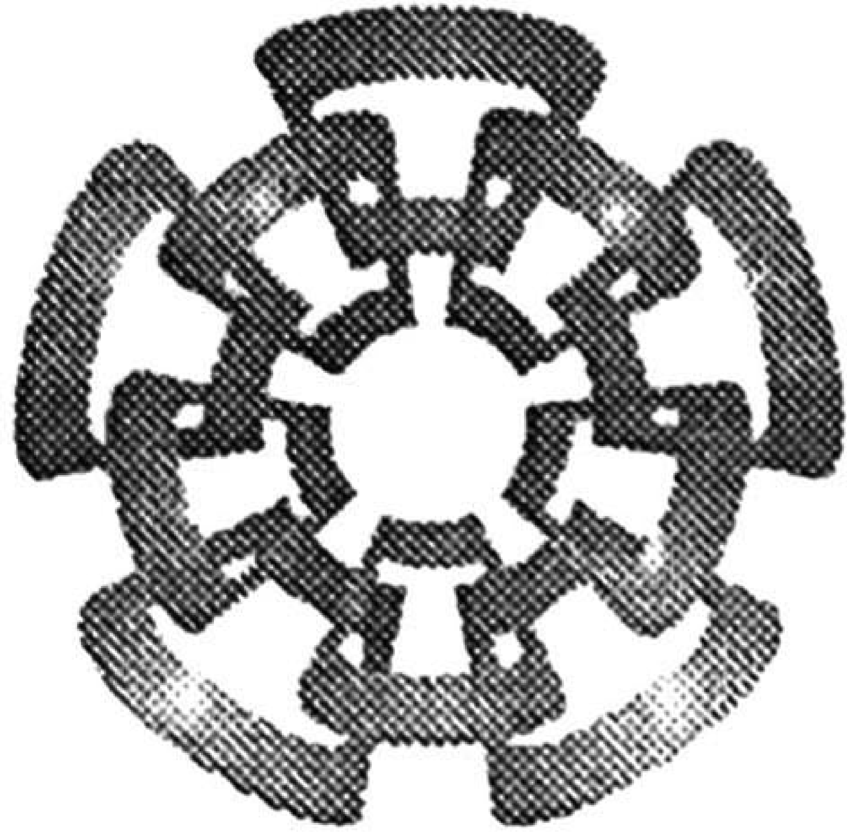
xx (86710.1)

CINVESTAV I.P.N.
SECCION DE INFORMACION
Y DOCUMENTACION



CINVESTAV - IPN

Centro de Investigación y de Estudios Avanzados del IPN
Unidad Guadalajara

UNA IMPLEMENTACIÓN EN VLSI DEL ALGORITMO DE VITERBI

TESIS QUE PRESENTA
LUIS MIGUEL BAZDRESCH SIERRA

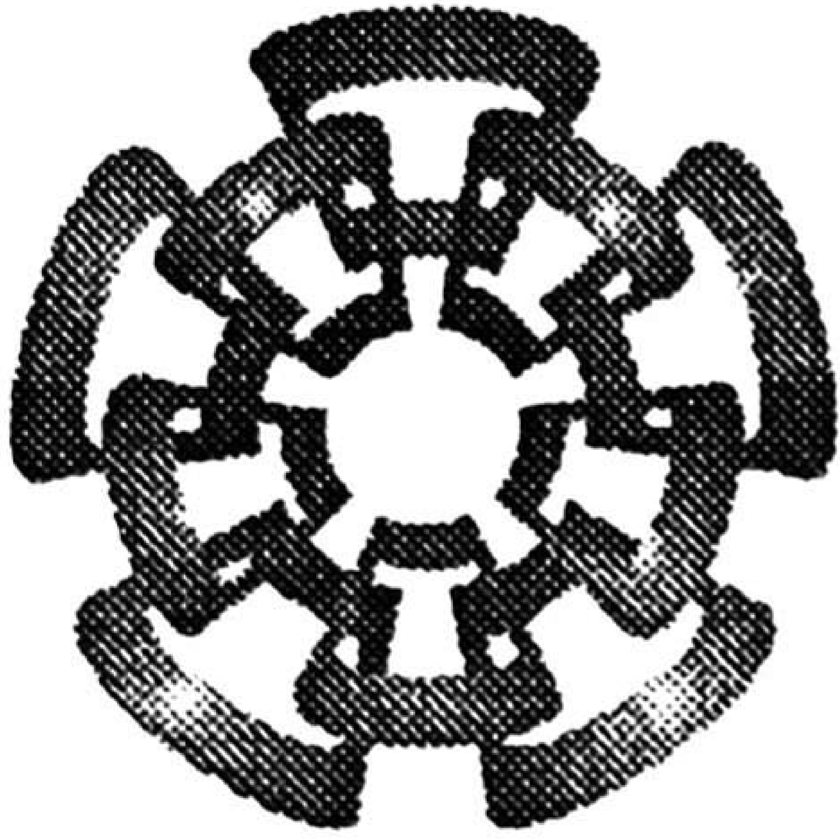
PARA OBTENER EL GRADO DE
MAESTRO EN CIENCIAS

EN LA ESPECIALIDAD DE
INGENIERÍA ELÉCTRICA

CINVESTAV
IPN
ADQUISICION
DE LIBROS

Guadalajara, Jal., Julio de 2000

CLASIF.:
ADQUIS.: Tesis 2001
FECHA: 29-III-01
PROCED.: Depto Servicios Bibliográficos
$



CINVESTAV - IPN

Centro de Investigación y de Estudios Avanzados del IPN

Unidad Guadalajara

A VLSI IMPLEMENTATION OF THE VITERBI ALGORITHM

TESIS QUE PRESENTA
LUIS MIGUEL BAZDRESCH SIERRA

PARA OBTENER EL GRADO DE
MAESTRO EN CIENCIAS

EN LA ESPECIALIDAD DE
INGENIERÍA ELÉCTRICA

Guadalajara, Jal., Julio de 2000

Una Implementación en VLSI del Algoritmo de Viterbi

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

por:

Luis Miguel Bazdresch Sierra

Becario del CONACYT, expediente no. 121115

Directores de Tesis:

Dr. Manuel Edgardo Guzmán Rentería
Dr. Arturo Veloz Guerrero

CINVESTAV del IPN Unidad Guadalajara, Julio de 2000

A VLSI Implementation of the Viterbi Algorithm

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

por:

Luis Miguel Bazdresch Sierra

Becario del CONACYT, expediente no. 121115

Directores de Tesis:

Dr. Manuel Edgardo Guzmán Rentería
Dr. Arturo Veloz Guerrero

CINVESTAV del IPN Unidad Guadalajara, Julio de 2000

Resumen

Los códigos convolucionales representan una técnica muy poderosa de codificación de canal, ya que presentan una ganancia de codificación mayor que la de los códigos de bloque. Un codificador convolucional introduce dependencia entre los símbolos transmitidos, por lo que éstos no pueden ser decodificados de manera independiente, sino que es necesario tomar en cuenta toda la secuencia recibida. El efecto de esta dependencia es una mayor inmunidad al ruido que otras técnicas, con la desventaja de requerir un decodificador de mayor complejidad. El algoritmo de Viterbi es un método de decodificación de códigos convolucionales, que es óptimo en el sentido de minimizar la probabilidad de error. Este algoritmo es conceptualmente sencillo, pero su implementación eficiente en el sentido de área requerida y velocidad de decodificación presenta muchos obstáculos. En esta tesis se presenta una arquitectura para la implementación en un circuito integrado del algoritmo de Viterbi que está muy cerca del óptimo en cuanto a probabilidad de error, requiere un área pequeña y es muy rápido, cuando se compara con otras implementaciones existentes. Para lograr esto, el algoritmo Viterbi original fue estudiado y modificado para disminuir el número de operaciones que realiza, la cantidad de memoria que requiere, y para descubrir y aprovechar al máximo su paralelismo inherente.

Abstract

Convolutional codes represent a powerful channel coding technique, since they present a coding gain larger than block codes. A convolutional coder introduces dependency among the transmitted symbols, so these can no longer be decoded independently of each other, and it becomes necessary to take into account the complete received symbol sequence. The effect of this dependency is better noise immunity than other techniques, with the disadvantage of requiring a decoder of increased complexity. The Viterbi algorithm is a method of decoding convolutional codes, which is optimal in the sense of minimizing the probability of error. This algorithm is conceptually simple, but its efficient implementation, in the sense of required area and decoding rate, presents several challenges. In this thesis, an architecture for the implementation of the Viterbi algorithm in an integrated circuit is presented. This architecture is very close to the optimum in probability of error, requires a small area and is very fast, when compared with other existing implementations. To achieve this, the Viterbi algorithm was analyzed and modified to reduce the number of operations it performs, the quantity of memory it requires, and to discover and seize its inherent parallelism.

Agradecimientos

Aunque el nombre que aparece en la portada de este trabajo me identifica como su único autor, en realidad esta tesis es producto de la confluencia de una cantidad muy grande de circunstancias, personas y voluntades. Voy a aprovechar esta página para nombrar y agradecer a los que, en mi mente y en mi corazón, son los más importantes (en ningún orden particular).

Gracias a mi familia, desde mis abuelos hasta mis primos, porque siempre me han brindado su amor y su apoyo de manera incondicional.

Gracias a Rebeca por su paciencia en las largas noches y fines de semana en que la computadora acaparaba toda mi atención.

Gracias, muy especiales, a Manuel Guzmán, por compartir su conocimiento conmigo, por darme la oportunidad de trabajar con él, por las largas conversaciones, por los regaños, por los ánimos, por los consejos, por su amistad.

Gracias a Arturo Veloz, por ser tan generoso con su conocimiento, y por predicar con el ejemplo.

Gracias a todos mis profesores, y al CINVESTAV, por dedicar tiempo y esfuerzo en convertirme en mejor persona.

Gracias al CONACYT, porque sin su ayuda, no por material menos importante, hubiera sido muy difícil realizar este trabajo.

Por todas estas influencias, y otras incontables y hasta inconscientes, que han hecho de mí lo que soy, por la vida que me ha tocado vivir, doy también gracias a Dios.

A Rebeca

Contents

1 **Introduction** 1

1.1 Overview 1

1.2 Objectives 2

2 **Channel coding in communication systems** 5

2.1 Channel Coding 5

2.2 Convolutional codes 7

2.3 Decoding: the Viterbi algorithm 12

2.3.1 General form of the decoding problem 14

2.3.2 Formal definition of the Viterbi algorithm 15

2.3.3 A note on computational and memory complexity 18

2.3.4 A note on decoding delay 19

2.3.5 The implementable algorithm 20

2.3.6 A note on coding gain 24

2.3.7 Punctured convolutional codes 24

2.3.8 Optimum codes 24

3 **The Viterbi algorithm** 27

3.1 Analysis of complexity 27

3.2 Variables of the algorithm 28

3.3 The unmodified algorithm 29

3.4 Distance normalization 30

3.5 Survivor path memory modifications 37

3.5.1 Survivor path memory length 38

3.5.2 Number of branches decoded at a time 41

3.5.3 Traceback method 43
3.6 Memory Organization 48
3.6.1 Path representation 48
3.6.2 Survivor path memory blocks 49
3.7 Summary of results 51
3.8 Fully optimized algorithm 52

4 Implementation 55
4.1 Architecture 55
4.1.1 The ACS module 56
4.1.2 Survivor Path Memory 64
4.1.3 Traceback Unit 67
4.1.3 Other considerations 70

5 Results 71
5.1 Simulation results 71
5.2 Experimental results 72
5.2.1 Lab verification 73
5.2.2 Area resources used 73
5.2.3 Decoding rate and clock period 74
5.2.4 Decoding delay 74
5.3 Comparison with other implementations 74

6 Conclusions 79
6.1 Review of objectives 79
6.2 Future work 79
6.3 Final remarks 80

Appendix A 83

Appendix B **89**

Appendix C **95**

Appendix D **99**

References **101**

1 Introduction

1.1 Overview

This thesis consists in the study, evaluation, analysis and implementation of the Viterbi algorithm for decoding convolutional codes. This algorithm doesn't rely on very complex operations, but is difficult to implement because of two peculiarities: it operates on a very large number of variables, and its complexity grows exponentially as the error-correcting capability of the convolutional code improves.

The Viterbi algorithm has a broad range of applications; as a matter of fact, it has equivalents in several areas of engineering and computer science. The problem it solves is that of finding the shortest path between two points in a given graph. In this thesis, the algorithm is studied for a specific application: optimum decoding of convolutional codes.

Chapter 2 of this thesis is called *Channel coding in communication system*It verses on the problem of channel coding from the perspective of information theory. Convolutional codes are studied as a solution to this problem, from which the decoding problem arises. The Viterbi algorithm is studied in a theoretical manner, and it is shown that it decodes convolutional codes optimally.

Chapter 3 is called *The Viterbi algorithm*In this part of the thesis the Viterbi algorithm is studied and analyzed. A number of parameters that affect its performance and behavior (both in terms of bit error probability and algorithm complexity) are found. These effects are analyzed using simulation, with the aim of setting the grounds for an architecture that can be implemented in VLSI and meets the required performance criteria.

The nature of the Viterbi algorithm is such that, in order to implement it, some

optimizations and changes must be performed on it. In order to do that, it is necessary first to find the areas where optimizations and changes can be made without affecting the algorithm's performance, or where these affect it in a quantifiable, controllable way.

It is important to note that a considerable reduction of the required number of operations per received symbol is *conditio sine qua non* to implement the algorithm in VLSI with a reasonable information rate and cost.

In chapter 4, *Architecture and Implementation*, a specific application is chosen (regarding constraint length, channel model, required performance, etcetera) and the knowledge acquired in the previous two sections is used to design an architecture that meets the specifications and can be implemented in VLSI.

Afterwards, a circuit based on this architecture is designed and tested. The design is written in VHDL, and testing is done both in a simulation environment and in the laboratory, in hardware.

A summary of the results is presented in Chapter 5; some conclusions and final thoughts are presented in Chapter 6.

Some ancillary material is presented in the appendixes at the end of this report. Appendix A extends on the theory of maximum likelihood decoding and the detection of vectors in noise presented in Chapter 2. Appendix B presents some basic material on the calculation of signal-to-noise ratio in a communications system, which is useful to interpret the results of the simulations realized. Appendix C contains the test plans used to verify the C program, as well as the implementation, both in simulation and in the laboratory.

A listing of the C program used to simulate the Viterbi algorithm, as well as its VHDL implementation, are included in a CD-ROM. This is because of the length of the programs, which amount to around 50 pages. Appendix D explains the organization of the material on the CD-ROM.

1.2 Objectives

This thesis pursues the following objectives:

I. To acquire a thorough comprehension of the Viterbi algorithm and its application in the decoding of convolutional codes

II. To analyze how the properties of the Viterbi algorithm change when the values of its parameters are made to vary, and

III. To implement the algorithm in VLSI, meeting the following cost, area, and performance specifications:

i. it must fit in an Altera FLEX10K100 EPLD

ii. it must not require any external memory

iii. it must be optimized for maximum speed (as opposed to minimum area)

iv. it must decode one symbol per clock cycle

v. it must be designed for codes of rate ½, constraint length no less than 7

The main result of this thesis is a working VLSI architecture for Viterbi decoding of convolutional codes that is very fast, yet meets the area and cost requirements. In order to be able to propose such an architecture, though, the Viterbi algorithm must be thoroughly understood. It is not sufficient to understand the mechanics, or the sequence of operations, that the algorithm performs. It is also necessary to understand how the algorithm behaves when it is modified in certain ways, in order to minimize its complexity while conserving its error correcting capability.

