Axiomatix

Marina del Rey • California

A PARAMETRIC STUDY OF THE COMPLEXITY OF SEQUENTIAL DECODERS

FINAL REPORT
VOLUME I

Contract No.: NAS 9-12091

Prepared by

Gaylord K. Huth

Axiomatix
13900 Panay Way, Suite 110M
Marina del Rey, California 90291

Prepared for

National Aeronautics and Space Administration
Manned Spacecraft Center
Houston, Texas 77058

Axiomatix Report No. R7201-1
27 January 1972

TABLE OF CONTENTS

LIST OF FIGURES

Figure | Page

VOLUME II

LIST OF TABLES

SECTION 1.0

INTRODUCTION

This study has examined the complexity and performance of practical sequential decoders. Upon comparing sequential decoders with Viterbi decoders, the study found that sequential decoders outperform Viterbi decoders at low data rates (19.2-76.8 kbps) for the same complexity. However, Viterbi decoders are less complex than sequential decoders at high data rates (9.1 Mbps) for the same performance. Section 5.0 presents plots of complexity versus the required signal energy per bit/single-sided noise spectral density (E_b/N_o) to obtain an output probability of error per bit of 10^{-4} for an ideal coherent PSK demodulation. For code rate 1/2 and hard decisions on the demodulated symbols and the same complexity, the sequential decoder requires 1.3 dB less E_b/N_o than the Viterbi decoder at 19.2 kbps data rate. For rate 1/2 and hard decisions, even for the high 9.1 Mbps data rate, the sequential decoder requires 0.5 dB less E_b/N_o than a Viterbi decoder of the same complexity. For rate 1/2 and three-bit quantization on the demodulated symbols, the sequential decoder requires 0.8 dB to 0.65 dB less E_b/N_o than the Viterbi decoder of the same complexity for data rates 19.2 kbps to 76.8 kbps. However, for the 9.1 Mbps data rate, the Viterbi decoder only requires half the complexity to achieve the same performance as the sequential decoder.

Alternately, it is found that, for three-bit quantization, a sequential decoder with code rate 1/2 requires 0.45 dB to 0.3 dB less E_b/N_o than a Viterbi decoder of the same complexity with code rate 1/3 for data rates from 19.2 kbps to 76.8 kbps. To obtain the same performance using a Viterbi decoder with rate 1/3, five to three times the complexity is required for data rates from 19.2 kbps to 76.8 kbps. However, at this complexity, the sequential decoder with code rate 1/3 requires 0.5 dB less E_b/N_o than the Viterbi decoder with code rate 1/3.

To obtain these results, Section 2.0 presents fundamental properties of convolutional codes. Section 3.0 presents properties of the Viterbi decoder, beginning with a description of the decoding algorithm, then finding the complexity of the decoder and comparing it to the performance results obtained by computer simulation. Sequential decoding is discussed in Section 4.0 by first describing the algorithm and then presenting the important performance and design parameters. The complexity of the decoder is determined and compared to computer simulated performance in Section 5.0. Section 6.0 concludes the report with areas of further study which deal with the interfaces between the demodulation and the decoding and between the decoding and reconstruction of the data from data compression techniques.

SECTION 2.0

CONVOLUTIONAL CODE STRUCTURE

In the simplest context, a convolutional code is the output of a finite-state machine consisting of a K-stage shift register (memory length is K) and n linear algebraic functions of the memory K. The input data to the convolutional encoder is normally binary but this is not necessary. The data is shifted into the register ν bits at a time, giving a code rate of ν/n. The codes of most interest are for $\nu = 1$ and most of the discussions will involve these codes. The optimum decoder for convolutional codes is the Viterbi maximum likelihood decoder. The Viterbi decoder is simpler than a corresponding block decoder giving identical performance in terms of probability of error per bit. However, here again, the complexity of the Viterbi decoder increases exponentially with the memory length of the convolutional code, while the probability of error decreases exponentially with the memory length. Sequential decoding of convolutional codes is a sub-optimum decoder. However, its complexity is almost independent of the memory length of the convolutional code. Hence, under certain circumstances, the complexity to achieve a design probability of error per bit is much less using a sequential decoder than a Viterbi maximum likelihood decoder.

The properties of convolutional codes can be presented best by an example. Figure 2.1 illustrates an encoder for a convolutional code

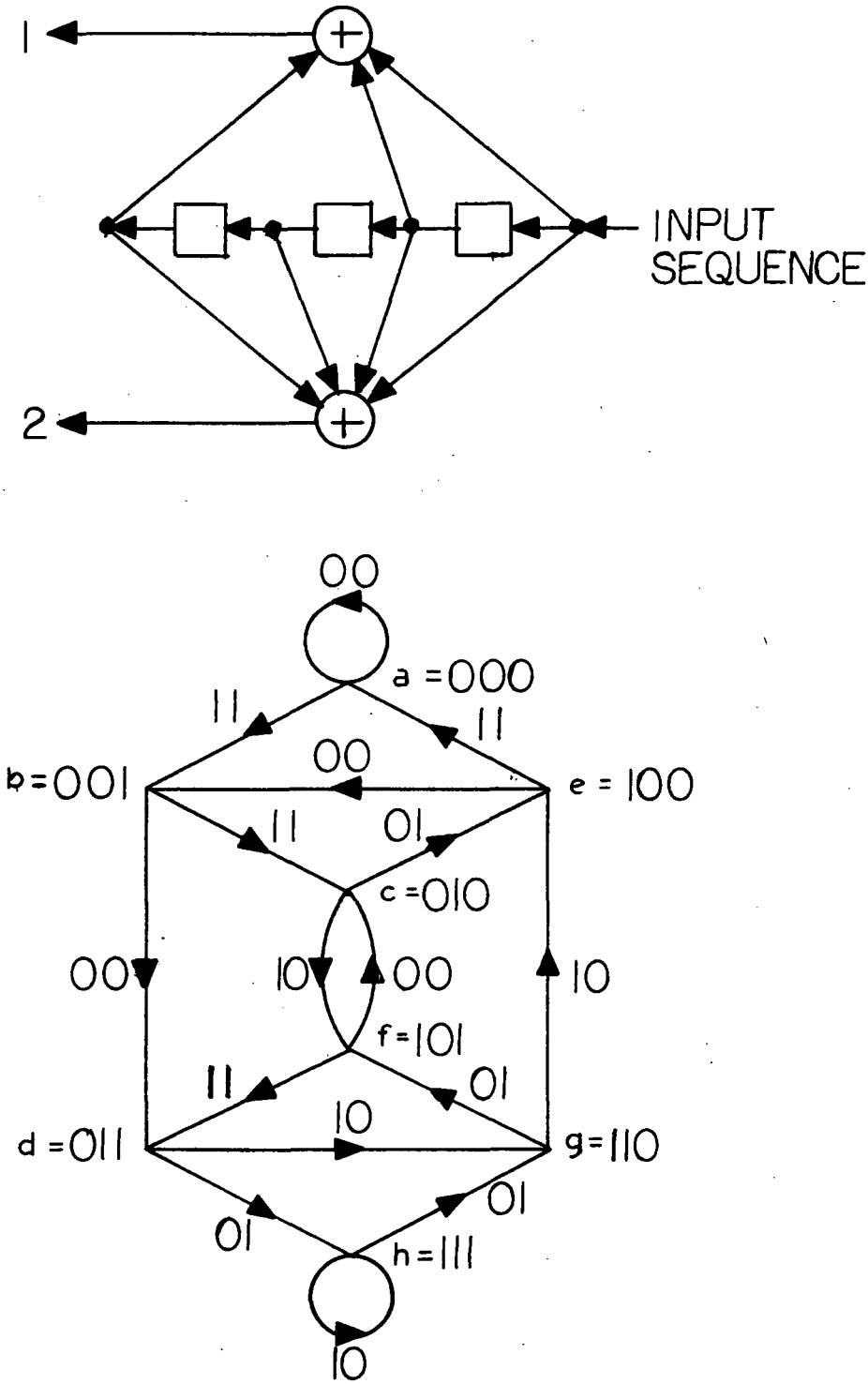


Figure 2.1 Example Convolutional Encoder and State Diagram

of code rate R = 1/2 and memory length K = 3. The summing device is an "exclusive or" (modulo-2 addition) as defined in Table 2.1. Thus, zero plus zero equals zero and one plus one equals zero. Otherwise, the result of addition is one. After each input bit, the output lines

Table 2.1 Modulo-2 Addition

+	0	1
0	0	1
1	1	0

marked 1 and 2 in Figure 2.1 are sampled sequentially and transmitted successively over a single channel or over separate channels. Hence, for each input information bit, two binary symbols are output for transmission over the channel. When the output of the second adder is sampled and transmitted, the content of the encoder is shifted and the input information bit is shifted into the encoder with a new information bit being present on the input sequence line. Hence, the output symbols are a function of the last K information bits and the present input information bit on the input sequence line. If the content of the encoder is defined as the state of the encoder, then there are 2^K possible states, and the input information bit determines the transition between states. For example, if the state of the encoder is 001 and the input information bit is 1, then the output of the encoder is 00 and, after the shift of the encoder,

the new encoder state is 011. Therefore, the 2^K possible states can be represented on a state diagram with the transitions corresponding to an input information bit of 0 or 1. The state diagram corresponding to the encoder is presented in Figure 2.1. The output symbols are placed on the transition corresponding to the input information bit which produces the particular output. Hence, in the example, the transition b = 001 to d = 011 has output symbols 00 corresponding to the input information bit of 1 for this transition. In this sense, a sequence of input bits represents a path in the state diagram, and the symbols on the path represent the output code word transmitted. For example, the input sequence 1001 starting with encoder in the zero state a corresponds to the path from a to b to c to e to b and the output code word of 11110100.

An alternate representation of the code words of a convolutional code is a tree diagram or code tree. The tree diagram of the encoder presented in Figure 2.1 is illustrated in Figure 2.2. In the tree diagram, if the first input bit is 0, then the corresponding output symbols 00 are shown on the first upper branch, while if the first input is 1, then the corresponding output symbols 11 are shown on the first lower branch. Similarly, if the second input bit is 0, the output symbols of the code word correspond to the next upper branch. Hence, each node in the tree represents a state of the encoder with the upper branch emanating from the node corresponding to an input bit of 0 and the lower branch corresponding to an input bit of 1. Therefore, for any input sequence, there

