THE UNIVERSITY OF ALBERTA

DECODING REED-SOLOMON CODES WITH A SIGNAL PROCESSOR

by

DAVID J. PETERSON

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL ENGINEERING

EDMONTON, ALBERTA

SPRING 1987

ISBN 0-315-37794-1

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled DECODING REED-SOLOMON CODES WITH A SIGNAL PROCESSOR submitted by DAVID J. PETERSON in partial fulfilment of the requirements for the degree of MASTER OF SCIENCE.

C. G. Englefield
Co-supervisor

[illegible]
Co-supervisor

[illegible]

[illegible]

Date 21st April 1987

Abstract

This thesis considers the implementation of a decoder for certain types of Reed-Solomon codes. A relatively simple implementation based on the Texas Instruments TMS32010 signal processor chip is shown to provide performance comparable to that of more complex bit slice designs. Secondly, a synchronization technique based on the Viterbi Algorithm is presented and analyzed. This technique provides character and frame synchronization without relying on a phase locked loop or related analog circuitry.

Preface

The Reed-Solomon codes are a powerful class of error correcting codes that have found recent application in audio compact disks [1], CD-ROM devices [2], space communications [3,4], and high speed text transmission [5]. While the design of Reed-Solomon encoder hardware is relatively straightforward, the design of decoders involves several critical issues. Four basic decoder architectures have been extensively studied to date:

(a) microprocessor-based designs [16,17];

(b) microprogrammed/bit slice CPU [16,17];

(c) microprogrammed/discrete logic CPU [16,17,37];

(d) custom VLSI [17,36,38].

The speeds of these implementations vary dramatically. The bit slice design provides a throughput over ten times higher than that of an '8086-based decoder. In turn, the discrete logic and custom VLSI designs are roughly an order of magnitude faster than the bit slice design.

The arithmetic used in the processing of Reed-Solomon codes takes place in a Galois field and is quite different from the familiar integer or complex arithmetic. In fact, the implementation of Galois field multiplication is a major issue in the design of decoders. The microprocessor decoder mentioned above requires many machine cycles to carry out a multiplication while the other designs implement Galois field multiplication with a large custom circuit.

It is well known that the arithmetic of *prime* Galois fields can be routinely implemented with standard integer hardware. Unfortunately, the unavailability of fast economical integer processors seems to have made this approach to Reed-Solomon coding unattractive to earlier designers. In fact, all of the designs mentioned above are based on a Galois field of *characteristic 2*, which is quite different from a prime field. However, the recent introduction of the Texas Instruments TMS32010, a 5 MIP 16 bit signal processor chip, suggests that the use of prime fields should be investigated further. In this thesis, it will be shown that the arithmetic of certain types of prime Galois fields can be implemented very efficiently with the TMS32010.

The major topics of the thesis are the following:

(1) The relationships between several versions of the Reed-Solomon decoding algorithm are examined in Chapter 1.

(2) A TMS32010 implementation of a decoding algorithm is presented in Chapter 2. The performance of this system is compared to that of earlier implementations based on microprocessors and bit slice components. The '32010 is shown to provide a throughput close to that of the more cumbersome bit slice design mentioned in (b) above. The replacement of the '32010 with recently introduce~~d~~ signal processor devices such as the TMS320C25, the Motorola DSP56000, or the TMS320C30 would increase the throughput by a factor of two

to more than four. While a decoder based on one of these powerful signal processors would still be several times slower than the discrete logic (c) or custom VLSI (d) decoders, it would be more flexible and convenient than the latter designs.

(3) Any implementation of an error correcting code requires the encoder and decoder to be synchronized. This is typically done with a combination of analog and digital techniques. A method of inserting timing information into the encoded data and recovering it at the decoder without any analog processing is presented and its performance is analyzed in Chapter 3. Theoretical analysis of the timing recovery technique yields interesting upper bounds on the probability of synchronization failure for some types of multilevel (ie: nonbinary) communication channels. A '32010 implementation of this technique for the binary channel is also discussed in Chapter 3.

(4) A TMS32010-based hardware system is designed and built as part of this thesis. The programming and operation of the system is detailed in a separate report [39]. The approach taken in this thesis is to develop and analyze general algorithms and to make reference to [39] for the machine code used to implement these algorithms. The running time of any time-critical routine is also discussed.

Table of Contents

List of Tables

List of Figures

Chapter 1

The Decoding of Reed-Solomon Codes

The Reed-Solomon codes were devised by Reed and Solomon in 1960 [6]. Since the inception of these codes, much research has been devoted to the problem of decoding them in the presence of errors. While the Reed-Solomon codes are optimal in a certain theoretical sense [7,Chap.11], the main justification for their use in practical systems is the relative ease with which they can be decoded. In this chapter, Reed-Solomon codes are defined and a slight variant of Berlekamp's algorithm for decoding them [8,Chaps.7,10] is presented. Some modifications of this algorithm are also discussed.

1.1 An introduction to Reed-Solomon codes

1.1.1 The field GF(11)

The concepts of *field* and *primitive root of unity* are the most fundamental concepts underlying Reed-Solomon codes and algebraic coding theory in general. Probably the most succinct description of a field to be found anywhere in the literature appears in [9,p.674]:

> "Field: An algebraic system admitting addition, subtraction, multiplication, and division."

Some technical details are missing from or hidden in this definition. For example, a field always contains an additive identity 0 and a multiplicative identity 1, and the usual associative and commutative laws of multiplication and addition hold.

The most pervasive example of a field is the set of complex numbers with the usual addition, subtraction, multiplication, and division operations. A less familiar example is the set of integers {0,1,2,...,10} with addition ("⊕") and multiplication ("⊗") defined as follows:

$$a \oplus b \equiv \text{remainder obtained when } a+b \text{ is divided by } 11;$$

$$a \otimes b \equiv \text{remainder obtained when } a \times b \text{ is divided by } 11.$$

⊕ and ⊗ are called *modulo 11 (mod 11)* addition and multiplication. For *any* two integers a and b, one writes *a = b mod 11* if 11 is a divisor of a−b.

Note that $5 \oplus 6 = 0$, $5 \otimes 9 = 1$ (6 is the additive inverse of 5 and 9 is the multiplicative inverse of 5). In general, any integer between 1 and 10 has a multiplicative inverse and an additive inverse which also lies between 1 and 10. Hence the integers {0,1,2,...,10} form a field under mod 11 addition and multiplication. This field is called a *Galois field of order 11* and is written as *GF(11)*.

The first ten powers of 2 in GF(11) are:

$$2^1 = 2$$

$$\begin{array}{lll} 2^2 = 2 \otimes 2 = 4 & 2^5 = 10 & 2^8 = 3 \\ 2^3 = 2^2 \otimes 2 = 8 & 2^6 = 9 & 2^9 = 6 \\ 2^4 = 2^3 \otimes 2 = 5 & 2^7 = 7 & 2^{10} = 1 \end{array}$$

Hence, every nonzero element of GF(11) can be represented as a power of 2. For this reason, 2 is called a *primitive element* of GF(11). This property of the number 2 will prove to be quite useful.

1.1.2 A single error correcting code over GF(11)

Suppose one wishes to transmit a seven number message over an unreliable channel (for example, a telephone number "transmitted" by word-of-mouth). How can the message be protected against errors?

Consider the message *4323848*. This message can be represented as the first seven coefficients of a *codeword polynomial* C(x):

$$C(x) = 4x^8+3x^7+2x^6+3x^5+8x^4+4x^3+8x^2+ax+b$$

where integers a and b are chosen so that

$$C(2) = C(4) = 0 \bmod 11.$$

Hence, a = 6, b = 1. The nine coefficients *432384861* of C(x) are then transmitted over the channel.

Suppose a single error occurs during transmission. For example, suppose the receiver obtains the message *432334861*. Writing the received message as a polynomial R(x):

$$R(x) = 4x^8+3x^7+2x^6+3x^5+3x^4+4x^3+8x^2+6x+1,$$

the receiver can compute the *syndromes*

$$R(2) = 8 \bmod 11,$$
$$R(4) = 7 \bmod 11.$$

It will now be shown how the syndromes can be used to correct the error.

When a single error affecting the i^{th} coefficient of $C(x)$ occurs, the received message $R(x)$ can be written as

$$R(x) = C(x) \oplus Ex^i,$$

where $E \in GF(11)$, $E \neq 0$. Hence,

$$R(2) = C(2)\oplus[E\otimes 2^i] = E\otimes 2^i \bmod 11,$$
$$R(4) = C(4)\oplus[E\otimes 4^i] = E\otimes 4^i \bmod 11.$$

These mod 11 equations imply that

$$2^i = R(4)\otimes R(2)^{-1} \bmod 11,$$
$$E = R(2)^2\otimes R(4)^{-1} \bmod 11.$$

In the case under consideration, the first of the latter equations implies

$$2^i = 7\otimes 8^{-1} = 7\otimes 7 = 5,$$

and the results of 1.1.1 in turn imply that i = 4. The second equation implies

$$E = 8^2 \otimes 7^{-1} = 6.$$

Now that i and E have been found, the receiver recovers the original codeword C(x) by subtracting Ex^i from R(x):

$$C(x) = 432334861 - 000060000 = 432384861 \bmod 11.$$

The recovery of C(x) from R(x) is called *decoding*.

The set of all polynomials

$$C(x) = \sum_{i=0}^{8} c_i x^i$$

with coefficients in GF(11) and which satisfy C(2) = C(4) = 0 mod 11 is called a *single error correcting Reed-Solomon code of block length 9 over GF(11)*. In the remainder of this chapter, the definition and decoding of multiple error correcting Reed-Solomon codes over arbitrary fields is discussed.

1.2 Reed-Solomon codes over arbitrary fields

The example of 1.1.1 shows that fields need not have an infinite number of elements. In general, any field having a finite number of elements q is called a *Galois field of order q* and is represented by the symbol *GF(q)*. The number of elements q in a Galois field must always be a power of a prime number [8,p.102]. The prime number is called the *characteristic* of the field.

Let n be a positive integer. A field element α is called a *primitive* n^{th} *root of unity* if n is the smallest positive integer for which $\alpha^n=1$. $\exp[2\pi i/n]$ is a primitive n^{th} root of unity in the field of complex numbers. Unlike the complex field, a Galois field does not contain primitive n^{th} roots of unity for every positive integer n. Specifically, GF(q) contains a primitive n^{th} root of unity if and only if n is a divisor of q-1 [8,pp.88-90]. The primitive $q-1^{th}$ roots of unity in GF(q) are called the *primitive field elements* of GF(q). It has already been demonstrated that 2 is a primitive field element of GF(11).

Let *F* be a field containing a primitive n^{th} root of unity α. For any positive integer $t < n/2$, the set of polynomials C(x) having coefficients in *F* and which satisfy

(i) $\deg C(x) < n$,

(ii) $C(\alpha)=C(\alpha^2)= \ . \ . \ . =C(\alpha^{2t})=0$,

is called a *t error correcting Reed-Solomon code of block length n over F*. The polynomials C(x) satisfying (i) and (ii) are called the *codewords* of the code. The special codeword

$$g(x)=(x-\alpha)(x-\alpha^2) \ . \ . \ . \ (x-\alpha^{2t})$$

is called the *generator polynomial* of the code. Every codeword is a polynomial multiple of g(x).

The reference to error correction is motivated by the following communication example. It is desired to transmit a sequence $f_0, f_1, \ldots, f_{n-2t-1}$ of symbols in F to some receiver. The prospective communicators have access to a communication channel capable of transmitting symbols in F, but the channel occasionally changes a transmitted symbol f to some other symbol f' in F by the time it reaches the receiver. The communicators can alleviate this problem by agreeing that the transmitter will compute the Reed-Solomon codeword

$$C(x) = \sum_{i=0}^{n-2t-1} f_i x^{i+2t} + \sum_{i=0}^{2t-1} p_i x^i,$$

where the p_i terms are chosen so that C(x) satisfies condition (ii) above. The transmitter then sends the coefficients of C(x) over the channel. The computation of C(x) is called *encoding* and can be implemented with a simple shift register circuit [8,Chap.5], [13,Chap.6]. The f_i terms are called the *information symbols* of the codeword. The p_i terms are called *parity symbols*.

Some of the coefficients of C(x) might be received in error due to the channel, so the receiver will obtain a polynomial R(x) = C(x) + E(x), where the number of nonzero coefficients of E(x) is equal to the number of channel errors. In this section it will be shown that if no more than t errors occur, the receiver can determine the error polynomial E(x) from the knowledge of R(x) alone. After E(x) has been so determined, it can be subtracted from R(x) to give C(x). The receiver can then read the first n-2t coefficients of C(x) to obtain the original sequence $f_0, f_1, \ldots, f_{n-2t-1}$. The recovery of the original transmitted sequence from the received word R(x) is called *decoding*.

1.2.1 On the choice of the field

The typical communication channel is a binary channel, so the elements of the field *F* over which the Reed-Solomon code is defined must be represented as a finite sequence of binary digits. This can be conveniently done by using a field of the form $GF(2^m)$. Each element of this field can be represented as an m bit symbol. Another motive for restricting *F* to be a Galois field is that there is no such thing as round-off error in a Galois field computer. The effect of round-off error on the decoding of a Reed-Solomon code over the complex numbers or any other infinite field does not appear to have been studied. Nonetheless, Reed-Solomon codes over certain infinite fields have been proposed for encoding two-dimensional data arrays [18] and for impulse noise cancellation [19].

Reed-Solomon codes over $GF(2^m)$ are useful for binary channels on which the noise manifests itself as intermittent bursts of bit errors. For example, a burst of m consecutive bit errors will affect at most two consecutive characters of a codeword. Hence, it will be treated as only a single or double character error. Reed-Solomon codes are often *concatenated* with less sophisticated codes to take advantage of this property [4],[5],[13,Chap.7],[16,Chap.3]. The characters of several consecutive Reed-Solomon codewords can also be *interleaved* for further protection against very long burst errors [1],[3],[37]. This technique is used in the system described at the end of Chapter 3.

The arithmetic of the fields GF(2^m) is quite different from integer arithmetic or complex arithmetic [8,Chaps.2, 4]. However, arithmetic in some of the fields GF($2^m \pm 1$), where $2^m \pm 1$ is prime, can be implemented fairly easily with standard integer hardware. This will be demonstrated in Chapter 2 with $2^m \pm 1$ = 127 and 257.

The use of GF($2^m \pm 1$) can present some inconveniences. The information to be encoded is typically a stream of binary data. The encoder must segment this stream into information symbols and encode consecutive blocks of symbols into codewords. Suppose a field of the form GF($2^m - 1$) is used. If the encoder segments the binary data stream into symbols of l bits, one must have $l \leq m-1$ to guarantee that all possible information symbols can be represented as field elements. However, the parity symbols added by the encoder may take any value in GF($2^m - 1$) and will require m bits each. Similarly, when GF($2^m + 1$) is used, any m bit sequence is allowed as an information symbol, but the resulting parity symbols may require m+1 bits. In the latter case, the problem can be eliminated by using a simple method due to Solomon [20]. This method ensures that the parity symbols of any codeword can be represented as m bit characters. For reasonably small values of t, it involves a modest increase in the complexity of the encoder. Solomon's method reduces the number of information symbols in the codeword by one, but each of the remaining information symbols may take any one of 2^m possible values. No modifications of the accompanying $GF(2^m+1)$ decoding algorithm are required. By modifying the t error correcting code of length 2^m over $GF(2^m+1)$ in this way, one obtains a code that is virtually equivalent to the t error correcting code of length 2^m-1 over $GF(2^m)$.

1.2.2 The key equation

If $a_0, a_1, a_2, \ldots$ belong to a field F, the expression

$$A(z) = \sum_{i=0}^{\infty} a_i z^i$$

is called a *generating function* over F. If $A(z)$ and $B(z)$ are generating functions, their sum is defined as

$$A(z) + B(z) = \sum_{i=0}^{\infty} (a_i + b_i) z^i, \qquad (1.1)$$

and their product is defined as

$$A(z)B(z) = \sum_{i=0}^{\infty} \left(\sum_{j=0}^{i} a_{i-j} b_j \right) z^i. \qquad (1.2)$$

It is clear that any polynomial $f(z)$ over F can be regarded as a generating function with finitely many nonzero coefficients, and that (1.1) and (1.2) coincide with polynomial addition and multiplication if A(z) and B(z) are polynomials.

It can be shown that if $a_0 \neq 0$, then there exists a unique generating function I(z) such that [8,p.73]

$$A(z)I(z) = z^0 + 0z + 0z^2 + \ldots = 1.$$

I(z) is called the *multiplicative inverse* of A(z) and is written as 1/A(z). It is easy to verify that

$$1/(1-az) = 1 + az + a^2z^2 + \ldots \qquad (1.3)$$

for any a belonging to *F*.

If the generating functions A(z) and B(z) agree in the first n coefficients, one writes *A(z) = B(z) mod z^{n+1}*. It is straightforward to verify that if A(z) = B(z) mod z^{n+1}, then

$$C(z)A(z) = C(z)B(z) \bmod z^{n+1} \qquad (1.4)$$

for any generating function C(z).

The generating function idea can be applied to the communication example described at the beginning of sec. 1.2 as follows [8,pp.218-221]. Suppose that errors occur during transmission of the i_1th, i_2th, . . . , i_eth coefficients c_i of the codeword C(x). Then the error polynomial can be written as

$$E(x) = Y_1 x^{i_1} + Y_2 x^{i_2} + \ldots + Y_e x^{i_e}$$

where the Y_i terms, called the *error magnitudes*, are non-zero. Define the *syndromes* S_k of $E(x)$ as

$$S_k = E(\alpha^k), \; k = 1,2,3, \ldots$$

where α is the primitive n^{th} root appearing in the definition of the Reed-Solomon codes. Then

$$S_k = Y_1 X_1^k + Y_2 X_2^k + \ldots + Y_e X_e^k,$$

where $X_j = \alpha^{i_j}$ for each j.

Defining the generating function $S(z) = \sum_{k=1}^{\infty} S_k z^k$, the previous expression for S_k implies

$$\begin{aligned} 1 + S(z) &= 1 + \sum_{k=1}^{\infty} \sum_{j=1}^{e} Y_j X_j^k z^k \\ &= 1 + \sum_{j=1}^{e} Y_j \sum_{k=1}^{\infty} X_j^k z^k \\ &= 1 + \sum_{j=1}^{e} Y_j X_j z/(1-X_j z), \end{aligned}$$

from (1.3). Now defining the polynomial

$$\sigma(z) = \prod_{j=1}^{e} (1-X_j z), \tag{1.5}$$

and multiplying both sides of the previous equality by $\sigma(z)$, one has

$$[1 + S(z)]\sigma(z) = \sigma(z) + \sum_{j=1}^{e} Y_j X_j z \prod_{k\neq j}(1-X_k z). \qquad (1.6)$$

The polynomial $\sigma(z)$ is called the *error locator polynomial*. The polynomial appearing on the right hand side of (1.6) is called the *error evaluator polynomial* and is written as $\omega(z)$.

The receiver can compute the first 2t syndromes from the received word R(x):

$$S_k = E(\alpha^k) = R(\alpha^k), \ 1 \le k \le 2t,$$

because $C(\alpha^k) = 0$ for these values of k. From (1.4) and (1.6),

$$(1 + \sum_{k=1}^{2t} S_k z^k)\sigma(z) = \omega(z) \bmod z^{2t+1}. \qquad (1.7)$$

It will be shown in 1.2.3 that the receiver can use (1.7) to determine the polynomials $\sigma(z)$ and $\omega(z)$ from the syndromes $S_1, S_2, \ldots, S_{2t}$ if the number of errors e satisfies $e \le t$. Once the receiver has found $\sigma(z)$, it can determine the error locations $i_1, i_2, \ldots, i_e$ by evaluating $\sigma(\alpha^{-i})$ for each $0 \le i \le n-1$ and saving the i's for which $\sigma(\alpha^{-i}) = 0$. From (1.5), i is an error location if and only if $\sigma(\alpha^{-i}) = 0$. This search procedure is known as the *Chien search*. Finally, the receiver can compute the error magnitudes Y_i from $\omega(z)$ and the newly determined X_i's as follows. From the expression for $\omega(z)$ in (1.6),

$$\omega(X_i^{-1}) = \sigma(X_i^{-1}) + \sum_{j=1}^{e} X_i^{-1} X_j Y_j \prod_{k \neq j} (1 - X_k X_i^{-1})$$

$$= 0 + Y_i \prod_{k \neq i} (1 - X_k X_i^{-1}),$$

implying that

$$Y_i = \omega(X_i^{-1}) / \prod_{k \neq i} (1 - X_k X_i^{-1}). \qquad (1.8)$$

This completes the determination of the error polynomial E(x).

Equation (1.7) is called the *key equation* [8,p.179]. The solution of the key equation for $\sigma(z)$ and $\omega(z)$ given the S_i's is discussed from various perspectives in [8,Chaps.7, 10], [10], [11], [12], [13,Chaps.7-9], [14], and [15, Chap. 8]. A simplified version of Berlekamp's algorithm for solving the key equation [8,Algorithm 7.4] is presented below. The derivation of this algorithm is somewhat different from the derivation of Berlekamp's Algorithm 7.4, so it is given in full detail in Appendix 1.

1.2.3 Solving the key equation

A polynomial of the form $f(z)=z^m+f_{m-1}z^{m-1}+ \ldots +f_1z+f_0$ (ie: a polynomial having leading coefficient 1) is called a *monic* polynomial. The *greatest common divisor (gcd)* of two polynomials A(z) and B(z) is the monic divisor of both A(z) and B(z) having largest degree. The gcd of A(z) and B(z) is written as (A(z),B(z)). The *Euclidean Algorithm* [8,pp.25-27] can be used to compute the gcd of two polynomials. An occasionally useful consequence of this algorithm is that there exist polynomials a(z), b(z) such that a(z)A(z) + b(z)B(z) = (A(z),B(z)).

Two polynomials A(z), B(z) are said to be *relatively prime* if (A(z),B(z))=1. Consider the polynomials $\sigma(z)$, $\omega(z)$ introduced in 1.2.2. Any polynomial divisor of $\sigma(z)$ having positive degree must have one of the X_i^{-1} terms as a root. From (1.8), $\omega(X_i^{-1}) \neq 0$ for every X_i, so this divisor of $\sigma(z)$ cannot also be a divisor of $\omega(z)$. It follows that any polynomial divisor of both $\sigma(z)$ and $\omega(z)$ must have degree zero, implying that $\sigma(z)$ and $\omega(z)$ are relatively prime.