To achieve maximum performance, the algorithm's operations must be performed in parallel. The Viterbi algorithm is expressed in such a way that the operations that can be executed in parallel are easily detected. How to implement this parallelism, and at what cost, is something that must be analyzed.

The area requirements imply that the ability of the circuit to have configurable parameters, such as generating polynomials, constraint length, memory length, etcetera, will have the last priority.

2 Channel coding in communication systems

2.1 Channel Coding

The main objective of communications engineering is to transmit information faster, with fewer errors, and at a lower cost. These three objectives are mutually exclusive - when one is fully met, the others are not. Channel coding has to do with the objective of transmitting with fewer errors.

A message transmitted through a channel, whichever it might be, suffers distortions and modifications, which produce errors [1], [2], [3]. When a message contains redundancy (that is, that uses more symbols than strictly required to convey its meaning), the receiver has a better chance of reverting the effects of those distortions and modifications. The channel coding problem is how to invent methods to create this redundancy in such a way that the information rate is kept as high as possible, and the probability of error is reduced as much as possible, while the decoder is kept as simple as possible. These conditions are, again, mutually exclusive, and when designing a communication system it is necessary to find the best tradeoff for each particular situation.

Claude Shannon built the pillars of communication theory with his two main theorems [21]. Shannon also established the information transmission capacity limit for given bandwidth and power. This limit is known as "channel capacity." In his so-called "second Shannon theorem" or "channel coding theorem", he showed that there exist decodable channel codes that make the probability of error as small as desired, provided the

information rate is less than channel capacity.

Shannon showed that channel codes that reduce the probability of error as much as desired exist, but he didn't show how to build them. The channel coding theorem is an existence theorem, and Shannon's proof is not constructive. It has taken many people many years to find codes that approach the established limit.

To quantify a code's efficiency, the difference between the signal-to-noise ratio (*SNR*) needed to obtain the same bit error rate (*BER*) of a coded and an uncoded message is calculated. This quantity is known as **coding gain** Several things should be kept in mind when coding gain is calculated:

First, the source rate must be kept constant. In general, a more powerful code adds more redundancy to a message, which degrades reception because the channel rate increases; however, the added error-correcting capability of the decoder provides a net performance improvement over uncoded transmission.

Second, it is assumed that the channel bandwidth is infinite, and that the only signal disturbance is due to additive noise. Usually, additive white Gaussian noise is assumed.

Third, it is assumed that the same modulation scheme is used in the systems being compared.

Fourth, it is important to realize that communication systems are usually compared using **bit-normalized signal-to-noise ratio** or E_b/N_0, instead of just signal-to-noise ratio. (Please refer to Appendix B for further elaboration on signal-to-noise ratio). E_b/N_0 is a better measurement than *SNR* because it takes into account the bit rate and the bandwidth of the system under consideration. The relation between *SNR* and E_b/N_0 is:

$$\frac{E_b}{N_0} = \frac{ST}{N_0} = \frac{S}{RN_0} = \frac{SW}{RN_0W} = \frac{S}{N}\left(\frac{W}{R}\right)$$

where

E_b is the energy contained in the signal that corresponds to one bit;

N_0 is twice the amplitude of the power spectral density of the noise, assumed white, measured in watts per hertz;

W is the bandwidth of the system;

S is the average modulating signal power;

T is the bit time duration;

$R = 1/T$ is the bit rate;

$N = N_0W$ is the filtered noise power.

The maximum coding gain achievable, according to Shannon theory, is around 11.5dB. Typical coding gains for convolutional codes are around 7dB [1].

It should be noted that coding gain can also be used to compare different codes. In this case, it should be specified which code is being used as reference.

This is the motivation behind the whole channel coding theory. Convolutional codes represent one of the main achievements in this field.

2.2 Convolutional codes

Convolutional codes [8] add structured redundancy to a data sequence. In a finite state machine, the present state corresponds to the contents of the memory elements of the machine. The next state is determined from the present state and its input. The machine's output is a logic function of the input and its present state (or, as a particular case, of just the present state). A convolutional coder is a state machine that takes k input bits at a time, serially, and produces n output bits. These n output bits represent, in a way, the machine's output as a result of the transition from one state to another. The combinational logic used to calculate the output is an exclusive-or of some of the stored bits. Figure 1 shows the diagram of a generic convolutional coder.

It is said that a convolutional coder has memory, because in order to calculate the n output bits that correspond to the k input bits at any given time, it also takes into account M-k previous input bits, which are stored in the machine's memory. M is the total amount of memory of the coder. The number of bits stored in the machine, divided by k (M/k) is known as the **constraint length** of the coder, and is designated by L. The rate k/n is known as the **code rate**, r. A code is commonly specified as a triplet (L, k, n).

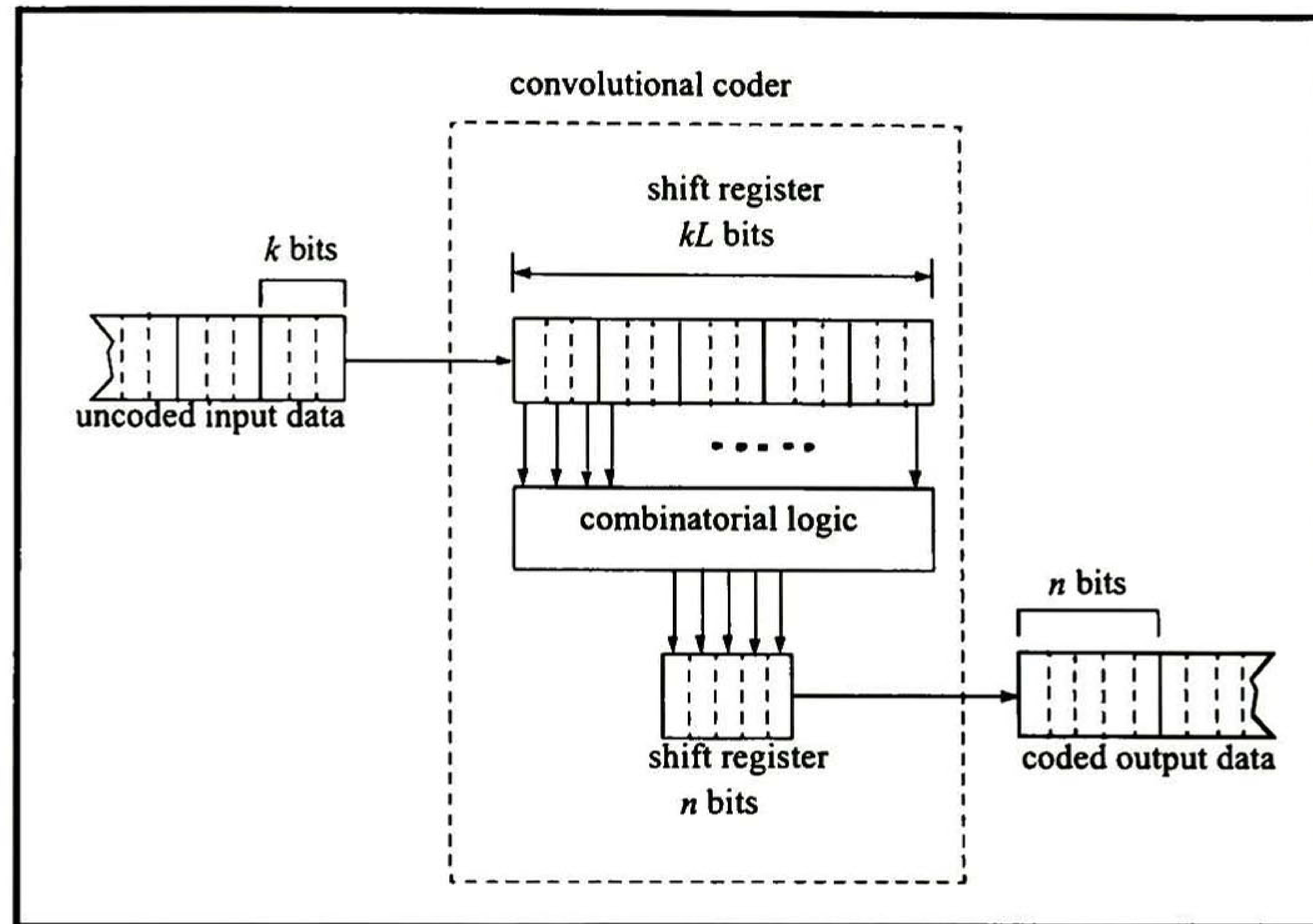


Figure 1. *Schematic Diagram of a Convolutional Coder*

The **minimum free distance** d_{free}, of a convolutional code measures the error-correcting capability of the code. A convolutional code can correct $(d_{free}-1)/2$ error bits within a few constraint lengths, where a few means from 3 to 5 [1].

There are several ways to represent a convolutional coder. It can be done in schematic form; by specifying its structure; by the logic function that calculates its output; by a state diagram; or graphically, through a tree or trellis.

Perhaps the more straightforward way to represent a convolutional coder is through its schematic diagram. It is convenient and simple, but its main disadvantage is that it makes a deeper analysis of the code very difficult. Figure 2 shows a schematic for a code of rate ½ and constraint length 3.

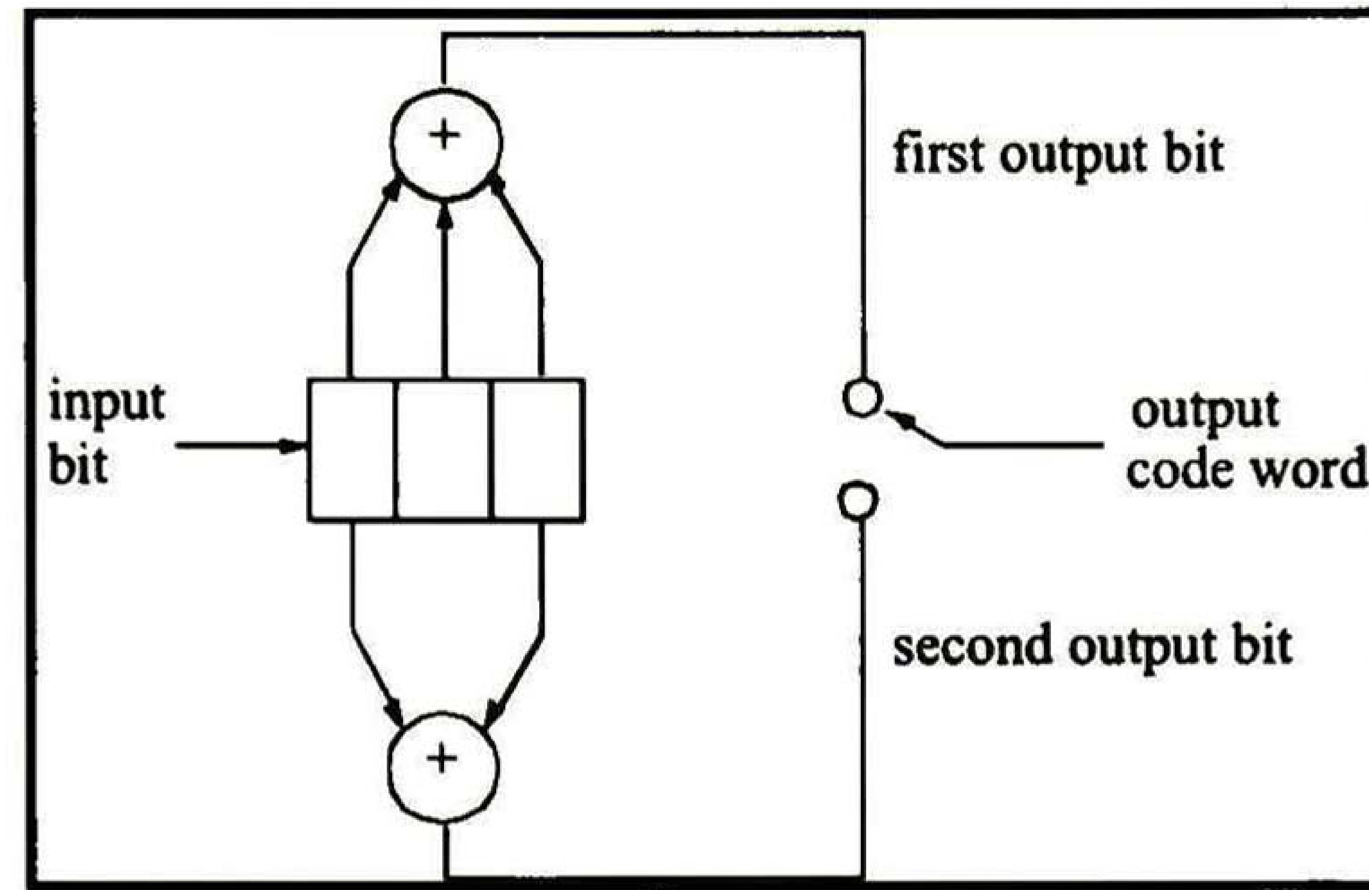


Figure 2. *A rate ½, L=3 convolutional coder*

A coder can also be succinctly represented by *n* **generator polynomials** Each polynomial has *Lk* coefficients, each of value 0 or 1. The coefficient indicates whether the stored bit is included in the XOR function that calculates the output. In turn, each polynomial can be expressed as a **connection vector**that includes only the coefficients. For example, the generator polynomials and their corresponding vector for the coder in figure 2 are:

$$p_1(D) = D^2 + 1$$
$$p_2(D) = D^2 + D + 1$$
$$g_1 = [1\,0\,1]$$
$$g2 = [1\,1\,1]$$

The vectors can be expressed in a numerical base, eight for example, in order to have an even briefer representation. In this example, g_1 would be represented as [5] and g_2 as [7]. A set **S** of all generators can be defined, and in this case it would be **S**={5,7}.

A convolutional coder is a finite state machine and as such can be represented by a state diagram. This representation presents the coder at a higher level of abstraction, concentrating on the function and not on the structure of the coder. From a state diagram, it is very easy to see the relationships between the states, and the output for any given input is readily found. Its main advantage is that some properties of the coder, like its minimum free distance, can be obtained from this diagram. For large *L* (or *r*), however, the state diagram

becomes impractical, due to its size.

It should be noted that, since the convolutional coder is a finite state machine, all theory developed to analyze such circuits can also be applied to the study of the coder. However, such theory has been developed with emphasis on the analysis and synthesis of digital circuits, and is not always useful from the point of view of communications theory. Figure 3 shows the state diagram for the example coder.

