as intelligent terminals[1] are indicative of a trend toward relocation of certain hardware functions to microprogram or machine level program. One such function which is a particularly good candidate, for various reasons, has already been moved into program in several machines (e.g., IBM System 360/25 Integrated Communication Adapter[2] and the IBM 1130[3]). This function is error control using an error detection Cyclic Redundancy Check (CRC). A CRC is a variable length shortened cyclic code in which a message is a code word if, and only if, the message polynomial $M(x)$ is divisible by the generator polynomial $G(x)$.

Error detection and correction codes have been studied extensively for more than 15 years. The most comprehensive references,[4,5] as well as the majority of papers written in the area, measure the encoding and decoding complexity in terms of the cost of hardware and the time for decoding. With some notable exceptions,[6,7] very little attention is given to the problem of encoding and decoding using machine level or microinstructions. However, in some cases such as the Berlekamp algorithm[3] for BCH codes, it may very possibly be easier to write a program for certain steps of the decoding procedure than to design hardware. Programmed error correction is especially appealing for use with high rate codes when error probabilities are low, since, in this case, a major portion of the correction process need only be performed when errors actually occur. Allocation of a significant amount of hardware for these relatively infrequent events is expensive. Furthermore, rapidly advancing memory technology helps to make program-controlled devices not only economically feasible but attractive.

One part of the problem is addressed in this paper. It is the problem of encoding or generating check bits. The solution, however, also applies to the decoding problem for error detection codes of this type. A similar approach, based on the properties of the companion matrix, has been used for parallel hardware devices.[8,9] With this approach, efficient and attractive programs can be developed for software or firmware. Subroutines developed here require as few as six instructions with sequential instruction execution to update a 16-bit remainder for eight new information bits. A program directly simulating a shift register would require at least three instructions (EXCLUSIVE OR, SHIFT, and BRANCH) per bit, or 24 instructions for an eight-bit update.

MATRIX APPROACH TO CYCLIC CODES

In this section, we review the relationship between multiplication by the companion matrix and polynomial division used to generate a code word. We then generalize the operation to an m-bit character-by-character operation developing a matrix equation to update the calculated redundancy m bits at a time. The appendix will be helpful to those familiar with the shift register in order to further justify the connection between the shift register operation and the matrix multiplication.

Generally, the check bit generation process is one of determining $R(x) = x^h I(x) \bmod G(x)$ where $I(x)$ is the polynomial whose coefficients are the information bits and h is the number of check bits. We can next let the coefficients of $R(x)$ be an h bit vector, R, and let G be the h by h companion matrix shown below. The binary digits, g_i, $i = 1, 2, 3 \ldots h-1$, are the coefficients of the generator polynomial.

$$G=\begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ & & & \cdot & \\ & & & \cdot & \\ & & & \cdot & \\ 0 & 0 & 0 & \dots & 1 \\ 1 & g_1 & g_2 & \dots & g_{h-1} \end{bmatrix}$$

Then, if we let $b(1)=i_{k-1}$ be the first information bit (the $k-1$th coefficient of $I(x)$) and $b(k)=i_0$ be the last information bit, it is clear (see the appendix or Reference 7) that the remainder R can be calculated iteratively using the following formula:

$$A(t+1)=\{A(t)+[0, 0, \dots, 0, b(t+1)]\}\cdot G \quad (1)$$

and setting $R=A(k)$. It should be noted that $A(t)$ represents the remainder of $x^h I_t(x)$ divided by $G(x)$ which is the calculated redundancy after the first t information bits, $I_t(x)$, have been taken into account.

We now define $B(t+1)=[0, 0, \dots, 0, b(t+1)]$ and rewrite Equation (1 or A2) as

$$A(t+1)=[A(t)+B(t+1)]G. \quad (2)$$

Equation (2) is the basic matrix description of the polynomial division process (circuit function) on a bit-by-bit basis. The advantage of the matrix approach is realized when one extends it to a multibit or character level. We can do this for m bits-per-character as follows, assuming $m \leq h$. Repeated use of Equation (2) yields:

$$\begin{aligned} A(t+m) &= [A(t+m-1)+B(t+m)]\cdot G \\ &= \{[A(t+m-2)+B(t+m-1)]\cdot G \\ &\quad +B(t+m)\}\cdot G \\ &\quad \vdots \\ &= A(t)\cdot G^m + \sum_{j=1}^{m} B(t+j)\cdot G^{m-j+1}. \end{aligned} \quad (3)$$

