

A parallel BCH decoder
by Benjamin Arthur Laws

A thesis submitted to the Graduate Faculty in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY in ELECTRICAL ENGINEERING
Montana State University
© Copyright by Benjamin Arthur Laws (1970)

Abstract:
The subject of this thesis is the design of a special purpose digital computer to decode the binary BCH codes.

The content of the thesis may be summarized as follows: First, a review of BCH decoding algorithms and decoders is made, including the applicable micro cellular and macrocellular techniques. Second, the BCH decoding algorithms are discussed and organized for execution by the parallel processor. Third, a detailed design for a parallel processor is described. Also included are discussions of fault detection and correction, operation for different code word lengths, and operation for different numbers of errors. Fourth, a computer program which simulates in detail the logic of the parallel processor is described. Fifth, a recommendation is made for further research ip the areas of parallel processor and algorithm design.

A PARALLEL BCH DECODER

by

BENJAMIN ARTHUR LAWS, JR.

A thesis submitted to the Graduate Faculty in partial fulfillment of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

Approved:

Paul E. Ehrlich
Head, Major Department

C.K. Rushforth
Chairman, Examining Committee

K. Goering
Graduate Dean

MONTANA STATE UNIVERSITY
Bozeman, Montana

June, 1970

ACKNOWLEDGMENT

The author wishes to thank Dr, C. K. Rushforth for his guidance and encouragement during the course of this graduate work and thesis research.

The financial support ot the graduate work provided by the Electrical Engineering Department and National Defense Education Act Title IV Fellowship No. 67-06601 is gratefully acknowledged, as is the thesis support of National Defense Education Act Title IV Fellowship No. 67-06601 and Office of Naval Research Contract No. 8-0009-603.

TABLE OF CONTENTS

		Page
Chapter 1:	INTRODUCTION TO THE BCH DECODING PROBLEM	1
1.1	Introduction	2
1.2	Coding Theory	4
1.3	Decoding Algorithms for the BCH Codes	6
1.4	Decoder Design Using Large Scale Integration	8
	1.4.1 Micro-cellular Arrays	8
	1.4.2 Macro-cellular Arrays	9
1.5	Organization of a Parallel Processor to Decode the BCH Codes	11
1.6	Relevance of Previous Work	13
1.7	Organization of the Remaining Chapters	14
Chapter 2:	DECODING ALGORITHMS FOR BCH CODES	15
2.1	Introduction	16
2.2	The Galois Field	16
	2.2.1 Representation of Field Elements	17
	2.2.2 The Addition Algorithm	22
	2.2.3 The Multiplication Algorithm	22
	2.2.4 Multiplicative Inversion	25
2.3	The BCH Codes	26

2.3.1 The Parity Check Algorithm 34

2.3.2 The Berlekamp Algorithm 35

2.3.3 The Chien Search Algorithm 38

2.4 Parallel Processor Algorithms. 40

2.4.1 The Modified Parity Check Algorithm 45

2.4.2 The Modified Berlekamp Algorithm . . 45

2.4.3 The Modified Chien Search Algorithm 51

2.4.4 A Detailed Example 51

2.5 Conclusion 61

Chapter 3: DETAILED DECODER DESIGN 62

3.1 Introduction 63

3.2 Simple Logic Elements 63

3.2.1 The Comparator Cascade 63

3.2.2 The Bit Locator Cascade 65

3.2.3 Input Selector Unit 68

3.3 Simple Storage Elements 68

3.3.1 The Data Register 71

3.3.2 The Shifting Data Register 71

3.3.3 The Incrementing Data Register 74

3.4 Galois Field Adder 74

3.5 Galois Field Multiplication 74

3.5.1 The Sequential Multiplier 77

	3.5.2 A Cellular Multiplier	77
	3.5.3 Fast Multiplication	80
	3.5.4 Fault Detection in Multipliers	82
	3.5.5 Generalization of Multipliers	90
3.6	Multiplicative Inversion in a Galois Field	92
	3.6.1 The Sequential Inverter	92
	3.6.2 The Combinational Inverter	93
	3.6.3 The Read Only Memory as an Inverter	95
	3.6.4 Error Checking of Multiplicative Inverters	95
3.7	System Block Diagram	96
3.8	Processor Interconnection	96
3.9	The Slave Processors	98
	3.9.1 The Parity Check Units	100
	3.9.2 The Chien Search Unit	100
	3.9.3 The Slave Register Set	103
	3.9.4 The Arithmetic Unit	105
	3.9.5 Internal Routing Network	105
	3.9.6 The Slave Control Unit	109
3.10	The Master Processor	113
	3.10.1 The Master Accumulator	113
	3.10.2 The Control and Decision Unit	116

3.11	System Timing	118
3.12	Fault Detection	121
3.13	Fault Correction	125
3.14	Operation over Other Fields	125
3.15	Circuit Simplifications	126
3.16	Conclusion	127
Chapter 4:	THE SIMULATION PROGRAM	128
4.1	Introduction	129
4.2	Program Description	129
4.3	The Simulation Subprograms	138
4.4	The Utility Subprograms	156
4.5	Program Results	158
4.6	Conclusion	158
Chapter 5:	CONCLUSION	160
5.1	Summary	161
5.2	Suggestions for Further Research	162
Appendix A:	APL Simulation Results	164
Appendix B:	Assembly Language Simulation and Results	170
Bibliography:		206

LIST OF TABLES

Table		Page
2.1	Three Representations for $GF(2^4)$	19
3.1	Multiplier Operations	105
3.2	Routing Schedule	107
4.1	Input Command Formats	132
4.2	Simulation and Utility Subprograms	133
4.3	Control Lines and Registers	135
4.4	Instruction Line Settings	138

LIST OF FIGURES

Figure		Page
1.1	The Communication Process	3
1.2	Block Diagram of the Berlekamp Processor	10
1.3	Block Diagram of the BCH Decoder	12
2.1	Multiplication by α	21
2.2	An APL Program for Galois Field Multiplication	24
2.3	A Multiplicative Inversion Algorithm	24
2.4	Parity Check Algorithm	36
2.5	Berlekamp Algorithm	39
2.6	Chien Search Algorithm	41
2.7	Block Diagram of the Parallel Processor	43
2.8	Processor Registers	44
2.9	The Parity Check Algorithm for a Parallel Processor	46
2.10	Shift Register Data Flow in R0 and R1	48
2.11	The Berlekamp Algorithm for a Parallel Processor	49
2.12	The Chien Search Algorithm for a Parallel Processor	52
2.13	Slave Constant Registers	53
2.14	Parity Check Calculations	55
2.15	Chien Search Calculations	60

3.1	Simple Logic Elements	64
3.2	Comparator Cascade	66
3.3	Bit Locator Cascade	67
3.4	Input Selector Unit	69
3.5	Various Flip-Flop Types	70
3.6	Data Register	72
3.7	Shifting Data Register	73
3.8	Incrementing Data Register	75
3.9	Galois Field Adder	76
3.10	Sequential Multiplier Circuit	78
3.11	Cellular Array Multiplier	79
3.12	Multiplication in $GF(2^4)$	81
3.13	Fast Multiplication Circuit	83
3.14	Fault Detection for Multiplication by α	85
3.15	Fault Detection for Cellular Multiplier	87
3.16	Fault Detection for $GF(2^4)$	88
3.17	Fault Detection for the Fast Multiplier	89
3.18	Generalized Multiplier Array	91
3.19	Multiplicative Inversion	94
3.20	Processor Interconnection	97
3.21	Block Diagram of the Slave Processor	99
3.22	Parity Check Unit	101
3.23	Chien Search Unit	102

3.24	Slave Register Set	104
3.25	Arithmetic Unit	106
3.26	Internal Routing Network	108
3.27	Slave Control Unit	110
3.28	Degree Sigma Unit	111
3.29	Register Selection Unit	112
3.30	Block Diagram of the Master Processor	114
3.31	Master Accumulator	115
3.32	Central Decision Unit	117
3.33	System Timing Diagram	119
3.34	Fault Detection for a Sequential Shift Register	123
4.1	The Main Program	130
4.2	Parity Check Routine	139
4.3	Master Processor Simulation	140
4.4	Slave Processor Simulation	151
4.5	Internal Routing Network	154
4.6	Multiplier Routine	155
4.7	Chien Search Routine	157

ABSTRACT

The subject of this thesis is the design of a special purpose digital computer to decode the binary BCH codes.

The content of the thesis may be summarized as follows: First, a review of BCH decoding algorithms and decoders is made, including the applicable microcellular and macrocellular techniques. Second, the BCH decoding algorithms are discussed and organized for execution by the parallel processor. Third, a detailed design for a parallel processor is described. Also included are discussions of fault detection and correction, operation for different code word lengths, and operation for different numbers of errors. Fourth, a computer program which simulates in detail the logic of the parallel processor is described. Fifth, a recommendation is made for further research in the areas of parallel processor and algorithm design.

CHAPTER 1

INTRODUCTION TO THE BCH DECODING PROBLEM

1.1 Introduction

One of the fundamental requirements of our technology is reliable communications. The communication process can be modeled as shown in Figure 1.1. The block labeled SOURCE is the originator of some form of information. The ENCODER translates the source information into a form suitable for transmission across the CHANNEL. As the information flows across the CHANNEL, it is changed or corrupted by noise. The DECODER then translates the received signal into a form acceptable to the SINK. The SINK is the destination of the information.

The transmission of binary information is becoming increasingly important. Although channels require an analog representation of binary data, the analog translator may be considered a part of the channel. The output of the encoder is a stream of binary digits, as is the input to the decoder. The channel is probablistic, with some probability for complementing any given digit which is transmitted across the channel.

If a particular sequence of binary digits is transmitted, there is probability that some of these digits will be changed during transmission. The transmitted message will be lost if this happens. It is the task of the encoder to provide redundancy in the transmitted signal so that the correct bit sequence will be received even though some of the digits were complemented by the channel.

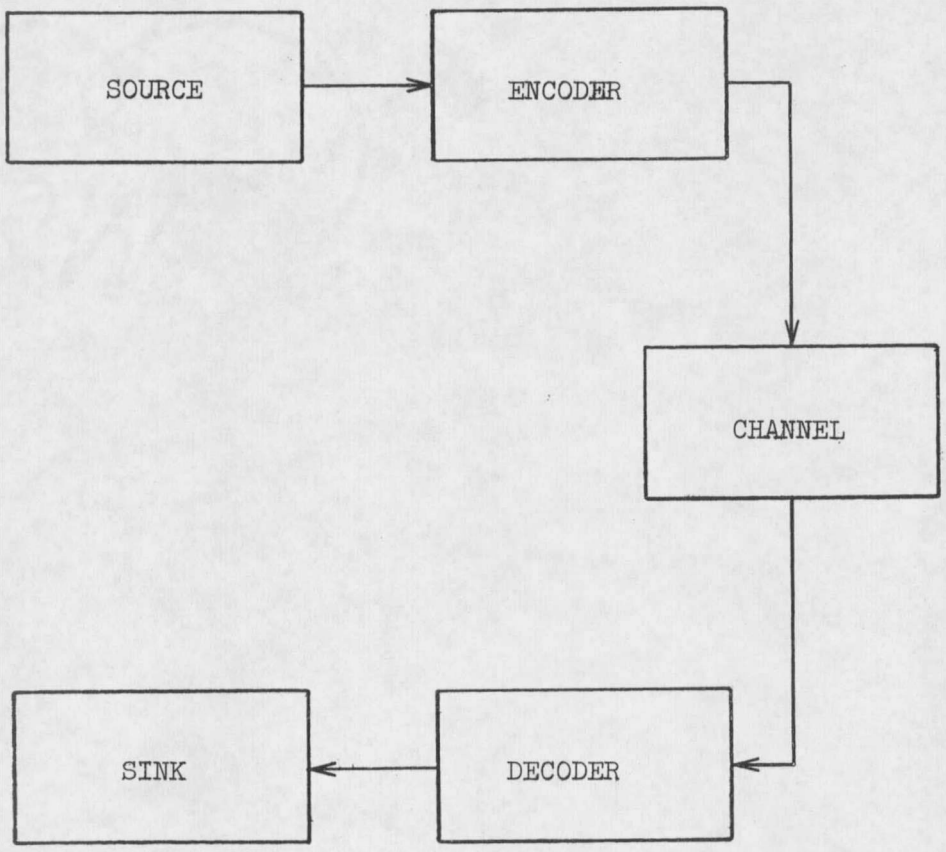


Figure 1.1. The Communication Process

1.2 Coding Theory

The motivation for coding theory was provided in 1948 by C. E. Shannon. He proved that messages can be transmitted over a channel, with an arbitrarily small probability of receiving an incorrect message, as long as the transmission rates are below channel capacity. In order to achieve this low probability of error, long complex codes are required. The central problem of Coding Theory is to find codes which will yield a low probability of error and will also be realizable in terms of hardware.

Consider the problem of encoding a sequence of binary digits. Suppose the source digits are broken into blocks of L digits. The encoder must add a set of N-L redundant (check) digits so the decoder may correct channel errors. Encoding is relatively easy to accomplish in comparison to decoding, and many simple encoders have been proposed.

The most difficult problem is the decoding--to determine which bits in the received word are in error (or to determine which L message digits were sent). Suppose the encoder provides code words such that the minimum Hamming distance between any two code words is d. Now define t to be the largest integer less than or equal to (d-1)/2. When a code word is transmitted across the channel, if the number of errors is less than or equal to t, the transmitted code word can be determined by comparing the received word with all possible code words until the

smallest Hamming distance is found. This procedure is known as maximum likelihood decoding. The process is simple conceptually, but it requires the complexity of the decoder to increase exponentially with code word length. Even if there were only 2^{32} code words, not an unreasonable number, over 2 billion comparisons would have to be made. If a comparison could be done every microsecond, over 2,000 seconds would be required to decode each received word.

One of the primary goals of Coding Theory is to find codes which can be decoded without requiring an unreasonable amount of time or equipment. The BCH codes satisfy this requirement-- although with few exceptions, the BCH decoder cannot correct as many errors as a maximum likelihood decoder.

The BCH codes were discovered in 1960 by R. C. Bose and D. K. Ray-Chaudhuri, and independently by A. Hocquenghem in 1959. They provide at least one solution to the coding problem. Only the primitive binary BCH codes will be considered, where the length of the code word is $N=2^m-1$, for $m>2$. The error-correcting capability of a length 2^m-1 code is specified by a parameter, t, in the encoding and decoding algorithms. No more than mt parity check bits are required by a t-error-correcting code. The length and error correction capability of these codes are quite flexible.

Non-binary BCH codes of length $N=p^m-1$ may be defined, where p is a prime number. Since we are concerned with transmission of

binary data, the non-binary BCH codes will not be considered.

The encoding problem for BCH codes may be solved by using a linear feedback shift register or a cellular array (1, 2, 4)*. The check digits may be computed using a generator polynomial for the particular error-correcting capability and code length desired. A number of tables useful in decoding and encoding BCH codes are given by Peterson (12).

The implementation of the BCH codes is based on arithmetic operations over a Galois field. The Galois field with p^m elements is represented $GF(p^m)$, where p is a prime number. This field defines a BCH code of length $N=p^m-1$. Again, since only the binary BCH codes are being considered, p will always be 2.

1.3 Decoding Algorithms for the BCH Codes

The first decoding algorithm for the BCH codes was published in 1960 by W. W. Peterson (8). This algorithm divides the computation into three parts:

1. Computation of the parity checks or syndrome digits.
2. Transformation of the syndrome digits into the coefficients of an error-locating polynomial--a polynomial whose roots indicate the location of errors in the word.
3. Solution of the error locating polynomial for its roots--each root indicates a correction in the received word.

*Numbers in parenthesis refer to numbered references in the Bibliography.

The algorithms are guaranteed to correct up to t errors. Peterson's decoding algorithm is the easiest to understand. However, if $e \leq t$ errors occur the algorithm requires the computation of (t-e) determinants ranging in size from t by t to e by e. A set of e simultaneous equations must then be solved to obtain the coefficients of the error locating polynomial.

T. C. Bartee and E. I. Schneider (13) have actually built and tested a special purpose computer for decoding a length 127 five-error-correcting code. The machine is a fixed-program digital computer with a single arithmetic processor, and uses Peterson's algorithm. It features a combinational circuit to perform the Galois field multiplication operation.

In 1964, R. T. Chien (9) published an algorithm which showed a simpler way to perform step 3 of the Peterson algorithm, and a way to combine steps 2 and 3 so that less time and equipment are required. Decoders were shown for correcting a small number of errors. However, in the general case, it was still necessary to solve a set of e simultaneous equations to correct e errors.

The biggest step forward came when E. R. Berlekamp (1) described an efficient algorithm for step 2. This algorithm does not require the calculation of determinants. Instead of operating on matrices, the algorithm operates on length t+1 vectors or polynomials. The algorithm requires execution of t iterations and computes a trial

error locating polynomial to determine a new trial polynomial. After t iterations, the roots of the trial error locating polynomial give the location of t or fewer errors which may have occurred anywhere in the received word.

1.4 Decoder Design Using Large-Scale Integration

With the development of Large-Scale Integration (LSI), much attention is being given to cellular arrays for implementing both logic functions and computer systems. A cellular array is a set of interconnected cells arranged in some regular fashion. Each cell in a micro-cellular array contains a small number of gates and/or storage elements, while each cell of a macro-cellular array contains a large number of gates and/or storage elements.

1.4.1 Micro-Cellular Arrays

A comprehensive survey of cellular technology was compiled by R. C. Minnick (11). In addition, Minnick has published a large number of results produced by project work at Stanford Research Institute (10).

K. N. Levitt and W. H. Kautz (2) have published a paper showing cellular arrays for both encoding and decoding certain binary error correcting codes.

1.4.2 Macro-Cellular Arrays and Processors

A number of macro-cellular processors are of interest. The first is the SOLOMON (Simultaneous Operation Linked Ordinal Modular Network) computer introduced in 1962 by Slotnick, Borck and McReynolds (3). This computer has a central control and a square array of processing elements. The central control unit provides instructions to each processing element. Each processing element has storage, an arithmetic processor, and the ability to transfer data serially to and from its four nearest neighbors in the array.

The ILLIAC IV computer is a direct descendent of SOLOMON, but is much larger and has a much greater computational facility in the processor cells (3). It is intended to be supervised by an extremely powerful general purpose computer. The machine is currently under construction by the Burroughs Corporation.

The parallel matrix processor designed by L. E. Cannon (3) is also of interest. This machine is a square array of processor cells interconnected to perform matrix operations efficiently. It is a specially-organized matrix processor intended for use with a general purpose computer.

Berlekamp (1) provides a description of a specially-organized computer for implementing step 2 of the BCH decoding algorithm. Figure 1.2 shows the block diagram given by Berlekamp. Each processor cell or slave processor can do multiplication and addition in a

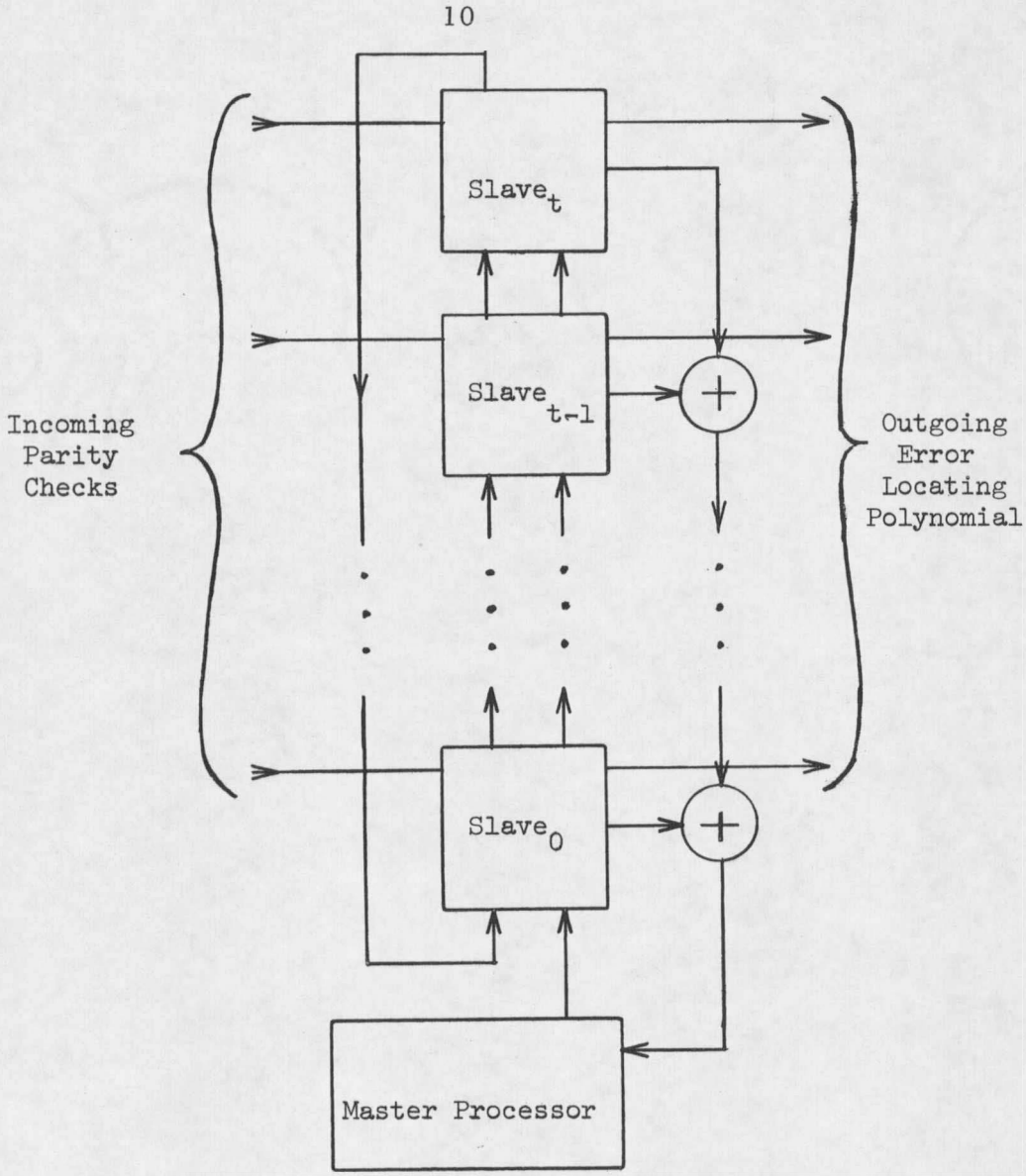


Figure 1.2. Block Diagram of the Berlekamp Processor

Galois field. The slaves may communicate elements between each other in the upward direction. The central control, or master processor, can communicate control signals and one field element to all slaves and can perform multiplicative inversion. A network of adders can take a sum of elements from all slave processors. In order to correct t errors, one master and t+1 slaves are required. The parity checks are entered into the appropriate slave registers and after t iterations, the error locating polynomial is returned in another set of slave registers. Five registers are needed in each slave.

Figure 1.3 shows the block diagram of the BCH decoder. The leftmost block computes the parity checks or syndrome digits. The center block transforms the syndrome digits into the coefficients of the error locating polynomial and the rightmost block evaluates this polynomial for each element of the Galois field to determine which bits in the received word are in error. The received word buffer stores the incoming bits and the output of the Chien search unit is combined with the buffer output in an exclusive or gate to correct the digits which are in error.

1.5 Organization of a Parallel Processor to Decode the BCH Codes

The organization which was suggested by Berlekamp has been used as a starting point in this design. The processor requires seven registers--five for storage and two for sequential shifting of information

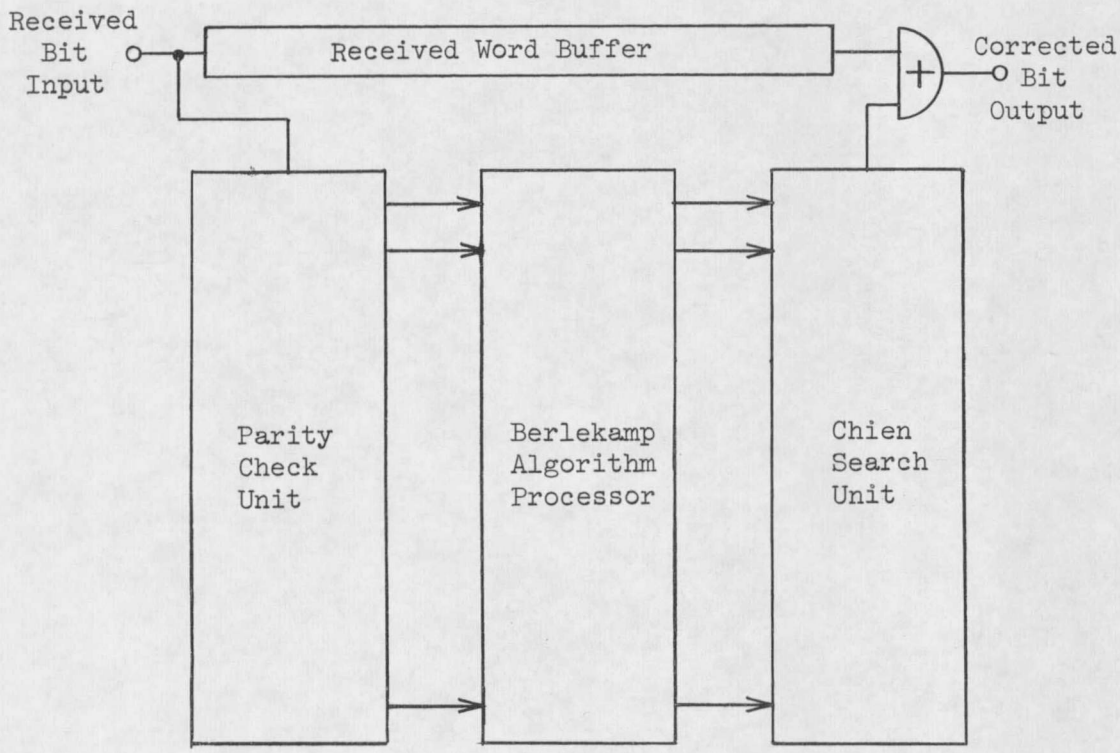


Figure 1.3. Block Diagram of a BCH Decoder

between slaves. The slave processors were designed with a number of considerations in mind:

1. The number of external connections to the slave processors was kept as small as possible.

2. The algorithms were compressed to do as much computation as possible during each clock cycle--the multipliers operate during every cycle, and many register transfers are done during each cycle.

3. The circuitry for performing each part of the algorithm was designed to operate for any code word of length 2^m-1 (up to some maximum bit storage).

4. The number of errors to be corrected by the processor is variable, being limited only by the maximum number of slave processors available and the length of the code. If t+1 processors are available, a maximum of t errors can be corrected.

5. Circuitry is developed which will provide detection of single bit faults in the slave and master processors' circuitry.

6. The decoder can be modified to allow deletion of faulty processors when a fault is detected.

1.6 Relevance of Previous Work

Previous micro-cellular work prompted the development of a cellular array for performing Galois field multiplication over any field of reasonable size. Macro-cellular work prompted design of the processor as a specially-organized, special purpose, parallel processor. The development of the Berlekamp algorithm provided a starting point for the design of such a processor. Finally, the development of LSI may soon allow complete processors to be constructed on just a few chips so that the use of t processors become even more practical.

1.7 Organization of the Remaining Chapters

Chapter 2 discusses the theory of Galois fields and BCH codes. The algorithms for decoding BCH codes are presented and modified for use by the parallel processor. Numerous APL programs were written to illustrate the algorithms. A detailed example is given for the operation of the decoder.

Chapter 3 presents the detailed logic design for the slave and master processors. The modifications required to decode any code length are discussed along with slave deletion and fault detection.

Chapter 4 is a discussion of a detailed simulation program of the parallel processor, written in the assembly language for the XDS Sigma 7.

Finally, Chapter 5 summarizes the important results and gives suggestions for further research.

CHAPTER 2

DECODING ALGORITHMS FOR BCH CODES

2.1 Introduction

Numerous algorithms for encoding and decoding the BCH codes have been described in the literature (1, 4, 8, 9). The encoding algorithm is quite easily implemented with a sequential shift register circuit. However, the decoding algorithms are considerably more difficult to implement. In this chapter algorithms will be presented which will lead to the design of a decoder for the BCH codes.

In order to understand the BCH codes and the associated encoding and decoding algorithms, a knowledge of abstract algebra, especially Galois field theory, is required. An excellent introduction to algebra and the theory of BCH codes is given by Gallager (4). We provide a brief summary of that material here.

2.2 The Galois Field

In general, a Galois field, denoted $GF(p^m)$, may have p^m elements, where p is a prime and $p \geq 2$. BCH codes may be defined using any Galois field, but the digits of the code must be taken from the set $\{0, 1, 2, \ldots, p-1\}$. Since we are interested only in the binary BCH codes, and since the arithmetic operations in $GF(p^m)$ are difficult to implement for $p > 2$, only the special case of $p = 2$ will be considered. $GF(2^m)$ is a set of 2^m elements with $m \geq 1$ and two operations called addition (+) and multiplication ($\cdot$) such that the

following axioms are satisfied:

1. For all a, b in $GF(2^m)$, a+b is in $GF(2^m)$ and a · b is in $GF(2^m)$.

2. For all a, b in $GF(2^m)$, a+b=b+a and a · b=b · a.

3. There exists an additive identity, 0, and a multiplicative identity, 1, such that for all a in $GF(2^m)$, a+0=a and a · 1=a. In addition a · 0=0.

4. For all a in $GF(2^m)$, there exists an additive inverse b such that a+(b)=0. In the binary Galois fields, every element a is its own inverse; that is a+a=0, a=-a.

5. For all a in $GF(2^m)$ such that $a \neq 0$, there exists a multiplicative inverse, a^{-1}, such that $a \cdot a^{-1} = 1$.

6. For all a, b, c in $GF(2^m)$, the distributive law is satisfied. That is $c \cdot (a+b) = (c \cdot a) + (c \cdot b)$.

2.2.1 Representation of Field Elements

Associated with every Galois field is an element α, called a primitive element. Any nonzero element of the field can be represented as a power of α. That is, the set $\{0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^m-2}\}$ represents all 2^m field elements. The exponent notation is used in the standard sense; e.g. α^5 means $\alpha \cdot \alpha \cdot \alpha \cdot \alpha \cdot \alpha$.

In order to relate this representation to a representation for the addition operator, an irreducible polynomial for the field must be introduced. This polynomial is defined as

$$f(x) = x^m + \sum_{i=1}^{m-1} f_i x^i + 1,$$

where the f_i coefficients are contained in $GF(2) = \{0, 1\}$. The only

requirement of this polynomial is that it be of order m for $GF(2^m)$, and that it have no factors with coefficients in GF(2). A polynomial which meets these requirements is called an irreducible polynomial of degree m.

Suppose that α is a primitive of $GF(2^m)$ such that $f(\alpha)=0$. If f(x) is an irreducible polynomial of degree m and if f(x) is the polynomial of lowest degree such that $f(\alpha)=0$, then f(x) is said to be a primitive polynomial of degree m. This polynomial may be used to define a representation for $GF(2^m)$ which is isomorphic to any other representation for $GF(2^m)$. Recall that b=-b so that a+b=c and a=b+c are equivalent in $GF(2^m)$. By definition of f(x), if $f(\alpha)=0$, then

$$f(\alpha) = \alpha^m + \sum_{i=1}^{m-1} f_i \cdot \alpha^i + 1 = 0$$

or,

$$\alpha^m = \sum_{i=1}^{m-1} f_i \cdot \alpha^i + 1.$$

This equation means that any element of $GF(2^m)$ can be represented not only as a power of α, but as a polynomial in α of order m-1 or less. Consider as an example the field $GF(2^4)$ with irreducible polynomial $f(x) = x^4+x+1$. Table 2.1 shows three representations for the elements of this field. The left column shows the representation of field elements expressed as powers of α, the center column as

Power of α	Polynomial in α	Binary Representation
(0)	(0)	0000
$1=\alpha^0$	1	0001
α	α	0010
α^2	α^2	0100
α^3	α^3	1000
α^4	$\alpha + 1$	0011
α^5	$\alpha^2 + \alpha$	0110
α^6	$\alpha^3 + \alpha^2$	1100
α^7	$\alpha^3 + \alpha + 1$	1011
α^8	$\alpha^2 + 1$	0101
α^9	$\alpha^3 + \alpha$	1010
α^{10}	$\alpha^2 + \alpha + 1$	0111
α^{11}	$\alpha^3 + \alpha^2 + \alpha$	1110
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$	1111
α^{13}	$\alpha^3 + \alpha^2 + 1$	1101
α^{14}	$\alpha^3 + 1$	1001

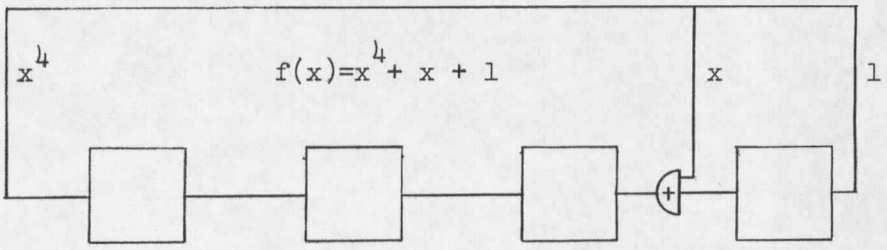
Table 2.1. Three Representations for $GF(2^4)$

polynomials in α, and the right column as binary m-tuples. The coefficients for these m-tuples are taken directly from the polynomials in α.

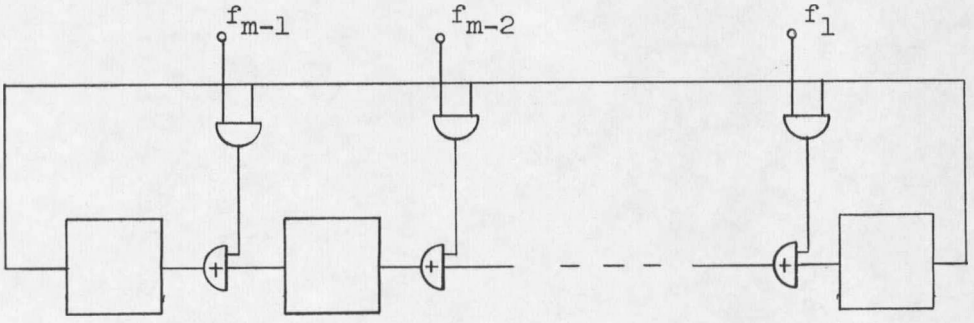
Now consider the multiplication of two elements, say $\alpha^3 \cdot \alpha = \alpha^4$. But $\alpha^4 = \alpha + 1$ since the irreducible polynomial is $x^4 + x + 1$. Therefore, $\alpha^3 \cdot \alpha = \alpha^4 = \alpha + 1$, or in binary notation,

$$(1000) \cdot (0010) = (0011).$$

Multiplication by α is quite easily implemented using a linear feedback shift register, as shown in Figure 2.1(a). Figure 2.1(b) shows the circuit for a general field. Multiplication of an element $a(\alpha)$ by α is accomplished by a left shift. If the α^{m-1} coefficient of $a(\alpha)$ is 1, the irreducible polynomial $f(x)$ is used to reduce the degree of the element $\alpha \cdot a(\alpha)$ to m-1 or less.



(a)



(b)

Figure 2.1. Multiplication by α

2.2.2 The Addition Algorithm

The operation of addition is quite easily implemented in $GF(2^m)$. Suppose $a(\alpha)$ and $b(\alpha)$ are elements of the field. Then

$$a(\alpha) = \sum_{i=0}^{m-1} a_i \cdot \alpha^i,$$

and

$$b(\alpha) = \sum_{i=0}^{m-1} b_i \cdot \alpha^i.$$

The sum of these two elements is defined by

$$a(\alpha) + b(\alpha) = \sum_{i=0}^{m-1} (a_i + b_i)\, \alpha^i,$$

where $a_i + b_i$ indicates modulo-two addition. The exclusive-or logic function performs this addition operation, and bit by bit parallel addition is accomplished by using m exclusive-or gates operating in parallel.

2.2.3 The Multiplication Algorithm

Suppose $a(\alpha)$ and $b(\alpha)$ are elements of $GF(2^m)$. The product of $a(\alpha)$ and $b(\alpha)$ is defined as

$$a(\alpha) \cdot b(\alpha) = \sum_{i=0}^{m-1} b_i \left(\sum_{j=0}^{m-1} a_j \alpha^{i+j} \right).$$

Any power of α greater than m-1 in this product must be reduced using the irreducible polynomial to form an m-bit binary representation. Suppose Z is an m-bit shift register with feedback wired for multiplication by α. An algorithm to perform multiplication of two field elements is:

M1. Set the accumulator, Z, to 0. Set j to m-1.

M2. Multiply Z by α and add to Z the m-bit binary element $b_j \wedge a(\alpha)$. The $\wedge$ operator is the logical AND.

M3. Decrease j by 1 and if j is not less than 0, return to step M2. Otherwise, the product $a(\alpha) \cdot b(\alpha)$ is contained in Z.

The algorithm computes the partial products

$$b_m a(\alpha)$$

$$(b_m a(\alpha))\alpha + b_{m-1} a(\alpha)$$

$$((b_m a(\alpha))\alpha + b_{m-1} a(\alpha))\alpha + b_{m-2} a(\alpha)$$

$$\vdots$$

$$a(\alpha) \cdot b(\alpha).$$

Figure 2.2 shows an APL program for performing multiplication of two m-bit field elements. The vectors X and Y hold the two elements to be multiplied, while the vector Z is the accumulator. Modulo 2 addition is performed by the $\neq$ operator. Z[1] holds the highest order term of the polynomial. The irreducible polynomial is stored in IRPOLY.

```
     ∇  MULTXY
[   1]  Z←(ρ X)ρ 0
[   2]  IT←1
[   3]  →((0=+/Y)∨(0=+/X))/0
[   4]  JT←ρ Y
[   5]  MULTB1: Z←1↑Z
[   6]  T1←Z[JT]
[   7]  Z[JT]←0
[   8]  Z←Z≠(IRPOLY∧T1)≠X∧Y[IT]
[   9]  IT←IT+1
[  10]  →(IT≤JT)/MULTB1
     ∇
```

Figure 2.2. An APL Program for Galois Field Multiplication

```
     ∇  INVERSE
[   1]  T1←X
[   2]  Z←((¯1+ρX)ρ0),1
[   3]  INV1: T2←T1[1]
[   4]  T1[1]←0
[   5]  T1←(T2∧IRPOLY)≠1↑T1
[   6]  T2←Z[1]
[   7]  Z[1]←0
[   8]  Z←(T2∧IRPOLY)≠1↑Z
[   9]  →(0≠+/T1[ι¯1+ρT1])/INV1
     ∇
```

Figure 2.3. A Multiplicative Inversion Algorithm

2.2.4 Multiplication Inversion

For any nonzero element a in $GF(2^m)$, there is an element b such that $a \cdot b = 1$. The multiplicative inversion algorithm must find the element b for any element a such that this condition holds. An algorithm which may be implemented using sequential shift registers is given by Berlekamp [1]. Another approach is to consider the m-input m-output truth table formed by listing the elements and their inverses. A combinational function may then be designed to implement the inverse operation. Bartee and Schneider (5) give another method for designing a combinational inverter.

