1992

A hardware realization of the generalized euclidean algorithm for multisequence shift-register synthesis

Paul Vincent Kraft
Lehigh University

AUTHOR:

Kraft, Paul V.

TITLE:

A Hardware Realization of the Generalized Euclidean Algorithm for Multisequence Shift-Register Synthesis

DATE: October 11, 1992

A Hardware Realization of the Generalized Euclidean Algorithm for Multisequence Shift-Register Synthesis

by

Paul Vincent Kraft

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

August, 1992

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

August 27, 1992
(date)

Weiping Li
Thesis Advisor

Chairman of Department

Acknowledgements

I would like to thank both Dr. Kenneth K. Tzeng for my introduction to the field of Error Correction Coding and his inspiration for this project and Dr. Weiping Li for his continual support and advice and whose aid made this project possible. A special thanks to my parents for their eternal encouragement and to Ms. Sally Wengert for her amazing patience.

Table of Contents

List of Figures

Abstract

With the advent of the information age, the need to transmit and store data is of ever increasing importance. The possibility of data corruption is always prevalent due to noisy transmissions channels or storage medium. Much work has been done in the field of Error Correction Coding to minimize such errors without a dramatic reduction in the information rate. This paper will introduce an algorithm that will allow certain error correcting codes to utilize their correcting capabilities to a greater degree and suggest a possible method as to how this algorithm might be realized in a hardware format.

Chapter I - Introduction

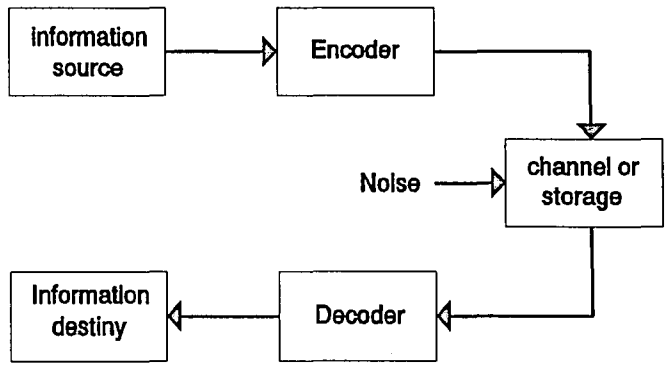


Figure 1 Data encoding/decoding system

When digital data is either transmitted or stored, there exists the possibility that errors might be introduced due to noisy channel or storage medium and corrupt the digitized information. Since the late forties, it was realized that it was possible to reduce the number of errors incurred without dramatically decreasing the information rate by encoding the original data before transmission or storage. A simple block diagram of such a system is illustrated in Figure 1.

A Bose-Chaudhuri-Hocquenghem (BCH) code [1] is a cyclic code where the generator polynomial, g(x), is chosen to be the minimum degree polynomial, having coefficients in GF(q). If α is a nonzero element of $GF(q^m)$, with m some positive integer, g(x) has roots α^m, $\alpha^{m+1}, \cdots, \alpha^{m+d-2}$, where d, $d \geq 2$, is any integer such that the d-1 roots are all distinct.

As Hartmann indicates [2], all cyclic codes can theoretically detect and correct up to $(d_m-1)/2$ errors, where d_m represents the minimum distance of the code. However, most

decoding methods do not allow correcting beyond $(d_0-1)/2$ errors, where d_0 represents the BCH bound and in most cases is less than the minimum distance of that code. Therefore, any such code is not realizing its error-correcting capabilities to the fullest.

Hartmann and Tzeng [3] point out that many cyclic codes are constructed utilizing a generator polynomial that has multiple sets of consecutive roots. In such a case, it is possible for multiple syndromes to be calculated for a particular received codeword. Utilizing these multiple syndromes allows error correction beyond that normally indicated by the BCH bound.

A generalization of the BCH bound was presented by Hartmann and Tzeng [4] in the case where multiple sets of consecutive roots are available. They presented a new, lower bound on the number of correctable errors that is based on the number of sets of consecutive roots and is greater than that of the normal BCH bound. For such a code, errors up to the Hartmann-Tzeng (HT) bound can be detected and corrected.

James Massey [5] demonstrated that the decoding problem for BCH codes is identical to the problem of synthesizing the shortest length linear feedback shift register (LFSR) that is capable of generating a given finite sequence. The shift-register synthesis algorithm given by Massey coincides with the iterative algorithm introduced by Berlekamp [6].

Recently, Feng and Tzeng [7] introduced an algorithm that

would be capable of taking advantage of the ability to detect errors beyond that of the normal BCH bound, up to the HT bound. In the process of constructing their algorithm, Feng and Tzeng presented a generalized form of the polynomial division algorithm and a generalized Euclidean algorithm. Their algorithm would be capable of determining the shortest length LFSR that was capable of generating multiple sets of defined sequences.

Just as Massey illustrated that the process for determining the shortest length shift register that can generate a single finite sequence is identical to that for solving BCH codes with a single syndrome, the algorithm presented by Feng and Tzeng for the generation of a shortest length LFSR capable of generating multiple sequences is identical to that of decoding cyclic codes with multiple syndromes sequences up to the HT bound. When their algorithm is used in this respect, if the number of errors in the received vector is guaranteed to be detectable and correctable by the HT bound, then the result represents the error-locator polynomial that corresponds to the given syndromes.

Since the HT bound is normally greater than the BCH bound of a given code, utilizing Feng and Tzeng's Generalized Euclidean algorithm would be attractive in order to take advantage of the increased error correcting capabilities previously untapped by other methods.

The purpose of this work is to explore the feasibility of

a hardware implementation of the Generalized Euclidean algorithm. A hardware version might offer speed enhancements which would be desirable in areas such as real-time decoding of coded incoming data.

The problem of finding the shortest length LFSR can be described as such: Let $S_0^{(h)}$, $S_1^{(h)}$, $\cdots$, $S_{n-1}^{(h)}$ for $h=0,1,\cdots,m-1$ denote m sequences, each of length n, over a Field F. Let $U(z)= \delta_0+\delta_1 z+\cdots+\delta_{l-1}z^{l-1}+z^l$ be a monic polynomial over F. If $S_i^{(h)}+ \delta_{l-1}S_{i-1}^{(h)}+\cdots+\delta_1 S_{i-l+1}^{(h)}+\delta_0 S_{i-l}^{(h)}=0$, for $i=l,l+1,\cdots,n-1$, and $h=0,1,\cdots,m-1$, then U(z) completely specifies all connections in an LFSR of length l that generates each of the above m sequences, with initial loading $S_0^{(h)}$, $S_1^{(h)},\cdots,$ $S_{l-1}^{(h)}$, illustrated in Figure 2.

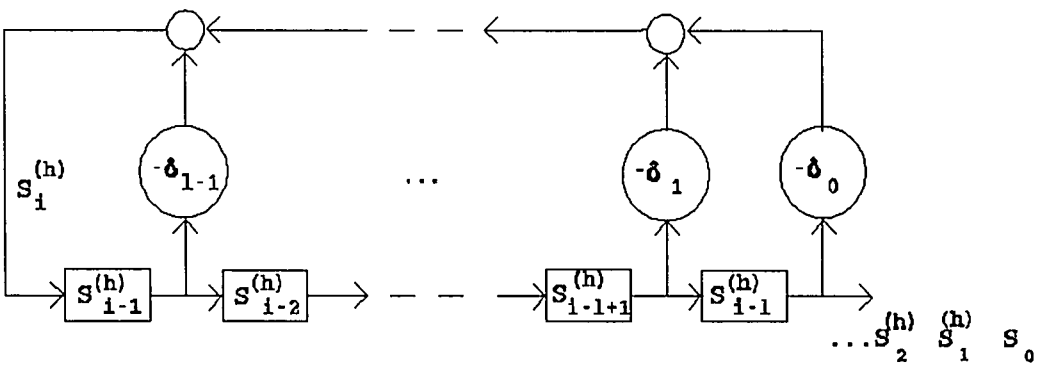


Figure 2. Linear feedback shift register of length l specified by U(z).

In the remainder of the paper, Chapter II will present the Generalized Euclidean algorithm in detail, Chapter III

will present a design overview, and Chapter IV presents design aspects in greater detail. The complete listing of all software and circuit files can be found in the appendices.

Chapter II - Shift-Register Synthesis Algorithm

The algorithm as presented by Gui-Liang Feng and Kenneth K. Tzeng accepts a list of multiple sequences and produces a mathematical representation of the shortest length LFSR that is capable of generating those sequences. The algorithm first starts by generating initial polynomials that are dependant on the number of sequences and their length, represented by *m* and *n*, respectively. It then proceeds to perform polynomial divisions based on their presented *Generalized Division Algorithm*. The results from these divisions are then utilized to perform polynomial multiplications, the results of which will become the final solution upon termination of the algorithm.

If Feng and Tzeng's Euclidean algorithm is utilized to calculate an error-locator polynomial, then the parameters, *m* and *n*, are determined by the specific code used. Specifically, *m* is determined by the number of sets of consecutive roots in the generator polynomial and *n* is determined by the number of minimal polynomials used to create the generator polynomial.

Feng and Tzeng's algorithm is as follows:

Generalized Euclidian Algorithm for Multisequence Shift-Register Synthesis

Step 1: (Initialization) Let

$$r_0=\sum_{h=0}^{m-1} z^{m-1-h} \sum_{i=0}^{n-1} S_i^{(h)} z^{m(n-1-i)}, \qquad (1)$$

$$U_0(z)=1,\ b_0^{(h)}(z)=z^{mn+h},\ V_0^{(h)}(z)=0$$

for all h=0,1,...,m-1

j=0.

Step 2: (Indexing) j=j+1.

Step 3: (Division) Calculate $r_j(z)$, $p_j(z^m)$, and $q_j^{(h)}(z^m)$ from $r_{j-1}(z)$ and $b_{j-1}^{(h)}(z)$ so that

$$r_j(z)=p_j(z^m)\,r_{j-1}(z)+\sum_{h=0}^{m-1} q_j^{(h)}(z^m)\,b_{j-1}^{(h)}(z).$$

Let $v_{j-1}=\deg r_{j-1}(z) \bmod m$, $b_j^{(v_{j-1})}(z)=r_{j-1}(z)$ and $b_j^{(h)}(z)=b_{j-1}^{(h)}(z)$ for all $h\neq v_{j-1}$.