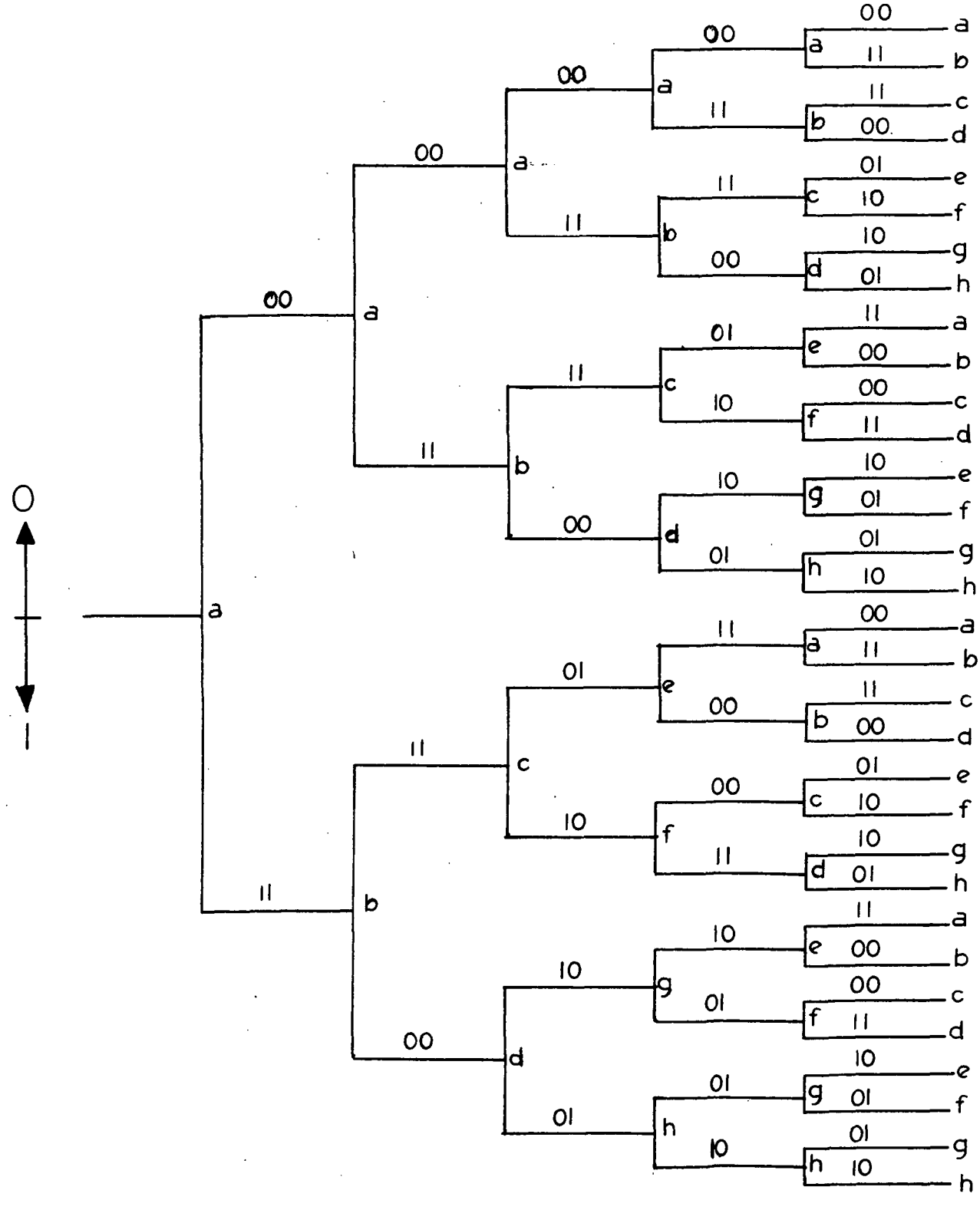


Figure 2.2 Tree Diagram of Convolutional Encoder of Figure 2.1

exists a collection of branches called a path, and the symbols on the branches along the path represent the code word. For the encoder in Figure 2.1, the nodes of the tree diagram in Figure 2.2 are labeled with states of the encoder. By the fourth input bit (K + 1 = 4), all the states appear twice at that level in the tree diagram. In the state diagram, by the fourth input bit, there are two paths that begin at the zero state and end at each state. For example, the input sequences 1011 and 0011 each represent a path that ends at state d = 011. The first input sequence corresponds to the path from a to b to c to f to d, while the second input sequence corresponds to the path from a to a to a to b to d. Furthermore, in the state diagram and the tree diagram, the extension of either of these two paths after the fourth input bit will lead to the same output symbols for a given sequence of input bits of the extension. Therefore, the paths are said to have remerged, which is easily seen in the state diagram. This observation prompted Forney[1] to propose the trellis diagram in which the two identical states of the tree diagram at the K+1 input bit are joined. Figure 2.3 presents the trellis diagram for the example in Figure 2.1. The trellis diagram illustrates the fact that, after the Kth input bit, each state has two paths leading to it.

An important subclass of convolutional codes is called "transparent codes." Transparent codes, in conjunction with differential coding, eliminate the 180 degree phase ambiguity of biphase modulation

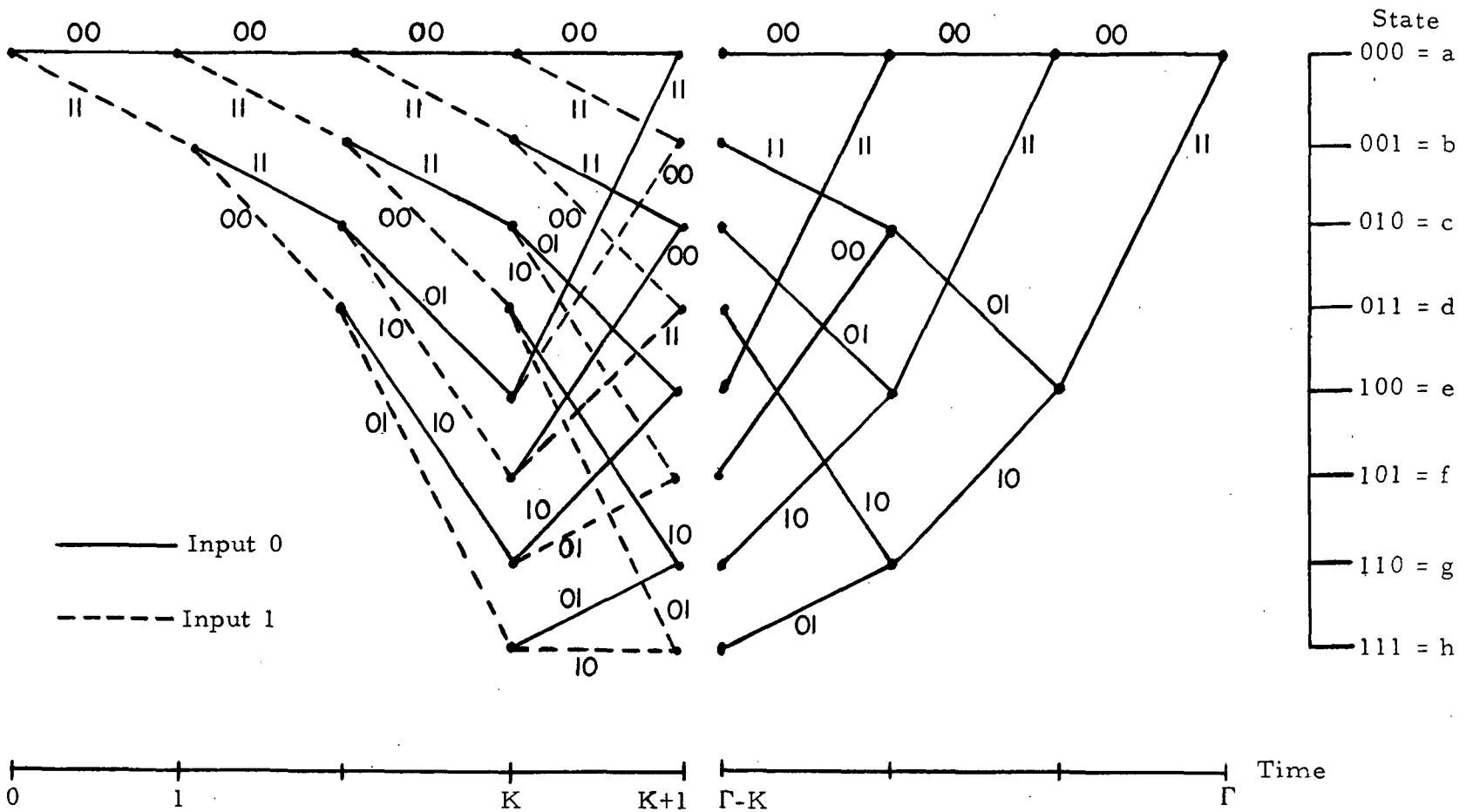


Figure 2.3 Trellis Diagram for a K = 3 Binary Convolutional Code Whose Path Length is Γ Branches

and reduce the fourfold 90 degree phase ambiguity of quadriphase modulation to a 90 degree phase ambiguity. Also, the degradation in performance from ideal using this technique is small, typically about 0.1 dB for sequential decoding and 0.2 dB for Viterbi decoding with memory length K = 3 at output probability of error per bit of 10^{-4}. Thus, because of their importance, the properties of transparent codes are presented.

Differential coding is commonly used to deal with the 180 degree phase ambiguity of PSK demodulation. Let z_i be the ith binary symbol out of the differential encoder and let y_i be the ith binary symbol into the differential encoder; then:

$$z_i = y_i + z_{i-1} \tag{2.1}$$

where the addition is modulo-2. Thus, z_{i-1} is the "reference" for y_i. Let r_i be the received binary symbol; then:

$$r_i = z_i + e_i + \theta_i \tag{2.2}$$

where e_i is the channel noise, equal to one or zero, and θ_i represents the reference phase ambiguity of the demodulator, which is equal to one or zero. The differential decoder output is w_i and is given by:

$$\begin{aligned} w_i &= r_i + r_{i-1} \\ &= (z_i + e_i + \theta_i) + (z_{i-1} + e_{i-1} + \theta_{i-1}) \\ &= y_i + (e_i + e_{i-1}) + (\theta_i + \theta_{i-1}) \end{aligned} \tag{2.3}$$

The output of the differential decoder, w_i, is equal to the corresponding input to the differential encoder, y_i, corrupted by the sum of two noises and the sum of two phase ambiguities. If no phase transition of the

demodulator occurred at time i-1, $\theta_i + \theta_{i-1} = 0$ and the phase ambiguity is eliminated.

Differential encoding does increase the probability of error. In the absence of a phase transition:

$$w_i = y_i + (e_i + e_{i-1}) \tag{2.4}$$

The probability of an error out of the differential decoder, assuming independent channel errors, is:

$$p' = p\left[(e_i + e_{i-1}) = 1\right] = 2p(1-p)$$

$$p' \simeq 2p \tag{2.5}$$

If errors occur in bursts, as is true out of a Viterbi or a sequential decoder, then $e_i + e_{i-1} = 0$ when adjacent symbols are in error, and the probability of error out of the differential decoder is much less than 2p.

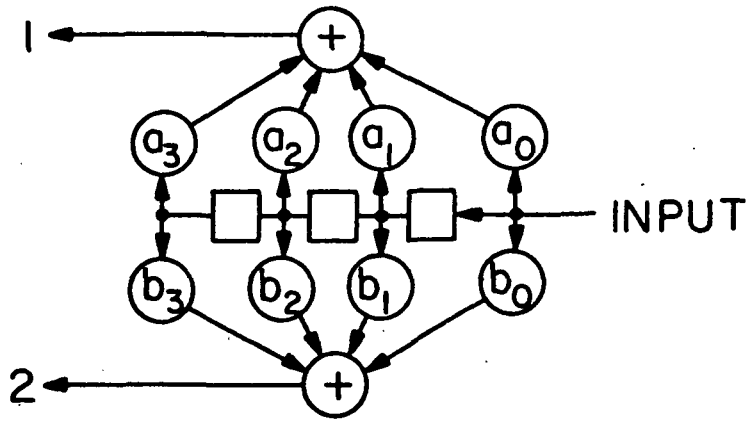
A code is <u>transparent</u> to phase ambiguity if, when $x = (x_1, x_2, \ldots x_j, \ldots)$ is a code word, then its complement $\bar{x} = (x_1+1, x_2+1, \ldots, x_j+1, \ldots)$ is also a code word. For such a code, if the received vector $y = (y_1, y_2, \ldots, y_i, \ldots)$ is decoded into x, then the decoder will decode the complement $\bar{y} = (y_1+1, y_2+1, \ldots, y_i+1, \ldots)$ into $\bar{x}$. Now, if this transparent encoder and decoder are placed between the differential encoder and decoder, then the phase ambiguity θ_{i-1} and θ_i will be passed through the transparent decoder to the differential decoder, whereupon the differential decoder can eliminate the ambiguity. The probability of error out of the differential decoder can, at most, double

the error rate out of the transparent decoder, but this is much less than the channel error rate and little degradation results.

In Figure 2.4, a rate 1/2 feed forward canonical convolutional encoder of K=3 stages and its state diagram are presented as an example. The code symbol, due to the $\underline{a}$ generator assigned to each transition, is the modulo-2 addition of the indicated summations over the a_i's on each transition. Similarly, the other n-1 generators (the b_i's in this case) for a code rate 1/n assign code symbols to the transitions. Hence, each transition has a n-dimensional vector of code symbols assigned to it.

For a code to be transparent, the complement of any code word must also be a code word. Therefore, since the all-zeroes vector is a code word of all codes, then the all-ones vector must be a code word of a transparent code. For the all-ones vector to be a code word, the self loop around the all-ones state must have a one assigned to it by each generator. The symbol assigned to the self loop around the all-ones state is the modulo-2 sum of the taps of the generator, as seen in Figure 2.4. Thus, each of the n generators of a transparent code must have an odd number of taps. Finally, an odd number of taps for all the generators is a sufficient condition for a transparent code, as will be illustrated with an example, since convolutional codes are group codes.