The following result is a straightforward variation of [10,Theorem 1] and is stated and proved as Claim 2 of Appendix 1.

Claim: Let S(z) be a generating function over a field *F* such that S(0) = 0. Suppose polynomials $\sigma(z)$ and $\omega(z)$ over *F* satisfy

(1) $(1 + S)\sigma = \omega \bmod z^p$,

(2) $\sigma(0) = 1$,

(3) $(\sigma,\omega) = 1$,

for some positive integer p (note that the generating function S(z) and the polynomials $\sigma(z)$, $\omega(z)$ introduced in 1.2.2

satisfy these conditions for every p).

Suppose nonzero polynomials $\sigma'(z)$, $\omega'(z)$ can be found (by whatever means possible) such that

(1') $(1 + S)\sigma' = \omega' \bmod z^p$,

(2') $\sigma'(0) = 1$,

(3') $\max[\deg \sigma', \deg \omega'] \leq \max[\deg \sigma, \deg \omega]$.

Then if $\max[\deg \sigma, \deg \omega] < p/2$, we have $\sigma' = \sigma$ and $\omega' = \omega$.█

Suppose $S(z)$, $\sigma(z)$, $\omega(z)$, and p are as described in the first part of the previous claim. We now present an algorithm that, when applied to $S(z)$, produces a sequence of polynomials $\sigma_0(z)$, $\omega_0(z)$, $\sigma_1(z)$, $\omega_1(z)$, ..., $\sigma_{p-1}(z)$, $\omega_{p-1}(z)$ such that for each $0 \leq k \leq p-1$:

(a) $(1 + S)\sigma_k = \omega_k \bmod z^{k+1}$,

(b) $\sigma_k(0) = 1$,

(c) $\max[\deg \sigma_k, \deg \omega_k] \leq \max[\deg \sigma, \deg \omega]$.

If $\max[\deg \sigma, \deg \omega] < p/2$, the claim implies $\sigma_{p-1}(z) = \sigma(z)$ and $\omega_{p-1}(z) = \omega(z)$. The relevance of this to the decoding of Reed-Solomon codes is seen by considering the decoding of a noisy received word $R(x) = C(x) + E(x)$, where $C(x)$ is a codeword of a t error correcting Reed-Solomon code. Taking

$$S(z) = \sum_{i=1}^{2t} S_i z^i$$

and $p = 2t+1$, where $S_1, S_2, \ldots, S_{2t}$ are the first 2t syndromes of $E(x)$, the key equation (1.7) implies that conditions (1), (2), (3) of the claim are satisfied when $\sigma(z)$ is the error locator polynomial and $\omega(z)$ is the error evaluator polynomial. Furthermore, it is evident that if e errors occur, then

$$\max[\deg \sigma, \deg \omega] = e.$$

Therefore, if $e \leq t$, then

$$\max[\deg \sigma, \deg \omega] < (2t+1)/2 = p/2.$$

It follows that the application of this algorithm to $S(z)$ will yield $\sigma_{2t}(z) = \sigma(z)$ and $\omega_{2t}(z) = \omega(z)$, thereby solving the key equation.

Berlekamp's Iterative Algorithm:

(1) Set $\sigma_0 = 1$, $\omega_0 = 1$, $\tau_0 = 1$, $\gamma_0 = 0$, $D(0) = 0$, $k = 0$.

(2) If ' $= p-1$, stop. Otherwise, define

$$\sigma_{k+1} = \sigma_k - \Delta_k z \tau_k,$$
$$\omega_{k+1} = \omega_k - \Delta_k z \gamma_k,$$

where Δ_k is the coefficient of z^{k+1} in the product

$$[1 + S(z)]\sigma_k(z).$$

(3) If $\Delta_k = 0$ or if $D(k) \geq (k+1)/2$, put $D(k+1) = D(k)$ and define

$$\tau_{k+1} = z\tau_k,$$
$$\gamma_{k+1} = z\gamma_k.$$

But if $\Delta_k \neq 0$ and $D(k) < (k+1)/2$, put $D(k+1) = k+1-D(k)$ and define

$$\tau_{k+1} = \Delta_k^{-1}\sigma_k,$$
$$\gamma_{k+1} = \Delta_k^{-1}\omega_k.$$

(4) Increment k and return to (2).

The fact that the polynomials produced by this algorithm satisfy (a), (b), and (c) is proved in Appendix 1.

1.2.4 Remarks

(1) Berlekamp's Iterative Algorithm is a simpler version of Berlekamp's Algorithm 7.4 [8,p.184]. Both algorithms produce polynomials σ_k, ω_k, τ_k, γ_k, $0 \le k \le p-1$, such that $\sigma_{p-1} = \sigma$ and $\omega_{p-1} = \omega$ whenever

$$\max[\deg \sigma, \deg \omega] < p/2.$$

However, simple examples show that the intermediate polynomials σ_k, ω_k, τ_k, γ_k, $k < p-1$, produced by the two algorithms do not always agree.

By removing the calculations involving ω_k and γ_k from Berlekamp's Iterative Algorithm, one obtains the algorithm presented by Massey [11]. Blahut [13,p.180] calls the latter algorithm the *Berlekamp-Massey Algorithm*. This algorithm computes $\sigma(z)$ by working through the σ_k and τ_k polynomials alone. After execution, the coefficients of $\omega(z)$ can be computed from the original mod z^p equation

$$[1 + S(z)]\sigma(z) = \omega(z) \bmod z^p.$$

This approach is slightly faster and more memory efficient than the approach involving all the σ_k, ω_k, τ_k, γ_k polynomials. An implementation of the Berlekamp-Massey Algorithm is discussed in 2.2.3 below.

(2) If $\max[\deg \sigma, \deg \omega] \ge p/2$, the behavior of these algorithms is difficult to predict. If more than t errors

occur during the transmission of a codeword of the t error correcting Reed-Solomon code, it cannot be said with certainty whether or not the polynomials σ_{2t}, ω_{2t} produced by Berlekamp's Iterative Algorithm coincide with the error locator polynomial σ and the error evaluator polynomial ω. Hence, the ideal response to such an error pattern would be for the receiver to indicate "decoding failure" and to either accept the received word R(x), as bad as it is, or to ask the transmitter to send the codeword again. While such a response cannot be guaranteed for all large error patterns (consider the case where the error polynomial E(x) is a nonzero codeword), Berlekamp's Iterative Algorithm has two easily recognized "failure modes":

(i) if D(k) exceeds t for some k, then equation (1.19) of Appendix 1 implies that more than t errors have occurred.

(ii) if $\sigma_{2t}(z)$ doesn't have D(2t) distinct roots of the form α^{-i}, $0 \le i \le n-1$, then (1.19) implies that more than t errors have occurred.

Thus, by monitoring the size of the D(k) terms during execution of the algorithm and by counting the number of roots obtained during the Chien search, some incorrectable error patterns can be recognized.

(3) Berlekamp's Algorithm 7.4 and its variations described above are not the only known methods of solving the key equation (1.7). By applying the Euclidean Algorithm to the polynomials

$$z^{2t}, \quad S_{2t}z^{2t-1} + S_{2t-1}z^{2t-2} + \ldots + S_1$$

one obtains finite sequences of polynomials $\sigma_k'(z)$, $\omega_k'(z)$ converging to $\sigma(z)$ and $\omega(z)$, respectively, whenever no more than t channel errors occur. This method is described in [7,sec.12.8,12.9], [14], and [15,Chap.8]. In fact, according to [7,p.369],

". . . decoding using the Euclidean Algorithm is by far the simplest (method) to understand."

Whether or not this is true, implementations of the Euclidean Algorithm tend to be slower than implementations of the Berlekamp-Massey Algorithm [7,p.369], [15,p.261], [16, Appendix A].

1.2.5 Speeding up the solution of the key equation

It has been shown that the key equation (1.7) can be solved by applying Berlekamp's Iterative Algorithm to the generating function

$$S(z) = \sum_{i=1}^{2t} S_i z^i$$

with p = 2t+1. If fewer than t errors occur in the received word, it is possible to terminate the algorithm before k reaches 2t. Specifically, if σ is the error locator polynomial, ω is the error evaluator polynomial, and $e \le t$ errors occur, then

$$\max[\deg \sigma, \deg \omega] = e < (2e+1)/2.$$

By the claim of 1.2.3, this implies

$$\sigma_{2e} = \sigma_{2e+1} = \ldots = \sigma_{2t} = \sigma,$$

and

$$\omega_{2e} = \omega_{2e+1} = \ldots = \omega_{2t} = \omega.$$

Hence, the algorithm could have been stopped anytime after the 2eth iteration, saving execution time if $e < t$. Chen [12] gives a condition ensuring an exit from the Berlekamp-Massey Algorithm after t+e iterations. It is shown in Appendix 2 that the same exit condition stops Berlekamp's Iterative Algorithm after t+e iterations.

Chen's exit condition can be stated as follows. The replacement of the original exit condition

"If $k = 2t$, stop"

in Berlekamp's Iterative Algorithm with the exit condition

"If $D(k) = k-t$ or if $k = 2t$, stop".

will ensure the correction of any pattern of up to t errors and an exit after exactly t+e iterations of the algorithm, where e is the actual number of errors. As already noted in 1.2.4, the Berlekamp-Massey Algorithm is a "subalgorithm" of Berlekamp's Iterative Algorithm, so this condition will cause an early exit from the Berlekamp-Massey Algorithm as well. This was first proved by Chen [12].

The number of computations involved in the kth iteration of any of these algorithms is proportional to k. Therefore, the use of this "early exit" modification reduces the total computational load by the factor

$$\sum_{k=1}^{t+e} k \Big/ \sum_{k=1}^{2t} k = (t+e)(t+e+1)/(2t)(2t+1).$$

Hence, if e is small compared to t, the modified algorithm computes σ and ω in approximately one quarter of the time required by the original algorithm.

The following potential problem with this speedup technique is not mentioned in [12]: for any error pattern of e errors, $e \le t$, there is a different error pattern of at most t+e+1 errors such that the two patterns have the same syndromes

$$S_1, S_2, \ldots, S_{t+e}.$$

This "aliasing" property is proved in Appendix 3. Its significance is that when the second error pattern occurs, the use of the early exit condition will cause the production of the error locator polynomial σ and the error evaluator polynomial ω corresponding to the first error pattern. The subsequent steps in the decoding procedure then erroneously compute the error positions and error magnitudes of the first error pattern and subtract the corresponding error polynomial from the received word. The resulting "corrected" word has as many as $t+2e+1$ errors. However, if the original version of Berlekamp's Iterative Algorithm was used, there is a chance that one of the failure modes mentioned in 1.2.4 would occur and the received word would at least be recognized as being beyond repair. Considering the special case $e = 0$, this argument says that there are error patterns of $t+1$ errors that an early exit - modified algorithm will mistake for the zero error condition. The only nonzero error patterns that Berlekamp's Iterative Algorithm could mistake for the zero error condition are themselves Reed-Solomon codewords, which always have at least $2t+1$ nonzero components [13,pp.9,10].

The design of a microprocessor-based decoder for the six error correcting Reed-Solomon code of block length 63 over GF(127) is discussed in Chapter 2. This early exit technique is not used there because the execution of the original Berlekamp-Massey Algorithm constitutes less than 20% of the running time of the entire decoding procedure. Over 70% of the running time is devoted to the calculation of the syndromes $S_1, S_2, \ldots, S_{12}$ and to the Chien search.

Chapter 2

A Microprocessor-Based Reed-Solomon Decoder

The purpose of this chapter is to give a detailed description of a Reed-Solomon decoder design based on the Texas Instruments TMS32010 Signal Processor chip. The design is intended for codes over certain Galois fields of the form GF($2^m \pm 1$), where $2^m \pm 1$ is a prime number. Attention will be restricted to the fields GF(127) and GF(257). In section 2.1, the major hardware components of the decoder system are described. Section 2.2 covers the programming of the decoder system. Some of the more technical hardware details are contained in [39]. It will be seen that this system has a codeword throughput rate comparable to that of a more cumbersome design based on a microprogrammed bit slice architecture [16,Chap.7].

2.1 Overview of decoder hardware

The decoder hardware can be represented as shown in fig. 2.1. The function and structure of each component shown in fig. 2.1 except for Boot ROM and Buffer B is described below. Boot ROM, Buffer B, and the user interface circuitry are detailed in [39]. Devices connected to the external data buses will be described in Chapter 3.

2.1.1 TMS32010 Digital Signal Processor

The major features of the TMS32010 chip are the following:

(1) 200 ns machine cycle;

(2) 16 bit instruction and data words;

(3) single machine cycle 16 by 16 bit multiplier;

(4) 0-15 bit barrel shifter;

(5) 32 bit ALU and accumulator;

(6) 144 16 bit data registers DAT0 - DAT143.

The program to be executed by the '32010 is stored in Boot ROM and transferred to Program Memory before the '32010 is activated. Given that a program is resident in Program Memory, the '32010 executes the instruction stored at address A ("instruction A") by placing A on the Address Bus and reading the resulting Program Memory output from the Data Bus into an internal 16 bit instruction register. The instruction register contents then drive the '32010's internal circuitry (such as the components described in (3)-(6) above) as specified by instruction A. The only instructions requiring more than a single machine cycle are branches, data transfers to and from external devices on the Data Bus, and subroutine calls and returns. The execution sequence is halted by applying a low voltage (0.0 - 0.8 V) to the device's RS (reset) pin. Once the voltage on the RS pin returns to the high state (2.0 - 5.0 V), the '32010 begins execution at Program Memory address 0.

The 16 by 16 bit multiplier circuit is capable of multiplying two 16 bit two's complement operands and storing the result in a 32 bit *P (product) register* in a single

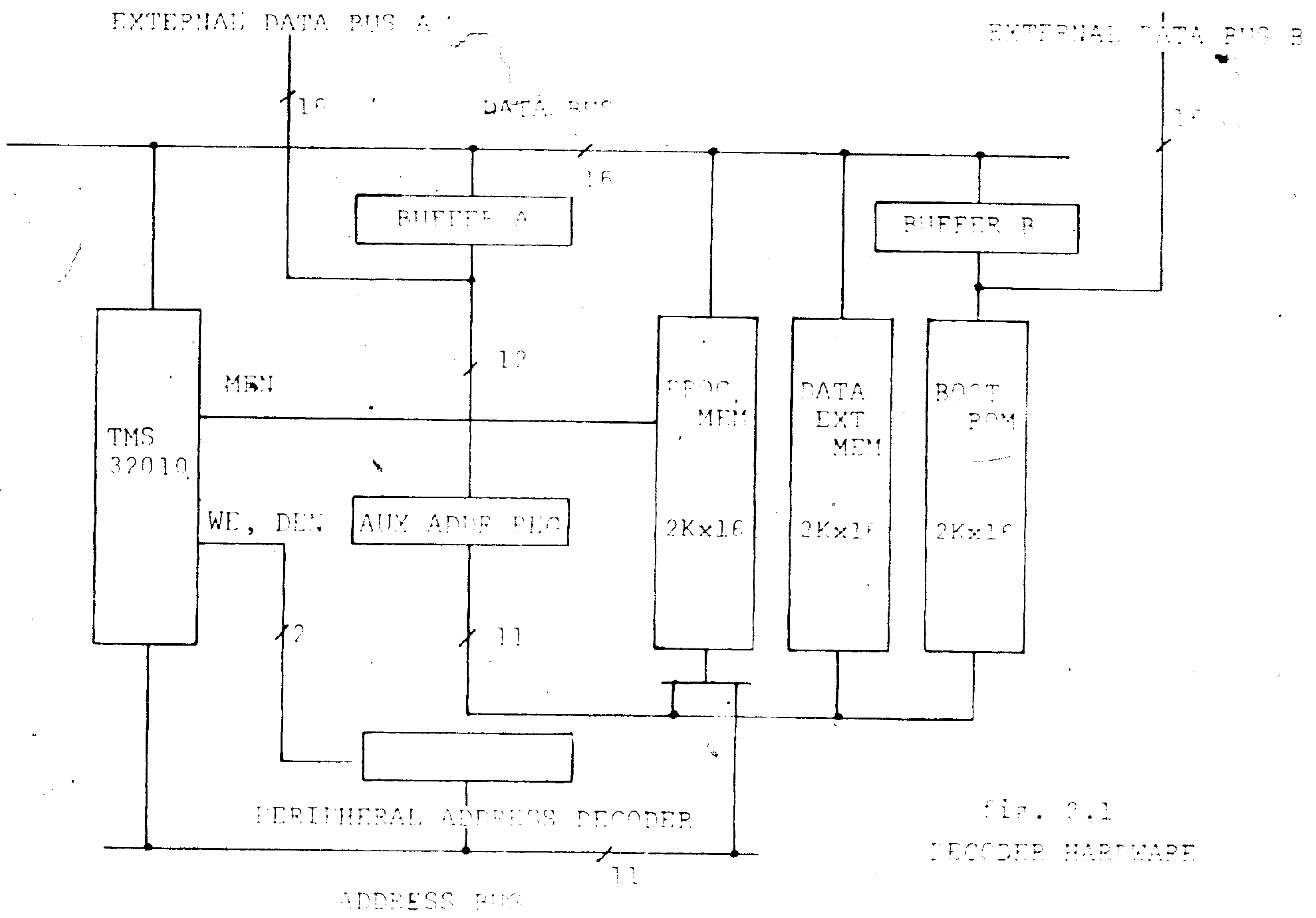


fig. 2.1
DECODER HARDWARE

machine cycle. One of the operands is always obtained from a register called the *T register*. The source of the other operand depends on the instruction. The "MPY DATi" instruction specifies that the other operand is the entry of data register DATi, while the "MPYK c" instruction specifies that the other operand is the number c embedded within the MPYK instruction word.

The barrel shifter circuit is used in conjunction with addition, subtraction, and accumulator initialization instructions. It allows a data register entry to be shifted left by 0 to 15 bits prior to adding it to or subtracting it from the present accumulator entry, or prior to storing it in the accumulator. For example, the "ADD DATi, x" instruction specifies that the contents of DATi are to be shifted left by x bits and that the shifted term is to be added to the present accumulator entry. The "LAC DATi, x" instruction specifies that the contents of DATi are to be shifted left by x bits and that the shifted term is to be loaded into the accumulator. The barrel shifter is useful for fast floating point arithmetic operations, but it will also prove to be valuable for arithmetic operations in the types of Galois fields mentioned in the introduction to this chapter.

The ALU is capable of applying standard arithmetic and logic operations ("ADD", "SUB", "AND", "OR", "XOR") to 32 bit operands from the accumulator, the barrel shifter, or the P register. The result of an ALU operation is stored in the accumulator. Any ALU operation is executed in a single machine cycle.

The data registers DAT0 - DAT143 provide the T register, multiplier, and barrel shifter with operands. Data is supplied to the data registers from the accumulator or from off-chip devices. For example, the "SACL DATi" instruction stores the 16 least significant bits of the accumulator in DATi. Writing the 32 bit accumulator entry as $2^{16-y}x_1 + x_2$, where $0 \leq x_2 \leq 2^{16-y}-1$, the "SACH DATi, y" instruction stores the 16 least significant bits of x_1 in DATi. The only values of y for which the latter instruction is defined are $y = 0, 1,$ or 4. The SACH DATi, 4 instruction is part of the fast polynomial evaluation routine over GF(127) or GF(257) described in 2.2.1 below.

The DATi entry can be placed on the Data Bus with the "OUT DATi" instruction. An external device (such as Data Extension Memory) can then read DATi from the Data Bus. Conversely, a value placed on the Data Bus by an external device can be loaded into DATi with the "IN DATi" instruction. Both the IN and OUT instructions require two machine cycles. Thus, in the interest of speed, programs should be designed so that frequently used operands are stored in the data registers DAT0 —DAT143 rather than in devices external to the '32010. The significant effect of these I/O instructions on the '32010's FFT execution speed is illustrated in [21].

Further details on the TMS32010 hardware will be presented when they are required. Full details on the device are given in [22].

2.1.2 Peripheral Address Decoder

As explained above, the '32010 communicates with external devices (other than Program Memory) by using IN and OUT instructions. The Peripheral Address Decoder is used to control the device with which the '32010 exchanges data during the execution of one of these I/O instructions.

The '32010 is capable of communicating with 8 *input devices* and 8 *output devices*. Each device is assigned a 3 bit *port address* which is specified in any I/O instruction involving communication with that device. For example, in the OUT DATi instruction used to transfer DATi to the output device at port j, 3 of the 16 bits in the instruction are reserved for j and the instruction is written as "OUT DATi, PAj". During the first machine cycle of an I/O instruction, the instruction fetch process described at the beginning of 2.1.1 is carried out. Once the instruction has been loaded into the instruction register, it will be recognized as an I/O instruction by the end of the machine cycle. At the beginning of the second machine cycle, the '32010 places the port address specified in the struction on address lines ADDR0-ADDR2 and holds all other address lines in the low voltage state. The second machine cycle is completed as follows:

(i) On an IN DATi, PAj instruction, the '32010 drops its DEN (data enable) pin from the high state to the low state. DEN returns to the high voltage state and the '32010 reads the Data Bus at the end of the second machine cycle.

(ii) On an OUT DATi, PAj instruction, the '32010 drops its WE (write enable) pin from the high state to the low state and places the DATi entry on the Data Bus. WE returns to the high voltage state and the '32010 stops driving the Data Bus at the end of the second machine cycle.

The '32010 fetches the next instruction in the program during the next machine cycle.

The Peripheral Address Decoder converts the voltage conditions on the Address Bus and the WE, DEN lines into a signal directed toward the desired I/O device. A realization of the Peripheral Address Decoder is suggested in [22] and is reproduced in fig. 2.2.