Step 4: (Multiplication) Find $U_j(z)$ from $U_{j-1}(z)$ and $V_{j-1}^{(h)}(z)$ so that

$$U_j(z)=p_j(z)\,U_{j-1}(z)+\sum_{h=0}^{m-1} q_j^{(h)}(z)\,V_{j-1}^{(h)}(z). \qquad (3)$$

Let

$$V_j^{(v_{j-1})}(z)=U_{j-1}(z) \quad \text{and} \quad V_j^{(h)}(z)=V_{j-1}^{(h)}(z), \qquad (4)$$

for all $h \neq v_{j-1}$.

Step 5: (Test of Completion) If deg $r_j(z) \geq$ deg $U_j(z^m)$, go back to step 2. Otherwise, go to step 6.

Step 6: (Algorithm results) Let k=j. Then $\delta U_k(z)$ is a shortest length LFSR. Where δ is a constant such that $\delta U_k(z)$ is monic. Stop.

Included with the algorithm are conditions under which the algorithm result is guaranteed to be unique. If:

$$l_k \leq g_k^{(h)}, \qquad \text{for all } h=0,1,\cdots,m-1 \tag{5}$$

where l_k= deg $U_k(z)$ and $m \cdot g_k^{(h)}$=(deg $b_k^{(h)}(z)$)-h, then $\delta U(z)$ is the unique shortest length LFSR.

Example 1: Let $\{S_i^{(0)}\}^7_{i=0}$={10101101} and $\{S_i^{(1)}\}^7_{i=0}$ ={01010100} be two binary sequences of length 8. Thus m=2, n=8, h=0,1. Following the preceding algorithm, we have the following.

(Step 1): (Initialization)

$r_0 = z^{15}+z^{12}+z^{11}+z^8+z^7+z^5+z^4+z$. $U_0(z)=1$, $b_0^{(0)}(z)=z^{16}$, $b_0^{(1)}(z)=z^{17}$, $V_0^{(0)}(z)=0$, $V_0^{(1)}(z)=0$, j=0.

(Step 2): (Indexing) j=1.

(Step 3): (Division)

$r_1(z) = z^2 r_0(z)+0 \cdot b_0^{(0)}(z)+1 \cdot b_0^{(1)}(z) = z^{14}+z^{13}+z^{10}+z^9+z^7+z^6+z^3$, $p_1(z^2)= z^2$, $q_1^{(0)}(z^2)= 0$, $q_1^{(1)}(z^2)= 1$. Since v_0= 1, then $b_1^{(1)}(z)= r_0(z)= z^{15}+z^{12}+z^{11}+z^8+z^7+z^5+z^4+z$, $b_1^{(0)}(z)= b_0^{(0)}(z)= z^{16}$.

(Step 4): (Multiplication)

$U_1(z)= z \cdot U_0(z)+0 \cdot V_0^{(0)}(z)+1 \cdot V_0^{(1)}(z)= z$. Then $V_1^{(1)}(z)= U_0(z)= 1$, $V_1^{(0)}(z)= V_0^{(0)}(z)= 0$.

(Step 5): (Test of Completion)

Since deg $r_1(z)= 14 \geq 2 =$ deg $U_1(z^2)$, go to step 2.

(Step 2): (Indexing) j=2.

(Step 3): (Division)

$r_2(z) = z^2 \cdot r_1(z)+1 \cdot b_1^{(0)}(z)+1 \cdot b_1^{(1)}(z) = z^9+z^7+z^4+z$, $p_2(z^2)= z^2$, $q_2^{(0)}(z^2)= 1$, $q_2^{(1)}(z^2)= 1$. Since $v_1= 0$, then $b_2^{(0)}(z)= r_1(z)= z^{14}+z^{13}+z^{10}+z^9+z^7+z^6+z^3$, $b_2^{(1)}(z)= b_1^{(1)}(z)= z^{15}+z^{12}+ z^{11}+z^8+z^7+z^5+z^4+z$.

(Step 4): (Multiplication)

$U_2(z)= z \cdot U_1(z)+1 \cdot V_1^{(0)}(z)+1 \cdot V_1^{(1)}(z)= z^2+1$. Then $V_2^{(0)}(z)=U_1(z)= z$, $V_2^{(1)}(z)= V_1^{(1)}(z)= 1$.

(Step 5): (Test of Completion)

Since deg $r_2(z) = 9 \geq 4 =$ deg $U_2(z^2)$, go to step 2.

(Step 2): (Indexing) j=3.

(Step 3): (Division)

$r_3(z) = (z^6+z^4) \cdot r_2(z)+0 \cdot b_2^{(0)}+1 \cdot b_2^{(1)} = z^{12}+z^{10}+z^4+z$, $p_3(z^2)= z^6+z^4$, $q_3^{(0)}(z^2)= 0$, $q_3^{(1)}(z^2)= 1$. Since $v_2= 1$, then $b_3^{(1)}(z)= r_2(z)= z^9+z^7+z^4+z$, $b_3^{(0)}(z)= b_2^{(0)}(z)= z^{14}+z^{13}+z^{10}+z^9+z^7+z^6+ z^3$.

(Step 4): (Multiplication)

$U_3(z) = (z^3+z^2) \cdot U_2(z)+0 \cdot V_2^{(0)}(z)+1 \cdot V_2^{(1)}(z) = z^5+z^4+z^3+z^2+1$. Then $V_3^{(1)}(z)= U_2(z)= z^2+1$, $V_3^{(0)}(z)= V_2^{(0)}(z)= z$.

(Step 5): (Test of Completion)

Since deg $r_3(z)= 12 \geq 10=$ deg $U_3(z^2)$, go to step 2.

(Step 2): (Indexing) j=4.

(Step 3): (Division)

$r_4(z) = (z^2+1) \cdot r_3(z)+1 \cdot b_3^{(0)}(z)+(z^4+z^2) \cdot b_3^{(1)}(z) = z^8+z^7+z^6+z^5+z^4+z^3+z$, $p_4(z^2)= z^2+1$, $q_4^{(0)}(z^2)= 1$, $q_4^{(1)}(z^2)= z^4+z^2$. Since $v_3= 0$, then $b_4^{(0)}(z)= r_3(z)= z^{12}+z^{10}+z^4+z$, $b_4^{(1)}(z)=$

$b_3^{(1)}(z) = z^9+z^7+z^4+z$.

(Step 4): (Multiplication)

$U_4(z) = (z+1)\cdot U_3(z)+1\cdot V_3^{(0)}(z)+(z^2+z)\cdot V_3^{(1)}(z) = z^6+z^4+z^3+z+1$.

Then $V_4^{(0)}(z) = U_3(z) = z^5+z^4+z^3+z^2+1$, $V_4^{(1)}(z) = V_3^{(1)}(z) = z^2+1$.

(Step 5): (Test of Completion)

Since deg $r_4(z) = 8 < 12 =$ deg $U_4(z^2)$, go to step 6.

(Step 6): (Algorithm Results)

$k=4$, $U_4(z) = z^6+z^4+z^3+z+1$ is a shortest length LFSR which generates $\{S_i^{(0)}\}^7_{i=0}=\{10101101\}$ and $\{S_i^{(1)}\}7_{i=0}=\{01010100\}$. Stop.

Chapter III - Design Overview

As mentioned previously, the Generalized Euclidean Algorithm accepts multiple sets of sequences and parameters, *m* and *n*, which represent the number of sets and the sequence length, respectively. The result of the algorithm is a single sequence. If the system design is viewed as a single functional block, we have what is illustrated in Figure 3.

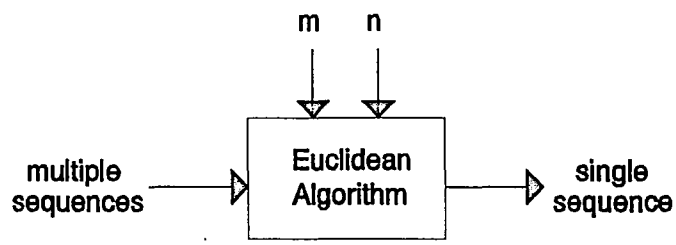


Figure 3

Breaking the above figure into smaller, more descriptive function blocks, the algorithm can be designed as shown in Figure 4. After initial polynomials are constructed, polynomial divisions are conducted. Using results from the divisions, polynomial multiplications are conducted until the termination point is indicated by results of both stages.

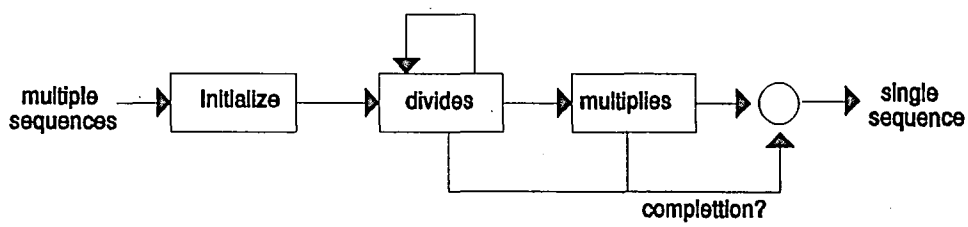


Figure 4

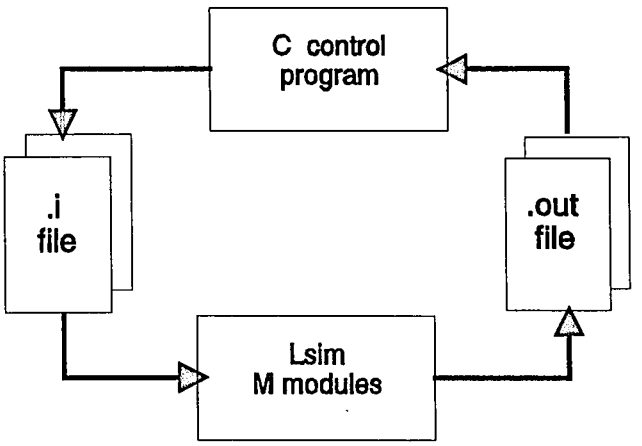


Figure 5 Overall system design

Unlike most processes that are realized in VLSI, in which data flow usually proceeds in a simple, linear direction, the generalized Euclidean algorithm presented in this paper has complex, circular data flow with an abundance of condition testing.

Therefore, it was decided that the overall process of the algorithm would be implemented most efficiently by dividing it into two sections: data flow control and data processing.