To illustrate the structure of a transparent convolutional code and the transfer function, consider the example with binary input data



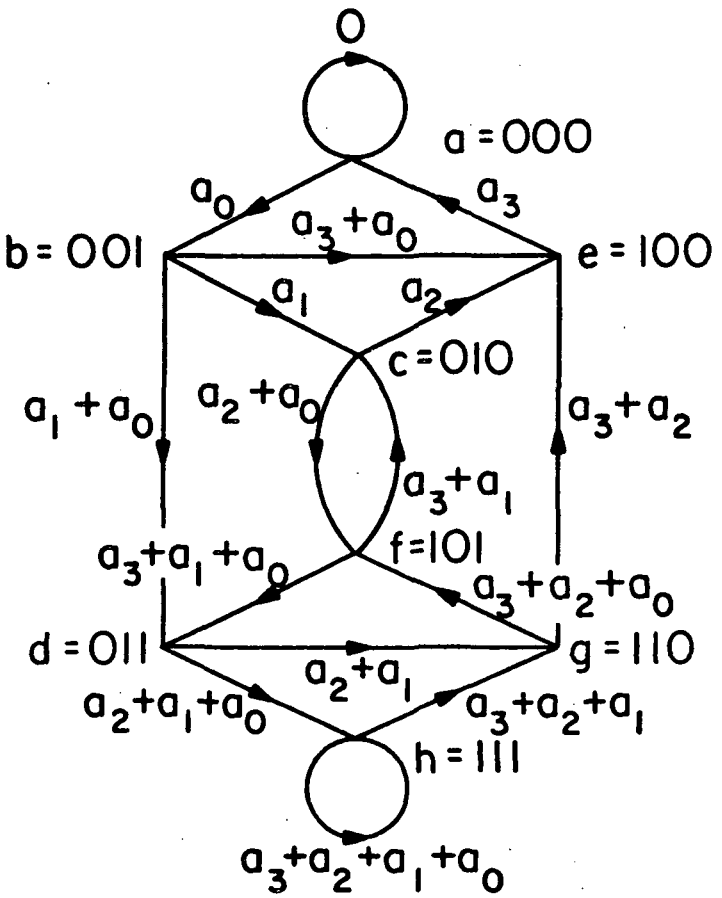
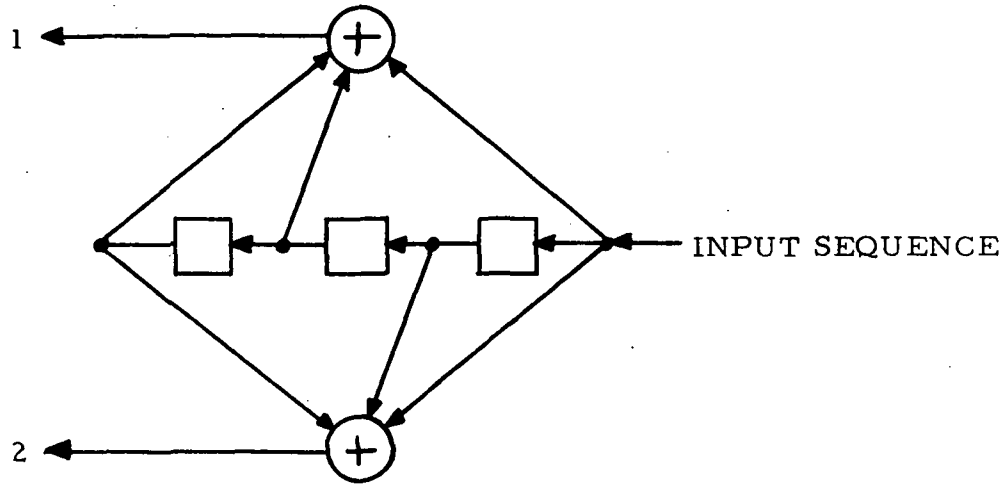


Figure 2.4 **Rate 1/n Convolutional Code Encoder and State Diagram**

K = 3 and n = 2, presented in Figure 2.5. Suppose the all-zeroes code word was transmitted and a phase transition occurs so that the received sequence is ...00 00 00 11 11 11 ..., the closest (least number of disagreements) possible code word is ...00 00 00 11 10 00 11 11 11 ..., which differs from the received sequence in three positions. The input sequence corresponding to this code word is ...000111... and the output of the differential decoder is ...0001000.... Thus, only one error occurs in the data at the output of the differential decoder due to the phase transition. Since convolutional codes are group codes, this is equally true of any other code word. For example, consider the input sequence ...10101010... to the convolutional encoder resulting in the code word ...01 10 01 10 01 10.... Suppose the received sequence was ...01 10 01 01 10 01 10 01 10.... The closest code word is ...01 10 01 01 11 10 10 01 10 ..., which differs in three positions. The input sequence corresponding to this code word is ...101101010... and the output of the differential decoder is ...11101111.... Thus, the output of the differential decoder again only contains a single error due to the phase transition.



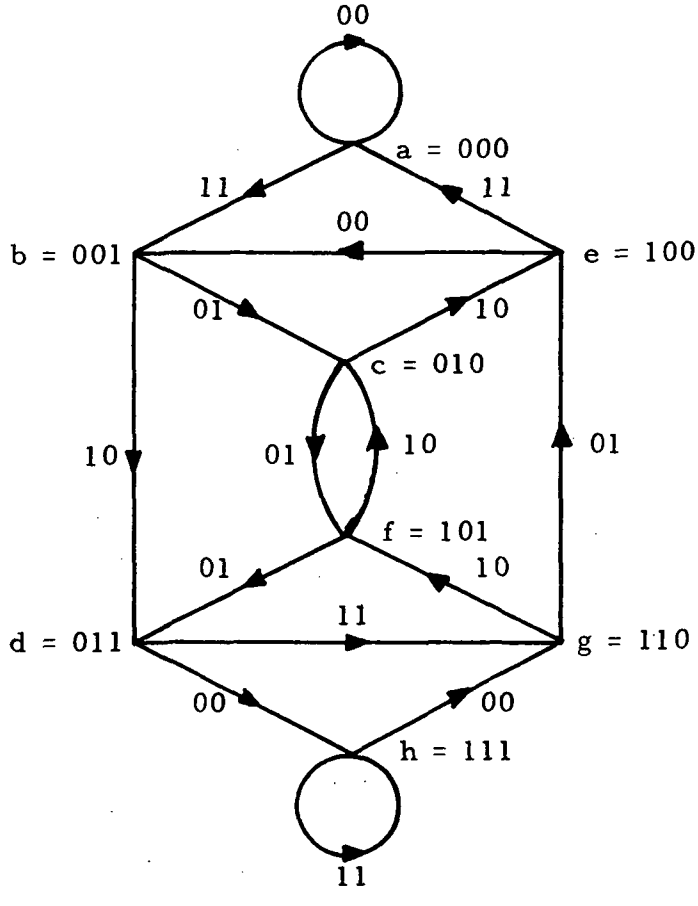


Figure 2.5 Example Encoder and State Diagram

SECTION 3.0

VITERBI MAXIMUM LIKELIHOOD DECODER

A scheme for decoding convolutional codes has been proposed by Viterbi,[2] which was subsequently shown to be synonymous with the maximum likelihood decoding by Forney.[1] If the input data stream consists of a sequence of statistically independent equally likely symbols, then this is the optimal decoding procedure under the criterion of minimization of probability of error over the memoryless channel.

If the transmission channel is memoryless, then the channel errors occur independently from channel symbol to channel symbol. In the case of a binary symmetric channel (BSC) corresponding to hard decision demodulation, a channel error transforms a channel symbol from 0 to 1 or from 1 to 0, and channel errors occur independently from symbol to symbol with probability p if the channel is memoryless. For quantized demodulation of a memoryless channel, the quantized value of the received symbol is independent of other received symbols. In the additive white Gaussian noise (AWGN) channel, the probability of a given quantized value of a received symbol can be computed from the Gaussian probability density.

If the input (information) sequences to the encoding device of any code are equally likely, the decoder which minimizes the overall error probability is one which examines the error-corrupted quantized received

sequence and chooses the data sequence corresponding to the transmitted code sequence which has the greatest probability of being transmitted (i.e., most likely sequence). In the case of the BSC, this corresponds to choosing the data sequence of the transmitted code sequence closest to the received sequence in the sense of Hamming distance, that is, the transmitted sequence which differs from the received sequence in the least number of positions. For quantized demodulation of the AWGN channel, this corresponds to choosing the data sequence $\{x_i\}$ of the transmitted code sequence $\{y_{ij}\}$ which minimizes the interproduct:

$$I = \sum_{i=1}^{\Gamma} \sum_{j=1}^{n} y_{ij} \, r_{ij} \tag{3.1}$$

with respect to the quantized received sequence $\{r_{ij}\}$, where the code rate is $1/n$ and the message length is Γ. The received sequence, r_{ij}, is quantized to Q bits corresponding to numbers 0 to 2^Q-1, such that 0 is a transmitted zero with the highest probability and 2^Q-1 is a transmitted one with the highest probability. The transmitted code symbol, y_{ij}, is represented by Q zeroes for a transmitted zero and Q ones for a transmitted one. In this case, the digital implementation of equation (3.1) is to exclusive OR each of the Q bits of r_{ij} with the Q bits of y_{ij} and to sum the resulting binary numbers to form the interproduct to be minimized. For example, with three bit quantization and the quantized received sequence $\{000, 101, 011, 100\}$, the interproduct for the

transmitted code sequence $\{0, 1, 0, 1\}$ is $\{0, 2, 3, 3\} = 8$, which is the minimum interproduct over all possible four-bit sequences.

To define the maximum likelihood decoder for convolutional codes, the example presented in Figure 2.1 to illustrate the convolutional code structure will be used. Referring to the trellis diagram in Figure 2.3, it is noted that, by the K+1 input bit, there are two paths that begin at the all-zeroes state and end at each of the 2^K possible states. Of these two paths, only the path that minimizes the interproduct in equation (3.1) need be retained. For no matter what the subsequent received symbols may be, they will affect each of these paths in exactly the same way. Therefore, the paths are said to be remerged.

After each received branch of received symbols, comparisons of the two paths remerging at each state are compared to find the path that minimizes the interproduct in equation (3.1). The two possible paths were the survivors at the previous input bit for different states. For example, considering the BSC and the trellis diagram in Figure 2.3, the comparison at state "b" after the fifth received branch is between the survivors at states "a" and "e" after the fourth received branch. After the fourth received branch, if the received sequence is 11 11 10 00, then the survivor at state "a" is the code sequence 11 11 01 11 corresponding to the data sequence 1000 and is at Hamming distance 3 from the received sequence. The survivor at state "e" for this received

sequence is the code sequence 11 00 10 10 corresponding to the data sequence 1100 and is also at Hamming distance 3 from the received sequence. If the next branch of the received sequence is 11 so that the received sequence becomes 11 11 10 00 11 after the fifth received branch, the surviving path from state "a" and the surviving path from state "e" merge at state "b". The survivor after the fifth received branch at state "b" is the path coming from state "a" which remains at Hamming distance 3 from the received sequence. In this way, the decoder may proceed through the received sequence. After each branch has been received, one surviving path and its distance from the received sequence, which is more generally called its metric, is stored. This decision-making process and path elimination procedure is performed for each state after each branch is received. This is the untruncated version of the maximum likelihood (Viterbi) decoding algorithm. If all surviving paths coincide over some branch in the past, then those bits can be output as the final decision of the decoder for that branch.

The only difficulty which may arise is the possibility that, in a given comparison between remerging paths, the distances or metrics are equal. Then a random selection of one of the equal distance paths may be made as is done for block code words at equal distance paths from the received word. If both paths are preserved, further received

symbols will affect both metrics in exactly the same way. Therefore, the random choice would eventually have to be made.

There remains only the question of terminating and truncating the algorithm and ultimately decoding on one path rather than 2^K (equal to 8 in the example). Termination is easily carried out by forcing the last K input bits of the input sequence to be zeroes. Then the final state of the code necessarily is state "a", and consequently the ultimate survivor is the survivor at state "a", after the insertion into the encoder of K dummy zeroes and transmission of the resulting nK code symbols. In terms of the trellis diagram, this means that the number of states is reduced from 2^K to 2^{K-1} by the insertion of the first zero, from 2^{K-1} to 2^{K-2} by the insertion of the second zero, etc., until the Kth zero reduces the number of states from 2 to 1. This is shown for K = 3 in the right side of Figure 2.3.

For each of the 2^K convolutional encoder states, there is a basic storage requirement in the decoder. First of all, each state has associated with it a metric. For a BSC, the metric may simply be the cumulative Hamming distance on the path leading to that state. The question then is how many bits of storage are required for each metric. Since any cumulative metric will eventually overflow the storage provided for it, some mechanism must be provided for re-normalizing the metrics from time to time. Enough metric storage must be provided so that, when the largest metric overflows and is re-normalized, the lowest

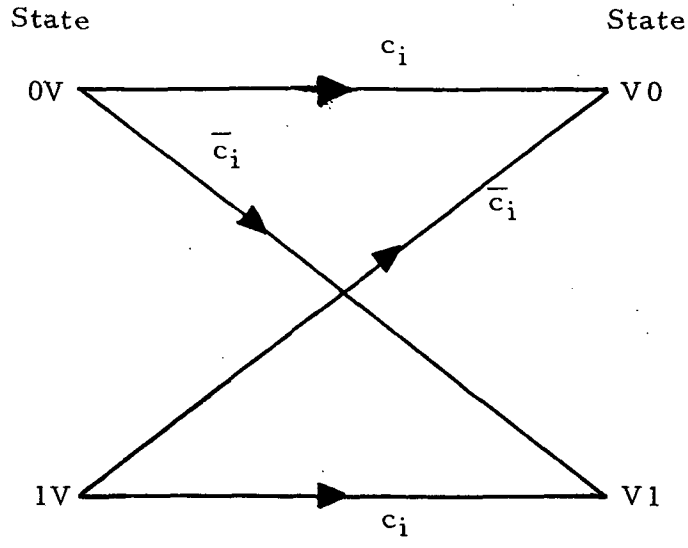
metric should not underflow. The storage required depends strongly on the code used.