The APL program in Figure 2.3 shows a very crude way to compute the inverse of an element X. The element to be inverted is placed in the vector T1 and multiplied successively by α^{-1}. At the same time, the element in T2 is initialized to be 0...01, and multiplied by α. When T1 is the element 0...01, T2 contains the inverse of X. This method is quite inefficient since computation time is proportional to 2^m.

Probably the fastest and most easily implemented method for performing multiplicative inversion is with a fast random-access memory. The element to be inverted is used as an address to the memory location where the inverse element is stored. Two shift registers may be used to initialize the memory. One shift register multiplies by α, the other by α^{-1}. They are both initialized to the

unity field element, and after the first shift, α^{-1} is stored in the memory location corresponding to α. After n shift cycles, α^{-n} is stored in locations α^{n}. The operation is continued until all field elements and their inverses have been computed and stored.

2.3 The BCH Codes

Before introducing the algorithms used to decode the BCH codes, let us discuss the mathematics behind the decoding algorithms. Consider a sequence of $N=2^{m}-1$ binary digits--a word. This set of digits may be written as a polynomial

$$c(x) = \sum_{i=0}^{N-1} c_i x^i.$$

The c_i correspond to the binary digits in the N-bit word. There are 2^N possibilities for c(x). The BCH encoding algorithm selects a subset of these 2^N possibilities such that the minimum Hamming distance between any two code words allows t or fewer errors to be corrected. The Galois field $GF(2^m)$ is used to define a primitive BCH code of length $N=2^{m}-1$.

The polynomial c(x) is a code word of a t error correcting code if and only if the following equations are satisfied:

$$\begin{aligned} c(\alpha) &= 0 \\ c(\alpha^2) &= 0 \\ c(\alpha^3) &= 0 \\ &\vdots \\ c(\alpha^{2t-1}) &= 0. \end{aligned}$$

Again, α is a primitive element of $GF(2^m)$. These calculations are called parity checks or syndrome digits.

Suppose that c(x) is transmitted across the channel and the j^{th} bit of c(x) is complemented. The received word, r(x), is defined as:

$$r(x) = c(x) + n(x) = c(x) + x^j.$$

The polynomial n(x) represents the changes made by the channel (noise). Now suppose we make the parity check calculations using r(x):

$$\begin{aligned} r(\alpha) &= c(\alpha) + n(\alpha) = 0 + \alpha^j = \alpha^j \\ r(\alpha^2) &= c(\alpha^2) + n(\alpha^2) = 0 + (\alpha^2)^j = (\alpha^j)^2 \\ r(\alpha^3) &= c(\alpha^3) + n(\alpha^3) = 0 + (\alpha^3)^j = (\alpha^j)^3 \\ &\vdots \qquad\qquad\qquad\qquad \vdots \\ r(\alpha^{2t-1}) &= c(\alpha^{2t-1}) + n(\alpha^{2t-1}) = 0 + (\alpha^{2t-1})^j = (\alpha^j)^{2t-1}. \end{aligned}$$

The $c(\alpha^i)$ terms are all zero by definition of the code word, c(x). The location of the error in the c_j bit is given by the field element α^j.

Each bit of the received word corresponds to a field element:

codebit-- c_{N-1} c_{N-2} c_2 c_1 c_0

corresponds to-- α^{N-1} α^{N-2} α^2 α 1

Suppose t errors occur in bit locations corresponding to the field elements $x_1, x_2, x_3, \ldots, x_t$. The parity check calculations yield:

$$r(\alpha) = \sum_{i=1}^{t-1} x_i = S_1$$

$$r(\alpha^2) = \sum_{i=1}^{t-1} x_i^2 = S_2$$

$$r(\alpha^3) = \sum_{i=1}^{t-1} x_i^3 = S_3$$

$$r(\alpha^{2t-1}) = \sum_{i=1}^{t-1} x_i^{2t-1} = S_{2t-1}$$

The elements $S_1, S_2, S_3, \ldots, S_{2t-1}$ are known elements obtained by calculation from the received word and the elements $x_1, x_2, x_3, \ldots, x_t$ are the unknown elements giving the location of the t errors. The problem is to solve this set of 2t-1 equations in t unknowns. The solution is complicated by the fact that the unknowns are raised to powers in the simultaneous equations.

In 1961, Peterson (8) proposed a method for solving these equations. Instead of solving for the unknowns directly, the parity check calculations $S_1, S_2, \ldots, S_{2t-1}$ were transformed into coefficients of an error-locating polynomial. This polynomial has $x_1, x_2, \ldots, x_t$ as its roots. The general form of the polynomial for t errors is:

$$\sigma(x) = x^t + \sigma_1 x^{t-1} + \sigma_2 x^{t-2} + \ldots + \sigma_{t-1} x + \sigma_t$$

$$= x^t + \sum_{i=1}^{t} \sigma_i x^{i-1}$$

$$= (x+x_1)(x+x_2)\ldots(x+x_t).$$

Consider the problem for $t=1$. The syndrome calculation is just $S_1 = x_1$, and the error locating polynomial is $x+x_1 = x+S_1 = 0$. That is, $\sigma_1 = S_1$. If $t=2$, we have:

$$S_1 = x_1 + x_2$$

$$S_2 = x_1^2 + x_2^2$$

$$S_3 = x_1^3 + x_2^3.$$

The error locating polynomial is

$$\sigma(x) = x^2 + \sigma_1 x + \sigma_2$$

$$= (x + x_1)(x + x_2)$$

$$= x^2 + (x_1 + x_2)x + x_1 x_2 .$$

Therefore,

$$\sigma_1 = x_1 + x_2 = S_1 ,$$

and

$$\sigma_2 = x_1 x_2 = \frac{x_1^2 x_2 + x_1 x_2^2}{x_1 + x_2}$$

$$= \frac{x_1^3 + x_1^2 x_2 + x_1 x_2^2 + x_2^3 + S_3}{x_1 + x_2}$$

$$= \frac{S_1 S_2 + S_3}{S_1} .$$

This means that

$$S_3 + S_1 S_2 + S_1 \sigma_2 = 0 .$$

But $S_1 = \sigma_1$, yielding the simultaneous equations

$$\sigma_1 + 0\sigma_2 = S_1$$
$$S_2\sigma + S_1\sigma_2 = S_3.$$

In general, the transformation is given by the matrix equation

(9)

$$A\sigma = B$$

where

$$A = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ S_2 & S_1 & 1 & \ldots & 0 \\ S_4 & S_3 & S_2 & \ldots & 0 \\ . & . & . & & . \\ . & . & . & & . \\ . & . & . & & . \\ S_{2t-2} & S_{2t-3} & S_{2t-4} & & S_{t-1} \end{bmatrix},$$

$$\bar{\sigma} = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \cdot \\ \cdot \\ \cdot \\ \sigma_t \end{bmatrix}, \text{ and } \qquad B = \begin{bmatrix} S_1 \\ S_3 \\ S_5 \\ S_7 \\ \cdot \\ \cdot \\ \cdot \\ S_{2t-1} \end{bmatrix}$$

These equations are known as Newton's identities. When $t=2$, the matrix equations are the same as those previously derived:

$$\begin{bmatrix} 1 & 0 \\ S_2 & S_1 \end{bmatrix} \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix} = \begin{bmatrix} S_1 \\ S_3 \end{bmatrix}.$$

The only problem remaining is the solution of the error-locating polynomial $\sigma(x)$ for the elements $x_1, x_2, \ldots, x_t$. Since $GF(2^m)$ is a finite field, there is a finite number of possibilities for these roots. Each root corresponds to a bit in the received word;

therefore, the error locations may be found by evaluating $\sigma(x)$ for each element in the Galois field. The computations are

$$\sigma(1)$$

$$\sigma(\alpha)$$

$$\sigma(\alpha^2)$$

$$\sigma(\alpha^3)$$

$$\cdot$$

$$\cdot$$

$$\cdot$$

$$\sigma(\alpha^{N-1}).$$

If $\sigma(\alpha^i)$ is 0, then the bit C_i corresponding to α^i should be complemented.

In 1964, Chien (9) described a more efficient way to organize these calculations. By definition of $\sigma(x)$, $\sigma(1)$ is given by

$$\sigma(1)=1+\sigma_1+\sigma_2+\ldots+\sigma_t.$$

Now suppose the first term of the sum is multiplied by α^t, the second by α^{t-1}, etc. The expression becomes

$$\sigma(\alpha) = \alpha^t + \sigma_1\alpha^{t-1} + \sigma_2\alpha^{t-2} + \cdots + \sigma_t.$$

Repeated application of this operation yields $\sigma(\alpha^2)$, $\cdots$, $\sigma(\alpha^{N-1})$.

The algorithm developed by Berlekamp produces an error locating polynomial of the form

$$\sigma(x) = \sigma_t x^t + \sigma_{t-1}x^{t-1} + \cdots + \sigma_1 x + 1.$$

The roots of this equation are the inverses of the actual error locations. The bit corresponding to α^{N-1} is received first in order to facilitate calculation of the parity checks.

With this material as background, the three parts of the decoding algorithm are now considered in more detail.

2.3.1 The Parity Check Algorithm

Suppose we wish to calculate the j^{th} parity check, $S_j = C(\alpha^j)$. The following algorithm may be used:

P1. Initialize: Set PC, the parity check accumulator, to zero. Set i to $N-1 = 2^m - 2$.

P2. Set $PC = PC \cdot \alpha^j + C_i$.

P3. Set i to i-i. If i is less than zero, terminate, otherwise go to step P2.

This algorithm calculates C_{N-1}, $(C_{N-1}\alpha^j) + C_{N-2}$, $(C_{N-1}\alpha^j + C_{N-2})\alpha^j + C_{N-3}$, etc. until $C(\alpha^j)$ is produced.

An APL program which calculates the parity checks required to correct t errors is shown in Figure 2.4. The odd-order parity checks are calculated using the parity check algorithm above, while the even-order parity checks are calculated from the formula $C(\alpha^{2i}) = [C(\alpha^i)]^2$. The routine MULTXY multiplies X and Y in $GF(2^m)$ and places the product in the vector Z.

2.3.2 The Berlekamp Algorithm

The Berlekamp algorithm (1) converts the parity checks into the error locating polynomial, $\sigma(z)$, whose roots determine the location of errors in the code word. To describe this algorithm, the syndrome digits are represented by the polynomial

$$1+S = 1 + \sum_{i=1}^{2t-1} C(\alpha^i)z^i = 1 + \sum_{i=1}^{2t-1} S_i z^i .$$

The error locating polynomial is given by

$$\sigma(z) = 1 + \sum_{i=1}^{t} \sigma_i z^i .$$

An auxiliary polynomial, $\tau(z)$, is also required by the algorithm. All polynomial coefficients are elements of $GF(2^m)$.

```
     ∇ PARITY
[  1] IA←1
[  2] S←((2×T),M)ρ 0
[  3] PX←(2×T)ρ 0
[  4] X←A1
[  5] BA1: IB←0
[  6] Z←0
[  7] BA2: IB←IB+1
[  8] Y←Z≠A0∧CW[IB]
[  9] →(IB≥ρ CW)/BA5
[ 10] MULTXY
[ 11] →BA2
[ 12] BA5: Z←Y
[ 13] S[IA;]←Z
[ 14] IB←2×IA
[ 15] IP←X
[ 16] BA4: →(IB>2×T)/BA3
[ 17] X←Z
[ 18] Y←Z
[ 19] MULTXY
[ 20] S[IB;]←Z
[ 21] PX[IB]←1
[ 22] IB←2×IB
[ 23] →BA4
[ 24] BA3: X←IP
[ 25] BA6: Y←1↑A1
[ 26] MULTXY
[ 27] X←Z
[ 28] IA←IA+2
[ 29] →(IA>¯1+2×T)/BA7
[ 30] →(PX[IA])/BA4
[ 31] →BA1
[ 32] BA7: S←(1 0+ρS)ρA0,,S
     ∇
```

Figure 2.4. Parity Check Algorithm

The parenthesized superscript notation, $\sigma^{(2k)}$ will be used to identify polynomials computed during each iteration. The algorithm is:

B1. Initialize: Define $\sigma^{(0)} = \tau^{(0)} = 1$, $k=0$.

B2. Let $\Delta_1^{(2k)}$ be the coefficient of z^{2k+1} in the product $(1+S)\ \sigma^{(2k)}$.

B3. Let $\sigma^{(2k+2)} = \sigma^{(2k)} + \Delta_1^{(2k)}\ (z\ \ \tau^{(2k)})$.

If $k=t$, terminate.

B4.

$$\text{Let } \tau^{(2k+2)} = \begin{cases} z^2 \cdot \tau^{(2k)} \text{ if } \Delta_1^{(2k)} = 0, \text{ or if} \\ \qquad \text{degree } \sigma^{(2k)} > k. \\ \\ z\cdot\sigma^{(2k)} \cdot (\Delta_1^{(2k)})^{-1} \text{ if } \Delta_1^{(2k)} \neq 0 \\ \qquad \text{and degree } \sigma^{(2k)} \leq k. \end{cases}$$

Increment k by 1 and return to B2.

Figure 2.5 shows an APL program designed to implement the Berlekamp algorithm. The variable correspondences should be obvious. SIG [1;] gives the element for σ_0, SIG [2;] for σ_1, etc. The polynomial is listed in the reverse order to facilitate calculation of $\Delta_1^{(2k)}$. That is, S[2×T;] corresponds to $S_0 = I$, and S [1;] corresponds to $S_{2t-1} = C(\alpha^{2t-1})$. The second dimension of all arrays is m, the size of the field. The Iota function is used to find the index of the first non-zero term in the reversed or reduction of the SIG polynomial. Then this value is subtracted from the maximum length of the polynomial to obtain the degree of the polynomial in SIG. The routine MULT is identical to the routine MULTXY, except that the APL function definition is slightly different. MULTR does the element by element multiplication between two polynomials.

2.3.3 The Chien Search Algorithm

This algorithm must find the roots of a polynomial of degree t in $GF(2^m)$. These roots are found by simply evaluating the polynomial at each element in the field. The Chien search is simply an organized way to perform these calculations using $\sigma(\alpha^i)$ to compute $\sigma(\alpha^{i+1})$. Imagine a set of registers, denoted CSR, such that CSR_0 corresponds to $\sigma_0 = 1$, CSR_1 to σ_1, ... , CSR_t to σ_t. The algorithm is (1):

C1. Set the CSR accumulators to the coefficients of $\sigma(z)$. Set i to $N-1 = 2^m-2$.

C2. Multiply CSR_1 by α^1, CSR_2 by α^2, ... , CSR_t by α^t, and store each product back in the appropriate CSR register.

```
     ∇ BERLEKAMP
[   1] SIG← TAU← ((T+1),M) ρ0
[   2] INITIALIZE:K←0
[   3] SIG[1;M]← TAU[1;M]← 1
[   4] RESTART: DEL1← ≠/[1] S[1+T+(ι T+1)-2×K;] MULTR SIG
[   5] TEMP← (ρ SIG) ρ (M ρ 0),,SIG[ι T;]
[   6] X← ∨/[2] SIG
[   7] DSIG← (ρ X)- X[1+(ρ X) - ι ρ X] ι 1
[   8] TAU← (ρ TAU) ρ (M ρ 0),,TAU[ι T;]
[   9] SIG← SIG ≠ TAU MULTR ( ρ TAU) ρ DEL1
[  10] →(K=T)/0
[  11] →((0 = ∨/DEL1) ∨ (DSIG > K))/BM1
[  12] TAU← TEMP MULTR (ρ SIG) ρ INVERT DEL1
[  13] → BM2
[  14] BM1: TAU← (ρ TAU) ρ (M ρ 0),,TAU[ι T;]
[  15] BM2: K←K+1
[  16] →RESTART
     ∇

     ∇ Z←X MULT Y
[   1] MULTXY
     ∇

     ∇ RZ←RX MULTR RY
[   1] MULTRXY
     ∇

     ∇ Z←INVERT X
[   1] INVERSE
     ∇
```

Figure 2.5. Berlekamp Algorithm

C3. Add (modulo 2) the contents of all CSR registers. The CSR_0 register should contain the field element 1. If the sum is zero, complement the C_i codebit.

C4. Set i to i - 1. If i is less than zero, terminate. Otherwise, return to step C2.

Figure 2.6 shows an APL implementation of this algorithm. The function POLY evaluates the error locating polynomial at a point X in the field. The function CHIEN supplies the necessary changes in the field element X and corrects the digit in CW if the polynomial is zero.

2.4 Parallel Processor Algorithms

The algorithms described above must be modified somewhat if they are to be performed by a parallel processor. The algorithms will be presented in terms of vector operations so the applications for a parallel processor become more obvious. By using a parallel processor, the time required to decode a word can be substantially reduced. In other words, we are trading equipment for time. In order to calculate the parity checks, we must compute 2t-1 digits. Each of these digits may be computed independently with 2t-1 processors. Next, the Berlekamp algorithm may be performed by a set of t+1 processors. Most of the operations required by the Berlekamp algorithm are performed on length t+1 vectors. For the Chien search, each coefficient of the error locating polynomial must be multiplied by the appropriate power of α. Since there are t+1 terms, processing time may be reduced by using t+1 processors.

```
     ∇ CHIEN
[  1] IA←1
[  2] IB←0
[  3] PX←SIG
[  4] X←A1
[  5] BC1: POLY
[  6] CW[IA]←CW[IA]=∨/Z
[  7] IB←IB+~∨/Z
[  8] →(IB=DSIG)/0
[  9] IA←IA+1
[ 10] Y←A1
[ 11] MULTXY
[ 12] X←Z
[ 13] →(IA≤ρ CW)/BC1
     ∇
```

```
     ∇ POLY
[  1] Z←0
[  2] IP←1
[  3] BP2:→(IP=(ρ PX)[1])/BP1
[  4] Y←PX[IP;]≠Z
[  5] MULTXY
[  6] IP←IP+1
[  7] →BP2
[  8] BP1:Z←Z≠PX[IP;]
     ∇
```

Figure 2.6. Chien Search Algorithm

In addition to the vector processors or slave processors, there is a master processor. This unit provides control and clocking signals to all slave processors and also performs the operation of multiplicative inversion as required by the Berlekamp algorithm. The master processor also makes the various decisions required by the algorithms.

Figure 2.7 shows the block diagram of the processors for the BCH decoder. The received word is passed sequentially through the received word buffer, and each bit is communicated to the processors. There are t+1 slave processors and one master processor. Each slave is able to compute two of the parity checks for one received word while concurrently determining the error locating polynomial for another word and performing the Chien search for a third word.

Figure 2.8 shows the registers contained in each slave processor and the registers contained in the master processor. Each row of registers represents the registers in one slave. Each column represents a particular register set, e.g. R0, in all slaves. One register, say $R0_3$, can store one m-bit field element. The C1 and C2 registers hold the powers of α necessary for computing the parity checks. The C3 register holds the powers of α necessary for computing the Chien search. PC1 and PC2 are the parity check accumulators. CSR is the Chien search accumulator. The registers R0, R1, R2, R3, R4, SR0 and SR1 are used by the Berlekamp algorithm. Each row of the figure denotes the registers of one slave processor. The registers MA, K and D at the bottom of the

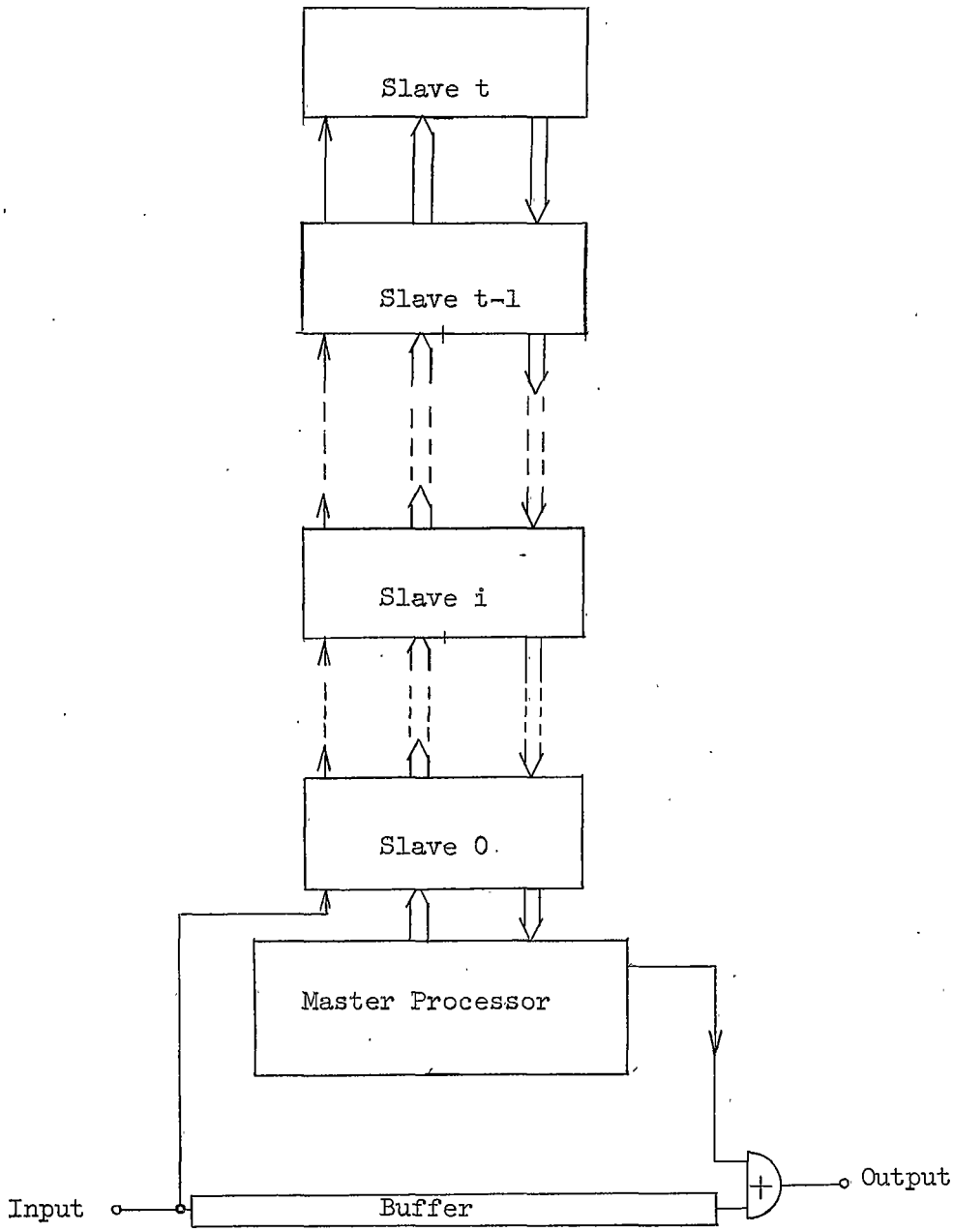


Figure 2.7. Block Diagram of the Parallel Processor

	R0	R1	R2	R3	R4	SR0	SR1
Slave 3	$R0_3$	$R1_3$	$R2_3$	$R3_3$	$R4_3$	$SR0_3$	$SR1_3$
Slave 2	$R0_2$	$R1_2$	$R2_2$	$R3_2$	$R4_2$	$SR0_2$	$SR1_2$
Slave 1	$R0_1$	$R1_1$	$R2_1$	$R3_1$	$R4_1$	$SR0_1$	$SR1_1$
Slave 0	$R0_0$	$R1_0$	$R2_0$	$R3_0$	$R4_0$	$SR0_0$	$SR1_0$

	C1	C2	PC1	PC2	C3	CSR
Slave 3	$C1_3$	$C2_3$	$PC1_3$	$PC2_3$	$C3_3$	CSR_3
Slave 2	$C1_2$	$C2_2$	$PC1_2$	$PC2_2$	$C3_2$	CSR_2
Slave 1	$C1_1$	$C2_1$	$PC1_1$	$PC2_1$	$C3_1$	CSR_1
Slave 0	$C1_0$	$C2_0$	$PC1_0$	$PC2_0$	$C3_0$	CSR_0

MA	K	D

Figure 2.8. Processor Registers

figure are in the master processor, and hold the value of $\Delta^{(2k)}$, k, and degree ($\sigma^{(2k)}$) respectively.

2.4.1 The Modified Parity Check Algorithm.

This algorithm makes use of the two parity check calculators in each of the t+1 slave processors. The algorithm is:

P1'. Set PC1=PC2=0, in all slaves, and set i=N-2.

P2'. Set PC1=PC1$\cdot$C1+C_i and PC2=PC2$\cdot$C2+C_i, where C_i is either the 0 or 1 field element, depending upon whether the received bit is 0 or 1.

P3'. Set i=i-1. If i is less than 0, terminate. Otherwise, return to P2'.

Figure 2.9 shows an APL program which implements this algorithm. Note that the received bit, CW[IP] is added only to the rightmost bit in the PC1 or PC2 registers. The order of the parity checks is completely dependent upon the contents of C1 and C2; that is, if α^i is stored in $C1_i$, S_i will be computed in $PC1_i$.

2.4.2 The Modified Berlekamp Algorithm.

The following register correspondences will be used:

R0 = Parity checks
R1 = Parity checks
R2 = Sigma polynomial
R3 = Tau polynomial
R4 = Temporary storage for $z^2\tau^{(2k)}$
SR0 = Shifting register
SR1 = Shifting register
MA = Master accumulator--holds Δ_l and Δ_l^{-1}
K = Iteration count
D = Degree sigma register.

```
     ∇  PARITY
[  1]  PC1← PC2←(ρ C1) ρ 0
[  2]  IP← 1
[  3]  PCBR: PC1← C1 MULTR PC1
[  4]  PC1[;M]← CW[IP]≠PC1[;M]
[  5]  PC2← C2 MULTR PC2
[  6]  PC2[;M]← CW[IP]≠PC2[;M]
[  7]  IP← IP+1
[  8]  →(IP ≤ ρ CW)/ PCBR
[  9]  PC2[2;]← ((M-1) ρ 0),1
[ 10]  PC2[2 + ι 2;] ← (2,M) ρ 0
[ 11]  S←((2 + 2×T),M) ρ (,PC1),,PC2
     ∇
```

Figure 2.9. The Parity Check Algorithm for a Parallel Processor

Note that SR0 and SR1 are used for shifting field elements between slaves. SR0 provides a shift of 1 slave while SR1 provides a shift of 2 slaves. That is, during a particular clock cycle the contents of $SR0_i$ move to $SR0_{i+1}$, and the contents of $SR1_i$ move to $SR1_{i+2}$. The outputs of slave t are connected to provide a circular shift for SR1. Figure 2.10 shows the flow of data in the R0 and R1 registers which must occur as the algorithm progresses.

The modified algorithm is presented below.

B1'. Let K=0, and SR0=SR1=R2=$1+0\cdot z+0\cdot z^2+\ldots+0\cdot z^t$.

B2'. Set MA to the sum of the products of registers R1 and R2 in all slaves. Shift SR0 and place the result in R3. Shift SR1 and place the result in R4. Let SR0=R2 and SR1=R0. Let K=K+1.

B3'. Let D be given the value of the degree of the sigma polynomial stored in R2. Let R2=R2+MA•R3. Shift SR1 and replace R0. Reload SR1 with R1. Replace the contents of the MA with the inverse of the element previously contained in the MA. If the number stored in the K register is equal to t, terminate.

B4'. Shift SR1 and replace R1. In slaves 0 and 1, set R1=R0 and R0=SR0. If the contents of the D register are greater than or equal to the contents of the K register, or if the contents of MA is the 0 field element, let SR0=SR1=R4. Otherwise, let SR0=SR1=SR0•MA. Return to step B2'.

An APL program which implements this modified algorithm is shown in Figure 2.11. The Routine MULTRXY multiplies the slave registers pair by pair. The INVERSE routine computes the inverse of X, and returns the value in Z.

Each of the steps in the algorithm corresponds to a complete

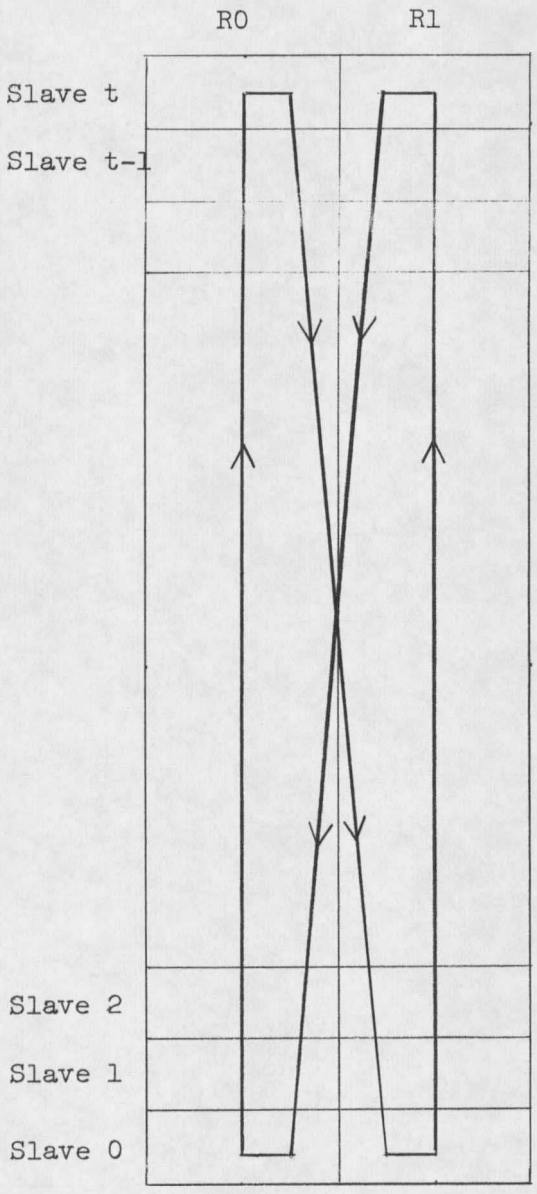


Figure 2.10. Shift Register Data Flow in R0 and R1

```
     ∇  BERM2
[  1]  INIT
[  2]  CYC1: RX← R1
[  3]  RY← R2
[  4]  MULTRXY
[  5]  MA←≠/[1] RZ
[  6]  SR1←R3
[  7]  SHIFTSR1
[  8]  R5←SR1
[  9]  SR0←R3
[ 10]  SHIFTSR0
[ 11]  R3←SR0
[ 12]  K←K+1
[ 13]  X←∨/ R2
[ 14]  DSIG←(ρX) - X[ 1+(ρ X)-⍳ρ X]⍳ 1
[ 15]  CYC2: X← IMA← MA
[ 16]  →(0=+/X)/BM2
[ 17]  INVERSE
[ 18]  IMA←Z
[ 19]  BM2: SR0←R2
[ 20]  SHIFTSR0
[ 21]  R4←SR0
[ 22]  SR1←R0
[ 23]  SHIFTSR1
[ 24]  R0←SR1
[ 25]  RX←R3
[ 26]  RY←(ρ R3)ρ MA
[ 27]  MULTRXY
[ 28]  R2← RZ≠ R2
[ 29]  →(K=T)/0
[ 30]  MA← IMA
[ 31]  CYC3: SR1← R1
[ 32]  SHIFTSR1
[ 33]  T2← 2+⍳ ¯2+(ρ SR1)[1]
[ 34]  R1[ T2;]← SR1[T2;]
[ 35]  R1[ 1 2;]← R0[ 1 2;]
[ 36]  R0[ 1 2;]← SR1[ 1 2;]
[ 37]  RX← R4
[ 38]  RY←(ρ R4)ρ MA
[ 39]  MULTRXY
[ 40]  T1←(DSIG≥K)∨~∨/MA
[ 41]  R3←(T1∧R5)∨RZ∧~ T1
[ 42]  →CYC1
     ∇
```

Figure 2.11. The Berlekamp Algorithm for a Parallel Processor

```
     ∇  INIT
[   1]  K← 0
[   2]  DSIG← 0
[   3]  R0← S[ T+ 2-⍳ T+1;]
[   4]  R1←(⍴ R0)⍴ 0
[   5]  RZ←R1
[   6]  R1[ 2+ T-⍳ T- 1;]← S[ T+ 1+⍳T-1;]
[   7]  R2←(⍴ R0)⍴(- M)↑( M×T+1)⍴1
[   8]  R3← R2
[   9]  SR1←R0
[  10]  SHIFTSR1
[  11]  R0←SR1
[  12]  SR1←R1
[  13]  SHIFTSR1
[  14]  T2← 2+⍳ ¯2+(⍴ SR1)[1]
[  15]  R1[ T2;]← SR1[ T2;]
[  16]  R1[ 1 2;]← R0[ 1 2;]
[  17]  R0[ 1 2;]← SR1[ 1 2;]
     ∇

     ∇  SHIFTSR0
[   1]  SR0[⍳↑ T+ 1;]← SR0[⍳ T+1;]
     ∇

     ∇  SHIFTSR1
[   1]  SR1[ 2⍳↑ T+1;]← SR1[⍳ T+1;]
     ∇


     ∇  MULTRXY
[   1]  IMR← 0
[   2]  BMR: →(IMR≥(⍴ RX)[1])/0
[   3]  IMR← IMR+ 1
[   4]  X← RX[IMR;]
[   5]  Y← RY[IMR;]
[   6]  MULTXY
[   7]  RZ[IMR;]←Z
[   8]  →BMR
     ∇
```

Figure 2.11. (continued)

clock cycle in the final processor design. All register transfers excluding shift operation, are done at the end of each clock cycle.

2.4.3 The Modified Chien Search Algorithm

The modified Chien search algorithm is presented below.

C1'. Initialize: Set CSR = R2.

C2'. Compute CSR • C3 and add the multiplier outputs of all slave Chien search multipliers. If the sum is zero, correct the corresponding received bit. Replace CSR with the product CSR • C3.

C3'. Repeat C2' until the last received bit is processed. Then terminate.

An APL program which realizes this algorithm is given in Figure 2.12.

2.4.4 A Detailed Example

Suppose that a length 15 triple error correcting BCH code is to be decoded. The all zero codeword was transmitted, and the following word was received:

r_{14}	r_{13}	r_{12}	r_{11}	r_{10}	r_9	r_8	r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0
0	0	1	0	0	0	0	0	0	0	1	0	0	0	1

Let us follow the step by step operation of the decoder. Figure 2.13(a) shows the constants which must be stored in C1, C2, and C3 in order to decode a t-error-correcting code, and Figure 2.13(b) shows the

```
     ∇ CHIEN
[  1] CSR←SIG
[  2] IC←1
[  3] CHBR: CSR← CSR·MULTR C3
[  4] CW[IC]← CW[IC] = ∨/' '≠/[1] CSR
[  5] IC← IC+1
[  6] →(IC ≤ ρ CW)/CHBR
     ∇
```

Figure 2.12. The Chien Search Algorithm for a Parallel Processor

C1	C2	C3	
α^2	α^{3+t}	α^t	Slave t
α^3	α^{4+t}	α^{t-1}	Slave t-1
.	.	.	
.	.	.	
.	.	.	
α^{t-1}	0	α^3	Slave 3
α^t	0	α^2	Slave 2
α^{t+1}	1	α	Slave 1
α^{t+2}	α	1	Slave 0

(a)

C1	C2	C3	
0100	0000	1000	Slave 3
1000	0000	0100	Slave 2
0011	0001	0010	Slave 1
0110	0010	0001	Slave 0

(b)

Figure 2.13. Slave Constant Registers

constants necessary to decode the length 15 triple error correcting code. Four slaves are required. Suppose the leftmost bit is transmitted first. Figure 2.14 shows the contents of PC1 and PC2 during the parity check calculations. The PC1 and PC2 registers are initially zero. They are multiplied by C1 and C2, and the first received digit is added. The digit added at each step is shown directly above the register results. No change occurs in PC1 and PC2 until the third digit is processed. Note that three PC registers --in slaves 1, 2 and 3--are still zero after the last calculation. The two registers in slaves 2 and 3 hold calculations which are never used by the algorithm, and the PC2 register in slave 1 must always be 1 when the parity checks are transferred to the Berlekamp algorithm. There will always be three unused parity check registers, no matter how large t is. The final result of the parity check calculations is in the lower righthand corner of the figure.

Now, the parity checks in PC1 and PC2 are transferred into R0 and R1 respectively, and the registers R2, SR0, SR1 and K are initialized as required by B1'. The resulting register contents are:

R0	R1	R2	R3	R4	SR0	SR1
1110	0000	0000	0000	0000	0000	0000
0010	0000	0000	0000	0000	0000	0000
1011	0001	0000	0000	0000	0000	0000
0110	1101	0001	0000	0000	0001	0001

MA	K	D
0000	0	0.

	r_{14}		r_{13}		r_{12}		r_{11}	
Received Bits:	0		0		1		0	
Slave	PC1	PC2	PC1	PC2	PC1	PC2	PC1	PC2
3	0000	0000	0000	0000	0001	0000	0100	0000
2	0000	0000	0000	0000	0001	0000	1000	0000
1	0000	0000	0000	0000	0001	0000	0011	0000
0	0000	0000	0000	0000	0001	0001	0110	0010

	r_{10}		r_{9}		r_{8}		r_{7}	
Received Bits:	0		0		0		0	
Slave	PC1	PC2	PC1	PC2	PC1	PC2	PC1	PC2
3	0011	0000	1100	0000	0101	0000	0111	0000
2	0110	0000	1010	0000	1111	0000	0001	0000
1	0101	0000	1111	0000	0010	0000	0110	0000
0	0111	0100	0001	1000	0110	0011	0111	0110

	r_{6}		r_{5}		r_{4}		r_{3}	
Received Bits:	0		0		1		0	
Slave	PC1	PC2	PC1	PC2	PC1	PC2	PC1	PC2
3	1111	0000	1101	0000	0110	0000	1100	0000
2	1000	0000	1100	0000	1011	0000	0111	0000
1	1010	0000	1101	0000	0101	0000	1111	0000
0	0001	1100	0110	1011	0110	0100	0111	1000

	r_{2}		r_{1}		r_{0}	
Received Bits:	0		0		1	
Slave	PC1	PC2	PC1	PC2	PC1	PC2
3	0101	0000	0111	0000	1110	0000
2	1101	0000	0010	0000	0010	0000
1	0010	0000	0110	0000	1011	0000
0	0001	0011	0110	0110	0110	1101

Figure 2.14. Parity Check Calculations

Execution of the Berlekamp algorithm now begins. During clock cycle 1, step B2' is executed; during clock cycle 2, step B3'; and during clock cycle 3, step B4'. These three cycles are repeated t times, except that cycle 3 is not executed on the last iteration.

During clock cycle 1, the contents of R1 and R2 are multiplied and added to specify the contents of MA. SR0 and SR1 shift sequentially during the clock cycle, and at the end of the cycle, respecify R3 and R4 respectively. At the end of clock cycle 1, R2 is placed in SR0 and R0 is placed in SR1. The value of the K register is incremented by 1. The register contents after clock pulse 1 are:

R0	R1	R2	R3	R4	SR0	SR1
1110	0000	0000	0000	0000	0000	1110
0010	0000	0000	0000	0001	0000	0010
1011	0001	0000	0001	0000	0000	1011
0110	1101	0001	0000	0000	0001	0110

MA	K	D
1101	1	0

During cycle 2, SR0 and SR1 are again shifted up one and two slaves respectively and the contents of the R2 registers are added to the product of the contents of the MA register and the contents of the R3 register. These results are transferred to R2 at the end of the cycle. The D register is loaded with the degree of polynomial in the R2 register. The shifted value in SR1 is returned to R0, and SR1 is loaded with the contents of R1. The contents of R4 and SR0 are saved for use during cycle 3. The MA register receives the inverse of its

previous contents. The registers after clock cycle 2 are:

R0	R1	R2	R3	R4	SR0	SR1
1011	0000	0000	0000	0000	0000	0000
0110	0000	0000	0000	0001	0000	0000
1110	0001	1101	0001	0000	0001	0001
0010	1101	0001	0000	0000	0000	1101

MA	K	D
0100	1	0.