Equation (3) expresses the remainder at time $t+m$ in terms of the remainder at time t and the next m input bits $b(t+1), b(t+2), \dots, b(t+m)$. This equation can be put into a better form by using the "shifting" property of the companion matrix G.

$$A(t+m)=A(t)\cdot G^m + [0, 0, \dots, 0, b(t+m), b(t+m-1), \dots, b(t+1)]\cdot G^m. \quad (4)$$

If indeed we are operating with m bits per character and $A(t)$ is the remainder after some character has been sent, then $A(t+m)$, given by Equation (4), is the remainder after the next character has been sent and $b(t+m), b(t+m-1), \dots, b(t+1)$ is the bit string of length m representing that next character, where $b(t+1)$ is the first bit sent.

Since we will be using this from now on, it is convenient to make a slight change of notation. We define

$$A_j=[a_{0,j}, a_{1,j}, \dots, a_{h-1,j}]$$

as the remainder after the jth character, and

$$C_j=[0, 0, 0, \dots, 0, c_{0,j}, c_{1,j}, \dots, c_{m-1,j}]$$

as an h component vector where

$$c_{0,j}, c_{1,j}, c_{2,j} \dots c_{m-1,j}$$

is the bit string of length m representing the jth character and $c_{m-1,j}$ is the first bit of the character transmitted. That is

$$a_{i,j}=a_i(t) \qquad \text{for} \quad i=0, 1, \dots, h-1$$

and

$$\begin{aligned} c_{0,j} &= b(t+m) \\ c_{1,j} &= b(t+m-1) \\ &\vdots \\ c_{m-1,j} &= b(t+1). \end{aligned}$$

With this notation Equation (5) becomes the character-by-character version of Equation (2)

$$A_{j+1}=[A_j+C_{j+1}]\cdot G^m. \quad (5)$$

This equation expresses the remainder after $j+1$ characters as a function of the remainder after j characters and the $j+1$st character for $m \leq h$ bits per character. It is the fundamental result which we apply below.

MATRIX IMPLEMENTATION OF CYCLIC CODES

This matrix description of cyclic checking leads directly and intuitively to several different programmed checking implementations. It is this feature which makes the approach valuable. Since instruction sets, core availability, and instruction execution times vary widely, three approaches will be described.

It is very convenient to describe these subroutines in APL[10] with a single line of APL representing a single machine language instruction. For those interested in the exact operation of the simulated machine language instruction, a knowledge of basic APL is required; otherwise, the marginal machine instructions and

comments should clearly indicate the general nature of the operation on each line of code. It is assumed that there are four 16-bit registers which are available to the programmer. These are represented by the APL vector variables RA, RB, and RC with the fourth being the base register which is used for the return branch to the main program. In APL, RA[1;] represents the high order byte of register RA and RA[2;] represents the low-order byte of the same register. The storage area for tables is represented by the matrix SA which is as large as necessary.

Although we have assumed a 16-bit data path for the three examples, it is easy to write similar subroutines for an eight-bit ALU by partitioning the G^8 matrix in a different manner. We will use the terms, "byte" and "halfword" to mean eight and 16 bits respectively.

In general, our methods below are iterative schemes for finding the remainder using the recurrence relationship

$$A_{j+1}=[A_j+C_{j+1}]G^m.$$

For simplicity we define what we call a "working remainder" W_{j+1},

$$\begin{aligned}W_{j+1}&=[A_j+C_{j+1}]\\&=[a_{0,j},\ \ldots,\ a_{h-m-1,j},\ (a_{h-m,j}\oplus c_{0,j+1}),\\&\quad \ldots,\ (a_{h-1,j}\oplus c_{m-1,j+1})]\\&=[w_{0,j+1},\ w_{1,j+1},\ \ldots,\ w_{h-1,j+1}]\end{aligned}$$

Basically, our problem is to find A_{j+1} given W_{j+1} and G^m using

$$A_{j+1}=W_{j+1}G^m.$$

Since W_{j+1} is a binary vector of length h, it can take no more than 2^h values. The following methods, called the "one-256-halfword-table look-up," the "two-32-halfword-table look-up," and the "binary summation" method, are various ways to perform this job.