In addition to the metric storage, for each coder state, storage must be provided to keep track of the bits on the path leading up to that state. Ideally, in a Viterbi decoder, paths of great length should be stored, enabling the decoder to delay bit decision until all state paths agree on the bit in question. Practically speaking, the shorter the path storage needed, the cheaper the decoder. The number of memory lengths in a path to be stored will be illustrated by computer simulations relating performance to path storage. It is possible that, for the path storage length, not all of the paths will agree on the information bit which the decoder must output. In this case, the bit corresponding to the path with the best metric is output.

3.1 DESCRIPTION OF DECODING ALGORITHM

To implement a Viterbi decoder, first consider the comparisons of survivor paths at each state. It is convenient to perform two state comparisons together because the two states, V0 and V1, both have as predecessor the states 0V and 1V as shown in Figure 3.1. Here, V is any K-1 binary digit sequence. For the case of K = 3, letting V = 10, both states 100 and 101 are preceded by 010 and 110.

The Viterbi decoder algorithm uses five tables in memory. Four of the tables have 2^K entries, one for each state. These four tables are organized such that two tables represent the present condition of



V = any K-1 binary digit sequence

c_i = code symbols corresponding to the state transition

$\overline{c}_i$ = complement of c_i

Figure 3.1 An Example of State Transitions of a Convolutional Code

the decoder, and two tables represent the condition that the decoder will be in after the next received symbols are taken into account. In the following discussion, all arithmetic will be decimal, and X will be a decimal index beginning with one rather than the binary digit sequence V beginning with zero in Figure 3.1. The present condition of the decoder is represented by METRIC(L,X) and PATH(L,X). METRIC(L,X) is the accumulated metric (Hamming distance between hypothesis path and received path) for each state. PATH(L,X) is a table that stores the actual path leading to each state. Thus, PATH(L,X) must have enough bits of storage for each state to accommodate the designed path truncation. The future condition of the decoder is represented by METRIC(I,X) and PATH(I,X). Now, if L = 1, then I = 2 and METRIC(1,X) and PATH(1,X) are present condition and METRIC(2,X) and PATH(2,X) are the future condition. Next, I is set to 1 and L becomes 2, and thus the entries in METRIC(1,X) and PATH(1,X) can be discarded and the tables become storage for the future condition of the decoder. There are 2^{K+1} entries for HYPOTH(S,N) corresponding to the two possible predecessor states for each state. Each entry in HYPOTH(S,N) is the output digits of the encoder for the transition between the two states. Thus, HYPOTH(S,N) is just a table representation of the state diagram. The state entries for the five tables are arranged in numerical state order. The first entry corresponds to state 00...00, the second to state 00...01, the third to state 00...10, etc.

To begin the decoder algorithm, the tables with L = 1 will be the present condition. PATH(L,X) is set to zeroes for each entry, METRIC(L, 1) is set to zero, and the entries of METRIC(L,X) for X not equal to 1 are set to a value about 2K. This condition represents the decoder on the all-zeroes path corresponding to the origin of the tree. The transition from the present condition to the future condition begins by letting X = 1. The incremental metric for c_i in Figure 3.1 is computed by exclusive "OR"-ing (EOR, digital logic for bit by bit modulo-2 addition) the entry in HYPOTH(1, 1) and the received branch symbols. This incremental metric is added to the cumulative metric of state X. Likewise, the incremental metric for $\overline{c}_i$ is added to the metric of state $2^{K-1}+X$. These two sums are compared; the winning metric is stored in METRIC(I, 2X-1). Suppose the path from state X won. The path stored in PATH(L,X) is shifted to the left by one position and stored in PATH(I, 2X-1). If the path from state $2^{K-1}+X$ had won, the path stored in PATH(L, $2^{K-1}+X$) is shifted to the left by one position, a 1 is added in the rightmost position and is stored in PATH(I, 2X-1). The same adding and comparing is done using $\overline{c}_i$ and c_i to get the new entry 2X for the future condition. Now, X is increased by one, and another pair of state comparisons is carried out. By the time X has taken on all of its 2^{K-1} values, each state of the future condition has a new path and metric associated with it. The algorithm next searches for the best path metric in the present condition. The oldest bit on the path associated with this metric is the decoder output. Now the entries

in the tables representing the present condition can be deleted. Then the future condition tables are used for the present condition and a new future condition is computed, storing results in the now empty tables.

A flow chart of a computer program that performs the outlined procedure for Viterbi decoding is presented in Figure 3.2. To start the algorithm, the input is read, L=2, I=1 and several constants that are used many times are defined. These defined constants are $Y = 2^{K-1}$ which, when added to the index X, give the second half of the number of states, and Y1 = Y+1 to be used to indicate when the future condition tables are filled. Also, $Y2 = 2^{K}+1$ is used by the algorithm when the best path is being searched for as an indicator that all the states have been searched, and $YM = 2^{M-1}$ is used to find the oldest bit in the best path to be output. The value M is the number of bits in each path that are stored or the design truncation of the paths. As the algorithm begins actual decoding, X is set to 1, and L and I are interchanged so that the previous future condition becomes the present condition. The next set of calculations is to compute temporary metric values. These temporary values are computed by EOR(HYPOTH(2X-1,N), INPUT) for N = 1, 2 and added to the appropriate stored value of metric for paths coming from X and $2^{K-1}+X$. These temporary metric values are compared for the smallest value to decide which path should be shifted and stored in PATH(I, 2X-1) and which temporary metric should be stored in METRIC(I, 2X-1). Note that, in terms of decimal arithmetic,

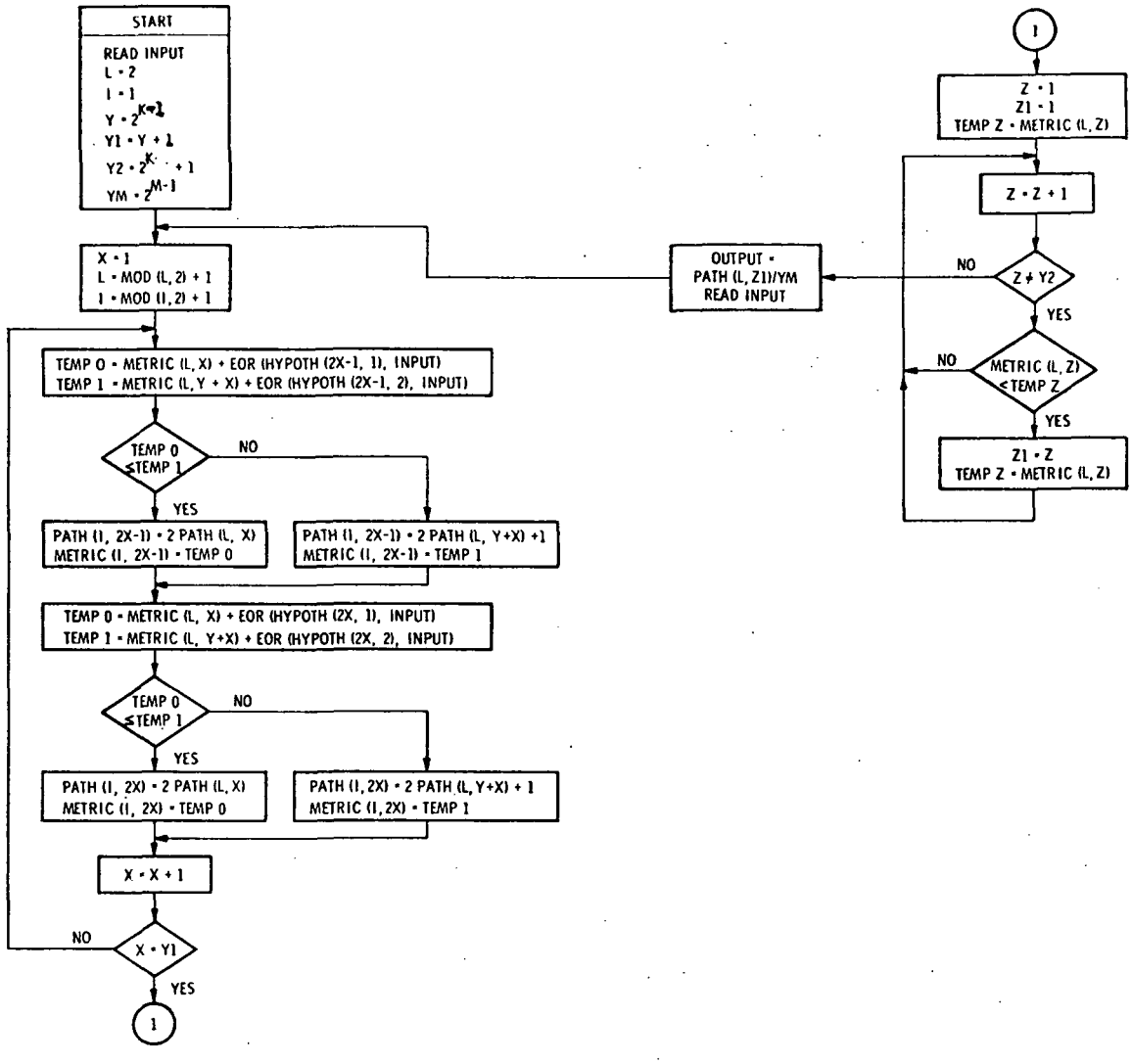


Figure 3.2 Flow Chart of Viterbi Decoder Algorithm

the path is shifted to the left by multiplying by 2, and if TEMP1 is the winning temporary metric value, a 1 is added to the shifted path. This procedure is then repeated for entry 2X in the future condition tables. Finally, X is updated by 1 and the procedures are repeated for the new value of X. If X is equal to $2^{K-1}+1$, then the future condition tables are filled and the path associated with the best metric of the present condition is to be found. When this path is found, it is divided by 2^{M-1} which shifts the path to the right, leaving only the oldest bit in the computer word OUTPUT.

To illustrate the changes in the tables as the received data is processed, an example is presented. Figure 2.3 illustrated the trellis diagram for the example to be used. Figure 3.3 presents the table representation of the state diagram. Note that for X = 1 corresponding to state "a", entry N = 1 is the transition digits coming from X (state "a") and entry N = 2 is the transition digits coming from state $2^{K-1}+X$ (state "e"). To begin processing, Figure 3.4 illustrates the present condition in tables METRIC(1,X) and PATH(1,X) and the resulting future condition after processing one input bit (one received branch). At the start, METRIC(1,1) is set to zero corresponding to the decoder considering the all-zeroes path as the best path (the origin of the tree). METRIC(1,X) for all $X \neq 1$ is set to 6; this number is arbitrary but should be about 2K. PATH(1,X) for all X is set to zero. Referring to the trellis diagram in Figure 2.3, with the received digits (11), the best

	X	N	HYPOTH (X, N)
a	1	1	00
	1	2	11
b	2	1	11
	2	2	00
c	3	1	11
	3	2	00
d	4	1	00
	4	2	11
e	5	1	01
	5	2	10
f	6	1	10
	6	2	01
g	7	1	10
	7	2	01
h	8	1	01
	8	2	10

Figure 3.3 State Transition Table for Example

X	METRIC(1, X)	PATH(1, X)
a 1	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
b 2	6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c 3	6	0 0 0 0 0 0 0 0 0 **0** 0 0 0 0 0 0
d 4	6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
e 5	6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
f 6	6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
g 7	6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
h 8	6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

X	METRIC(2, X)	PATH(2, X)
a 1	2	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
b 2	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
c 3	6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
d 4	6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
e 5	6	0 **0** 0 0 0 0 0 0 0 0 0 0 0 0 0
f 6	6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
g 7	6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
h 8	6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
RECEIVED		00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 11

Figure 3.4 Present and Future Conditions at Startup.

path to state 1("a") is from state 1("a"), giving a metric value of 2. The best path to state 2("b") is from state 1("a"), giving a metric value of 0. In state 4("d"), the best path leading to this state is from state 6("f"), giving a metric value of 6. Since the best path leading to state 4("d") is from state 6("f") or $2^{K-1}+X$, a 1 is shifted into PATH(2, 4). This procedure is continued for all the states filling METRIC(2, X) and PATH(2, X). The surviving paths in the trellis diagram after one received branch are presented in Figure 3.5. In this diagram, it is seen that all paths except the path ending at state 4("d") have a preceding state that has an index of four or less. The path ending on state 4 has state 6 as its previous state (or $2^{K-1}+X$). All paths originate at the all-zeroes state beginning with K+1 previous branches corresponding to the startup conditions.