During clock cycle 3, the decision is made to accept either the contents of R4 as the new value of τ or to take the contents of SR0 multiplied by the contents of the MA register as the new value. The SR1 register is shifted up two slaves, and then transferred back to R1. The transfer is slightly different in slaves 0 and 1 to provide a circular shift as required by the algorithm. The new value for τ is inserted into SR0 and SR1. The contents of the registers at the end of this cycle are:

R0	R1	R2	R3	R4	SR0	SR1
1011	0001	0000	0000	0000	0000	0000
0110	1101	0000	0000	0001	0000	0000
0000	1110	1101	0100	0000	0100	0100
0000	0010	0001	0000	0000	0000	0000

MA	K	D
0100	1	0.

This completes the execution of the first iteration. Cycle 1 of iteration 2 produces:

R0	R1	R2	R3	R4	SR0	SR1
1011	0001	0000	0000	0100	0000	1011
0110	1101	0000	0100	0000	0000	0110
0000	1110	1101	0000	0000	1101	0000
0000	0010	0001	0000	0000	0001	0000

MA	K	D
1000	2	0.

After cycle 2, iteration 2, we have:

R0	R1	R2	R3	R4	SR0	SR1
0000	0001	0000	0000	0100	0000	0001
0000	1101	0110	0100	0000	1101	1101
1011	1110	1101	0000	0000	0001	1110
0110	0010	0001	0000	0000	0000	0010

MA	K	D
1111	2	1.

After cycle 3, iteration 2:

R0	R1	R2	R3	R4	SR0	SR1
0000	1110	0000	0000	0100	0000	0000
0000	0010	0110	0111	0000	0111	0111
0001	1011	1101	1111	0000	1111	1111
1101	0110	0001	0000	0000	0000	0000

MA	K	D
1111	2	1.

This completes iteration 2. Now, iteration 3, cycle 1 begins; after execution we have:

R0	R1	R2	R3	R4	SR0	SR1
0000	1110	0000	0111	1111	0000	0000
0000	0010	0110	1111	0000	0110	0000
0001	1011	1101	0000	0000	1101	0001
1101	0110	0001	0000	0000	0001	1101

MA	K	D
1100	3	1.

The final cycle is cycle 2 of iteration 3, since this cycle produces the final error location polynomial:

R0	R1	R2	R3	R4	SR0	SR1
0001	1110	0010	0111	1111	0110	1110
1101	0010	1110	1111	0000	1101	0010
0000	1011	1101	0000	0000	0001	1011
0000	0110	0001	0000	0000	0000	0110

MA	K	D
1010	3	2.

Register R2 contains the error locating polynomial. This result is transferred to the Chien search registers during processing of the last digit.

During the processing of the first digit of the word to be corrected, the contents of the CSR registers are multiplied by the contents of the C3 registers and the result is stored back in the CSR accumulators. The multiplier outputs are also added. If this sum is zero, the bit leaving the storage buffer is corrected. In this example, the CSR accumulators are initialized as:

CSR
1110
0010
1011
0001.

The CSR register operations for each codebit are shown in Figure 2.15. The sum of the calculation is shown below the register contents, and the corrected codebit is shown below that. The corrections (bits denoted by *) indicate that the all zero word was transmitted, and

Slave	CSR	CSR	CSR	CSR
3	0011	1011	0111	1101
2	1101	0001	0100	0011
1	1001	0001	0010	0100
0	0001	0001	0001	0001
Sum:	0110	1010	0000	1011
Corrected Bit:	0	0	0*	0

Slave	CSR	CSR	CSR	CSR
3	0010	0011	1011	0111
2	1100	0101	0111	1111
1	1000	0011	0110	1100
0	0001	0001	0001	0001
Sum:	0111	0100	1011	0101
Corrected Bit:	0	0	0	0

Slave	CSR	CSR	CSR	CSR
3	1101	0010	0011	1011
2	1001	0010	1000	0110
1	1011	0101	1010	0111
0	0001	0001	0001	0001
Sum:	1110	0100	0000	1011
Corrected Bit:	0	0	0*	0

Slave	CSR	CSR	CSR
3	0111	1101	0010
2	1011	1010	1110
1	1110	1111	1101
0	0001	0001	0001
Sum:	0011	1001	0000
Corrected Bit:	0	0	0*

Figure 2.15. Chien Search Calculations

the decoder has functioned correctly.

2.5 Conclusion

Decoding algorithms for the BCH codes may be performed quite efficiently by a special purpose computer organized as a parallel processor. The code word length and number of errors can be changed by changing the contents of the C1, C2 and C3 registers. A number of APL programs were written to verify that these algorithms perform the required decoding operations. The programs were linked with the proper calling sequence to perform the entire decoding operation. Appendix A lists the coordinating program along with the results for a number of code words.

CHAPTER 3

DETAILED DECODER DESIGN

3.1 Introduction

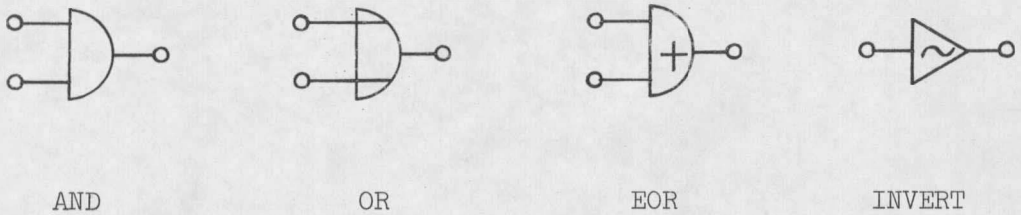
In this chapter, the detailed logic design is presented for the parallel processor described in Chapter 2. First, the simpler logic elements will be described. Then more complex units--the Galois field multiplier and inverter, including the fault detection logic for these units. The slave processor cell is discussed, and the block diagram for the processor is shown. Each unit of the slave processor is described in detail. The same approach is used to describe the master processor. Finally, the system timing, fault detection, slave replacement and generalizations to other fields are discussed.

3.2 Simple Logic Elements

The four simple logic elements to be used are the AND, OR, EOR (exclusive-or) and INVERT circuits. The symbols and truth tables for these elements are shown in Figure 3.1(a). The notation used for gate input and output inversion is shown in Figure 3.1(b), where the simple elements are used to form more complex logic units.

3.2.1 The Comparator Cascade

During execution of the Berlekamp algorithm, two comparisons must be made. One compares the degree of the trial error locating polynomial to the iteration count, and the other compares the iteration count to the number of errors to be corrected. The comparator cell (3)



AND

x	y	f
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	f
0	0	0
0	1	1
1	0	1
1	1	1

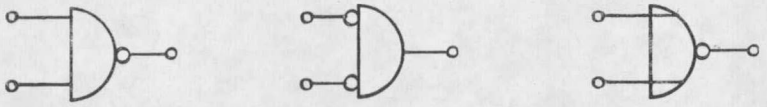
EOR

x	y	f
0	0	0
0	1	1
1	0	1
1	1	0

INVERT

x	f
0	1
1	0

(a)



NAND NOR NOR

(b)

Figure 3.1. Simple Logic Elements

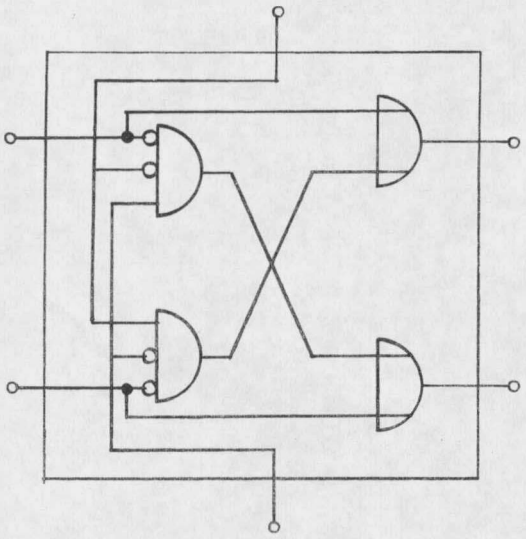
is shown in Figure 3.2(a). A symbol for the cell appears in Figure 3.2(b), and the cascade for comparison of two positive integers is shown in Figure 3.2(c).

If the binary number on the a_i inputs is larger than the binary number on the b_i inputs, the output h will be a 1 and the output g a 0. Similarly, if the number on the b_i inputs is larger than the number on the a_i inputs, the output of the g line will be a 1, and h will be a 0. If the numbers are equal, both g and h will be 0. The condition g=h=1 should not occur. Note that the left hand cell in the cascade has inputs of 0.

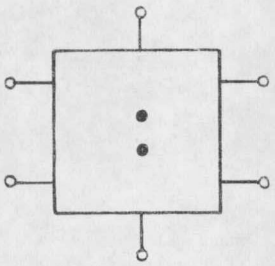
3.2.2 The Bit Locator Cascade

When the degree of the trial error locating polynomial is determined, the term of highest order must be found. Also, in the general multiplier circuit, the highest-order term of the primitive polynomial must be found. The bit locator cascade is used for these purposes. The bit locator cell is shown in Figure 3.3(a) and its symbol in 3.3(b).

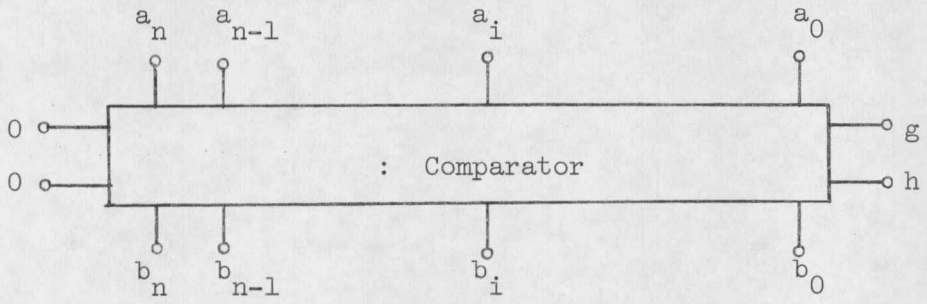
The bit locator cascade in Figure 3.3(c) has zero outputs on all b_i lines except for the b_i line corresponding to the rightmost 1 value on the a_i lines. For example, if the a lines were set to 111011010001000, the b lines would be 000000000001000. Note that the arrow indicates the direction of the search and the input to the first cell must be 0. If it is



(a)



(b)



(c)

Figure 3.2. Comparator Cascade

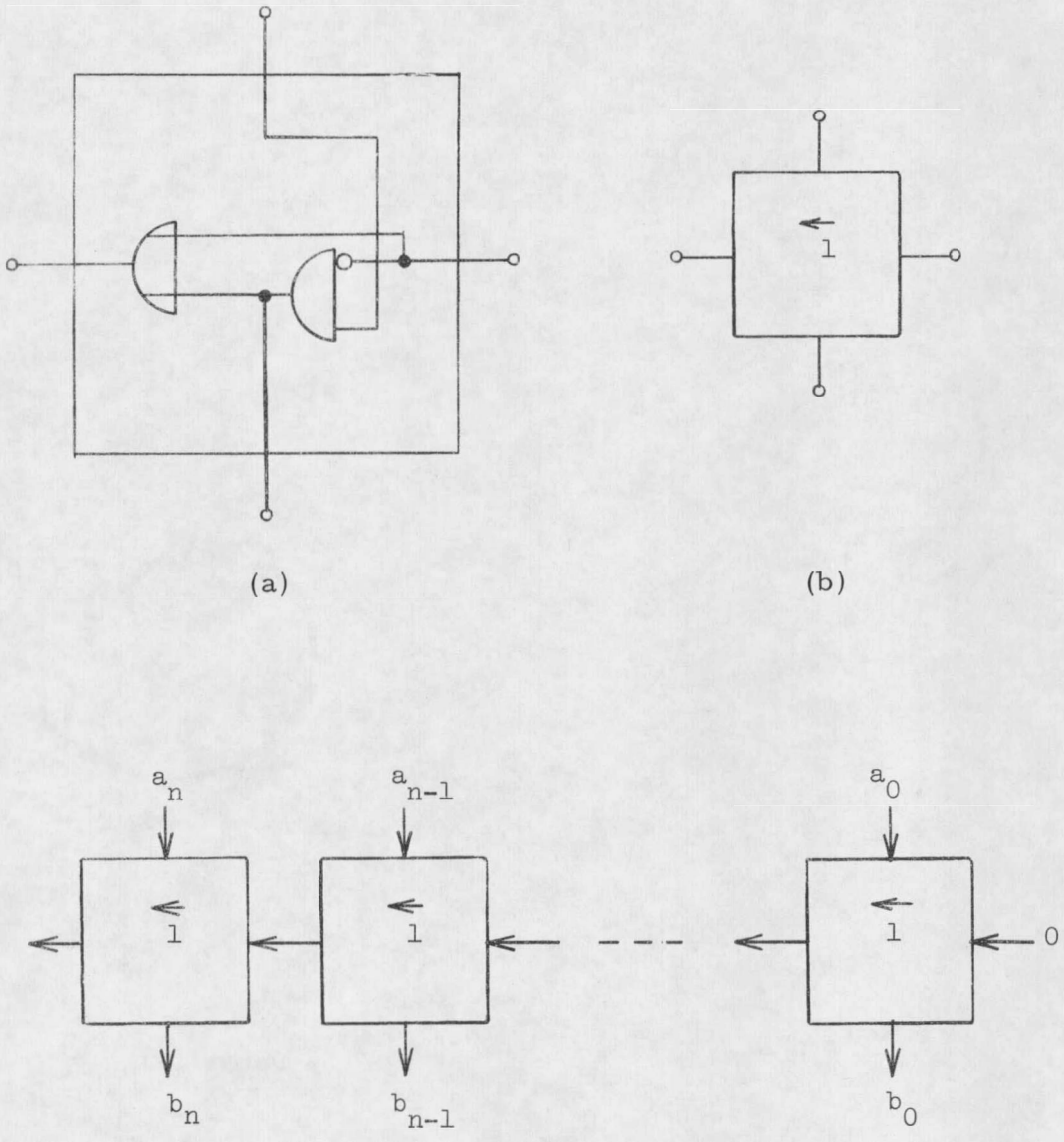


Figure 3.3. Bit Locator Cascade

non-zero, all b outputs will be zero.

3.2.3 Input Selector Unit

In the slave processor, the inputs to the multiplier are different during each cycle. These inputs are selected by a simple routing device called the input selector unit or ISU (3).

Figure 3.4(a) shows the internal circuit for a particular cell of the input selector unit. The input selector unit is used to select one of two input lines to be passed to the output. If the SELECT line is a 1, the line denoted '1' is passed to the output. If the SELECT line is 0, the line denoted '0' is passed to the output. Figure 3.4(b) shows the symbol for one unit. The units may be ganged as in Figure 3.4(c) to allow selection of one of two sets of data lines.

3.3 Simple Storage Elements

In addition to the simple logic elements, a number of simple storage elements are required. These are the data (D), set-reset (SR), and trigger (T) flip-flops as shown in Figure 3.5(a).

The D flip flop sets the value of Q to the value of the input line D when clock line I is changed from 1 to 0. The S-R flip flop sets Q to 1 if line S is 1, and sets Q to 0 if R is 1. The action occurs when clock line C is changed from 1 to 0. If the S and R inputs are equal, the value of Q is unchanged. The T flip flop complements the value of Q when line

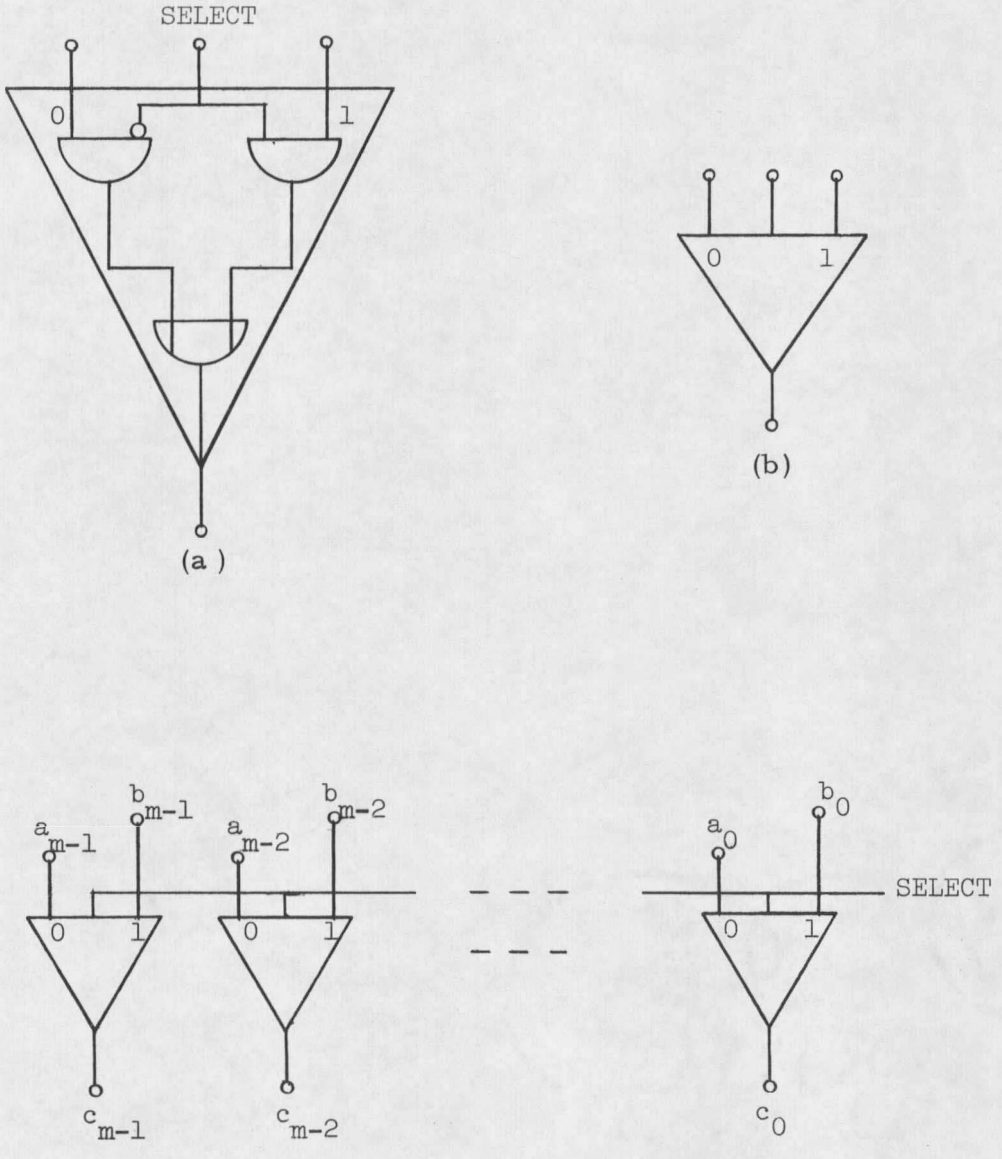
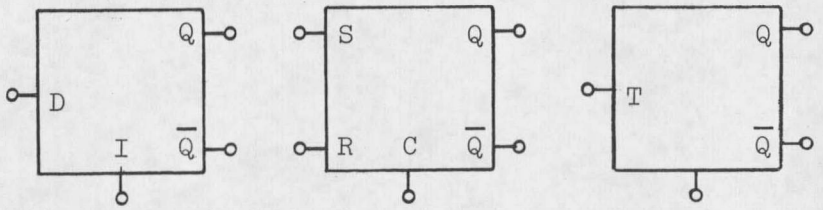
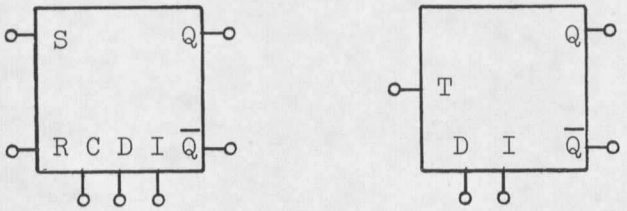


Figure 3.4. Input Selector Unit



(a)



(b)

Figure 3.5. Various Flip-Flop Types

T is changed from 1 to 0. Combinations of the three blocks are allowed, as shown in Figure 3.5(b).

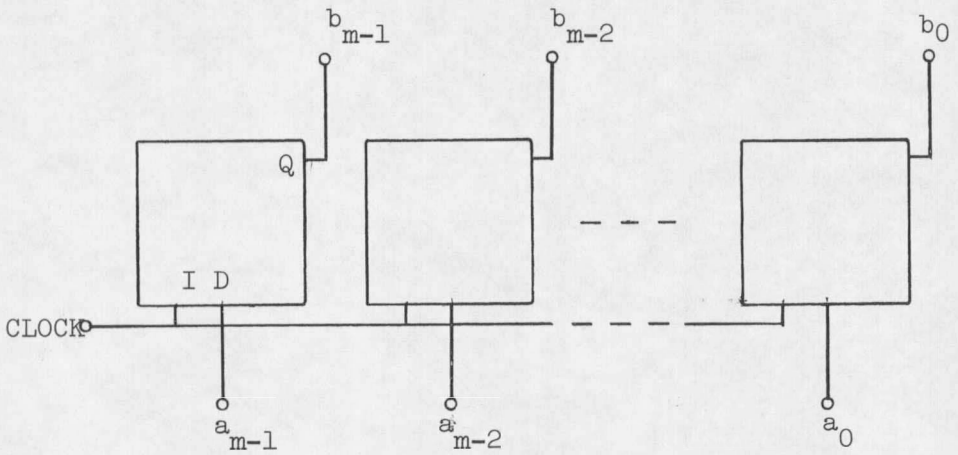
3.3.1 The Data Register

The field-element storage unit is the data register shown in Figure 3.6. In order to store an element of $GF(2^m)$, m bits are required. All field element storage is parallel read/write, as shown in Figure 3.6(a). When a clock pulse is applied on the CLOCK line, the values presented on the a_i lines are stored in the corresponding flip flops. Figure 3.6(b) shows the symbol to be used for the m-bit data register. These registers are also used to store the integers required by the Berlekamp algorithm--the degree of the error location polynomial and the number of errors to be corrected.

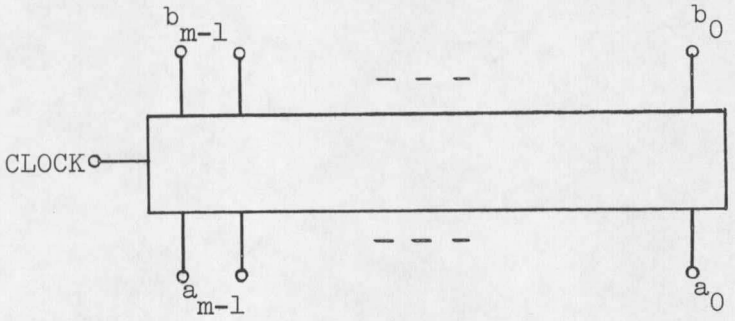
3.3.2 The Shifting Data Register

The SR0 and SR1 registers in the slave processors must transfer field elements sequentially between slave processors, and transfer elements in parallel to and from other slave registers. For this purpose, a combination of the SR and D flip flops is used.

The shifting data register in Figure 3.7(a) is identical to the storage register discussed previously, except that when a clock pulse is applied to the SHIFT lines, the bit pattern stored in the register is shifted right one position. The leftmost input, e, specifies the new

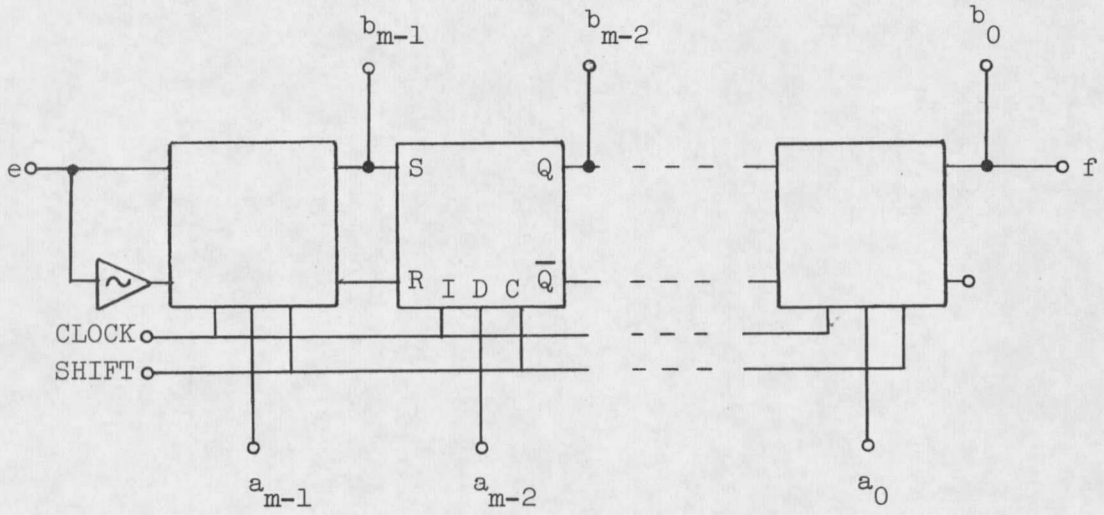


(a)

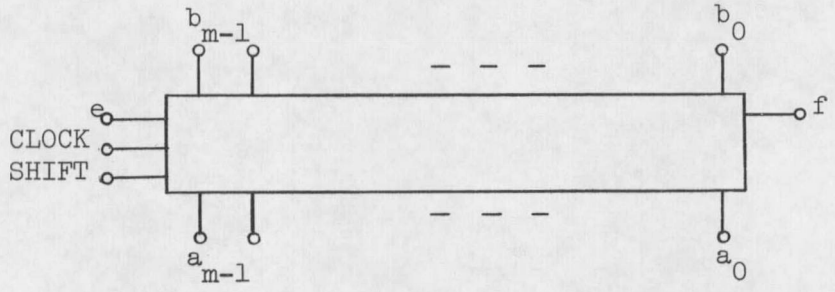


(b)

Figure 3.6. Data Register



(a)



(b)

Figure 3.7. Shifting Data Register

state of the left hand flip flop, while the rightmost bit must be passed on to another shift register via the f line. Figure 3.7 (b) shows the symbol for the shifting data register.

3.3.3 The Incrementing Data Register

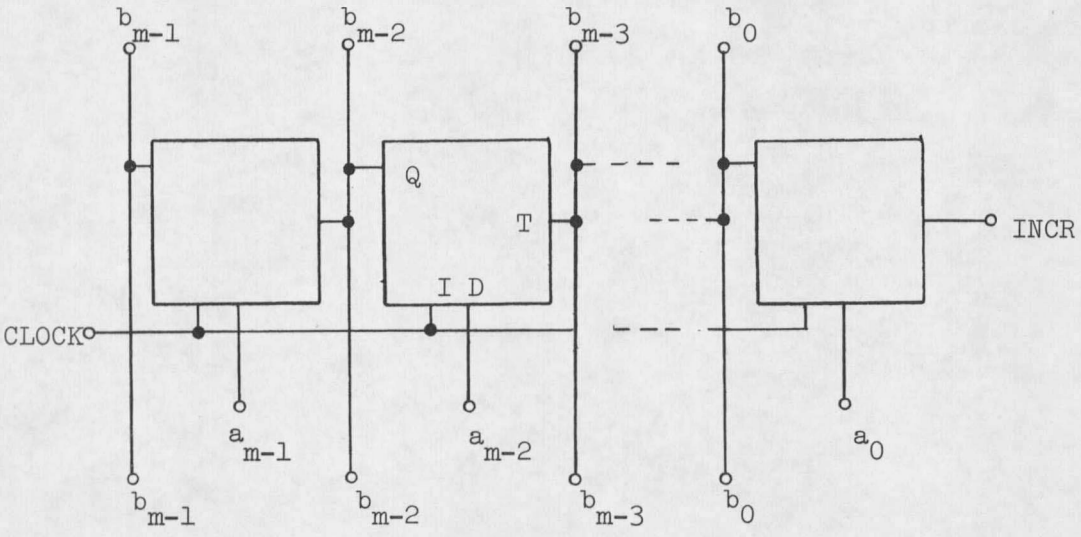
During execution of the Berlekamp algorithm, the value of the iteration count must be incremented by 1. The incrementing data register is shown in Figure 3.8 (a). When a clock pluse is applied on the INCR line, the integer stored in the register is incremented by 1. Figure 3.8 (b) shows the symbol for the register.

3.4 Galois Field Adder

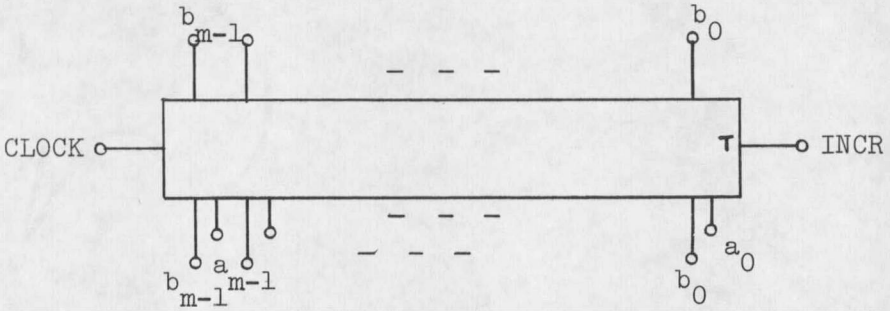
As stated in Chapter 2, the addition operation in GF (2^m) can be performed in parallel by m exclusive-or gates. Figure 3.9 (a) shows the m gates. The a_i inputs are added modulo 2 bit by bit to the b_i inputs. The c_i lines give the binary element representing the sum of the a and b elements in GF (2^m). Figure 3.9 (b) shows the block symbol for the adder.

3.5 Galois Field Multiplication

In Chapter 2 we showed that the multiplication of two elements, a and b, in GF (2^m) can be broken down into a series of multiplications by α. In this section, a number of m-bit multipliers are developed, including sequential, cellular and combinational circuits. Modifications are developed to allow fault detection and the cellular array is modified to allow operation over any field.

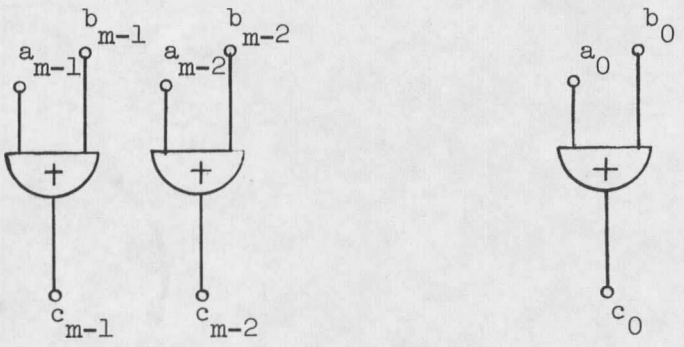


(a)

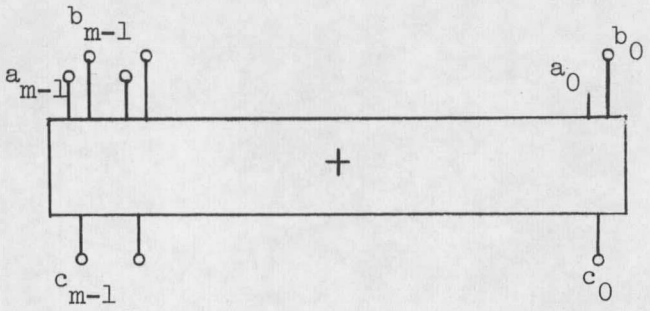


(b)

Figure 3.8. Incrementing Data Register



(a)



(b)

Figure 3.9. Galois Field Adder

3.5.1 The Sequential Multiplier

The circuit in Figure 3.10 shows a sequential multiplier (4). There are two shift registers--the Z register has linear feedback and is the accumulator for the multiply operation. The other register holds the b coefficients. The field primitive polynomial is f(x). The multiplier makes the following computation:

Cycle	Value in Z register
1	ab_{m-1}
2	$[(ab_{m-1})\cdot\alpha + a\cdot b_{m-2}] \bmod f(\alpha)$
3	$[((ab_{m-1})\alpha + ab_{m-2})\alpha + ab_{m-3}] \bmod f(\alpha)$
.	
.	
.	
m	$[((\cdots(a(\alpha)b_{m-1})\alpha + \cdots)\alpha + a(\alpha)b_1)\alpha + a(\alpha)b_0] \bmod f(\alpha)$

This circuit requires m clock pulses--as each pulse is applied to the clock line, the Z and b registers are shifted left.

3.5.2 A Cellular Multiplier

The cellular array shown in Figure 3.11 (a) multiplies the two elements a and b in GF (2^m). The operation is very similar to the operation of the sequential multiplier. The first row computes the same partial product as cycle 1 of the sequential multiplier. There are m rows in the array, and each row makes the same calculation as the corresponding cycle of the sequential multiplier. The product is produced on the c lines.

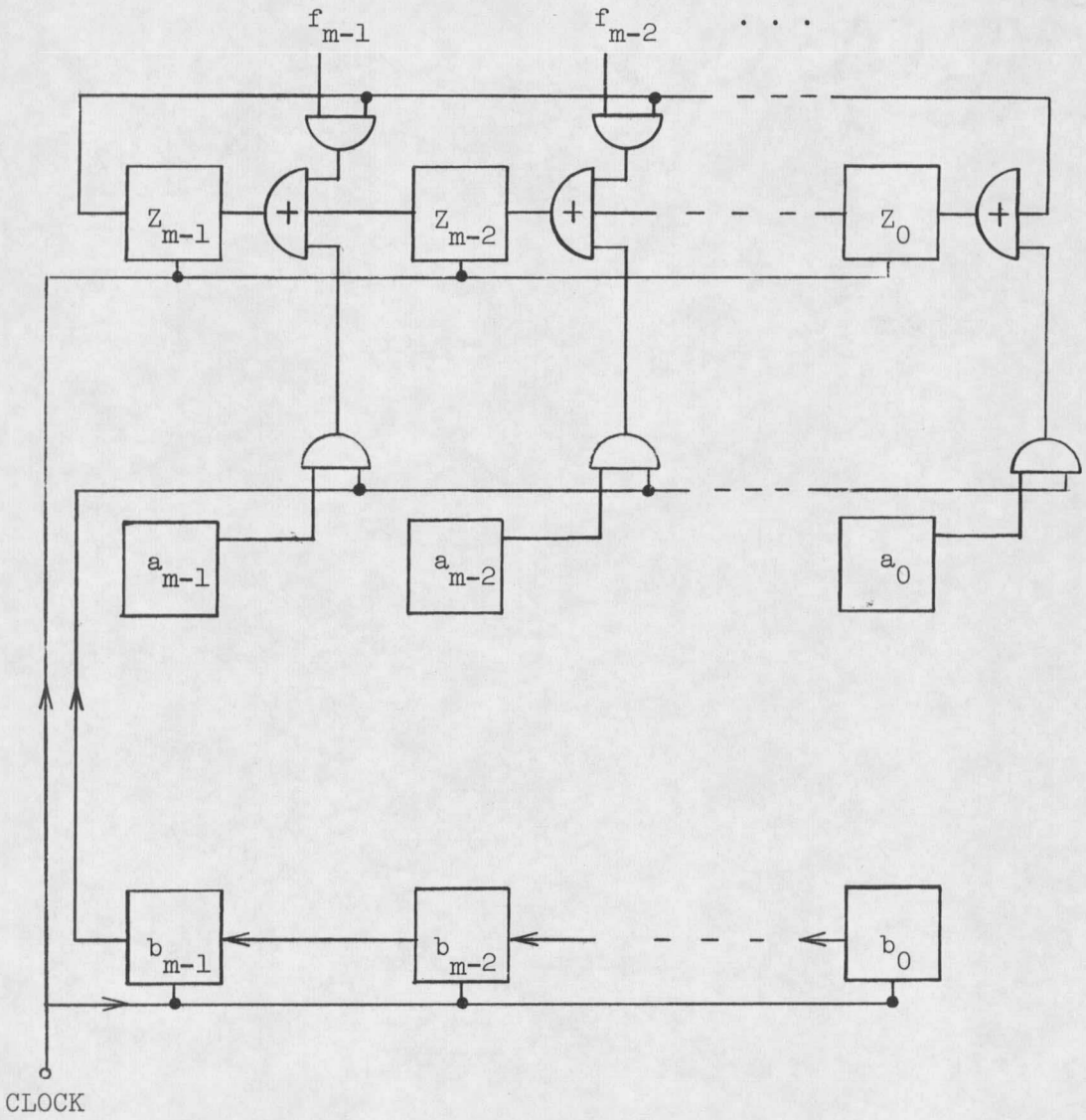


Figure 3.10. Sequential Multiplier Circuit

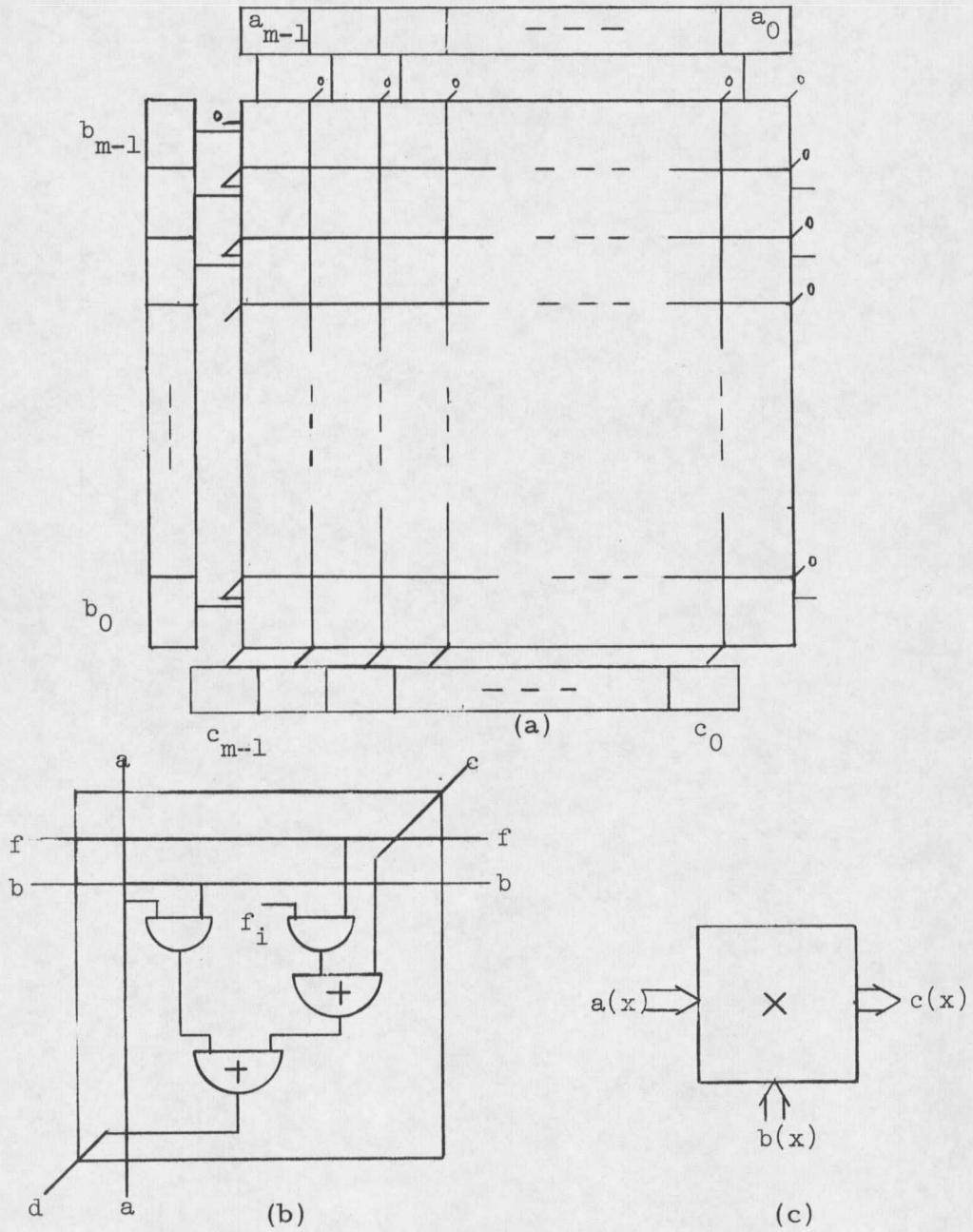


Figure 3.11. Cellular Array Multiplier

The structure of a cell is shown in Figure 3.15 (b). In the $(i, j)^{th}$ cell, the bit product $a_j b_i$ is added into the product. The output is diagonal from the lower left corner of the cell. One cell input is in the upper right corner, and the other input is the feedback (f) line. The line is driven by the leftmost cell in the $(i-1)^{st}$ row. The purpose is to reduce m^{th} order terms using the equation $f(\alpha)=0$, where $f(x)$ is the primitive polynomial for $GF(2^m)$.