Purely for ease of notation, we now fix the values of h and m. We will let the number of parity bits be $16(h=16)$ and the number of bits per character be eight $(m=8)$. Substitution in (5) gives us the fundamental equation

$$A_{j+1}=W_{j+1}G^8 \tag{6}$$

where

$$\begin{aligned}W_{j+1}&=A_j+C_{j+1}\\A_0&=[0,\ 0,\ 0,\ \ldots,\ 0,\ 0]\\A_j&=[a_{0,j},\ a_{1,j},\ \ldots,\ a_{15,j}]\\C_j&=[0,\ 0,\ \ldots,\ 0,\ c_{0,j},\ c_{1,j},\ \ldots,\ c_{7,j}]\end{aligned}$$

are all 16-bit vectors, and

G^8 = the companion matrix raised to the 8th power.

We note again for emphasis that $c_{7,j}$ is the first bit of the jth character while the $j=1$st character is the first character transmitted or received.

One-256-halfword-table look-up method

This is a simple one-table look-up method which requires a significant amount of storage and frequently will be impractical for codes with more than eight bits-per-character. However, it embodies most of the basic ideas of the matrix approach and is a good starting place. In an instruction set with the logical EXCLUSIVE OR operation, the forming of W_{j+1} is trivial. The next step is to find A_{j+1} which can be found by multiplying W_{j+1} by G^8. This can be done very rapidly by table look-up. Rather than blindly storing all 2^{16} halfwords which can result from this operation, we notice that G^8 has the form

$$G^8=\left[\frac{0\mid I}{X}\right].$$

Thus $W_{j+1}G^8$ can be written

$$W_{j+1}{}^{(L)}X\oplus W_{j+1}{}^{(H)}[0\mid I]$$

where $W_{j+1}{}^{(H)}$ is an eight-bit vector comprising the high-order eight bits of W_{j+1} and $W_{j+1}{}^{(L)}$ represents the low-order eight bits of W_{j+1}. If byte operations are available, the product $W_{j+1}{}^{(H)}\cdot[0\mid I]$ is simply moving the byte from the high-order half of a 16-bit register to the low-order half. The second instruction in Table I performs this operation. The second product above requires a table look-up for one of 256 halfwords representing all possible values of $W_{j+1}{}^{(L)}\cdot X$. This is done in instruction four after the program has shifted the address left one bit in order to force the address to a halfword boundary. The table is assumed to be located on a 512 byte boundary. Its address is stored in the seven low-order bits of the high-order byte of the RB register. The two results are EXCLUSIVE ORed together in the fifth instruction and the table address is restored in the last instruction before the return branch. Table I shows the program which will update the CRC for a full eight-bit character.

This is called the one-table, one-step look-up method. It is very fast but may be impractical because of the quantity of core required.

Two-32-halfword-table look-up method

A more practical subroutine for CRC character update relative to core storage requirements is the two-table method. In this method, we further partition the matrix X above into two matrices Y and Z. Thus we

TABLE I—Subroutine Using One-256-Halfword Look-up

Initial conditions for all subroutines:
Register RA contains the old CRC, A_j
Register RB2 contains the new character, C_{j+1}.
Final conditions for all subroutines:
Register RA contains the new CRC, A_{j+1}.

∇ CRC1

EXCLUSIVE OR RB2, RA2	[1] RB[2;] ← RB[2;] ≠ RA[2;]	Form $W_{j+1}^{(L)}$
MOVE RB2, RA1	[2] RC[2;] ← RA[1;]	Form $W_{j+1}^{(H)}[0 \mid I]$
SHIFT LEFT RB, 1	[3] RB ← ((15ρ1), 0) ∧ 1φ(16ρRB)	Form address
LOAD RA, RB	[4] RA ← 2 8 ρ(16ρ2) ⊤ SA[2 ⊥ RB]	Load $W_{j+1}^{(L)}X$
EXCLUSIVE OR RA, RC	[5] RA[2;] ← RA[2;] ≠ RC[2;]	Form A_{j+1}
ROTATE LEFT RB, 15	[6] RB ← 2 8 ρ(15φRB)	Reset address
BRANCH RETURN	∇	Return