Due to its complexity, data flow control and decision making would remain as software. For the purpose of this paper, control was implemented as a C program, launched under the Unix operating system.

Data processing, such as polynomial multiplication or division, would be implemented as hardware circuitry, being simulated with Mentor Graphic's Lsim simulator and the M language.

Starting the algorithm, the control program first reads from a file the number of syndromes available, *m*, their nlength, *n*, and the syndrome values. From this, initial polynomials are constructed and testing occurs to determine the next course of action. The overall process can be illustrated as in Figure 5. The control program reads information from a file and, from this information, creates a another file consisting of a listing of Lsim commands. The control program then launches the Lsim simulator which, based on the file created by the control program, produces results that are written to an output file. These results are, in turn, read by the control program and the cycle continues until the control program determines that specific conditions are met.

For the purpose of this project, it was decided to consider sequences of binary value only. This implies that all polynomials will have coefficients from GF(2) and any cyclic code for which this design is intended would also be restricted to coefficients from GF(2).

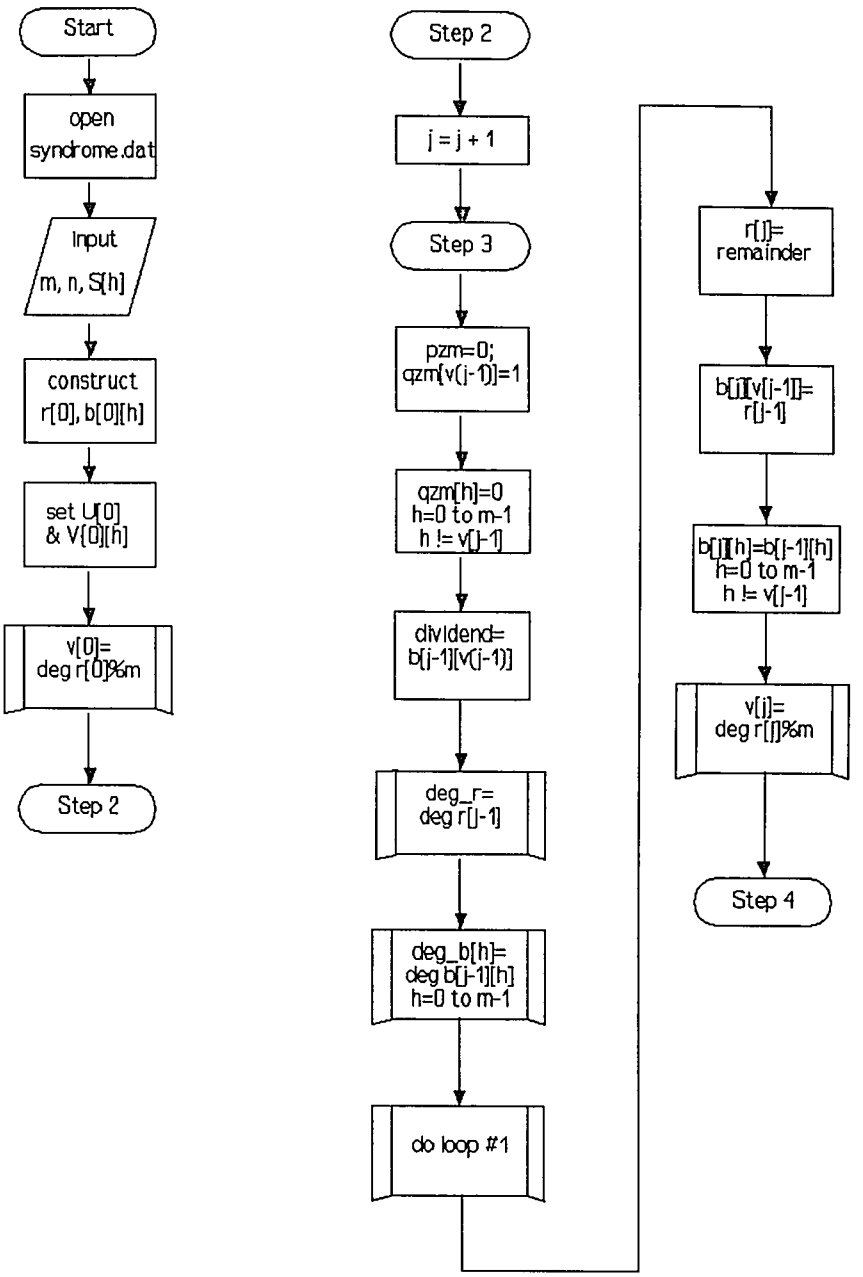
The actual design structure of both the software control and hardware simulation remains as a function of the parameters *m* and *n*, in order that simulations can be conducted with multiple values. It should be noted, however, that within the C software control, each polynomial is represented by an Unsigned Long Integer, with bit position *i* representing the polynomial coefficient of the variable with

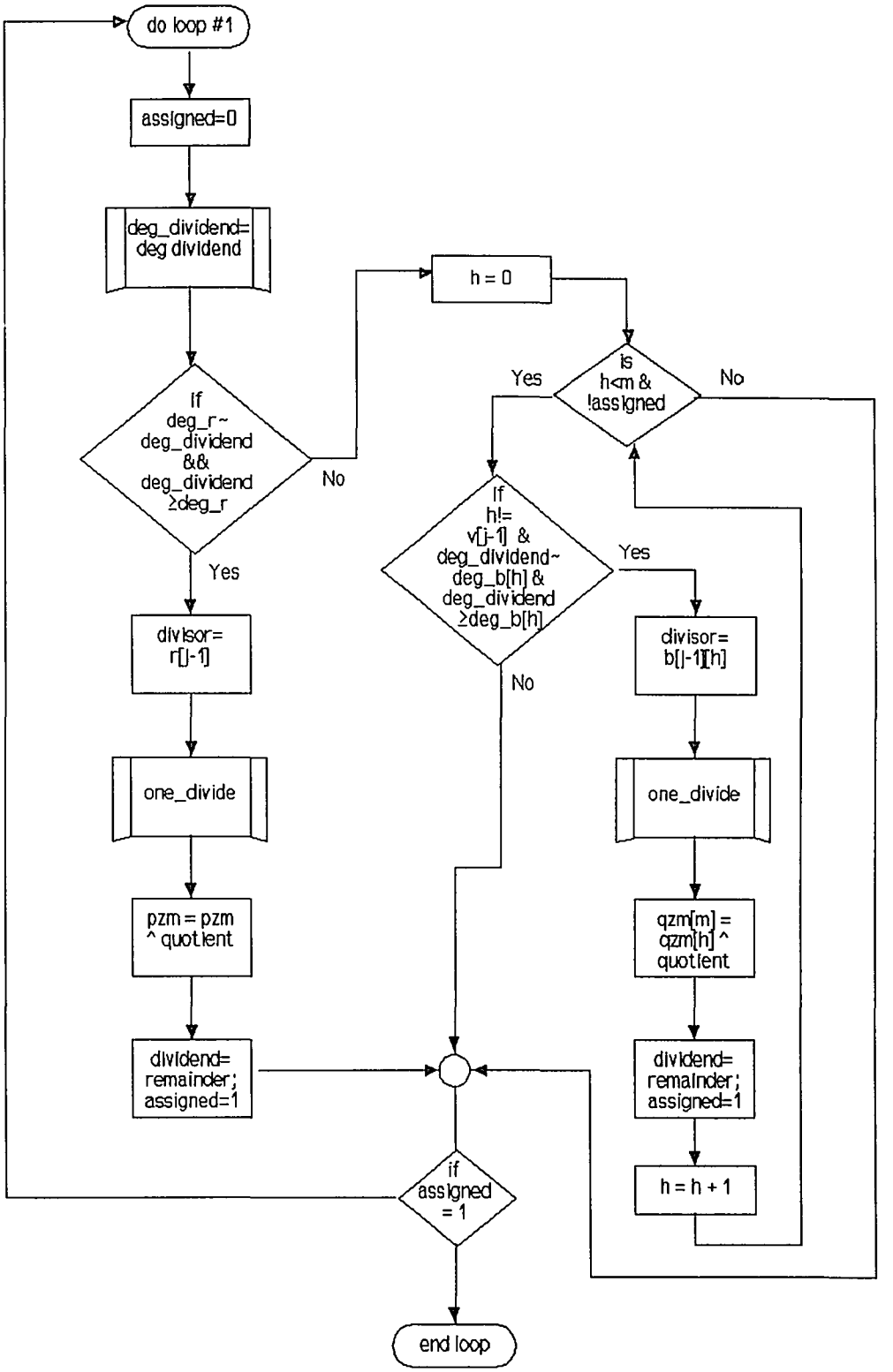
degree i. The consequences of this is that, due to the intrinsic nature of the algorithm, the value $m \cdot n+m$, which will be the highest degree polynomial, must remain less than or equal to 32, the bit size for Unsigned Long Integers within C.