The cellular array multiplier has a number of advantages. It is faster than the sequential multiplier and does not require clock signals for control. Approximately $2m+1$ gate delays are required. The regularity of the array should allow fabrication as a large scale integrated circuit. The array can also be generalized to operate over any binary field. However, the cellular array must be tested to insure that all cells are functioning properly. If a fault is detected, the faulty cell must be isolated and replaced.

3.5.3 Fast Multiplication

Bartee and Schneider (5) have shown a combinational multiplier circuit which can be constructed to operate even faster than the cellular array multiplier. Consider multiplication of two elements a and b in $GF(2^4)$. The operation is shown in Figure 3.12. Terms of order greater than 3 are reduced using the field primitive polynomial x^4+x+1. A fast multiplier is constructed by producing all cross product terms in AND gates simultaneously, and then adding the terms necessary for

$$
\begin{array}{l}
\quad a_3\alpha^3 + a_2\alpha^2 + a_1\alpha + a_0 \\
\quad b_3\alpha^3 + b_2\alpha^2 + b_1\alpha + b_0 \\
\hline
b_0a_3\alpha^3+b_0a_2\alpha^2+b_0a_1\alpha+b_0a_0 \\
b_1a_2\alpha^3+b_1a_1\alpha^2+b_1a_0\alpha \\
\qquad\qquad +b_1a_3\alpha+b_1a_3 \\
b_2a_1\alpha^3+b_2a_0\alpha^2 \\
\qquad b_2a_3\alpha^2+b_2a_3\alpha \\
\qquad\qquad b_2a_2\alpha+b_2a_2 \\
b_3a_0\alpha^3 \\
b_3a_3\alpha^3+b_3a_3\alpha^2 \\
\qquad b_3a_2\alpha^2+b_3a_2\alpha \\
\qquad\qquad b_3a_1\alpha+b_3a_1 \\
\hline
\quad c_3\alpha^3 + c_2\alpha^2 + c_1\alpha + c_0
\end{array}
$$

$$c_3 = b_0a_3 + b_1a_2 + b_2a_1 + b_3a_0 + b_1a_3$$

$$c_2 = b_0a_2 + b_1a_1 + b_2a_0 + b_2a_3 + b_3a_3 + b_3a_2$$

$$c_1 = b_0a_1 + b_1a_0 + b_1a_3 + b_2a_3 + b_2a_2 + b_3a_2 + b_3a_1$$

$$c_0 = b_0a_0 + b_1a_3 + b_2a_2 + b_3a_1$$

Figure 3.12. Multiplication in GF(2^4)

each digit of the final product. The addition circuitry is in the form of a tree as shown in Figure 3.13.

If the n additions are performed by a linear string of gates, n gate delays are required. By using the tree configuration, only $\lceil \log n \rceil$ time units are required, where $\lceil x \rceil$ is the smallest integer not less than x. The cellular array requires $2m+1$ gate delays, while the fast multiplier requires only $1+\lceil \log 2m \rceil$ gate delays. However, the cellular array can be modified so that it will operate over any field, while the fast multiplier must be constructed for a particular field.

3.5.4 Fault Detection in Multipliers

Since circuitry is subject to failure, some means of fault detection is necessary in the multiplier circuit. In order to reduce the magnitude of the problem, faults which affect only one bit will be detected. For instance, in the cellular multiplier, the failure of one cell will be detectable. Detection of single errors is accomplished by attaching a single parity bit.

Since multiplication of two arbitrary elements a and b is a series of simple multiplications by α, some means of detecting failure in this operation is necessary. Again, consider the field $GF(2^4)$ with irreducible polynomial x^4+x+1. Suppose we start with the element

0001.

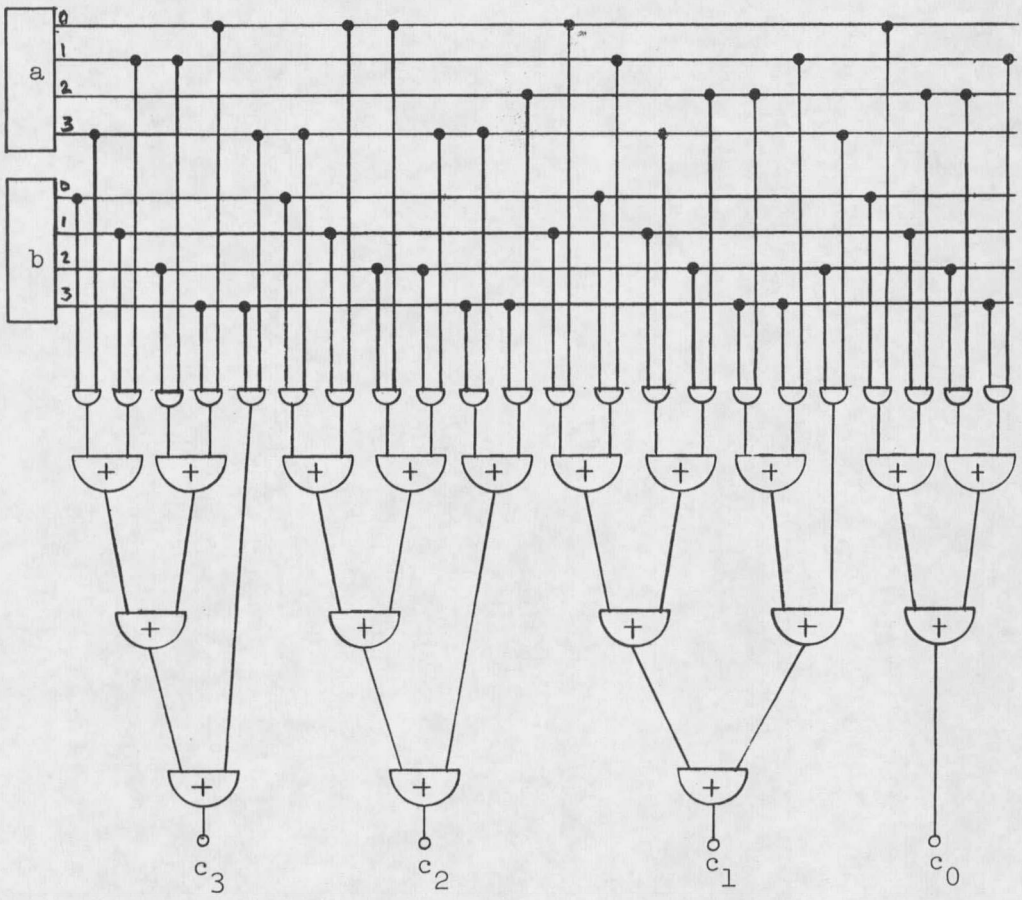


Figure 3.13. Fast Multiplication Circuit

If this element is multiplied by α, the parity is unchanged and the result is

0010.

Two more multiplications by α produce no change in the parity:

1000.

The next multiplication requires feedback, and this time the parity is changed:

0011.

If α is a generator for GF(2^m), it must be able to produce 2^m-1 distinct elements. These elements are of both even and odd parity. Since the feedback is modulo 2 addition, by linearity the parity must change every time a multiplication by α causes a term of order 2^m to be reduced. Multiplication of 0100 by α causes no change in parity; but multiplication of 1100 by α causes the parity to change since the leftmost bit is 1. If the computer parity is incorrect, a fault affecting a single bit has occurred. Figure 3.14 shows a fault detection circuit for multiplication by α.

The same method can be used to check multiplication of two elements a and b. Figure 3.12 shows the cross product terms produced by multiplication over GF(2^4). Assume further that a and b have parity bits a_p and b_p, respectively. These are

$$a_p = a_0 + a_1 + a_2 + a_3$$

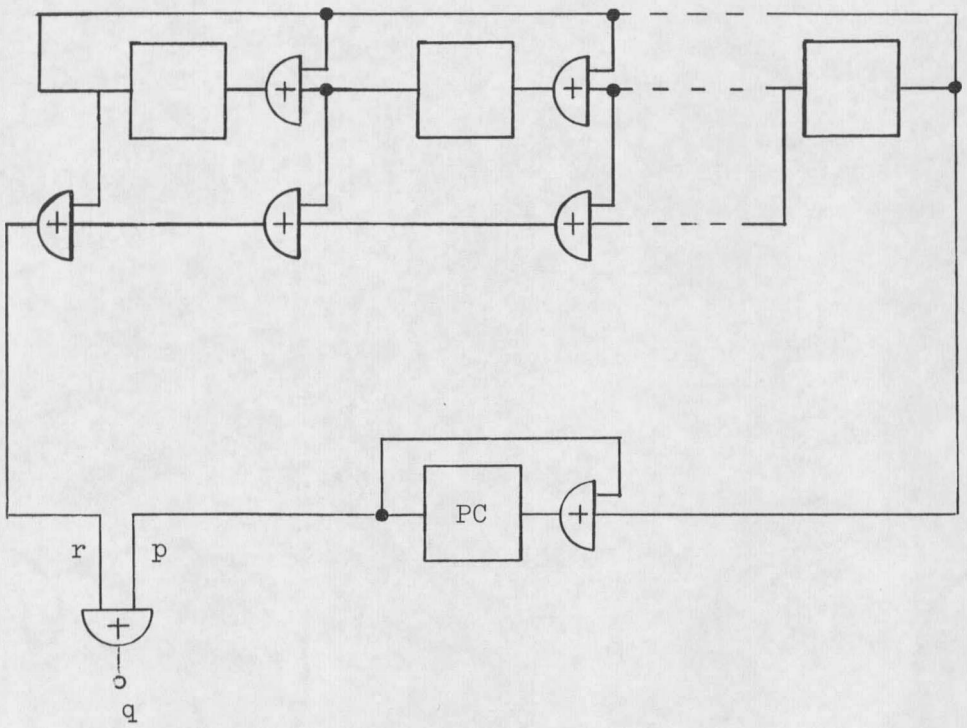


Figure 3.14. Fault Detection for Multiplication by α

$$b_p = b_0 + b_1 + b_2 + b_3.$$

When a_p and b_p are multiplied (ANDed), we have

$$a_p b_p = (a_0 + a_1 + a_2 + a_3) \cdot (b_0 + b_1 + b_2 + b_3)$$

$$= (a_0 b_0 + a_0 b_1 + a_0 b_2 + a_0 b_3 + a_1 b_0 + a_1 b_1 + a_1 b_2 + a_1 b_3 +$$

$$a_2 b_0 + a_2 b_1 + a_2 b_2 + a_2 b_3 + a_3 b_0 + a_3 b_1 + a_3 b_2 + a_3 b_3).$$

This simple AND produces the mod 2 sum of all possible cross products. Now, suppose we sum the c coefficients:

$$c_p = c_1 + c_2 + c_3 + c_4.$$

In Figure 3.12, the terms used as feedback appear an even number of times, and therefore we need a correction to $a_p \cdot b_p$ before it will have the same parity as c_p. Each time a term is used as feedback, it should be added to $a_p \cdot b_p$--a check is made each time we multiply by α. This applies to all three types of circuits. The sequential multiplier can be checked by forming $a_p \cdot b_p$ and attaching a parity bit to the accumulator. The sum of $a_p \cdot b_p$ and the accumulator parity bit after m cycles must be the same as the mod 2 sum of the accumulator bits.

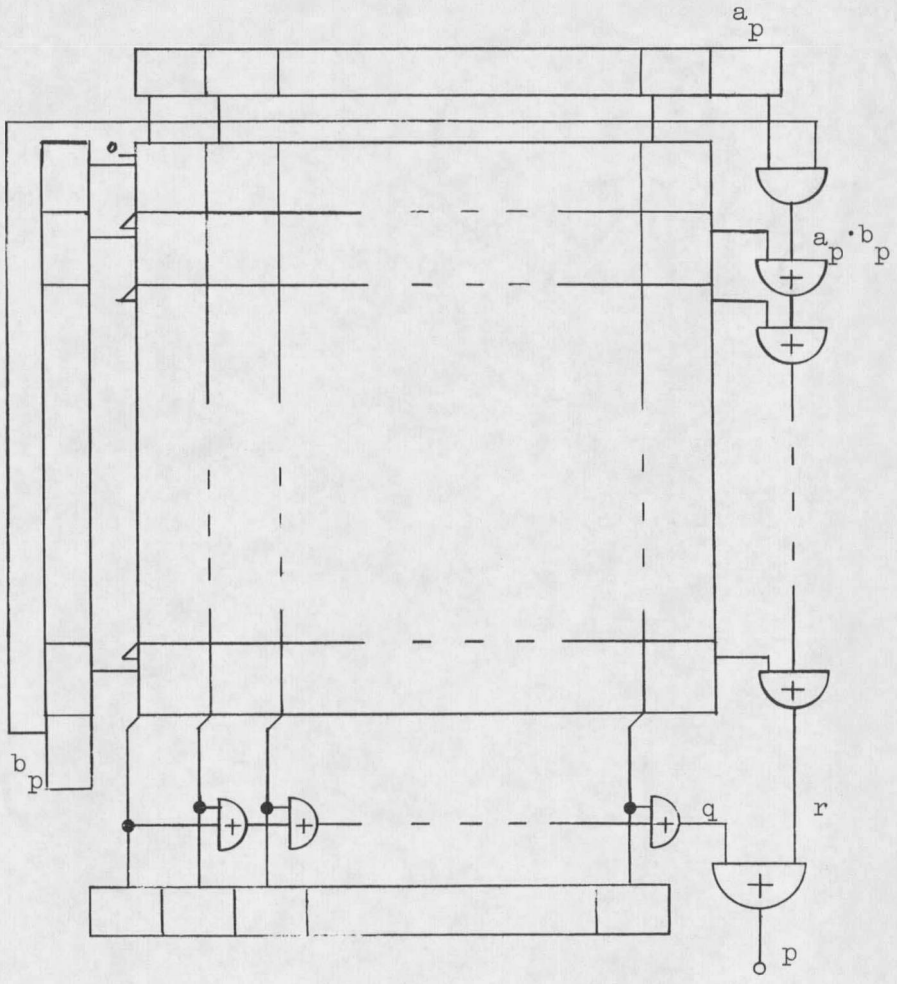


Figure 3.15. Fault Detection for the Cellular Multiplier

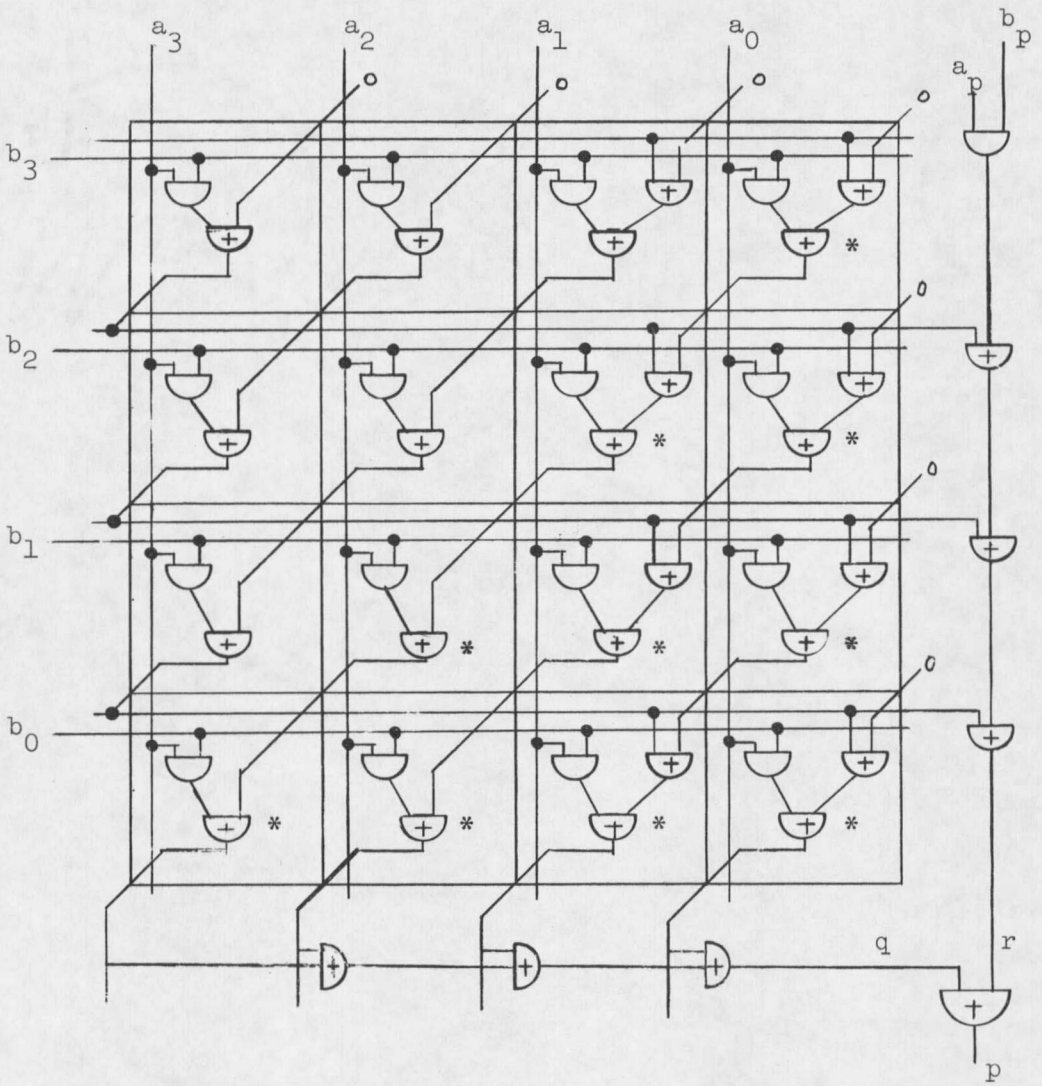


Figure 3.16. Fault Detection for GF(2^4)

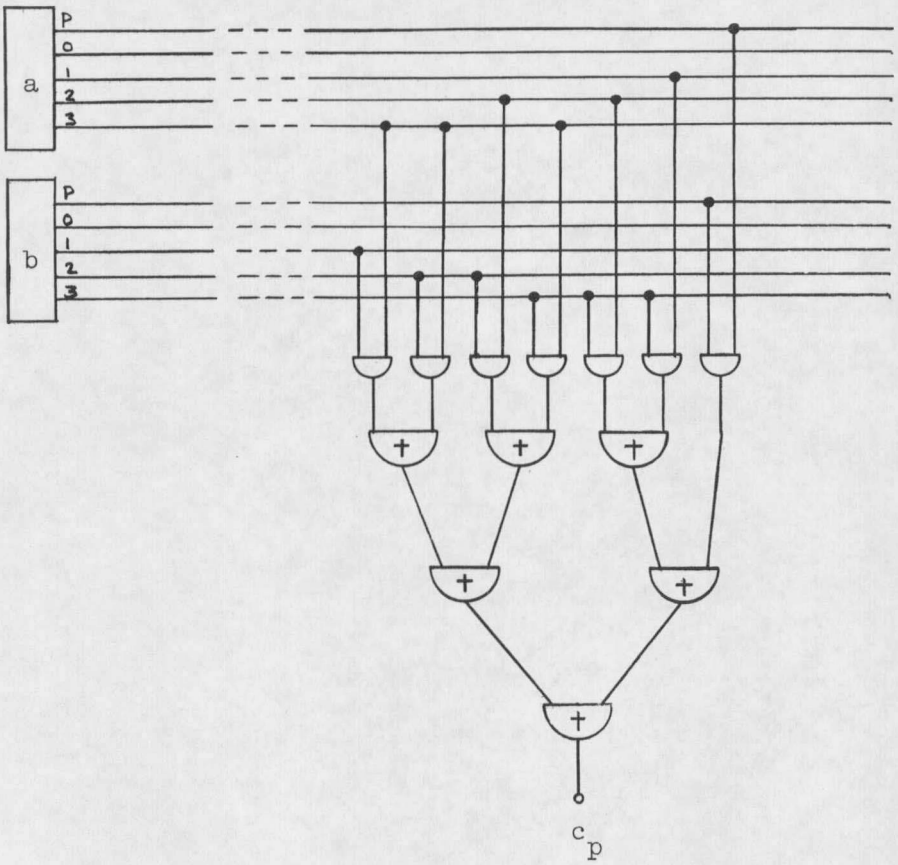


Figure 3.17. Fault Detection for the Fast Multiplier

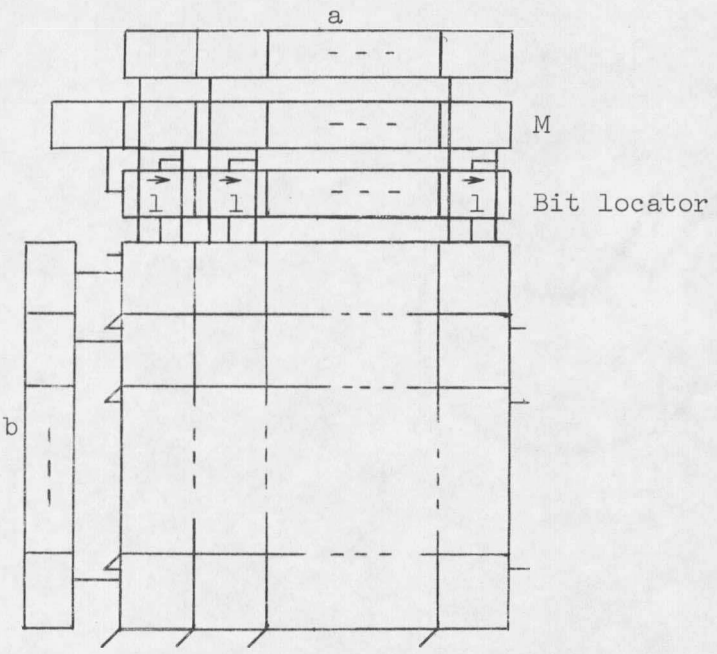
Figure 3.15 shows the fault detector circuitry in the cellular array. A mod 2 sum (r) is taken of all feedback terms and added to $a_p \cdot b_p$. This should be identical to the mod 2 sum of the outputs (q)--p should be 0.

In order to see how this works, consider Figure 3.16. This is a multiplier for GF(2^4) modulo x^4+x+1. If a failure occurs in a cell denoted by *, only one bit will be incorrect in the output. If a failure occurs in any other cell, the parity of the feedback sum r will be changed, but the parity of q will be unchanged since the fault is fed back into two *'d cells.

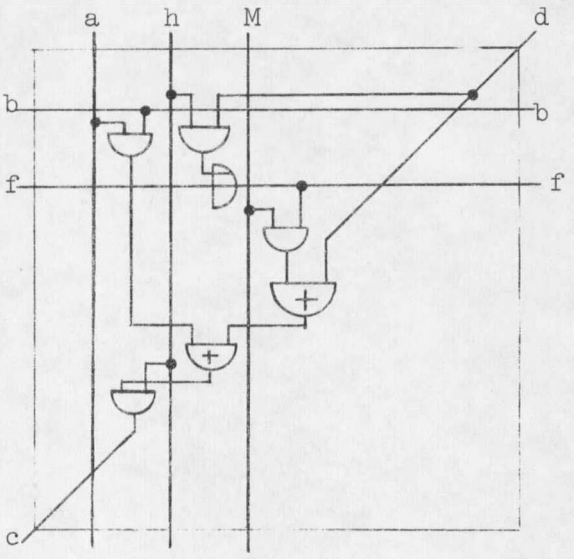
A similar scheme may be used to test the fast multiplier. All terms used as feedback are added to $a_p \cdot b_p$. This should yield the same parity as the sum of the c outputs. The circuit is shown in Figure 3.17.

3.5.5 Generalization of Multipliers

The cellular array multiplier is quite easily modified to allow operation over a number of different fields. Figure 3.18 (a) shows the interconnection and layout of the generalized array. The irreducible polynomial is placed on the M lines, right justified, so that the highest order terms are toward the left. All m+1 coefficients must be used. For instance, x^4+x+1 would be written 00$\cdots$010011. The multiplicand is right justified on the a inputs with high order terms toward the left of the array, and the multiplier is similarly justified on the b inputs



(a)



(b)

Figure 3.18. Generalized Multiplier Array

with high order terms toward the top. The bit locator circuit on the M lines finds the high order bit of the field primitive polynomial and selects the feedback cells. Figure 3.18 (b) shows a particular cell. The multiplier and multiplicand bits enter on the b and a lines, respectively. The M lines feed the primitive polynomial down. The f lines supply the feedback. The value of the f line is added only if the vertical M line is 1.

The h lines come from the bit locator cascade, indicating the high order term of the primitive polynomial. If the h line is 1 in a particular cell, the diagonal output of data is blocked, and transferred to the f line to provide the feedback.

This circuit requires all field elements to have zeros in the unused high order digits. The fault detection circuitry is practically unchanged.

3.6 Multiplicative Inversion in a Galois Field

The Berlekamp algorithm requires computation of the inverse of a field element during each of t iterations. Since the operating speed of the decoder is dependent upon the time required to perform the multiplicative inverse, a circuit is needed which will compute the inverse as rapidly as possible.

3.6.1 The Sequential Inverter

Berlekamp (1) has proposed a sequential inverter which requires approximately 4m clock cycles to calculate the inverse. Although it is

relatively slow, the circuit has the advantage of being able to operate over any field.

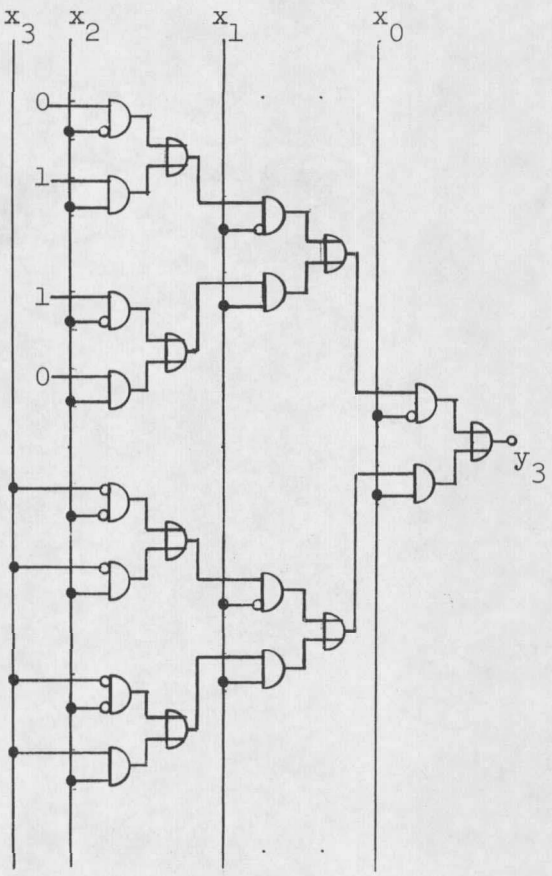
3.6.2 The Combinational Inverter

Probably the fastest way to do multiplicative inversion is to construct an m-input m-output combinational network. Bartee and Schneider (5) have proposed a technique for realizing a combinational inverter. The technique is quite difficult to perform by hand, but a computer program would be useful in designing inverters by this method. The problem may also be solved using ordinary multiple output function techniques (7), but computer solution is still required.

Figure 3.19 (a) shows the truth table for the multiplicative inversion function over $GF(2^4)$ with irreducible polynomial x^4+x+1, and Figure 3.19 (b) shows the circuit which realizes the functions for the y_3 digit of the inverse. Each pair of entries in the truth table specifies the input of an AND gate for the circuit. The input is inverted if the pair of entries is the complement of x_0. If the entries for y_3 are the same as x_0, the input is not inverted, and if the entries are both 0 or both 1, the AND gate input is 0 or 1, respectively. The second input to the AND gate is dependent upon x_1. If x_1 is 0, the input is inverted. If x_1 is 1, the input is not inverted. The variables x_2 and x_3 are introduced in succeeding levels to the right so that each input variable selects a block of the truth table. That is, the x_3 variable selects the top or bottom half of the truth table, the x_2 variable selects the top

x_3	x_2	x_1	x_0	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	1	0	0	1
0	0	1	1	1	1	1	0
0	1	0	0	1	1	0	1
0	1	0	1	1	0	1	1
0	1	1	0	0	1	1	1
0	1	1	1	0	1	1	0
1	0	0	0	1	1	1	1
1	0	0	1	0	0	1	0
1	0	1	0	1	1	0	0
1	0	1	1	0	1	0	1
1	1	0	0	1	0	1	0
1	1	0	1	0	1	0	0
1	1	1	0	0	0	1	1
1	1	1	1	1	0	0	0

(a)



(b)

Figure 3.19. Multiplicative Inversion

or bottom half of the previously selected half, and x_1 and x_0 compute the correct truth table entry. The y_3 variable then appears on the right hand output of the circuit.

The circuits for the y_2, y_1, and y_0 outputs may be constructed independently or may use intermediate outputs of the first network. Minnick (6) has shown techniques for minimizing these types of circuits; however, the number of cross connections between various parts of the circuit may increase as the number of gates is reduced. The speed of the circuit is proportional to m-1 for $GF(2^m)$.

3.6.3 The Read Only Memory as an Inverter

The decreasing cost and rapid access time of read only memories make such devices pratical for use as a multiplicative inverter. All field elements are represented as m-bit numbers and the elements are used as addresses to a random access memory location containing the m-bit inverse element, as described in Section 2.2.4.

3.6.4 Error Checking of Multiplicative Inverters

The easiest way to perform a failure-detection test on the inverter is to multiply the element by its inverse. If the product is not the multiplicative identity, a failure has occurred. If a dedicated multiplier is available, no increase in processing time will result from this check since the inverter can caluclate the next inverse while the multiplier calculates the last check.

As a check on the combinational circuit, an extra column may be added to the truth table so that all function outputs have even parity. Then if a result is incorrect because of a single gate failure, the parity of the inverse will be odd instead of even.

The random access memory may be checked by storing an extra parity bit so that each inverse has even parity.

3.7 System Block Diagram

Figure 2.7 shows the system block diagram. The buffer shown at the bottom of the diagram is a sequential shift register which contains the received words currently being processed. The slave processors contain two parity check calculators, a Chien search calculator, and circuitry required to perform the Berlekamp algorithm. The master processor contains the multiplicative inverter, an accumulation register, the system controller, and the algorithm decision circuitry. The system controller issues clock pulses and control signal to all slaves, and advances the words in the buffer.

3.8 Processor Interconnection

Figure 3.20 shows the external connections of the slave and the master processors. Each slave receives identical control and clocking signals via the I lines. The contents of the accumulator in the master processor are presented to each slave processor on the MA lines. The incoming digits are transmitted on the COD line. The slaves also communicate results back to the master processor. The AC lines return a

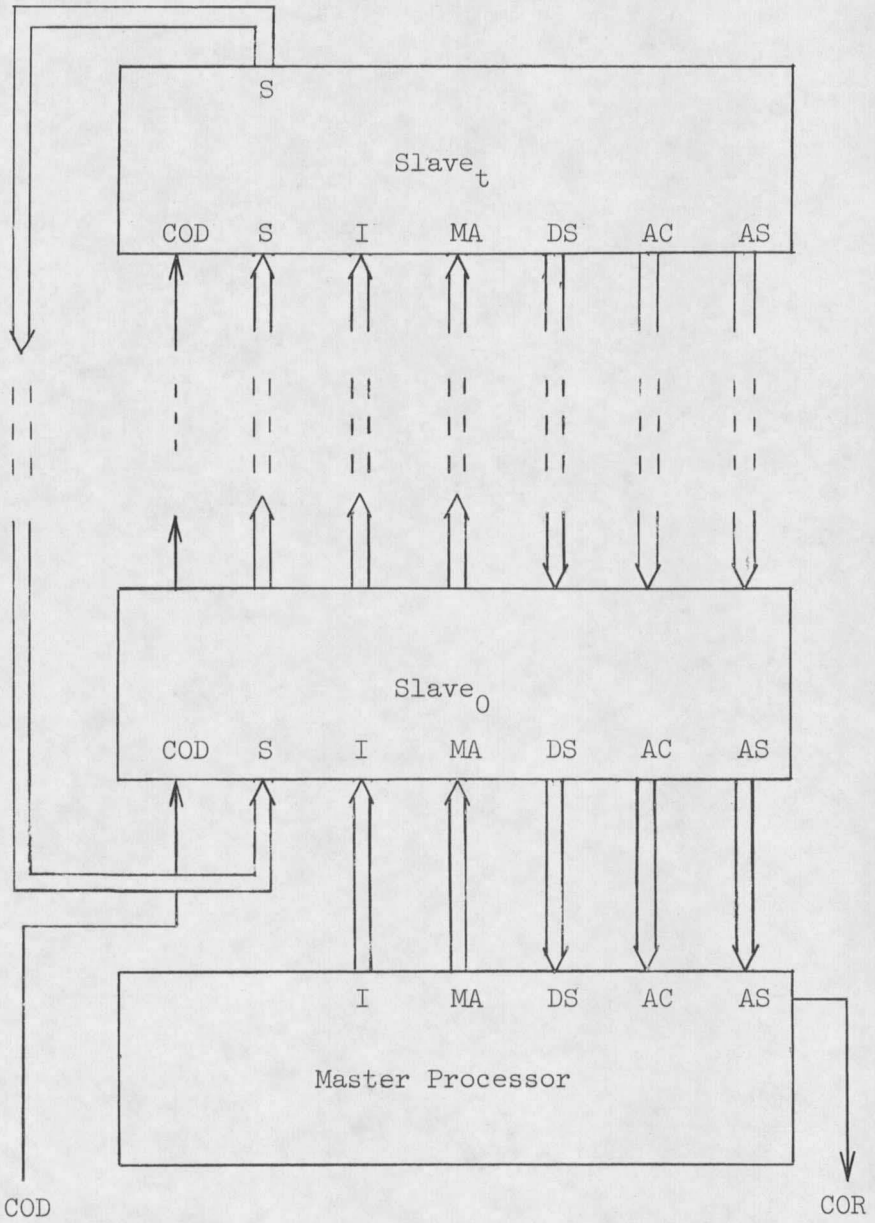


Figure 3.20. Processor Interconnection

sum of elements, one addend from each slave processor, to the master processor. The AS lines return the sum of elements required by the Chien search. The DS lines communicate the degree of the trial error locating polynomial to the master processor. The COR line is the logical NOR of the AS lines, and if it is 1, the rightmost bit in the buffer is complemented.

3.9 The Slave Processors

Figure 3.21 shows the block diagram of a slave processor. The PCU1 and PCU2 units calculate the two syndrome digits required by the Berlekamp algorithm.

Most of the circuitry is used by the Berlekamp algorithm; the SRS is the slave register set, and contains five data registers and two shifting registers. The SCU provides clocking for the SRS and also computes the degree of the error locating polynomial stored in the slaves. The AU is the Galois field arithmetic unit which performs the field multiplication and addition operations. There are also two Galois field adders to introduce addends into the AS and AC lines. The IRN, or internal routing network, routes data from other registers, the arithmetic unit, and the parity check units into the slave registers, the AC adder, and the Chien search units. The CSU or Chien search unit makes the computations necessary for the Chien search. It supplies an addend to the AS line accumulator.

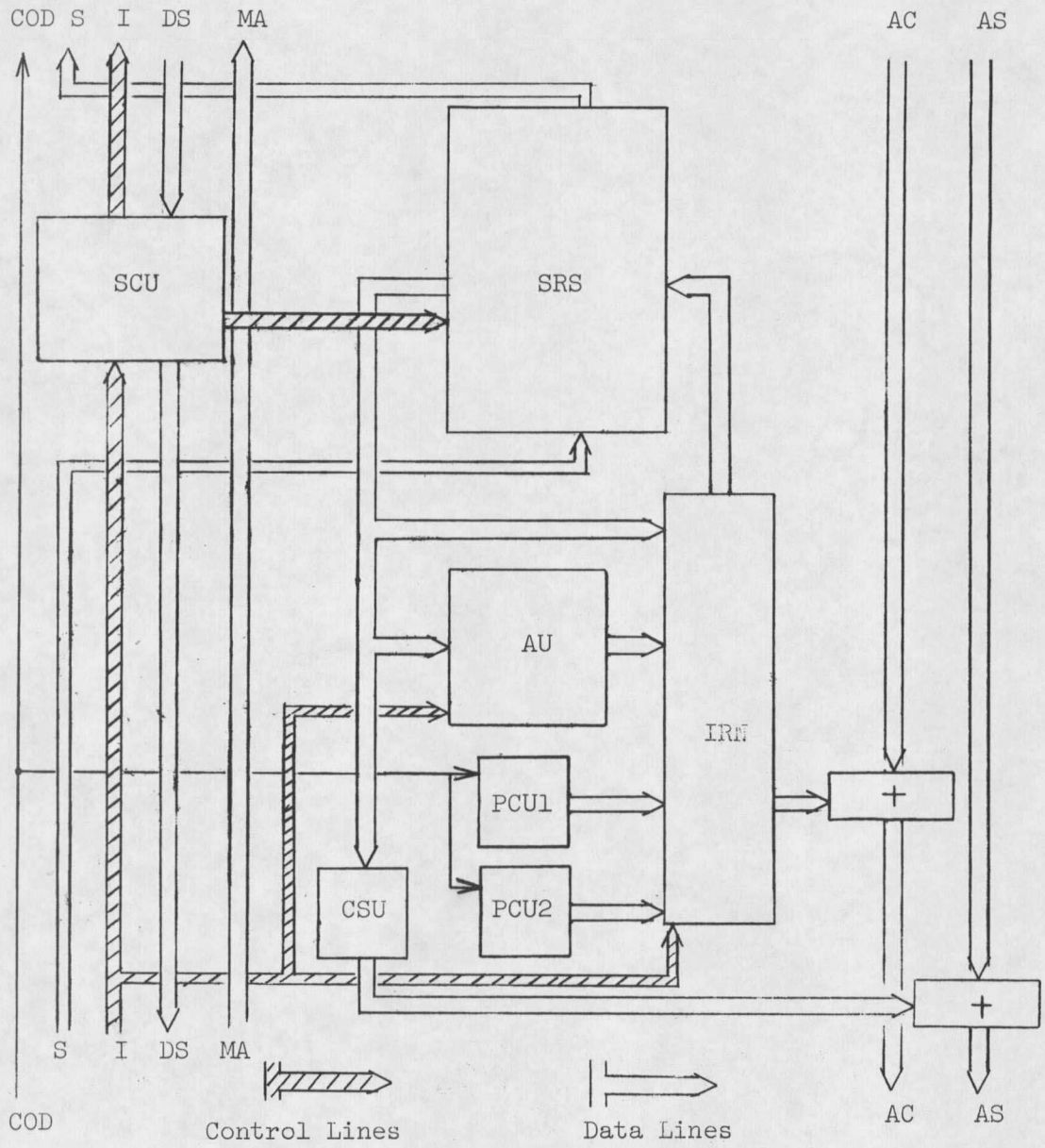


Figure 3.21. Block Diagram of the Slave Processor

3.9.1 The Parity Check Units

These two units (PCU1 and PCU2) are identical, except for external connections and internal constants. Figure 3.22 shows a diagram of $PCU1_i$, in slave i. The $C1_i$ register contains α^{t+2-i} so PCU1 will compute S_{t+2-i}. The constant in $C2_i$ is α^{2t+3-i} (for PCU2). The parity check units in all slaves execute the parallel processor parity check algorithm.