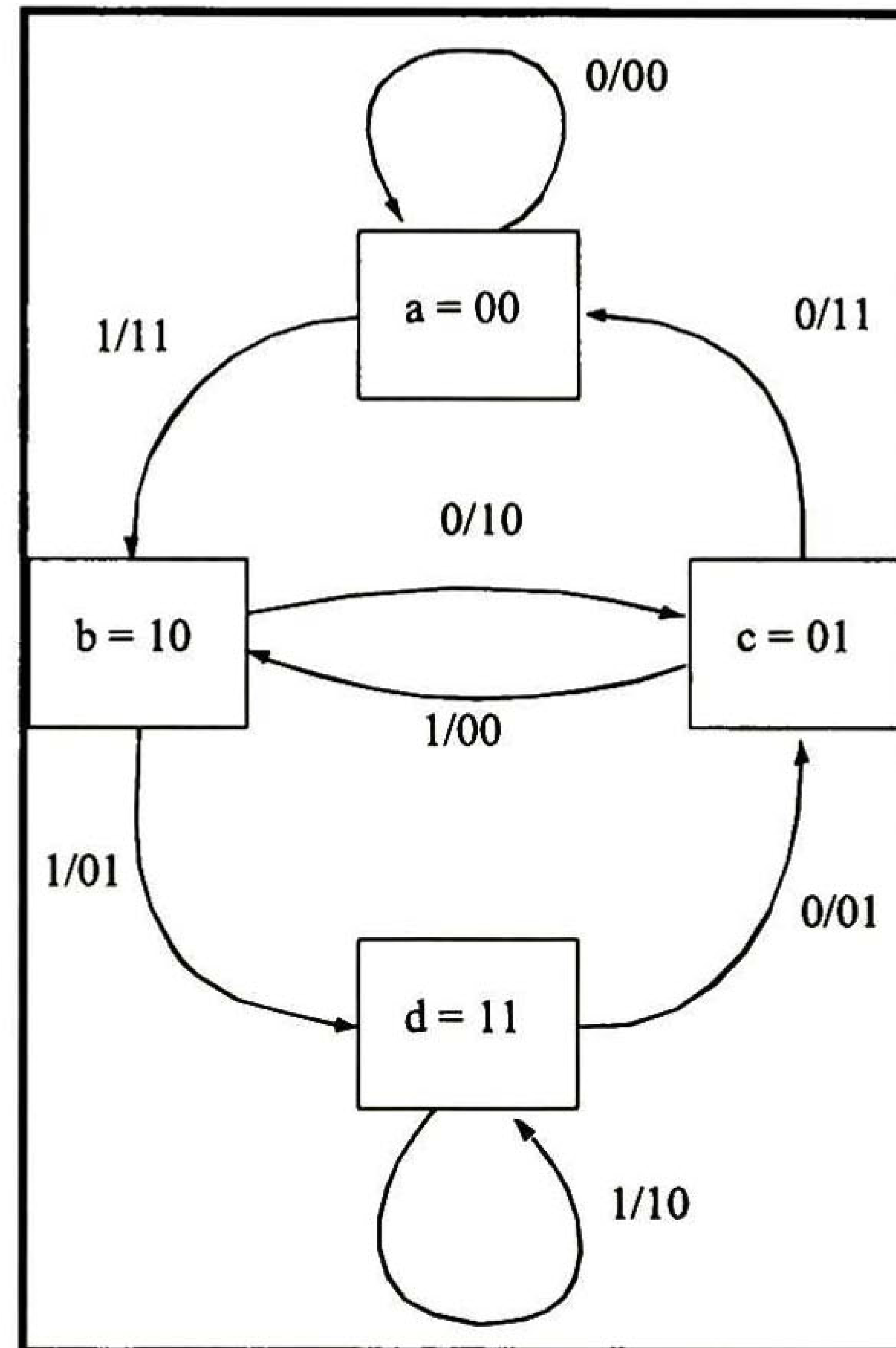


Figure 3. *State diagram of example coder*

So far, no representation is very good at tracking the evolution of the coder as time passes. The tree diagram is especially useful for this purpose. It starts at the initial state of the coder, and branches out tracing all input possibilities. To find the output and state evolution for any input sequence, a branch in the tree is chosen at each time iteration, and followed starting from the root.

The advantage of the tree is that it emphasizes the effect of time on the coder, adding a dimension that previous representations lack. However, for large L (or r), the tree grows very quickly and is not useful beyond a few iterations. The tree for the example coder is shown in figure 4.

It can be seen that the tree is very regular. The same structure is repeated time after time. This repeating structure can be identified and collapsed into what is called a trellis diagram. It is equivalent to a tree, evolving with time; the difference is that the trellis does not expand as time passes. As opposed to all previous representations, trellises can be constructed for codes with large L (or r). The trellis diagram for the example coder is shown in figure 5.

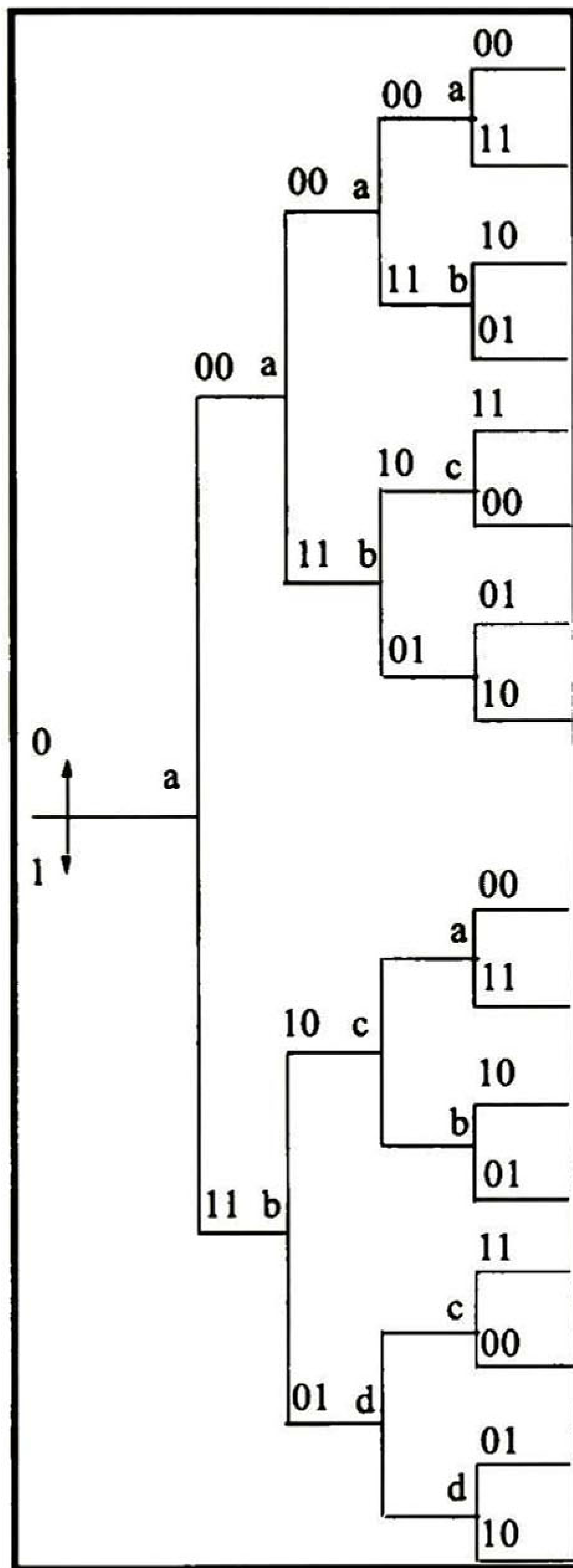


Figure 4. Tree diagram of example coder

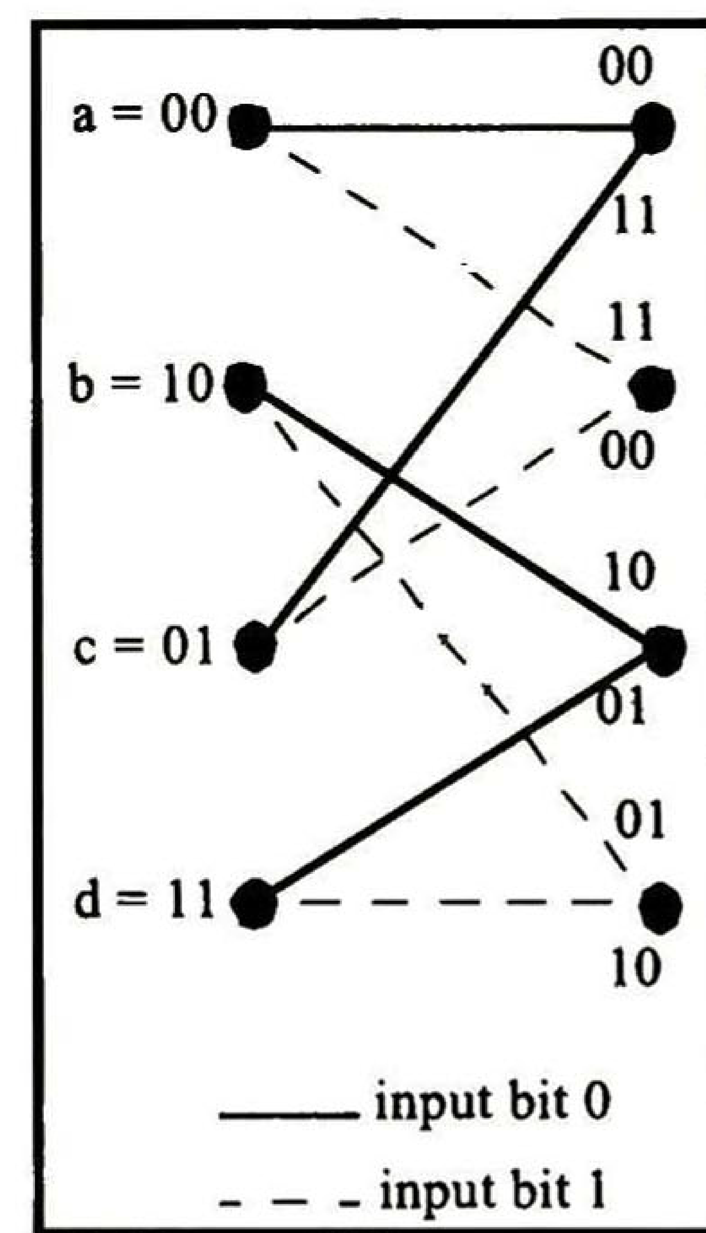


Figure 5. Trellis diagram of example coder

A possible sequence of states in a trellis is called a **path**. A transition from one state to another is called a **branch**.

As will be seen later, trellises are especially important for the Viterbi algorithm [4]. Note that the evolution of the coder in time traces a path through its trellis. The task of the

Viterbi algorithm is to build a trellis for the received sequence, and then to find the most likely path through it. The trellis is a representation that is particularly well suited to the decoding problem.

2.3 Decoding: the Viterbi algorithm

For a code to be useful, it must be decodable, that is, there must be a way to estimate the original message from the received one. There are several algorithms for convolutional decoding: Viterbi, sequential (proposed by Wozencraft and modified by Fano), stack, feedback, and syndrome decoding to name several.

Omura [22] showed that convolutional codes can be decoded optimally, in the maximum likelihood sense, using the Viterbi algorithm.

Let a transmitter generate a binary data sequence, **T**. This sequence is first source-encoded to eliminate all redundancy, and then channel-encoded by a convolutional coder to generate a sequence, $\mathbf{U}^t$, out of the set of all valid sequences, $\mathbf{U}=\{\mathbf{U}^1, \mathbf{U}^2, \ldots \mathbf{U}^z, \ldots\}$. Note that, if **T** is infinite, **U** has an infinite number of elements, i.e., the number of possible encoded sequences is infinite. If **T** is a finite sequence, then **U** has a finite number of elements and the convolutional coder is reduced to a block coder. Let **Z** be the symbol sequence received from a channel with additive white Gaussian noise (i.e., noise samples are independent). Then, the receiver can estimate the transmitted encoded sequence $\mathbf{U}^t$ with minimum probability of error if all the **likelihood functions**$P(\mathbf{Z}\,|\,\mathbf{U}^m)$, where $m=1, 2, \ldots, z, \ldots,$ are compared and the most likely is chosen. That is, the sequence $\mathbf{U}^x$ is chosen as the transmitted sequence if

$$P(\mathbf{Z}|\mathbf{U}^x) = \max_{for\ all\ m} P(\mathbf{Z}|\mathbf{U}^m) \tag{1}$$

For a code of rate $1/n$, the likelihood function can be expressed as:

$$P(\mathbf{Z}|\mathbf{U}^m) = \prod_{i=1}^{\infty} P(Z_i|U^m{}_i) \tag{2}$$

where Z_i is the ith code word of **Z**, and $U^m{}_i$ is the ith code word of $\mathbf{U}^m$. A code word corresponds to a branch in the trellis. Each code word is formed by n code symbols.

The received code words in sequence **Z** come from a demodulator. The demodulator can operate in one of two ways. One way is to make a firm, or hard, decision as to whether each received code word Z_i represents one of the valid code words. In this case, the output of the demodulator is always one of the valid code words. When this kind of demodulator is used, the decoder is called a **hard-decision decoder**

On the other hand, the demodulator can feed the decoder not just with a code word, but also with a measure of confidence attached to it. This measure of confidence is related to the position of the received vector in its decision region. This measure of confidence is extra information that helps the decoder to better reconstruct the received sequence. Such a decoder is called **soft-decision decoder**

Normally, a soft-decision demodulator output is a n-bit word that represents 2^n levels of confidence. Having 8 levels of confidence (3 bits) results in a 2dB coding gain with respect to hard-decision decoding. An infinite number of levels of confidence results in a 2.2dB gain. For this reason, 8 levels of confidence are almost always used along with Viterbi decoding [1]. Since soft-decision decoding doesn't add much to the algorithm's complexity, hard-decision decoding is seldom used and will not be considered in the rest of this work.

As is seen in Eqs. (1) and (2) above, the likelihood function compares all possible sequences $\mathbf{U}^m$ against the received sequence **Z**. This function does not provide information on the probability of error for a given symbol; it only provides a likelihood measure for a sequence. It is possible that the most likely sequence contains several very unlikely symbols, as long as the whole sequence remains likely.

Finally, the product in Eq. (2) can be transformed into a sum by taking the logarithms of the individual probabilities. Note that the logarithm is a monotonically increasing function. Doing this reduces the problem to finding $\mathbf{U}^m$ such that

$$\sum_{i=1}^{\infty} \log P(Z_i | U^m{}_i) \tag{3}$$

is maximized.

In Appendix A it is shown that the problem can be reduced to that of finding $\mathbf{U}^m$ that

minimizes:

$$\sum_{i=1}^{\infty} f_T(Z_i, U^m{}_i) \tag{4}$$

where f_T is a function that calculates the Euclidean distance between Z_i and $U^m{}_i$, and code words and received signals are interpreted as vectors in a signal space.

A trellis diagram defines all possible paths that a coder can take. Each branch in the trellis can be associated with the vector that corresponds to coder output for that branch. For example, in the trellis in Fig. 5, the branch that goes from state *a* to state *a* (output 00) could be associated with vector (1,1), while the branch that goes from state *a* to state *b* (output 11) could be associated with vector (-1, -1), in some signal space.

The Viterbi algorithm uses a trellis to calculate the Euclidean distance between each branch and the received vector at each time iteration, as indicated in Eq. (4). From all the branches that arrive at a given state, the one with minimum distance is kept, and the others are discarded. The branch distances for each state are then added, so the total distance for each path is known. Then, the path with the minimum distance is chosen and decoded.

In this way, if the trellis representation of the code is used, then the decoding problem is reduced to finding the path through it that maximizes the likelihood function, or, equivalently, the shortest path. The Viterbi algorithm makes this approach feasible, in terms of cost and decoding speed, because it finds and eliminates paths that cannot possibly be the most likely one, even before their final likelihood is calculated. Then, the most likely path is chosen from a reduced set of **surviving paths**

2.3.1 General form of the decoding problem

In a more general way, the problem at hand is that of an FSM that generates output symbols, which are observed through a noisy medium, with the effect that some transitions are not observed correctly. It is desired to find the most likely sequence of input symbols to the FSM. A representation of this problem can be seen in figure 6.