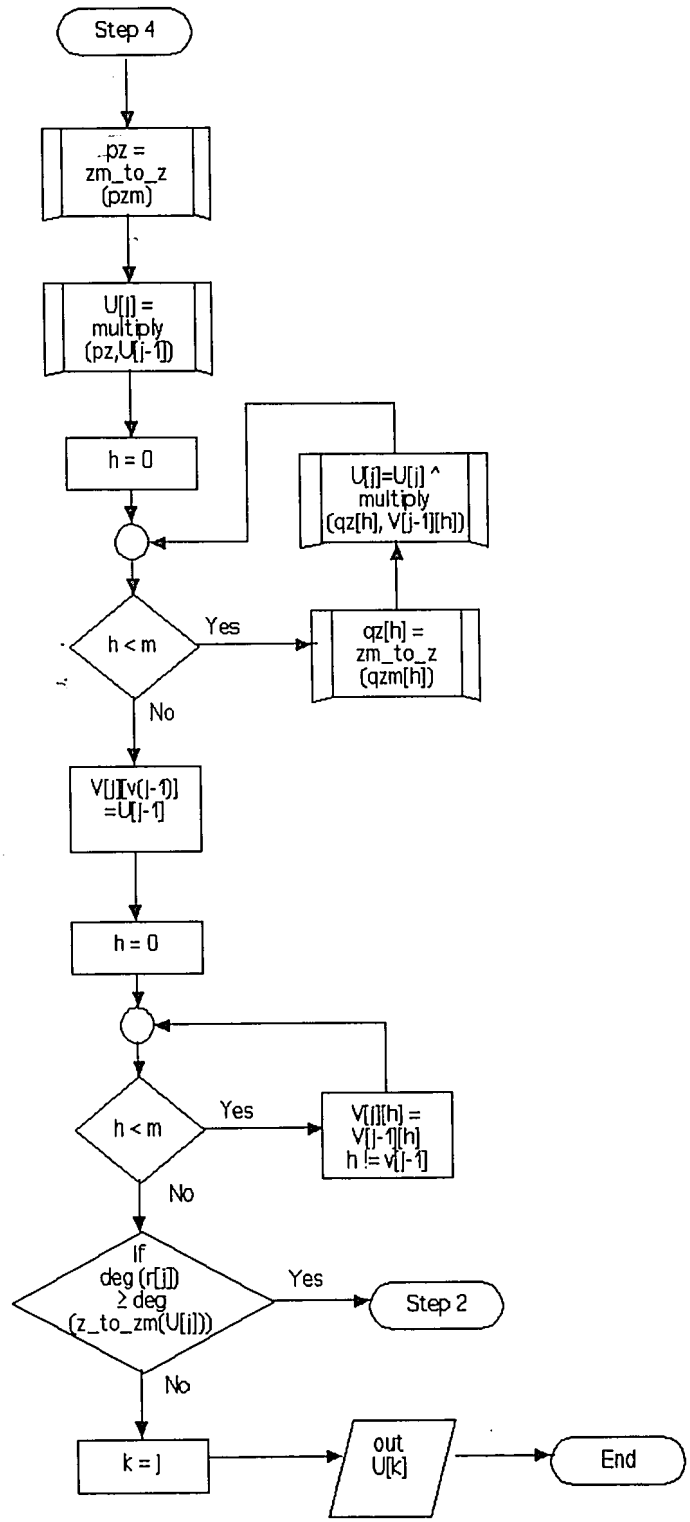
It should also be noted, that since all hardware circuitry is constructed as a function of m and n, the size and complexity of each Lsim module increases linearly with m or n.

Complete listings of the C control program and all M language circuit models can be found in Appendix A and Appendix B, respectively.

The following charts outline the processing flow of the control program:




do loop #1
assigned=0
deg_dividend=
deg dividend
If
deg_r~
deg_dividend
&&
deg_dividend
≥deg_r
No
Yes
divisor=
r[j-1]
one_divide
pzm = pzm
^ quotient
dividend=
remainder;
assigned=1
h = 0
is
h<m &
!assigned
Yes
No
If
h!=
v[j-1] &
deg_dividend~
deg_b[h] &
deg_dividend
≥deg_b[h]
Yes
No
divisor=
b[j-1][h]
one_divide
qzm[m] =
qzm[h] ^
quotient
dividend=
remainder;
assigned=1
h = h + 1
if
assigned
= 1
end loop


Step 4
pz = zm_to_z (pzm)
U[j] = multiply (pz,U[j-1])
h = 0
U[j]=U[j] ^ multiply (qz[h], V[j-1][h])
h < m
Yes
qz[h] = zm_to_z (qzm[h])
No
V[j][v(j-1)] =U[j-1]
h = 0
h < m
Yes
V[j][h] = V[j-1][h] h != v[j-1]
No
If deg (r[j]) ≥ deg (z_to_zm(U[j]))
Yes
Step 2
No
k = j
out U[k]
End

The majority of the M circuit models are very straightforward and thus require little explanation. One exception if the following circuit that accepts two polynomials and their relative degrees and computes the Remainder and Quotient result of a single long division step.

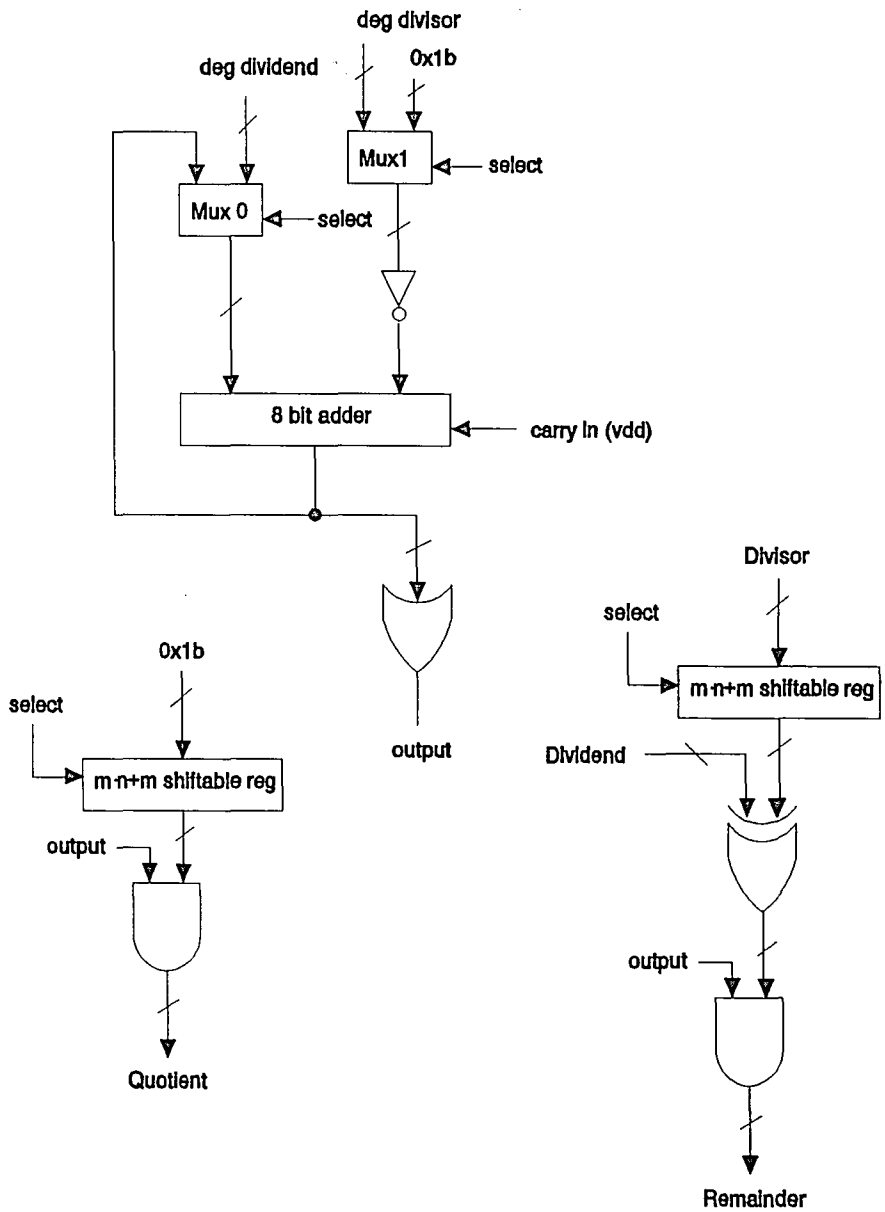


Figure 6 One step divider circuit

Chapter V - Results and Conclusions

Appendix C gives the complete listing of a particular simulation. For this case, the parameters were three sequences of length eight. After five steps, the algorithm terminated. The program also indicated that the result was unique. Therefore, if this simulation was conducted such that the initial three sequences represented syndrome sequences, the resulting polynomial would be the error locator polynomial, whose roots are the error positions in the received codeword, and the number of errors is guaranteed to be recoverable by the HT bound.

Throughout the design process, it became apparent that the Generalized Euclidean algorithm was of such complextity that full implementation into hardware was not a feasible option. Thus, all descision making was delegated into a software section that directed the flow of data through various hardware modules.

In a full realization, if speed is a concern, the software control could easily take the from of a dedicated ROM, which, when linked with the data processing modules, would complete the algorithm.

The software and circuiut modules presented here could also easily be adapted to operate with sequence coefficients of a non-binary nature, such as members of $GF(2^m)$, with only minor adjustments.