At the start of the second machine cycle of the OUT DATi, PAj instruction, j appears on the A0-A2 pins of the LS138 devices and the E_3 pins are in the high voltage state. The subsequent voltage transition on WE causes pin E_2 of LS138(1) to go low. The O_j pin of this device responds by falling to the low voltage state while all the other O_i pins remain high. Thus, by terminating O_j to the output device with port address j, a voltage pulse is transmitted to this device on the second cycle of the instruction. The device must be designed to read the Data Bus in response to this voltage pulse, thereby completing the data transfer. Similarly, input devices are activated by voltage pulses from LS138(2) and must be designed to drive the Data Bus in response to these pulses.

The circuitry driving the E_3 pins is required for the execution of the "TBLW" (table write) and "TBLR" (table read) instructions. These instructions, involving data transfer between the '32010 and Program Memory, are avoided because they each require 3 machine cycles. Full details on TBLW and TBLR are given in [22]. The operation of the circuitry driving the E_3 pins should be fairly obvious once an understanding of these instructions is obtained.

2.1.3 Buffer A and the Auxiliary Address Register

Buffer A and the Auxiliary Address Register are shown in fig. 2.3. Whenever any of the A_W, E_W, F_W, G_W, or H_W lines from the Peripheral Address Decoder are pulsed low by an OUT instruction, the Data Bus voltages are momentarily reproduced on External Data Bus A. The output device specified in the OUT instruction reads External Data Bus A at this time. The buffer is required because the '32010 is physically incapable of driving the Data Bus if all of the I/O devices are directly connected to it [22]. The B/T2 line is used for the test purposes described in [39]. This line is held in the high voltage state during program execution.

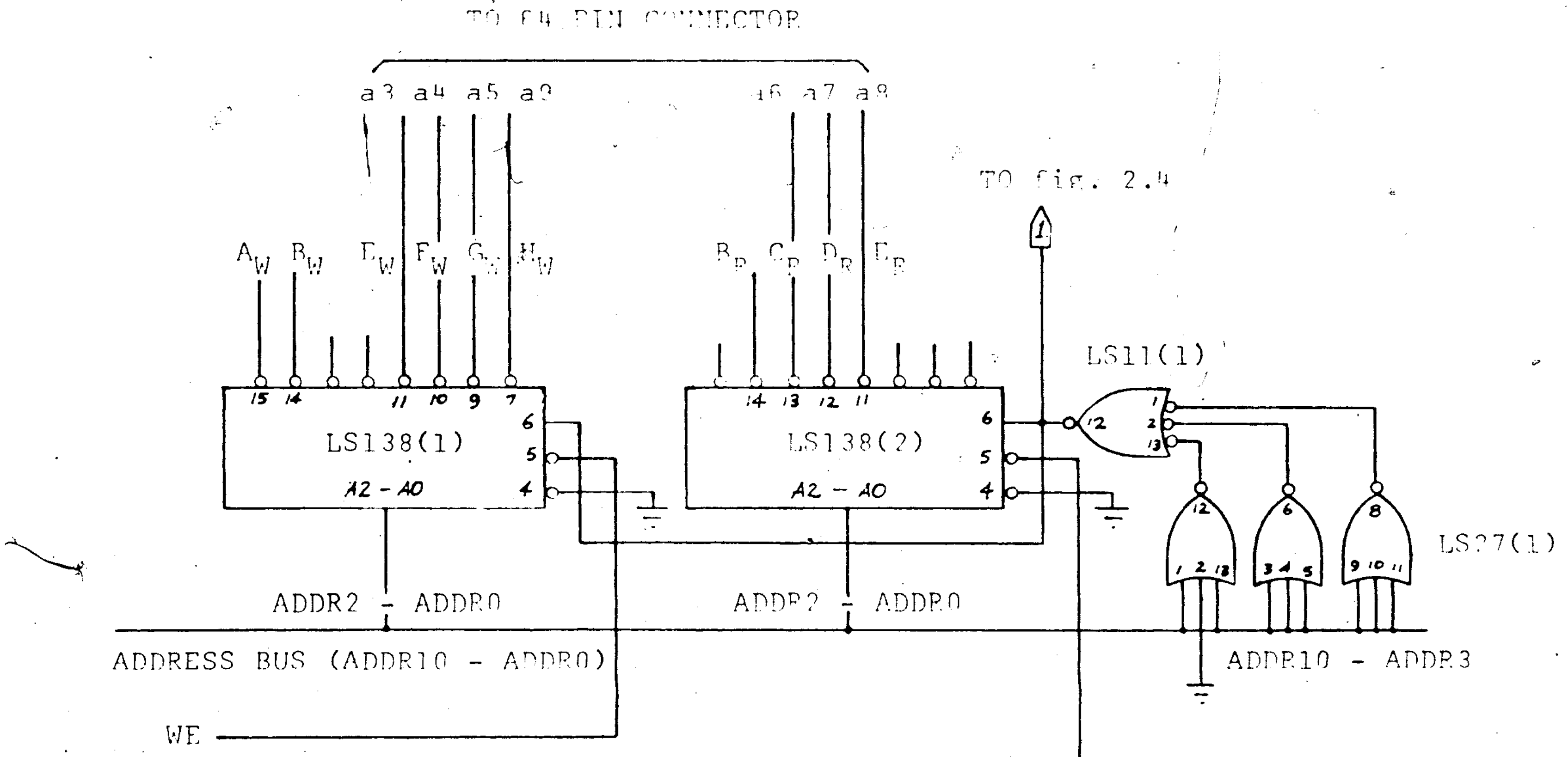


fig. 2.2
PERIPHERAL ADDRESS DECODER

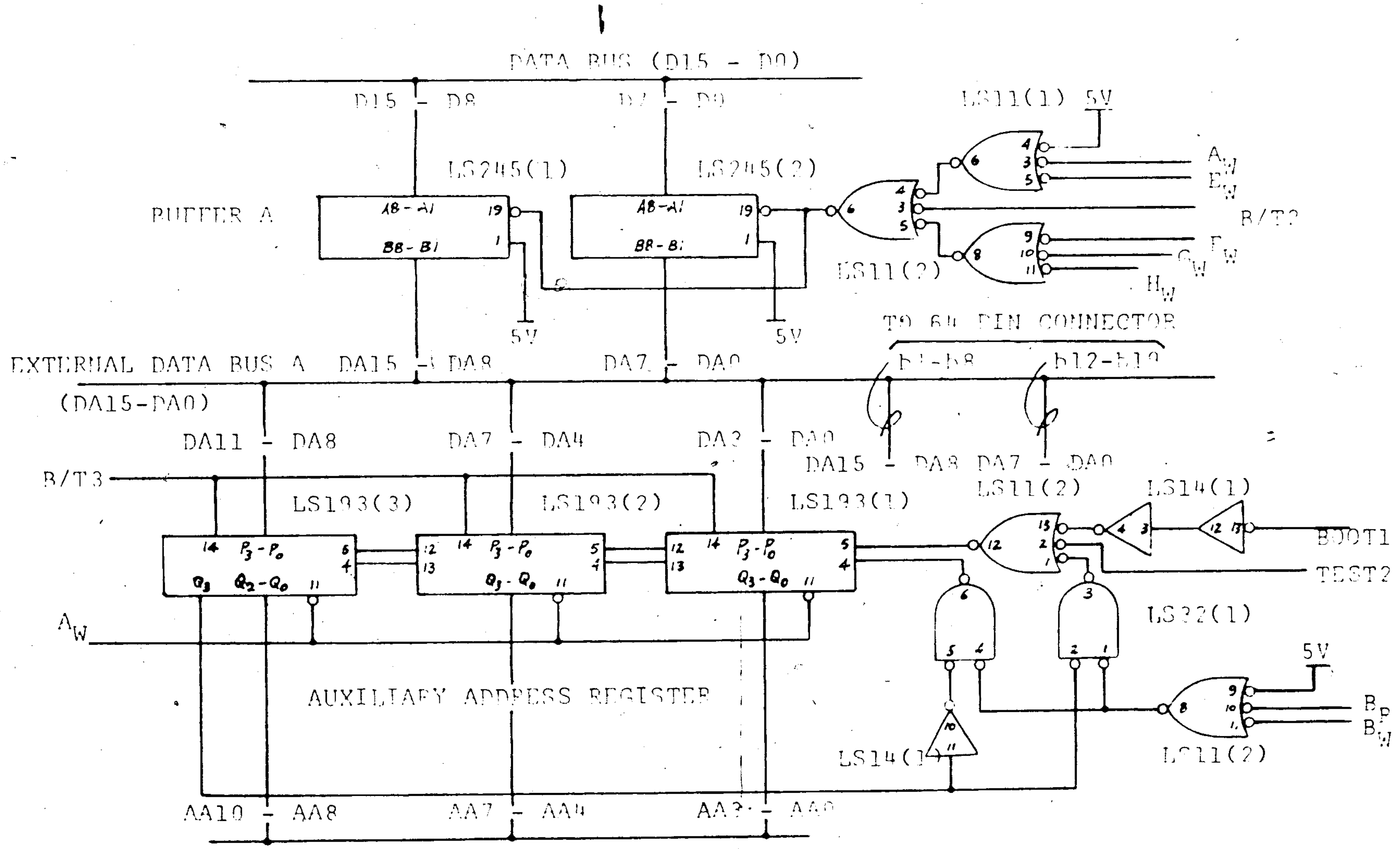


fig. 2.3
BUFFER A / AUXILIARY ADDRESS REGISTER

The Auxiliary Address Register is a 12 bit storage register. It is an output device and is assigned the port address 0. The OUT DATi, PA0 instruction causes the 12 least significant bits of the DATi entry to be written into the register.

The Auxiliary Address Register has two major functions: it is used in the transfer of a program from Boot ROM to Program Memory prior to execution, and it is used to address Data Extension Memory during execution. The first of these functions is not essential to the present discussion and is described in [39]. Concerning the second, the 11 least significant bits of the register address the 2k×16 Data Extension Memory. The most significant bit is used to determine the next state of the register after the '32010 reads from or writes to Data Extension Memory. If this bit is a zero, the register is automatically incremented at the end of a Data Extension Memory access, while if the bit is a one, the register is decremented. The purpose of these "autoincrement" and "autodecrement" features is to eliminate the need for time consuming OUT instructions to port address 0 when Data Extension Memory locations are accessed sequentially.

The Auxiliary Address Register can be manipulated directly by the user via the B/T3 and TEST2 lines. This capability is useful for testing Program Memory and Data Extension Memory and is described in [39]. During execution the TEST2 line is held in the high voltage state while the B/T3 line is held low.

2.1.4 Program Memory

Program Memory consists of two MCM2016H-55 2k×8 static RAM chips [23] and a small number of discrete logic circuits as shown in fig. 2.4. These high speed memory components are used because of the stringent timing requirements of the TMS32010 [22]. The Program Memory address has two possible sources: the Address Bus or the Auxiliary Address Register. The Address Bus addresses Program Memory during program execution while the Auxiliary Address Register addresses the memory during the transfer of a program from Boot ROM to Program Memory prior to execution. As mentioned in 2.1.3, the details of the latter situation are discussed in [39]. The functions of the TEST1 and BOOT1 lines are also described in [39]. During execution these lines are maintained in the high voltage state.

The flow of instructions from Program Memory to the '32010 instruction register along the Data Bus is regulated by the signal on the '32010 MEN (memory enable) pin. The signal on MEN is reproduced (with a time delay) at the E (chip enable) and G (output enable) pins of the Program Memory chips. The MEN pin is in the high voltage state at the beginning of each machine cycle, causing the Program Memory chips to be disabled. During an instruction fetch, the '32010 puts MEN into the low voltage state early in the machine cycle and the resulting low voltage on the E and G pins activates the Program Memory chips. The memory chips then drive the Data Bus with the memory entry corresponding to the present state of the address lines. If an instruction fetch is not to be carried out during the machine cycle (for example, during the second cycle of an I/O instruction), MEN is held in the high voltage state for the duration of the cycle. The Program Memory chips remain in the disabled state for the entire machine cycle, thereby freeing the Data Bus for communication between the '32010 and external devices.

The LS32 gate and the two lower AS08 gates in fig. 2.4 are activated during the TBLW instruction. As mentioned in 2.1.2, this is of no concern here because TBLW will never be used. The operation of these components should be clear from the description of TBLW given in [22].

2.1.5 Data Extension Memory

Data Extension Memory, like Program Memory, consists of two MCM2016H-55 2k×8 static RAMs and a small number of discrete logic circuits as shown in fig. 2.5. Data Extension Memory is both an input device and an output device and is assigned the port address 1. Thus the OUT DATi, PA1 instruction writes DATi into the Data Extension Memory location addressed by the Auxiliary Address Register. IN DATi, PA1 loads DATi with the Data Extension Memory entry addressed by the Auxiliary Address Register. As indicated in 2.1.3, the Auxiliary Address Register is automatically incremented or decremented at the end of each such Data Extension Memory access.

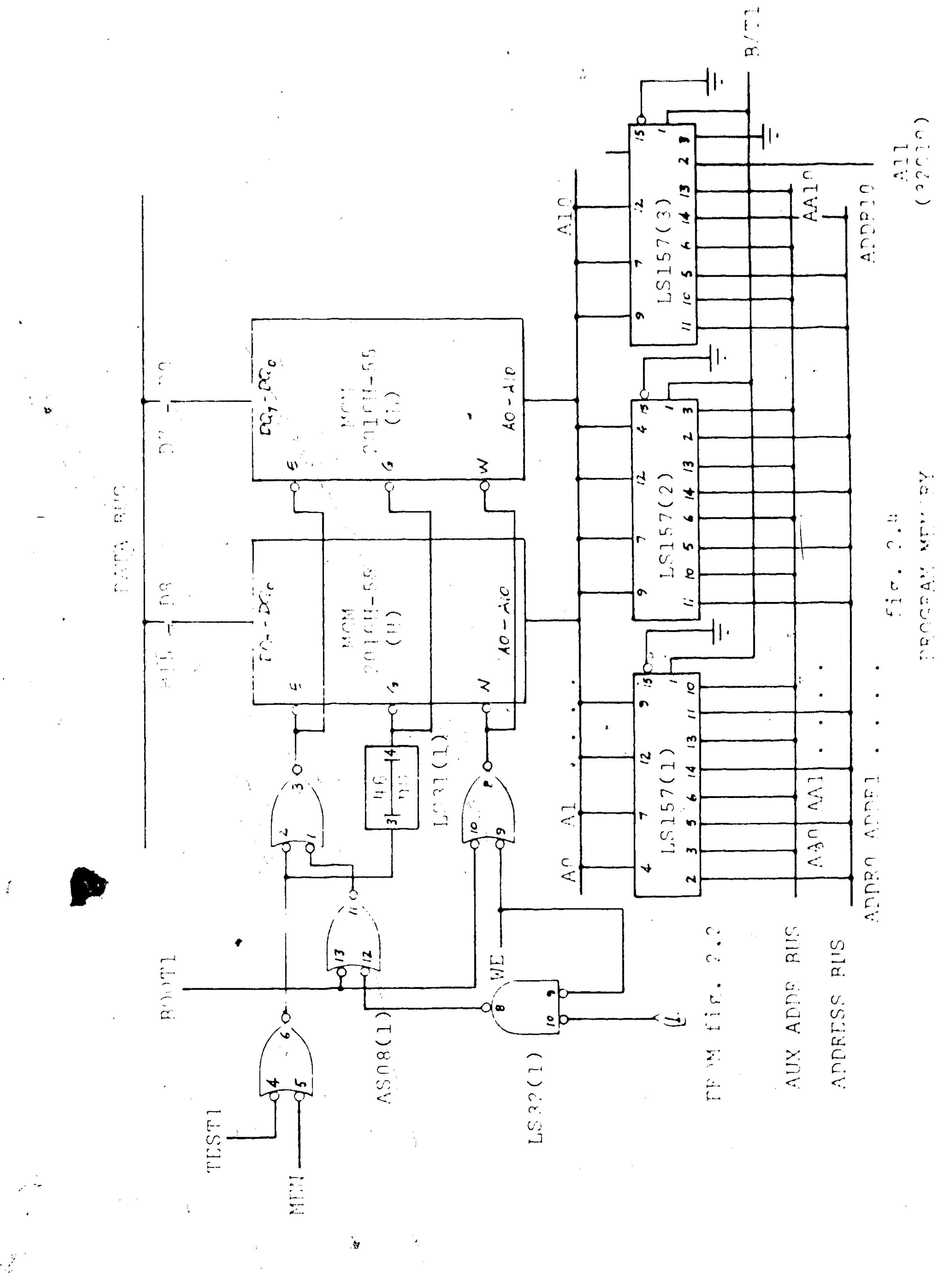


Fig. 2.8
PROGRAM MEMORY

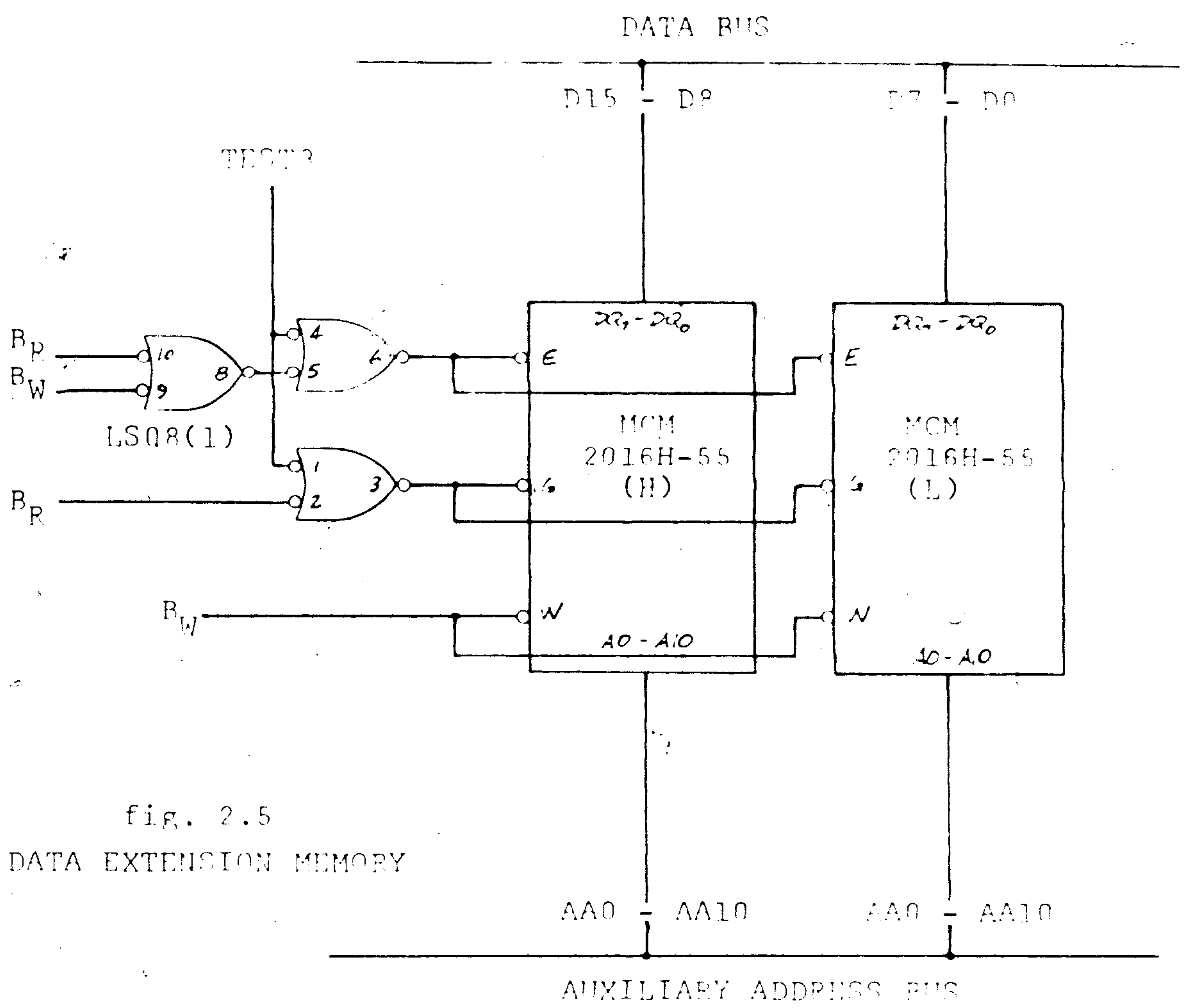


fig. 2.5
DATA EXTENSION MEMORY

2.2 Programming the Reed-Solomon decoder

In this section, the programming of the decoding procedure described in 1.2.2 for the six error correcting Reed-Solomon code of block length 63 over GF(127) is discussed. The two most important data structures in the program are explained in 2.2.1 and 2.2.3 and the throughput of the decoder is compared to that of other decoder designs [16,17] in 2.2.5.

The basic structure of the decoding procedure is as follows. The coefficients of the received word R(x) are stored in sequential Data Extension Memory addresses. The '32010 first computes the twelve syndromes $S_1, S_2, \ldots, S_{12}$ of ~~the~~ the received word. This involves the evaluation of R(x) at twelve separate points of GF(127). The efficient evaluation of polynomials is discussed in 2.2.1. After the syndromes are obtained, the Berlekamp-Massey Algorithm is used to obtain the error locator and error evaluator polynomials. The programming of this algorithm is discussed in 2.2.3. After the error locator and error evaluator polynomials are found, the Chien search is executed, which involves more polynomial evaluation. Then the error magnitudes are computed and subtracted from the appropriate coefficients of the received word. The failure modes described in 1.2.4 are easily incorporated into the program.

2.2.1 The evaluation of polynomials

The most frequent operation in this Reed-Solomon decoder is the evaluation of a polynomial at a point in GF(127). The evaluation of polynomials is the basis of the syndrome computation and the Chien search.

Consider a polynomial $f(x) = f_n x^n + f_{n-1} x^{n-1} + \ldots + f_0$. We wish to determine $f(\alpha)$ for some α. The Horner's rule representation

$$f(\alpha) = (\ldots((f_n\alpha+f_{n-1})\alpha+f_{n-2})\alpha+\ldots)\alpha+f_0$$

[9,p.467] suggests the following approach:

Polynomial evaluation routine:

```
(1) Initialize: $S \leftarrow f_n$, $I \leftarrow n$.
(2) Test I: If $I \leftarrow 0$, stop [with $S \leftarrow f(\alpha)$].
(3) Update S: $S \leftarrow \alpha S + f_{I-1}$.
(4) Update I: $I \leftarrow I-1$.
(5) Return to (2).
```

The multiply and add operation in step 3 is called the *prototypical inner loop* [17]. An efficient method of implementing this routine for polynomials over GF(127) using the TMS32010 is discussed below.