The output digit is a zero since the present condition path with the smallest metric has zero as the oldest bit (leftmost). However, this output bit corresponds to the startup conditions. The first bit corresponding to the received digits will be output after M+K bits have been received.

Figure 3.6 illustrates the present and future conditions after three input bits have been received. Note that METRIC(1, X) and PATH(1, X) are obtained by processing METRIC(2, X) and PATH(2, X) of Figure 3.4. The trellis diagram of Figure 3.7 shows the surviving paths after the METRIC(1, X) and PATH(1, X) are obtained. It should be noted that path B is closer (fewer differences) to the received path. However, by the

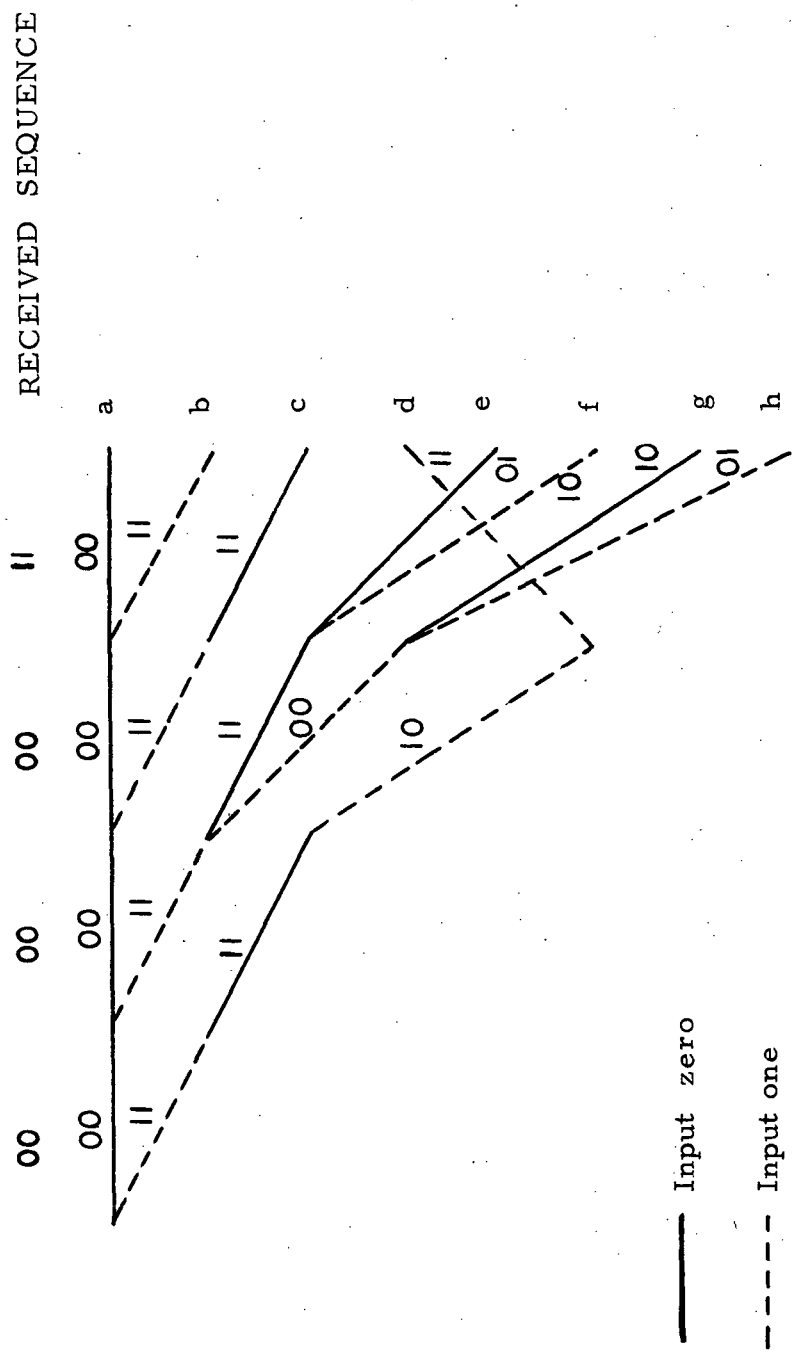


Figure 3.5 Trellis Diagram After One Received Branch

X	METRIC(1, X)	PATH(1, X)															
a 1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b 2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
c 3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d 4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e 5	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f 6	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
g 7	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
h 8	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
RECEIVED		00	00	00	00	00	00	00	00	00	00	00	00	00	00	11	00

X	METRIC(2, X)	PATH(2, X)															
a 1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b 2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
c 3	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d 4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e 5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f 6	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
g 7	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h 8	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RECEIVED		00	00	00	00	00	00	00	00	00	00	00	00	00	11	00	00

Figure 3.6 Present and Future Conditions After Three Input Bits

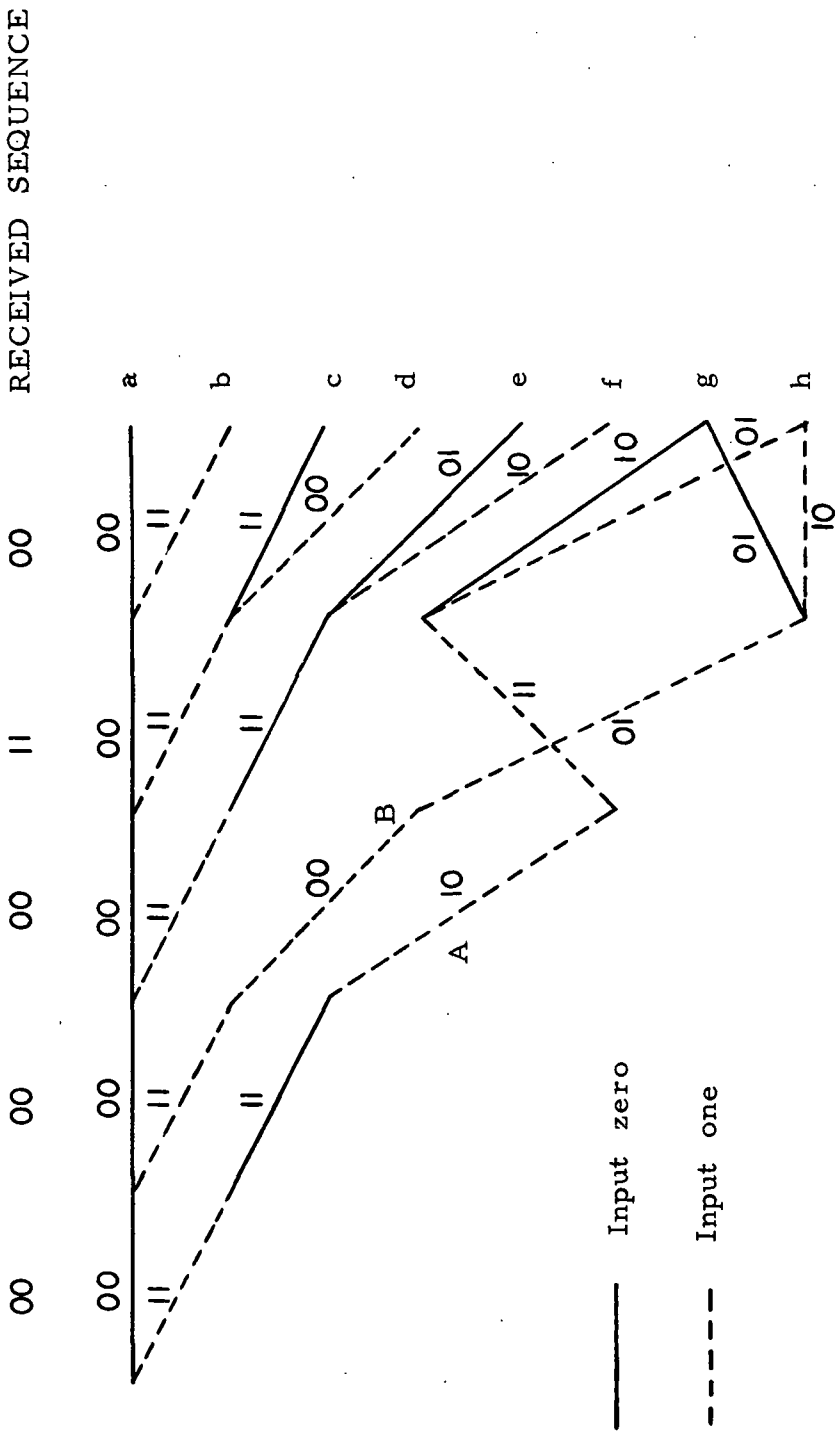


Figure 3.7 Trellis Diagram After Two Received Branches

algorithm convention, in case of equal metrics for the two paths, the path emanating from the state with the smallest index is chosen. Thus, path A is chosen to states g and h, due to the startup metric weights assigned to each state. In this case, all the paths originate at the all-zeroes state beginning with K+2 previous branches corresponding to the startup conditions.

The path A to state g and h, present after only two branches had been received, is discarded after three branches have been received. Hence, when the future conditions tables in Figure 3.6 are filled, the state of the metrics has now progressed completely from their initial values and represent the metrics of the surviving paths corresponding to just the received digits.

Figure 3.8 represents the present and future conditions after M = 16 input bits. Note in the present condition tables METRIC(1,X) and PATH(1,X), states 7 and 8 have paths containing a 1 in the position corresponding to the first digit of the received data to be decoded. Also, these states have a metric very close to the correct all-zeroes path. In the trellis diagram shown in Figure 3.9, it is seen that this path originates at the all-zeroes state beginning with 16 previous branches. Thus, the occurrence of surviving paths of this length illustrates the need for a large path memory. Simulations indicate that M = 4K is necessary to make the effects of long surviving paths negligible in comparison to the error correction capability of the code in conjunction with

X	METRIC(1, X)	PATH(1, X)															
a 1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b 2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
c 3	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
d 4	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
e 5	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
f 6	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
g 7	6	0	0	0	1	1	0	1	0	1	0	1	0	0	1	0	0
h 8	6	0	0	0	1	1	0	1	0	1	0	1	0	0	1	0	0
RECEIVED		11	00	00	00	00	10	00	00	00	00	00	11	00	00	00	00

X	METRIC(2, X)	PATH(2, X)															
a 1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b 2	6	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
c 3	6	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
d 4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
e 5	6	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
f 6	6	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
g 7	7	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
h 8	7	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
RECEIVED		00	00	00	00	10	00	00	00	00	00	11	00	00	00	00	00