write G^8 as

$$G^8 = \left[\frac{0 \mid I}{Y/Z}\right]$$

Here, the Y and Z matrices are four by 16 binary matrices and $W_{j+1}^{(L)}$ is broken into two four-bit vectors $W_{j+1}^{(LL)}$ and $W_{j+1}^{(LH)}$. Thus, the new calculation becomes

$$A_{j+1} = W_{j+1}^{(LH)} \cdot Y \oplus W_{j+1}^{(LL)} \cdot Z \oplus W_{j+1}^{(H)} \cdot [0 \mid I].$$

Each of the products is a 16-bit row vector. The program now requires two look-up operations for the first two terms and a byte move for the last term. All three terms must then be EXCLUSIVE ORed together. The program is shown in Table II.

Binary summation method

Finally, it is possible to perform this whole operation without tables. This is done by performing the matrix multiplication by program rather than by table look-up. This requires a parity test as a condition on the branch instruction, however. This branching condition will be called PTYRC, the even parity of register RC. Looking back to the defining equation

$$A_{j+1} = [C_{j+1} + A_j] \cdot G^8 = W_{j+1} \cdot G^8.$$

Let $D_k = [d_{0,k}, d_{1,k}, \ldots, d_{15,k}]$ be the kth column of G^8. Then the high-order position of the new remainder A_{j+1} is given by

$$a_{0,j+1} = \sum_{i=0}^{15} d_{i,1} \cdot w_{i,j+1}$$

which is operationally the same as ANDing the first column of the matrix G^8 with the working remainder W_{j+1} and finding the even parity of the result. This parity is the value of $a_{0,j+1}$. Similarly, we can find the remaining bits by ANDing W_{j+1} with each column D_{k+1} and find the even parity to determine $a_{k,j+1}$ $0 \leq k \leq 15$.

$$a_{k,j+1} = \sum_{i=0}^{15} d_{i,k+1} \cdot w_{i,j+1}$$

This operation can be carried out in a program as illustrated in Table III.

The program shown here requires more than 80 words

TABLE II—Subroutine Using Two-32-Halfword Look-up

∇ CRC2

EXCLUSIVE OR RB2, RA2	[1] RB[2;]←RB[2;]≠RA[2;]	Form $W_{j+1}^{(L)}$
MOVE RA2, RB2	[2] RA[2;]←RB[2;]	Save $W_{j+1}^{(L)}$
AND RB2, H'FO'	[3] RB[2;]←RB[2;]∧ 1 1 1 1 0 0 0 0	Mask address
ROTATE LEFT RB2	[4] RB[2;]← φRB[2;]	Form address
LOAD RC, RB	[5] RC←2 8 ρ(16ρ2)⊤SA[2+2⊥16ρRB]	Load $W_{j+1}^{(LH)}Y$
EXCLUSIVE OR RC2, RA1	[6] RC[2;]←RC[2;]≠RA[1;]	$W_{j+1}^{(H)}[0\ I] \oplus W_{j+1}^{(LH)}Y$
MOVE RB2, RA2	[7] RB[2;]←RA[2;]	Get $W_{j+1}^{(L)}$
AND RB2, H'OF'	[8] RB[2;]←RB[2;]∧0 0 0 0 1 1 1 1	Form address
EXCLUSIVE OR RB2, H'10'	[9] RB[2;]←RB[2;]≠0 0 0 1 0 0 0 0	Form address
ROTATE LEFT RB, 1	[10] RB[2;]←1φRB[2;]	Form address
LOAD RA, RB	[11] RA←2 8 ρ(16ρ2)⊤SA[2+2⊥16ρRB]	Load $W_{j+1}^{(LL)}Z$
EXCLUSIVE OR RA, RC	[12] RA←RA≠RC	Form A_{j+1}
BRANCH RETURN	∇	Return