Two integers i, j are said to be *mod 127 representations* of each other if i-j is a multiple of 127. It is fairly easy to see that any integer i has a unique mod 127 representation r(i) in the range

$$R = \{-63, -62, \ldots, 0, \ldots, 62, 63\}.$$

It can be shown that for any nonzero i in R there is a unique nonzero i' in R such that r(ii') = 1. Hence, defining the product of i and j in R to be r(ij), any nonzero element of R has a unique multiplicative inverse in R. In fact, if we define the sum [difference] of i and j in R to be r(i+j) [r(i-j)], R is a field under this type of arithmetic. Henceforth, the symbol "GF(127)" will denote this particular field structure.

Suppose i is stored in DATj of the TMS32010 (recall that the data registers, being 16 bits wide, can accomodate any integer between -2^{15} and $2^{15}-1$). Also suppose that the binary number 0000 1111 1111 1111 (= $2^{12}-1$) is stored in DATk, 64 is stored in DATl, and 127 is stored in DATm. Then r(i) can be computed and stored back in DATj as follows:

r(i) routine:

```
        (1) LAC DATj, 5
        (2) SACH DATj, 4
        (3) AND DATk
        (4) ADD DATj, 5
        (5) SACH DATj, 4
        (6) AND DATk
        (7) ADD DATj, 5
```

```
 (8) SUB DATl, 5
 (9) BLZ (11)
(10) SUB DATm, 5
(11) ADD DATl, 5
(12) SACL DATj
(13) LAC DATj, 11
(14) SACH DATj, 0
```

All of the instructions appearing here are described in 2.1.1 except for the "BLZ (11)" instruction at step (9). This instruction causes a branch to step (11) if the accumulator entry resulting from step (8) is less than zero. Otherwise, execution continues at step (10). It is straightforward and somewhat tedious to prove that this routine actually produces r(i), so no proof is given. The main feature worth noting about the r(i) routine is its substantial length.

Only 7 bits are required to represent any element of GF(127), which is an apparent waste of the '32010's 16 bit data handling capability. However, the 16 bit capability does allow many different mod 127 representations of an element of GF(127). For example, 23 has the representations

$$-1247, -612, -104, 150, 658, 1293,$$

each of which can be formatted as a 16 bit two's complement number.

The use of certain types of mod 127 representations makes it unnecessary to include the time consuming r(i) routine in the prototypical inner loop of a polynomial evaluation. The r(i) routine need only be executed once at the end of the polynomial evaluation. With this in mind, the following simple claim is presented.

Claim: If the integer $i = i_1 2^{12} + i_2$, $0 \le i_2 \le 2^{12}-1$, satisfies $|i| \le 2^{20} + 2^{18}$, then $i_2 + 2^5 i_1$ is a mod 127 representation of i and $|i_2 + 2^5 i_1| < 2^{14}$.

Proof:

$$i = i_1 2^{12} + i_2 = 2^5 i_1 (2^7-1) + i_2 + 2^5 i_1,$$

so $i - (i_2 + 2^5 i_1)$ is a multiple of $2^7-1 = 127$. Furthermore,

$$\begin{aligned} 2^7 |i_2 + 2^5 i_1| &= |2^{12} i_1| + 2^7 i_2 \\ &\le |i| + i_2 + 2^7 i_2 \\ &< 2^{20} + 2^{18} + 2^{12} + 2^{19} \\ &< 2^{20} + 2^{19} + 2^{19} = 2^{21}. \end{aligned}$$

This completes the proof of the claim. ∎

This result suggests a reasonably efficient '32010 implementation of the prototypical inner loop:

Given that $\alpha \in GF(127)$ is stored in the T register and

$$\begin{aligned} &DATj \leftarrow S,\ |S| < 2^{14}, \\ &DATn \leftarrow A,\ -2^{15} \le A \le 2^{15}-1, \\ &DATk \leftarrow 0000\ 1111\ 1111\ 1111_2; \end{aligned}$$

a mod 127 representation of $\alpha S + A$ having magnitude less than 2^{14} can be computed and stored back in DATj as follows:

(1) MPY DATj. This instruction computes αS and loads the product into the P register. The product has magnitude less than $(2^6)(2^{14}) = 2^{20}$.
(2) PAC. This instruction transfers the P register contents to the accumulator.
(3) ADD DATn. Now $\alpha S + A$ is stored in the accumulator. This term has magnitude less than $2^{20} + 2^{15}$, so the previous claim can be used to reduce $\alpha S + A$ to a mod 127 representation having magnitude smaller than 2^{14}.
(4) SACH DATj, 4. Writing $\alpha S + A = i = i_1 2^{12} + i_2$, this operation stores i_1 in DATj as explained in 2.1.1.
(5) AND DATk. This instruction computes the logical AND of the 16 least significant bits of the accumulator and the contents of DATk, storing the result back in the 16 least significant accumulator positions. The upper 16 bits of the accumulator are zeroed by this instruction. With the given

value of DATk, the execution of this instruction leaves i_2 stored in the accumulator.

(6) ADD DATj, 5. At the end of this instruction, $i_2 + 2^5 i_1$ is stored in the accumulator. This is a mod 127 representation of $\alpha S + A$ having magnitude less than 2^{14}.

(7) SACL DATj. $i_2 + 2^5 i_1$ is stored back in DATj by this instruction.

When this 7 step prototypical inner loop is used in the polynomial evaluation routine, the successive values of S always have magnitude less than 2^{14}. When the routine is complete, a mod 127 representation of $f(\alpha)$ having magnitude less than 2^{14} is stored in DATj. This result can be reduced to its mod 127 representation in GF(127) by applying the r(i) routine to it.

If one insists on reducing each of the intermediate values of S to its mod 127 representation in GF(127), it is necessary to include the 14 step r(i) routine in the prototypical inner loop. The resulting polynomial evaluation routine would require roughly three times as much execution time even for polynomials of small degree.

Returning to the six error correcting Reed-Solomon code of block length 63 over GF(127), the evaluation of the syndromes $S_1, S_2, \ldots, S_{12}$ can be programmed in about 140 lines of '32010 code. The number of machine cycles the program requires to compute j syndromes is approximately

$$66 + 62(7j + 4) + 18j.$$

initialization   j polynomial evaluations   j mod 127 reductions

Hence, the evaluation of 12 syndromes requires about 5738 machine cycles. This result will be used in 2.2.5 to estimate the maximum throughput of the decoder.

2.2.2 Remarks

(1) 9 is a primitive 63rd root of unity in GF(127) so the codewords of the six error correcting code of length 63 over GF(127) can be chosen to have roots 9, $r(9^2)$, $r(9^3)$, ..., $r(9^{12})$.

(2) The syndrome evaluation program appears in [39]. Aside from the prototypical inner loop and the final reduction to GF(127) representations, the program is fairly trivial. The autoincrement feature of the Auxiliary Address Register is used during the reading of the coefficients of R(x) from Data Extension Memory.

(3) The set {-128, -127, ..., 0, ..., 127, 128} can be given a field structure via a mapping r'( ) analogous to the mapping r( ) used in 2.2.1. Let GF(257) denote this field structure. The mapping r'(i) can be implemented with a routine very similar to the r(i) routine of 2.2.1. Furthermore, a GF(257) version of the prototypical inner loop is

```
(1) MPY DATj (where DATj ← S, |S| < 2^14)
(2) PAC
(3) ADD DATn (where DATn ← A, -2^15 ≤ A < 2^15)
(4) SACH DATj, 4
(5) AND DATk (where DATk ← 0000 1111 1111 1111_2)
(6) SUB DATj, 4
(7) SACL DATj
```

which differs from the GF(127) prototypical inner loop only in the sixth instruction. The updated value of S produced by the GF(257) version always has magnitude smaller than 2^{14}.

2.2.3 Implementing the Berlekamp-Massey Algorithm

Introducing new variables p(k), Δ, and $\Delta\tau_k(z)$ and eliminating the computations involving ω_k and γ_k, Berlekamp's Iterative Algorithm for the decoding of t error correcting codes can be rewritten as follows:

(1) Set $\sigma_0 = \Delta\tau_0 = \Delta = 1$, $p(0) = 2t$, $D(0) = k = 0$.

(2) If $k = 2t$, stop. Otherwise define

$$\sigma_{k+1} = \sigma_k - \Delta_k \Delta^{-1} z\{\Delta\tau_k(z)\},$$

where Δ_k is the coefficient of z^{k+1} in the product

$$[1 + S(z)]\sigma_k(z).$$

(3) If $\Delta_k = 0$ or if $D(k) \geq (k+1)/2$, put $D(k+1) = D(k)$ and define

$$\Delta\tau_{k+1} = z\Delta\tau_k,$$
$$p(k+1) = p(k)-1.$$

But if $\Delta_k \neq 0$ and $D(k) < (k+1)/2$, put $D(k+1) = k+1-D(k)$ and define

$$\Delta\tau_{k+1} = \sigma_k,$$
$$p(k+1) = p(k),$$
$$\Delta = \Delta_k.$$

(4) Increment k and return to (2).

Except for the term p(k), this is the Berlekamp-Massey Algorithm mentioned in 1.2.4. The polynomial τ_k can be obtained from $\Delta\tau_k$ through the formula $\tau_k(z) = \Delta^{-1}\{\Delta\tau_k(z)\}$. The most important feature of this algorithm is that the bulk of step 3 can be carried out with simple updates of p, D, and Δ and with a single conditional *pointer register swap*. All of the Galois field arithmetic appears in step 2.

Consider the storage format depicted in fig. 2.6. Here we have

$$\sigma_k(z) = \sum_i \sigma_{k,i} z^i,$$

$$\Delta\tau_k(z) = \sum_i \Delta\tau_{k,i} z^i.$$

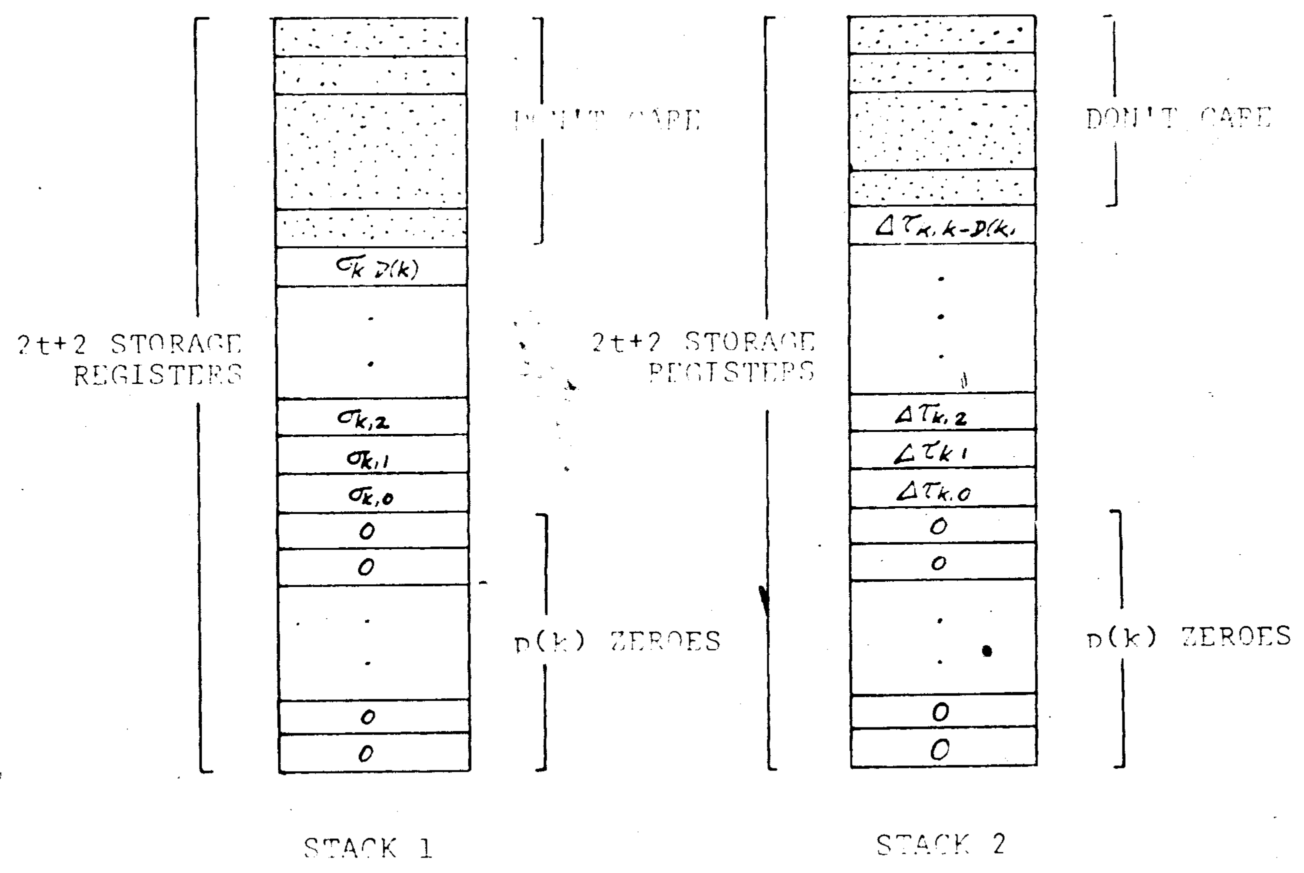


fig. 2.6
STACK STORAGE FORMAT

Each of stacks 1 and 2 is assumed to be composed of 2t+2 sequential memory locations embedded within a larger common memory. In addition to these stacks, there are two *pointer registers* R1 and R2, where Ri contains the memory address of the lowest register in stack i, i=1,2. Stack i is said to be *addressed* by Ri. It is not obvious that a stack of depth 2t+2 is large enough to accommodate p(k) zeroes and D(k)+1 coefficients of $\sigma_k(z)$ or that it is large enough to accommodate p(k) zeroes and k-D(k)+1 coefficients of $\Delta\tau_k(z)$. It is shown in Appendix 4 that $p(k)+D(k) \leq 2t+1$ and $p(k)+k-D(k) \leq 2t+1$ for all $0 \leq k \leq 2t$, implying that these stacks *are* sufficiently large.

After Δ_k and σ_{k+1} have been computed in step 2, step 3 can be executed as follows:

(3.1) If $\Delta_k \neq 0$ and $D(k) < (k+1)/2$, go to (3.3). Otherwise continue.

(3.2) $D(k+1) = D(k)$, $p(k+1) = p(k)-1$. Go to (3.4).

(3.3) $D(k+1) = k+1-D(k)$, $p(k+1) = p(k)$, $\Delta = \Delta_k$. Swap R1 and R2 entries (stack 1 becomes stack 2 and vice versa).

(3.4) Leave p(k+1) zeroes in the bottom of the stack addressed by R1. Store the first D(k+1)+1 coefficients of $\sigma_{k+1}(z)$ above these zeroes.

It can be verified that the execution of (3.1)-(3.4) will set up the memory conditions of fig. 2.6 with k replaced by k+1. When decoding with the TMS32010, the two stacks can be situated somewhere within the 144 data registers DAT0 - DAT143. The four steps (3.1) - (3.4) are quite easy to program.

2.2.4 Remarks

(1) A technique similar to the use of stacks and pointer registers is discussed in [16,pp.139-141]. However, the issue of stack size is not addressed there.

(2) The t=6 GF(127) version of the algorithm discussed in 2.2.3 can be programmed in approximately 160 '32010 instructions. This program appears in [39]. The following upper bound on its execution time holds.

Let N be the actual number of machine cycles required to execute the program. Then

$$N \le \underbrace{139}_{\text{initialization}} + \sum_{k=0}^{11} \{ \underbrace{4}_{\text{test/increment } k} + \underbrace{4k+34}_{\text{compute } \Delta_k} + \underbrace{27}_{\text{compute } \Delta_k/\Delta}$$

$$+ \underbrace{23}_{\text{compute } D(k+1) \text{ and test for decoding failure}} + \underbrace{9(k+1-D(k))+20}_{\text{compute } \sigma_{k+1}}$$

$$+ \underbrace{14}_{\text{register swap}} + \underbrace{4D(k+1)+13}_{\text{update stack 1}} \} + \underbrace{4}_{\text{exit}}$$

$$= 143 + \sum_{k=0}^{11} 13(k+1) + \sum_{k=0}^{11} [4D(k+1)-9D(k)] + 12(131)$$

$$\leq 143 + 13(6)(13) - 9D(0) + 4D(12) + 12(131)$$

$$\leq 2753 \text{ machine cycles.}$$

(3) The multiplicative inversion occurring at step (2) of the Berlekamp-Massey algorithm is accomplished by reading the inverse of Δ from a look-up table. Subsequent inversions occurring in the computation of the error magnitudes are carried out in the same way.

2.2.5 Decoder performance

After the Berlekamp-Massey Algorithm is executed, it remains to do the Chien search and the error magnitude computation. For the six error correcting code of block length 63 over GF(127), the Chien search involves up to 63 evaluations of a polynomial of degree ≤ 6 over GF(127). The ideas of 2.2.1 can be used to produce a simple (98 lines) TMS32010 Chien search program. The basic idea of the program is to look up the powers $9^0, 9^{-1}, 9^{-2}, \ldots, 9^{-62}$ in a table in Data Extension Memory, evaluate $\sigma_{12}(z)$ at these powers, and store the values of i for which $\sigma_{12}(9^{-i}) = 0$. The running time of the program is

$$\underbrace{16}_{\text{initialization}} + \underbrace{63(7D(12) + 30)}_{\text{63 evaluations of } \sigma_{12}(z)} + 14D(12)$$

$$\leq 4636 \text{ machine cycles.}$$

The error magnitude computation is based on equation (1.8) of Chapter 1. The error evaluator polynomial $\omega(z)$ must first be generated from $\sigma_{12}(z)$ via the key equation (1.7) as explained in 1.2.4. This can be programmed in 35 lines of TMS32010 code. Then the magnitudes are computed from (1.8) and subtracted from the corresponding coeffici[illegible] of the received word. This requires 150 lines of code[illegible]he total running time of these two programs is bounded above by 1641 machine cycles.

Recalling the cycle counts from 2.2.1 and 2.2.4, the entire decoding procedure for one word of the six error correcting code takes at most 5738 + 2753 + 4636 + 1641 = 14,768 machine cycles. Hence, the maximum throughput of the decoder is at least

$$\frac{(\simeq 7 \text{ bits/symbol})(63 \text{ symbols/codeword})}{14{,}768 \times 200 \times 10^{-9} \text{ seconds/codeword}}$$

$$= 149 \text{ kbit/sec.}$$

As mentioned in 2.2.2, polynomials over GF(257) can be evaluated as easily as polynomials over GF(127) using the TMS32010. In fact, all of the operations of Reed-Solomon decoding can be done just as easily in GF(257) as they are done in GF(127). Accordingly, a decoding routine for the six error correcting code of length 256 over GF(257) has also been programmed on the '32010. This decoder has a maximum throughput of at least 225 kbit/sec. In this case, almost 90% of the decoding time is spent on the syndrome computations and on the Chien search, emphasizing the importance of efficient polynomial evaluation.

The main motivation for the choice of the six error correcting code is that a decoder implementation for the six error correcting code of length 255 over GF(256) is discussed in [16,Chap.6]. This implementation, based on the Intel 8086 microprocessor [24,Chap.5], has a maximum throughput of only 18.5 kbit/sec. Although it is estimated that an implementation of the same decoder using the Motorola MC68000 [24,Chap.7] running at 24 Mhz would give a throughput of 110 kbit/sec [16,p.90], this is still less than half the rate at which the TMS32010 can decode the length 256 code over GF(257). Since the two codes in question are virtually identical (recall 1.2.1) and since present versions of the '68000 are limited to 16.7 Mhz, there is a clear advantage to be had by using GF(257) and the TMS32010.

The arithmetic of GF(256) is quite different from that of GF(257). The elements of GF(256) can be represented as 8 bit symbols in such a way that the sum of two symbols is obtained as their exclusive OR. Computing the product of two symbols is more difficult. In [16,Chap.6] the elements of GF(256) are represented as powers of a fixed primitive element and a product is obtained by adding the exponents of the two operands. This approach involves some difficulties. For example, the zero element of GF(256) isn't a power of a primitive element and requires special treatment. It is doubtful that a TMS32010 implementation of GF(256) arithmetic would be any more efficient than a 24 Mhz '68000 implementation.

The design of a decoder for the eight error correcting code of length 255 over GF(256) is considered from several standpoints in [16,Chap.7]. Three designs based on the AM2900 family of bit slice components [24,Chap.8] are discussed. A "basic" design consisting of approximately 30 SSI/MSI/memory chips and 3 bit slice CPU components is capable of a 200 kbit/sec throughput. By comparison, the TMS32010 design can decode the eight error correcting code of length 256 over GF(257) at a rate of 175 kbit/sec (In this case, 89% of the decoding time is devoted to the syndrome computation and the Chien search).

The throughput of the basic bit slice design can be doubled by doubling the amount of hardware and finally raised to 650 kbit/sec by adding still more hardware. The latter design involves about 100 SSI/MSI/memory chips and 6 bit slice components [16,p.167]. A similar speed improvement can be obtained by adding hardware to the TMS32010 design. However, the replacement of the '32010 with more recent signal processor devices offers significant speed improvements without increasing the parts count of the decoder.

Since the introduction of the TMS32010 in 1983, some significantly more powerful signal processor devices have been developed. The TMS320C25, introduced in 1986 [25], has all the functional capabilities of the TMS32010 and operates at twice the '32010's execution speed (ie: 100 ns machine cycle). Therefore, the processor provides twice the throughput of the '32010. The Motorola DSP56000, also introduced in 1986, works with 24 bit data words and has a 97.5 ns machine cycle. Based on preliminary information [26], it is estimated that this device could decode the eight error correcting Reed-Solomon code of length 256 over GF(257) at a rate near 500 kbit/sec.