During each clock cycle, a different received digit is placed on the COD line. The parity check unit multiplies the contents of the PC1 register by the contents of the C1 register and adds the unity field element to the product if COD=1. Normally, this result is then stored back in PC1. However, during the processing of the last bit of every received word, the INIT line is 1 and the system is reinitialized. The parity check for one received word has been computed and appears on the output lines of PCU1. The INIT line causes the PC1 register to be cleared to zero. The parity check which has been computed is transthrough the IRN to the appropriate slave register in the SRS.

3.9.2 The Chien Search Unit

The Chien search unit (CSU) is shown in Figure 3.23. When the INIT line is 1 (during the processing of the last bit of a received word), the CSR_i register is loaded with the new error locating polynomial. In the slave i the C3 register holds α^i. During every clock cycle for each received digit, the contents of CSR_i and $C3_i$ are multiplied and passed to the AS accumulator inputs. The product is also stored in CSR_i. When

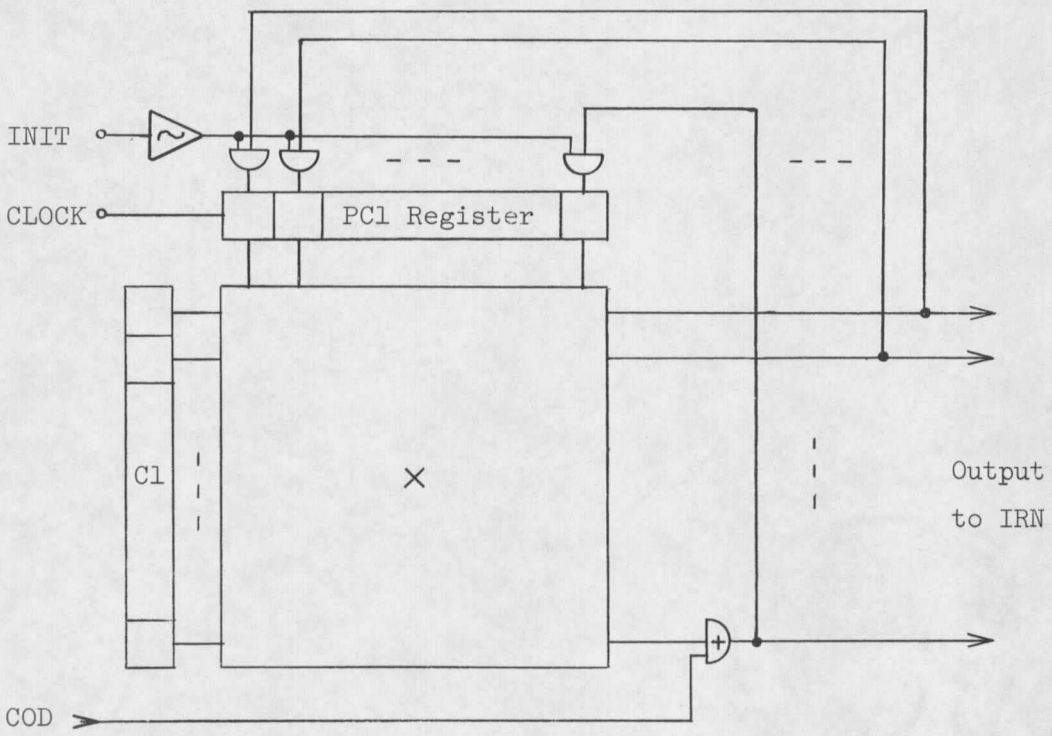


Figure 3.22. Parity Check Unit

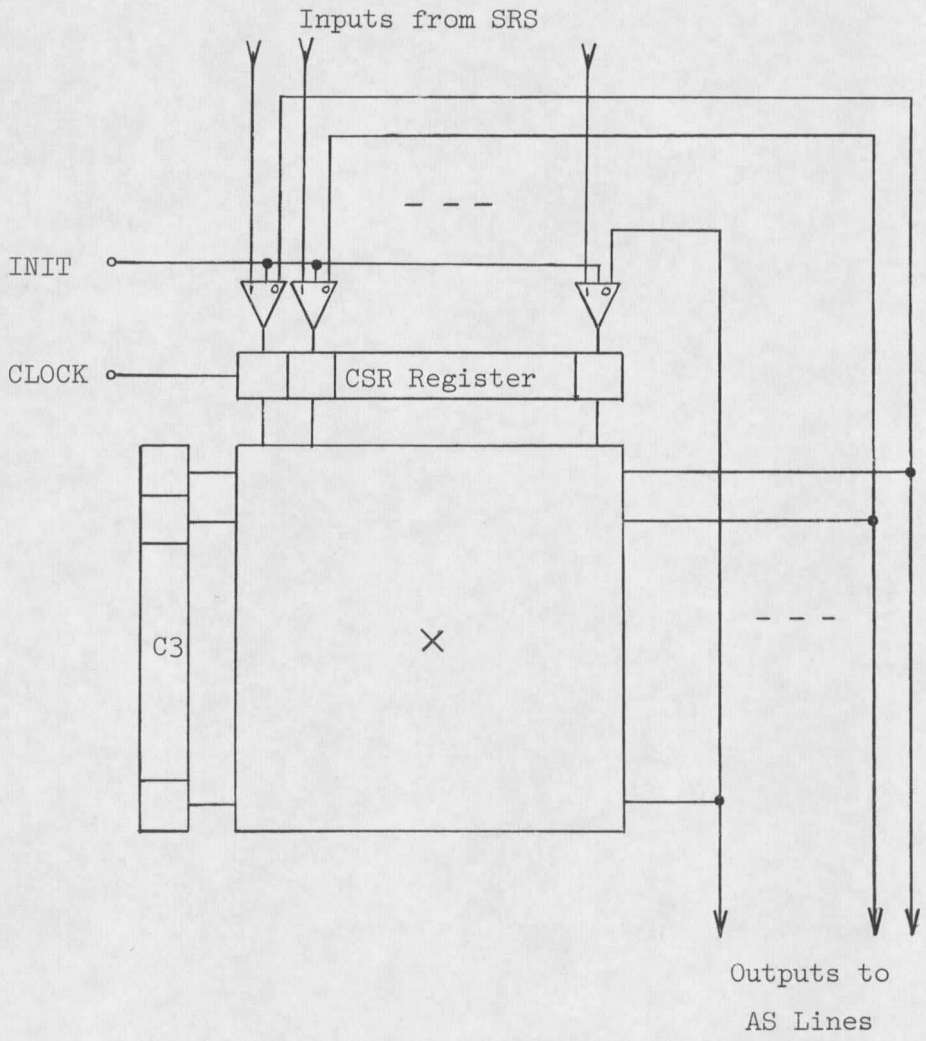


Figure 3.23. Chien Search Unit

INIT = 1, the product is still formed and sent to the accumulators, but the register receives the new error locating polynomial instead of the product.

3.9.3 The Slave Register Set

The SRS is shown in Figure 3.24. It contains five m-bit data storage registers designated R0, R1, R2, R3, R4, and two shifting storage registers, SR0 and SR1. All of these registers are used by the Berlekamp algorithm for the computation of the error locating polynomial. All registers have parallel inputs and outputs. In addition, SR0 and SR1 have serial inputs and outputs. Each of the seven registers has a clock input for parallel data transfer designated CR0, CR1, CR2, CR3, CR4, CSR0 and CSR1. SR0 and SR1 also have clock inputs for the sequential transfer of data designated CS0 and CS1.

$SR0_i$ must shift data to slave i+1, but $SR1_i$ must shift data to slave i+2. Therefore, three shift lines enter each slave. One connects $SR0_i$ to $SR0_{i-1}$ and $SR0_{i+1}$, while the other two lines connect $SR1_i$ to $SR1_{i-2}$ and $SR1_{i+2}$.

The ZR2 line is 1 if the contents of R2 are non zero. This is computed by a set of OR gates connected to R2. The information on ZR2 is used to determine the degree of the trial error locating polynomial stored in the R2 registers of the slaves.

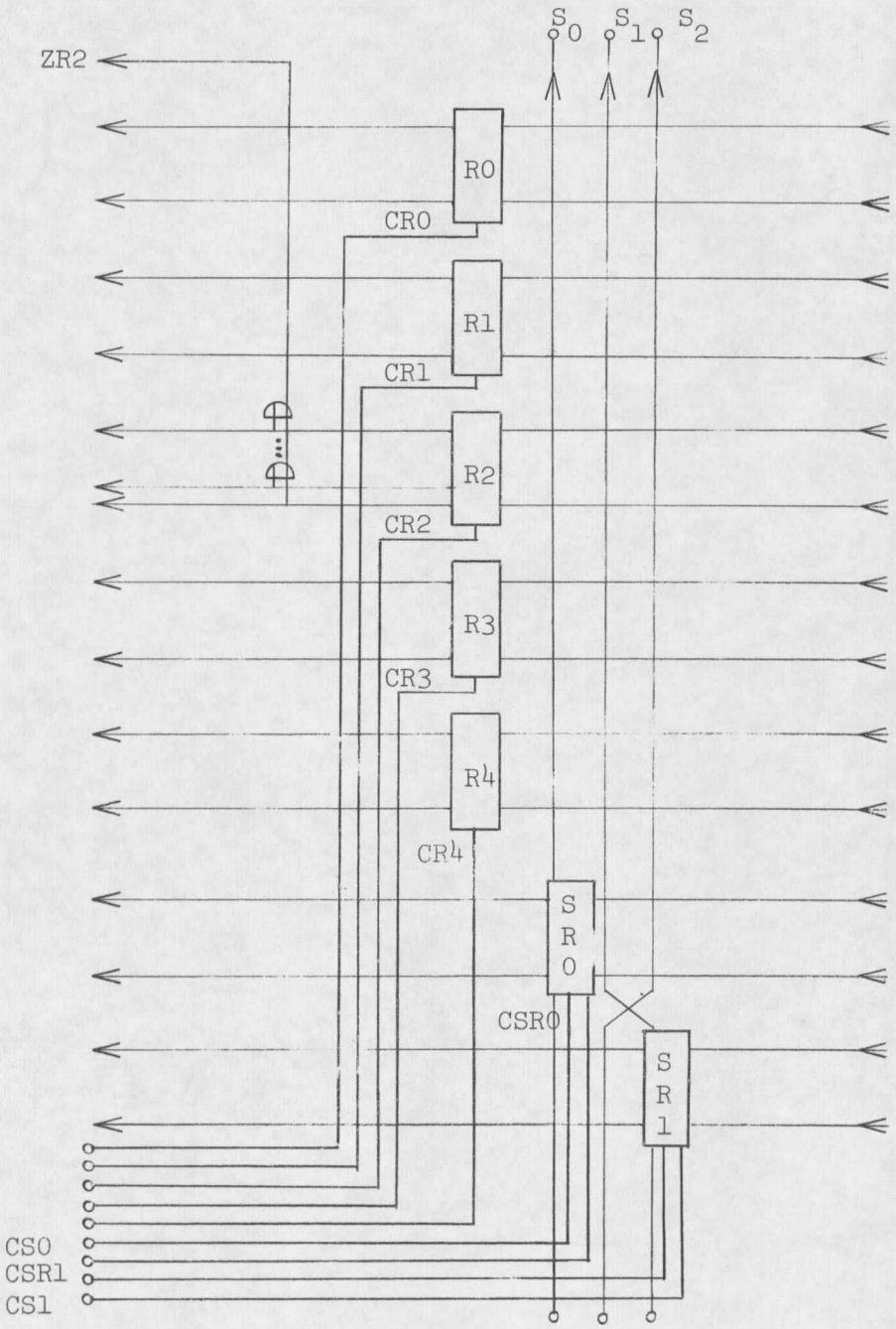


Figure 3.24. Slave Register Set

3.9.4 The Arithmetic Unit

This unit contains a Galois field multiplier and adder, and several input selection units, as shown in Figure 3.25. The register inputs are m-bit parallel lines from registers R1, R3, SR0 and R2, and the MA lines. There are also control inputs from the I_0 and I_1 control lines to govern which inputs are used. Table 3.1 shows the calculation during each cycle.

Cycle	Product
1	R1*R2
2	(R3*MA)+R2
3	SR0*MA

Table 3.I. Multiplier Operations

Three outputs are provided, one for each of the three possible results computed.

3.9.5 Internal Routing Network

Table 3.2 shows the register transfers which occur during each clock cycle--IC designates the initialization cycle, C1 the first clock cycle, C2 the second, and C3 the third. The internal routing network is shown in Figure 3.26. The single lines for input and output represent the m parallel lines actually required. The inputs are all in parallel from the registers, the arithmetic unit and the parity check units. The ICON lines are zero, except in slave 0, where they contain the multiplicative identity for the field. The PC1 and PC2 lines hold

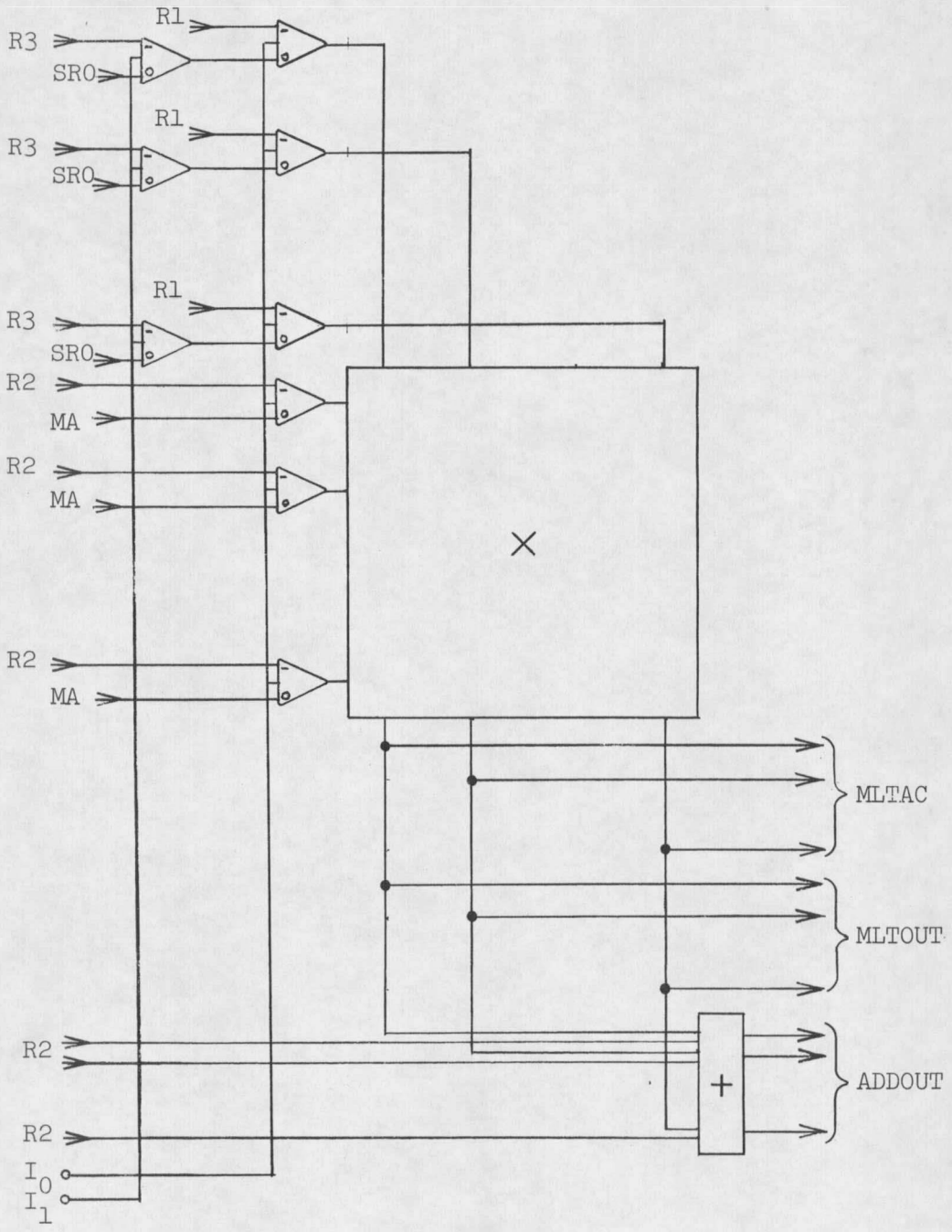


Figure 3.25. Arithmetic Unit

Cycle:	IC	C1	C2	C3	C3	C3
I_0	0	1	0	0	0	0
I_1	0	0	1	0	0	0
I_2	0	0	0	1	1	1
RS	0	0	0	0	1	0
SC	0	0	0	0	0	1
Input Register	----------Outputs to Registers----------					
R0		SR1				R1
R1			SR1			
R2	CSR	SR0				
R4					SR0,SR1	
SR0		R3				
SR1		R4	R0	R1	R1	R0
MLTAC		ACIN				
MLTOUT				SR0,SR1		SR0,SR1
ADDOUT			R2			
ICON	R2 SR0,SR1					
PC1	R0					
PC2	R1					

Table 3.2. Routing Schedule

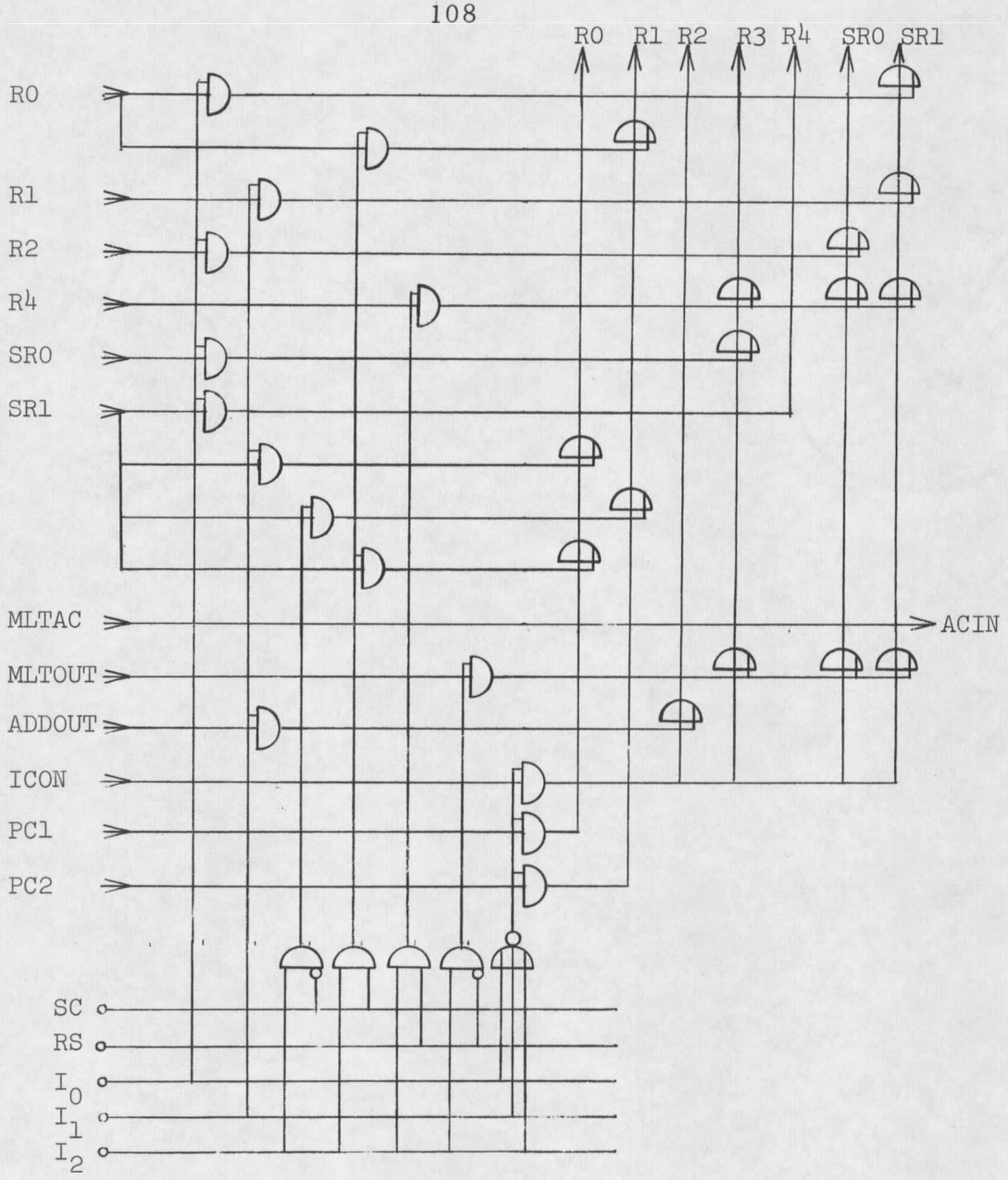


Figure 3.26. Internal Routing Network

the parity checks, and are used only during reinitialization.

Each register in the slave register set receives its input from the IRN, and there is an IRN output ACIN which goes to the AC accumulator inputs.

There are five control lines. The I_0, I_1 and I_2 lines determine which cycle is being executed, and during cycle 3, the RS line is used to indicate whether the multiplier output or the value in R4 should be used. The SC line is also a control input. This line is a 1 if the slave number is 0 or 1, and is used to product the circular flow required for the data contained in the two parity check storage registers R0 and R1.

3.9.6 The Slave Control Unit

Figure 3.27 shows the block diagram of the controller. The DSU (Degree Sigma Unit) is used to determine the degree of the polynomial stored in the R2 registers, and the RSU (Register Selection Unit) is used to determine which of the seven registers in the SRS are to be clocked during any given clock cycle.

The DSU is shown if Figure 3.28. The D register holds an integer i between 0 and t, for slave i. The bit locator unit in the DSU receives the OR sum of all bits in the R2 register on the ZR2 line, and if the register in slave i holds the highest nonzero term of the polynomial, the value i contained in the D register is gated onto the DS lines to be used by the master processor. The DSU produces a

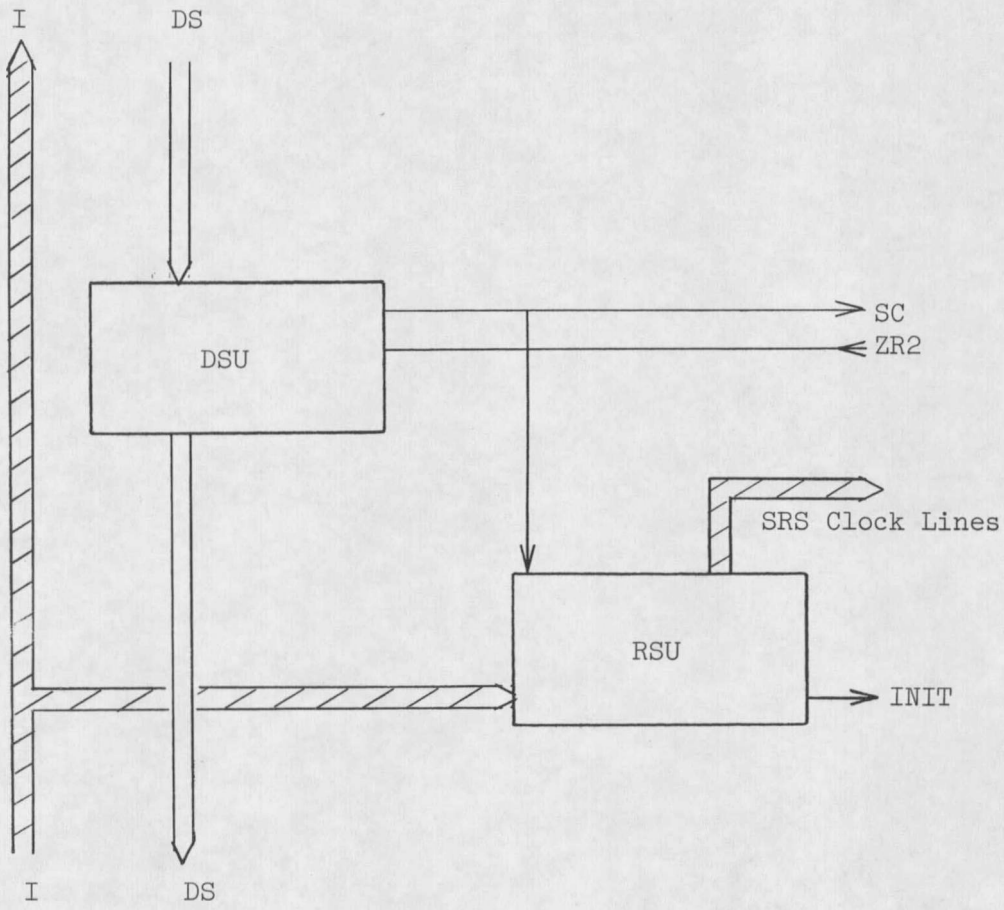


Figure 3.27. Slave Control Unit

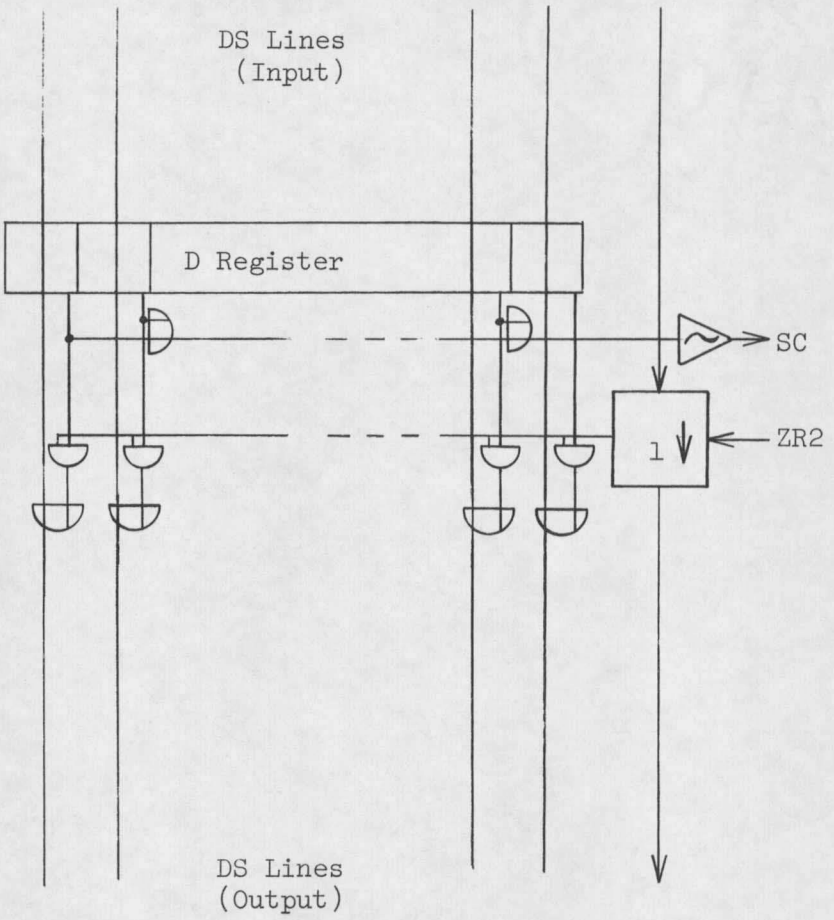


Figure 3.28. Degree Sigma Unit

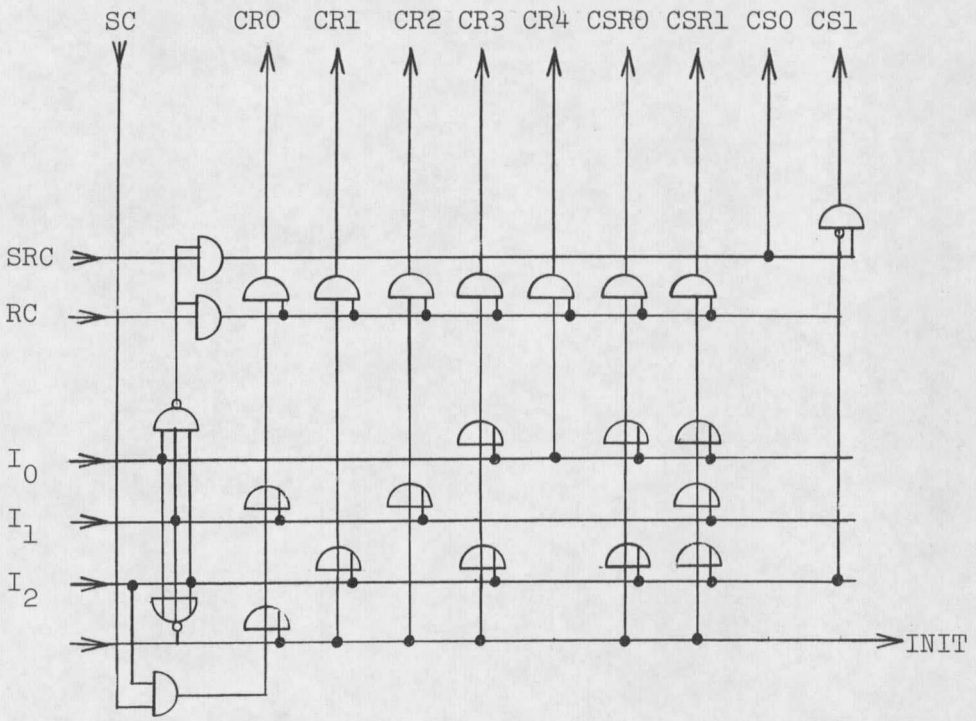


Figure 3.29. Register Selection Unit

1 output on the SC line if the slave number is zero or one.

The RSU in Figure 3.29 supplies the clock signal to the seven registers in the SRS. The I lines determine which of the registers are to be clocked, and the SC line determines variations to provide the circular flow of data in R0 and R1. If lines I_0-I_2 are all 1, no registers are clocked. If these lines are all 0 and the INIT line is 1, the registers are clocked for reinitialization.

3.10 The Master Processor

A block diagram for the master processor is shown in Figure 3.30. The master processor has four functions: to accept and return the field element from the AC accumulator lines, to return the multiplicative inverse of that element, to make the necessary decisions for the Berlekamp algorithm, and to supply timing and sequencing information to the entire system.

The COR line is a NOR of the AS accumulator lines from the Chien search units. The COR line is 1 if the bit currently being shifted out of the buffer is to be corrected.

3.10.1 The Master Accumulator

The MA (Master Accumulator) is shown in Figure 3.31. It is basically a data register, and can accept field elements from either the AC lines during cycle 1, or the IVOUT (Inverter Output) lines during cycle 2. The input selection units determine which will be the input. The contents of the MA are available to all slave processors via the MA

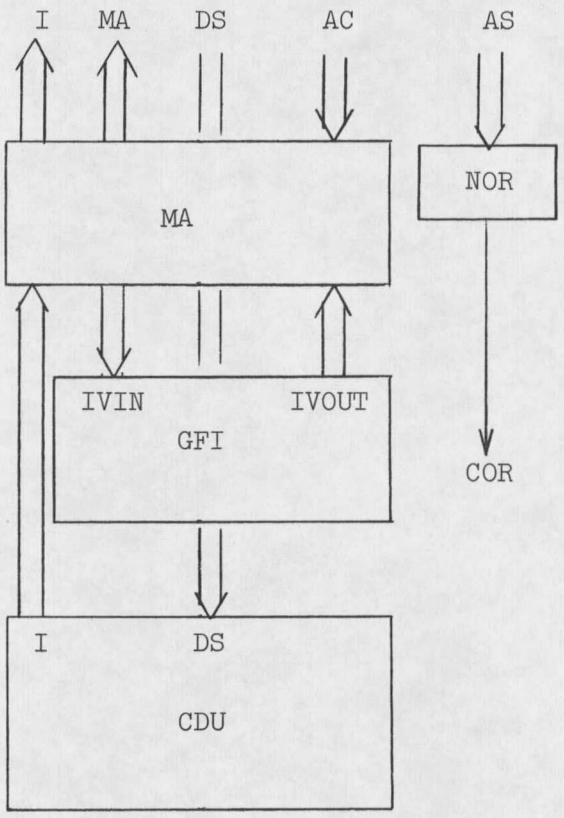


Figure 3.30. Block Diagram of the Master Processor

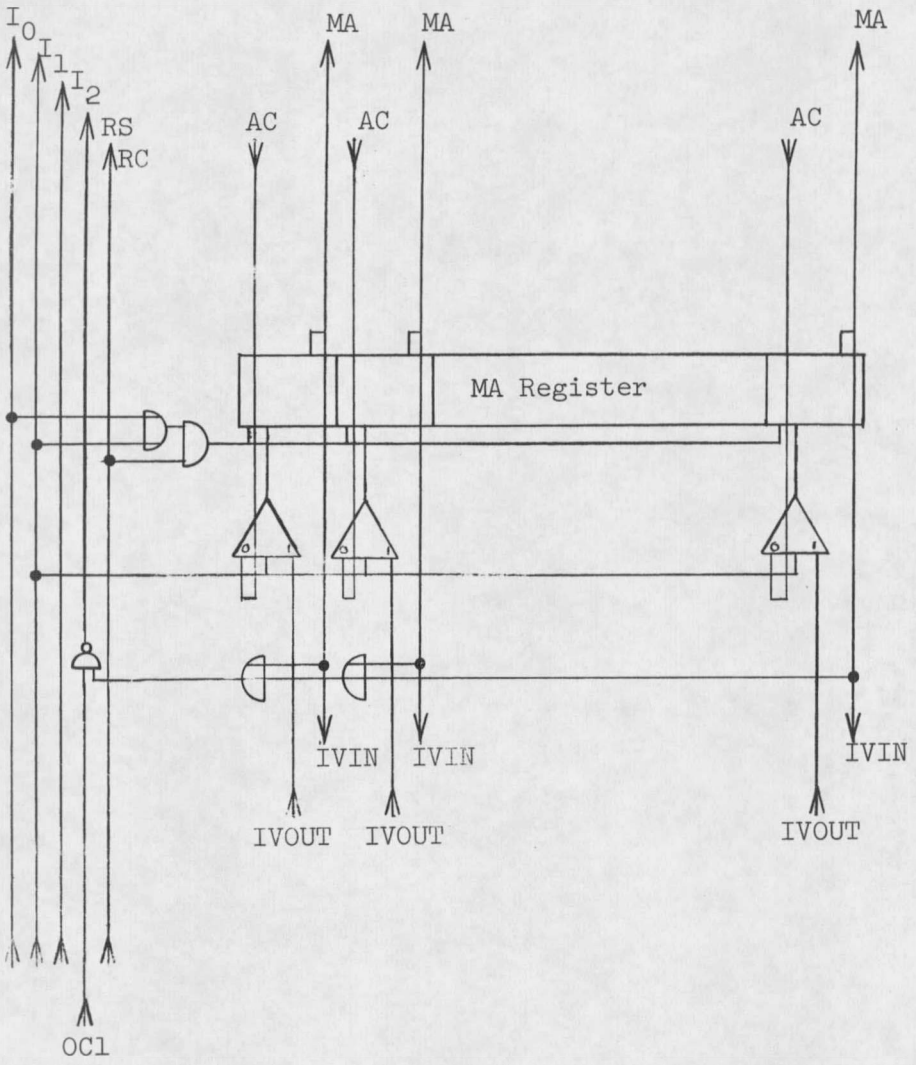


Figure 3.31. Master Accumulator

lines, and to the inverter via the IVIN lines. The master accumulator is clocked only during cycles 1 and 2.

The OC1 line from the central decision unit (CDU) is combined with the OR sum of the master processor to drive the RS line. This line makes the decision required by the Berlekamp algorithm.

3.10.2 The Control and Decision Unit

Figure 3.32 shows the circuitry required for the CDU (Control and Decision Unit). The modulo N counter supplies a 1 signal during the last clock cycle for each N bit received word. A length 2^m-1 word will require a 1 counter output every 2^m-1 clock cycles for system reinitialization.

The C1 comparator compares the degree of the trial error locating polynomial in the DS Register to the iteration count in the K register and places a 1 in the OC1 line if D is greater than K. The DS register receives its value from the DS lines during cycle 2. The value on these lines is the degree of the trial error locating polynomial as computed by the slave processors.

The value of the K register is incremented by 1 during cycle 1. During cycle 2, the contents of the K and T registers are compared by the C2 comparator. If the two are equal, the WT bit is set to 1. Then the I_0-I_2 lines will all be set to 1, causing the processors computing the Berlekamp algorithm to wait until the next parity check is ready. During reinitialization, the WT bit is set to 0.