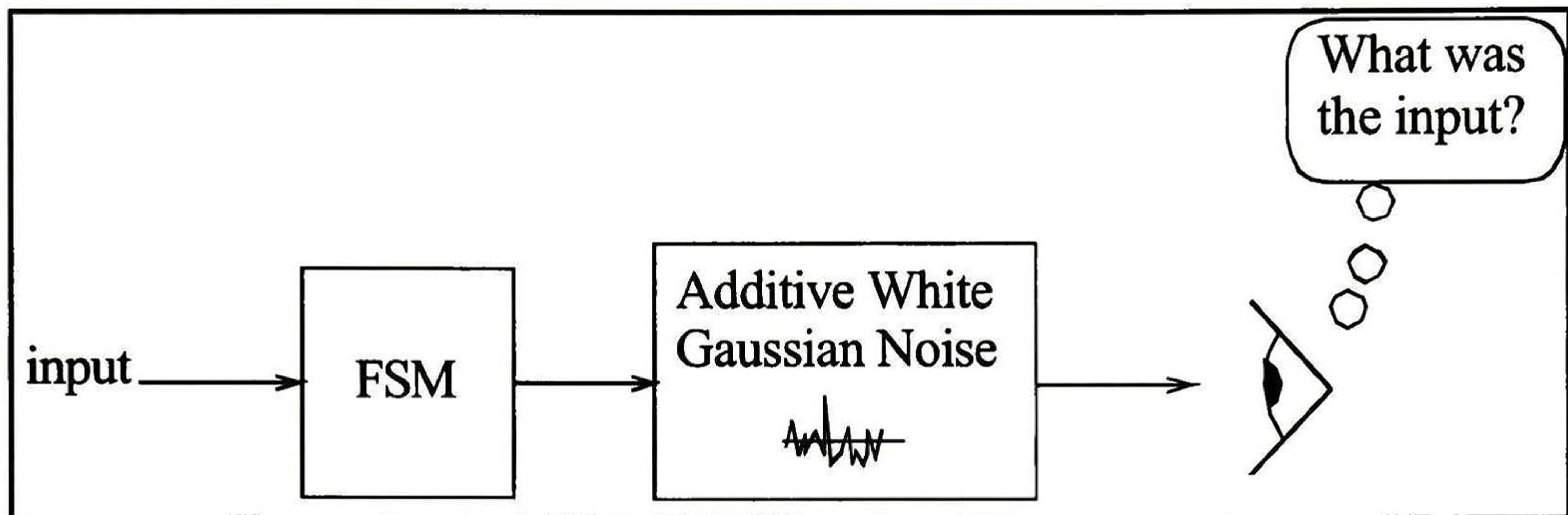


Figure 6. *General form of the decoding problem*

As was seen in section 2.3, the problem can also be stated as that of finding a path in a trellis that meets a certain criterion (i.e., is the shortest). This problem has been widely studied in several areas of engineering and computer science, and is also known as dynamic programming.

2.3.2 Formal definition of the Viterbi algorithm

What follows is the formal definition of the Viterbi algorithm. The initial value of the different variables used is specified in its definition. The intention in the presentation of the algorithm is to expose the operations that might be executed in parallel as much as possible.

Definitions

Let:

$R = r_1, r_2, r_3, \ldots$ the sequence of received, noisy symbols that are to be decoded

U the set of states of the convolutional coder

T a trellis, defined as the pair (S, g_T) where

Se U x U is a set of ordered pairs (x, y), specifying that state x is connected to state y. That is, (x, y) is a branch in the trellis

g_T: S -> $\mathbf{R}^n$ is a function that takes an element of S and returns the vector that corresponds to the output of the coder for that element. This vector can be seen as a label associated to branch (x, y), and as such is frequently called **branch label**

M a vector, where the element in position x is the accumulated distance for the path that ends in state x. It is initialized to zero

M_temp: a vector used to store temporal values, before storing them in M

D a two-dimensional array where the algorithm stores the paths it creates. Each column is a time iteration, and each row is a state. Element (w, z) contains the previous state for state w at time iteration z. It is initialized to zero

O is the output (decoded) sequence

|| is the concatenation operator. $x_3 \,||\, (x_2, x_1) = x_3, x_2, x_1$

Algorithm: Viterbi_Theoretical(R, U, T)

Inputs: A sequence of noisy symbols

Output: A sequence of uncoded symbols

```
Viterbi_Theoretical(R, U, T)
{
  // calculate path metrics and shortest path
  t = 0;                        // t counts number of symbols in R
  while(! is_empty(R)) {        // repeat for all symbols in R, taken in sequence
    t = t + 1;                  // update iteration counter
    r = getNext(R);             // get next element from R and remove it
    for (x : x e U) {           // find minimumdistance between each state in coder
                                // trellis and received symbol
      m = infinity;
      for (y : (y, x) e S) {    // find minimum distance between each branch that
                                // ends in state x and received symbol
        d = distance(r, g_T(y, x)) + M(y);   // d is distance between received
                                             // symbol andthe current branch, plus
                                             // total length of path up to state where
                                             // current branch originates
        if (d < m) {
            m = d;              // find and store minimum distance
            o = y;              // and state from which it originates
        }
      }
      M_temp(x) = m;            // new path distance is m
      current_state = D(minimum(M_temp), t);   // find shortest path (minimum
                                               // accumulated distance)
      D(x, t) = o;                  // store previous state in the trajectory
    }
    M = M_temp;                     // update path distances
  }
  // once all elements of R have been read, identify shortest path. For each iteration, the
  // previous state for each path is stored in D. Then, the path can be traced back in its
  // entirety. At each iteration, the current branch can be decoded to reconstruct the
  // original sequence. This process is known as traceback.
  while(! t = 0) {                  // traceback whole trajectory
    next_state = D(current_state, t);       // find previous state
    O = O || decode(next_state, current_state); // calculate the coder input symbol
                                                // that corresponds to this branch
    current_state = next_state;     // move back one step in D
    t = t - 1;
  }
}
```

Note that, as stated previously, this algorithm stores a potentially infinite amount of

data, and has a potentially infinite decoding delay.

The quantity $d =$ **distance**$(r, g_T(y, x))$ is the distance between the current branch label and the received symbol. It is known as **branch metric** Each element stored in M represents the total length of the corresponding path, and is called **path metric**

The algorithm above has been defined as using traceback to traverse the trellis and decode the chosen path. It should be noted that one alternative to traceback exists. This alternative, known as register exchange, uses a number of shift registers equal to the number of states of the trellis [5], [9]. Each shift register i always holds the survivor sequence for state i. To accomplish this, it is necessary to update the entire contents of every shift register every iteration. Then, the decoded data is obtained from the output of the shift registers.

The traceback algorithm, on the other hand, uses a memory array (D, in the algorithm above) to store pointers to the previous state. When a path is chosen, it is reconstructed by "tracing back", from one state to the previous one.

The main disadvantage of the register exchange technique is that the wiring area needed for updating the stored data grows very fast with the number of states of the encoder. Another disadvantage of a register exchange implementation is that it is very difficult to program it to work for different trellises.

For these reasons, the traceback algorithm is almost always chosen in favor of register exchange. In this work only the traceback version of Viterbi decoding will be studied.

2.3.3 A note on computational and memory complexity

Commonly, the time complexity of an algorithm is measured by the number of operations it must perform, and how this number grows when the number of inputs to the algorithm grows.

In these terms, the Viterbi algorithm is very complex. At each time iteration (every time a new symbol arrives at the decoder), 2^k paths per state must be considered, and 2^{Lk} states must be analyzed. For practical applications, the number of iterations per second that the decoder must perform may vary from a few thousand to several million.

The same applies to the memory requirements. The memory to store the survivor

paths must contain an entry per state, and a path metric per state must also be stored.

Since k and L appear as exponents, it is said that the computational complexity of the algorithm is exponential. Every time L is increased in 1, the number of states doubles. Likewise, every time k is increased in 1, the number of paths into each state doubles. To illustrate, figure 7 shows how the number of operations performed by the algorithm grows with L, while figure 8 shows the same for memory requirements. Figure 7 shows the number of operations per decoded bit. The numbers are estimated from simulations of the algorithm, as explained further in section 3.1.

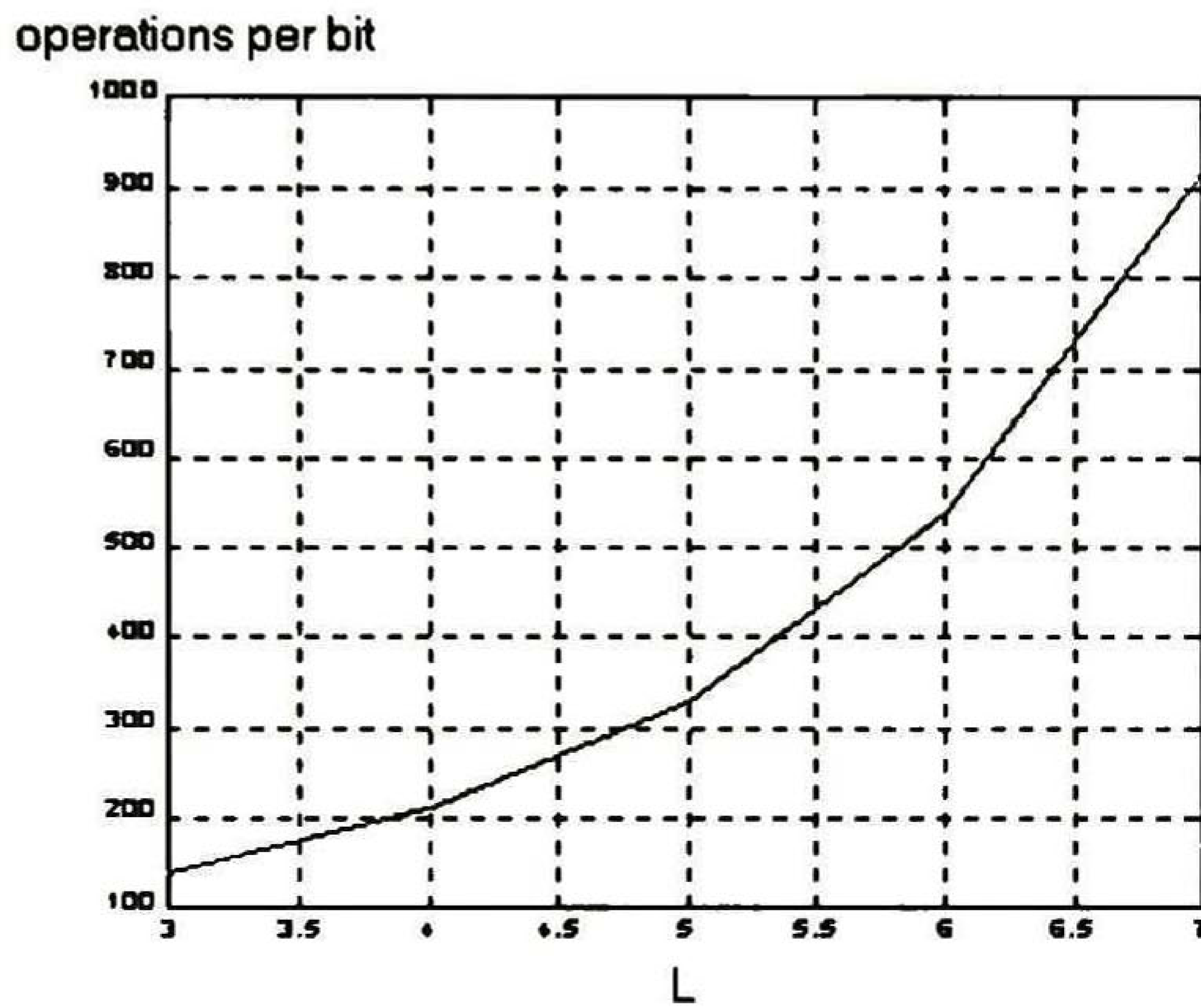


Figure 7. L vs. Operations per decoded bit

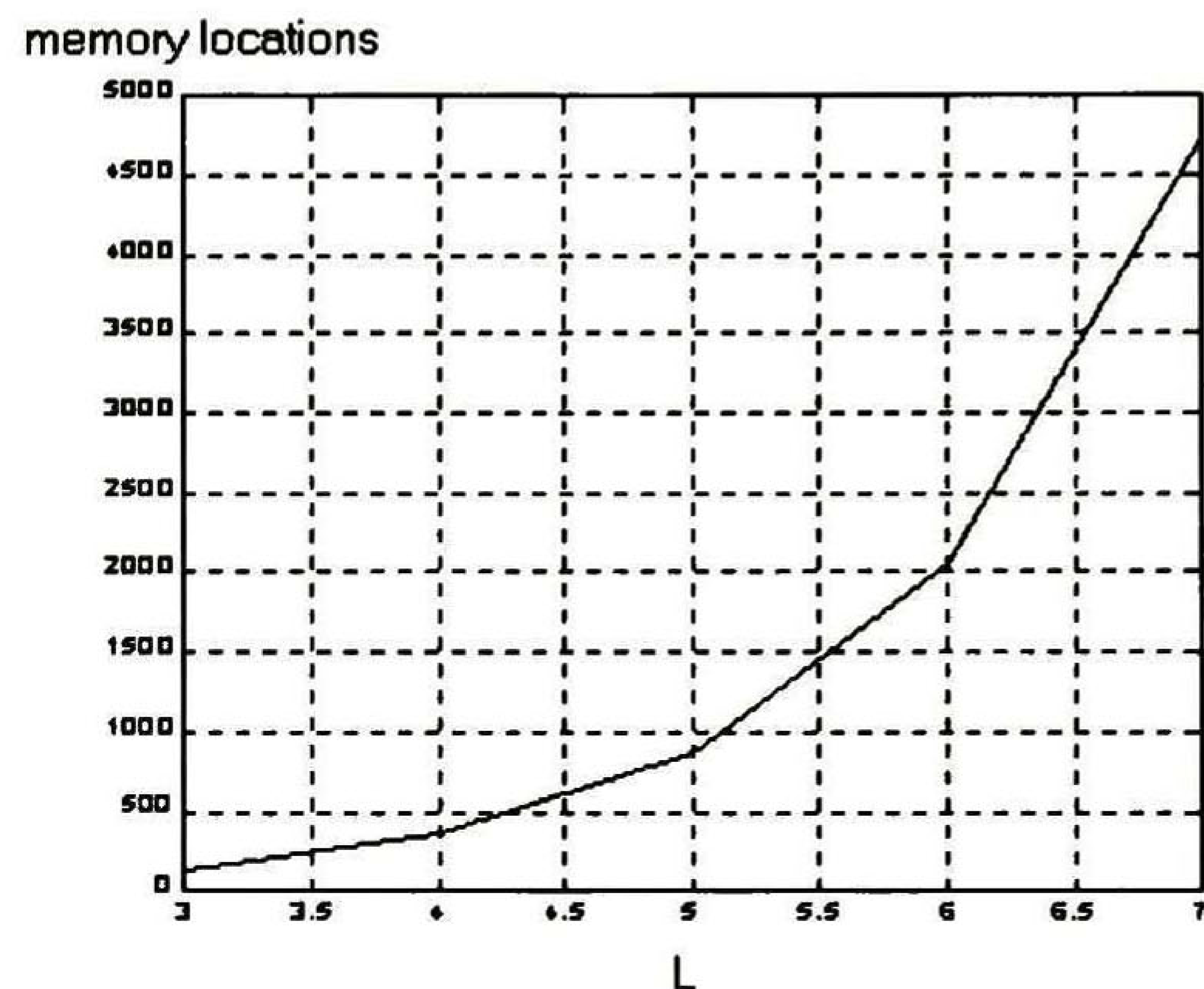


Figure 8. L vs. memory requirements

Note that the maximum likelihood criterion defined in Eq. (1) requires that the sequences $\mathbf{Z}$ and $\mathbf{U}^m$ are compared in their entirety, that is, that the complete sequences must be compared. What this means is that the algorithm may choose the most likely path only when all symbols have arrived; after that, one path is chosen and decoded. In practice, this means that impossibly large amounts of memory and decoding delay (see next section) are required.