The most promising development in signal processor technology is the announcement of the Texas Instruments TMS320C30 [40]. It boasts a 60 ns machine cycle, 32 bit words, and up to 33 million floating point operations per second. It is roughly estimated that this device could decode Reed-Solomon codes four to six times faster than the TMS32010. The actual throughput realized with this device will depend heavily on how well its internal parallelism is used. The '320C30 also has several on-chip features that reduce external logic requirements (2k×32 RAM, serial ports, timers, etc.).

In summary, these later generation processors provide throughput rates near those of the fastest [illegible]slice designs of [16]. However, a signal processor syst[illegible]uires only 10 to 30 ICs,' a substantial improvement over the 100+ IC count of the fastest bit slice design. The comparison between the eight error correcting designs considered in [16] and some eight error correcting designs based on signal processors is summarized in Table 1.

The 24 bit '56000 chip and the 32 bit '320C30 will likely admit efficient implementations of multiplication, addition, and subtraction over GF(2^{16}+1). However, multiplicative inversion in this large field is not straightforward. The storage of inverses of each of the 2^{16} nonzero elements in a look-up table would require 1 Mb of memory. An alternative inversion method based on the Euclidean Algorithm for integers [8,pp.21-25] would eliminate the need for the look-up table, but would require a significant amount of computation time.

'The decoder system designed as part of this thesis contains 41 ICs [39]. 17 of these ICs form a user interface, which allows the programming and testing of the system. Once a program has been debugged, this interface is no longer needed. The program could be burned into ROM, which permanently replaces Program Memory, and the user interface circuitry removed. This leaves a 24 chip system.

Technology	Source	$q^{(*)}$	Speed	IC Count	Notes
8086	16,p.167	256	15 kb/s	uP+15	
68000	16	256	90	uP+20	(1),(2)
32010	This thesis	257	175	uP+20	
2900	16	256	200	3uP+30	
2900	16	256	400	6uP+60	(2)
320C25	This thesis	257	400	uP+15	(2)
56000	This thesis	257	500	uP+15	(2)
2900	16	256	650	6uP+100	(2)
320C30	This thesis	257	700k - 1M	uP+10	(2)
GF1TM	16	256	3 Mb/s	200 ECL	(3)
GF1TM	16	256	5 Mb/s	260 ECL	(3)

(*) $n = q-1$
(1) Assumes 24 MHz clock (present technology - 17 MHz).
(2) Estimated - not constructed.
(3) GF1TM is a trademark of Cyclotomics.

TABLE 1

PERFORMANCE OF EIGHT ERROR CORRECTING REED-SOLOMON DECODERS

Chapter 3

Synchronization

In any digital communication system the transmitter and the receiver must be synchronized. The receiver must know when and how often the channel should be sampled to recover the data stream produced by the transmitter. Secondly, if the transmitted data consists of a sequence of codewords of some error correcting code, the receiver must know when each codeword ends and a new one begins. The first level of synchronization is called *character synchronization*. A few well established methods of analog processing can be applied to the data sequence to acheive character synchronization [29], [30,p.193]. The second level of synchronization is called *frame synchronization*. Some general methods of obtaining frame synchronization are discussed in [31]. In this chapter, a method of obtaining both character synchronization and frame synchronization without using any analog processing is presented and analyzed.

3.1 An approach to the synchronization problem

Suppose that the data symbols used by the transmitter and receiver belong to some finite alphabet F (F may be any finite set). Let S_0, S_1, S_2, ... be a sequence of symbols in F that is known to both the transmitter and the receiver. If the transmitter produces the data symbols D_0, D_1, D_2, ... and inserts consecutive S_i terms before every n^{th} data symbol, one obtains the sequence

$$S_0, D_0, \ldots, D_{n-1}, S_1, D_n, \ldots, D_{2n-1}, S_2, \ldots$$

Suppose that the symbols of this sequence are transmitted over the channel at a rate of 1/T symbols per second. If the receiver samples the channel every T' seconds, where $T' \le T$, it will obtain a sample of every term in the sequence. It must determine which of these samples correspond to the data symbols D_i.

In addition to the possible timing discrepancy between the transmitter and receiver, channel noise may cause an occasional sampling error at the receiver. However, if the noise isn't too severe, the receiver will be able to recognize many of the S_i terms within the sequence of samples. It is easy to see that if

$$n/(n+1) < T'/T \le 1, \tag{3.1}$$

then samples of consecutive S_i terms are separated by either n or n+1 samples. Henceforth, (3.1) will be assumed to hold. In 3.1.1, it is shown how the *Viterbi Algorithm* can make use of this property to estimate the position of each S_i term in the sequence of samples. The positions of the data samples can then be estimated. The performance of the Viterbi Algorithm is analyzed in 3.1.2 and 3.1.4.

When $T'/T < 1$, the receiver observes an occasional "extra" data symbol between consecutive S_i terms. The extra character arises because the receiver is sampling the data stream too fast: two samples of a single transmitted character sometimes occur. Thus, the extra character is statistically dependent on an adjacent character. This dependence makes any analysis of the system very difficult. For this reason, attention is restricted to the simpler (but very similar) mathematical model described in 3.1.1.

The motivation for this technique is that it does not require the transmitter and the receiver to have a common timing reference, thereby eliminating the need for a phase locked loop and related analog circuitry [29], [30,p.193]. The only requirement is that condition (3.1) be satisfied. T' and T may even vary with time as long as they never violate the limits specified in (3.1).

3.1.1 The Viterbi Algorithm

For the reasons outlined above, the following model is considered.

The transmitter sends a sequence ~~of~~ the form

$$S_0,\ D_0,\ \ldots,\ D_{n-1},\ S_1,\ D_n,\ \ldots,\ D_{2n-1},\ S_2,\ \ldots \tag{3.2}$$

where S_0, S_1, S_2, ... are *synch symbols* known to both the transmitter and the receiver. The terms of sequence (3.2) are independent F-valued random variables (RVs). Each D_i has the distribution

$$\Pr\{D_i = x\} = D(x),\ x \in F,$$

while each S_i has the distribution

$$\Pr\{S_i=x\} = S(x).$$

The channel inserts independent F-valued RVs into the sequence (3.2), producing a new sequence

$$X_0,\ X_1,\ X_2,\ \ldots,\ X_j,\ \ldots \tag{3.3}$$

Each X_i is either a synch symbol, a data symbol, or one of the RVs inserted by the channel. The inserted RVs are called *insertion errors* and are assumed to have the same distribution $D(x)$ as the data symbols. It is assumed that no more than one insertion error appears between any two consecutive synch symbols.

Suppose that the receiver obtains exactly one sample Y_i of each RV X_i in sequence (3.3) [instead of occasionally obtaining two samples of a single RV in sequence (3.2)]. The samples are themselves F-valued RVs and are assumed to be independent of each other. Y_i is also assumed to be independent of X_j if $i \neq j$. Furthermore, for each x,y in F, the *conditional probability*

$$p(y|x) \equiv \Pr\{Y_i=y,\ X_i=x\}/\Pr\{X_i=x\}$$

is assumed to be a known channel parameter that does not depend on the subscript i.

By loading the Y_i RVs int[illegible]uccessive columns of a depth-n+1 memory, the storage format of fig. 3.1 is obtained. The shaded squares represent the locations of samples of the synch sequence (*synch samples*).

For simplicity, let "$[k,j]$" denote the memory location in the jth row of the kth column of fig. 3.1. Define a *path of length l* to be any sequence

$$\{[k_0,j_0],\ [k_1,j_1],\ \ldots,\ [k_l,j_l]\}$$

of memory locations satisfying $k_0=j_0=0$ and either

$$(n+1)k_i+j_i-\{(n+1)k_{i-1}+j_{i-1}\} = n+1, \qquad (3.4a)$$

or

$$(n+1)k_i+j_i-\{(n+1)k_{i-1}+j_{i-1}\} = n+2 \qquad (3.4b)$$

for every $i = 1,2,\ldots,l$. The memory locations

$$\{[K_0,J_0],\ [K_1,J_1],\ \ldots,\ [K_l,J_l]\}$$

occupied by the first $l+1$ synch samples form a path of length l.

Let $p(Y_j|S_i)$ be the RV taking the value $p(y|x)$ when $Y_j=y$ and $S_i=x$. If the channel noise isn't too bad, the RV

$$\prod_{i=0}^{l} p(Y_{(n+1)K_i+J_i}|S_i)$$

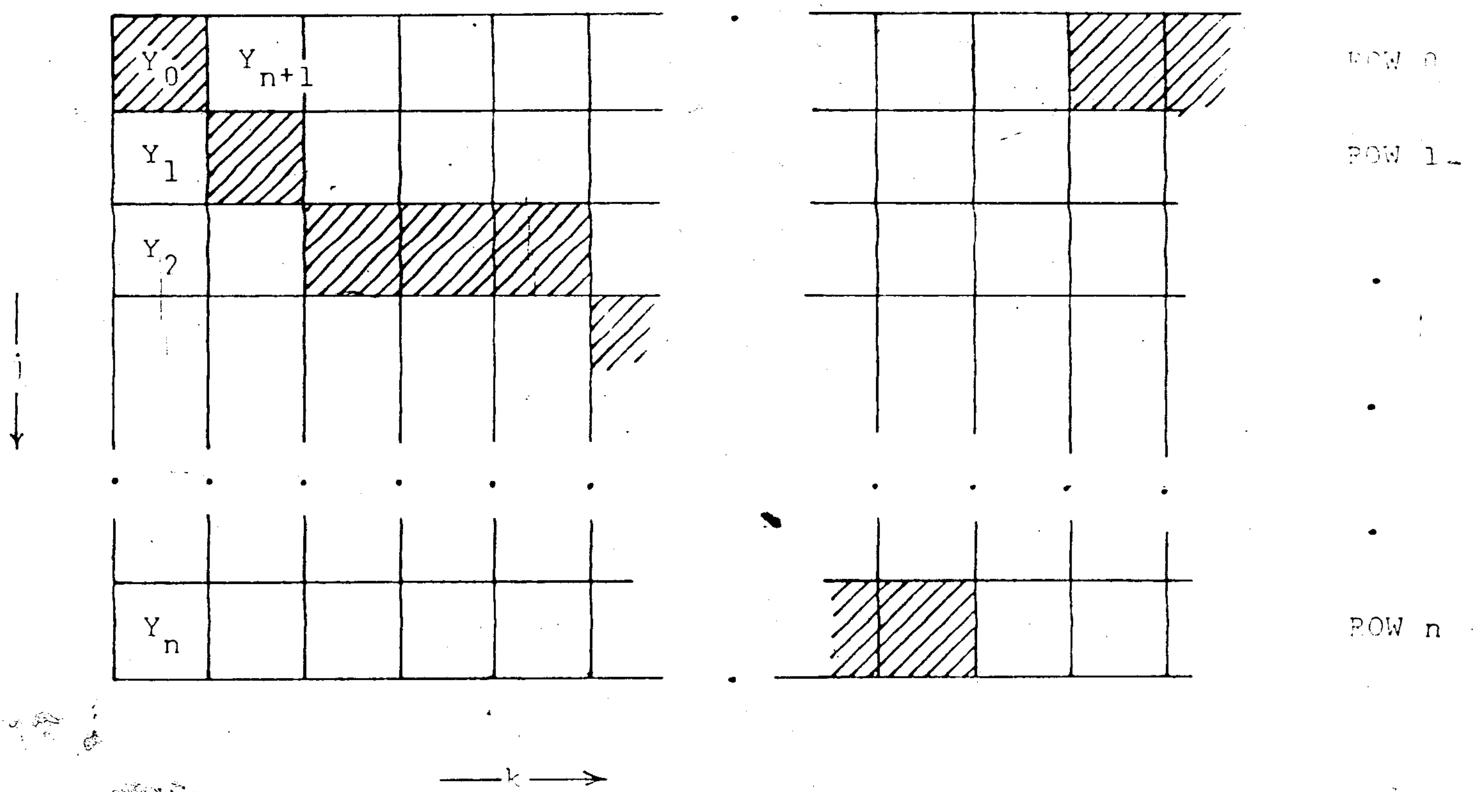


fig. 3.1
SAMPLE STØRAGE FORMAT

should have a high probability of exceeding the RV

$$\prod_{i=0}^{l} p(Y_{(n+1)k_i+j_i}|S_i)$$

associated with any other path

$$\{[k_0,j_0], \ldots, [k_l,j_l]\}$$

of length l. Hence, by assigning the *metric*

$$\ln \prod_{i=0}^{l} p(Y_{(n+1)k_i+j_i}|S_i) = \sum_{i=0}^{l} \ln p(Y_{(n+1)k_i+j_i}|S_i)$$

to each path of length l, the path formed by the synch samples will likely have the highest metric. This heuristic argument also applies to the more general metric

$$\sum_{i=0}^{l} [\ln p(Y_{(n+1)k_i+j_i}|S_i) + R],$$

where R is any fixed real number. From this standpoint, the receiver will obtain a good estimate of the positions of the first l+1 synch samples by selecting the path of length l having the largest metric. However, since there are 2^l paths of length l, a brute force search is impractical.

The decoding of *convolutional codes* can be described in terms of paths and metrics similar to these [32,pp.227-381]. The *Viterbi Algorithm* is a computationally efficient method of finding the paths having the largest metric. This algorithm can be modified to suit our purposes as follows:

Notation:

If $P = \{[k_0,j_0], \ldots, [k_{l-1},j_{l-1}]\}$ is a path of length l-1 and k_l, j_l satisfy conditions (3.4a) or (3.4b) for i=l, then the symbol

$$P*[k_l,j_l]$$

denotes the path

$$\{[k_0,j_0], \ldots, [k_{l-1},j_{l-1}], [k_l,j_l]\}$$

of length l.

Viterbi Algorithm:

(1) Initialize:

$$(P_0,m_0,l_0) \leftarrow ([0,0], \ln p(Y_0|S_0) + R, 0);$$

$$(P_{-1},m_{-1},l_{-1}) \leftarrow (\emptyset, -\infty, 0);$$

$$(P_j,m_j,l_j) \leftarrow (\emptyset, -\infty, 0), \ 1 \le j \le n;$$

$$k \leftarrow 0.$$

(2) Temporary storage:

$$(\pi,\mu,\lambda) \leftarrow (P_n, m_n, l_n).$$

(3) Update existing paths:

For each $0 \le j \le \min\{k+1,n\}$, define

$$\mu_{j,0} = m_j + \ln p(Y_{(n+1)(k+1)+j} | S_{l_j+1}) + R,$$

$$\mu_{j,1} = m_{j-1} + \ln p(Y_{(n+1)(k+1)+j} | S_{l_{j-1}+1}) + R.$$

If $\mu_{j,0} < \mu_{j,1}$ or if $j = k+1$,

$$(P_j, m_j, l_j) \leftarrow (P_{j-1}*[k+1,j],\ \mu_{j,1},\ l_{j-1}+1).$$

Otherwise,

$$(P_j, m_j, l_j) \leftarrow (P_j*[k+1,j],\ \mu_{j,0},\ l_j+1).$$

(4) Update (P_{-1}, m_{-1}, l_{-1}):

$$(P_{-1}, m_{-1}, l_{-1}) \leftarrow (\pi,\mu,\lambda).$$

(5) $k \leftarrow k+1$. Return to (2).

The behavior of this algorithm is summarized as a claim:

Claim: For each $0 \le j \le \min\{k+1,n\}$, the object P_j existing after the k+1th iteration of steps (2)-(5) is a path of length l_j with metric m_j and having endpoint $[k+1,j]$. It is called the *survivor* ending at $[k+1,j]$. No path of length l_j having endpoint $[k+1,j]$ has metric exceeding m_j. ∎

The proof of this claim is a straightforward induction on k. Step (1) ensures that the stated conditions hold for $k=0$.

Each iteration of steps (2)-(5) of the Viterbi Algorithm advances the path search by one column of memory locations. The complexity of the kth iteration is proportional to $\min\{k+1,n\}+1 \le n+1$. If the entire transmitted sequence contains N+1 synch symbols, the algorithm is stopped as follows: When any l_j reaches N, save P_j and m_j. Then replace m_j with $-\infty$. As soon as all l_j's equal or exceed N, stop. Search the set of saved path-metric pairs for the one with the largest metric. Write the associated path as

$$\{[K_0,J_0]^*,\ [K_1,J_1]^*,\ \ldots,\ [K_N,J_N]^*\}.$$

The receiver estimates the position of the kth synch sample to be $[K_k,J_k]^*$.

Once this estimate of the synch sample positions is obtained, the receiver may assume that the first n samples immediately following $[K_k,J_k]^*$ correspond to the data symbols

$$D_{kn},\ D_{kn+1},\ \ldots,\ D_{kn+(n-1)}. \qquad (3.5)$$

If $[K_k, J_k]^*$ is the actual location of the kth synch sample, the only way for this assumption to be wrong is to have an insertion error between S_k and $D_{kn+(n-1)}$. Hence, this decision strategy will work well if insertion errors are rare (ie: if $T' \simeq T$). However, if the kth synch position estimate is incorrect, the entire data segment (3.5) will likely be misaligned. An upper bound on the probability of such a *synchronization error* is presented in 3.1.2.

3.1.2 Performance analysis of the Viterbi Algorithm

Let $\{[K_0,J_0], [K_1,J_1], \ldots, [K_N,J_N]\}$ be the memory locations containing the synch samples of the entire transmitted sequence. This path is called the *correct path* and $[K_i,J_i]$ is called the ith *node* of the correct path. A path $\{[k_0,j_0], [k_1,j_1], \ldots\}$ satisfying

$$[k_m, j_m] = [K_m, J_m],$$

$$[k_{m+1}, j_{m+1}] \neq [K_{m+1}, J_{m+1}],$$

is said to *diverge* from the correct path at node m. It *remerges* with the correct path at node m+p if

$$[k_{m+i}, j_{m+i}] \neq [K_{m+i}, J_{m+i}], \ 0 < i < p,$$

$$[k_{m+p}, j_{m+p}] = [K_{m+p}, J_{m+p}].$$

Suppose that each of the paths produced by the Viterbi Algorithm never slips or advances by n+1 samples relative to the correct path (see fig. 3.2). It will be shown in 3.1.4 that such an event, called a *frame loss*, has a low probability of occurrence in cases of interest. Under this hypothesis, we wish to estimate the probability of a synchronization error at the kth synch position, $0 < k \le N$.

The only way for a synch error to occur at the kth position is for the path $\{[K_0,J_0]^*, [K_1,J_1]^*, \ldots, [K_N,J_N]^*\}$ produced by the Viterbi Algorithm to diverge from the correct path at some node m, where $m < k$. This path can then do one of two things:

(1) it can remerge with the correct path at some node $m+j$, where $k < m+j \le N$,

or

(2) it may never remerge with the correct path.

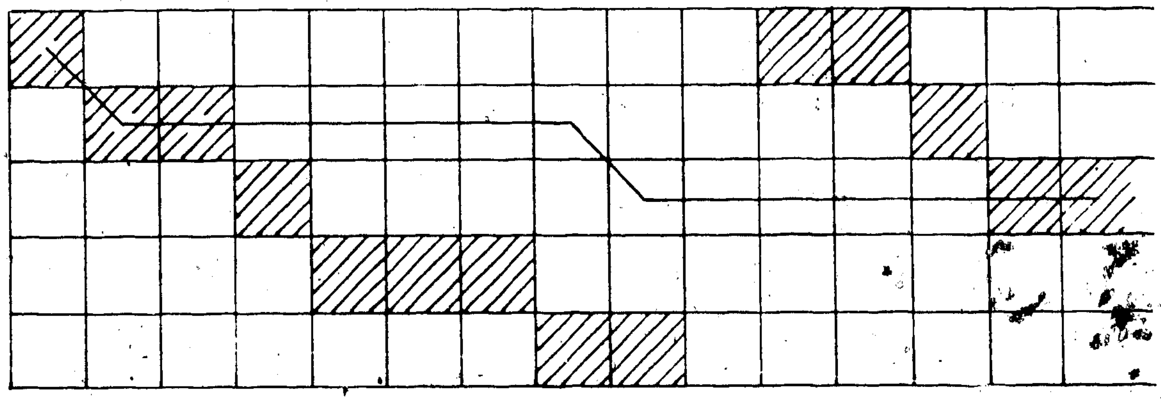
Define

$$Z \equiv \sum_{s\in F} S(s) \sum_{y_1\in F} \sum_{y_2\in F} p(y_1)[p(y_1|s)p(y_2|s)]^{1/2}.$$

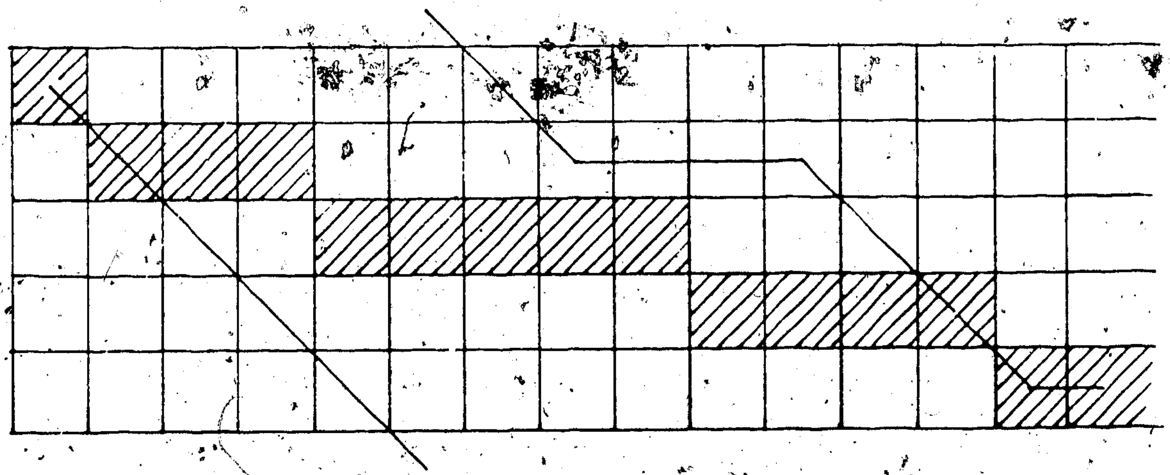
In Appendix 5 it is shown that the probability that condition (1) holds is upper bounded by

$$\sum_{m=0}^{k-1} \sum_{j=k+1-m}^{N-m} (2Z)^{j-1}$$

$$= 2Z(1-(2Z)^k)(1-(2Z)^{N-k})/(1-2Z)^2,$$



(a) Path slipping by n+1 samples relative to correct path (n=4).