Appendix A - Control Program Listing

```
#include <stdio.h>
#include <math.h>

void pause();
unsigned long multiply();
unsigned long zm_to_z();
unsigned long z_to_zm();

main()
{
     FILE *fp1, *fp_syn;
     char *S,ch[4];
     int m,n,h,i,j,k,deg_r,*deg_b,v[20],assigned,
          deg_divisor, deg_dividend;
     int lk, *gk;
     unsigned long int r[20], *b[20], *V[20], U[20],
          dividend, divisor, quotient, remainder, *qzm, *qz,
          pz, pzm;
     if((fp_syn=fopen("syndrome3.dat","r"))==NULL) {
          puts("cannot open file\n");
          exit(1);
          }
     fscanf(fp_syn,"%d %d\n", &m, &n);
     printf("m=%d, n=%d\n", m,n);
     S=(char *)malloc(m*n*sizeof(char));
     for(i=0; i<=m-1; i++) {
          for(j=0; j<=n-1; j++) fscanf(fp_syn,"%c",
                    S+(i*n)+j);
               fscanf(fp_syn,"\n");
               }
          for(i=0; i<=m-1; i++) {
               printf("S[%d]=",i);
               for(j=0; j<=n-1; j++) printf("%c",
                    *(S+(i*n)+j));
               printf("\n");
               }
     k=m*n;
     for(j=0; j<=n-1; j++) {
          for(i=0; i<=m-1; i++) {
               k--;
               if(*(S+(i*n)+j)=='1') {
               printf(" +X^%d",k);
               r[0]=r[0]+(unsigned long)pow(2.0, (double)k);
                    }
               }
          }
     printf("\n");
     printf("r[0]= %lx\n", r[0]);
     disp_poly(r[0]);
```

```
qzm=(unsigned long *)malloc(m*sizeof(long));
if(!qzm){
     printf("not enough memory for qzm\n");
     exit(1);
     }
qz=(unsigned long *)malloc(m*sizeof(long));
if(!qz){
     printf("not enough memory for qz\n");
     exit(1);
     }
for(i=0; i<20; i++) {
     b[i]=(unsigned long *)malloc(m*sizeof(long));
     if(!b[i]){
          printf("not enough memory for b[%d]\n",i);
          exit(1);
          }
     }
for(i=0; i<20; i++) {
     V[i]=(unsigned long *)malloc(m*sizeof(long));
     if(!V[i]){
          printf("not enough memory for V[%d]\n",i);
          exit(1);
          }
     }
for(h=0; h<=m-1; h++) {
     *(b[0]+h)=(unsigned long)pow(2.0,(double)(m*n+h));
     printf("b[0](%d) = %lx\n",h,*(b[0]+h));
     *(V[0]+h)=0;
     }
v[0]=deg(r[0],m,n)%m;
printf("v[0] = %d\n",v[0]);
U[0]=1;
fclose(fp_syn);

deg_b=(int *)malloc(m*sizeof(int));
if(!deg_b){
     printf("not enough memory for deg_b\n");
     exit(1);
     }
gk=(int *)malloc(m*sizeof(int));
if(!gk) {
     printf("not enough memory for gk\n");
     exit(1);
     }

j=0;
do {
     /* Step 2 */
     j++; printf("j = %d\n",j);

     /* Step 3 */
     pzm=0;
```

```
*(qzm+v[j-1])=1;
for(h=0; h<m; h++) {
     if(h != v[j-1]) *(qzm+h)=0;
     }
dividend=*(b[j-1]+v[j-1]);
deg_r=deg(r[j-1],m,n);
printf("deg r[%d] = %d\n",j-1,deg_r);
for(h=0; h<m; h++){
     *(deg_b+h)=deg(*(b[j-1]+h),m,n);
     printf("deg b[%d](%d) = %d\n",j-1,h,
          *(deg_b+h));
     }
do{
     assigned=0;
     deg_dividend=deg(dividend,m,n);
     if(congruent(deg_dividend,deg_r,m) &&
               deg_dividend>=deg_r) {
          printf("dividend ~ r[%d]\n",j-1);
               divisor=r[j-1];
          deg_divisor=deg(divisor,m,n);

          puts("dividend = ");
          disp_poly(dividend);
          puts("divisor = ");
          disp_poly(divisor);

     one_divide(dividend,divisor,&quotient,
     &remainder,deg_dividend,deg_divisor,m,n);
          pzm=pzm ^ quotient;
          dividend=remainder;
          assigned=1;

          puts("quotient = ");
          disp_poly(quotient);
          puts("remainder = ");
          disp_poly(remainder);
          }
     else {
          for(h=0; h<m && !assigned; h++) {
          if(h!=v[j-1] && congruent(deg_dividend,
          *(deg_b+h),m) && deg_dividend >=
                         *(deg_b+h) ) {

          printf("dividend ~ b[%d](%d)\n",j-1,h);
          divisor=*(b[j-1]+h);
          deg_divisor=deg(divisor,m,n);

          puts("dividend = ");
          disp_poly(dividend);
          puts("divisor = ");
          disp_poly(divisor);
```

```
     one_divide(dividend,divisor,&quotient,
     &remainder,deg_dividend,deg_divisor,m,n);

          puts("quotient = ");
          disp_poly(quotient);
          puts("remainder = ");
          disp_poly(remainder);

          *(qzm+h)=*(qzm+h) ^ quotient;
          dividend=remainder;
          assigned=1;
          }       /* end if   */
          }              /* end for  */
     }                      /* end else */
} while(assigned);
puts("done dividing\n");
r[j]=remainder;
printf("r[%d] = ",j);
disp_poly(r[j]);
*(b[j]+v[j-1])=r[j-1];
printf("b[%d](%d) = ",j,v[j-1]);
disp_poly(*(b[j]+v[j-1]));
for(h=0; h<m; h++) {
     if(h != v[j-1]) {
          *(b[j]+h)=*(b[j-1]+h);
          printf("b[%d](%d) = ",j,h);
          disp_poly(*(b[j]+h));
          }
     }
v[j]=deg(r[j],m,n)%m;
printf("v[%d] = %d\n",j,v[j]);
printf("pzm = ",j);
disp_poly(pzm);
printf("qzm[%d] =",v[j-1]);
disp_poly(*(qzm+v[j-1]));
for(h=0; h<m; h++) {
     if(h != v[j-1]) {
          printf("qzm[%d] =",h);
          disp_poly(*(qzm+h));
          }
     }

/* Step 4 */
pz = zm_to_z(pzm,m,n);
U[j] = multiply(pz,U[j-1],m,n);
for(h=0; h<m; h++) {
     *(qz+h) = zm_to_z( *(qzm+h),m,n);
     U[j]=U[j]^multiply(*(qz+h), *(V[j-1]+h),m,n);
     }
*(V[j]+v[j-1])=U[j-1];
for(h=0; h<m; h++) {
     if (h != v[j-1]) {
```

```
                    *(V[j]+h)=*(V[j-1]+h);
                    }
               }
          printf("U[%d] = ",j);
          disp_poly(U[j]);
          for(h=0; h<m ;h++) {
               printf("V[%d](%d) = ",j,h);
               disp_poly(*(V[j]+h));
               }
     } while(deg(r[j],m,n)>=deg( z_to_zm(U[j],m,n),m,n));

     /* Step 6 */
     lk=deg(U[j],m,n);
     for(h=0; h<m; h++) {
          *(gk+h)=(deg( *(b[j]+h),m,n)-h)/m;
          }

     printf("k= %d\n",j);
     printf("Shortest LFSR = U[%d] = ",j);
     disp_poly(U[j]);
     printf("lk= %d\n",lk);
     for(h=0; h<m; h++) printf("gk[%d]= %d   ",h,*(gk+h));
     puts("\n");
}

deg(p,m,n)
unsigned long p;
int m,n;
{
        FILE *fp;
        char ch[4],str[40],temp[8];
        int i,deg;
        if(p==0) return 0;
        else{
        if((fp=fopen("deg2.i","w"))==NULL){
                puts("cannot open file deg2.i\n");
                exit(1);
                }
        fprintf(fp,"bus p_in[%d:0] x\n",m*n+m-1);
        fprintf(fp,"bus out[7:0] d\n");
        fprintf(fp,"order out[7:0] p_in[%d:0]\n",m*n+m-1);
        fprintf(fp,"empty out[7:0]\n");
        fprintf(fp,"empty p_in[%d:0]\n",m*n+m-1);
        fprintf(fp,"Delete 5\n");
        fprintf(fp,"0x%lx p_in[%d:0]\ns\n",p,m*n+m-1);
        fprintf(fp,"Print deg2.out\nquit\n");
        fclose(fp);
        strcpy(str,"Lsim -t scmos -f deg2,");
        strcat(str,sprintf(temp,"%d,%d .",m,n));

        puts(str);
        system(str);
```

```
        if((fp=fopen("deg2.out","r"))==NULL){
                puts("cannot open file deg2.out\n");
                exit(1);
                }
        do {
                for(i=2; i>=0; i--) ch[i+1]=ch[i];
                ch[0]=getc(fp);
        } while
       (!(ch[3]=='0'&&ch[2]=='.'&&ch[1]=='0'&&ch[0]=='0'));
        fscanf(fp,"%d",&deg);
        fclose(fp);
        return deg;
        }
}

congruent(x,y,m)
int x,y,m;
{
        if(x%m==y%m) return 1;
        else return 0;
}

one_divide(dividend,divisor,quotient,remainder,deg_dividend,