2.3.4 A note on decoding delay

Decoding delayis defined as the time that a decoder takes to decode a symbol after

it arrives at its input.

Quantifying this delay is important because certain applications are very sensitive to it (i.e., voice), whereas others are very tolerant to it (i.e., file transfers).

The Viterbi algorithm presents a decoding delay proportional to the length of its input sequence, because the whole sequence must be stored before decoding it. For most practical applications, this delay is unacceptable.

Note that, trying to avoid this limitation, an infinite sequence of data could be broken into finite blocks, so that each block requires finite memory and delay to decode. This approach is not always feasible, and effectively turns the convolutional encoder into a block coder.

2.3.5 The implementable algorithm

There are three main obstacles for implementing the Viterbi algorithm as it was described. The first is that it requires, in principle, an infinite amount of memory to store the trellis. The second is the sheer number of operations it requires to decode a data sequence. The third is that the path metrics will, in time, cause an overflow in the registers used to store them.

In this section, the algorithm is modified to make it use a finite amount of memory, D, even though the message sequence might be infinite. This change makes the algorithm feasible, even if still it does not solve the problems of the number of operations and the storage of the path metrics. These problems are studied in the next chapter.

The use of a finite D is justified because it is observed that, given sufficient time, all paths in the trellis tend to converge toward one, unique path [1], [6]. If this path can be identified, then it can be decoded even when the complete path is not known. What is done in the algorithm is, once D is full, the minimum distance path so far is traced back, and its oldest branch is decoded. If D is deep enough, all paths in the trellis will converge to this branch with high probability, and the decoded symbol will be correct.

It should be remarked that it is not possible to make the probability that the paths will converge equal to one with finite D. When the paths do not converge to the same branch,

there is a probability that the wrong branch will be reached. If this happens, an error will be made by the algorithm. Because of this situation, the implementable algorithm is not optimum, in the sense that it has a probability of bit error, P_b, larger than that of the theoretical algorithm.

The memory D is used in a circular manner; that is, the memory space used by the branch just decoded is used to store the distance information of the next branch received. Two pointers are needed to address the memory: one that points to the start, and another that points to the end of the memory. In the algorithm, variable *start* is used to point at the beginning of the trellis, and t is used to point at its end. Variable *Dsize*, which corresponds to the size of memory D, is needed to wrap the pointers when they reach the end of the memory. Carry bits are discarded.

What follows is the algorithm modified as described above.

Definitions

Let:

D an array as defined above, but of finite size

Dsize the number of columns of D. The number of columns corresponds to the number of time iterations that can be stored in D

mod(x, y) a function that returns x *modulus* yIt is used to wrap the memory pointers to the number of columns of D

$R = r_1, r_2, r_3, \ldots$ the sequence of received, noisy symbols that are to be decoded

U the set of states of the convolutional coder

T a trellis, defined as the pair (S, g_T) where

Se U x U is a set of ordered pairs (x, y), specifying that state x is connected to state y. That is, (x, y) is a branch in the trellis

g_T: S -> $\mathbf{R}^n$ is a function that takes an element of S and returns the vector that corresponds to the output of the coder for that element. This vector can be seen as a label associated to branch (x, y), and as such is frequently called **branch label**

M a vector, where the element in position x is the accumulated distance for the path that ends in state x. It is initialized to zero

M_temp: a vector used to store temporal values, before storing them in M

D a two-dimensional array where the algorithm stores the paths it creates. Each column is a time iteration, and each row is a state. Element (w, z) contains the previous state for state w at time iteration z. It is initialized to zero

O is the output (decoded) sequence

|| is the concatenation operator. $x_3 \,||\, (x_2, x_1) = x_3, x_2, x_1$

Algorithm: Viterbi_Implementable(R, U, T);

Inputs: A sequence of noisy symbols

Output: A sequence of uncoded symbols

```
Viterbi_Implementable(R, U, T)
{
  start = 0;                     // pointer to start (oldest branch) of structure D
  t = 0;                         // t counts number of symbols in R
  while(! is_empty(R)) {         // repeat for all symbols in R, taken in sequence
    t = mod(t + 1, Dsize);       // update iteration counter
    r = getNext(R);              // get next element from R and remove it
    for (x : x e U) {            // find minimum distancebetween each state in coder
                                 // trellis and received symbol
      m = infinity;
      for (y : (y, x) e S) {     // find minimum distance between each branch that
                                 // ends in state x and received symbol
        d = distance(r, gT(y, x)) + M(y);
        if (d < m) {
            m = d;               // find and store minimum distance
            o = y;               // and state from which it originates
        }
      }
      M_temp(x) = m;             // New path distance is m
      current_state = D(minimum(M_temp), t);    // find shortest path
      D(x, t) = o;                        // store previous state in the trajectory
    }
    M = M_temp;                           // Update path distances
    if (full(D))   {                      // if decoder memory is full
      j = t;                              // j points to end of D (most recent branch)
      while(TRUE) {                               // repeat until start of D is reached
        next_state = D(current_state, j);         // find next state in path
        if (j = start) {                          // check if oldest branch has been
                                                  //  reached
          O = O || decode(next_state, current_state);      // decode symbol that
                                                           // corresponds to oldest
                                                           // branch in path
          start = mod(start + 1, Dsize);                   // update start of D
          break;
        }
        current_state = next_state;               // move back one step in D
        j = mod(j - 1, Dsize);                    // update pointer to D
      }
    }
  }
}
```

This algorithm is, in principle, attainable, because it only needs a finite, constant amount of memory.

2.3.6 A note on coding gain

It has been mentioned that a coding gain of about 7dB was to be expected when using convolutional coding. Since, strictly speaking, the modified Viterbi algorithm is no longer optimal (for the reasons mentioned above), some loss in coding gain is to be expected.

However, when choosing a proper value for D (around $5L$) in the algorithm above, the performance loss is so small as to be undetectable. This happens because the probability of choosing the correct path after about $5L$ iterations is very high.

In the next chapter, many of the variables affecting the algorithm are modified and the resulting behavior quantified. It is shown that a very efficient architecture, incorporating many changes to the algorithm, is attainable without much loss in error bit rate performance.

2.3.7 Punctured convolutional codes

A code of rate ½ takes one bit as input and outputs two bits. However, if the coder is modified so that every fourth output bit is not transmitted (i.e., it is deleted), then the code has been transformed into a 2/3 code. If two out of six output bits are removed, then the code has been transformed into a 3/4 code [26].

This technique for converting an $r=1/2$ code into a $r_1>1/2$ code is known as puncturing. The bit removal is not arbitrary; it follows rules known as puncturing pattern. This pattern must be known to the decoder. It has been shown that, in some cases, a punctured r_1 code has the same performance as a non punctured r_1 code. The punctured code can still be decoded by the 1/2 Viterbi decoder, provided that erasures are inserted in place of the removed bits.

Using this technique, the rate 1/2 decoder presented in this thesis may, with the addition of puncturing logic, be used to decode punctured convolutional codes.

2.3.8 Optimum codes

Through exhaustive, empirical searching, the best code vectors have been found for a number of code rates and constraint lengths [2]. For 1/2, $L=7$ codes like the one implemented in this thesis, the optimum connection vectors have been found to be (1001111)

and (1101101) [1]. In all simulations and results presented hereafter, a code that uses these vectors is assumed.

3 The Viterbi algorithm

3.1 Analysis of complexity

As mentioned previously, the time complexity of an algorithm is defined as the number of operations it must perform. The complexity of the Viterbi algorithm grows exponentially: whenever L is increased in one, the complexity doubles.

An implementable version of the Viterbi algorithm has already been presented. However, the question still remains whether it can be implemented at reasonable cost, and what speed (in decoded bits per second) it can attain. The degradation in bit error rate performance introduced by limiting the memory size has not been quantified, either.

The algorithm's performance depends on other variables besides L; the survivor path length and the metric storage method, for example. All these variables must be identified, and their effect on the performance of the algorithm quantified.

The aim of this analysis is to provide information on how the algorithm can be modified to achieve maximum bit rate, and what the effects are on bit error rate performance.

To carry out this analysis, a computer simulation model of the algorithm was built. The variables to change were identified, and a simulation test plan was developed to systematize the process. The simulation test plan is presented in Appendix C.

The complexity measurements were made taking into account that the objective is a hardware implementation. With this in mind, three kinds of operations were identified:

- *ckp* operations are those that take a clock period to complete. These operations include every *for* iteration in the algorithm, and operations that involve register transfers, or interchange of values between variables.
- *mem* operations represent accesses to the survivor path memory
- *alo* operations are arithmetic or logic operations

The Viterbi algorithm lends itself to parallelism, at least in some parts. In this section, however, parallelism is not considered; the concern is only to modify the algorithm in order to reduce the number of operations that need to be performed. Parallelism is addressed in the next chapter.

The analysis was made for an $L=7$, $r=1/2$ decoder, for an AWGN channel. QAM-4 modulation is assumed, with the signal constellation shown in figure 7.

Results are presented in graphics or tables, as appropriate. The analysis is usually performed for several values of E_b/N_0. Reference is made also to bit error probability, or P_b.

The total number of operations presented refers to the average per decoded bit. For decoding speed calculations, a clock frequency of 25MHz is assumed.

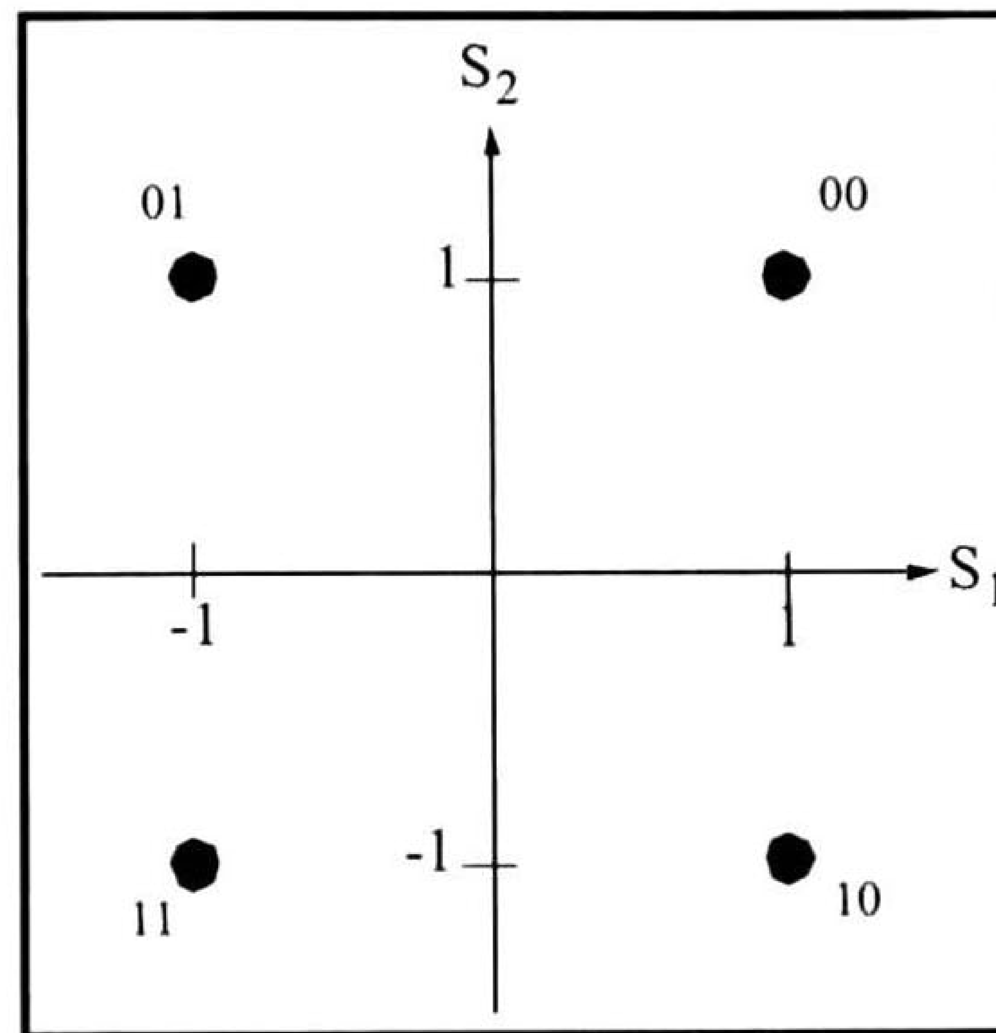


Figure 7. *Signal constellation for QAM-4*

3.2 Variables of the algorithm

It was determined that the following variables of the algorithm can be modified:

- distance normalization method
- size of survivor path memory
- traceback method

The distance metric associated to each path must be stored at all times. This distance grows constantly, and, since the registers that store it are of finite length, they will eventually overflow. An overflow is to be avoided, since it will turn what was a large distance into a very small one.

One way to avoid overflows is through **register normalization**[6]. This operation is very time consuming, since the usual way to do it is to find the smallest distance calculated so far, and then subtract it from all registers. There are other ways to do it, which are investigated below.

The survivor path memory length has a very clear impact on all aspects of decoding performance. On one hand, the larger the memory, the better the possibility that all paths have converged to the most likely path. On the other hand, a larger the memory means more silicon area and a longer traceback operation. The objective of this part of the analysis is to find the optimum survivor path memory length, as well as quantifying the cost of reducing this size.

There are, likewise, several methods to perform the traceback. The survivor path memory can also be organized in several ways. These aspects of the algorithm are also analyzed.

3.3 The unmodified algorithm

For comparison purposes, it is best to present first the results obtained with an unmodified version of the Viterbi algorithm. In order to minimize the effect of the survivor path memory length on the results, a length of 100 was chosen. Results are presented in figure 9 and table 1.

Just from the number of clock cycles required, it is seen that the maximum bit rate attainable by this algorithm is about 25MHz*bit/8678 = 2.88 kilobits per second. In the best case, when memory operations take one clock cycle, as well as all arithmetic and logic operations, the maximum bit rate is reduced to 2.82kbps.