(b) Path advancing by n+1 samples relative to correct path (n=4).

fig. 3.2
EXAMPLES OF FRAME LOSSES

and that the probability that (2) holds is upper bounded by

$$\sum_{m=0}^{k-1} (2Z)^{N-m}$$

$$= (2Z)^{N-k+1}(1-(2Z)^{k})/(1-2Z).$$

Thus, the total probability of a synch error at any position k is upper bounded by the sum of these two expressions. This sum is given by

$$2Z(1-(2Z)^{N+1-k})(1-(2Z)^{k})/(1-2Z)^{2}$$

$$< 2Z/(1-2Z)^{2}, \quad 2Z < 1.$$

If Z is very small, this provides a useful upper bound on the probability of a synchronization error.

3.1.3 Examples

Z can be expressed as

$$Z = \sum_{s \in F} S(s)f(s),$$

where

$$f(s) = \sum_{y_1 \in F} \sum_{y_2 \in F} p(y_1)[p(y_1|s)p(y_2|s)]^{1/2}.$$

It is clear that Z is minimized by choosing the distribution S(x) of the synch characters as follows:

$$S(x) = 1 \text{ if } f(x) \leq f(x') \text{ for all } x' \neq x,$$

= 0 otherwise.

This means that Z is minimized by finding the $x_0 \in F$ at which f(x) takes its smallest value and setting $S_i = x_0$, i = 0,1,2,... Practically speaking, this is a welcome result. The transmitter and receiver only have to remember x_0 instead of a complicated sequence of RVs.

Example 1: Suppose the alphabet *F* has size Q, D(x) is the uniform distribution on *F*, and the channel is noiseless (ie: the Q×Q matrix $[p(y|x)]$ is the identity matrix). Then $f(s) = D(s) = Q^{-1}$ for every $s \in F$, and $Z = Q^{-1}$ for every choice of the distribution S(x). Hence the synch error upper bound in this case is

$$2Q^{-1}/(1-2Q^{-1})^2, \quad Q > 2.$$

Q must be very large for this bound to be of any use. Furthermore, the bound will likely be even worse for noisy channels.

Example 2: Suppose the data symbols and insertion errors never take some value $s_0 \in F$ [ie: $D(s_0)=0$]. Define

$$S(x) = 1 \text{ if } x = s_0,$$
$$= 0 \text{ otherwise.}$$

[ie: $S_i = s_0$, i = 0,1,2,...]. With this choice of S(x), one has

$$Z = \sum_{y_1 \in F} \sum_{y_2 \in F} p(y_1)[p(y_1|s_0)p(y_2|s_0)]^{1/2}.$$

It is easy (but messy) to verify that

$$Z \le [\max_{x \ne s_0} p(s_0|x) + \sum_{y \ne s_0} p(y|s_0)^{1/2}][1 + \sum_{y \ne s_0} p(y|s_0)^{1/2}]$$

in this case. Hence, as the channel approaches the noiseless condition, Z approaches zero, implying that the synch error probability also approaches zero. Thus, by choosing the synch symbols S_i to be some fixed symbol lying outside of the set of data or insertion symbols, the synch error probability can be made as small as desired by using a sufficiently good channel. This is intuitively reasonable.

Example 3: If $F = \{0,1\}$, $D(0) = D(1) = 1/2$, and $p(0|1) = p(1|0) = P_e$, then

$$Z = 1/2 + [P_e(1-P_e)]^{1/2}$$

for all choices of the distribution S(x). Hence, $2Z \ge 1$, implying that the $2Z/(1-2Z)^2$ bound is invalid in this case. In spite of the fact that the results of this section say nothing useful about this example, the synchronization scheme will be implemented for a binary channel. This system and its observed performance are discussed in section 3.2.

3.1.4 On the probability of frame loss

A frame loss, introduced in 3.1.2, occurs when the Viterbi Algorithm produces a survivor P of length p-1 or p+1 that ends at memory location $[K_p,J_p]$ of the correct path. The former case is called a *frame advance* and corresponds to an advance of n+1 samples relative to the correct path (fig. 3.2b). The latter case is called a *frame slip* and corresponds to a slip of n+1 samples (fig. 3.2a). We will estimate the probability that the first frame loss occurs at memory location $[K_p,J_p]$. The case of a fram[illegible] considered first.

Write the survivor P [illegible]

$$\{[K_0,J_0]^s,\ [K_1,J_1]^s, \ldots,\ [K_{p+1},J_{p+1}]^s\}.$$

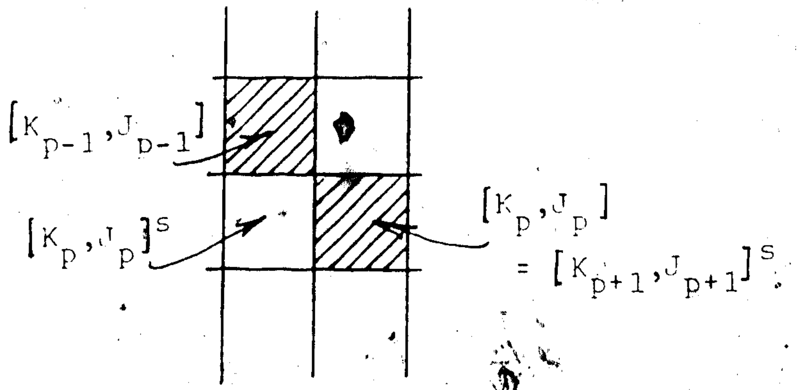
By hypothesis, $[K_{p+1},J_{p+1}]^s = [K_p,J_p]$. The possible orientations of the memory locations $[K_p,J_p]^s$, $[K_{p-1},J_{p-1}]$, and $[K_p,J_p]$ are shown in fig. 3.3.

In Appendix 6, it is shown that the probability that the Viterbi Algorithm produces a survivor of the form P is upper bounded by

$$e^{½R}(2Z)^{n+1}/(1-2Z)$$

whenever $2Z < 1$. Hence,

$P_s(p) \equiv$ Pr{The first frame loss is a frame slip at $[K_p,J_p]$}



OR. . .

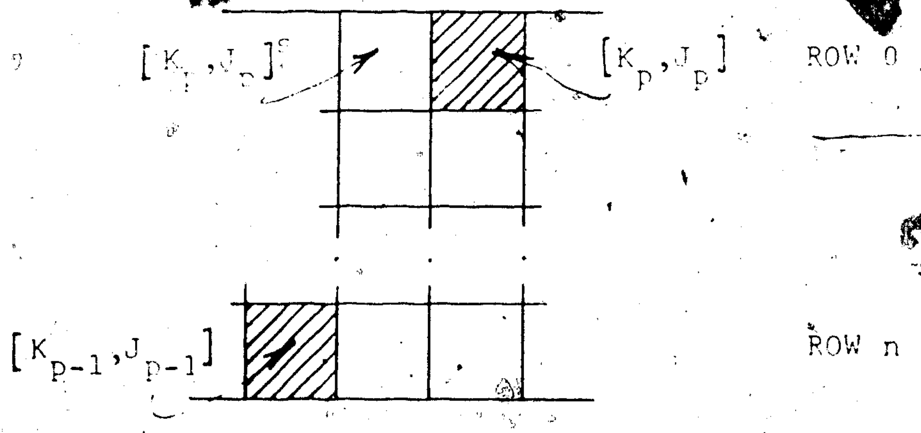


fig. 3.3
DETAILS OF FRAME SLIP

$$\leq e^{\frac{1}{2}R}(2Z)^{n+1}/(1-2Z)$$

whenever $2Z < 1$.

A similar argument shows that the probability

$$P_a(p) \equiv \Pr\{\text{The first frame loss is a frame advance at } [K_p,J_p]\}$$

satisfies

$$P_a(p) < e^{-\frac{1}{2}R}(2Z)^{n+1}/(1-2Z)$$

whenever $2Z < 1$. Hence the total probability $P_s(p) + P_a(p)$ that the first frame loss occurs at $[K_p,J_p]$ satisfies

$$P_s(p) + P_a(p) < (e^{\frac{1}{2}R}+e^{-\frac{1}{2}R})(2Z)^{n+1}/(1-2Z)$$

whenever $2Z < 1$.

Obviously if Z is reasonably small and n is moderately large, this probability is very small. The upper bound is minimized by the choice $R = 0$, where it takes the value

$$2(2Z)^{n+1}/(1-2Z).$$

3.1.5 Remarks

(1) The performance analysis of a decoder for certain convolutional codes can be carried out in a manner similar to the analysis of Appendices 5 and 6 [32,p.242-258]. One

obtains upper bounds on the probability of a decoding error that are similar to the upper bounds given in 3.1.2 and 3.1.4.

(2) The version of the Viterbi Algorithm proposed in 3.1.1 is similar to an algorithm of Ungerboeck [33]. The latter algorithm is used to reduce phase jitter in synchronous AM/PM systems.

(3) It is implicit in the previous analysis that the receiver knows exactly when the transmitted sequence starts. The frame synchronization methods described in [31] can be used to ensure this. Although these methods are based on the assumption that some form of character synchronization has been achieved, they appear to be adaptable to "slightly unsynchronized" channels [ie: systems satisfying condition (3.1)].

3.2 An implementation of the synchronization scheme

In this section, the implementation of the preceding synchronization scheme for the binary channel is discussed.

3.2.1 An overview

Suppose that the information to be transmitted to the receiver is formatted as a sequence of 7 bit symbols. By appending a known synch bit to each of these symbols, one obtains the transmission format of section 3.1 with $F = \{0,1\}$ and $n=7$. If the resulting data stream is sampled at more than twice the data rate, then at least two samples of each transmitted bit is obtained. This contrasts with the previous sampling method, which only guarantees that one sample of each transmitted symbol is obtained. The purpose of the higher sampling rate is to provide samples that are taken near the centre of each consecutive bit. If the samples are stored in a memory as shown in fig. 3.1 with n=15 and the sampling rate is sufficiently close to twice the transmitted bit rate, consecutive synch samples are separated by either 15 or 16 intermediate samples. Hence, the positions of the synch samples can be estimated by applying the Viterbi Algorithm with n=15. In the absence of knowledge about the channel noise, it is intuitively reasonable to assign the metric

$$-\sum_{i=0}^{n} Y_{(n+1)k_i+j_i} \oplus S_i$$

to the path $\{[k_0,j_0], \ldots, [k_n,j_n]\}$, where "$\oplus$" denotes exclusive OR.

The memory can be conveniently realized as a pair of 2k×8 static RAMs. This of course constrains the length of the transmitted message to less than 2048 8 bit bytes. A message length of 504 bytes will be used.

Each survivor $\{[k_0,j_0], \ldots, [k_n,j_n]\}$, produced by the Viterbi Algorithm can be represented as a binary sequence $\{\delta_1,\delta_2,\ldots,\delta_n\}$, where $\delta_i=0$ if equation (3.4a) holds and $\delta_i=1$ if (3.4b) holds. It is obviously more efficient to store sequences of this type than to store the parameters $[k_i,j_i]$

of each node of every survivor. When the path with the highest metric is selected at the end of the Viterbi Algorithm, it can be reconstructed from its (known) endpoint and the associated δ sequence. The receiver can then assume that the 2nd, 4th, ..., 14th samples immediately following the ith node of the path correspond to the bits of the ith symbol.

As mentioned in 3.1.5, the receiver must know exactly when the transmitted sequence begins. The easiest way to ensure this condition is to use a "handshake" format in which the receiver requests the transmission of the message via a feedback line. A format of this type is described below.

3.2.2 Description and performance

The D5 microcomputer kit is a versatile but relatively slow system based on the Motorola MC6802 microprocessor [34]. For the present purposes, it is used to accept and store 8 bit words generated by the TMS32010, after which it retransmits them serially to the '32010. This application requires the minor hardware modifications shown in fig. 3.4. The TMS32010 buffer memory and D5 interface hardware are shown in fig. 3.5.

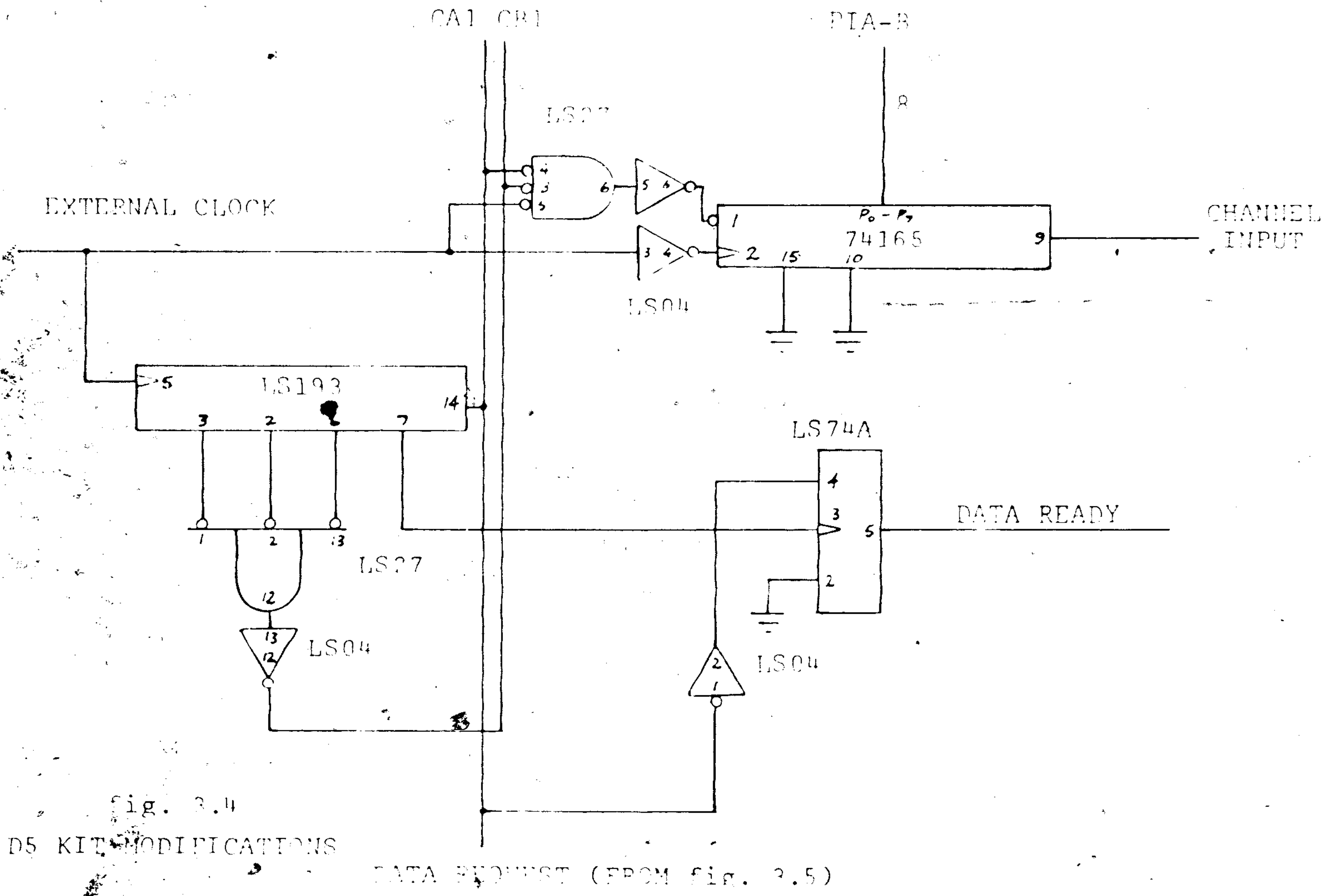


fig. 3.4
D5 KIT MODIFICATIONS

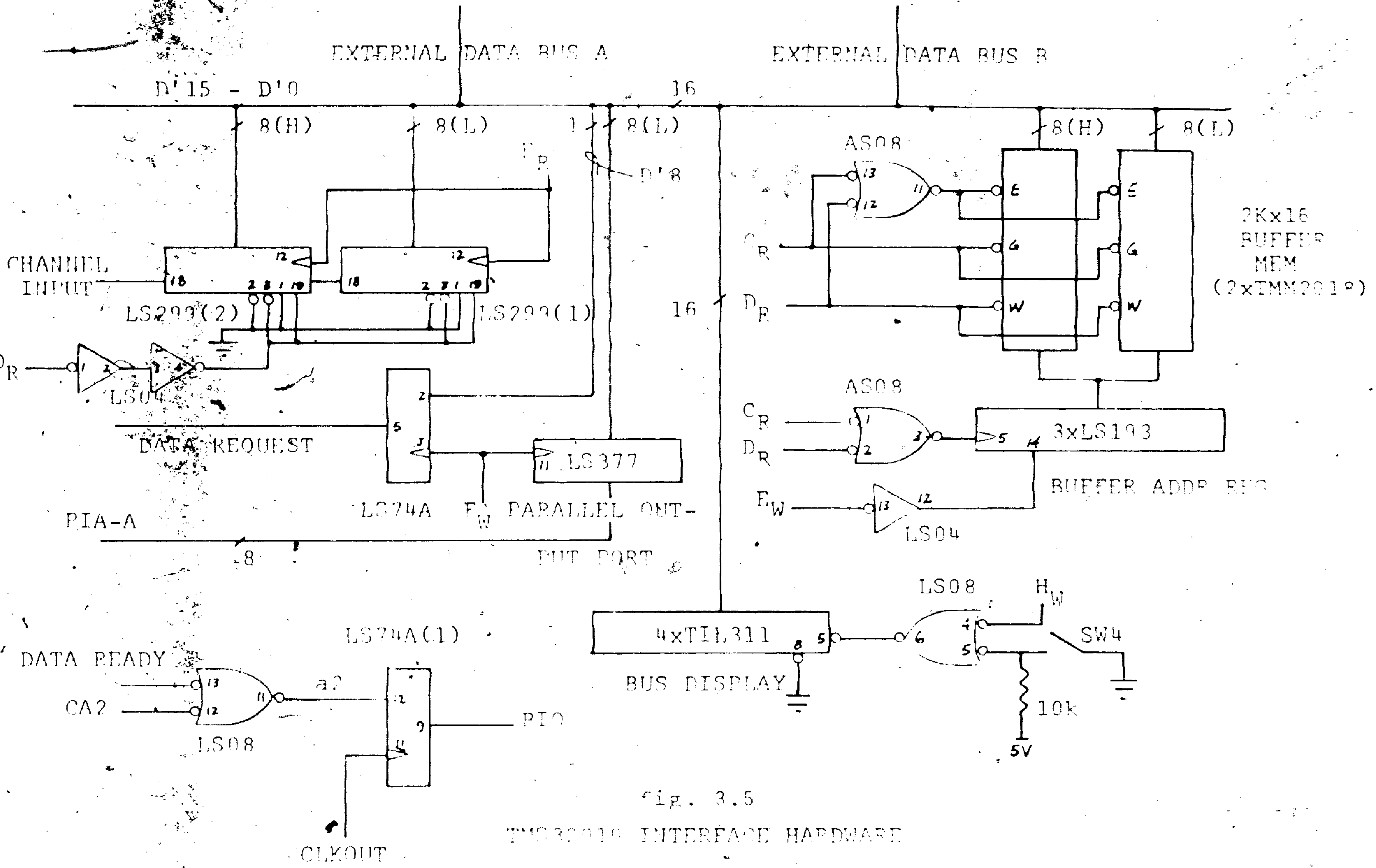


fig. 3.5
TMS32010 INTERFACE HARDWARE

Operation:

(1) The TMS32010 generates 8 codewords of the six error correcting Reed-Solomon code of length 63 over GF(127). Symbols in GF(127) are represented as integers between -63 and 63 and require 7 bits. Each codeword is contructed as follows. The '32010 generates a sequence of 441 bits of a length $2^{16}-1$ *PN sequence* [35]. This sequence is then segmented into 63 7 bit integers, each of which is reduced to its mod 127 equivalent in GF(127). Twelve of these symbols are then *erased* [8,sec.10.4] and are replaced with parity symbols. Full details on this erasure method appear in [8,sec.10.4] and [13,sec.9.2].

Once the codewords are generated, the '32010 adds a synch bit to each of the 7 bit symbols. The resulting 63×8 array of 8 bit symbols is transmitted to the D5 kit for storage in memory. The D5 is configured to receive data through *PIA port A* [34,sec.9.5], which is driven by the '32010's parallel output port. The '32010 transfers each 8 bit symbol in the array to the D5 kit by writing it into the parallel output port. This write operation causes a falling voltage transition on the Data Request line. In response to the resulting voltage transition on the CA1 pin, the D5 kit raises the CA2 pin to the high voltage state and reads PIA port A into memory. When the read is complete, CA2 falls back to the low voltage state. Meanwhile, the '32010 monitors CA2 via the BIO pin. When it observes a complete low →

high → low transition on CA2, it raises the Data Request line high again. It then writes the next byte of the array into the parallel output port.

(2) After receiving the entire array through PIA port A, the D5 is ready to return it to the '32010 through PIA port B. When the '32010 is ready to receive the data, it sets Data Request low. In response, the Data Ready line (fig. 3.4) falls from the high voltage state to the low voltage state exactly one half bit time before the first bit of the message is transmitted. The '32010 detects this transition on the BIO pin and begins sampling the channel when the first bit is transmitted.

The circuitry of fig. 3.4 converts the data on PIA port B to serial format. The serial data is clocked out of the '74165 shift register by the external clock signal. The external clock signal can be supplied by any TTL compatible pulse generator (such as the Wavetek 178). The allowable frequency range of the external clock signal is discussed in the next paragraph.

(3) The '32010 samples the channel at 100 kHz. A new channel sample is loaded into the rightmost storage cell of the LS299(2) shift register with every "IN, PA4" instruction. The two LS299 shift registers are read after every 16th channel sample with the "IN, PA3" instruction. The execution of this instruction causes the 16 bit shift register

entry to be written into the 2k×16 buffer memory. The buffer memory address register is automatically incremented at the end of each such write operation.