TABLE III—Subroutine for Binary Summation Method

```
                    ∇ CRC3
EXCLUSIVE OR RB, RA  [1]  RB←2 8 ρ(16ρRB)≠(16ρRA)      Form W_{j+1}
LOAD RA, ZERO        [2]  RA←2 8 ρ0                    Set A_{j+1} to zero
LOAD RC, D1          [3]  RC←SA[1;]                    Load D1
AND RC, RB           [4]  RC←2 8 ρRC∧(16ρRB)           Calculate D1W_{j+1}
BRANCH [7], PTRC     [5]  →(≠/(1, 16ρRC))/SECONDBIT    Branch if a_{0,j+1}=0
EXCLUSIVE OR RA, H'8000'
                     [6]  RA[1;]←RA[1;]≠1 0 0 0 0 0 0 0 Set a_{0,j+1}=1
LOAD RC, D2          [7]  SECONDBIT:RC←SA[2;]          Load D2
AND RC, RB           [8]  RC←2 8 ρRC∧(16ρRB)           Calculate D2W_{j+1}
BRANCH [11], PTRC    [9]  →(≠/(1, 16ρRC))/THIRDBIT     Branch if a_{1,j+1}=0
EXCLUSIVE OR RA, H'4000'
                     [10] RA[1;]←RA[1;]≠0 1 0 0 0 0 0 0 Set a_{1,j+1}=1
```

And so on for the third through the 15*th* bits.

```
LOAD RC, D16         [12] SIXTEENTHBIT:RC←SA[16;]      Load D16
AND RC, RB           [13] RC←2 8 ρRC∧(16ρRB)           Calculate D16W_{j+1}
BRANCH [16], PTRC    [14] →(≠/(1, 16ρRC))/OUT          Branch if a_{15,j+1}=0
EXCLUSIVE OR RA, H'0001'
                     [15] RA[2;]←RA[2;]≠0 0 0 0 0 0 0 1 Set a_{15,j+1}=1
BRANCH RETURN        [16] OUT:→0                       Return
```

of storage. However, a reduction in the storage requirement is possible by forming a loop to calculate the 16 binary sums. Further reduction is also possible when a specific polynomial is chosen and a combination of this and other schemes is used. For example, using $G(x) = x^{16}+x^{15}+x^2+1$, the number of instructions can be reduced to less than 20, making this method competitive with the other two given here. The key to this method is the branch instruction which tests the condition of the parity of the 16 bits in the accumulator. This is the last of the three matrix-oriented methods to be discussed and generally requires less core storage and more execution time than the previous two.

Other methods which partition the G^8 matrix in other ways are possible and may be better in specific cases.

SUMMARY

Using a matrix description of the operations required to generate the check bits in a cyclic redundancy error-detection scheme leads to new approaches to the software implementation problem. Certain variations are in use today and have proven to be superior to direct shift register simulation programs in most cases. With an apparent increase in programmable terminals and multiplexers, such approaches are likely to become even more important in the future.

REFERENCES

1 W L SCHILLER R L ABRAHAM R M FOX A VAN DAM
A microprogrammed intelligent graphics terminal
IEEE Transactions on Computers Vol C20 No 7 1971

2 A W MAHOLIC H H SCHWARZELL
Integrated microprogrammed communications control
Computer Design November 1969

3 *IBM 1130 synchronous communications adapter subroutine*
SRL File 1130-30 Form C26-3706-4 IBM Corporation White Plains New York

4 W W PETERSON
Error-correcting codes
The M.I.T. Press Cambridge Mass 1961

5 E R BERLEKAMP
Algebraic coding theory
McGraw-Hill Book Company New York 1968

6 I B OLDHAM R T CHIEN D T TANG
Error detection and correction in a photo-digital storage system
IBM Journal of Research and Development Vol 12 No 6 1968

7 R T CHIEN
Burst-correcting codes with high-speed decoding
IEEE Transactions on Information Theory Vol IT-15 No 1 January 1969

8 M Y HSIAO K Y SIH
Serial to parallel transformation of feedback shift register circuits
IEEE Transactions on Electronic Computers Vol EC-13 pp 738-740 December 1964

9 A M PATEL
A multi-channel CRC register
AFIPS Conference Proceedings Vol 38 pp 11-14 Spring 1971

10 K E IVERSON
A Programming Language
Wiley New York 1962

APPENDIX

Here, we will show how a shift register is used to perform the functions required to generate or verify a code word (calculate the proper h bits of redundancy). Then it can be shown that the operation of a shift