Figure 3.8 Present and Future Conditions after M=16 Input Bits

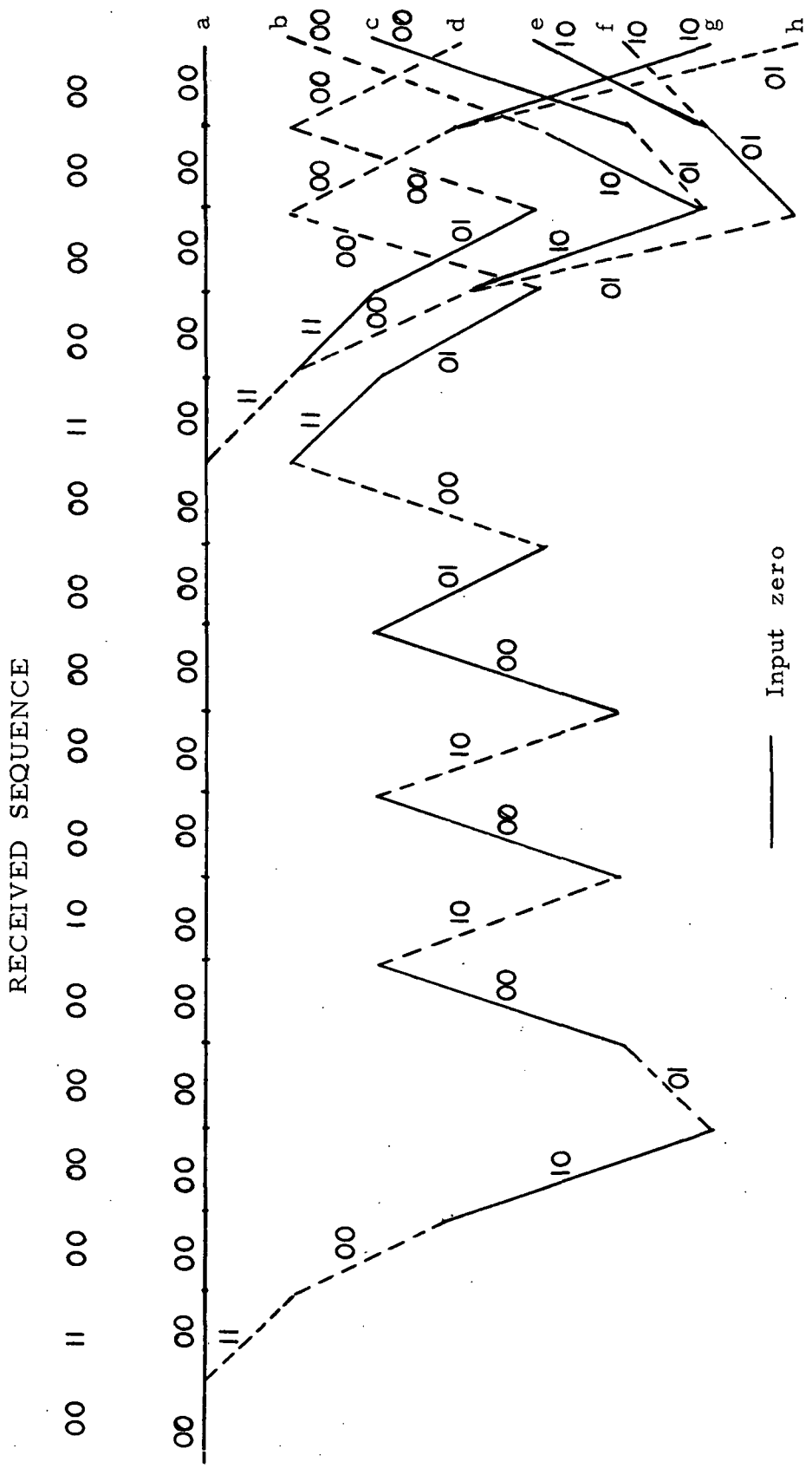


Figure 3.9. Trellis Diagram After 16 Received Branches

a decoder with infinite path memory. As may be noted, after processing one more input bit, all the paths in the future condition have zero as the first bit to be output corresponding to the received sequence. Thus, the algorithm will decode the first bit of the all-zeroes path as the correct path.

3.2 PERFORMANCE OF A TRANSPARENT CODE VERSUS A NONTRANSPARENT CODE

The performance of Viterbi decoding can be tightly bounded from the transfer function of the convolutional code used to encode the data with a technique developed by Viterbi.[3] The transfer function T(L,N,D) of a convolutional code relates the Hamming distance between code words, the number of input ones corresponding to each code word, and the length of each code word. The distance structure, the Hamming distance between code words, is found by considering the Hamming weights of the code words (i.e., the number of output ones in a code word) since convolutional code words are group codes. Alternately, the state $\underline{a}$ = 000 is split and its self loop is eliminated from the state diagram. On each transition between the states, let the exponent of a dummy variable D correspond to the number of ones in the symbols on the transition, assign a dummy variable L to the transition to indicate that the length of the path between the two states is one, and finally let the exponent of a dummy variable N correspond to whether the input bit for the transition was a zero or one. For example, the transition from state $\underline{b}$ = 001 to state $\underline{c}$ = 010 of the transparent code in Figure 2.4 would be LN^0D or

LD^2, and the transition from state $\underline{d}$ = 011 to state $\underline{h}$ = 111 would be LND^0 or LN. The resulting diagram is the modified state diagram of Figure 3.10.

By considering the modified state diagram as a signal flow graph, the closed form transfer function is obtained as shown in Appendix I. For biphase modulation with additive white Gaussian noise, assuming ideal coherent demodulation, Viterbi has shown that the probability of error per bit of the decoder is bounded by:

$$P_b < \text{erfc}\sqrt{\frac{2E_s d}{N_o}} \exp\left(\frac{E_s d}{N_o}\right) \left.\frac{dT(L,N,D)}{dN}\right|_{L=1,N=1,D=\exp(-E_s/N_o)} \tag{3.2}$$

where d is the weight of the smallest weight path, called free distance, and E_s/N_o is the signal energy per symbol/single-sided noise spectral density. The value of d for the transparent code in Figure 2.4 was 6. The derivative with respect to N of the transfer function of this code is:

$$\left.\frac{dT(L,N,D)}{dN}\right|_{L=1,N=1} = T'(D) = \frac{D^6(4-2D^2-3D^4+2D^6)}{(1-5D^2+2D^6)^2} \tag{3.3}$$

as seen from the transfer function in Appendix I. For E_s/N_o = 3dB(E_b/N_o = 6dB), the probability of error per bit of the transparent code in the example is bounded by $P_b < 2.3\text{x}10^{-6}$.

As a comparison of the performance of the transparent code in the example to a good nontransparent code, consider the best code for K=3, as shown in Figure 2.1. In this case, the derivative with respect

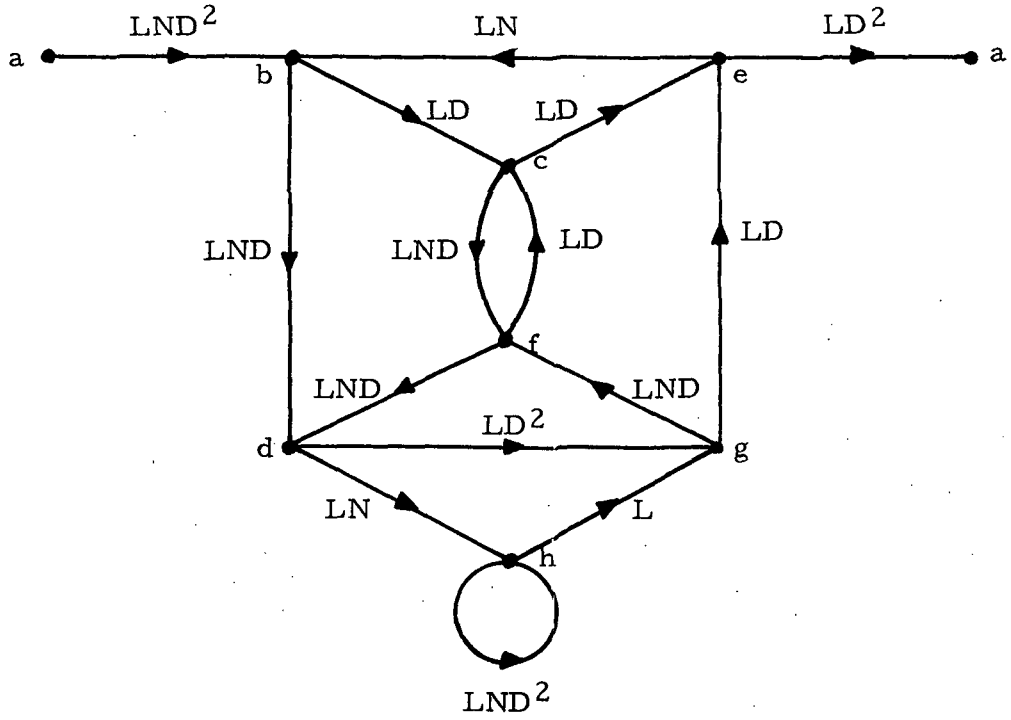


Figure 3.10 Modified State Diagram for the Example Convolutional Code

to N of the transfer function for this code is:

$$T'(D) = \frac{D^6(2-D-2D^2+D^3+D^5)}{(1-2D-D^3)^2} \qquad (3.4)$$

The value of free distance d is also 6 for this code. For E_s/N_o = 3 dB, the probability of error per bit of this nontransparent code is bounded by $P_b < 1.66 \times 10^{-6}$. Thus, the error rate of the transparent code is about 1.4 times greater than the error rate of the best nontransparent code. Simulations have shown that the increase in error rate using differential coding is about 1.2. Therefore, the error rate for the transparent code in the example, in conjunction with differential coding, would be about 1.7 times the error rate of the best nontransparent code at K=3 and no phase ambiguity.

The degradation from ideal in E_b/N_o is about 0.2 dB at an output probability of error per bit of 10^{-4} for the K=3 transparent code with differential coding. Because of the simplicity of resolving the phase ambiguity, the small degradation in performance makes the combination of a transparent code and differential coding a very attractive technique.

3.3 COMPLEXITY OF THE VITERBI DECODER

The complexity of a hardware implementation of the Viterbi maximum likelihood decoder will be measured in complexity bits. A complexity bit is defined as a bit of storage or a latch, a bit in an addition or comparison, or a switch. Using this complexity measure, Viterbi decoders can be compared for performance versus complexity

for various values of memory length K and code rate R with sequential decoders. Also, each portion of the Viterbi decoder is described separately so that an average cost per complexity bit may be established. As the price of devices changes, a new average cost per complexity bit can be computed giving the new cost. New MSI circuit arrangements may change the physical number of integrated circuits (IC) necessary to implement each decoder portion, but the number of the complexity bits will remain the same. To provide insight into the physical size of the Viterbi decoder, the number of IC's used by LINKABIT[4] in their design of a 10 Mbps TTL Viterbi decoder for memory length K = 3 (K = 4, using the LINKABIT notation of constraint length) will be included.

3.3.1 Branch Metric Computation

To implement the hardware to compute the branch metrics EOR (HYPOTH, INPUT) in the flow diagram of Figure 3.2, first note that, in Figure 3.1, the c_i and $\overline{c}_i$ (complement c_i) represent the code symbols corresponding to the transition. For rate 1/2, c_i can take on the values 00, 01, 10, and 11; for rate 1/3, c_i can take on the eight possible values of a three-bit binary number. Thus, there are 2^n branch metrics to be computed from the received code symbols for code rate 1/n. If each received code symbol is quantized to Q bits, then $2^n \lceil \log_2 n(2^Q-1) \rceil$ bits would be necessary to represent the branch metrics, where $\lceil x \rceil$ is the least integer greater than or equal to x.

If, for code rate 1/2, r_1 and r_2 represent the received code symbols quantized to Q bits, then they may be labeled from 0 to 2^Q-1. Denoting the branch metrics as m_{00}, m_{01}, m_{10}, and m_{11} for the four possible metrics, then:

$$\begin{aligned} m_{00} &= r_1 + r_2 \\ m_{01} &= r_1 + 2^Q-1 - r_2 \\ m_{10} &= 2^Q-1 - r_1 + r_2 \\ m_{11} &= 2(2^Q-1) - r_1 - r_2 \end{aligned} \tag{3.5}$$

Thus, for rate 1/2, there must be four adders of $\lceil \log_2 n(2^Q-1) \rceil$ bits each. For rate 1/2, there is only one addition to be performed by each adder. Using TTL, the computation of the branch metrics can be performed in 24 nanoseconds for Q = 3. For rate 1/3, there must be eight adders. In this case, first the sum of two of the received quantized symbols is added and the third symbol is added to the result. The computation of the branch metrics can be performed in 60 nanoseconds for Q = 3 using TTL. However, for a serial bit stream and low data rate, the first addition can be performed while the third symbol is being received, and only 30 nanoseconds are necessary to compute the branch metrics after the third symbol is received.

The total complexity of the branch metric processor when the branch metrics are computed for each received branch of coded digits is:

$$\text{branch metric processor} = 2^{n+1}\lceil \log_2 n(2^Q-1) \rceil + nQ \text{ complexity bits} \tag{3.6}$$

The term nQ is the number of bits necessary to store the symbols of the received branch as it is received. In addition, there are $2^n \lceil \log_2 n(2^Q-1) \rceil$ complexity bits associated with the 2^n adders and also the same number of complexity bits to store the resulting branch metrics to be accessed by the arithmetic unit.