In order for the Viterbi Algorithm to work, the clock driving the D5 transmitter must have a frequency of no more than one half the receiver's sampling rate. Hence, the transmitter clock has an upper limit of 50 kHz. It is fairly easy to derive a lower frequency limit by using an argument similar to that of equation (3.1) in section 3.1. However, since we choose to work with transmitter frequencies very close to the 50 kHz upper limit, the derivation is omitted.

(4) Once the D5 has transmitted all 504 bytes in the array, it waits for the '32010 to generate and send a new array as explained in (1) above.

When the '32010 has collected 511×16 = 8176 channel samples, it applies the Viterbi Algorithm to the buffer memory contents to estimate the positions of the synch samples. It then estimates the transmitted data symbols by applying the decision rule mentioned in 3.2.1. Finally it applies the Reed-Solomon decoding algorithm to each of the 8 words in the resulting array. The '32010 compares the received symbols to the transmitted symbols before and after decoding and counts the number of errors. These error counts are stored and a new array of codewords is generated as described in (1). Steps (1) - (4) can be repeated as many times as desired. The PN sequence produces a different array "almost every time" (The arrays repeat at every n^{th} iteration, where n is the smallest integer such that $n \cdot 8 \cdot 441$ is a multiple of $2^{16}-1 = 3 \cdot 5 \cdot 17 \cdot 257$. Hence, $n = 5 \cdot 17 \cdot 257 = 21845$).

The test results of Table 2 are obtained with $2^{13} = 8192$ iterations of the preceding four steps (ie: $8192 \cdot 504 \simeq 4 \times 10^6$ bytes). In spite of the fact that the channel is noiseless, the synchronization method introduces a significant number of errors (refer to the "symbol error rate prior to decoding" column).

3.2.3 Remarks

(1) The sequence of 504 synch bits embedded in each message consists of 8 copies of a length 63 PN sequence. This type of sequence is expected to reduce the probability of a frame loss, described in 3.1.4. The metrics of paths involving frame losses should decrease rapidly as the Viterbi Algorithm progresses due to the highly peaked autocorrelation function of the PN sequence [35]. The error patterns observed in the tests indicate that frame losses actually never occur.

The pre-decoding error patterns are observed to occur in bursts concentrated around the nodes where $\delta_i = 1$. The codewords of each message are *interleaved* [3] to reduce the effect of these errors on the post-decoding error rate.

(2) The Viterbi Algorithm can be programmed in about 240 TMS32010 instructions. The subsequent estimation of the data symbols requires 120 instructions. The total running time of the two routines is upper bounded by 995,000 machine cycles, which corresponds to a maximum throughput of

$$\frac{(7 \text{ bits/symbol})(504 \text{ symbols/message})}{995{,}000 \times 200 \text{ ns}}$$

$$= 17.3 \text{ kbit/s.}$$

Thus, these two routines constitute a significant processing bottleneck (when compared to the speed of the Reed-Solomon decoding algorithm). Loosely speaking, the main difficulty with this approach is that a great deal of data (16 binary sequences $\{\delta_1, \ldots, \delta_n\}$ of length up to 504 bits, together with the transmitted message) must be packed into the limited memory resources of the system. A significant amount of processing time is required to "pack" and "unpack" this data as the Viterbi Algorithm progresses.

TABLE 2

transmitter frequency	symbol error rate prior to decoding	symbol error rate after decoding
49.99 kHz	1.15×10^{-2}	no errors ($< 2\times10^{-7}$)
49.97 kHz	1.67×10^{-2}	no errors ($< 2\times10^{-7}$)
49.95 kHz	1.91×10^{-2}	2.37×10^{-5}
49.93 kHz	2.40×10^{-2}	3.92×10^{-5}

Chapter 4

Summary and Conclusions

The purpose of this thesis is twofold: first, to investigate the implementation of Reed-Solomon decoders with the TMS32010 signal processor chip, and second, to present a synchronization technique that makes no use of phase locked loops.

The construction and decoding of Reed-Solomon codes is illustrated with an example of a single error correcting code over GF(11). Multiple error correcting Reed-Solomon codes over arbitrary fields are then introduced and their decoding is discussed. A slight variation of Berlekamp's algorithm for solving the key equation is developed and a method of accelerating Berlekamp's algorithm due to Chen is discussed. It is shown that the use of Chen's method involves a greater chance of miscorrecting large error patterns than does Berlekamp's algorithm.

The implementation of Galois field arithmetic on the TMS32010 is considered: it is shown that the 16 bit data handling capability of the '32010 can be used to implement fairly efficient mathematical operations in the fields GF(127) and GF(257). Based on these results, the programming of the Reed-Solomon decoding algorithm for codes over these fields is discussed. Codes over GF(127) are most efficiently applied when an alphabet of 127 symbols is used (ie: ASCII less a character). Furthermore, as shown by Solomon, an (n,k) Reed-Solomon code over GF(257) contains a (nonlinear) (n,k-1) subcode over GF(257)-{0}, allowing the use of eight bits per transmitted symbol. For high code rates, an encoder for the subcode can be realized as a practical modification of the usual Reed-Solomon encoder. Thus, codes over these prime fields may be applied to binary systems almost as easily as codes over fields of characteristic two.

A stack storage format for decoding is presented and analyzed. The results of Appendix 4 show how large these stacks must be. While similar decoding methods have been proposed elsewhere by Cohen, the issue of stack size is not addressed there.

A Reed-Solomon decoder based on the TMS32010 is designed and built as part of this thesis. Full details on this hardware system are presented in a separate report. It is demonstrated that the throughput of the '32010-based decoder for codes over GF(257) is comparable to that of a more complicated bit-slice design intended for codes over GF(256).

Cohen has shown that it is possible to improve the performance of the bit-slice Reed-Solomon decoder by adding a large amount of hardware. However, the performance of the '32010-based decoder can be improved to an even greater extent by replacing the TMS32010 with a more powerful signal processor such as the TMS320C30. This replacement would likely involve a reduction of the decoder parts count due to the increasing sophistication of the more recent signal processor chips. For example, it is estimated that a 10 chip

'320C30-based decoder for the (256,240) Reed-Solomon code over GF(257) would provide a throughput higher than that obtained with a 100 chip bit-slice decoder for the (255,239) code over GF(256). Thus, the present trend towards more powerful signal processor devices allows the construction of relatively cheap decoders that are faster and far more compact than bit-slice designs.

A method of inserting timing information into a transmitted data stream and recovering it at the receiver without relying on well-known timing recovery techniques is presented. The receiver uses a version of the Viterbi Algorithm, which is more commonly used in the decoding of convolutional codes. While the direct analysis of the technique appears to be difficult, the study of an idealized model (the independent insertion channel) yields interesting upper bounds on synchronization error and frame loss probabilities.

A '32010 implementation of this technique on a binary channel provides a throughput of over 17 kbit/s. It is felt that the throughput obtained with this technique could be significantly increased by adding a form of memory to the receiver. The "synch path" traced by the receiver during the first data block might be used in subsequent data blocks to help eliminate many unlikely paths that the Viterbi Algorithm would otherwise search. Furthermore, the use of this synchronization technique with multilevel signals is expected to provide better error performance than that obtained in the binary case. Whatever the application may be, a special purpose hardware architecture would provide much higher throughput than that obtained with the '32010.

Bibliography

[1] H. Hoeve, et al, "Error correction and concealment in the compact disk system," *Phillips Technical Review*, vol. 40, pp. 166-172, 1982.

[2] P. Chen, "The compact disk ROM: how it works," *IEEE Spectrum*, vol. 23, no. 4, pp. 44-49, 1986.

[3] E. R. Berlekamp, "Bit-serial Reed-Solomon encoders," *IEEE Trans. Inform. Theory*, vol. 28, pp. 869-874, 1982.

[4] IEEE Information Theory Group Newsletter, Spring 1986, p. 8.

[5] B. C. Mortimer, et al, "A high performance error-correcting scheme for the Canadian broadcast Telidon system based on Reed-Solomon codes," presented at *Thirteenth Biennial Symposium on Communications*, Kingston, ON, June 2-4, 1986.

[6] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. SIAM*, vol. 8, pp. 300-304, 1960.

[7] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error Correcting Codes*. Amsterdam: North-Holland, 1977.

[8] E. R. Berlekamp, *Algebraic Coding Theory*. Laguna Hills: Aegean Park Press, 1984.

[9] D. E. Knuth, *The Art of Computer Programming*, vol. 2, *Seminumerical Algorithms*. Reading: Addison-Wesley, 1981.

[10] N. Patterson, "The algebraic decoding of Goppa codes," *IEEE Trans. Inform. Theory*, vol. 21, pp. 203-207, 1975.

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

[12] C. L. Chen, "High-speed decoding of BCH codes," *IEEE Trans. Inform. Theory*, vol. 27, pp. 254-256, 1981.

[13] R. E. Blahut, *Theory and Practice of Error Control Codes*. Reading: Addison-Wesley, 1983.

[14] L. R. Welch and R. A. Scholtz, "Continued fractions and Berlekamp's algorithm," *IEEE Trans. Inform. Theory*, vol. 25, pp. 19-27, 1979.

[15] R. J. McEliece, *The Theory of Information and Coding*. Reading: Addison-Wesley, 1977.

[16] E. T. Cohen, *On the Implementation of Reed-Solomon Decoders*. Ph.D. dissertation, University of California, Berkeley, 1983.

[17] E. T. Cohen, "Special purpose digital hardware," in *The Impact of Processing Techniques on Communications*. J. K. Skwirzynski, ed. The Netherlands: Martinus Nijhoff, 1985.

[18] R. E. Blahut, "Algebraic fields, signal processing, and error control," *Proc. IEEE*, vol. 73, pp. 874-893, 1985.

[19] J. K. Wolf, "Redundancy, the Discrete Fourier Transform, and impulse noise cancellation," *IEEE Trans. Comm.*, vol. 31, pp. 458-461, 1983.

[20] G. Solomon, "A note on alphabetic codes and fields of computation," *Inf. and Control*, vol. 25, pp. 395-398, 1974.

[21] L. R. Morris, "Good FFT software stretches processor performance," *IEEE Micro*, vol. 6, no. 2, pp. 4-5, 1986.

[22] *TMS32010 User's Guide*. Texas Instruments, 1985.

[23] *Motorola Memory Data*. Motorola, 1984.

[24] A. Osborne and G. Kane, *Osborne 16-Bit Microprocessor Handbook*. Berkeley: Osborne/McGraw-Hill, 1981.

[25] *Details on Signal Processing*, Texas Instruments, Issue 7, 1986.

[26] "Motorola's sizzling new signal processor," *Electronics*, pp. 30-33, March 10, 1986.

[27] W. I. Fletcher, *An Engineering Approach to Digital Design*. New Jersey: Prentice-Hall, 1980.

[28] *Motorola Schottky TTL Data*. Motorola, 1983.

[29] *IEEE Transactions on Communications* special issue on synchronization. vol. 28, Aug. 1980.

[30] J. G. Proakis, *Digital Communications*. New York: McGraw-Hill, 1983.

[31] R. A. Scholtz, "Frame synchronization techniques," *IEEE Trans. Comm.*, vol. 28, pp. 1204-1213, 1980.

[32] A. J. Viterbi and J. K. Omura, *Principles of Digital Communication and Coding*. New York: McGraw-Hill, 1979.

[33] G. Ungerboeck, "New application for the Viterbi Algorithm: Carrier phase tracking in synchronous data transmission systems," in *Proc. Nat. Telecomm. Conf.*, 1974, pp. 734-738.

[34] R. Bishop, *Basic Microprocessors and the 6800*. New Jersey: Hayden, 1979.

[35] R. L. Pickholtz, et al, "Theory of spread-spectrum communications - a tutorial," in *Spread-Spectrum Communications*. Cook, et al, ed. New York: IEEE Press, 1983.

[36] B. L. Johnson, "Design and hardware implementation of a versatile transform decoder for Reed-Solomon codes," in *The Impact of Processing Techniques on Communications*. J. K. Skwirzynski, ed. The Netherlands: Martinus Nijhoff, 1985.

[37] E. R. Berlekamp, "The technology of error correcting codes," *Proc. IEEE*, vol. 68, pp. 564-593, 1980.

[38] H. M. Shao, et al, "A VLSI design of a pipelined Reed-Solomon decoder," *IEEE Trans. Computers*, vol. 34, pp. 393-403, 1985.

[39] D. Peterson, "TMS32010 Hardware Manual," Technical Report, U. of A. Department of Electrical Engineering, 1987.

[40] *Details on Signal Processing*, Texas Instruments, Issue 10, 1987.

Appendix 1

Berlekamp's Iterative Algorithm

The following basic result on relatively prime polynomials will prove to be useful:

Claim 1: Let $A(z)$, $B(z)$ be relatively prime polynomials. Suppose $A'(z)$, $B'(z)$ are nonzero polynomials satisfying

$$A'(z)B(z) = A(z)B'(z). \tag{1.9}$$

Then $A(z)$ is a divisor of $A'(z)$, $B(z)$ is a divisor of $B'(z)$, and

$$\max[\deg A', \deg B'] \geq \max[\deg A, \deg B]. \tag{1.10}$$

If equality holds in (1.10), then

$$\deg A = \deg A' \text{ and } \deg B = \deg B'.$$

Proof: Choose polynomials $a(z)$, $b(z)$ such that $a(z)A(z) + b(z)B(z) = 1$. Multiplying this equality through by $A'(z)$, one obtains

$$A(z)a(z)A'(z) + b(z)A'(z)B(z) = A'(z),$$

implying

$$A(z)[a(z)A'(z) + b(z)B'(z)] = A'(z),$$

using (1.9). Hence, $A(z)$ is a divisor of $A'(z)$. The same reasoning shows that $B(z)$ is a divisor of $B'(z)$. Consequently,

$$\deg A \le \deg A',$$

and

$$\deg B \le \deg B'. \tag{1.11}$$

Furthermore, (1.9) implies

$$\deg A' - \deg A = \deg B' - \deg B. \tag{1.12}$$

From (1.12), if $\deg B = \deg B'$, then $\deg A = \deg A'$, and equality holds in (1.10). The proof will be completed by showing that the condition $\deg B < \deg B'$ implies strict inequality in (1.10).

Suppose, then, that $\deg B < \deg B'$. By (1.12),

$$\deg A = [\deg B - \deg B'] + \deg A' < \deg A'.$$

Thus, $\deg B < \deg B'$ and $\deg A < \deg A'$, implying strict inequality in (1.10). ∎

The next result is a variation of [10,Theorem 1].

Claim 2: Let $S(z)$ be a generating function over a field F such that $S(0) = 0$. Suppose polynomials $\sigma(z)$ and $\omega(z)$ over F satisfy

(1) $(1 + S)\sigma = \omega \bmod z^p$,

(2) $\sigma(0) = 1$,

(3) $(\sigma,\omega) = 1$,

for some positive integer p.

Suppose nonzero polynomials $\sigma'(z)$, $\omega'(z)$ can be found (by whatever means possible) such that

(1') $(1 + S)\sigma' = \omega' \bmod z^p$,

(2') $\sigma'(0) = 1$,

(3') $\max[\deg \sigma', \deg \omega'] \le \max[\deg \sigma, \deg \omega]$.

Then if $\max[\deg \sigma, \deg \omega] < p/2$, we have $\sigma' = \sigma$ and $\omega' = \omega$.

Proof: Multiplying (1) by σ' gives

$$\sigma'(1 + S)\sigma = \sigma'\omega \bmod z^p.$$

Applying (1'), this becomes

$$\sigma\omega' = \sigma'\omega \bmod z^p.$$

Now

$$\deg \sigma + \deg \omega' \leq 2(\max[\deg \sigma, \deg \omega]) < p,$$

and

$$\deg \sigma' + \deg \omega \leq 2(\max[\deg \sigma, \deg \omega]) < p.$$

Hence, the previous mod z^p equality is a true equality:

$$\sigma\omega' = \sigma'\omega.$$

By Claim 1 and (3'), σ is a divisor of σ' and $\deg \sigma = \deg \sigma'$. It follows that σ' is a field multiple of σ. By (2) and (2'), this field multiple must be 1, implying $\sigma' = \sigma$. Putting this result into the equality $\sigma\omega' = \sigma'\omega$ gives $\omega' = \omega$. This completes the proof of Claim 2. ∎

Suppose $S(z)$, $\sigma(z)$, $\omega(z)$, and p are as described in the first part of Claim 2. We now construct an algorithm that, when applied to $S(z)$, produces a sequence of polynomials $\sigma_0(z)$, $\omega_0(z)$, $\sigma_1(z)$, $\omega_1(z)$, . . . , $\sigma_{p-1}(z)$, $\omega_{p-1}(z)$ such that for each $0 \leq k \leq p-1$:

(a) $(1 + S)\sigma_k = \omega_k \bmod z^{k+1}$,

(b) $\sigma_k(0) = 1$,

(c) $\max[\deg \sigma_k, \deg \omega_k] \leq \max[\deg \sigma, \deg \omega]$.

If $\max[\deg \sigma, \deg \omega] < p/2$, Claim 2 implies $\sigma_{p-1}(z) = \sigma(z)$ and $\omega_{p-1}(z) = \omega(z)$.

Until further notice, let S(z), $\sigma(z)$, $\omega(z)$, *and* p *be as described in the first part of Claim 2, and let* k *be an integer satisfying* $0 \le k < p-1$.

Suppose $\sigma_k(z)$ and $\omega_k(z)$ satisfy (a) and (b). Also suppose that

(d) $\deg \omega_k(z) \le k$.

Then

$$(1 + S)\sigma_k = \omega_k + \Delta_k z^{k+1} \bmod z^{k+2},$$

Δ_k being the coefficient of z^{k+1} in the product $(1 + S)\sigma_k$. If we can find polynomials $\tau_k(z)$ and $\gamma_k(z)$ such that

(e) $(1 + S)\tau_k = \gamma_k + z^k \bmod z^{k+1}$,

(f) $\deg \gamma_k \le k$,

then defining

$$\sigma_{k+1} = \sigma_k - \Delta_k z \tau_k, \tag{1.13}$$

$$\omega_{k+1} = \omega_k - \Delta_k z \gamma_k, \tag{1.14}$$

(a), (b), and (d) are satisfied with k replaced by k+1 [8,p.181]. Thus, (1.13) and (1.14) can be used to generate polynomials $\sigma_0(z)$, $\omega_0(z)$, $\sigma_1(z)$, $\omega_1(z)$, . . . , $\sigma_{p-1}(z)$, $\omega_{p-1}(z)$ such that (a) and (b) hold for each value of k. We next show how condition (c) can also be guaranteed. After this, the actual choice of τ_k and γ_k will be discussed.

The gcd (σ_k,ω_k) of σ_k and ω_k satisfies $(\sigma_k,\omega_k)(0) \neq 0$ because it is a divisor of σ_k and $\sigma_k(0) = 1$. Multiplying both sides of (a) by the generating function $1/(\sigma_k,\omega_k)$, we obtain

$$(1 + S)\sigma_k' = \omega_k' \bmod z^{k+1},$$

where $\sigma_k' = \sigma_k/(\sigma_k,\omega_k)$, and $\omega_k' = \omega_k/(\sigma_k,\omega_k)$. Hence, if σ_k and ω_k are not relatively prime, σ_k', ω_k' are lower degree solutions of (a). These can be multiplied by some field element (if necessary) to get $\sigma_k'(0) = 1$. Then the primed polynomials satisfy (a) and (b) and have lower degree than the original polynomials.

In view of condition (c), we are interested in finding solutions σ_k and ω_k of (a) having degrees as small as possible. Hence, it would be desirable to have σ_k and ω_k relatively prime in the first place. The next claim gives a simple condition under which $(\sigma_k,\omega_k) = 1$.

Claim 3: Suppose σ_k, ω_k, τ_k, and γ_k satisfy (a), (b), (d), (e), and (f). Then if

$$\deg \omega_k + \deg \tau_k \leq k,$$

and

$$\deg \sigma_k + \deg \gamma_k \leq k,$$

we have $(\sigma_k,\omega_k) = 1$.

Proof: Multiplying (e) by σ_k and using (a) gives

$$\tau_k\omega_k = \sigma_k(\gamma_k + z^k) \bmod z^{k+1}.$$

By (b),

$$\sigma_k(z)z^k = z^k \bmod z^{k+1},$$

implying

$$\tau_k\omega_k = \sigma_k\gamma_k + z^k \bmod z^{k+1}.$$

By the hypotheses of the claim, deg $\tau_k\omega_k \le k$ and deg $\sigma_k\gamma_k \le k$. Hence, the previous mod z^{k+1} equality is an actual equality:

$$\tau_k\omega_k - \sigma_k\gamma_k = z^k.$$

According to this formula, any common monic divisor of σ_k and ω_k must be a divisor of z^k. Hence, this divisor is a power of z. Since $\sigma_k(0) \neq 0$, the divisor must be $z^0 = 1$, implying that $(\sigma_k,\omega_k) = 1$. This proves Claim 3. ∎

By Claim 3, if there is an integer D(k) such that

$$0 \le D(k) \le k, \tag{1.15}$$

$$\max[\deg \sigma_k, \deg \omega_k] \le D(k), \tag{1.16}$$

$$\max[\deg \tau_k, \deg \gamma_k] \le k-D(k), \tag{1.17}$$

then $(\sigma_k,\omega_k) = 1$, as desired. The existence of such a D(k) also implies the following result.