          deg_divisor,m,n)
int deg_dividend, deg_divisor, m,n;
unsigned long int dividend, divisor, *quotient, *remainder;
{
        FILE *fp;
        char ch[5], str[40], temp[8], limit[8];
        int i;
        if(deg_divisor>deg_dividend) {
                *quotient=0;
                *remainder=dividend;
                }
        else{


sprintf(limit,"%d.00",4*(deg_dividend-deg_divisor+1));
        if((fp=fopen("one_div2.i","w"))==NULL) {
                puts("cannot open file one_div2.i\n");
                exit(1);
                }
        fprintf(fp,"bus dividend[%d:0] x\n",m*n+m-1);
        fprintf(fp,"bus divisor[%d:0] x\n",m*n+m-1);
        fprintf(fp,"bus deg_dividend[7:0] d\n");
        fprintf(fp,"bus deg_divisor[7:0] d\n");
        fprintf(fp,"bus quotient[%d:0] x\n",m*n+m-1);
        fprintf(fp,"bus remainder[%d:0] x\n",m*n+m-1);
        fprintf(fp,"order quotient[%d:0] remainder[%d:0] ",
               m*n+m-1,m*n+m-1);
        fprintf(fp,"dividend[%d:0] divisor[%d:0]\n",
               m*n+m-1,m*n+m-1);
```

```
        fprintf(fp,"empty quotient[%d:0]\n",m*n+m-1);
        fprintf(fp,"empty remainder[%d:0]\n",m*n+m-1);
        fprintf(fp,"empty dividend[%d:0]\n",m*n+m-1);
        fprintf(fp,"empty divisor[%d:0]\n",m*n+m-1);
        fprintf(fp,"Delete 9\n");
        fprintf(fp,"alias clk \"1 phi1;1 phi2;s;h phi1;s;1
               phi1;s;h phi2;s;\"\n");
        fprintf(fp,"0x%lx dividend[%d:0]\n",
              dividend,m*n+m-1);
        fprintf(fp,"0x%lx divisor[%d:0]\n",divisor,m*n+m-1);
        fprintf(fp,"0d%d deg_dividend[7:0]\n",deg_dividend);
        fprintf(fp,"0d%d deg_divisor[7:0]\n",deg_divisor);
        fprintf(fp,"h select\nclk\n1 select\n");
        for(i=0; i<(deg_dividend-deg_divisor); i++) {
                fprintf(fp,"clk\n");
                }
        fprintf(fp,"Print one_div2.out\nquit\n");
        fclose(fp);
        strcpy(str,"Lsim -f one_div2,");
        strcat(str,sprintf(temp,"%d,%d .",m,n));

        puts(str);
        system(str);

        if((fp=fopen("one_div2.out","r"))==NULL) {
                puts("cannot open file one_div2.out\n");
                exit(1);
                }
        if(4*(deg_dividend-deg_divisor+1)>=10) {

        do{
                for(i=3; i>=0; i--) ch[i+1]=ch[i];
                ch[0]=getc(fp);
        } while (!(ch[4]==limit[0] && ch[3]==limit[1] &&
          ch[2]==limit[2] && ch[1]==limit[3] &&
          ch[0]==limit[4]));
        }
        else {

        do{
                for(i=2; i>=0; i--) ch[i+1]=ch[i];
                ch[0]=getc(fp);
        } while (!(ch[3]==limit[0] && ch[2]==limit[1] &&
          ch[1]==limit[2] && ch[0]==limit[3]));
        }
        fscanf(fp,"%lx %lx",quotient,remainder);
        fclose(fp);
        }
}

disp_poly(x)
unsigned long int x;
```

```
{
        unsigned long int test_bit=(unsigned long)
                                        pow(2.0,31.0);
        int i,flag=0;
        for(i=31; i>=0; i--) {
                if(x & test_bit) {
                        if(i==0) printf(" +1");
                        else if(i==1) printf(" +z");
                        else printf(" +z^%d",i);
                        flag=1;
                        }
                test_bit = test_bit >> 1;
                }
        if(!flag) puts(" 0");
        printf("\n");
}


unsigned long multiply(p1,p2,m,n)
unsigned long p1,p2;
int m,n;
{
        FILE *fp;
        int i,max=m*n+m-1;
        char ch[4], str[40], temp[8];
        unsigned long int p_out;

        if((fp=fopen("multiply2.i","w"))==NULL) {
                puts("cannot open file multiply.i\n");
                exit(1);
                }
        fprintf(fp,"bus p1_in[%d:0] x\n",max);
        fprintf(fp,"bus p2_in[%d:0] x\n",max);
        fprintf(fp,"bus p_out[%d:0] x\n",max);
        fprintf(fp,"order p_out[%d:0] p1_in[%d:0]
               p2_in[%d:0]\n", max,max,max);
        fprintf(fp,"empty p_out[%d:0]\n",max);
        fprintf(fp,"empty p1_in[%d:0]\n",max);
        fprintf(fp,"empty p2_in[%d:0]\n",max);
        fprintf(fp,"Delete 7\n");
        fprintf(fp,"0x%lx p1_in[%d:0]\n",p1,max);
        fprintf(fp,"0x%lx p2_in[%d:0]\n",p2,max);
        fprintf(fp,"s\nPrint multiply2.out\nquit\n");
        fclose(fp);
        strcpy(str,"Lsim -f multiply2,");
        strcat(str,sprintf(temp,"%d,%d .",m,n));

        puts(str);
        system(str);

        if((fp=fopen("multiply2.out","r"))==NULL) {
                puts("cannot open file multiply2.out\n");
```

```
                exit(1);
                }
        do{
                for(i=2; i>=0; i--) ch[i+1]=ch[i];
                ch[0]=getc(fp);
        } while
        (!(ch[3]=='0'&&ch[2]=='.'&&ch[1]=='0'&&ch[0]=='0'));
        fscanf(fp,"%lx",&p_out);
        fclose(fp);
        return p_out;
}


unsigned long zm_to_z(p,m,n)
unsigned long p;
int m,n;
{
        FILE *fp;
        int i,max=m*n+m-1;
        char ch[4], str[40], temp[8];
        unsigned long int p_out;

        if((fp=fopen("zm_to_z_2.i","w"))==NULL) {
                puts("cannot open file zm_to_z_2.i\n");
                exit(1);
                }
        fprintf(fp,"bus p_in[%d:0] x\n",max);
        fprintf(fp,"bus p_out[%d:0] x\n",max);
        fprintf(fp,"order p_out[%d:0] p_in[%d:0]\n",
              max,max);
        fprintf(fp,"empty p_out[%d:0]\n",max);
        fprintf(fp,"empty p_in[%d:0]\n",max);
        fprintf(fp,"Delete 5\n");
        fprintf(fp,"0x%lx p_in[%d:0]\n",p,max);
        fprintf(fp,"s\nPrint zm_to_z_2.out\nquit\n");
        fclose(fp);
        strcpy(str,"Lsim -f zm_to_z_2,");
        strcat(str,sprintf(temp,"%d,%d .",m,n));

        puts(str);
        system(str);

        if((fp=fopen("zm_to_z_2.out","r"))==NULL) {
                puts("cannot open file zm_to_z_2.out\n");
                exit(1);
                }
        do{
                for(i=2; i>=0; i--) ch[i+1]=ch[i];
                ch[0]=getc(fp);
        } while
        (!(ch[3]=='0'&&ch[2]=='.'&&ch[1]=='0'&&ch[0]=='0'));
        fscanf(fp,"%lx",&p_out);
```

```
        fclose(fp);
        return p_out;
}

unsigned long z_to_zm(p,m,n)
unsigned long p;
int m,n;
{
        FILE *fp;
        int i,max=m*n+m-1;
        char ch[4], str[40], temp[8];
        unsigned long int p_out;

        if((fp=fopen("z_to_zm_2.i","w"))==NULL) {
                puts("cannot open file z_to_zm_2.i\n");
                exit(1);
                }
        fprintf(fp,"bus p_in[%d:0] x\n",max);
        fprintf(fp,"bus p_out[%d:0] x\n",max);
        fprintf(fp,"order p_out[%d:0] p_in[%d:0]\n",
                    max,max);
        fprintf(fp,"empty p_out[%d:0]\n",max);
        fprintf(fp,"empty p_in[%d:0]\n",max);
        fprintf(fp,"Delete 5\n");
        fprintf(fp,"0x%lx p_in[%d:0]\n",p,max);
        fprintf(fp,"s\nPrint z_to_zm_2.out\nquit\n");
        fclose(fp);
        strcpy(str,"Lsim -f z_to_zm_2,");
        strcat(str,sprintf(temp,"%d,%d .",m,n));

        puts(str);
        system(str);

        if((fp=fopen("z_to_zm_2.out","r"))==NULL) {
                puts("cannot open file z_to_zm_2.out\n");
                exit(1);
                }
        do{
                for(i=2; i>=0; i--) ch[i+1]=ch[i];
                ch[0]=getc(fp);
        } while
        (!(ch[3]=='0'&&ch[2]=='.'&&ch[1]=='0'&&ch[0]=='0'));
        fscanf(fp,"%lx",&p_out);
        fclose(fp);
        return p_out;
}
```