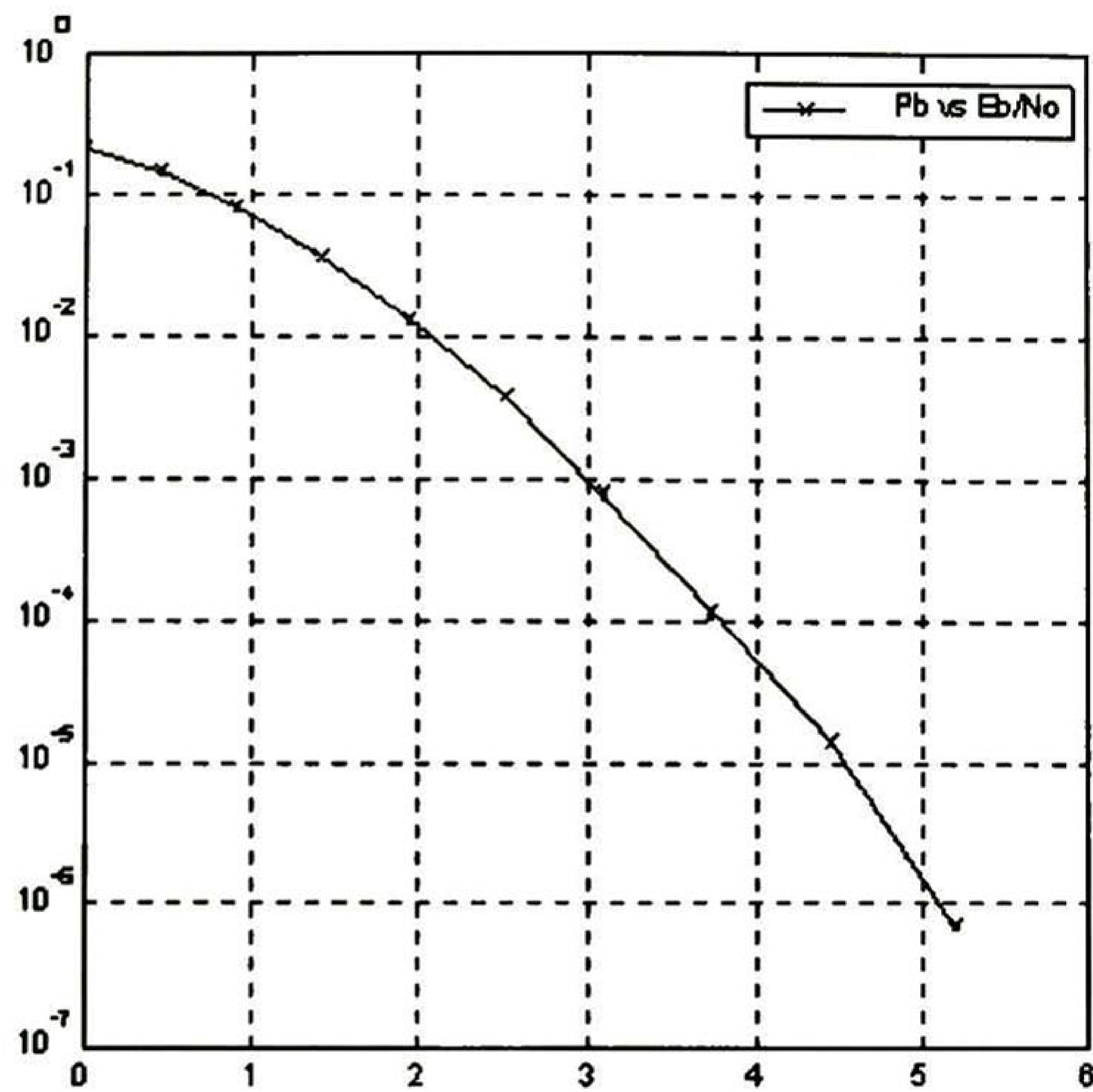


Figure 9 P_b vs. E_b/N_o

ckp	*mem*	*alo*
8678	164	8513

Table 1. Number of operations required for decoding one bit

It can be seen that the number of logic or arithmetic operations performed is not very significant. The major bottleneck in this algorithm is the raw number of clock cycles needed. However, the number of memory accesses must be taken into account also, since they can take a significant amount of time to complete, and could potentially turn into another bottleneck.

The algorithm must be optimized to reduce these two numbers.

3.4 Distance normalization

The path metrics that are calculated by the algorithm have to be stored in registers (the path metrics are variables M in the algorithm Viterbi_Theoretical, described in section 3.3.2). These values tend to grow as time passes, causing overflows in the registers. Register overflow causes the stored value to be inaccurate and smaller than it was, i.e., 255+2=1 if 8-bit registers are used. This problem is serious because the smallest path metric for each state has to be chosen at each iteration; this smallest metric cannot be found if there has been an overflow.

Figure 10 shows how the path metric for state 0 evolves for iterations 100,000 to 101,000 of the algorithm. Simulation shows that path metrics for all states behave similarly.

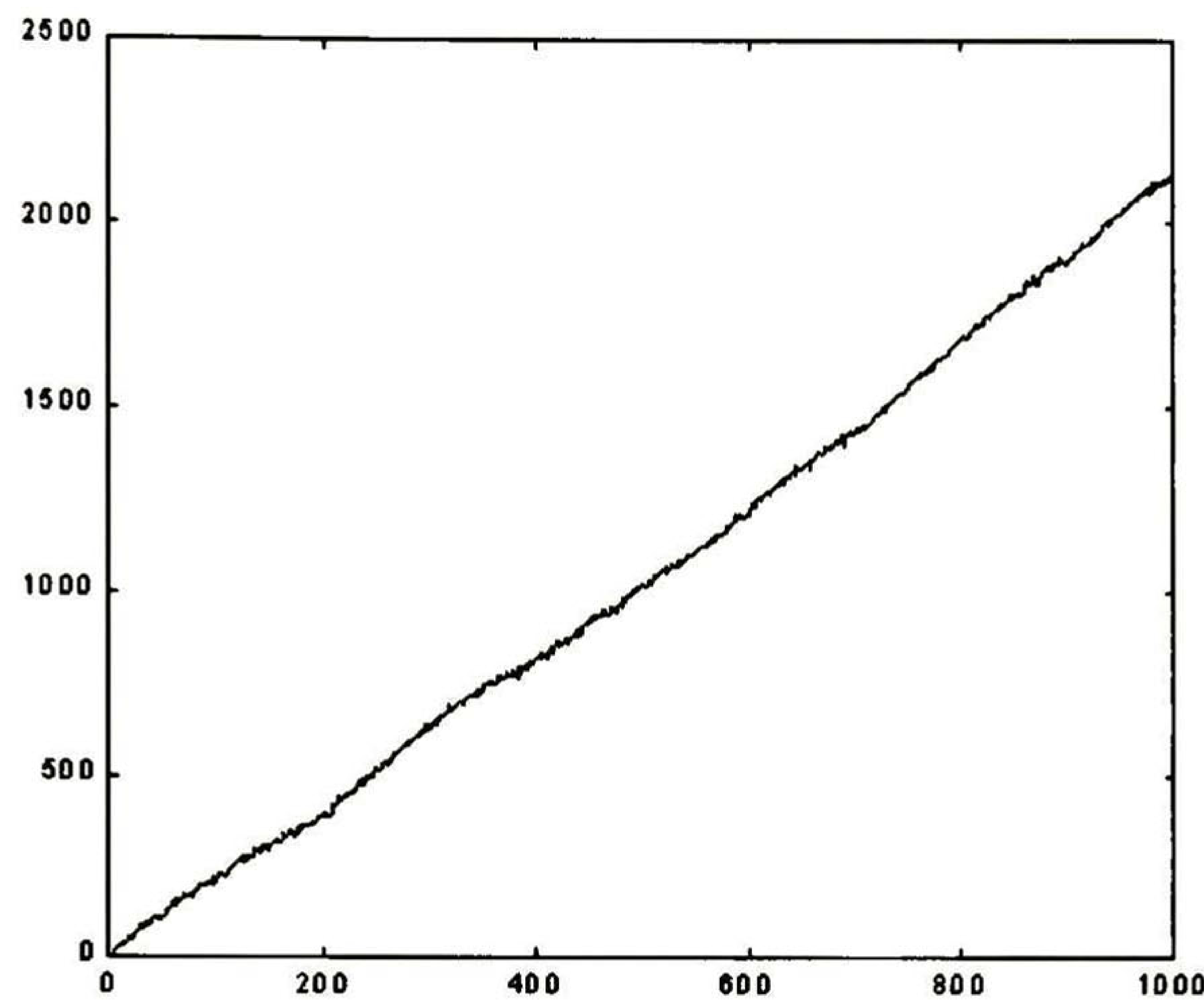


Figure 10. *Path metric for iterations 100,000 to 101,000*

It can be seen that the metrics grow constantly, and register overflow will eventually occur.

Several methods have been proposed to solve this problem. These methods are [6]:

Method 1: to normalize the path metric registers every time iteration

Method 2: to normalize the path metric registers every given number of iterations

Method 3: to use two's-complement arithmetic to avoid normalization

Method 2 can be seen as a generalization of method 1. Both have the advantage of keeping the path metrics within a range. One disadvantage is that, if the path metrics diverge widely from one another, then normalization alone might not be enough to avoid an overflow. In general, path metric behavior is not a well understood subject [7], except for a few, restricted cases. Simulation has not proven to be a reliable method for estimating path metric behavior over long periods of time. Another disadvantage is that the operations involved (comparison and subtraction) are very expensive in terms of silicon area and time necessary for completion.

The disadvantages of method 2 have as consequences unpredictable bursts of errors when a register overflows, and a slowdown of the algorithm because of the time needed to normalize all registers. No solution has been found to the overflow problem; it can only be

suggested to make the path metric registers large, and to normalize often, to minimize the probability that it happens.

The Viterbi algorithm, modified to normalize path metrics according to method 2, can be described as follows. Most definitions and comments have been omitted, to avoid repetition.

Definitions

Let:

NUM be a number that indicates how often to normalize the registers. *NUM* equal to one means to normalize every iteration

Algorithm: Viterbi_Implementable_Normalizing(R, U, T);
Inputs: A sequence of noisy symbols
Output: A sequence of uncoded symbols

```
Viterbi_Implementable_Normalizing(R, U, T)
{
   start = 0;                       // pointer to start (oldest branch) of structure D
   t = 0;                           // t counts number of symbols in R
   norm_counter = 1;                // counter used to know when to normalize path
                                    // metrics
   while(! is_empty(R)) {           // repeat for all symbols in R, taken in sequence
      t = mod(t + 1, Dsize);        // update iteration counter
      r = getNext(R);               // get next element from R and remove it
      for (x : x e U) {             // find minimum distance between each state in coder
                                    // trellis and received symbol
         m = infinity;
         for (y : (y, x) e S)       // find minimum distance between each branch that
                                    // ends in state x and received symbol
         {
            d = distance(r, g_T(y, x)) + M(y);
            if (d < m)
            {
                  m = d;            // find and store minimum distance
                  o = y;            // and state from which it originates
                  if (norm_counter = 0) {        // if it s time for normalization
                     m = m - minimum(M);         // subtract minimum metric
                  }
            }
         }
         M_temp(x) = m;             // New path distance is m
         current_state = D(minimum(M_temp), t);       // find shortest path
         D(x, t) = o;                          // store previous state in the trajectory
      }
      norm_counter = mod(norm_counter + 1, NUM);   // update norm_counter
      M = M_temp;                              // Update path distances
      if (full(D))    {                        // if decoder memory is full
         j = t;                                // j points to end of D (most recent branch)
         while(TRUE) {                            // repeat until start of D is reached
            next_state = D(current_state, j);     // find next state in path
            if (j = start) {                      // check if oldest branch has been
                                                  //  reached
               O = O || decode(next_state, current_state);    // decode symbol that
                                                              // corresponds to oldest
                                                              // branch in path
               start = mod(start + 1, Dsize);                 // update start of D
               break;
            }
            current_state = next_state;           // move back one step in D
            j = mod(j   1, Dsize);                // update pointer to D
         }
      }
   }
}
```

As can be seen, the algorithm requires first to find the minimum element of a vector, and then to subtract this number from all elements of another vector. Both comparison and subtraction, while conceptually simple, require large area in silicon and are slow.

The method, however, appears to be efficient, as shown in figure 11. This figure shows the same path metric as figure 10, but using the normalizing algorithm shown above, with *NUM* equal to one.

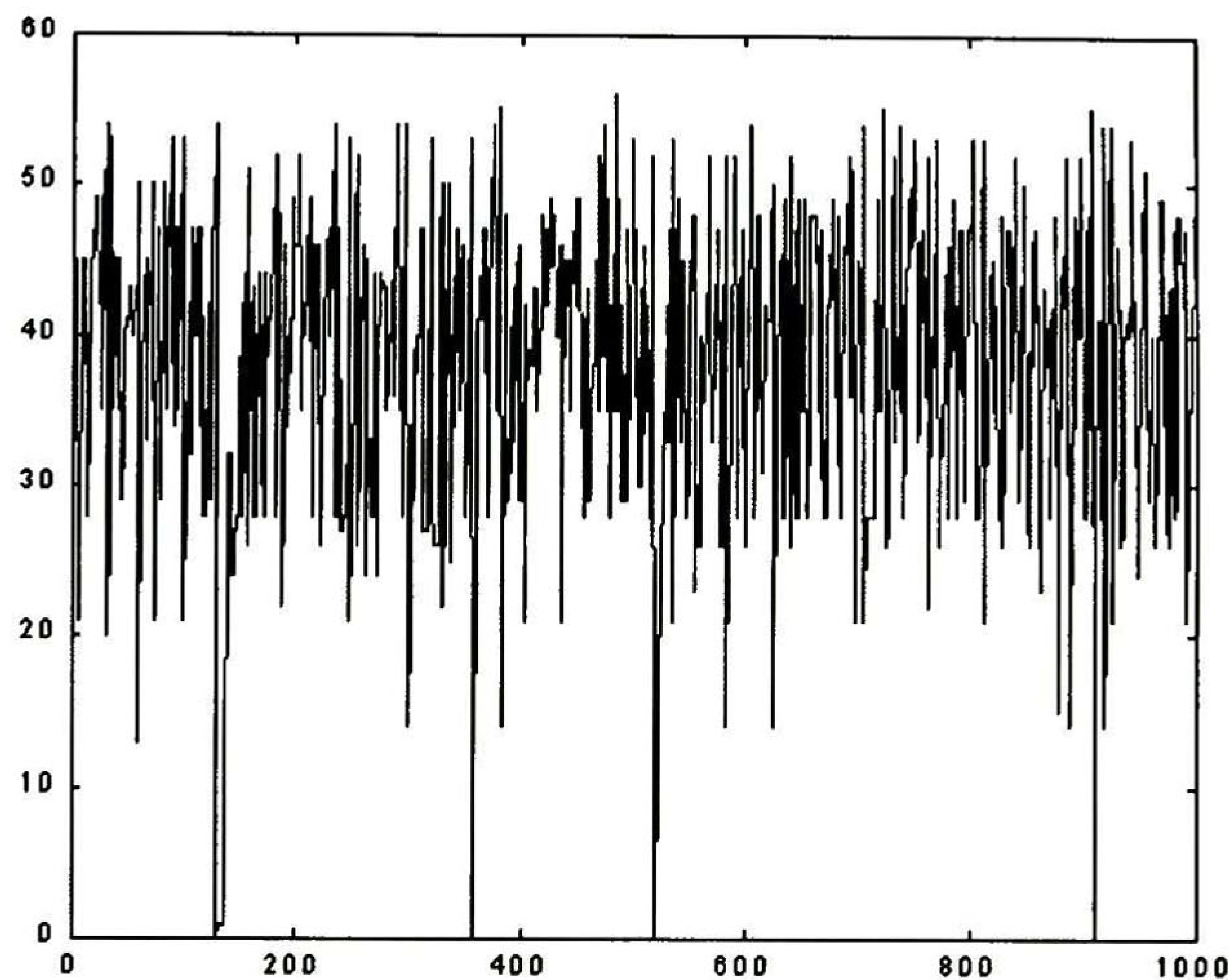


Figure 11. *Normalized path metric for iterations 100,000 to 101,000*

The path metric now appears to be bounded. Simulation of P_b vs. E_b/N_o shows no degradation from that of figure 9.