For low data rates, the complexity of the branch metric processor can be further reduced by using a single adder time-shared to compute the 2^n branch metrics. With the low data rates to be considered (19.2 kbps, 38.4 kbps, 57.6 kbps, and 76.8 kbps), the use of a single adder does not increase the number of processors in the arithmetic unit except for code rate 1/3, memory length K = 7 or 8, and data rate 76.8 kbps. In this latter case, the number of processors in the arithmetic unit does not increase if four adders are time-shared to compute the branch metrics. Using this time-sharing technique, the total complexity of the branch metric processor is:

$$\text{branch metric processor} = 12\lceil \log_2 3(2^Q-1) \rceil + 3Q \text{ complexity bits} \tag{3.7}$$

for code rate 1/3, K = 7 or 8, and data rate 76.8 kbps; otherwise:

$$\text{branch metric processor} = (2^n+1)\lceil \log_2 n(2^Q-1) \rceil + nQ \text{ complexity bits} \tag{3.8}$$

For high data rate Viterbi decoders, it is not feasible to calculate the branch metrics for each received branch of coded digits. Therefore, a read-only memory (ROM) is used to store the branch

metrics for the 2^{nQ} possible quantized branch symbols. If a ROM is used, then a metric compression technique is possible. As may be observed from the flow diagram in Figure 3.2, subtracting the same constant from all the branch metrics does not affect the decoder decisions. Therefore, if the smallest branch metric in equation (3.5) is subtracted from all the branch metrics, then one branch metric will be zero and all the others non-negative. If m_{00} is the smallest metric, for example, then:

$$\begin{aligned} m_{00} &= 0 \\ m_{01} &= 2^{Q}-1 - 2r_2 \\ m_{10} &= 2^{Q}-1 - 2r_1 \\ m_{11} &= 2(2^{Q}-1) \end{aligned} \tag{3.9}$$

Now, in equation (3.9), the branch metrics m_{01} and m_{10} will be odd, m_{00} and m_{11} are even. Therefore, for Q greater than one, if the two central quantization intervals are considered only one quantization interval, then the received code symbols can be labeled from 0 to $2^{Q}-2$. In this case, the term $2^{Q}-1$ in equation (3.9) is replaced with $2^{Q}-2$, and all the branch metrics are even. Therefore, the least significant bit may be discarded, decreasing the number of bits needed to represent the branch metrics. Although only using one central quantization interval instead of two produces some degradation, it is quite small. This metric compression technique is also applicable to rate 1/3 as is easily shown. The complexity of the branch metric processor

using the ROM is:

$$\text{branch metric processor} = 2^n(2^{nQ}+1)\underbrace{\left\lceil \log_2 n(2^{Q-1}-1)\right\rceil}_{\text{complexity bits}} + nQ \qquad (3.10)$$

The number of bits to represent each branch metric is $\left\lceil \log_2 n(2^{Q-1}-1)\right\rceil$ since the least significant bit is discarded. The ROM has 2^{nQ} words with $2^n\left\lceil \log_2 n(2^{Q-1}-1)\right\rceil$ bits each to represent the 2^n branch metrics. The term nQ represents the storage for the quantized symbols as they are received, and the additional $2^n\left\lceil \log_2 n(2^{Q-1}-1)\right\rceil$ bits represent the storage of the branch metrics output of the ROM for use by the arithmetic unit.

3.3.2 Arithmetic Unit Complexity

The number of bits needed to represent the 2^K state metrics, $S_1, S_2, \ldots S_{2K}$, must be minimized for the minimum complexity of the decoder. To minimize this number of bits, note that the minimum Hamming distance along all paths from the all-zeroes state to any other state may be calculated from the trellis or state diagram. The maximum of these Hamming distances must be less than or equal to nK since any state can be reached by K or less transitions, and each transition can have at most n ones associated with it. The actual value of the maximum of these Hamming distances is quite variable over all possible codes. Since the size of this value contributes to the decoder complexity, it is desirable to find good codes in terms of performance for which this value is small. To compute the complexity for a number of memory lengths, the value of nK is used. However,

for a particular memory length, an investigation should be performed to compare the various codes for complexity of the decoder versus performance.

Using the value nK, the most any state can differ from any other state in Hamming distance is nK, since the convolutional codes are group codes. For Q bits of quantization, then, the most any state metric can differ from any other state metric is $(2^Q-1)nK$. Using the metric compression technique on the branch metrics, the most any state metric can differ is $(2^{Q-1}-1)nK$ for Q greater than one.

Another important observation is the maximum increase of the smallest state metric. Consider Figure 3.1; if the state metric associated with the state 0V is zero, the state metric associated with state 1V is y, and the branch metrics are x and $n(2^Q-1)-x$, where x takes on integer values between 0 and $n(2^Q-1)$, then Table 3.1 describes the state metrics of states V0 and V1 for n = 2 and Q = 3.

Table 3.1 State Metrics for States V0 and V1

y		6														
x		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
State	V0	0	1	2	3	4	5	6	7	8	9	10	9	8	7	6
Metrics	V1	6	7	8	9	10	9	8	7	6	5	4	3	2	1	0

In this case, the minimum between the two state metrics is always less than or equal to 7. Therefore, for n = 2 and Q = 3, the maximum

the smallest metric can increase is 7. By similar observations, it is seen that, for n = 2 and Q = 1, the maximum increase of the smallest metric is 1; for n = 3 and Q = 3, it is equal to 11. For the metric compression technique, x can only take on values between 0 and $n(2^{Q-1}-1)$ and the branch metrics are x and $n(2^{Q-1}-1)-x$. Therefore, it can be similarly shown that the maximum the smallest metric could increase is 3 for rate 1/2 and Q = 3. For rate 1/3 and Q = 3, the smallest metric could increase by a maximum of 4. Hence, the state metrics can be normalized such that the smallest metric is less than or equal to the maximum the smallest metric can increase after each received branch. The number of bits needed to represent each state metric is:

$$\text{storage bits per state metric} = \left\lceil \log_2(nK(2^{Q}-1)+2S_m) \right\rceil \qquad (3.11)$$

with the 2^Q replaced by 2^{Q-1} if the metric compression technique is used, and where S_m is the maximum value of the smallest normalized metric.

There are 2^K state metrics that must be stored. If all operations are performed in parallel, then each state metric must be simultaneously accessible, and when all the additions and comparisons are performed at the end of the arithmetic unit cycles, the state metrics are replaced with their new values. Hence, for a fully parallel processor, 2^K state metrics must be stored. For low data rates, where a completely serial processor may be used, only 2^K state metrics must be stored, but an efficient storage organization is required. As was observed in Figure 3.1,

the state metrics of states V0 and V1 resulted from adding the branch metrics to the state metrics of states 0V and 1V. An efficient storage organization is proposed by LINKABIT[4], which is to store state metrics in two sections, one section for those states that have even parity and another section for those states that have odd parity. For example, if V has even parity, then states 0V and V0 have even parity and states 1V and V1 have odd parity. In performing a computation, the metrics of states 0V and 1V are read out of storage, the computation is performed, and the resulting metrics V0 and V1 are written into the storage locations from which the original metrics were read. The state metrics in this organization will be found in different locations at each computation. However, the address of the locations can be computed simply by a properly programmed K bit counter. When neither a fully parallel processor or a fully serial processor is used, then additional temporary storage is required for the state metrics. If more than two arithmetic units are time-shared, then 2^K state metrics must have temporary storage or the storage requirement doubles. Therefore, the required storage for the state metrics is:

$$\begin{aligned}
&\text{parallel state metric storage bits} && = 2^K \left\lceil \log_2(nK(2^Q-1) + 2S_m) \right\rceil \\
&\text{serial state metric storage bits} && = 2^K \left\lceil \log_2(nK(2^Q-1) + 2S_m) \right\rceil \\
&\text{time-shared state metric storage bits} && = 2^{K+1} \left\lceil \log_2(nK(2^Q-1) + 2S_m) \right\rceil
\end{aligned} \tag{3.12}$$

When the metric compression technique is used, the 2^Q in equation (3.12) is replaced by 2^{Q-1}.

The arithmetic unit performs the addition between the state metrics 0V and 1V with the appropriate branch metrics to determine the new state metric and path memory for first V0 and then V1 as shown in the flow diagram of Figure 3.2. In addition, the arithmetic unit determines when normalization is necessary. When normalization is necessary (i.e., when the smallest metric is greater than or equal to S_m after the computations are performed), the value of S_m is subtracted from the branch metrics that are to be used for the next received branch. The subtraction is quite simple and may be hard-wired. Normalization can result in negative branch metrics. Therefore, an extra bit is provided as a result of the subtraction to represent the sign bit. The total number of bits to be subtracted is 2^n for $n = 2, 3$ and $Q = 1$, and is $n2^n$ for $n = 2, 3$ and $Q = 3$ if S_m is chosen to be a power of two, which reduces the complexity of the subtraction and the complexity of determining the need for normalization. To determine when normalization is necessary, a check is made as to whether the metric is greater than S_m each time a new state metric is computed. If the new state metric is greater than S_m, then the normalization is not inhibited. However, as soon as some new state metric is less than or equal to S_m, then the normalization is inhibited. The comparison requires an OR-ing of the most significant bits, and the normalization inhibit requires a bit of storage for a total of two complexity bits per arithmetic unit. This normalization procedure is equally valid for for a fully parallel processor and a fully serial processor.

The inputs to the arithmetic unit are two state metrics and two branch metrics. The appropriate branch metrics are added to the state metrics, and the results are subtracted to determine which result is smaller. The smaller result is the new state metric. The complexity of each processor of the arithmetic unit is $5\left\lceil \log_2(nK(2^Q-1) + 2S_m)\right\rceil$ bits that are added, subtracted, switched to the output, or stored for transfer to the state metric storage. Thus, the complexity of each processor of the arithmetic unit is:

$$\text{Processor} = 5\left\lceil \log_2(nK(2^Q-1) + 2S_m)\right\rceil + 2 \text{ complexity bits} \quad (3.13)$$

where 2^Q is replaced by $2^Q - 1$ if the metric compression technique is employed.

For the LINKABIT 10 Mbps TTL decoder with K = 3, Q = 3, and n = 2, each processor of the arithmetic unit can be implemented with 10 integrated circuits. It should be noted that the three IC's used for adding the branch metrics to the state metrics and subtracting the results for comparison are large and relatively expensive.

Including the propagation delay of the state metric storage, an arithmetic operation can be performed in under 100 nanoseconds for the range of memory lengths K = 3 to 8 using TTL. Thus, for the 9.1 Mbps data rate, a fully parallel arithmetic unit is necessary or 2^K processors. In this case, the complexity is:

$$\text{Arithmetic unit for 9.1 Mbps} = 2^K(\underbrace{5\left\lceil \log_2(nK(2^Q-1)+2S_m)\right\rceil}_{\text{complexity bits}} + 2) \quad (3.14)$$

For the 19.2 kbps data rate, a fully serial processor is possible with memory length K = 3 to 8. The branch metrics are switched to the arithmetic unit by $2^n \lceil \log_2 n(2^Q-1) \rceil$ switches.

The switching of the branch metrics is determined by a convolutional encoder counting through the states of complexity n(K+1) complexity bits. The addressing of the state metric storage requires a counter of K complexity bits. Hence, the total complexity of the arithmetic unit at 19.2 kbps is:

$$\text{Arithmetic unit for 19.2 kbps} = 5 \lceil \log_2(nK(2^Q-1)+2S_m) \rceil + 2 + n + K(n+1) + 2^n \lceil \log_2 n(2^Q-1) \rceil \quad (3.15)$$

For the data rates of 38.4 kbps and 57.6 kbps, the complexity is the same as for 19.2 kbps given in equation (3.15) with memory length $K \leq 8$ and $K \leq 7$, respectively. But, for K = 8, the complexity for the 57.6 kbps data rate is:

$$\text{Arithmetic unit for 57.6 kbps} = 10 \lceil \log_2(nK(2^Q-1) + 2S_m) \rceil + 4 + n + K(n+1) + 2^n \lceil \log_2 n(2^Q-1) \rceil \quad (3.16)$$

For the 76.8 kbps data rate and memory length $K \leq 7$, the complexity is given by equation (3.15); for K = 8, the complexity is given by equation (3.16).

3.3.3 Path Memory Storage

As each received branch enters the decoder, decisions of the starting point of the path are made and stored in the path memory. This was shown in the flow diagram of Figure 3.2 and the accompanying

example. In the flow diagram, the last bit in the path memory corresponding to the minimum state metric was output as the decoded bit. However, simulations have shown that, with a path memory of 5(K+1) bits, the last bits of the path memories corresponding to state metrics less than or equal to S_m may be OR-ed together to form the decoded bit. At output probability of error per bit of 10^{-4}, there is negligible degradation using this technique to determine the decoded bit, which is significantly less complex than comparing all the state metrics for the minimum. For the parallel processor, this technique requires 2^K gates, but for the serial processor, an additional 2^K bits of storage are required to specify which state metrics were less than or equal to S_m.