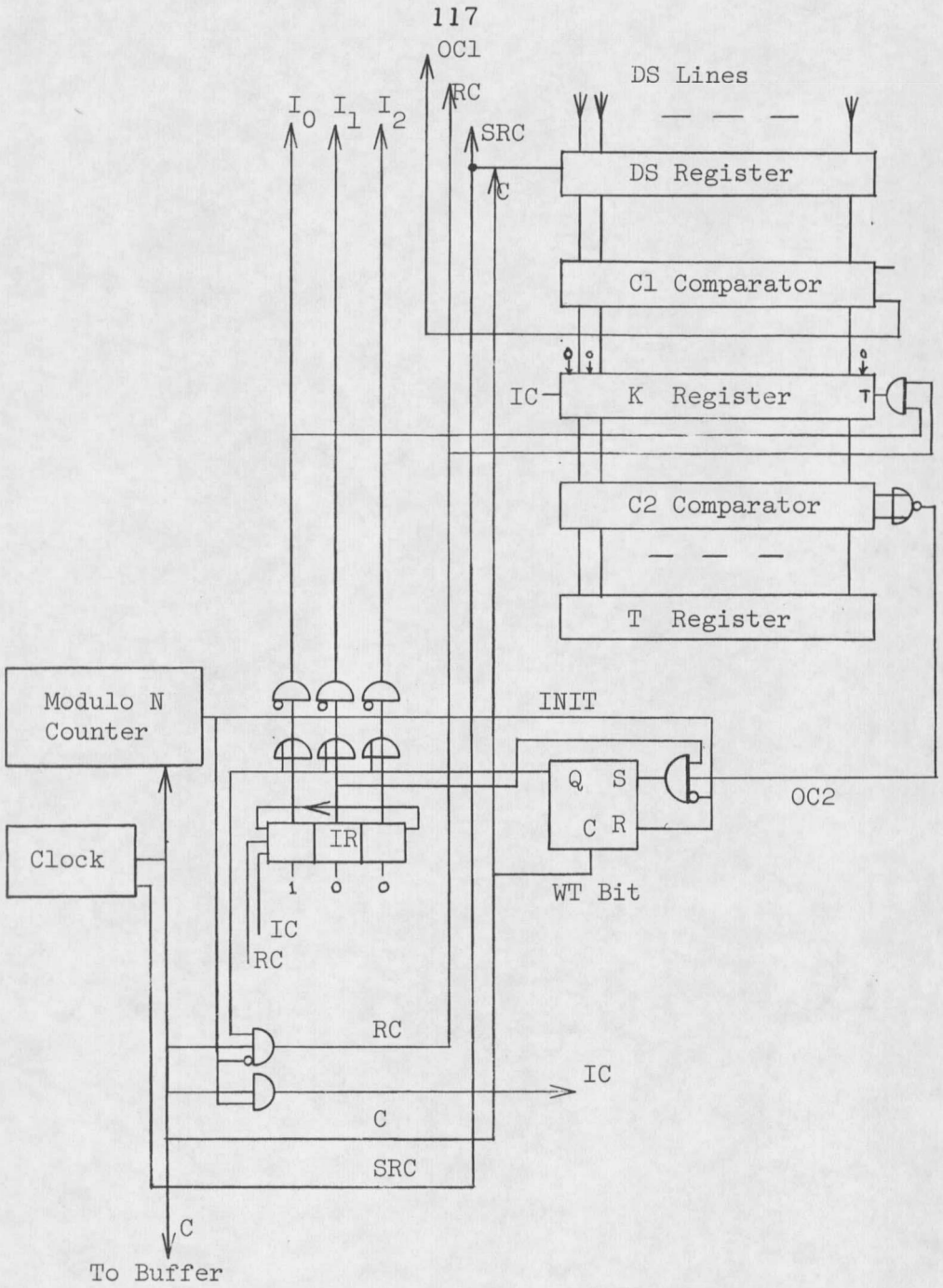


Figure 3.32. Central Decision Unit

The IR Register controls the cycling of the Berlekamp algorithm. A 100 output indicates cycle 1, 010 indicates cycle 2 and 001 indicates cycle 3. The register is initialized as 100.

The system clock is broken into three outputs; IC, RC, and C. The IC clock runs only during the initialization period, the RC clock during all other periods, except the wait period, and the C clock during all periods. The C and RC clocks are transmitted to the slaves and the C clock to the received word buffer. The SRC clock provides the clock pulses required by SR0 and SR1 to shift elements sequentially between slaves.

3.11 System Timing

Figure 3.38 shows the detailed timing diagram. When viewed from the right side, vertical divisions indicate the clock pulses. Each horizontal line shows the storage activity of a particular register, and the value written on the line during a particular cycle indicates the register or value which is stored. During the first part of each cycle (excluding the initialization cycle) SR0 and SR1 sequentially shift data between slaves. The shifted value is the input to the appropriate register during that same cycle. The contents of all other registers are unchanged for the duration of the cycle, and the new contents are stored at the end of the cycle.

There are N machine cycles for an N-bit codeword. During each cycle, the parity check and Chien search calculators operate. They

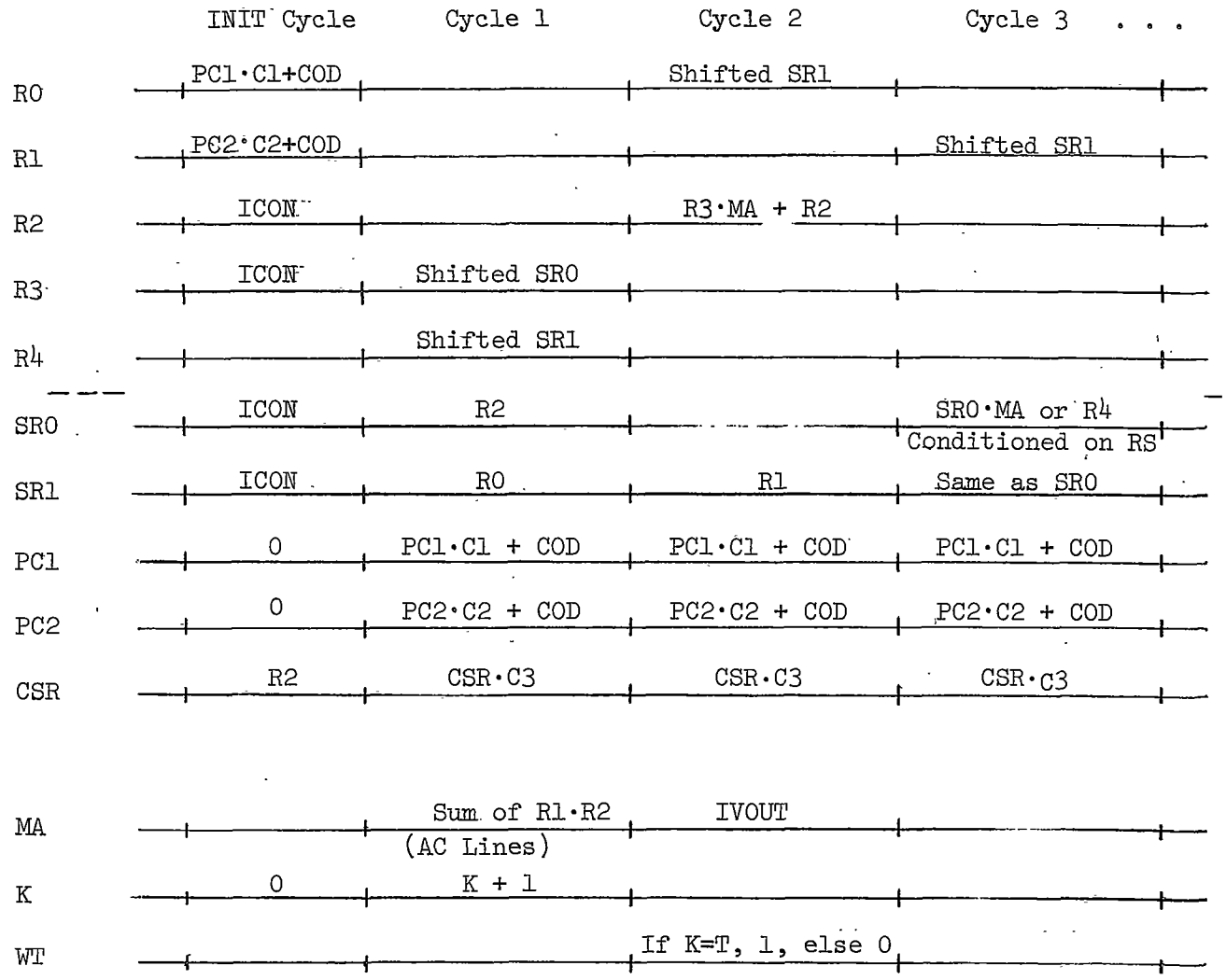


Figure 3.33. System Timing Diagram

reinitialize their accumulators while processing the N^{th} codebit.

At any given time, there are three words in the buffer. The parity checks are computed for the first, the Berlekamp algorithm for the second, and the Chien search for the third. The received digits are shifted sequentially across the buffer--each of the N clock cycles corresponds to a received digit.

While the parity check and Chien search processors run for N clock cycles, the processor which executes the Berlekamp algorithm must run for 3t clock cycles for each word: one reinitialization cycle and 3t-1 computation cycles, to correct t errors. If 3t=N, then the system can run synchronously without requiring a wait. If 3t is less than N, the processors performing the Berlekamp algorithm will be required to wait. The condition 3t greater than N is not allowed with this type of controller, since the parity check and Chien search units must run synchoronously to prevent data overrun of the received words.

The parity check and Chien search calculators could be run faster than the Berlekamp algorithm, so that 3t is approximately equal to N. Enough time must be allowed so that the new error locating polynomial is in R2 during initialization.

During each cycle of the Berlekamp algorithm, a multiplication is performed. At the same time, elements are being shifted between slaves under control of the SRC clock. If the sequential shift time is

two or more times as long as the multiply time, the sequential registers should shift more than one bit during each SRC clock cycle. This requires more lines between slaves, but allows processing to proceed at twice the rate.

The longest operation is the multiply and addition by the AC accumulator lines. For m=10, with t slaves, this operation would take about 300+15t nanoseconds using TTL logic. For t=100, 1800 ns are required. Including routing network and line delays, the minimum cycle time is perhaps 2 microseconds. This means we could process a length 1023 code and correct 100 errors at a rate of about one half megabit per second. If the fast multiplier circuit is used, and if more efficient means are used for adding the AC inputs, this might be reduced to about 480 ns, or a rate of 2 megabits per second. If faster logic is used, say MECL II or MECL III (Motorola Emitter Coupled Logic), these times might be cut in 1/3, allowing a rate of 6 megabits per second or better. The correction of 100 errors is rather extreme, since the information rate would be very low.

3.12 Fault Detection

A fault detection circuit for the multiplier was discussed in Section 3.5.4. The single parity check bit was used to detect a single bit failure. This method of error detection can be extended to detect failures in other parts of the processor.

First consider fault detection in the sequential shift registers, SR0 and SR1. A single parity bit will not detect single bit failure. Suppose a flip flop fails--either it refuses to change states, or it refuses to change the state of the next flip flop in sequence. This creates a burst of errors, even though only one flip flop has failed. For this reason, error detection during each shift cycle is not practical. A solution is to run an independent test of the shift registers during the reinitialization process for each codeword. Suppose each register has an even number of bits and contains the sequence 101010... ..10. By making only two shifts a failure of any flip flop can be detected. During the first shift, each flip flop must change its state properly, and also change the state of the next flip flop in sequence. If this does not happen, the pattern will be destroyed. During the second shift, each flip flop must make the opposite transition, and again, a failure will cause the pattern to be destroyed. Figure 3.34 shows the circuitry which will perform the check. After loading the proper sequence of bits, x and y outputs should both be 1. After the first shift, the x and y outputs are meaningless; after the second shift, they should both be 1. The shift must be circular.

If the sequential shift registers have been checked and found to be fault-free, the problem is reduced to fault detection inside each slave processor. If a single parity check bit is carried through all computation, a single bit failure can be detected. The only remaining

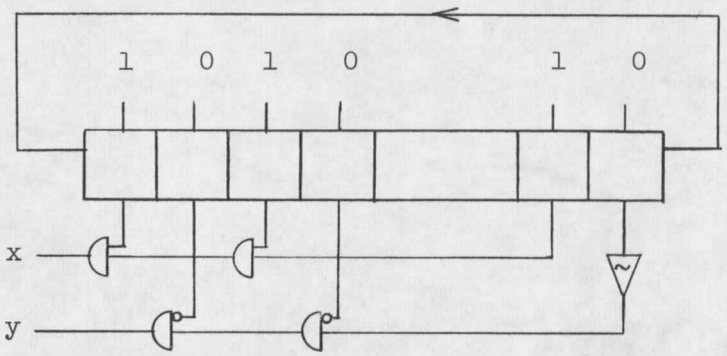


Figure 3.34. Fault Detection for a Sequential Shift Register

problem is to determine where in the processor the parity check should be verified. In Error Detecting Logic for Digital Computers (10), the authors list three considerations for determining the location of the error detection logic:

1. The effect of error detection upon system performance.
2. The cost of the additional hardware.
3. The probablity of detection, should a fault occur.

The optimum points for fault detection appear to be the four multiplier arrays in each slave. If the detection is placed on the PC1 and PC2 registers in the parity check units, the fault detection time can be overlapped with the multiply time. The same is true in the Chien search unit. If the CSR register outputs are checked, any error will be detected during the next multiply cycle. Since all data flow in the Berlekamp algorithm must eventually pass through the arithmetic unit, the outputs of the adder and multiplier seem to be the logical points to check for faults. This detection time can be overlapped with the delay time of the internal routing network. By overlapping the detection time with other processor functions, the first consideration is satisfied. The checking is done with 2m+1 exclusive-or gates at each location, thus satisfying the second consideration. Finally, the structure of the algorithms force all data to flow through the multipliers. The probability of fault detection is therefore high, and the third consideration is satisfied.

In the master processor, the parity of the MA register should be checked after the contents of the AC lines are stored. The MA register should again be checked after the inverse is loaded. The parity of the MA should be transmitted to all slaves via the MA lines. The DS lines should also have a single parity bit.

3.13 Fault Correction

When a fault is detected in a slave processor, the decoder must be stopped and the slave repaired by some means. If the number of slaves available is greater than the number of errors to be corrected, the slave processor may be deleted with some additional circuitry. First, each slave must have a bit which is 1 if the slave is active and 0 if the slave is passive. This bit should be set to 0 in the event of a slave processor failure. The sequential shift lines must be shunted around the slave, and the inputs to the AC and AS accumulators must be zero. Finally, some means for reloading the C1, C2, C3 and D registers in the remaining slaves must be provided. Since the contents of these registers in slave_{i+1} are simply related to the contents of the registers in slave_i, reloading can be done quite rapidly with additional circuitry and lines for each slave.

3.14 Operation over Other Fields

The decoder can be extended to operate over a number of fields. In general a means for loading the C1, C2 and C3 constants is needed like the one mentioned in section 3.12. The general multiplier array

is also needed, along with storage in the slave processors for the primitive field polynomial. The parity check and Chien search units which have been developed can be used over any field with the generalized multiplier circuit, and the processors need only have enough bit storage available to handle the largest field desired.

Due to the manner in which the parity checks are computed, the number of errors to be corrected is determined only by the contents of the T register in the CDU. Of course, the t+1 slaves must be available to correct t errors.

3.15 Circuit Simplifications

A number of simplifications are also possible in specific slaves. The PCU2 parity check units are not necessary in slaves 1, 2, and 3. Slave 1 must have the unity field element stored in the PC2 register, and PC2 in slaves 2 and 3 should contain the zero field element.

Slave 0 needs very little circuitry. No Chien search unit (CSU) is required. Register R2 may be the wired constant for the unity field element. R3 and R4 may be the wired constant zero. No multiplier is necessary for the Berlekamp algorithm, but the adder must add the unity field element to the proper line. The PCU1 and PCU2 parity check calculators must exist, and R0, R1, SR0 and SR1 must exist.

If the decoder is to be operated over only one field, a number of simplifications can be made in the design. The multipliers in the

parity check units (PCU1 and PCU2) can be replaced by much simpler networks or shift registers (1), as can the multiplier in the Chien search unit. The decoder should be able to operate much faster since the generalized multiplier array can be replaced with the fast multiplier circuit. The word length is fixed by the size of the field; however, the number of errors to be corrected is still variable.

3.16 Conclusion

A design has been developed for a BCH decoder capable of operating over a number of fields. The decoder can correct any number of errors, up to the maximum number of slave processors which are available. The circuitry required to detect faults affecting a single bit was developed and a method for deleting faulty processors was discussed. The design presented is quite flexible and can be used to construct either a generalized decoder capable of operation over a number of fields, or a much simpler decoder capable of operation over only a single field.

CHAPTER 4

THE SIMULATION PROGRAM

4.1 Introduction

In order to verify the operation of the decoder design presented in Chapter 3, a detailed simulation program was written. The program was written in SYMBOL, an assembly language for the XDS SIGMA-7 computer. The program was designed to allow variation in the number of slave processors, the code length, and the number errors to be corrected. The slaves are either active or passive, and the program has the ability to change the state of the slaves during execution. The program will operate either from terminal or cards under the XDS Batch Processing Monitor with Batch Timeshare Monitor. A number of simple interactive features were included for ease of operation from the terminal.

4.2 Program Description

The program consists of two types of subprograms--simulation subprograms and utility subprograms. The former represent a simulation of the actual machine design, while the latter provide the interface to the user.

Figure 4.1 shows the flowchart of the main program. First the constants for the Galois field are computed, and the table of multiplicative inverses is defined. The utility routines accomplish these tasks. Another utility routine reads in the word to be decoded. If none is entered, the program terminates. Now the simulation begins. The parity checks are computed first. Then the Berlekamp algorithm

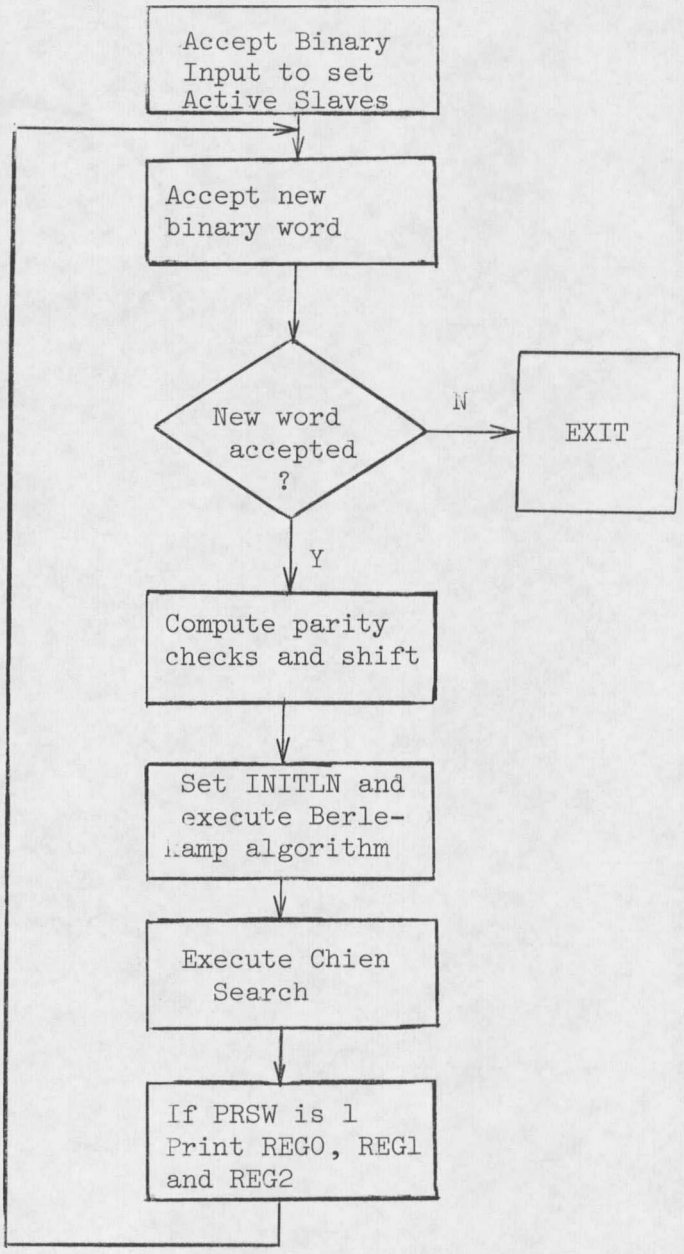


Figure 4.1. The Main Program

is executed, and finally, the error locating polynomial is used to correct the errors which have occurred in the codeword. Then, if desired another utility routine can be used to give a listing of the parity checks and the sigma polynomial in binary notation. The codewords should be entered as a sequence of binary digits, with or without spacing, as:

0101000000100001,

or

0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1.

Only one space between digits is allowed. The length 15 triple error correcting code might be decoded by the following command sequence:

```
RESTART
1111
EOD
T=3
M=4
0010000000100001
EOD
EOD
```

Table 4.1 lists the commands available as input control over the program variables. Table 4.2 lists the subprograms and their type and function.

Table 4.3 shows the variable names which are used in the simulation. There are four types of variables. The first is a vector slave register such as REGO. This register requires NOSLAVE words of storage, where NOSLAVE defines the number of slave processors in the decoder. The second type is the scalar variable.

Table 4.1 Input Command Formats

CODE	FUNCTION
EOD	"End of Data." Control is transferred out of the INOUT routine.
RESTART	The slave activation sequence is executed.
FILE	The echo mode is reversed.
T=5	The number of errors to be corrected is set to the value following the = sign, 5.
M=6	The order of the field is set to the value following the = sign, 6. This also determines the codeword length. The routine GFINIT is also called.

This type of variable requires one word of storage and holds either an integer value right justified, or a field element left justified. The control lines hold either 0 or 1. The third type is the array variable. This type holds two or more words to be used for various purposes. The fourth type consists of variables necessary to the operation of the program--temporary storage of arguments, return addresses for subprograms, I/O control words and buffers, etc. These variables will not be listed explicitly.

Table 4.2 Simulation and Utility Subprograms

NAME	TYPE	FUNCTION
PARITY	Simulation	Computes the parity checks for the codeword in CODEBUF. The results are returned in SYN1 and SYN2.
CHIEN	Simulation	Corrects the code digits in CODEBUF according to the sigma polynomial in the SIG accumulator.
RUNENT	Simulation	Simulates the Master Processor and portions of the slave processors.
INVERSE	Simulation	Computes the inverse of a Galois field element.
SLAVE	Simulation	Simulates portions of the slave processors and calls other slave simulation routines.
IRN	Simulation	Simulates the operation of the Internal Routing Network.
RSU	Simulation	Simulates the Register Selection Unit.
MULTXY	Simulation	Simulates the Galois field multiplier.
ISU	Simulation	Simulates the Input Selection Unit.
GFINIT	Utility	Initializes the constants used by the decoder for the Galois field operations.
PRINT	Utility	Prints a heading specified by the calling program.
INOUT	Utility	Accepts the control codes and the codewords from the input device, and lists the input on the output device, if desired.
OUTCW	Utility	Prints the corrected codeword on the output device.
PROUT	Utility	Lists registers REG0, REG1, and REG2 on the output device.

Table 4.2 (Continued)

NAME	TYPE	FUNCTION
ACTSL	Utility	Takes a binary vector from the input device to specify the activation lines.
PSHIFT	Utility	Provides a shift to align the parity checks for processing by the Berlekamp algorithm.
BUFDMP	Utility	Provides a snapshot dump of the contents of the registers selected by the calling sequence.

Table 4.3 Control Lines and Registers.

NAME	TYPE	FUNCTION
NOSLAVE	scalar	Defines the number of slave processors.
M	scalar	The order of the Galois field.
IRPOLY	scalar	The coefficients of the irreducible, left justified.
IRPOLYI	scalar	The coefficients of the irreducible polynomial, right justified.
IRPTAB	array	List of irreducible polynomials for GF(2^2) through GF (2^{10}).
RDMPSW	scalar	If 1, the BUFDMP routine is called.
PRSW	scalar	If 2, the PROUT routine is called.
CODEBUF	array	Holds the codeword.
INVM	array	The table of Inverse elements.
MA	scalar	The Master Accumulator.
DSREG	scalar	The Degree Sigma Register.
KREG	scalar	The K (iteration count) register.
TREG	scalar	Holds the number of errors, t, to be corrected.
OC1	scalar	Output of KREG:DSREG comparison.
OC2	scalar	Output of KREG:TREG comparison.
WTLINE	scalar	The Wait Bit--1 after termination of Berlekamp algorithm.
INITLN	scalar	Initialization Bit--1 during initialization--should be set to 1 before calling RUNENT. RUNENT resets it to 0 after initialization.
DLINE	scalar	Zero if MA=0, otherwise 1.

Table 4.3 (Continued)

NAME	TYPE	FUNCTION
I0, I1, I2	scalar	Instruction lines to the slave processors.
RSLINE	scalar	Controls source of the new auxiliary polynomial.
IR0, IR1, IR2	scalar	Supply inputs to the I0-I2 lines. These form the instruction register.
IMA	scalar	The input to MA before clocking.
IDSR	scalar	Input to the DSREG before clocking.
IKR	scalar	Input to the KREG before clocking.
ICON	vector	Supplies initialization constants for the slave processors.
ADDOUT	scalar	The adder output of the arithmetic unit.
MLTOUT	scalar	The multiplier output of the arithmetic unit.
MLTAC	scalar	The multiplier output for input to the accumulator adders.
REG0	vector	Corresponds to R0 in the logic design.
REG1	vector	Corresponds to R1.
REG2	vector	Corresponds to R2.
REG3	vector	Corresponds to R3.
REG	vector	Corresponds to R4.
SREG0	vector	Corresponds to SR0.
SREG1	vector	Corresponds to SR1.
SYN1	vector	Corresponds to PC1.
SYN2	vector	Corresponds to PC2.

Table 4.3 (Continued)

NAME	TYPE	FUNCTION
SIG	vector	Corresponds to CSR.
MALINE	scalar	Lines which transmit the contents of the MA to the slaves.
ACIN	vector	The accumulator inputs from the slaves.
TLINE	vector	For computing the degree of the sigma polynomial.
SALN	vector	Gives the configuration of active and passive slaves.
SCLN	vector	These lines are 1 for the lowest two slaves and 0 for all others.
IREGO	scalar	The input to REGO before clocking.
IREG1	scalar	Inputs to REG1.
IREG2	scalar	Inputs to REG2.
IREG3	scalar	Inputs to REG3.
IREG4	sclara	Inputs to REG4.
ISREG0	vector	Inputs to SREG0.
ISREG1	vector	Inputs to SREG1.
IRNT	array	Intermediate storage inside the Internal Routing Network.
SCL	array	Stores the clock pulse configuation for the slave registers.
MEXP	scalar	Contains the value 2^m-1.
A0	scalar	Contains the unity field element.

4.3 The Simulation Subprograms

Figure 4.2 shows the flowchart for the routine PARITY. The constants used to compute each parity check were calculated rather than stored, since very little computation time is consumed. The routine computes 1, $C(\alpha)$, $C(\alpha^2), \ldots, C(\alpha^{2t-1})$, and these are stored in reverse order in the two vectors SYN1 and SYN2. Each position of the array is checked against the SALN vector. The parity checks are inserted only in the elements corresponding to active slaves. If the number of active slaves is not large enough, a message is transmitted to the user and the program is restarted. The PSHIFT routine is called to reorder the parity checks for use by the Berlekamp algorithm.

Figure 4.3 shows the flow chart for the master processor simulation program RUNENT. The first action is to pass the instruction register outputs through AND gates to form the I0-I2 lines. The other AND gate input is the INITLN. Table 2.5 shows the settings of the control lines for the various machine cycles.

Table 4.4 Instruction Line Settings.

Lines I0	I1	I2	Machine Action
0	0	0	Initialize.
1	0	0	Execute Cycle 1.
0	1	0	Execute Cycle 2.
0	0	1	Execute Cycle 3.
1	1	1	Wait.

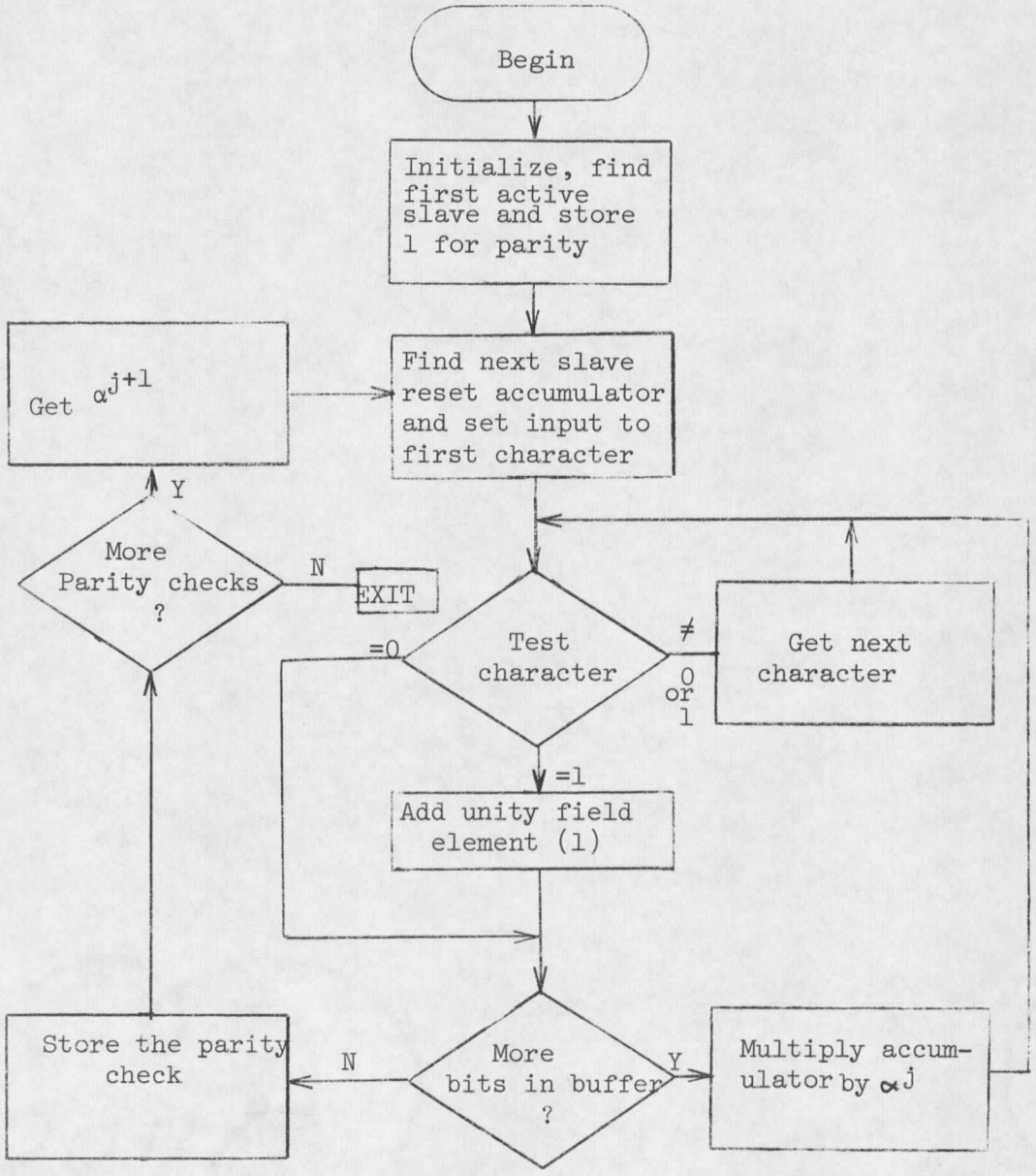


Figure 4.2. Parity Check Routine

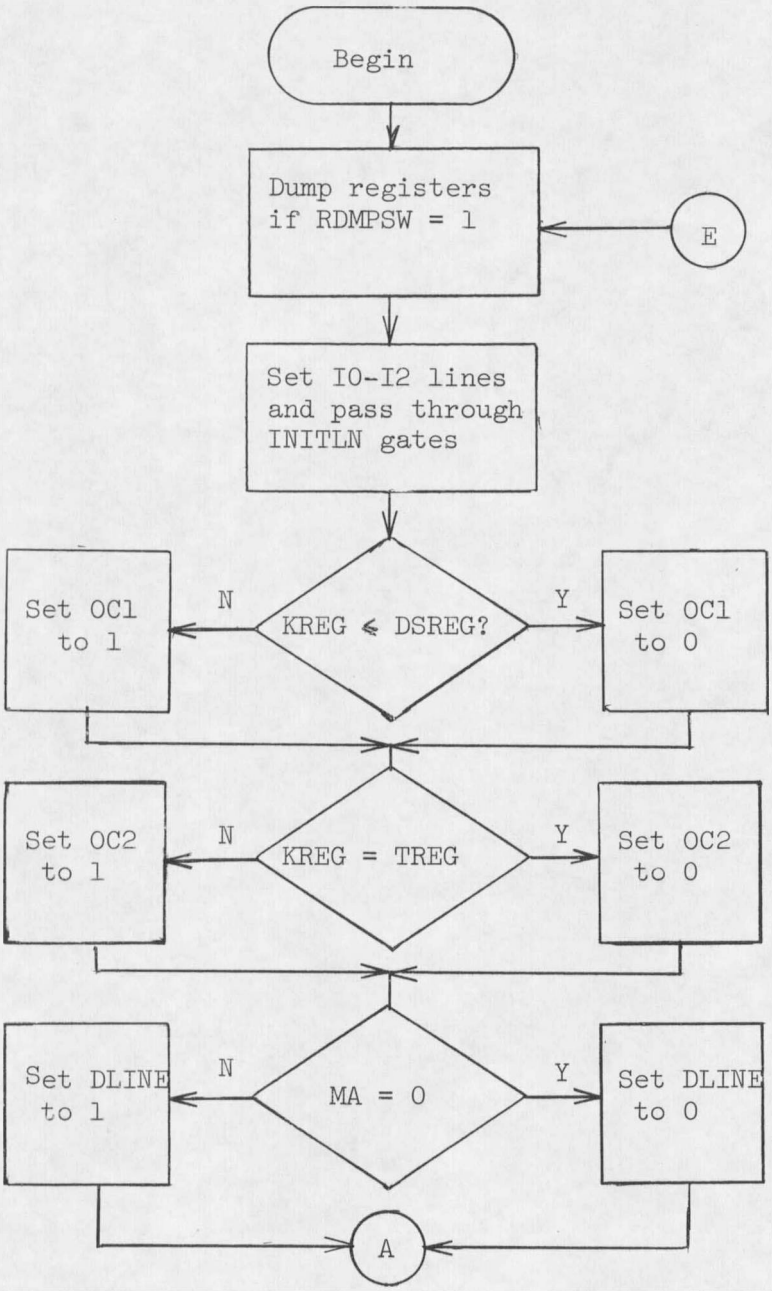


Figure 4.3. Master Processor Simulation

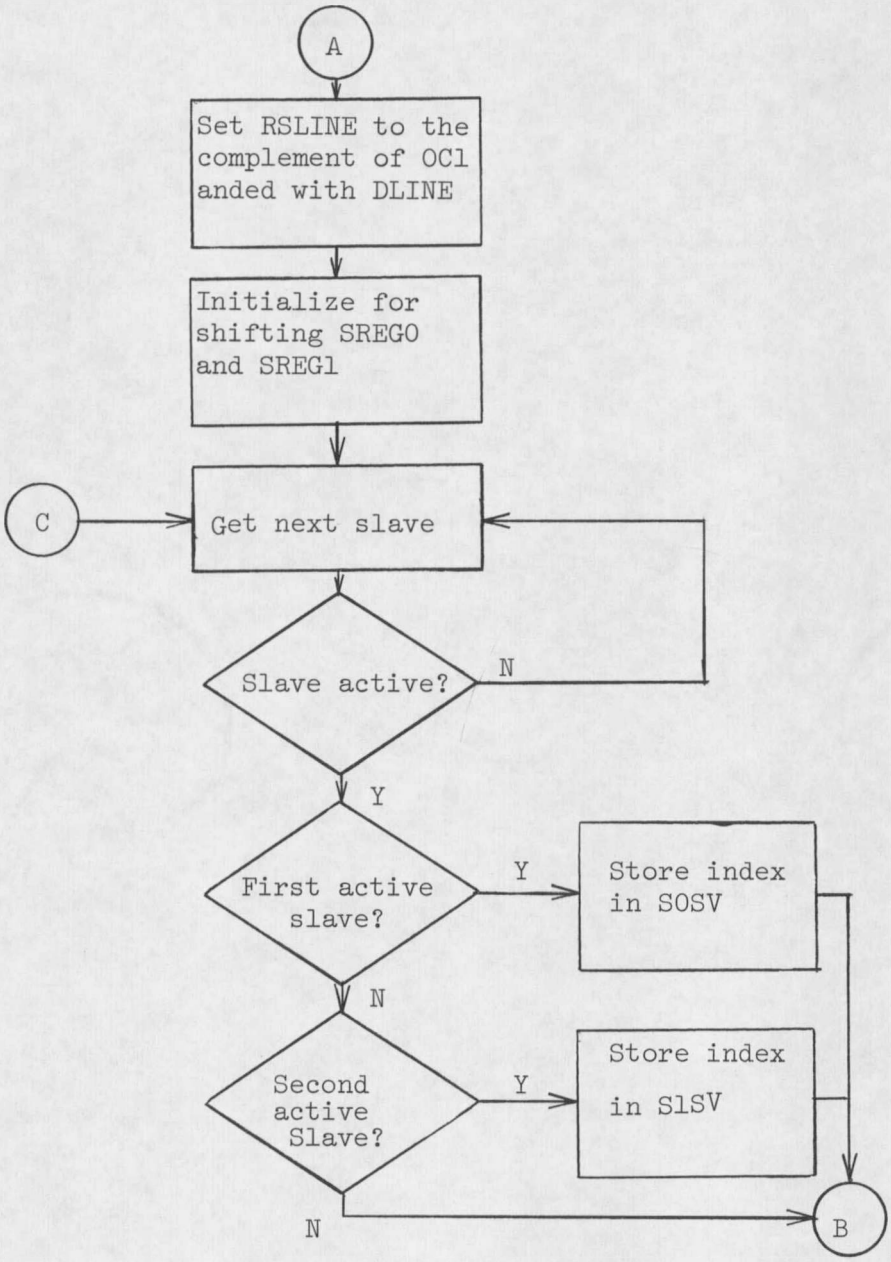


Figure 4.3. (continued)

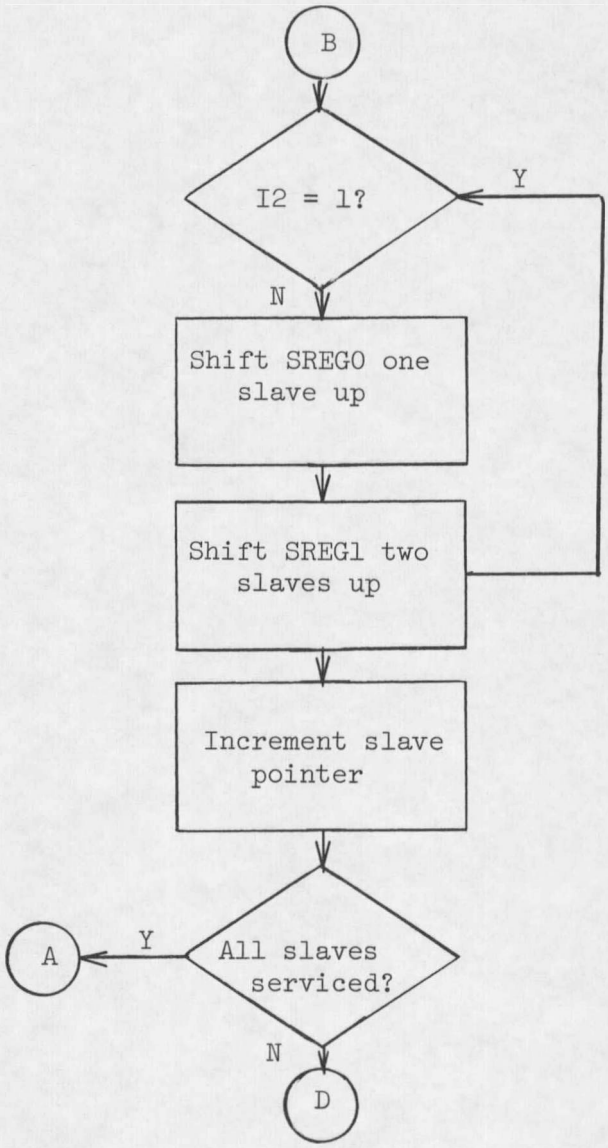


Figure 4.3. (continued)