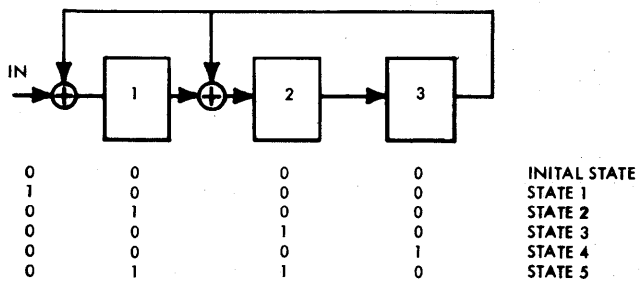

Figure A1—An elementary shift register

register on a bit-by-bit basis can be written in terms of matrix operations on vectors. Using this approach, it is possible to justify the several table look-up software schemes which are developed in the main text.

A feedback shift register is a device which stores bits in a serial string and is capable of shifting the string one bit at a time. There may be EXCLUSIVE OR and AND gates associated with the shift register which will operate when a shift takes place. The structure of a shift register is shown in Figure A1. The bit storage positions are indicated by a box (□) and the EXCLUSIVE OR gates are indicated by the "⊕." If the storage positions are denoted as shown, we can illustrate the operation by assuming that bit positions 1, 2, and 3 contain zero and that a one bit is placed on the "IN" lead. A single shift of the register by a clock pulse (not shown) will cause the "IN" to be EXCLUSIVE ORed with the feedback from position 3 and placed in position 1. Thus position $1=1(1\oplus 0=1)$. Now, let us assume that "IN" is set to zero and then another clock pulse occurs. Position $3\oplus$"IN"$=0$ is placed in position 1. Position $1\oplus$position $3(1\oplus 0=1)$ is placed in position 2.

A general shift register which performs division by

$$G(x)=1+g_1x+\cdots g_{h-1}x^{h-1}+x^h$$

is shown schematically in Figure A2. The AND gates are represented by the "⊙." Although the output does represent the quotient, of major interest to us is the

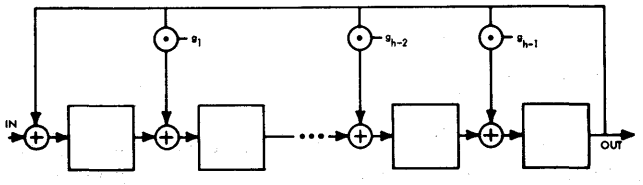

Figure A2—A general division shift register

contents of the shift register which is the h bit remainder

$$R(x)=r_0+r_1x+\cdots+r_{h-1}x^{h-1}$$

of the bits shifted in at any time. Thus, if we shift information bits

$$I(x)=i_0+i_1x+\cdots i_{k-1}x^{k-1}$$

into the shift register, highest degree coefficient first, we will have the remainder of $I(x)$ after all k bits have been entered. However, we would prefer to have the remainder of $x^hI(x)$ rather than the remainder of $I(x)$ so that we may append the remainder bits directly to the information. One way to do this would be to shift the shift register h times after $I(x)$ has been entered. However, this represents wasted time since we can wire the shift register differently in order to cause it to "pre-multiply" by x^h. This shift register is shown in Figure A3, and the remainder at time t will be denoted by the polynomial $A(x, t)$. After shifting $I(x)$ into this circuit, the remainder $R(x)$ of $x^hI(x)$ divided by $G(x)$ will be contained without further shifts; that is, $A(x, k)=R(x)$. If $R(x)$ is appended to $x^hI(x)$, a code word will be formed $(R(x)+x^hI(x))$. At the receiver, exactly the same circuit or program may be used to determine whether the received block is a code word.

In order to further illustrate the operation of the shift register, it is possible to develop a set of functional relationships between the bits that have entered the shift register and the contents of the register. These are the circuit equations for the shift register.