At the 9.1 Mbps data rate, when the decisions in the path memory are to be shifted to include a new decision and the decisions are to be transferred to an appropriate new location, the transfer must be in parallel rather than serially. Therefore, associated with each bit of storage there is a switch to transfer the bit to its new location. The total complexity of the path memory and the output of the decoded bit for 9.1 Mbps is:

$$\text{Path memory complexity for 9.1 Mbps} = \underset{\text{complexity bits}}{5(K+1)2^{K+1} + 2^K} \quad (3.17)$$

For the lower data rates, the transfer of decisions in the path memory may be performed serially. Since the state path memories are transferred simultaneously, only 2^K state path memories are

necessary rather than 2^{K+1} state path memories in the software implementation of the example. The actual transfer is performed by 2^K two-input multiplexers (switches). Hence, the total complexity of the path memory and output of the decoded bit for the lower data rates is:

$$\text{Serial path memory complexity} = (5(K+1)+1)2^K + 2^{K+1} \text{ complexity bits} \qquad (3.18)$$

3.4 PERFORMANCE VERSUS COMPLEXITY

The total complexity for the Viterbi maximum likelihood decoder for various values of the code memory length K and various data rates is described by the following equations derived from the results of the previous sections.

$n = 2,3$	$K \leq 8$	19.2 kbps	$= (2^n+2)\left\lceil \log_2 n(2^Q-1)\right\rceil + (2^K+5)\left\lceil \log_2(nK(2^Q-1) + 2S_m)\right\rceil + 2^K(5(K+1)+2) + 2 + n(Q+2^n+1) + (n+1)K$ (3.19)
$n = 2,3$	$K \leq 8$	38.4 kbps	
$n = 2,3$	$K \leq 7$	57.6 kbps	
$n = 2$	$K \leq 7$	76.8 kbps	
$n = 3$	$K \leq 6$	76.8 kbps	
$n = 2,3$	$K = 8$	57.6 kbps	$= (2^n+2)\left\lceil \log_2 n(2^Q-1)\right\rceil + (2^8+10)\left\lceil \log_2(8n(2^Q-1) +2S_m)\right\rceil + 2^8(47) + 4 + n(Q+2^n+1) + 8(n+1)$ (3.20)
$n = 2$	$K = 8$	76.8 kbps	
$n = 3$	$K = 7$	76.8 kbps	$= 13\left\lceil \log_2 3(2^Q-1)\right\rceil + (2^7+5)\left\lceil \log_2(21(2^Q-1) + 32)\right\rceil + 2^7(42) + 2 + 3(Q+9) + 32$ (3.21)
$n = 3$	$K = 8$	76.8 kbps	$= 13\left\lceil \log_2 3(2^Q-1)\right\rceil + (2^8+10)\left\lceil \log_2(24(2^Q-1) + 32)\right\rceil + 2^8(47) + 4 + 3(Q+9) + 32$ (3.22)
$K \leq 8$	$Q > 1$	9.1 Mbps	$= 2^n(2^{nQ}+1)\left\lceil \log_2 n(2^{Q-1}-1)\right\rceil + 2^K(6\left\lceil \log_2(nK(2^{Q-1}-1) + 2S_m)\right\rceil + 13 + 10K) + n(Q+2^n)$ (3.23)

$$K \leq 8 \qquad Q = 1 \quad 9.1 \text{ Mbps} \quad = 2^n(2^n+1)\lceil \log_2 n \rceil + 2^K(6\lceil \log_2(nK+4) \rceil + 13 + 10K) + n(2^n+1) \tag{3.24}$$

Table 3.2 presents the complexity of the Viterbi decoder for code rate 1/2. For data rates 19.2 kbps to 76.8 kbps, there is only a slight increase in the complexity for memory length K = 8. Also, for three-bit quantization and memory length K = 3, the complexity for the 9.1 Mbps data rate is about four times the complexity for the low data rates since the ROM used by the branch metric processor dominates the complexity of the high data rate. In other cases, the complexity for the 9.1 Mbps data rate is about twice the complexity for the other data rates. To indicate the physical size of the Viterbi decoder, LINKABIT designed a decoder for Q = 3, K = 3 and code rate 1/2 with TTL to operate at a data rate of 10 Mbps. This design required 180 integrated circuit chips. Using a similar design technique, it is found that, for Q = 3, K = 8, and code rate 1/2, a Viterbi decoder can be implemented with TTL to operate at a data rate of 9.1 Mbps requiring about 5500 integrated circuit chips. Thus, by increasing K from 3 to 8, the physical size increases by about a factor of 30 or approximately 2^5, which is an exponential increase in complexity with K. It may be noted that the complexity measure increases by about a factor of 21 where the smaller increase is due to the branch metric ROM being dominant at K = 3.

Table 3.3 presents the complexity of the Viterbi decoder for code rate 1/3. For data rates 19.2 kbps to 76.8 kbps, the complexity

TABLE 3.2. Complexity of Viterbi Maximum Likelihood Decoder for Code Rate 1/2

Quantization Q (bits)	Memory Length K	Data Rate kbps	Complexity Bits
1	3	19.2	257
		38.4	257
		57.6	257
		76.8	257
		9.1×10^3	566
	8	19.2	13,381
		38.4	13,381
		57.6	13,408
		76.8	13,408
		9.1×10^3	31,518
3	3	19.2	305
		38.4	305
		57.6	305
		76.8	305
		9.1×10^3	1,378
	8	19.2	13,825
		38.4	13,825
		57.6	13,962
		76.8	13,962
		9.1×10^3	33,818

TABLE 3.3 Complexity of the Viterbi Maximum Likelihood Decoder for Code Rate 1/3

Quantization Q (bits)	Memory Length K	Data Rate kbps	Complexity bits (Weighted)	
			TTL	MECL II
1	3	19.2	292	--
		38.4	292	--
		57.6	292	--
		76.8	292	--
		9.1×10^3	707	--
	8	19.2	13,421	--
		38.4	13,421	--
		57.6	13,448	--
		76.8	13,454	--
		9.1×10^3	31,659	--
3	3	19.2	367	--
		38.4	367	--
		57.6	367	--
		76.8	367	--
		9.1×10^3	17,081	1,657
	8	19.2	14,240	--
		38.4	14,240	--
		57.6	14,282	--
		76.8	14,297	--
		9.1×10^3	51,009	61,969

is essentially equal at a given memory length and number of quantization bits. For hard decisions, the complexity of the 9.1 Mbps data rate is about twice the complexity for the low data rates. Using TTL for the 9.1 Mbps data rate results in an extremely large ROM in the branch metric processor. Hence, for small K, the ROM dominates the complexity. An alternate approach is to use MECL II and compute the branch metrics with 2^n adders. In order to compare the complexity of a design using MECL II with a design using TTL, a weighting factor must be applied to reflect the differences in parts cost, design cost, and packaging cost. The author feels that a reasonable weighting factor is three for MECL II versus TTL. It should be pointed out that this is a subjective judgment on the part of the author and should not be considered exact. As the cost of TTL and MECL II vary and as new MSI circuit announcements are made, this relationship will change. The weighting factor is applied to the branch metric processor and the arithmetic unit giving a weighted complexity for the decoder of:

$$\text{MECL II decoder complexity } n=3, Q=3 = 273 + 2^K\left(18\left\lceil \log_2(nK(2^Q-1) + 2S_m)\right\rceil + 17 + 10K\right) \qquad (3.25)$$

Table 3.3 illustrates the weighted complexity for the MECL II decoder and the TTL decoder at the 9.1 Mbps data rate. It is observed that there is an order of magnitude decrease in the complexity using MECL II rather than TTL for K = 3, but there is little difference between the two implementations for K = 8; in fact, the TTL decoder

would have a relative smaller cost. The 9.1 Mbps data rate decoder for code rate 1/3 and three-bit quantization is about three times the complexity of the low data rate decoder.

Figure 3.11 presents a comparison of complexity as a function of the required signal energy per bit/single-sided noise spectral density (E_b/N_o) to obtain an output probability of error per bit equal to 10^{-4}. It may be observed that, at K = 8, there is a gain of about 2 dB in E_b/N_o for the same complexity by using three-bit quantization instead of hard decisions on the received symbols. Also, for code rate 1/2, there is a gain in E_b/N_o of about 0.3 dB for the same complexity with the lower data rates instead of the 9.1 Mbps data rate, except for K = 3 where the gain is increased to 0.8 dB due to the large ROM needed in the branch metric processor for the 9.1 Mbps data rate. For code rate 1/3 and low data rates, there is a gain of about 0.3 dB in E_b/N_o with $K \geq 4$ over code rate 1/2 for the same decoder complexity. This is also true for the 9.1 Mbps data rate with hard decisions on the received symbols. However, for three-bit quantization and the 9.1 Mbps data rate, code rate 1/3 illustrates only a gain of about 0.1 dB for K = 6 and about 0.2 dB for K = 7 and 8 over code rate 1/2 for the same decoder complexity. Thus, for $K < 6$, there is little utility in using code rate 1/3 instead of code rate 1/2 since the same performance can be obtained with the same decoder complexity for code rate 1/2.

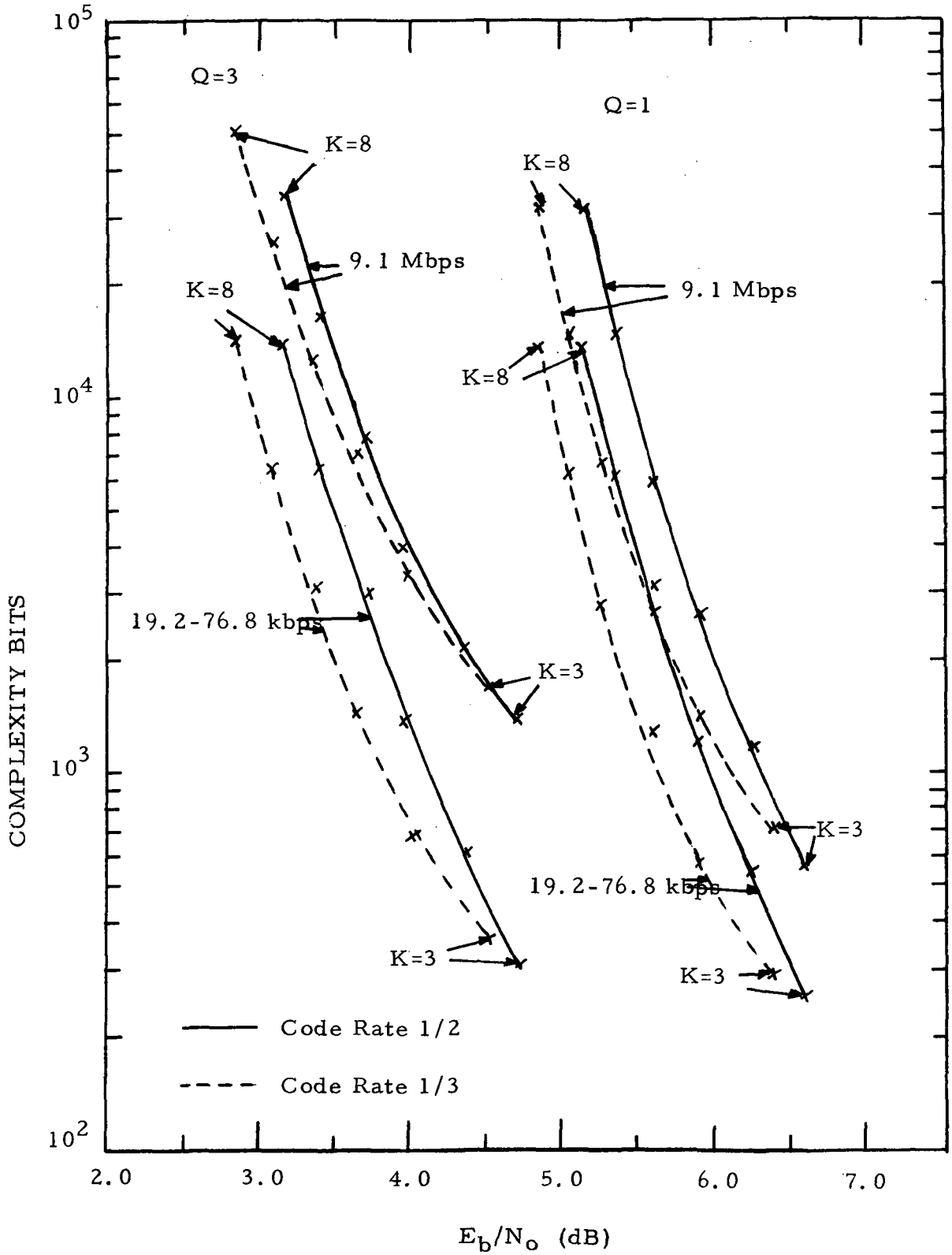


Figure 3.11 Comparison of Complexity Versus Performance At Output Probability of Error Per Bit of 10^{-4}