Appendix B - Lsim Module Listings

```
MODULE deg2(m,n)
int m,n;
{
IN LOGIC p_in[m*n+m];
OUT LOGIC out[8];
MEMORY LOGIC max_deg[8];
VDD vdd;
GND gnd;

IN LOGIC dummy1[8];

BUILD {
     int i, j, max_val, current;
     max_deg=m*n+m-1;
     max_val=max_deg;
     current=max_deg;
     for(i=0; i<max_val; i++) {
          for(j=0; j<max_val-i; j++) {
               INSTANCE(and, and[i][j], 2);
               INSTANCE(inverter, inv[i][j]);
               NET(inv[i][j].out, and[i][j].in_bus[1]);
               }
          NET(p_in[i], and[i][0].in_bus[0] );
          }
     for(i=0; i<max_val; i++) {
          NET(p_in[max_val], inv[i][0].in_bus[0] );
          }
     for(i=0; i<max_val-1; i++) {
          for(j=1; j<max_val-i; j++) {
               NET(and[max_val-j][j-1].out,
               inv[i][j].in_bus[0]);
               NET(and[i][j-1].out, and[i][j].in_bus[0]);
               }
          }
     for(i=0; i<8; i++) {
          INSTANCE(sinv, sinv[i]);
          NET(sinv[i].out, dummy1[i]);
          NET(dummy1[i], out[i]);
          }
     for(i=max_val; i>=0; i--) {
          for(j=0; j<8; j++) {
               INSTANCE(n_trans, n_trans[i][j],600, 200,
                -1, -1);
               NET(n_trans[i][j].d, sinv[j].in);
               }
          }
     for(j=0; j<8; j++) {
          NET(n_trans[max_val][j].g, p_in[max_val]);
          if(max_deg[j]==1) NET(n_trans[max_val][j].s, gnd);
```

```
            else NET(n_trans[max_val][j].s, vdd);
            }
      current--;
      max_deg=current;
      for(i=max_val-1; i>=0; i--) {
            for(j=0; j<8; j++) {
                  NET(n_trans[i][j].g,
                  and[i][max_val-1-i].out);
                  if(max_deg[j]==1) NET(n_trans[i][j].s, gnd);
                  else NET(n_trans[i][j].s, vdd);
                  }
            current--;
            if(current>=0) max_deg=current;
            }
      }
}



MODULE multiply2(m,n)
int m,n;
{
IN LOGIC p1_in[m*n+m];
IN LOGIC p2_in[m*n+m];
OUT LOGIC p_out[m*n+m];
MEMORY LOGIC max_deg[8];

IN LOGIC dummy1[m*n+m];

BUILD {
      int i,j, max=m*n+m-1;
      for(i=0; i<=max; i++) {
            for(j=0; j<=max-i; j++) {
                  INSTANCE(and, and[i][j], 2);
                  NET(p1_in[i], and[i][j].in_bus[0]);
                  NET(p2_in[j], and[i][j].in_bus[1]);
                  }
            }
      for(i=0; i<=max; i++) {
            INSTANCE(xor, xor[i], i+1);
            }
      for(i=0; i<=max; i++) {
            for(j=0; j<=max-i; j++) {
                  NET(and[i][j].out, xor[i+j].in_bus[i]);
                  }
            }
      for(i=0; i<=max; i++) {
            NET(xor[i].out, dummy1[i]);
            NET(dummy1[i], p_out[i]);
            }
      }
}
```

```
MODULE z_to_zm_2(m,n)
int m,n;
{
IN LOGIC p_in[m*n+m];
OUT LOGIC p_out[m*n+m];

VDD vdd;
GND gnd;

BUILD {
        int i, max=m*n+m-1;
        for(i=0; i<=max; i++) {
                if(i%m==0) NET(p_out[i], p_in[i/m]);
                else NET(p_out[i], gnd);
                }
        }
}
```

```
MODULE zm_to_z_2(m,n)
int m,n;
{
IN LOGIC p_in[m*n+m];
OUT LOGIC p_out[m*n+m];

VDD vdd;
GND gnd;

BUILD {
        int i, max=m*n+m-1;
        for(i=0; i<=max; i++) {
                if(m*i<=max) NET(p_out[i], p_in[m*i]);
                else NET(p_out[i], gnd);
                }
        }
}
```

```
MODULE one_div2(m,n)
int m,n;
{
IN LOGIC dividend[m*n+m];
IN LOGIC divisor[m*n+m];
IN LOGIC deg_dividend[8];
IN LOGIC deg_divisor[8];
```

```
IN LOGIC select;
IN LOGIC phi1;
IN LOGIC phi2;

OUT LOGIC quotient[m*n+m];
OUT LOGIC remainder[m*n+m];

VDD vdd;
GND gnd;

BUILD {
        int i,j;
        INSTANCE(nor, output, 8);
        for(i=0; i<8; i++) {
                INSTANCE(a221mux, mux0[i]);
                INSTANCE(a221mux, mux1[i]);
                INSTANCE(inverter, inv[i]);
                INSTANCE(bit_add, bit_add[i]);
                INSTANCE(reg, reg[i]);

                NET(deg_dividend[i], mux0[i].in0);
                NET(deg_divisor[i], mux1[i].in0);
                NET(reg[i].out, mux0[i].in1);
                if(i==0) NET(vdd, mux1[i].in1);
                else NET(gnd, mux1[i].in1);
                NET(mux0[i].select, select);
                NET(mux1[i].select, select);
                NET(mux1[i].out, inv[i].in_bus[0]);
                NET(mux0[i].out, bit_add[i].a);
                NET(inv[i].out, bit_add[i].b);
                if(i==0) NET(bit_add[i].carryin, vdd);
                else NET(bit_add[i].carryin,
                         bit_add[i-1].carry);
                NET(bit_add[i].sum, reg[i].in);
                NET(phi1, reg[i].phi1);
                NET(phi2, reg[i].phi2);

                NET(reg[i].out, output.in_bus[i]);
                }
        NET(bit_add[7].carry, gnd);

        for(i=0; i<m*n+m; i++) {
                INSTANCE(a2in_reg, quotient_reg[i]);
                INSTANCE(and, qand[i], 2);
                NET(select, quotient_reg[i].select);
                if(i==0) {
                        NET(quotient_reg[i].in1, vdd);
                        NET(quotient_reg[i].in2, gnd);
                        }
                else {
                        NET(quotient_reg[i].in1, gnd);
                        NET(quotient_reg[i].in2,
```

```
                        quotient_reg[i-1].out);
                }
            NET(phi1, quotient_reg[i].phi1);
            NET(phi2, quotient_reg[i].phi2);
            NET(quotient_reg[i].out, qand[i].in_bus[0]);
            NET(output.out, qand[i].in_bus[1]);
            NET(qand[i].out, quotient[i]);

            INSTANCE(a2in_reg, remain_reg[i]);
            INSTANCE(xor, rxor[i], 2);
            INSTANCE(and, rand[i], 2);
            NET(select, remain_reg[i].select);
            if(i==0) {
                    NET(remain_reg[i].in1, divisor[i]);
                    NET(remain_reg[i].in2, gnd);
                    }
            else {
                    NET(remain_reg[i].in1, divisor[i]);
                    NET(remain_reg[i].in2,
                     remain_reg[i-1].out);
                    }
            NET(phi1, remain_reg[i].phi1);
            NET(phi2, remain_reg[i].phi2);
            NET(remain_reg[i].out, rxor[i].in_bus[0]);
            NET(dividend[i], rxor[i].in_bus[1]);
            NET(rxor[i].out, rand[i].in_bus[0]);
            NET(output.out, rand[i].in_bus[1]);
            NET(rand[i].out, remainder[i]);
            }
        }
}
```