Let the bits in the shift register (Figure A3) at time t be represented by

$$a_0(t), a_1(t), a_2(t), \ldots, a_{h-1}(t)$$

where $a_0(t)$ is the leftmost bit in the shift register. We will also denote the bits which are shifted into the shift register as $b(t)$. That is, the contents of the shift register at time T include the effects of all $b(t)$ for $0<t\leq T$. Since the bits come at discrete times, both

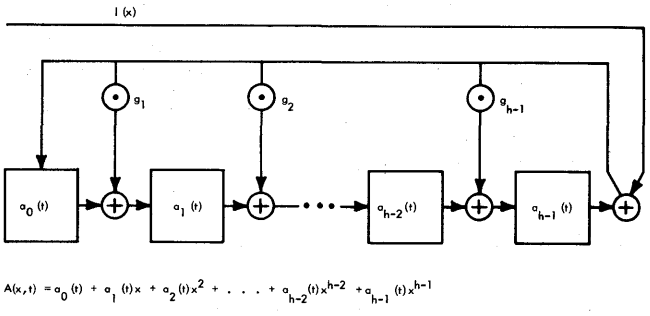

Figure A3—A shift register for pre-multiplication by x^h and division by $G(x)$

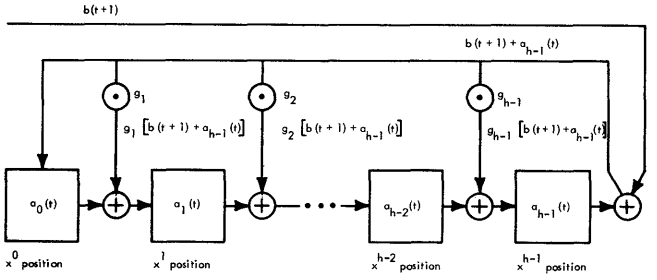

Figure A4—Development of circuit equations from the pre-multiply shift register

t and T are integers. Figure A4 may help the reader visualize this operation. From the figure, we can write the circuit equations directly.

$$\begin{aligned}
a_0(t+1) &= b(t+1) \oplus a_{h-1}(t) \\
a_1(t+1) &= a_0(t) \oplus g_1[b(t+1) \oplus a_{h-1}(t)] \\
a_2(t+1) &= a_1(t) \oplus g_2[b(t+1) \oplus a_{h-1}(t)] \\
&\vdots \\
a_{h-2}(t+1) &= a_{h-3}(t) \oplus g_{h-2}[b(t+1) \oplus a_{h-1}(t)] \\
a_{h-1}(t+1) &= a_{h-2}(t) \oplus g_{h-1}[b(t+1) \oplus a_{h-1}(t)]
\end{aligned} \tag{A1}$$

Since we set the register to zero before beginning to calculate the remainder, we have the initial conditions

$$a_0(0) = a_1(0) = a_2(0) = \cdots = a_{h-1}(0) = 0.$$

With these we can calculate any $a_i(T)$ given the $b(t)$ $(0 < t \leq T)$ and the generator polynomial

$$G(x) = 1 + g_1x + g_2x^2 + \cdots + g_{h-1}x^{h-1} + x^h.$$

These circuit equations will be used in the development of the matrix equations which are the subject of the main section.

In order to develop a matrix approach to the generation of a set of parity or check bits, we define a vector which consists of h binary components and represents the bits in the shift register at time t as defined above:

$$A(t) = [a_0(t), a_1(t), a_2(t), \ldots, a_{h-2}(t), a_{h-1}(t)].$$

Next, we define G to be the companion matrix of the polynomial $G(x)$ as shown in the main text.

From the circuit equations (A1), it is apparent that

$$\begin{aligned}
A(t+1) &= [a_0(t+1), a_1(t+1), a_2(t+1), \ldots, a_{h-1}(t+1)] \\
&= [0, a_0(t), a_1(t), \ldots, a_{h-2}(t)] \\
&\quad + [0, 0, \ldots, 0, b(t+1) \oplus a_{h-1}(t)] \cdot G.
\end{aligned}$$

Equation (A2) below follows immediately if one merely observes that

$$\begin{aligned}
&[a_0(t), a_1(t), \ldots, a_{h-2}(t), 0] \cdot G \\
&\qquad = [0, a_0(t), a_1(t), \ldots, a_{h-2}(t)]. \\
&A(t+1) = \{A(t) + [0, 0, \ldots, 0, b(t+1)]\} \cdot G.
\end{aligned} \tag{A2}$$

This is equation (1) of the main text.