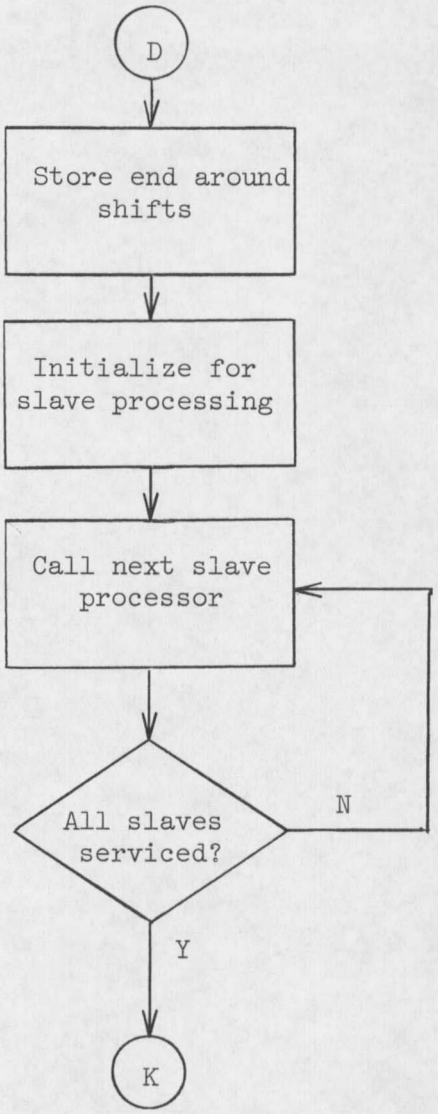


Figure 4.3. (continued)

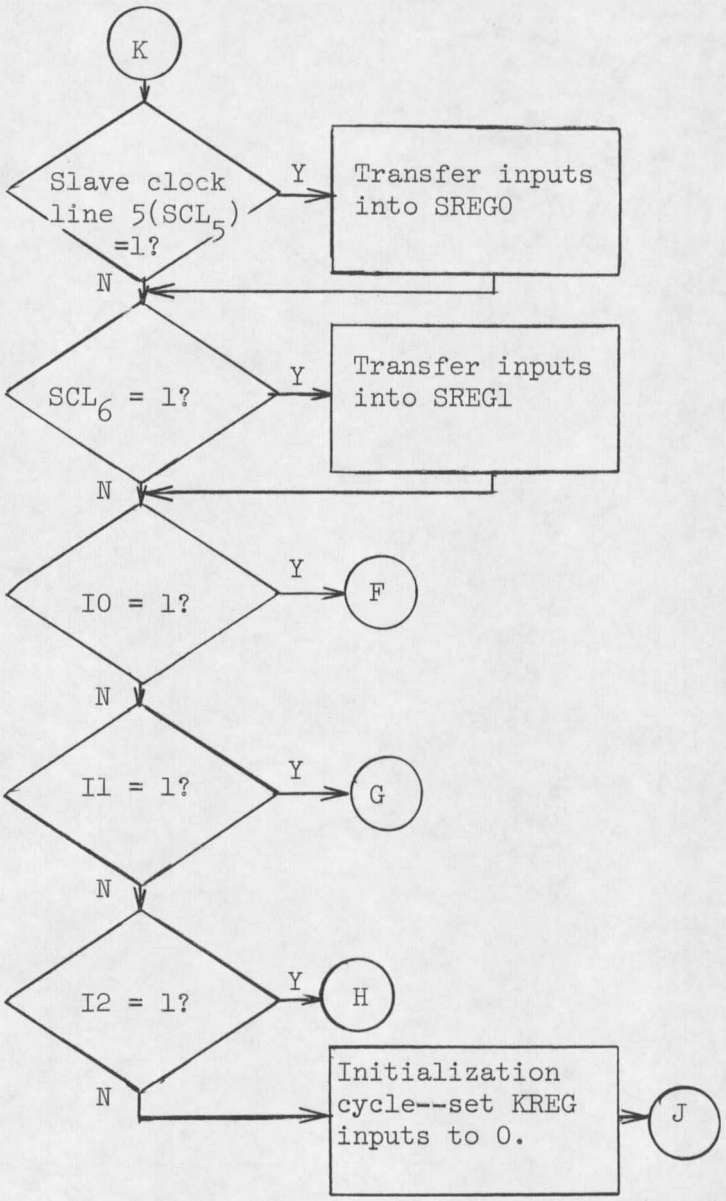


Figure 4.3. (continued)

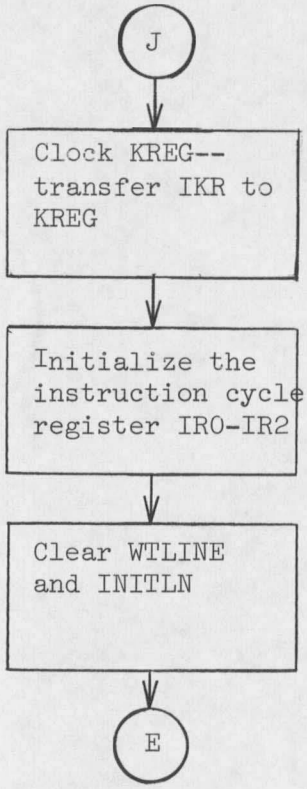


Figure 4.3. (continued)

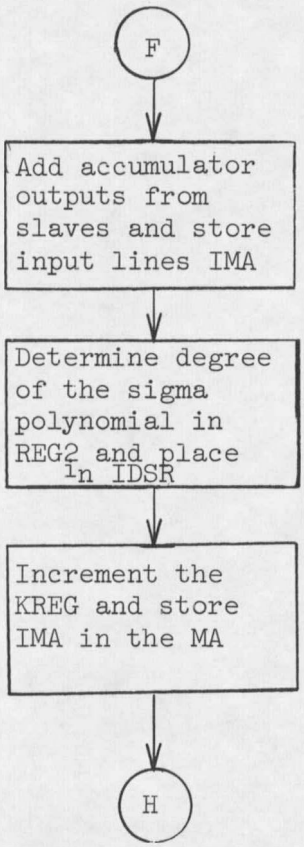


Figure 4.3. (continued)

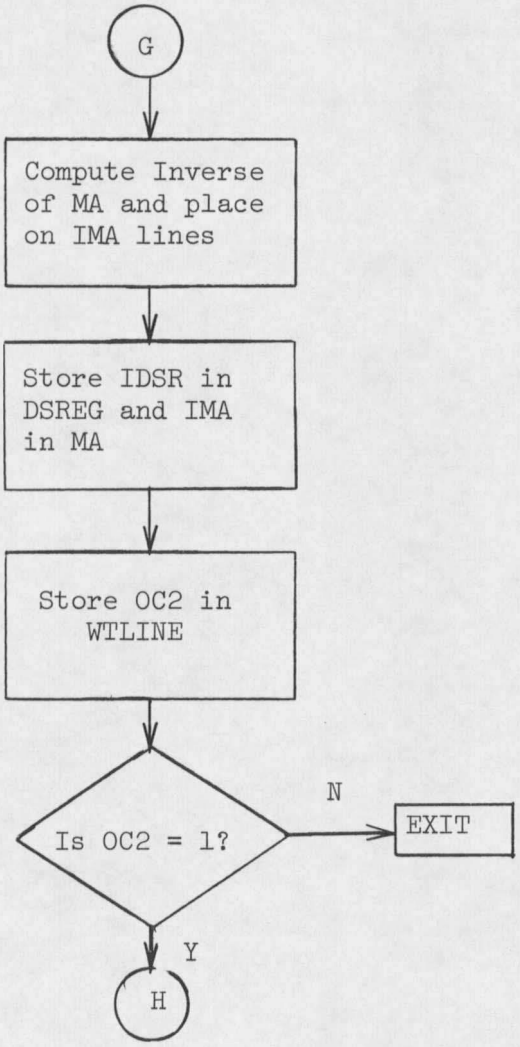


Figure 4.3. (continued)

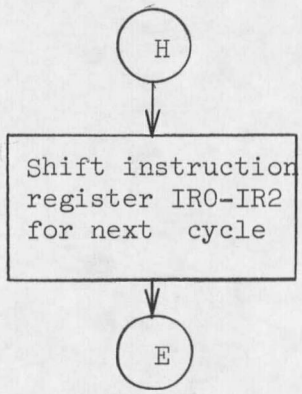


Figure 4.3. (continued)

Next, the comparisons are made to set the OC1 and OC2 lines. The DLINE is set to 0 if the master accumulator is 0, and the RSLINE is set to the complement of OC1 ANDed with the DLINE.

The SREG0 and SREG1 vectors are now shifted to simulate the sequential shift between slaves. Passive slaves are bypassed and SREG0 is not shifted if I2 = 1, since the register is a multiplier input during cycle 3.

The slave processor routine is called for each slave, both active and passive. The SCL_5 and SCL_6 lines are used to determine whether or not the inputs to the SREG0 and SREG1 vectors should be transferred.

During initialization, the inputs to the KREG are zeroed. The clock pulse is applied to the KREG and the WTLINE. The INITLN is cleared to 0 to simulate the action of the counter.

During Cycle 1, the ACIN accumulator lines are all added (modulo 2) and used to specify the input lines to the master accumulator, IMA. The degree of the sigma polynomial is determined using the TLINE vector; the result is placed in IDSR, the inputs to the DSREG. The clock pulse is applied to increment KREG and IMA is stored in MA.

During Cycle 2, the INVERSE routine is called to return the inverse of the element in MA and the result is stored in IMA. The clock pulse is applied to store IDSR in DSREG, IMA in MA, and OC1 in WTLINE. If OC1 is 1, the routine RUNENT terminates. During each

clock cycle, except initialization, the instruction register IR0-IR2 is shifted right one bit to prepare for the next cycle. The program execution returns to location RUNSEQ to execute the next cycle.

The slave processor simulator, SLAVE, flowchart is shown in Figure 4.4. The first action is to set the TLINE for the slave. If the slave is passive, the ACIN line and TLINE are cleared to 0 and the SLAVE routine exits. If the slave is active, the I0 and I1 lines are used to set the inputs to the multiplier in the arithmetic unit. The ISU routine is used to select the inputs. Then the multiply routine is called and the two outputs MLTOUT and MLTAC are specified with the product. The value of REG.2 in the slave is added (Mod 2) to the product and this result is stored in ADDOUT.

Now, the routines for the Internal Routing Network and the Register Selection Unit are executed. The IRN routine sets the slave register inputs, and the RSU routine selects the registers to be clocked. All that remains is the actual clocking of each slave register according to the values on the SCL lines.

Figure 4.5 shows the flowchart for the Internal Routing Network. The register inputs are selected according to the state of the I0, I1, I2, SCLN and RSLINE variables. A series of OR instructions simulates the collection of terms for the outputs.

The multiplier simulation routine MULTXY is shown in Figure 4.6. First, the iteration count is initialized and argument #2 is saved.

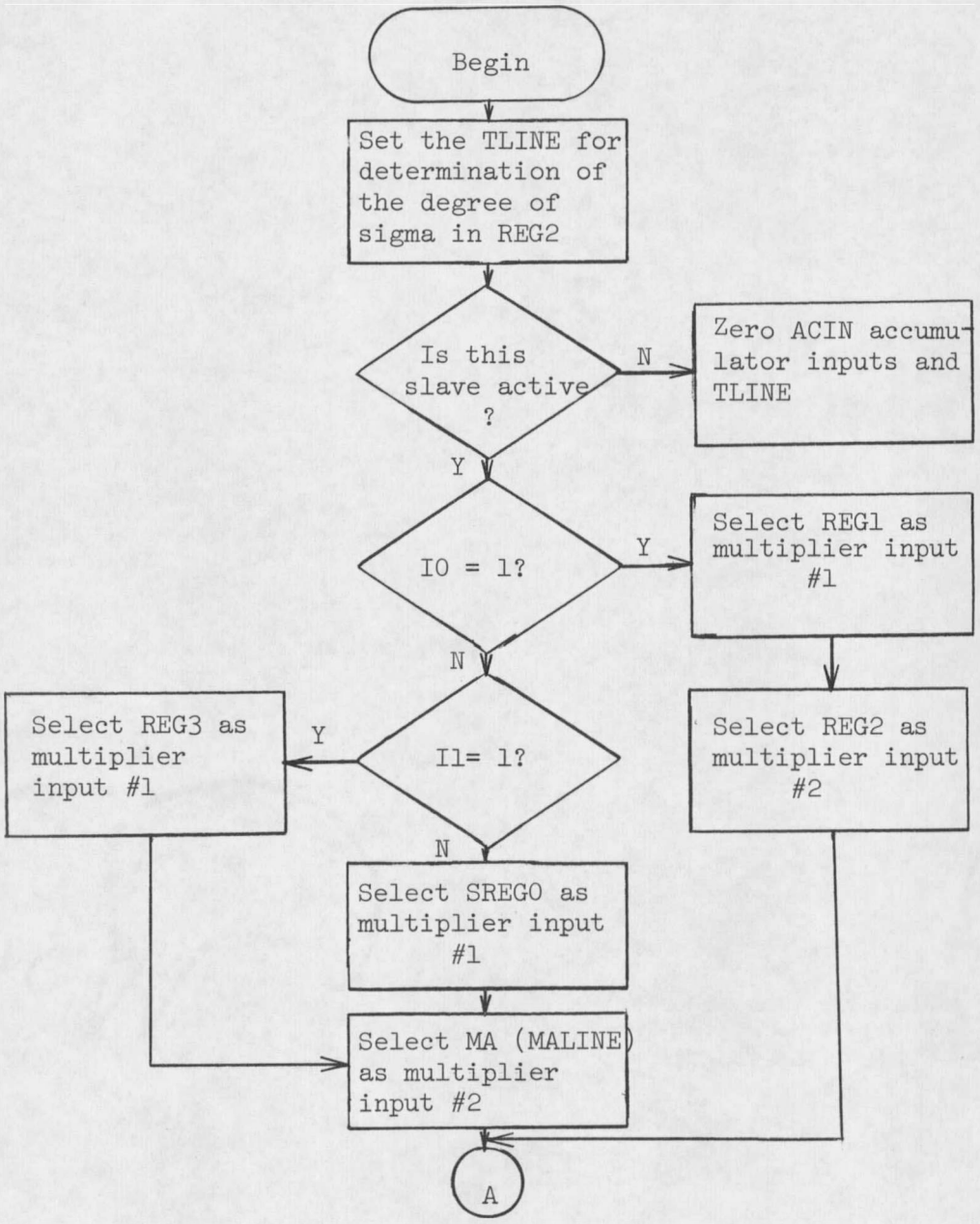


Figure 4.4. Slave Processor Simulation

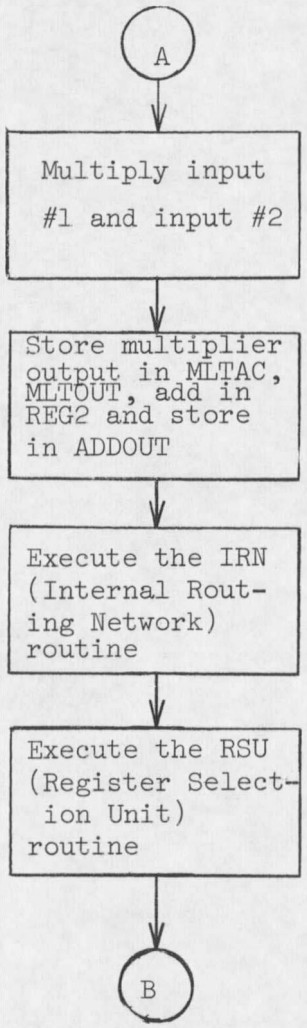


Figure 4.4. (continued)

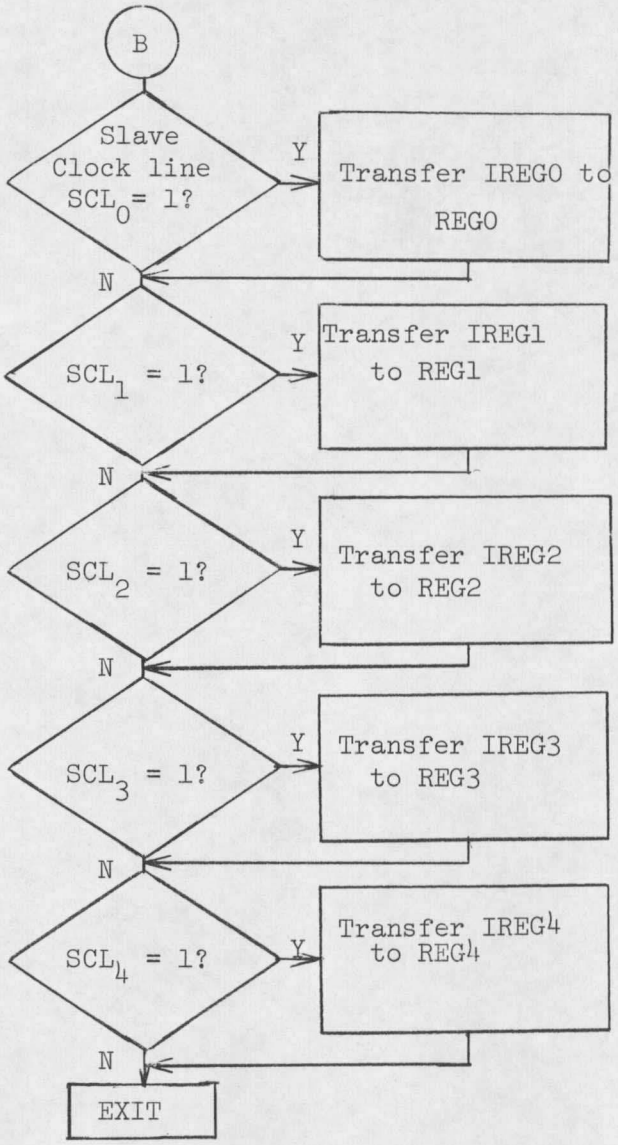


Figure 4.4. (continued)

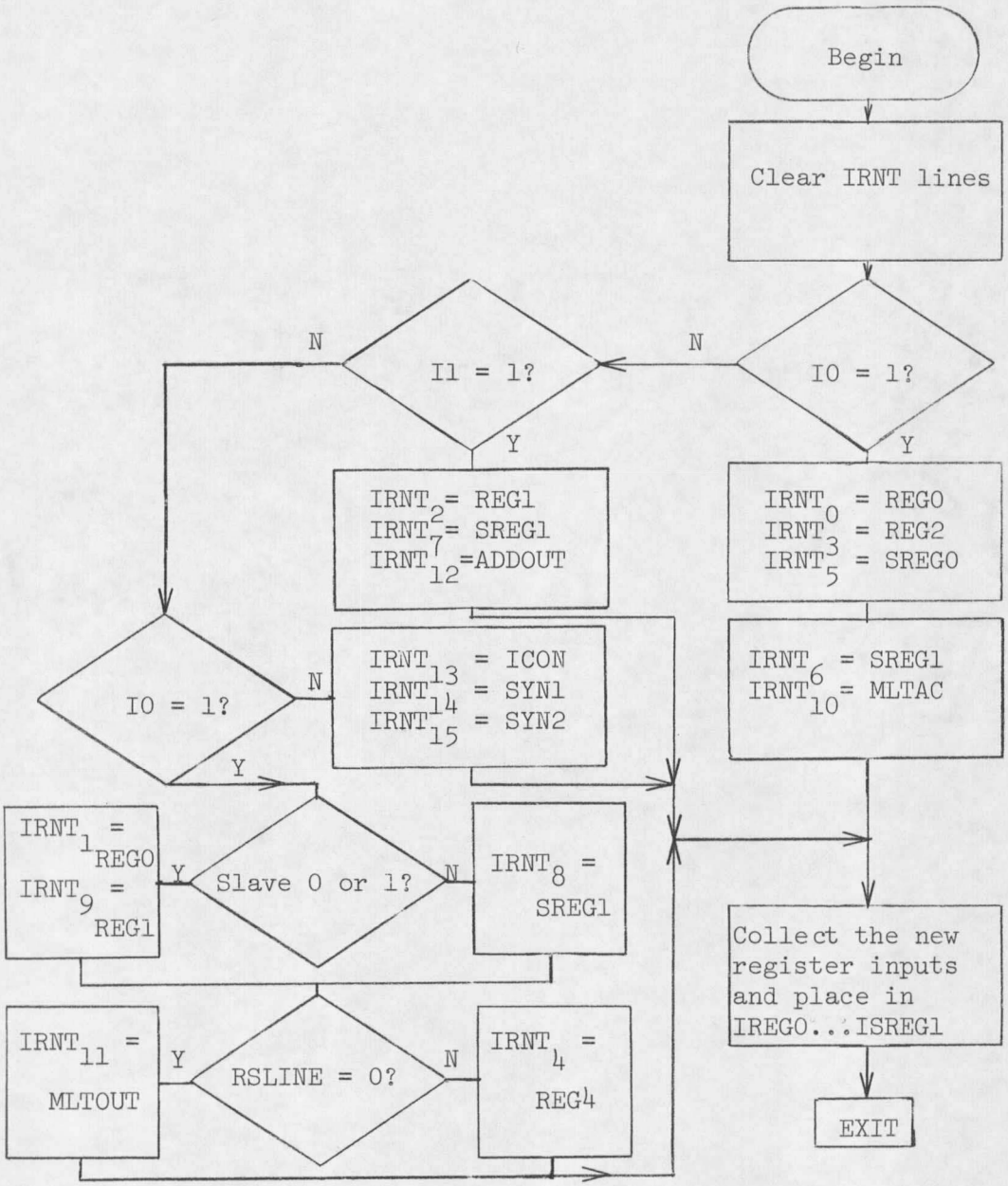


Figure 4.5. Internal Routing Network

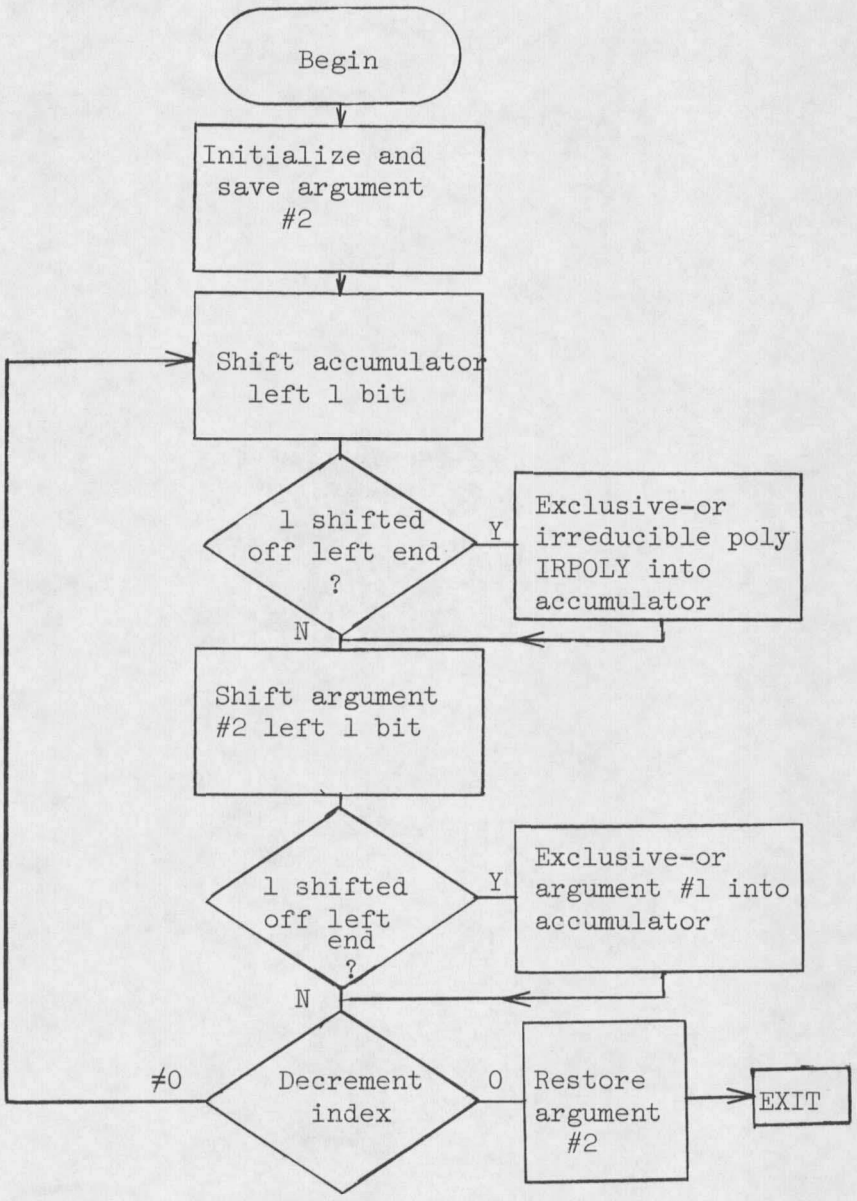


Figure 4.6. Multiplier Routine

The accumulator is shifted left, and if an overflow occurs, the irreducible polynomial is added modulo two to the accumulator. Argument #2 is shifted left; if overflow occurs, argument #1 is added modulo 2 into the accumulator. The cycle is repeated m times, and after argument #2 is restored, the routine exits.

The only simulation routines not shown in the flowcharts are the RSU and INVERSE routines. The RSU routine is simply a set of logical OR instructions which correspond to the circuit diagram for the Register Selection Unit. The INVERSE routine is a table look up routine. The table is constructed by GFINIT.

The final simulation program is the Chien search simulator, CHIEN. This routine is similar to PARITY in that the constants for multiplication of the SIG (CSR) vector are calculated rather than being stored in a constant vector. Figure 4.7 shows the flowchart for CHIEN. The SIG vector is first given the values in REG2. Then the product with the constant vector is formed and SIG is respecified. The sum of all elements of SIG is next computed, and if 0, the corresponding digit is corrected. The operation is repeated for each digit.

4.4 The Utility Subprograms

These programs provide the interface to the user through the input and output devices and execute certain initialization tasks. The programs are quite simple, and no flowcharts will be given. The

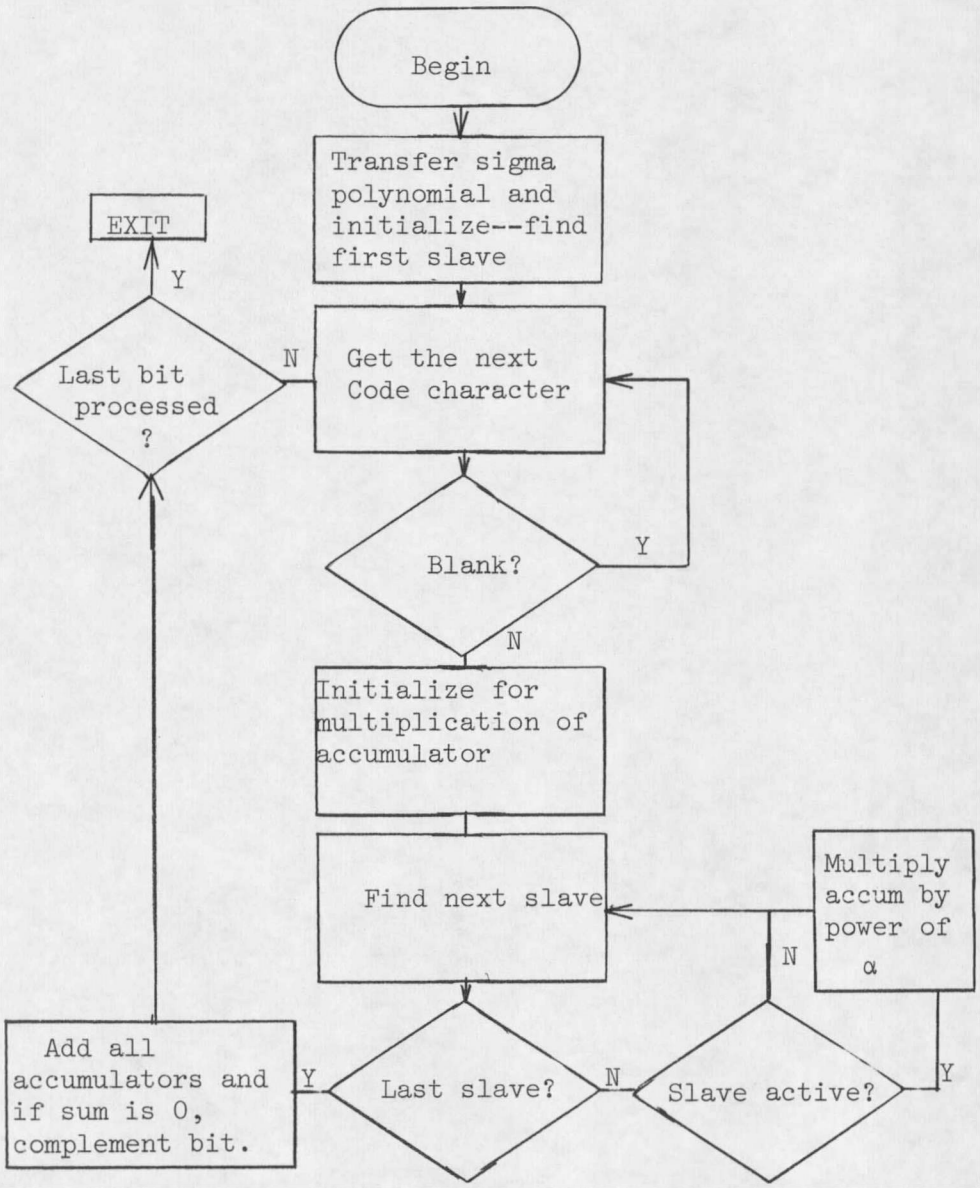


Figure 4.7. Chien Search Routine

listings are included in Appendix B.

4.5 Program Results

The simulation program was run for a large number of words, and decoded successfully whenever the number of errors was not greater than t, the maximum number to be corrected. Codes of varying lengths were run. Appendix B shows the output from a number of runs. The largest word decoded was a length 1023 code with 255 errors corrected. Computation time was about 4.5 minutes. Since the actual code word transmitted is irrelevant to the decoding algorithm once the parity checks have been computed, the all zero code word was used for most of the tests. For the shorter code words, the REG0, REG1 and REG2 vectors were printed. The listing shows both the active and passive slaves.

4.6 Conclusion

The simulation program verifies the logic design presented in Chapter 3. The program is reasonably short, and yet simulates in detail all important functions performed by the elements of the decoder. The simulator can operate for any reasonable codeword length and correct any reasonable number of errors, with reasonable time and memory requirements. The utility routines provide an important I/O interface with the user, and allow interactive operation with the user

via the teletype terminal. This interactive mode was found to be extremely valuable in implementing design changes and in locating and correcting "bugs." The program suggested a number of changes in the design as the work progressed.

CHAPTER 5

CONCLUSION

5.1 Summary

The problem considered in this thesis is the design of a special purpose computer to decode the BCH codes. System requirements for both fixed and variable-word-length decoders are discussed, but the significant problems of code word synchronization and interleaving are not considered. Fault detection circuitry is developed for the processors and fault correction is discussed.

The decoding algorithm can be performed by a single processor, but the time required to decode a word can be reduced by using multiple processors--we can trade equipment for time. Developments in LSI technology suggest that such a tradeoff might become quite attractive in the near future. While the duplication of LSI processors is less costly, the processor design constraints are somewhat different. LSI design places a permium on regularity and on reducing the number of external connections, but the total number of gates is less important than in conventional systems. Also, different types of testing procedures must be developed.

The techniques used in this design should be useful in implementing other algorithms with parallel processors. For example, the block diagram of the slave processor could be used as a starting point in designing systems to implement other algorithms. Several units in the slaves, such as the slave register set and the internal routing network, could very likely be used directly for other algorithms. Many

of the techniques used here might be implemented using a general computer design program coupled with a graphics interface to the designer.

During the research, the many simulation programs written proved to be invaluable in debugging and improving both processor and algorithm designs. Simulation of the processor design is a necessary step in any future work which is done. The simulation proved to be most useful when done on two levels; the first, a general simulation of the processor data flow, and the second a detailed simulation of the logic design.

This research, in combination with other research on parallel processor algorithms, should be helpful in developing the philosophy and organization of general purpose parallel processors.

5.2 Suggestions for Future Research

There are undoubtedly many other algorithms which can be performed by parallel processors. In the field of coding theory, research should determine whether convolutional codes can be decoded by a parallel processor.

As more complicated algorithms are examined, some method should be developed to determine which parts of an algorithm can be executed simultaneously. The method should determine whether the processors execute the same or different instruction sequences. Certain classes of algorithms may exist which can be performed

efficiently by parallel processors with a specific organization. For instance, certain matrix algorithms such as the Kalman Filter algorithm are performed efficiently by Cannon's machine (3). A study should be made first to define a set of machine organization equivalence classes for algorithms and second to classify algorithms in these equivalence classes. These results would be especially useful in defining a general purpose parallel processor.

The problem of fault detection was considered in Chapter 3. Fault detection for the cellular array multiplier proved to be particularly simple, because of the large number of linear operations performed by the array. Further research should see if the same techniques can be used to detect faults in other linear circuits.

Research should be performed to determine which of the units in this system are useful in designing other processors. Other units for processor design should also be investigated. The results of this research could be used to implement the general computer design program previously mentioned.