Claim 4: Suppose (a), (b), (d), (e), and (f) hold and suppose polynomials $\sigma^*(z)$, $\omega^*(z)$ satisfy

$$(1 + S)\sigma^* = \omega^* \bmod z^{k+1}, \qquad (1.18)$$
$$\sigma^*(0) = 1.$$

Then if (1.15), (1.16), and (1.17) hold, we have

$$\max[\deg \sigma^*, \deg \omega^*] \geq D(k).$$

Proof: Multiplying (e) by σ^* and using (1.18) gives

$$\tau_k\omega^* = \sigma^*(\gamma_k + z^k) \bmod z^{k+1},$$
$$= \sigma^*\gamma_k + z^k \bmod z^{k+1}.$$

Hence, $\tau_k\omega^* - \sigma^*\gamma_k = z^k \bmod z^{k+1}$, which implies

$$\deg\{\tau_k\omega^* - \sigma^*\gamma_k\} \geq k.$$

It is clear that $\deg\{\tau_k\omega^* - \sigma^*\gamma_k\} \leq \max[\deg \tau_k\omega^*, \deg \sigma^*\gamma_k]$ $\leq k - D(k) + \max[\deg \sigma^*, \deg \omega^*]$. Hence,

$$k \leq k - D(k) + \max[\deg \sigma^*, \deg \omega^*],$$

implying

$$D(k) \leq \max[\deg \sigma^*, \deg \omega^*]. \qquad \blacksquare$$

From (a), we can take $\sigma^* = \sigma_k$, $\omega^* = \omega_k$. If (1.15), (1.16), and (1.17) hold, Claim 4 and (1.16) imply that

$$\max[\deg \sigma_k, \deg \omega_k] = D(k). \tag{1.19}$$

We can also take $\sigma^* = \sigma$, $\omega^* = \omega$. If (1.15), (1.16), and (1.17) hold, then Claim 4 implies

$$\max[\deg \sigma_k, \deg \omega_k] = D(k) \le \max[\deg \sigma, \deg \omega].$$

Hence, condition (c) is satisfied.

Summarizing the previous results,

Suppose σ_k, ω_k, τ_k, γ_k satisfy (a), (b), (d), (e), and (f). If there is an integer $D(k)$ such that (1.15), (1.16), and (1.17) hold, then σ_k and ω_k are relatively prime, (1.19) holds, and condition (c) is satisfied.

It is easy to construct σ_0, ω_0, τ_0, γ_0, and $D(0)$ such that (a), (b), (d), (e), (f), (1.15), (1.16), and (1.17) hold for $k = 0$:

$$\sigma_0(z) = 1,\ \omega_0(z) = 1,\ \tau_0(z) = 1,\ \gamma_0(z) = 0,\ D(0) = 0.$$

(1.13) and (1.14) show how to construct σ_{k+1} and ω_{k+1} from σ_k, ω_k, τ_k, and γ_k. We now define

$$D(k+1) = D(k),$$
$$\tau_{k+1}(z) = z\tau_k(z),$$
$$\gamma_{k+1}(z) = z\gamma_k(z),$$

when $\Delta_k = 0$ or $D(k) \geq (k+1)/2$, and define

$$D(k+1) = k+1-D(k),$$
$$\tau_{k+1}(z) = \Delta_k^{-1}\sigma_k(z),$$
$$\gamma_{k+1}(z) = \Delta_k^{-1}\omega_k(z),$$

when $\Delta_k \neq 0$ and $D(k) < (k+1)/2$. It is straightforward to verify that if (a), (b), (d), (e), (f), (1.15), (1.16), and (1.17) hold for k, then they also hold with k replaced by k+1. Hence, σ_{k+1} and ω_{k+1} are relatively prime and condition (c) holds with k replaced by k+1.

This iterative construction is summarized in Berlekamp's Iterative Algorithm, presented in 1.2.3.

Appendix 2

Chen's exit condition

Suppose $e \le t$ errors occur. Claim 2 of Appendix 1 implies $\sigma_{t+e} = \sigma$ and $\omega_{t+e} = \omega$, so by (1.19),

$$D(t+e) = \max[\deg \sigma_{t+e}, \deg \omega_{t+e}]$$

$$= \max[\deg \sigma, \deg \omega]$$

$$= e.$$

In other words, $D(k)=k-t$ for $k=t+e$. Conversely, if $D(k)=k-t$ for some $t \le k \le 2t$, then for all $k' > k$, $D(k')=D(k)$ (Otherwise, since the $D(i)$'s form a nondecreasing sequence, there exists $k' > k$ such that $D(k) = D(k'-1) < D(k')$. This implies $D(k') = k'-D(k'-1) > k-D(k) = t$. However, Claim 4 implies $D(k') \le \max[\deg \sigma, \deg \omega] = e \le t$. This is a contradiction). In particular,

$$D(k) = D(2t)$$

$$= \max[\deg \sigma_{2t}, \deg \omega_{2t}]$$

$$= \max[\deg \sigma, \deg \omega]$$

$$= e,$$

implying that k-t=e, or k=t+e.

This argument shows that the replacement of the original exit condition

"If k = 2t, stop"

in Berlekamp's Iterative Algorithm with the exit condition

"If D(k) = k-t or if k = 2t, stop"

will ensure the correction of any pattern of up to t errors and an exit after exactly t+e iterations of the algorithm, where e is the actual number of errors.

Appendix 3

Proof of the aliasing property

Consider an error polynomial of the form

$$E(x) = Y_1 x^{i_1} + Y_2 x^{i_2} + \ldots + Y_e x^{i_e},$$

where $e \leq t$. The first $t+e$ syndromes corresponding to this error pattern are given by

$$S_k = \sum_{m=1}^{e} \alpha^{k i_m} Y_m, \quad 1 \leq k \leq t+e.$$

Select $0 \leq j_0 \leq n-1$ such that $j_0 \neq i_m$ for any $1 \leq m \leq e$. Then select $0 \leq j_1 < j_2 < \ldots < j_{t+e} \leq n-1$ such that $j_k \neq j_0$ for any $1 \leq k \leq t+e$. The $t+e$ by $t+e$ matrix

$$[\alpha^{k j_m}]_{1 \leq k, m \leq t+e}$$

is nonsingular [13,Theorem 7.2.1]. Hence, there is a unique vector $\{Z_1, Z_2, \ldots, Z_{t+e}\}$ such that

$$S_k - \alpha^{k j_0} = \sum_{m=1}^{t+e} \alpha^{k j_m} Z_m, \quad 1 \leq k \leq t+e.$$

But then

$$S_k = \sum_{m=0}^{t+e} \alpha^{k j_m} Z_m, \quad 1 \leq k \leq t+e,$$

where $Z_0 = 1$. It follows that the error polynomial

$$E'(x) = x^{j_0} + Z_1 x^{j_1} + \ldots + Z_{t+e} x^{j_{t+e}}$$

has the same first t+e syndromes as E(x). Obviously $E'(x) \neq E(x)$ and E'(x) has at most t+e+1 nonzero coefficients.

Appendix 4

On the size of the stacks

It will now be shown that $p(k)+D(k) \le 2t+1$ and that $p(k)+k-D(k) \le 2t+1$, as claimed in 2.2.3.

Claim: (a) Let $0 \le k < 2t$. If $\Delta_k \ne 0$ and $D(k) < (k+1)/2$, then

$$\max[\deg z\tau_k, \deg z\gamma_k] = D(k+1).$$

Let $0 \le k \le 2t$. Then

(b) $\max[\deg z^{p(k)}\tau_k, \deg z^{p(k)}\gamma_k] \le 2t$,

(c) $p(k)+D(k) \le 2t+1$,

(d) $p(k)+k-D(k) \le 2t+1$.

Proof:

(a): From (1.13), (1.14), and (1.19) of Appendix 1,

$$\sigma_{k+1} = \sigma_k - \Delta_k z\tau_k,$$
$$\omega_{k+1} = \omega_k - \Delta_k z\gamma_k,$$

and

$$\max[\deg \sigma_k, \deg \omega_k] = D(k)$$
$$< D(k+1) = \max[\deg \sigma_{k+1}, \deg \omega_{k+1}].$$

Property (a) follows immediately.

(b): The stated result is obviously true for $k = 0$. Suppose it holds for $0 \le k < 2t$.

(i) If $\Delta_k = 0$ or $D(k) \ge (k+1)/2$, then

$$z^{p(k+1)}\tau_{k+1} = z^{p(k)-1}z\tau_k = z^{p(k)}\tau_k.$$

Similarly, $z^{p(k+1)}\gamma_{k+1} = z^{p(k)}\gamma_k$. Hence,

$$\max[\deg z^{p(k+1)}\tau_{k+1}, \deg z^{p(k+1)}\gamma_{k+1}]$$

$$= \max[\deg z^{p(k)}\tau_k, \deg z^{p(k)}\gamma_k]$$

$$\le 2t,$$

by the inductive hypothesis.

(ii) If $\Delta_k \ne 0$ and $D(k) < (k+1)/2$, then by (a),

$$\max[\deg z^{p(k)}\tau_k, \deg z^{p(k)}\gamma_k] = p(k)-1+D(k+1).$$

Hence, the inductive hypothesis implies

$$p(k)-1+D(k+1) \le 2t. \tag{2.1}$$

Now

$$z^{p(k+1)}\tau_{k+1} = \Delta_k^{-1}z^{p(k+1)}\sigma_k$$

and

$$z^{p(k+1)}\gamma_{k+1} = \Delta_k^{-1} z^{p(k+1)}\omega_k,$$

implying

$$\max[\deg z^{p(k+1)}\tau_{k+1}, \deg z^{p(k+1)}\gamma_{k+1}]$$

$$= p(k+1) + \max[\deg \sigma_k, \deg \omega_k]$$

$$= p(k)+D(k)$$

$$\le p(k)+D(k+1)-1$$

$$\le 2t,$$

where the last inequality follows from (2.1). This proves (b) by induction.

(c) and (d): Both of these inequalities are obviously true for $k = 0$. Suppose both hold for $0 \le k < 2t$.

(i) If $\Delta_k = 0$ or $D(k) \ge (k+1)/2$, then

$$p(k+1)+D(k+1) = p(k)-1+D(k) < 2t+1,$$

and

$$p(k+1)+k+1-D(k+1) = p(k)-1+k+1-D(k) \le 2t+1,$$

by the inductive hypothesis.

(ii) If $\Delta_k \ne 0$ and $D(k) < (k+1)/2$, then

$$\begin{aligned} p(k+1)+k+1-D(k+1) &= p(k+1)+D(k) \\ &= p(k)+D(k) \leq 2t+1. \end{aligned}$$

Furthermore,

$$\begin{aligned} p(k+1)+D(k+1) &= p(k)+D(k+1) \\ &= \max[\deg z^{p(k)}\tau_k, \deg z^{p(k)}\gamma_k] + 1 \leq 2t+1, \end{aligned}$$

where the second equality follows from (a) and the subsequent inequality follows from (b). This proves (c) and (d) by induction and completes the proof of the claim. ∎

Appendix 5

A bound on synch error probability

Suppose [illegible]t condition (1) of 3.1.2 holds. Since $[K_m, J_m]^* =$ [illegible]$_m]$, the sequence

$$\{[K_0,J_0]^*,\ [K_1,J_1]^*,\ \ldots,\ [K_m,J_m]^*,\ [K_{m+1},J_{m+1}],\ \ldots,\ [K_{m+j},J_{m+j}]\}$$

is a path. Since $[K_{m+j},J_{m+j}] = [K_{m+j},J_{m+j}]^*$, it has the same endpoint as the path

$$\{[K_0,J_0]^*,\ [K_1,J_1]^*,\ \ldots,\ [K_{m+j},J_{m+j}]^*\}.$$

Let Y_i^0 be the RV stored at $[K_i,J_i]$ and let Y_i^* be the RV stored at $[K_i,J_i]^*$. Since the latter path is the survivor with endpoint $[K_{m+j},J_{m+j}]^*$, the claim following the Viterbi Algorithm in 3.1.1 implies that the metric

$$\sum_{i=0}^{m} \ln p(Y_i^*|S_i) + \sum_{i=m+1}^{m+j} \ln p(Y_i^0|S_i) + (m+j+1)R$$

of the first path does not exceed the metric

$$\sum_{i=0}^{m+j} \ln p(Y_i^*|S_i) + (m+j+1)R$$

of the second one. Hence, either

$$\sum_{i=m+1}^{m+j-1} \ln p(Y_i^0|S_i) \le \sum_{i=m+1}^{m+j-1} \ln p(Y_i^*|S_i),$$

or

$$\sum_{i=0}^{m} \ln p(Y_i^*|S_i) = -\infty,$$

or

$$\ln p(Y_{m+j}^*|S_{m+j}) = \ln p(Y_{m+j}^0|S_{m+j}) = -\infty.$$

Since $\ln p(Y_i^0|S_i) = -\infty$ with probability zero for all i and since

$$\sum_{i=0}^{m} \ln p(Y_i^*|S_i) \ge \sum_{i=0}^{m} \ln p(Y_i^0|S_i)$$

by the claim following the Viterbi Algorithm, the latter two events have probability zero.

Applying the *Chernoff bound* [32,p.122], the probability

$$\mathbf{Pr}\{\sum_{i=m+1}^{m+j-1} \ln p(Y_i^0|S_i) \le \sum_{i=m+1}^{m+j-1} \ln p(Y_i^*|S_i)\}$$

of the first event is upper bounded by

$$\mathbf{E} \exp \tfrac{1}{2}\{\sum_{i=m+1}^{m+j-1} \ln p(Y_i^*|S_i) - \sum_{i=m+1}^{m+j-1} \ln p(Y_i^0|S_i)\}$$

$$= \mathbf{E} \prod_{i=m+1}^{m+j-1} [p(Y_i^*|S_i)/p(Y_i^0|S_i)]^{1/2},$$

where **E** denotes expectation. This expectation is given by

$$\sum_{s,y^*,y^0} \Pr \bigcap_{i=m+1}^{m+j-1} \{S_i=s_i,\ Y_i^*=y_i^*,\ Y_i^0=y_i^0\} \times$$

$$\prod_{i=m+1}^{m+j-1} [p(y_i^*|s_i)/p(y_i^0|s_i)]^{1/2}, \qquad (3.6)$$

where the sum varies over all possible elements

$$s = (s_{m+1},\ldots,s_{m+j-1}),$$

$$y^* = (y_{m+1}^*,\ldots,y_{m+j-1}^*),$$

$$y^0 = (y_{m+1}^0,\ldots,y_{m+j-1}^0)$$

of $F \times F \times \ldots \times F = F^{j-1}$.

By hypothesis, no Y_i^* RV coincides with a Y_l^0 RV if $m+1 \le i,l \le m+j-1$. Hence,

$$\Pr \bigcap_{i=m+1}^{m+j-1} \{S_i=s_i,\ Y_i^*=y_i^*,\ Y_i^0=y_i^0\}$$

$$= \prod_{i=m+1}^{m+j-1} \Pr\{Y_i^*=y_i^*\}\Pr\{S_i=s_i,\ Y_i^0=y_i^0\}$$

$$= \prod_{i=m+1}^{m+j-1} p(y_i^*)S(s_i)p(y_i^0|s_i),$$

where $p(y) = \sum_{x \in F} p(y|x)D(x)$. (3.6) can then be written as

$$\sum_{s,y^*,y^0} \prod_{i=m+1}^{m+j-1} p(y_i^*)S(s_i)[p(y_i^*|s_i)p(y_i^0|s_i)]^{1/2}$$

$$= \prod_{i=m+1}^{m+j-1} \sum_{s_i \in F} \sum_{y_i^* \in F} \sum_{y_i^0 \in F} p(y_i^*)S(s_i)[p(y_i^*|s_i)p(y_i^0|s_i)]^{1/2}$$

$$= Z^{j-1},$$

where

$$Z \equiv \sum_{s \in F} S(s) \sum_{y_1 \in F} \sum_{y_2 \in F} p(y_1)[p(y_1|s)p(y_2|s)]^{1/2}. \qquad (3.7)$$

Summarizing,

$$\Pr\{\sum_{i=m+1}^{m+j-1} \ln p(Y_i^0|S_i) \le \sum_{i=m+1}^{m+j-1} \ln p(Y_i^*|S_i)\} \le Z^{j-1}.$$

Now there are at most 2^{j-1} possible choices for the set of RVs $Y^*_{m+1}, Y^*_{m+2}, \ldots, Y^*_{m+j-1}$ because there are at most 2^{j-1} possible choices for the set of nodes $[K_{m+1},J_{m+1}]^*, \ldots, [K_{m+j-1},J_{m+j-1}]^*$. Applying the *union bound* [32,p.61], we conclude that the probability that the path produced by the Viterbi Algorithm diverges from the correct path at node m and remerges at node m+j is upper bounded by $2^{j-1}Z^{j-1}$.

One more application of the union bound shows that the probability that condition (1) holds is upper bounded by

$$\sum_{m=0}^{k-1} \sum_{j=k+1-m}^{N-m} (2Z)^{j-1}$$

$$= 2Z(1-(2Z)^k)(1-(2Z)^{N-k})/(1-2Z)^2. \qquad (3.8)$$

Now suppose that condition (2) of 3.1.2 holds. The survivor with endpoint $[K_N,J_N]$ must have length N. If it had any other length, this would contradict the hypothesis that no path slips or advances by n+1 samples relative to the correct path. Hence, this survivor and its metric will be saved by the stopping procedure at the end of the algorithm. Write this survivor as

$$\{[K_0,J_0]^{**},\ [K_1,J_1]^{**},\ \ldots,\ [K_N,J_N]^{**}\}. \tag{3.9}$$

By hypothesis,

$$[K_m,J_m]^* = [K_m,J_m].$$

Hence, the sequence

$$\{[K_0,J_0]^*,\ [K_1,J_1]^*,\ \ldots,\ [K_m,J_m]^*,\ [K_{m+1},J_{m+1}],\ \ldots,\ [K_N,J_N]\}$$

is a length N-path ending at $[K_N,J_N]^{**} = [K_N,J_N]$. By the claim following the Viterbi Algorithm, the metric of this path does not exceed the metric of the survivor (3.9). In turn, the metric of the survivor (3.9) does not exceed the metric of the survivor

$$\{[K_0,J_0]^*,\ [K_1,J_1]^*,\ \ldots,\ [K_N,J_N]^*\}.$$

We conclude that

$$\sum_{i=0}^{m} \ln p(Y_i^*|S_i) + \sum_{i=m+1}^{N} \ln p(Y_i^0|S_i) + (N+1)R$$

$$\leq \sum_{i=0}^{m} \ln p(Y_i^*|S_i) + \sum_{i=m+1}^{N} \ln p(Y_i^*|S_i) + (N+1)R,$$

implying that

$$\sum_{i=m+1}^{N} \ln p(Y_i^0|S_i) \leq \sum_{i=m+1}^{N} \ln p(Y_i^*|S_i),$$

or

$$\sum_{i=0}^{m} \ln p(Y_i^*|S_i) = -\infty.$$

As before, the second of these events has probability zero. Applying the Chernoff bound to the first event, then applying union bounds as before, the probability that condition (2) holds is upper bounded by

$$\sum_{m=0}^{k-1} (2Z)^{N-m}$$

$$= (2Z)^{N-k+1}(1-(2Z)^k)/(1-2Z). \qquad (3.10)$$

Appendix 6

A bound on frame loss probability

Recalling the notation of 3.1.4, we write the survivor P as

$$\{[K_0,J_0]^s,\ [K_1,J_1]^s,\ \ldots,\ [K_{p+1},J_{p+1}]^s\}.$$

By hypothesis, $[K_{p+1},J_{p+1}]^s = [K_p,J_p]$.

The survivor P' ending at memory location $[K_{p-1},J_{p-1}]$ must have length p-1. Otherwise, the first frame loss would have occurred at $[K_{p-1},J_{p-1}]$ or at an earlier memory location. Let q be the largest integer such that $[K_q,J_q]^s = [K_q,J_q]$. Then

$$\{[K_0,J_0]^s,\ [K_1,J_1]^s,\ \ldots,\ [K_q,J_q]^s,\ [K_{q+1},J_{q+1}],\ \ldots,\ [K_{p-1},J_{p-1}]\} \tag{3.11}$$

is a path.

Let Y_i^s be the RV stored at $[K_i,J_i]^s$, Y_i^0 the RV stored at $[K_i,J_i]$. By the claim following the Viterbi Algorithm, the metric of the path (3.11) is no larger than the metric of P'. In turn, the metric of $P'*[K_p,J_p]$ is no larger than the metric of P. Otherwise, $P'*[K_p,J_p]$ would be the survivor ending at $[K_p,J_p]$ instead of P. Putting these two inequalities together gives:

$$\sum_{i=0}^{q} \ln p(Y_i^s|S_i) + \sum_{i=q+1}^{p} \ln p(Y_i^0|S_i) + (p+1)R$$

$$\leq \sum_{i=0}^{p+1} \ln p(Y_i^s|S_i) + (p+2)R,$$

implying

$$\sum_{i=q+1}^{p} \ln p(Y_i^0|S_i) \leq \sum_{i=q+1}^{p+1} \ln p(Y_i^s|S_i) + R,$$

or

$$\sum_{i=0}^{q} \ln p(Y_i^s|S_i) = -\infty.$$

By the usual argument, the second of these events has probability zero. The first event implies

$$\sum_{i=q+1}^{p} \ln p(Y_i^0|S_i) \leq \sum_{i=q+1}^{p} \ln p(Y_i^s|S_i) + R,$$

since $\ln p(Y_{p+1}^0|S_i) \leq 0$. Applying the Chernoff bound as in Appendix 5, the probability

$$\Pr\{\sum_{i=q+1}^{p} \ln p(Y_i^0|S_i) \leq \sum_{i=q+1}^{p} \ln p(Y_i^s|S_i) + R\}$$

is upper bounded by

$$e^{\frac{1}{2}R} Z^{p-q},$$

where Z is given in (3.7) of Appendix 5.

There are at most 2^{p-q} possible choices for the path segment

$$\{[K_{q+1},J_{q+1}]^s, \ldots, [K_p,J_p]^s\},$$

so there are at most 2^{p-q} choices for the set of RVs Y^s_{q+1}, ..., Y^s_p. Furthermore, since this path segment must slip by n+1 samples relative to the correct path, one must have $p-q \geq n+1$ or $q \leq p-n-1$. Applying these results and the union bound,

$$P_s(p) \equiv \Pr\{\text{The first frame loss is a frame slip at } [K_p, J_p]\}$$

$$\leq \sum_{q=0}^{p-n-1} 2^{p-q} e^{\frac{1}{2}R} Z^{p-q}.$$

Setting $j = p-q$, the previous sum can be rewritten as

$$e^{\frac{1}{2}R} \sum_{j=n+1}^{p} (2Z)^j$$

$$< e^{\frac{1}{2}R} \sum_{j=n+1}^{\infty} (2Z)^j$$

$$= e^{\frac{1}{2}R}(2Z)^{n+1}/(1-2Z),$$

whenever $2Z < 1$.