The value of variable *NUM* is related to the size of the registers. The larger the registers, the longer they can be allowed to grow without normalizing. A large value of *NUM* has some benefits regarding the time the normalization process takes, and if *NUM* is properly chosen, no degradation in performance is observed in P_b vs. E_b/N_o performance. Table 2 shows the operations required per decoded symbol for *NUM* equal to 10.It can be seen that the reduction in the number of operations performed is significant. It is 84% for clock periods, and almost 86% for the arithmetic operations. Using the same criteria as before, we can expect a decoding speed of about 16kbps, which represents an increase in a factor of 5.8 over

the unmodified algorithm.

ckp	*mem*	*alo*
1363	164	1197

Table 2. Number of operations required for decoding one bit, *NUM*=10

Method 3 [6] of path metric normalization is different from method 2 in that it doesn't attempt to avoid overflows. A different method of comparison is used to determine the smallest path metric in spite of an overflow. Let m_1 and m_2 be two path metrics to be compared, and D_{max} be the maximum difference between them. D_{max} is determined empirically, by simulation. Let the path metric registers be, at least, of size $2D_{max}$. Then, m_1 is less than or equal to m_2 if the most significant bit of m_1 m_2, in two's complement arithmetic, is equal to one; otherwise, $m_1 > m_2$. Following is the Viterbi algorithm modified according to this method.

Definitions

Let:

M a vector, where the element in position *x* is the accumulated distance for the path that ends in state *x*. Elements of *M* may not be larger than $2D_{max}$. *M* is initialized to zero

MSB(*x*) a function that returns the most significant bit of binary number *x*

sum2c(*x*, *y*) a function that returns $x+y$, in 2's-complement arithmetic

Algorithm: Viterbi_Implementable_2's_complement(*R*, *U*, *T*);
Inputs: A sequence of noisy symbols
Output: A sequence of uncoded symbols

```
Viterbi_Implementable_2's_Complement(R, U, T)
{
    start = 0;                       // pointer to start (oldest branch) of structure D
    t = 0;                           // t counts number of symbols in R
    while(! is_empty(R)) {           // repeat for all symbols in R, taken in sequence
      t = mod(t + 1, Dsize);         // update iteration counter
      r = getNext(R);                // get next element from R and remove it
      for (x : x e U) {              // find minimum distancebetween each state in coder
                                     // trellis and received symbol
        m = 2D_maxi
        for (y : (y, x) e S) {       // find minimum distance between each branch that
                                     // ends in state x and received symbol
          d = mod(distance(r, g_T(y, x)) + M(y), 2D_max);
          if (MSB(sum2c(d, m)) = 1) {              // if d<m
              m = d;                 // find and store minimum distance
              o = y;                 // and state from which it originates
          }
        }
        M_temp(x) = m;               // New path distance is m
        current_state = D(minimum(M_temp), t);      // find shortest path
        D(x, t) = o;                       // store previous state in the trajectory
      }
      M = M_temp;                          // Update path distances
      if (full(D))   {                     // if decoder memory is full
        j = t;                       // j points to end of D (most recent branch)
        while(TRUE) {                             // repeat until start of D is reached
          next_state = D(current_state, j);       // find next state in path
          if (j = start) {                        // check if oldest branch has been
                                                  //  reached
            O = O || decode(next_state, current_state);     // decode symbol that
                                                            // corresponds to oldest
                                                            // branch in path
            start = mod(start + 1, Dsize);                  // update start of D
            break;
          }
          current_state = next_state;             // move back one step in D
          j = mod(j   1, Dsize);                  // update pointer to D
        }
      }
    }
}
```

It should be noted that the modulus operation does not involve any real operation in hardware. When two quantities are added, ignoring the carry bit, and the result is stored in

a register of n bits, the quantity stored is the sum modulus 2^n

This method avoids register normalization, and at the same time provides for a way to perform the comparison operation using only arithmetic operations. Table 3 shows how this method performs in comparison with method 2.

ckp	*mem*	*alo*
422	164	321

Table 3. Number of operations required for decoding one bit, two's complement arithmetic

This method proves to be even better than method 2, with reductions of 95% in the number of clock periods needed to decode one symbol and 96% in the number of arithmetic operations. Now, the expected decoding speed is increased to around 43kbps, which represents a factor of about 15 with respect to the unmodified algorithm.

Both methods share the same disadvantage: they depend on knowledge about the path metrics behavior. This knowledge cannot be obtained, given the current state of research. The best approach is, then, to simulate and be conservative with the results. This problem will be addressed further in chapter 4. Given this situation, method 3 shows much better performance, and requires less silicon area, than method 2, and so it is the method selected for implementation.

3.5 Survivor path memory modifications

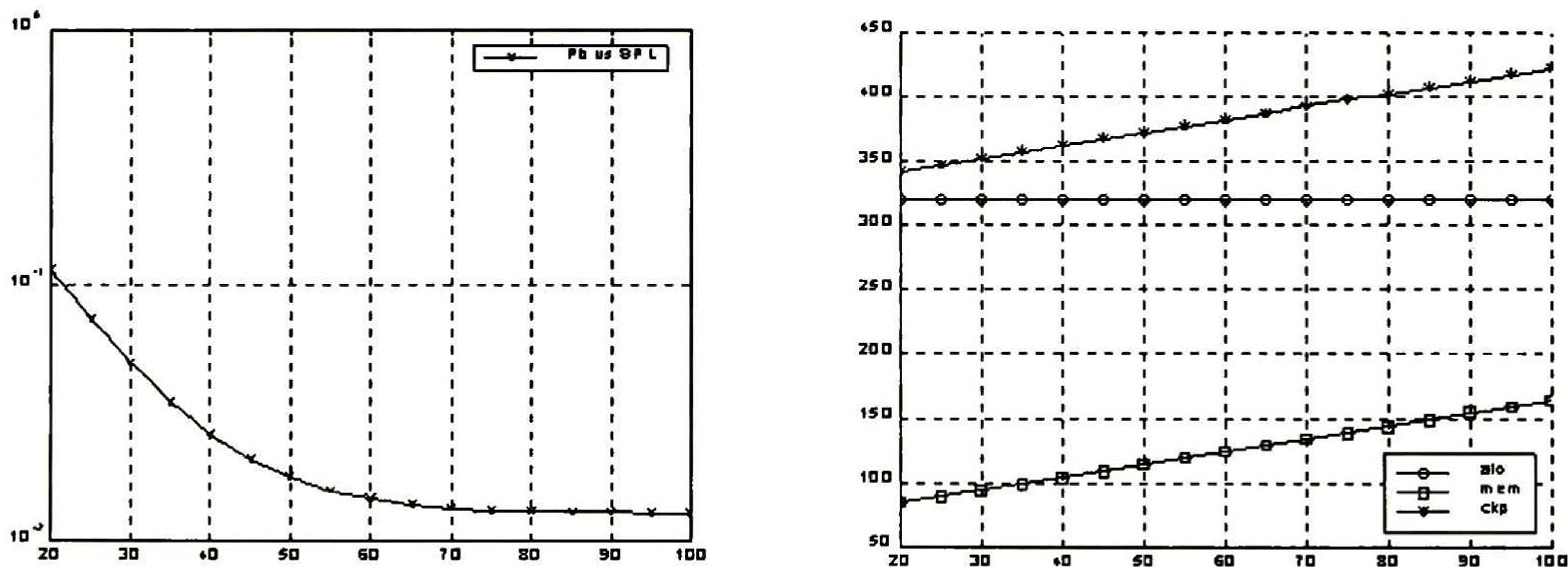
There are two points of the Viterbi algorithm related to the survivor path memory that can be modified to improve the decoding speed. One is the survivor path memory length: how small can it be made, so as to improve the traceback speed while maintaining the same, or close to the same, bit-error performance? The other is, if it will be assumed that all the paths will converge at some point in the survivor path memory, then it should be possible to decode several branches beyond the point of convergence, and in this way to decode several branches in a single traceback operation. These two points are addressed in what follows.

3.5.1 Survivor path memory length

The method used to evaluate this aspect of the algorithm was to fix E_b/N_o, and find P_b for various memory lengths. Five values of E_b/N_o were used: 1.93dB, 2.5dB, 3.1dB, 3.74dB, and 4.44dB. These values correspond to noise power s_o^2=0.8, 0.75, 0.7, 0.65, and 0.6, respectively, for the signal shown in figure 7.

The number of operations required in each case vary, because as the memory is made shorter, fewer memory accesses are required during traceback. However, a cost is paid in performance, since a shorter memory lowers the probability that all paths will converge, and so the probability of decoding the correct message is reduced. This is a situation where a compromise must be made between bit-error probability and decoding speed.

Results are shown in figures 12 and 13. Survivor path memory length is called *SPL*.



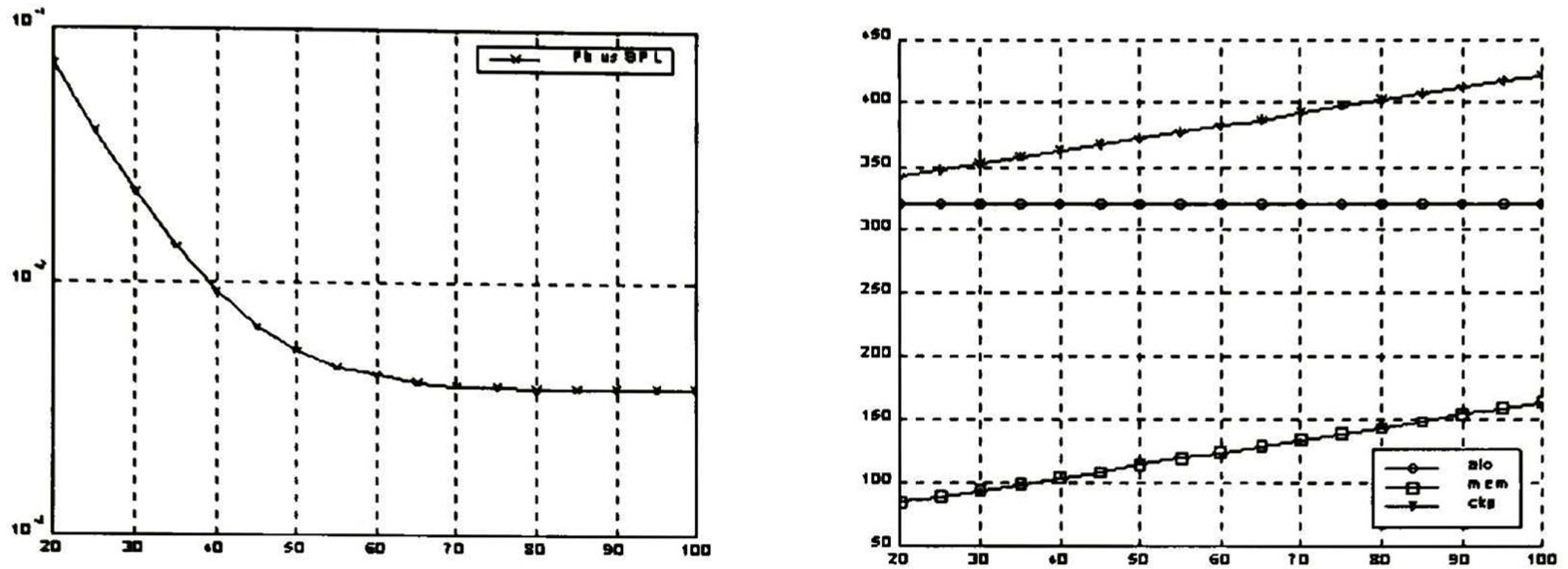
Figures 12 and 13. *SPL vs. P_b SPL vs. operations, E_b/N_o = 1.93dB*

It can be seen that bit-error performance is heavily dependent on *SPL* for E_b/N_o = 1.93dB. It starts to degrade for *SPL*=85 or so. This is to be expected, since noise power is large compared to signal power, and this will cause many received errors. This situation slows down path convergence. Also, it should be noted that the degradation of performance is rather sharp from SPL less than approximately 60. After 60, the rate of degradation is slow.

On the other hand, it can be seen that the required number of clock periods, as well as the required number of memory accesses, have a linear dependence *SPL*. The number of

memory accesses almost halves, from 164 to 84, while the number of clock periods goes from 422 to 342, a decrease of 19%. The number of arithmetic operations remains unchanged.

For E_b/N_o = 2.5dB, results are presented in figures 14 and 15.



Figures 14 and 15 *SPL vs. P_b SPL vs. operations, E_b/N_o = 2.5dB*

Figure 14 shows that P_b does not change significantly for *SPL*>70. The performance of the algorithm degrades very rapidly for *SPL*<70.

On the other hand, figures 13 and 15 are identical. The number of operations required per decoded branch are the same, regardless of the noise power. This is to be expected, as it can be seen in the description of the algorithm that the number of operations performed do not depend on the number of errors in the sequence. For this reason, only *SPL* vs. P_b results will be presented in what follows.

Figures 16, 17 and 18 show the results for E_b/N_o = 3.1dB, 3.74dB, and 4.44dB, respectively.