Appendix C - Simulation Output

Note: Comments are added for clarity and are written in italic form.

```
a.out

(Step 1 - Initialization)
m=3, n=8
S[0]=10101101
S[1]=01010100
S[2]=11100110
r[0]= aea9cc=
 +z^23 +z^21 +z^19 +z^18 +z^17 +z^15 +z^13 +z^11 +z^8 +z^7
+z^6 +z^3 +z^2
b[0](0) = 1000000
b[0](1) = 2000000
b[0](2) = 4000000
Lsim -t scmos -f deg2,3,8 .
(Most Lsim calls are accompanied by parameters m=3, n=8)

v[0] = 2

(Step 2 - Indexing)
j = 1

(Step 3 - Division)
Lsim -t scmos -f deg2,3,8 .

deg r[0] = 23
Lsim -t scmos -f deg2,3,8 .

deg b[0](0) = 24
Lsim -t scmos -f deg2,3,8 .

deg b[0](1) = 25
Lsim -t scmos -f deg2,3,8 .

deg b[0](2) = 26
Lsim -t scmos -f deg2,3,8 .

dividend ~ r[0]
Lsim -t scmos -f deg2,3,8 .

dividend =
 +z^26
divisor =
 +z^23 +z^21 +z^19 +z^18 +z^17 +z^15 +z^13 +z^11 +z^8 +z^7
+z^6 +z^3 +z^2
Lsim -f one_div2,3,8 .

quotient =
```

```
 +z^3

remainder =
 +z^24 +z^22 +z^21 +z^20 +z^18 +z^16 +z^14 +z^11 +z^10 +z^9
+z^6 +z^5
Lsim -t scmos -f deg2,3,8 .

dividend ~ b[0](0)
Lsim -t scmos -f deg2,3,8 .

dividend =
 +z^24 +z^22 +z^21 +z^20 +z^18 +z^16 +z^14 +z^11 +z^10 +z^9
+z^6 +z^5
divisor =
 +z^24
Lsim -f one_div2,3,8 .

quotient =
 +1
remainder =
 +z^22 +z^21 +z^20 +z^18 +z^16 +z^14 +z^11 +z^10 +z^9 +z^6
+z^5
Lsim -t scmos -f deg2,3,8 .

done dividing

r[1] =  +z^22 +z^21 +z^20 +z^18 +z^16 +z^14 +z^11 +z^10 +z^9
+z^6 +z^5
b[1](2) =  +z^23 +z^21 +z^19 +z^18 +z^17 +z^15 +z^13 +z^11
+z^8 +z^7 +z^6 +z^3 +z^2
b[1](0) =  +z^24
b[1](1) =  +z^25
Lsim -t scmos -f deg2,3,8 .

v[1] = 1
pzm =  +z^3
qzm[2] = +1
qzm[0] = +1
qzm[1] = 0

(Step 4 - Multiplication)
Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .
```

```
Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

U[1] =  +z
V[1](0) =  0

V[1](1) =  0

V[1](2) =  +1

(Step 5 - Test of Completion)
Lsim -f z_to_zm_2,3,8 .

Lsim -t scmos -f deg2,3,8 .

Lsim -t scmos -f deg2,3,8 .

(Step 2 - Indexing)
j = 2

(Step 3 - Division)
Lsim -t scmos -f deg2,3,8 .

deg r[1] = 22
Lsim -t scmos -f deg2,3,8 .

deg b[1](0) = 24
Lsim -t scmos -f deg2,3,8 .

deg b[1](1) = 25
Lsim -t scmos -f deg2,3,8 .

deg b[1](2) = 23
Lsim -t scmos -f deg2,3,8 .

dividend ~ r[1]
Lsim -t scmos -f deg2,3,8 .

dividend =
 +z^25
divisor =
 +z^22 +z^21 +z^20 +z^18 +z^16 +z^14 +z^11 +z^10 +z^9 +z^6
+z^5
Lsim -f one_div2,3,8 .

quotient =
 +z^3
remainder =
 +z^24 +z^23 +z^21 +z^19 +z^17 +z^14 +z^13 +z^12 +z^9 +z^8
Lsim -t scmos -f deg2,3,8 .
```

```
dividend ~ b[1](0)
Lsim -t scmos -f deg2,3,8 .

dividend =
 +z^24 +z^23 +z^21 +z^19 +z^17 +z^14 +z^13 +z^12 +z^9 +z^8

divisor =
 +z^24
Lsim -f one_div2,3,8 .

quotient =
 +1
remainder =
 +z^23 +z^21 +z^19 +z^17 +z^14 +z^13 +z^12 +z^9 +z^8
Lsim -t scmos -f deg2,3,8 .

dividend ~ b[1](2)
Lsim -t scmos -f deg2,3,8 .

dividend =
 +z^23 +z^21 +z^19 +z^17 +z^14 +z^13 +z^12 +z^9 +z^8
divisor =
 +z^23 +z^21 +z^19 +z^18 +z^17 +z^15 +z^13 +z^11 +z^8 +z^7
+z^6 +z^3 +z^2
Lsim -f one_div2,3,8 .

quotient =
 +1
remainder =
 +z^18 +z^15 +z^14 +z^12 +z^11 +z^9 +z^7 +z^6 +z^3 +z^2
Lsim -t scmos -f deg2,3,8 .

done dividing

r[2] =  +z^18 +z^15 +z^14 +z^12 +z^11 +z^9 +z^7 +z^6 +z^3
+z^2
b[2](1) =  +z^22 +z^21 +z^20 +z^18 +z^16 +z^14 +z^11 +z^10
+z^9 +z^6 +z^5
b[2](0) =  +z^24
b[2](2) =  +z^23 +z^21 +z^19 +z^18 +z^17 +z^15 +z^13 +z^11
+z^8 +z^7 +z^6 +z^3 +z^2
Lsim -t scmos -f deg2,3,8 .

v[2] = 0
pzm =  +z^3
qzm[1] = +1
qzm[0] = +1
qzm[2] = +1

(Step 4 - Multiplication)
Lsim -f zm_to_z_2,3,8 .
```

```
Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

U[2] =  +z^2 +1
V[2](0) =  0

V[2](1) =  +z
V[2](2) =  +1

(Step 5 - Test of Completion)
Lsim -f z_to_zm_2,3,8 .

Lsim -t scmos -f deg2,3,8 .

Lsim -t scmos -f deg2,3,8 .

(Step 2 - Indexing)
j = 3

(Step 3 - Division)
Lsim -t scmos -f deg2,3,8 .

deg r[2] = 18
Lsim -t scmos -f deg2,3,8 .

deg b[2](0) = 24
Lsim -t scmos -f deg2,3,8 .

deg b[2](1) = 22
Lsim -t scmos -f deg2,3,8 .

deg b[2](2) = 23
Lsim -t scmos -f deg2,3,8 .

dividend ~ r[2]
Lsim -t scmos -f deg2,3,8 .

dividend =
 +z^24
divisor =
 +z^18 +z^15 +z^14 +z^12 +z^11 +z^9 +z^7 +z^6 +z^3 +z^2
```

```
Lsim -f one_div2,3,8 .

quotient =
 +z^6
remainder =
 +z^21 +z^20 +z^18 +z^17 +z^15 +z^13 +z^12 +z^9 +z^8
Lsim -t scmos -f deg2,3,8 .

dividend ~ r[2]
Lsim -t scmos -f deg2,3,8 .
dividend =
 +z^21 +z^20 +z^18 +z^17 +z^15 +z^13 +z^12 +z^9 +z^8
divisor =
 +z^18 +z^15 +z^14 +z^12 +z^11 +z^9 +z^7 +z^6 +z^3 +z^2
Lsim -f one_div2,3,8 .

quotient =
 +z^3
remainder =
 +z^20 +z^14 +z^13 +z^10 +z^8 +z^6 +z^5
Lsim -t scmos -f deg2,3,8 .

done dividing

r[3] =  +z^20 +z^14 +z^13 +z^10 +z^8 +z^6 +z^5
b[3](0) =  +z^18 +z^15 +z^14 +z^12 +z^11 +z^9 +z^7 +z^6 +z^3
+z^2
b[3](1) =  +z^22 +z^21 +z^20 +z^18 +z^16 +z^14 +z^11 +z^10
+z^9 +z^6 +z^5
b[3](2) =  +z^23 +z^21 +z^19 +z^18 +z^17 +z^15 +z^13 +z^11
+z^8 +z^7 +z^6 +z^3 +z^2
Lsim -t scmos -f deg2,3,8 .

v[3] = 2
pzm =  +z^6 +z^3
qzm[0] = +1
qzm[1] = 0
qzm[2] = 0

(Step 4 - Multiplication)
Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .
```

```
Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

U[3] =  +z^4 +z^3 +z^2 +z
V[3](0) =  +z^2 +1
V[3](1) =  +z
V[3](2) =  +1

(Step 5 - Test of Completion)
Lsim -f z_to_zm_2,3,8 .

Lsim -t scmos -f deg2,3,8 .

Lsim -t scmos -f deg2,3,8 .

(Step 2 - Indexing)
j = 4

(Step 3 - Division)
Lsim -t scmos -f deg2,3,8 .

deg r[3] = 20
Lsim -t scmos -f deg2,3,8 .

deg b[3](0) = 18
Lsim -t scmos -f deg2,3,8 .

deg b[3](1) = 22
Lsim -t scmos -f deg2,3,8 .

deg b[3](2) = 23
Lsim -t scmos -f deg2,3,8 .

dividend ~ r[3]
Lsim -t scmos -f deg2,3,8 .

dividend =
 +z^23 +z^21 +z^19 +z^18 +z^17 +z^15 +z^13 +z^11 +z^8 +z^7
+z^6 +z^3 +z^2
divisor =
 +z^20 +z^14 +z^13 +z^10 +z^8 +z^6 +z^5
Lsim -f one_div2,3,8 .

quotient =
 +z^3
remainder =
 +z^21 +z^19 +z^18 +z^16 +z^15 +z^9 +z^7 +z^6 +z^3 +z^2
Lsim -t scmos -f deg2,3,8 .

dividend ~ b[3](0)
Lsim -t scmos -f deg2,3,8 .
```

```
dividend =
 +z^21 +z^19 +z^18 +z^16 +z^15 +z^9 +z^7 +z^6 +z^3 +z^2
divisor =
 +z^18 +z^15 +z^14 +z^12 +z^11 +z^9 +z^7 +z^6 +z^3 +z^2
Lsim -f one_div2,3,8 .

quotient =
 +z^3
remainder =
 +z^19 +z^17 +z^16 +z^14 +z^12 +z^10 +z^7 +z^5 +z^3 +z^2
Lsim -t scmos -f deg2,3,8 .

done dividing

r[4] =  +z^19 +z^17 +z^16 +z^14 +z^12 +z^10 +z^7 +z^5 +z^3
+z^2
b[4](2) =  +z^20 +z^14 +z^13 +z^10 +z^8 +z^6 +z^5
b[4](0) =  +z^18 +z^15 +z^14 +z^12 +z^11 +z^9 +z^7 +z^6 +z^3
+z^2
b[4](1) =  +z^22 +z^21 +z^20 +z^18 +z^16 +z^14 +z^11 +z^10
+z^9 +z^6 +z^5
Lsim -t scmos -f deg2,3,8 .

v[4] = 1
pzm =  +z^3
qzm[2] = +1
qzm[0] = +z^3
qzm[1] = 0

(Step 4 - Multiplication)
Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

U[4] =  +z^5 +z^4 +z^2 +z +1
V[4](0) =  +z^2 +1
V[4](1) =  +z
V[4](2) =  +z^4 +z^3 +z^2 +z

(Step 5 - Test of Completion)
```

```
Lsim -f z_to_zm_2,3,8 .

Lsim -t scmos -f deg2,3,8 .

Lsim -t scmos -f deg2,3,8 .

(Step 2 - Indexing)
j = 5

(Step 3 - Division)
Lsim -t scmos -f deg2,3,8 .

deg r[4] = 19
Lsim -t scmos -f deg2,3,8 .

deg b[4](0) = 18
Lsim -t scmos -f deg2,3,8 .

deg b[4](1) = 22
Lsim -t scmos -f deg2,3,8 .

deg b[4](2) = 20
Lsim -t scmos -f deg2,3,8 .

dividend ~ r[4]
Lsim -t scmos -f deg2,3,8 .

dividend =
 +z^22 +z^21 +z^20 +z^18 +z^16 +z^14 +z^11 +z^10 +z^9 +z^6
+z^5
divisor =
 +z^19 +z^17 +z^16 +z^14 +z^12 +z^10 +z^7 +z^5 +z^3 +z^2
Lsim -f one_div2,3,8 .

quotient =
 +z^3
remainder =
 +z^21 +z^19 +z^18 +z^17 +z^16 +z^15 +z^14 +z^13 +z^11 +z^9
+z^8
Lsim -t scmos -f deg2,3,8 .

dividend ~ b[4](0)
Lsim -t scmos -f deg2,3,8 .

dividend =
 +z^21 +z^19 +z^18 +z^17 +z^16 +z^15 +z^14 +z^13 +z^11 +z^9
+z^8
divisor =
 +z^18 +z^15 +z^14 +z^12 +z^11 +z^9 +z^7 +z^6 +z^3 +z^2
Lsim -f one_div2,3,8 .

quotient =
```

```
 +z^3
remainder =
 +z^19 +z^16 +z^13 +z^12 +z^11 +z^10 +z^8 +z^6 +z^5
Lsim -t scmos -f deg2,3,8 .

dividend ~ r[4]
Lsim -t scmos -f deg2,3,8 .

dividend =
 +z^19 +z^16 +z^13 +z^12 +z^11 +z^10 +z^8 +z^6 +z^5
divisor =
 +z^19 +z^17 +z^16 +z^14 +z^12 +z^10 +z^7 +z^5 +z^3 +z^2
Lsim -f one_div2,3,8 .

quotient =
 +1
remainder =
 +z^17 +z^14 +z^13 +z^11 +z^8 +z^7 +z^6 +z^3 +z^2
Lsim -t scmos -f deg2,3,8 .

done dividing

r[5] =  +z^17 +z^14 +z^13 +z^11 +z^8 +z^7 +z^6 +z^3 +z^2
b[5](1) =  +z^19 +z^17 +z^16 +z^14 +z^12 +z^10 +z^7 +z^5
+z^3 +z^2
b[5](0) =  +z^18 +z^15 +z^14 +z^12 +z^11 +z^9 +z^7 +z^6 +z^3
+z^2
b[5](2) =  +z^20 +z^14 +z^13 +z^10 +z^8 +z^6 +z^5
Lsim -t scmos -f deg2,3,8 .

v[5] = 2
pzm =  +z^3 +1
qzm[1] = +1
qzm[0] = +z^3
qzm[2] = 0

(Step 4 - Multiplication)
Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .

Lsim -f zm_to_z_2,3,8 .

Lsim -f multiply2,3,8 .
```

```
U[5] =  +z^6 +z^4 +1
V[5](0) =  +z^2 +1
V[5](1) =  +z^5 +z^4 +z^2 +z +1
V[5](2) =  +z^4 +z^3 +z^2 +z

(Step 5 - Test of Completion)
Lsim -f z_to_zm_2,3,8 .

Lsim -t scmos -f deg2,3,8 .

Lsim -t scmos -f deg2,3,8 .

(Step 6 - Algorithm Results)
Lsim -t scmos -f deg2,3,8 .

Lsim -t scmos -f deg2,3,8 .

Lsim -t scmos -f deg2,3,8 .

Lsim -t scmos -f deg2,3,8 .
k= 5
Shortest LFSR = U[5] =  +z^6 +z^4 +1
lk= 6
gk[0]= 6   gk[1]= 6   gk[2]= 6

/home/PL122/pk04>
```

References

[1] W.W. Peterson, *Error-Correcting Codes*. Cambridge, Mass: M.I.T Press, and New York: Wiley, ch. 9, 1961.

[2] C.R.P. Hartmann, "Decoding beyond the BCH bound," *IEEE Trans. Inform. Theory*, vol. IT-18, pp.441-444, May 1972.

[3] K.K. Tzeng and C.R.P. Hartmann, "Decoding beyond the BCH bound using multiple sets of syndrome sequences," *IEEE Trans. Inform. Theory*, vol IT-20, no. 2, pp.292-295, Mar. 1974.

[4] C.R.P. Hartmann and K.K. Tzeng, "Generalizations of the BCH bound," *Inform. Contr.*, vol 20, no. 5, pp.489-498, June 1972.

[5] J.L. Massey, "Shift register synthesis and BCH decoding," *IEEE Trans. Information Theory*, vol IT-15, pp. 122-127, Jan. 1969.

[6] E.R. Berlekamp, "Nonbinary BCH decoding," presented at the 1967 Internat'l Symp. on Information Theory, San Remo, Italy. ---*Algebraic Coding Theory*. New York: McGraw-Hill, 1968, chs. 7 and 10.

[7] G. Feng and K.K. Tzeng, "A Generalized Euclidean Algorithm for Multisequence Shift-Register Synthesis," *IEEE Trans. Information Theory*, vol. 35, no. 3, May 1989.

Vita

Paul Vincent Kraft - Born on July 16, 1967, Phillipsburg, New Jersey, to Joan and Harry Kraft. Graduating Class 1989, Lafayette College, Bachelor of Science, Electrical Engineering.

END OF

TITLE