APPENDIX A

APL SIMULATION RESULTS

```
     ∇  BCHB
[   1]  M←⍴ IRPOLY
[   2]  N←¯1+ 2* M
[   3]  A0←M⍵1
[   4]  A1← 1↑ A0
[   5]  'ENTER RECEIVED WORD'
[   6]  CW←⍞
[   7]  PARITY
[   8]  →( 1=+/+/[1] S)/ BCHB2
[   9]  BERM2
[  10]  SIG← R2
[  11]  DSIG← ¯1+(⍴ R2)[1]
[  12]  SIG← SIG[ DSIG + 2-⍳ DSIG+ 1;]
[  13]  CHIEN
[  14]  →(IB=0)/BCHB1
[  15]  'CORRECTED WORD IS'
[  16]  BCHB2: CW
[  17]  → 0
[  18]  BCHB1:'UNCORRECTABLE ERROR'
     ∇
```

```
      ⍝THIS IS A TEST OF A LENGTH 7 DOUBLE ERROR
      ⍝CORRECTING BCH CODE....
      ⍝THE IRREDUCIBLE POLYNOMIAL IS X*3 + X + 1

      IRPOLY←0 1 1
      T←2

      BCHB
ENTER RECEIVED WORD
□:
      0 0 1 0 1 0 0
CORRECTED WORD IS
 0 0 0 0 0 0 0

      BCHB
ENTER RECEIVED WORD
□:
      0 1 0 0 0 0 1
CORRECTED WORD IS
 0 0 0 0 0 0 0

      BCHB
ENTER RECEIVED WORD
□:
      0 0 0 1 0 1 0
CORRECTED WORD IS
 0 0 0 0 0 0 0

      BCHB
ENTER RECEIVED WORD
□:
      1 0 1 1 1 1 1
CORRECTED WORD IS
 1 1 1 1 1 1 1

      BCHB
ENTER RECEIVED WORD
□:
      1 1 1 0 0 0 0
UNCORRECTABLE ERROR
```

```
      ○THIS IS A TEST OF A LENGTH 15 TRIPLE ERROR
      ○CORRECTING BCH CODE...
      ○THE IRREDUCIBLE POLYNOMIAL IS X*4 + X + 1

      IRPOLY←0 0 1 1
      T←3


      BCHB
ENTER RECEIVED WORD
□:
      0 0 0 1 0 0 1 0 0 0 0 1 0 0 0
CORRECTED WORD IS
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0


      BCHB
ENTER RECEIVED WORD
□:
      0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
CORRECTED WORD IS
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0


      BCHB
ENTER RECEIVED WORD
□:
      1 1 1 1 0 1 1 1 1 1 1 1 1 1 1
CORRECTED WORD IS
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1


      BCHB
ENTER RECEIVED WORD
□:
      1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
UNCORRECTABLE ERROR
```

```
      ○THIS IS A TEST OF A LENGTH 15 TRIPLE ERROR
      ○CORRECTING BCH CODE--WITH A LISTING OF THE
      ○SYNDROME AND ERROR LOCATING POLYNOMIALS...

      IRPOLY←0 0 1 1
      T←3


      BCHB
ENTER RECEIVED WORD
□:
      1 0 0 0 0 0 1 0 0 0 1 0 0 0 0
CORRECTED WORD IS
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0


      ○SYNDROME DIGITS...
      S
 0 0 0 1
 1 1 1 1
 1 0 1 0
 1 0 1 0
 1 0 0 0
 0 1 1 0
 1 0 0 0


      ○ERROR LOCATING POLYNOMIAL...
      SIG

 1 1 1 0
 1 1 0 0
 1 1 1 1
 0 0 0 1
```

```
oTHIS IS A TEST OF A LENGTH 31 FIVE ERROR CORRECTING
oBCH CODE...THE IRREDUCIBLE POLYNOMIAL IS X*5 + X*2 + 1

      IRPOLY←0 0 1 0 1
      T←5


      BCHB
ENTER RECEIVED WORD
□:
      0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0
      0 0 0 1 0 1 0 0 0 0 0 0 0
CORRECTED WORD IS
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0

oSYNDROME....
S
 0 0 0 0 1
 1 1 1 1 1
 1 0 0 1 0
 0 1 1 1 1
 0 1 0 0 1
 0 0 1 1 1
 1 1 1 1 1
 0 0 0 1 1
 0 1 0 1 1
 1 0 1 1 1
 1 0 1 0 1

oSIGMA...
SIG
 1 1 1 0 1
 0 1 1 1 1
 1 0 0 0 0
 1 0 0 1 1
 1 1 1 1 1
 0 0 0 0 1
```

APPENDIX B

ASSEMBLY LANGUAGE

SIMULATION AND RESULTS

```
***                                                                  ***
* SIMULATION--THIS PROGRAM CALLS THE VARIOUS ELEMENTS                 *
* OF THE SIMULATION PROGRAM.                                          *
***                                                                  ***
          DEF       INOUT,PARITY,RUNENT,CHIEN,OUTCW,PROUT,GFINIT
          DEF       IOLAST,ADRS,LCOMP,INITLN,TRFPT
          DEF       PSHIFT,ACTSL
          DEF       START,REDO
          DEF       REG0,REG2,SALN,SCLN,TREG
          DEF       RDMPSW,PRSW,BTMSW
          REF       M:SI,M:LO
START     EQU       $
          CAL1,8    TRFPT
RESTART   EQU       $
          BAL,D2    GFINIT
          BAL,D2    ACTSL
REDO      EQU       $
          BAL,D2    INOUT
          LW,R2     IOLAST
          CW,R2     ADRS
          BE        OUT
          BAL,D2    PARITY
          BAL,D2    PSHIFT
          LW,R2     LCOMP
          STW,R2    INITLN
          BAL,D2    RUNENT
          BAL,D2    CHIEN
          BAL,D2    OUTCW
          MTW,0     PRSW
          BNE       $+2
          BAL,D2    PROUT
          B         REDO
OUT       EQU       $
          CAL1,9    1
```

```
***                                                              ***
* UTILITY--DATA DEFINITION AREA FOR SIMULATION PROGRAM.            *
***                                                              ***
NOSLAVE  EQU      10
NOERR    EQU      1
M        DATA     5
IRPOLY   DATA     X'28000000'
IRPOLYI  DATA     0
BTM      EQU      0
IRPTAB   EQU      $-2
         DATA     X'C0000000'
         DATA     X'60000000'
         DATA     X'30000000'
         DATA     X'28000000'
         DATA     X'0C000000'
         DATA     X'12000000'
         DATA     X'1D000000'
         DATA     X'08800000'
         DATA     X'02400000'
         DO1      BTM
         GOTO     BTM1
CRET     EQU      X'40404040'
BUFSZ    EQU      20
BUFSZB   EQU      80
         GOTO     BTM2
BTM1     BOUND    1
CRET     EQU      X'15000000'
BUFSZ    EQU      15
BUFSZB   EQU      60
BTM2     BOUND    1
R0       EQU      0                    SIGMA 7 REGISTER DESIGNATIONS
R1       EQU      1
R2       EQU      2
R3       EQU      3
R4       EQU      4
R5       EQU      5
```

```
R6          EQU         6
R7          EQU         7
S1          EQU         8
S2          EQU         9
X1          EQU         10
X2          EQU         11
D1          EQU         12
D2          EQU         13
D3          EQU         14
D4          EQU         15
NOBITS      EQU         32                  NUMBER OF BITS IN SIGMA 7
TRFPT       GEN,8,24    X'14',0                TO SET TRAPS
            DATA        X'3C000003'         TO IGNORE INTEGER OVERFLOW
SELSL       DATA,1      1,1,1,1
RDMPSW      DATA        0
BTMSW       DATA        BTM
PRSW        DATA        BTM
LCOMP       DATA        1                   TO COMPLEMENT MACHINE LINES
COMP        DATA        X'FFFFFFFF'         FOR PARALLEL  LINE COMPLEMENTS
XF0         DATA        '0   '              CHARACTER 0
XF1         DATA        '1   '              CHARACTER 1
R1SAVE      DATA        0
ARG1        DATA        0
SCNT        DATA        -1
S0SV        DATA        0
S1SV        DATA        0
```

```
***                                                                  ***
* UTILITY--INPUT/OUTPUT DATA DEFINITION AREA.                         *
***                                                                  ***
          DATA      '    '
CODEBUF   RES,1     2160              HOLDS THE CURRENT CODEWORD
BUFTOP    EQU       $                 UPPER LIMIT OF CODEWORD
READ      GEN,8,24  X'10',M:SI        FOR COEDWORD INPUT
          GEN,4,28  15,16
          DATA      IOHLT
          DATA      IOHLT
RADR      DATA      CODEBUF
          DATA      BUFSZB+1
WRITE     GEN,8,24  X'11',M:LO        FOR CODEWORD ECHO
          GEN,4,28  3,16
WADR      DATA      CODEBUF
          DATA      BUFSZB
ADRS      DATA      CODEBUF           ADDRESS OF CODEBUF
IOLAST    DATA      0                 POINTER TO LAST CARD IMAGE READ
BLNK      DATA      '    '
ECK       DATA      'EOD '
RCK       DATA      'REST'
TCK       DATA      'T   '
MCK       DATA      'M   '
FCK       DATA      'FILE'
CHMSK     DATA      X'F'
PRHD4     EQU       $
          TEXTC     'ACTIVE SLAVE CONFIGURATION:    '
          ORG       $-1
          DATA      CRET
PRHD1     EQU       $
          TEXTC     'RECEIVED  CODEWORD:    '
          ORG       $-1
          DATA      CRET
PRHD2     EQU       $
          TEXTC     'CORRECTED CODEWORD:    '
          ORG       $-1
```

```
          DATA      CRET
PRHD3     EQU       $
          DATA      X'4F606060'
          TEXT      '------------SYNDROME POLYNOMIAL-------------'
          TEXT      ' --SIGMA POLYNOMIAL--       '
          ORG       $-1
          DATA      CRET
ERRM1     TEXTC     'NOT ENOUGH SLAVES AVAILABLE--ENTER NEW '
ERRM2     TEXTC     'NOT ENOUGH SLAVES--MUST REASSEMB_E--'
ERRM3     TEXTC     'NOT ENOUGH CODEBITS ENTERED--'
ERRM4     TEXTC     'BUFFER OVERFLOW'
TPAGE     GEN,8,24  2,M:LO              NEW PAGE ON LINE PRINTER
MAXPAGE   EQU       9                   MAXIMUM FIELD TO PRINT ONE CW
OUTBUF    DATA,1    ' '                 OUTPUT BUFFER
          RES,1     119
OUTFMT    EQU       6
PRBUF     GEN,8,24  X'11',M:LO          PRINT OUTBUF
          GEN,4,28  3,16
          DATA      OUTBUF
          DATA      120
MOVBL     GEN,8,24  119,BA(OUTBUF)+1        CLEAR OUTBUF
MOVCB     GEN,8,24  BUFSZB,BA(CODEBUF)
```

```
***                                                                  ***
* SIMULATION--MASTER PROCESSOR DATA AREA--ALL DATA STORAGE.           *
* TANTS.                                                             *
***                                                                  ***
INVM      RES     1024
MA        RES     1                  MASTER ACCUMULATOR
DSREG     DATA    0                  DEGREE SIGMA REGISTER
KREG      DATA    0                  K REGISTER (ITERATION COUNT)
TREG      DATA    NOERR
OC1       DATA    0                  OUTPUT OF COMPARATOR FOR K AND DSIG
OC2       DATA    0                  OUTPUT OF COMPARATOR FOR K AND T
WTLINE    DATA    0
INITLN    DATA    0                  MASTER PROCESSOR CONTROL LINES
DLINE     DATA    0                  OR REDUCTION OF MA
IO        DATA    0
I1        DATA    0
I2        DATA    0
RSLINE    DATA    0
IRO       DATA    0
IR1       DATA    0
IR2       DATA    0
IMA       DATA    0                  THESE ARE INPUTS TO THE RESPECTIVE
IDSR      DATA    0                  LINES AND REGISTERS BEFORE THE
IKR       DATA    0                  CLOCK PULSE IS APPLIED
```

```
***                                                                  ***
* SIMULATION--SLAVE PROCESSOR DATA AREA.                              *
***                                                                  ***
ICON      RES       NOSLAVE
ADDOUT    RES       1                 ADDER OUTPUT
MLTOUT    RES       1                 MULTIPLIER OUTPUT
MLTAC     RES       1
REG0      RES       NOSLAVE           SLAVE REGISTER 0
REG1      RES       NOSLAVE           SLAVE REGISTER 1
REG2      RES       NOSLAVE           SLAVE REGISTER 2
REG3      RES       NOSLAVE           SLAVE REGISTER 3
REG4      RES       NOSLAVE           SLAVE REGISTER 4
SREG0     RES       NOSLAVE
SREG1     RES       NOSLAVE
SYN2      RES       NOSLAVE           SYNDROME BUFFER
SYN1      RES       NOSLAVE           SYNDROME BUFFER
SYNLOAD   EQU       $-1               LOAD ADDRESS FOR SYNDROME
SIG       RES       NOSLAVE           CHIEN SEARCH BUFFER
MALINE    RES       1
ACIN      RES       NOSLAVE           ACCUMULATOR INPUT
TLINE     RES       NOSLAVE           DSIG UNIT INPUTS
SALN      EQU       $
          RES,1     NOSLAVE
          BOUND     4
SCLN      EQU       $
          RES,1     NOSLAVE
          BOUND     4
IREG0     RES       1                 INPUTS TO THE REGISTERS
IREG1     RES       1
IREG2     RES       1
IREG3     RES       1
IREG4     RES       1
ISREG0    RES       NOSLAVE
ISREG1    RES       NOSLAVE
```

```
***                                                                  ***
* SIMULATION--THIS SUBPROGRAM COMPUTES THE PARITY CHECKS              *
* FOR THE CODEWORD STORED IN CODEBUF.                                 *
***                                                                  ***
CHARIN   DATA    0
RETPAR   DATA    0
PARITY   EQU     $
         STW,D2  RETPAR
         LI,R1   NOSLAVE+NOSLAVE
         LI,R2   0
         STW,R2  SYN2-1,R1
         BDR,R1  $-1
         LW,R6   MEXP
         AI,R6   -1
         LW,X1   A0
         SLS,X1  1
         LI,R1   1
         LW,D3   TREG
         SLS,D3  1
         LI,R7   NOSLAVE
         LI,D4   1
         BAL,D2  SKPAR
         LW,S1   A0
         STW,S1  SYNLOAD,R1
DOPARN   EQU     $
         BAL,D2  SKPAR
         LI,S1   0
         LI,R3   0
         LI,R2   -1
DOPAR    EQU     $
         LI,R4   0
GETCHAR  EQU     $
         AI,R2   1
         LB,R5   CODEBUF,R2
         CB,R5   XF0
         BE      DOMULT
```

```
         CB,R5     XF1
         BE        DOEOR
         AI,R4     1
         CI,R4     1
         BLE       GETCHAR
         BAL,D2    PRINT
         DATA      ERRM3
         B         REDO
DOEOR    EQU       $
         EOR,S1    AO
DOMULT   EQU       $
         AI,R3     1
         CW,R3     R6
         BE        STOPAR
         LW,X2     S1
         BAL,D2    MULTXY
         B         DOPAR
STOPAR   EQU       $
         STW,S1    SYNLOAD,R1
         AI,D4     1
         CW,D4     D3
         BGE       SCHAR
         SLS,X1    1
         BEV       $+2
         EOR,X1    IRPOLY
         B         DOPARN
SKPAR    EQU       $
         AI,R1     -1
         AI,R7     -1
         BGE       RSK
         CI,R1     -NOSLAVE-NOSLAVE+3
         BL        POUT
         AI,R7     NOSLAVE
RSK      EQU       $
         MTB,0     SALN,R7
         BE        SKPAR
```

```
          B         *D2
SCHAR     EQU       $
          STW,R2    CHARIN
          B         *RETPAR
POUT      EQU       $
          BAL,D2    PRINT
          DATA      ERRM1
          B         RESTART
***                                                              ***
* SIMULATION--THIS SUBPROGRAM EXECUTES THE CHIEN SEARCH ROUTINE.*
***                                                              ***
CHOUT     DATA      0
CHIEN     EQU       $
          STW,D2    CHOUT
          LI,R1     NOSLAVE
          LW,R3     REG2-1,R1
          STW,R3    SIG-1,R1
          BDR,R1    $-2
          LI,R2     -1
          LI,R5     0
NEXTDIG   EQU       $
          LW,X1     A0
SKBL      EQU       $
          AI,R2     1
          LB,R4     CODEBUF,R2
          CI,R4     X'40'
          BE        SKBL
          LI,R1     -1
NEXTCH    EQU       $
CHPAS     EQU       $
          AI,R1     1
          CI,R1     NOSLAVE
          BE        CHADD
          MTB,0     SALN,R1
          BNE       CHACT
          STW,R5    SIG,R1
```

```
         B         CHPAS
CHACT    EQU       $
         LW,X2     SIG,R1
         BAL,D2    MULTXY
         STW,S1    SIG,R1
         LW,X2     A0
         SLS,X2    1
         BAL,D2    MULTXY
         STW,S1    X1
         B         NEXTCH
CHADD    EQU       $
         LI,R3     0
         LI,R1     NOSLAVE
         EOR,R3    SIG-1,R1
         BDR,R1    $-1
         CI,R3     0
         BNE       NOCH
         EOR,R4    LCOMP
         STB,R4    CODEBUF,R2
NOCH     EQU       $
         CW,R2     CHARIN
         BGE       *CHOUT
         B         NEXTDIG
```

```
***                                                              ***
* SIMULATION--THIS ROUTINE SIMULATES THE MASTER PROCESSOR.        *
***                                                              ***
RUNRET   DATA     0
RUNENT   EQU      $
         STW,D2   RUNRET
RUNSEQ   EQU      $
         DO1      BTM
         GOTO     BTM1
         MTW,0    RDMPSW
         BE       $+4
         BAL,D2   BUFDMP
         DATA     REG0
         DATA     MALINE
BTM1     BOUND    1
         LI,R1    3
         LW,R3    INITLN
         EOR,R3   LCOMP
         LW,R2    IR0-1,R1
         AND,R2   R3
         STW,R2   I0-1,R1
         BDR,R1   $-3
*DO COMPARISONS
         LW,R4    LCOMP
         LI,R3    0
         STW,R3   OC1
         LW,R2    KREG
         CW,R2    DSREG
         BLE      $+2
         STW,R4   OC1
         STW,R4   OC2
         CW,R2    TREG
         BNE      $+2
         STW,R3   OC2
         STW,R4   DLINE
         LW,R2    MA
```

```
          BNE       $+2
          STW,R3    DLINE
          LW,R2     OC1
          AND,R2    DLINE
          EOR,R2    LCOMP
          STW,R2    RSLINE
* SET UP  SHIFT REGISTERS
          LI,R1     0
          LI,R5     0
SLSH      EQU       $
          MTB,0     SALN,R1
          BE        SHBYP
          CI,R5     0
          BG        SET1
          STW,R1    SOSV
          AI,R5     1
          B         SET0
SET1      EQU       $
          CI,R5     1
          BG        SET0
          AI,R5     1
          STW,R1    S1SV
SET0      EQU       $
          MTW,0     I2
          BNE       SHSKP
          XW,R2     SREG0,R1
SHSKP     EQU       $
          XW,R3     R4
          XW,R3     SREG1,R1
SHBYP     EQU       $
          AI,R1     1
          CI,R1     NOSLAVE-1
          BLE       SLSH
          LW,R1     SOSV
          MTW,0     I2
          BNE       $+2
```

```
          STW,R2    SREG0,R1
          STW,R4    SREG1,R1
          LW,R1     S1SV
          STW,R3    SREG1,R1
          LI,R1     0
          LI,R2     -1
          STW,R2    SCNT
NEXTS     EQU       $
          BAL,D2    SLAVE
          AI,R1     1
          CI,R1      NOSLAVE-1
          BLE       NEXTS
          LI,R1     NOSLAVE
CL6       EQU       $
          MTW,0     SCL+5
          BE        CL7
          LW,R2     ISREG0-1,R1
          STW,R2    SREG0-1,R1
          BDR,R1    $-2
CL7       EQU       $
          LI,R1     NOSLAVE
          MTW,0     SCL+6
          BE        CL8
          LW,R2     ISREG1-1,R1
          STW,R2    SREG1-1,R1
          BDR,R1    $-2
CL8       EQU       $
          MTW,0     I0
          BNE       CYC1
          MTW,0     I1
          BNE       CYC2
          MTW,0     I2
          BNE       CYC3
          LI,R2     0
          STW,R2    IKR
          B         INITCLK
```

```
CYC1      EQU       $
          LI,R2     0
          LI,R3     0
          LI,R4     0
          LI,R1     0
DOACCUM   EQU       $
          MTB,0     SALN,R1
          BE        SKAC
          EOR,R2    ACIN,R1
          MTW,0     TLINE,R1
          BE        $+2
          STW,R4    IDSR
          AI,R4     1
SKAC      EQU       $
          AI,R1     1
          CI,R1     NOSLAVE
          BL        DOACCUM
          STW,R2    IMA
          B         CYC1CLK
CYC2      EQU       $
          LW,X1     MA
          BAL,D2    INVERSE
          STW,X2    IMA
          B         CYC2CLK
CYC3      EQU       $
          B         CYC3CLK
INITCLK   EQU       $
          LW,R2     IKR
          STW,R2    KREG
          LW,R2     LCOMP
          STW,R2    IRO
          LI,R2     0
          STW,R2    IR1
          STW,R2    IR2
          STW,R2    INITLN
          STW,R2    WTLINE
```

```
          B         RUNSEQ
CYC1CLK   EQU       $
          MTW,1     KREG
          LW,R2     IMA
          STW,R2    MA
          STW,R2    MALINE
          B         MCLOCK
CYC2CLK   EQU       $
          LW,R2     IMA
          STW,R2    MA
          STW,R2    MALINE
          LW,R2     IDSR
          STW,R2    DSREG
          LW,R2     OC2
          STW,R2    WTLINE
          BE        *RUNRET
CYC3CLK   EQU       $
MCLOCK    EQU       $
          LW,R2     IR0
          XW,R2     IR1
          XW,R2     IR2
          STW,R2    IR0
          B         RUNSEQ
***                                                          ***
* SIMULATION--                                                *
* THIS SUBROUTINE COMPUTES THE INVERSE OF AN ELEMENT OF       *
* A GALOIS FIELD. INPUT ELEMENT IS LEFT JUSTIFIED IN X1       *
* AND INVERSE IS RETURNED IN X2.                              *
***                                                          ***
INVERSE   EQU       $
          LW,R3     X1
          LW,R2     M
          SLS,R3    -NOBITS,R2
          LW,X2     INVM,R3
          B         *D2
```

```
***                                                                  ***
* SIMULATION--                                                        *
* THIS ROUTINE SIMULATES THE OPERATION OF THE SLAVE PRO-              *
* CESSORS.  R1 GIVES THE CURRENT SLAVE.                               *
***                                                                  ***
SLVRET   DATA     0
SLAVE    EQU      $
         STW,D2   SLVRET
         LI,R3    0
         LW,R2    REG2,R1
         BE       $+2
         LW,R3    LCOMP
         STW,R3   TLINE,R1
         MTB,0    SALN,R1
         BNE      ACTIVE
         LI,R2    0
         STW,R2   ACIN,R1
         STW,R2   TLINE,R1
         B        *SLVRET
ACTIVE   EQU      $
* DO MULTIPLICATION
         LW,D1    I1
         LW,X1    REG3,R1
         LW,X2    SREG0,R1
         BAL,D2   ISU
         STW,S1   X2
         LW,X1    REG1,R1
         LW,D1    I0
         BAL,D2   ISU
         STW,S1   ARG1
         LW,X1    REG2,R1
         LW,X2    MALINE
         BAL,D2   ISU
         STW,S1   X2
         LW,X1    ARG1
         BAL,D2   MULTXY
```

```
          STW,S1    MLTAC
          STW,S1    MLTOUT
          EOR,S1    REG2,R1
          STW,S1    ADDOUT
          BAL,D2    IRN
          BAL,D2    RSU
SLCLOCK   EQU       $
          MTW,0     SCL
          BE        CL1
          LW,R2     IREG0
          STW,R2    REG0,R1
CL1       EQU       $
          MTW,0     SCL+1
          BE        CL2
          LW,R2     IREG1
          STW,R2    REG1,R1
CL2       EQU       $
          MTW,0     SCL+2
          BE        CL3
          LW,R2     IREG2
          STW,R2    REG2,R1
CL3       EQU       $
          MTW,0     SCL+3
          BE        CL4
          LW,R2     IREG3
          STW,R2    REG3,R1
CL4       EQU       $
          MTW,0     SCL+4
          BE        *SLVRET
          LW,R2     IREG4
          STW,R2    REG4,R1
          B         *SLVRET
```

```
***                                                              ***
* SIMULATION--                                                    *
* THIS ROUTINE SIMULATES THE INTERNAL ROUTING NETWORK OF A        *
* SLAVE PROCESSOR.  GATE LOCATIONS ARE DEFINED BY AN ARRAY.       *
***                                                              ***
IRNSZ    EQU     16
IRNT     RES     IRNSZ
IRN      EQU     $
         LI,R2   IRNSZ
         LI,R3   0
         STW,R3  IRNT-1,R2
         BDR,R2  $-1
         MTW,0   I0
         BE      IRN1
         LW,R2   REG0,R1
         STW,R2  IRNT
         LW,R2   REG2,R1
         STW,R2  IRNT+3
         LW,R2   SREG0,R1
         STW,R2  IRNT+5
         LW,R2   SREG1,R1
         STW,R2  IRNT+6
         LW,R2   MLTAC
         STW,R2  IRNT+10
         B       IRNOR
IRN1     EQU     $
         MTW,0   I1
         BE      IRN2
         LW,R2   REG1,R1
         STW,R2  IRNT+2
         LW,R2   SREG1,R1
         STW,R2  IRNT+7
         LW,R2   ADDOUT
         STW,R2  IRNT+12
         B       IRNOR
IRN2     EQU     $
```

```
          MTW,0     I2
          BE        IRNIN
          MTB,0     SCLN,R1
          BNE       IRN3
          LW,R2     SREG1,R1
          STW,R2    IRNT+8
          B         IRN4
IRN3      EQU       $
          LW,R2     REG0,R1
          STW,R2    IRNT+1
          LW,R2     SREG1,R1
          STW,R2    IRNT+9
IRN4      EQU       $
          MTW,0     RSLINE
          BE        IRN5
          LW,R2     REG4,R1
          STW,R2    IRNT+4
          B         IRNOR
IRN5      EQU       $
          LW,R2     MLTOUT
          STW,R2    IRNT+11
          B         IRNOR
IRNIN     EQU       $
          LW,R2     ICON,R1
          STW,R2    IRNT+13
          LW,R2     SYN1,R1
          STW,R2    IRNT+14
          LW,R2     SYN2,R1
          STW,R2    IRNT+15
IRNOR     EQU       $
          LW,R2     IRNT+14
          OR,R2     IRNT+9
          OR,R2     IRNT+7
          STW,R2    IREG0
          LW,R2     IRNT+15
          OR,R2     IRNT+8
```

```
OR,R2     IRNT+1
STW,R2    IREG1
LW,R2     IRNT+13
OR,R2     IRNT+12
STW,R2    IREG2
LW,R2     IRNT+13
OR,R2     IRNT+11
OR,R2     IRNT+5
OR,R2     IRNT+4
STW,R2    IREG3
LW,R2     IRNT+6
STW,R2    IREG4
LW,R2     IRNT+13
OR,R2     IRNT+11
OR,R2     IRNT+4
STW,R2    R3
OR,R2     IRNT+3
STW,R2    ISREG0,R1
OR,R3     IRNT+2
OR,R3     IRNT
STW,R3    ISREG1,R1
LW,R2     IRNT+10
STW,R2    ACIN,R1
B         *D2
```

```
***                                                              ***
* SIMULATION--                                                    *
* THIS ROUTINE SIMULATES THE REGISTER SELECTION UNIT              *
* IN THE SLAVE PROCESSORS.                                        *
***                                                              ***
SCLSZ    EQU      7
SCL      RES      SCLSZ
RSU      EQU      $
         STW,R1   R1SAVE
         LW,R0    INITLN
         LW,R1    R0
         LW,R2    R0
         LW,R3    R0
         LW,R5    R0
         LW,R6    R0
         OR,R3    I0
         LW,R4    I0
         OR,R5    I0
         OR,R6    I0
         OR,R0    I1
         OR,R2    I1
         OR,R6    I1
         OR,R1    I2
         OR,R3    I2
         OR,R5    I2
         OR,R6    I2
         LCI      SCLSZ
         STM,R0   SCL
         LW,R1    R1SAVE
         MTB,0    SCLN,R1
         BE       *D2
         LW,R2    SCL
         OR,R2    I2
         STW,R2   SCL
         B        *D2
```

```
***                                                                  ***
* SIMULATION--                                                        *
* THIS SUBROUTINE COMPUTES THE PRODUCT OF TWO ELEMENTS                *
* OF A GALOIS FIELD. INPUT ELEMENTS ARE LEFT JUSTIFIED                *
* IN X1 AND X2. PRODUCT IS IN S1.                                     *
***                                                                  ***
MULTS1   DATA    0
MULTXY   EQU     $
         LI,S1   0
         STW,X2  MULTS1
         LW,D1   M
MULTBR   EQU     $
         SLS,S1  1
         BCR,8   $+2
         EOR,S1  IRPOLY
         SLS,X2  1
         BCR,8   $+2
         EOR,S1  X1
         BDR,D1  MULTBR
         LW,X2   MULTS1
         B       *D2
***                                                                  ***
* SIMULATION--INPUT SELECTOR UNIT                                     *
* THIS DEVICE PASSES X1 TO S1 IF THE SELECT LINEIN  D1 IS 1,          *
* AND PASSES X2 TO S1 IF THE SELECT LINE (D1) IS ZERO.                *
***                                                                  ***
ISU      LCW,R2  M
         LW,S1   D1
         SLS,S1  NOBITS-1
         SAS,S1  1,R2
         LW,S2   S1
         EOR,S2  COMP
         AND,S1  X1
         AND,S2  X2
         OR,S1   S2
         B       *D2
```

```
***                                                              ***
* UTILITY--THIS SUBPROGRAM DEFINES A NEW GALOIS FIELD             *
* ACCORDING TO THE VALUE IN LOCATION M.                           *
***                                                              ***
MEXP     DATA     0
AO       DATA     0
GFRET    DATA     0
GFINIT   EQU      $
         STW,D2   GFRET
         LW,R2    M
         LW,R2    IRPTAB,R2
         STW,R2   IRPOLY
         LI,R1    1
         S,R1     *M
         STW,R1   MEXP
         LI,R1    1
         LCW,R2   M
         SLS,R1   NOBITS,R2
         STW,R1   AO
         LW,R2    M
         LW,R1    IRPOLY
         SLS,R1   -NOBITS,R2
         OR,R1    MEXP
         STW,R1   IRPOLYI
         LI,R1    0
         STW,R1   INVM
         LW,R1    AO
         STW,R1   INVM+1
         LW,R2    SDSV
         STW,R1   ICON,R2
         LW,R2    LCOMP
INVBR    EQU      $
         CW,R2    LCOMP
         BAZ      $+2
         EOR,R2   IRPOLYI
         SLS,R2   -1
```

```
         CI,R2     1
         BE        *GFRET
         SLS,R1    1
         BEV       $+2
         EOR,R1    IRPOLY
         STW,R1    INVM,R2
         B         INVBR
***                                                              ***
* UTILITY--PRINTS THE SPECIFIED HEADING.                          *
***                                                              ***
PRINT    EQU       $
PRINT    COM,8,4,3,17,15,17  X'6A',X'D',0,CF(1),0,AF(1)
         LW,R2     *D2
         AI,D2     1
         LW,R3     MOVBL
         MBS,R3    -1
         LW,R3     MOVBL
         AI,R3     -1
         LB,R4     *R2
         STB,R4    R3
         SLS,R2    2
         MBS,R2    1
         CAL1,1    PRBUF
         LW,R3     MOVBL
         SLS,R3    -2
         LI,R4     X'40'
         STB,R4    *R3
         B         *D2
```

```
***                                                              ***
* UTILITY--ACCEPTS A CODEWORD FROM THE M:SI DEVICE OR FILE.       *
* ALSO RECOGNIZES THE COMMANDS LISTED  BELOW.                     *
* EOD      MEANS END OF DATA                                      *
* RESTART  RETURN TO THE SLAVE ACTIVATION MODE                    *
* T        MEANS CHANGE THE TREG TO THE VALUE FOLLOWING T         *
* M        MEANS CHANGE THE M REGISTER TO THE VALUE FOLLOWING M   *
* FILE     MEANS SWITCH THE ECHO MODE UNDER BTM.                  *
***                                                              ***
IORET    DATA     0
INOUT2   EQU      $
         STW,D2   IORET
         CAL1,1   TPAGE
         PRINT    PRHD4
         B        INOUT1
INOUT    EQU      $
         STW,D2   IORET
         CAL1,1   TPAGE
         PRINT    PRHD1
INOUT1   EQU      $
         LW,R1    MOVCB
         MBS,R1   -4
         LI,R1    BUFSZ
         LW,R2    ADRS
         STW,R2   RADR
         STW,R2   WADR
         LI,R2    BUFTOP
IOBK     EQU      $
         CAL1,1   READ
         MTW,0    BTMSW
         BNE      $+2
         CAL1,1   WRITE
         LB,R3    *RADR
         CB,R3    ECK
         BE       IOHLT
         CB,R3    RCK
```

```
          BE        START
          CB,R3     TCK
          BE        TINPUT
          CB,R3     MCK
          BE        MINPUT
          CB,R3     FCK
          BE        FILIN
          AWM,R1    RADR
          AWM,R1    WADR
          CW,R2     RADR
          BG        IOBK
          PRINT     ERRM4
          B         REDO
IOHLT     EQU       $
          LW,R2     RADR
          STW,R2    IOLAST
          B         *IORET
TINPUT    EQU       $
          BAL,D2    INTNUM
          CI,R5     NOSLAVE
          BL        STORT
          PRINT     ERRM2
          B         INOUT1
STORT     EQU       $
          STW,R5    TREG
          B         INOUT1
MINPUT    EQU       $
          BAL,D2    INTNUM
          STW,R5    M
          BAL,D2    GFINIT
          B         INOUT1
FILIN     EQU       $
          DO1       1-BTM
          B         INOUT1
          LW,R3     BTMSW
          EOR,R3    LCOMP
```

```
          STW,R3    BTMSW
          B         INOUT1
INTNUM    EQU       $
          LB,R3     XFO
          LI,R4     0
CINT      EQU       $
          AI,R4     1
          CI,R4     10
          BG        IOBK
          CB,R3     *RADR,R4
          BGE       CINT
          LI,R5     0
NXINT     EQU       $
          LB,R3     *RADR,R4
          CB,R3     XFO
          BL        *D2
          AND,R3    CHMSK
          MI,R5     10
          AW,R5     R3
          AI,R4     1
          B         NXINT
***                                                          ***
* UTILITY--CODEWORD IS PRINTED ON THE M:LO DEVICE.             *
***                                                          ***
OUTRET    DATA      0
OUTCW     EQU       $
          STW,D2    OUTRET
          LW,R2     M
          CI,R2     MAXPAGE
          BL        $+2
          CAL1,1    TPAGE
          PRINT     PRHD2
          LI,R1     BUFSZ
          LW,R2     ADRS
          STW,R2    WADR
          LW,R3     IOLAST
```

```
NEXTCRD  EQU      $
         CAL1,1   WRITE
         AWM,R1   WADR
         CW,R2    WADR
         BNE      NEXTCRD
         B        *OUTRET
***                                                          ***
* REGISTERS REG0, REG1 AND REG2 ARE LISTED ON THE             *
* M:LO DEVICE WHEN THIS ROUTINE IS CALLED.                    *
***                                                          ***
PRRET    DATA     0
PROUT    EQU      $
         STW,D2   PRRET
         CAL1,1   TPAGE
         PRINT    PRHD3
         LW,R3    MOVBL
         MBS,R3   -1
         LB,D1    XFO
         LI,R1    NOSLAVE
PRBK     EQU      $
         LW,R2    SYN1-1,R1
         LW,R3    SYN2-1,R1
         LW,R4    REG2-1,R1
         LW,R5    LCOMP
         SLS,R5   31
         LI,R6    0
         LI,R7    1
PRBK2    EQU      $
         STB,D1   OUTBUF,R7
         STB,D1   OUTBUF+OUTFMT,R7
         STB,D1   OUTBUF+OUTFMT+OUTFMT+1,R7
         CW,R2    R5
         BAZ      $+2
         MTB,1    OUTBUF,R7
         CW,R3    R5
         BAZ      $+2
```

```
         MTB,1    OUTBUF+OUTFMT,R7
         CW,R4    R5
         BAZ      $+2
         MTB,1    OUTBUF+OUTFMT+OUTFMT+1,R7
         AI,R7    2
         SLS,R5   -1
         AI,R6    1
         CW,R6    M
         BL       PRBK2
         CAL1,1   PRBUF
         BDR,R1   PRBK
         B        *PRRET
***                                                          ***
* UTILITY--THIS ROUTINE SETS UP THE ACTIVE SLAVES ACCORDING    *
* TO THE INPUT STATEMENT.                                      *
***                                                          ***
ACTRET   DATA     0
ACTSL    EQU      $
         STW,D2   ACTRET
         BAL,D2   INOUT2
         LI,R1    0
         LI,S1    NOSLAVE
         AI,S1    3
         SLS,S1   -2
         LI,R3    0
ACTST    EQU      $
         LW,R2    CODEBUF,R1
         CW,R2    BLNK
         BE       ACTCL
         AND,R2   SELSL
         STW,R2   SALN,R1
         STW,R3   SCLN,R1
         AI,R1    1
         CW,R1    S1
         BL       ACTST
ACTCL    EQU      $
```

```
          LI,R2     NOSLAVE
          LI,R3     0
          STW,R3    ICON-1,R2
          BDR,R2    $-1
          LW,R2     LCOMP
          LW,R3     A0
          LI,R1     -1
          AI,R1     1
          MTB,0     SALN,R1
          BE        $-2
          STW,R1    SOSV
          STB,R2    SCLN,R1
          STW,R3    ICON,R1
          AI,R1     1
          MTB,0     SALN,R1
          BE        $-2
          STW,R1    S1SV
          STB,R2    SCLN,R1
          B         *ACTRET
***                                                          ***
* UTILITY--THIS ROUTINE PROVIDES THE FIRST CIRCULAR            *
* SHIFT OF THE PARITY CHECKS.                                  *
***                                                          ***
PSHIFT    EQU       $
          LI,R1     0
          LI,R3     0
          LI,R5     0
PSHN      EQU       $
          MTB,0     SALN,R3
          BE        PSHBYP
          CI,R5     0
          BNE       PSET1
          AI,R5     1
          STW,R1    SOSV
          B         PSET
PSET1     EQU       $
```

```
          CI,R5     1
          BG        PSET
          AI,R5     1
          STW,R1    S1SV
PSET      EQU       $
          XW,R2     R4
          XW,R2     SYN2,R1
PSHBYP    EQU       $
          AI,R3     1
          CI,R3     NOSLAVE
          BL        $+2
          LI,R3     0
          AI,R1     1
          CI,R1     NOSLAVE+NOSLAVE
          BL        PSHN
          LW,R1     SOSV
          STW,R4    SYN2,R1
          LW,R1     S1SV
          STW,R2    SYN2,R1
          B         *D2
***                                                              ***
* UTILITY--REGISTER DUMPING ROUTINE--PROVIDES A SNAPSHOT DUMP     *
* IN HEXADECIMAL NOTATION OF THE REGISTERS SELECTED.              *
***                                                              ***
XCO       DATA      X'00000CO'
RXFO      DATA      '   0'
CHSAVE    DATA      0
BUFDMP    EQU       $
          CAL1,1    TPAGE
          LW,R3     MOVBL
          MBS,R3    -1
          LW,R2     *D2
          AI,R2     -1
          AI,D2     1
          LW,R3     *D2
          AI,D2     1
```

```
BUFBK3   EQU     $
         LI,R4   NOSLAVE
         CI,R4   30
         BLE     $+2
         LI,R4   30
BUFBK1   EQU     $
         LW,R7   *R2,R4
         LI,R5   1
BUFBK2   EQU     $
         LI,R6   0
         SLD,R6  4
         CI,R6   9
         BLE     $+4
         AI,R6   -9
         OR,R6   XCO
         B       $+2
         OR,R6   RXFO
         STB,R6  CHSAVE,R5
         AI,R5   1
         CI,R5   3
         BLE     BUFBK2
         LW,R6   CHSAVE
         STW,R6  OUTBUF,R4
         BDR,R4  BUFBK1
         CAL1,1  PRBUF
         AI,R2   NOSLAVE
         CW,R2   R3
         BLE     BUFBK3
         B       *D2
         END     START
```

```
ACTIVE SLAVE CONFIGURATION:
1111111111
EOD

RECEIVED WORD:
M=7
T=9
0000100000000010000000001000000000100000000000000000000000000000001000000000
0000100000100000000000100000000000100000000000
EOD

CORRECTED WORD:
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000

RECEIVED WORD:
M=4
T=1
100110000000100
EOD

CORRECTED WORD:
100110000000000
```

```
RECEIVED WORD:
RESTART
ACTIVE SLAVE CONFIGURATION:
1111110000
EOD

RECEIVED WORD:
M=5
T=5
0100000100010000010010000000000
EOD

CORRECTED WORD:
0000000000000000000000000000000

----------SYNDROME POLYNOMIAL---------- --SIGMA POLYNOMIAL--
      0 0 0 0 0          0 0 0 0 0          0 0 0 0 0
      0 0 0 0 0          0 0 0 0 0          0 0 0 0 0
      0 0 0 0 0          0 0 0 0 0          0 0 0 0 0
      0 0 0 0 0          0 0 0 0 0          0 0 0 0 0
      1 1 0 1 1          0 0 1 0 0          0 0 0 1 0
      0 0 1 0 1          1 0 1 1 1          1 1 1 0 1
      0 0 0 1 0          0 0 0 0 0          0 1 1 0 0
      0 0 1 1 1          0 0 0 0 0          0 0 0 1 1
      1 0 0 0 1          0 0 0 0 1          0 1 1 0 1
      0 0 1 1 1          0 1 1 0 1          0 0 0 0 1

RECEIVED WORD:
EOD

*EXIT*
```

BIBLIOGRAPHY

1. Berlekamp, E. R., Algebraic Coding Theory, McGraw-Hill Book Company, New York, 1968.

2. Levitt, K. N., and W. H. Kautz, "Cellular Arrays for the Parallel Implementation of Binary Error-Correcting Codes," IEEE Transactions on Information Theory, Vol. IT-15, September, 1969, pp. 597-607.

3. Cannon, L. C., "A Cellular Computer to Implement the Kalman Filter Algorithm," Ph. D. Thesis, Montana State University, August, 1969.

4. Gallager, R. G., Information Theory and Reliable Communication, John Wiley and Sons, Inc., New York, 1968.

5. Bartee, T. C., and D. I. Schneider, "An Electronic Decoder for Bose-Chaudhuri-Hocquenghem Error-Correcting Codes," IRE Transactions on Information Theory, Vol. IT-8, September, 1962, pp. S17-24.

6. Minnick, R. C., et al; "Cellular Logic," Final Report, SRI Project 5087, AFCRL Contract AF 19(628)-4233; AFCRL 66-613, April, 1966.

7. McCluskey, E. J., Introduction to the Theory of Switching Circuits, McGraw-Hill Book Company, 1965.

8. Peterson, W. W., "Encoding and Error-Correction Procedures for the Bose-Chaudhuri Codes," IRE Transactions on Information Theory, Vol. IT-6, September, 1960, pp. 459-470

9. Chien, R. T., "Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes," IEEE Transactions on Information Theory, Vol. IT-10, October, 1964, pp. 357-363.

10. Seller, F. F., M. Hsiao, and L. W. Bearnson, Error Detecting Logic for Digital Computers, McGraw-Hill Book Company, 1968.

11. Minnick, R. C., "A Survey of Microcellular Research," Journal of the Association for Computing Machinery, Vol. 14, April, 1967, pp. 213-241.

12. Peterson, W. W., Error-Correction Codes, MIT Press and John Wiley and Sons, Inc., New York, 1961.

13. Bartee, T. C., and D. I. Schneider, "Computation with Finite Fields," Information and Control, Vol. 6, 1963, pp. 79-98.

MONTANA STATE UNIVERSITY LIBRARIES
3 1762 10010899 0

D378
L44
cop.2
Laws, Benjamin A.
A parallel BCH decoder

NAME AND ADDRESS

D378
L44
Cop. 2