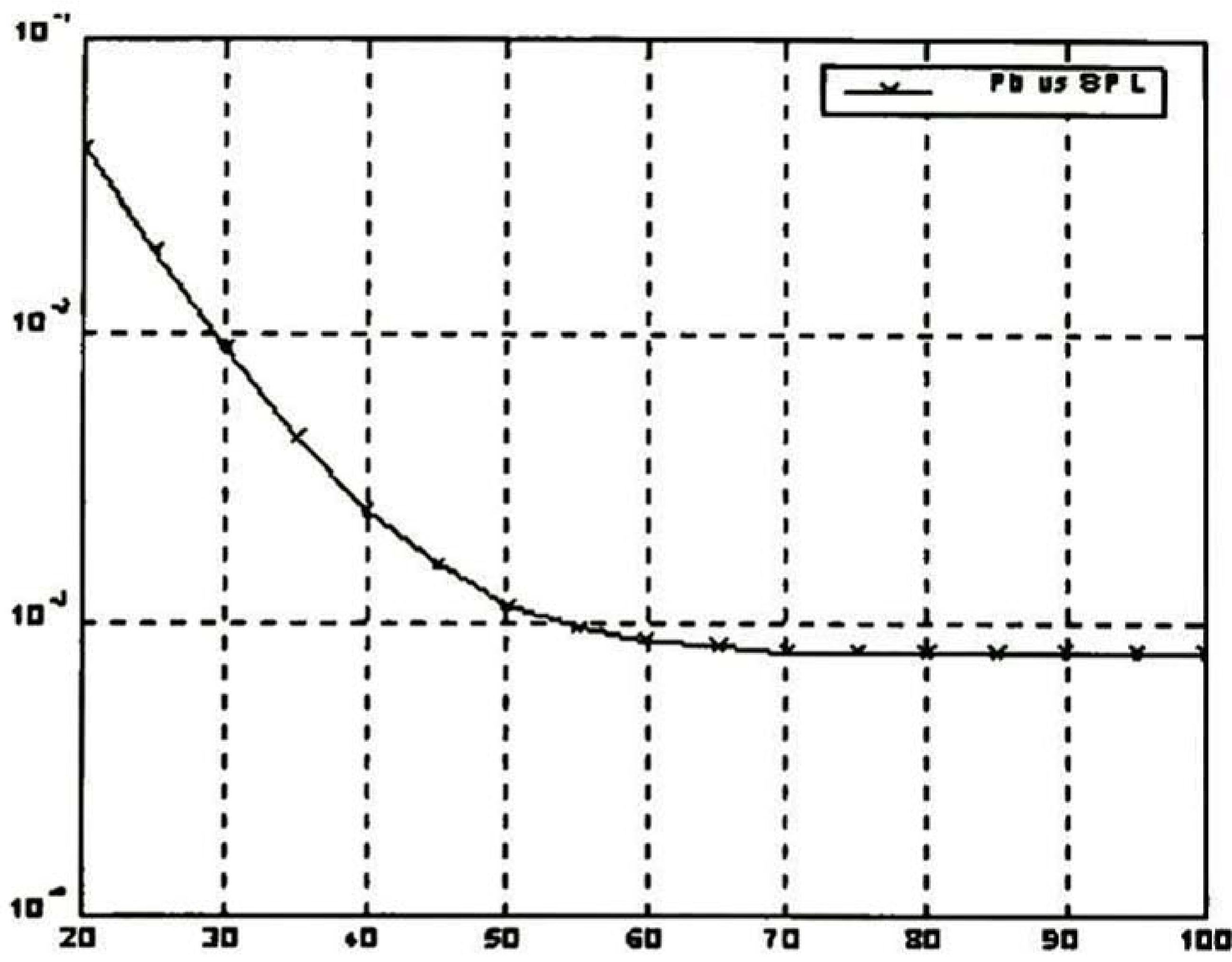


Figure 16. *SPL vs.* P_b E_b/N_o = 3.1dB

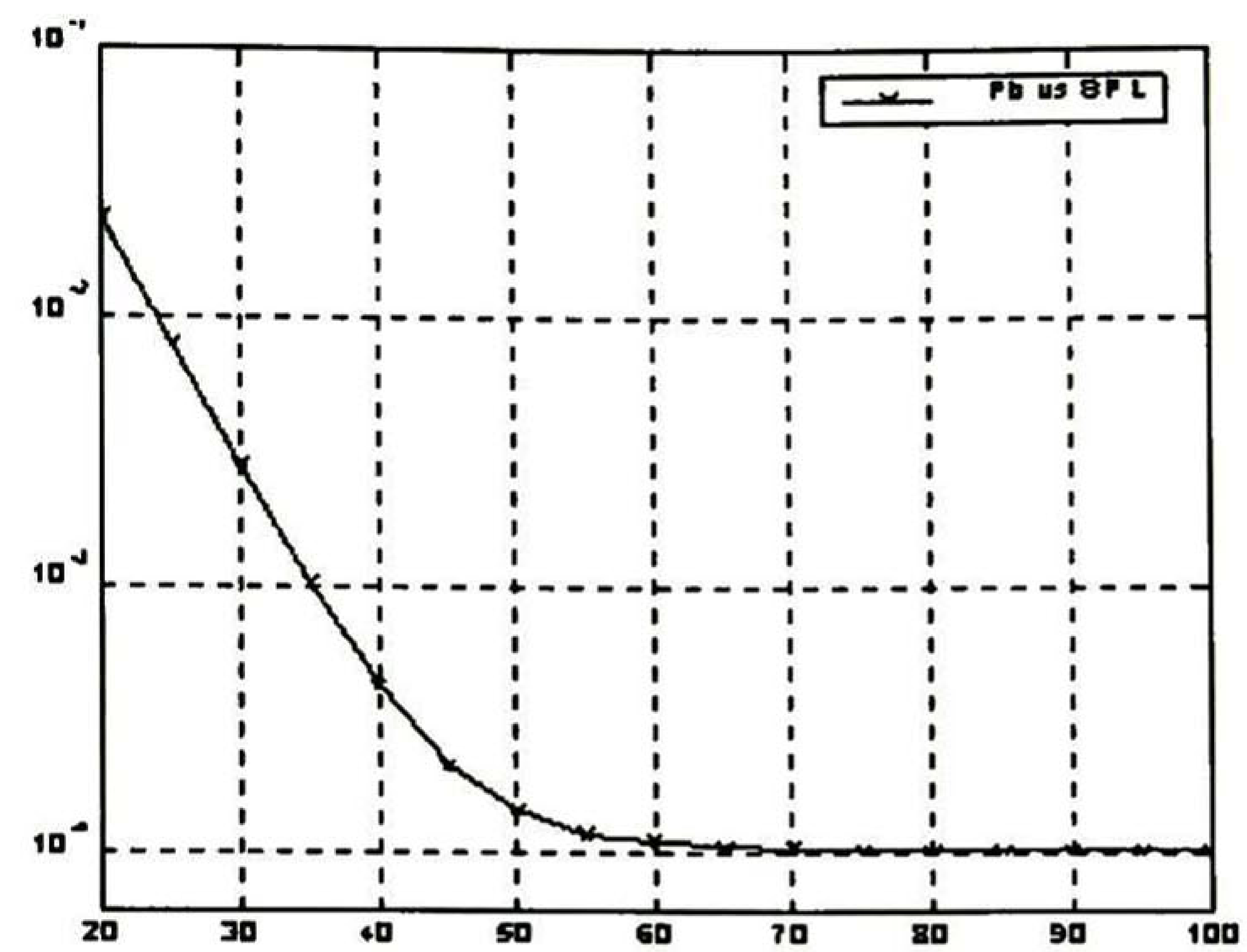


Figure 17. *SPL vs.* P_b E_b/N_o = 3.74dB

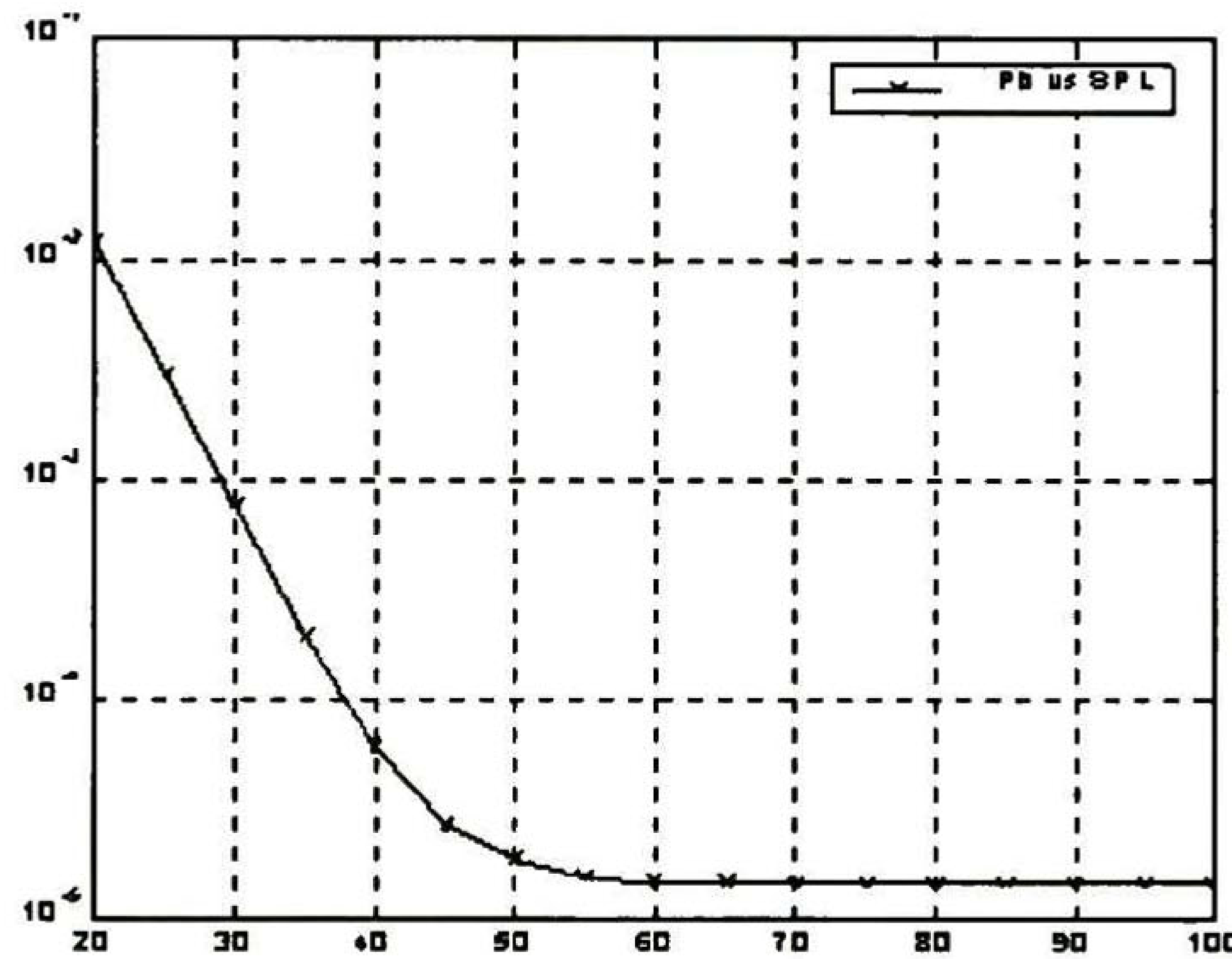


Figure 18. *SPL vs.* P_b E_b/N_o = 4.44dB

Several conclusions can be drawn from these results. First, it can be seen that P_b does not degrade slowly with *SPL*; if *SPL* is lower than a threshold, then P_b degrades very rapidly. Above the threshold, P_b does not improve significantly. The consequence is that it is not feasible to trade memory for bit-error rate performance; the best course of action is to choose *SPL* as close to the threshold as possible.

Second, the number of arithmetic or logic operations does not depend on *SPL*, as expected from the algorithm. The number of clock cycles, as well as the number of memory accesses, depends linearly on *SPL*. Going from a memory length of 100 to a length of 20 decreases the number of clock cycles from 422 to 342 (about 19%), and the number of

memory accesses from 164 to 84 (about 51%). The consequence is that this modification is better for reducing the number of memory accesses than the other two types of operations.

Real communications systems rarely operate at P_b more than 10^{-3} The reason is that even the most forgiving applications, like voice, become too unreliable to be of any use for this quantity of errors. For the Viterbi decoder being proposed (r=1/2, L=7), P_b=10^{-3} is achieved, theoretically, at E_b/N_o approximately equal to 3dB.

If it is assumed that the Viterbi decoder being proposed is not going to be used in applications where E_b/N_o is less than 3dB, then it can be concluded that the *SPL* used should be around 70. If memory length is less than 70, the bit-error rate starts to degrade.

Table 4 shows the number of operations required with *SPL*=70.

ckp	*mem*	*alo*
397	139	321

Table 4. Number of operations required for decoding one bit, *SPL*=70

Now, the expected bit rate is about 47kbps, which represents an improvement of about 9% over the previous algorithm optimization (section 3.4).

3.5.2 Number of branches decoded at a time

So far, it has been assumed that, after *SPL* iterations of the algorithm, all paths in the trellis converge to a branch found at the start of the trellis; that is, if all paths are traced back *SPL* branches to the start of the trellis memory, all paths will arrive at the same branch.

This idea can be exploited to reduce the number of memory accesses needed per decoded branch. If the minimum survivor path memory length, *SPL*, is increased by some quantity, q, then $q+1$ branches can be decoded in a single traceback operation. This is because if all paths are assumed to converge to the same branch in a memory of length *SPL*, then all paths will also converge to the oldest $q+1$ branches in a memory of length $SPL+q$.

On the other hand, the main disadvantage of this approach is that, in the event that

paths do not converge to a single branch, and the wrong branch is chosen during traceback, then a large number of errors will occur, instead of a single one. This could cause a large deterioration on the decoder's performance.

If an *SPL* of 70 is assumed, the results for Q=10, Q=20, and Q=30 are presented in the following figures and tables. In each case, the results are compared against those of the unmodified algorithm, presented in figure 9.

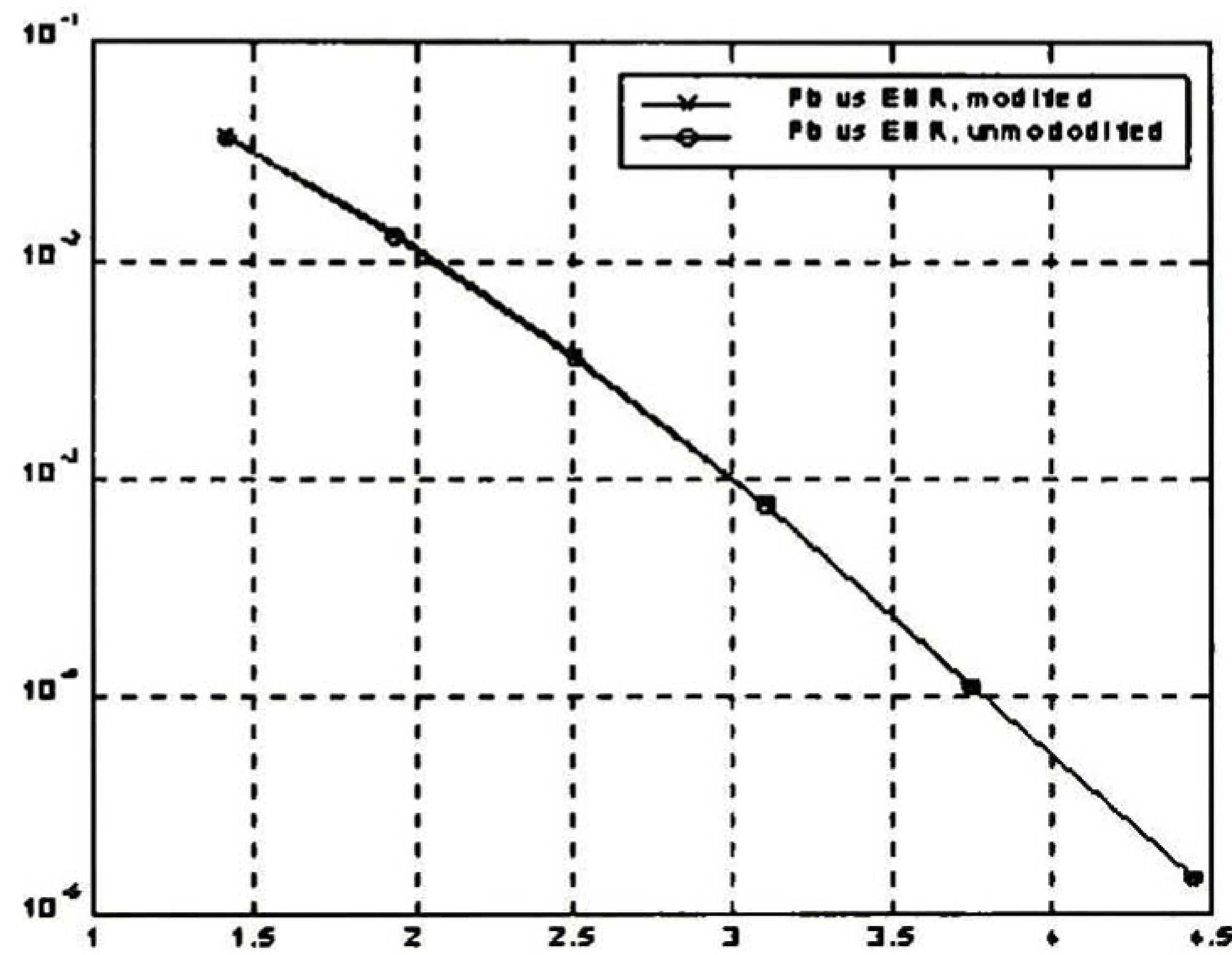


Figure 19. *E_b/N_o vs. Pb, SPL =70, Q=10*

ckp	*mem*	*alo*
330	72	321

Table 5. Number of operations required for decoding one bit, *SPL*=70, *Q*=10

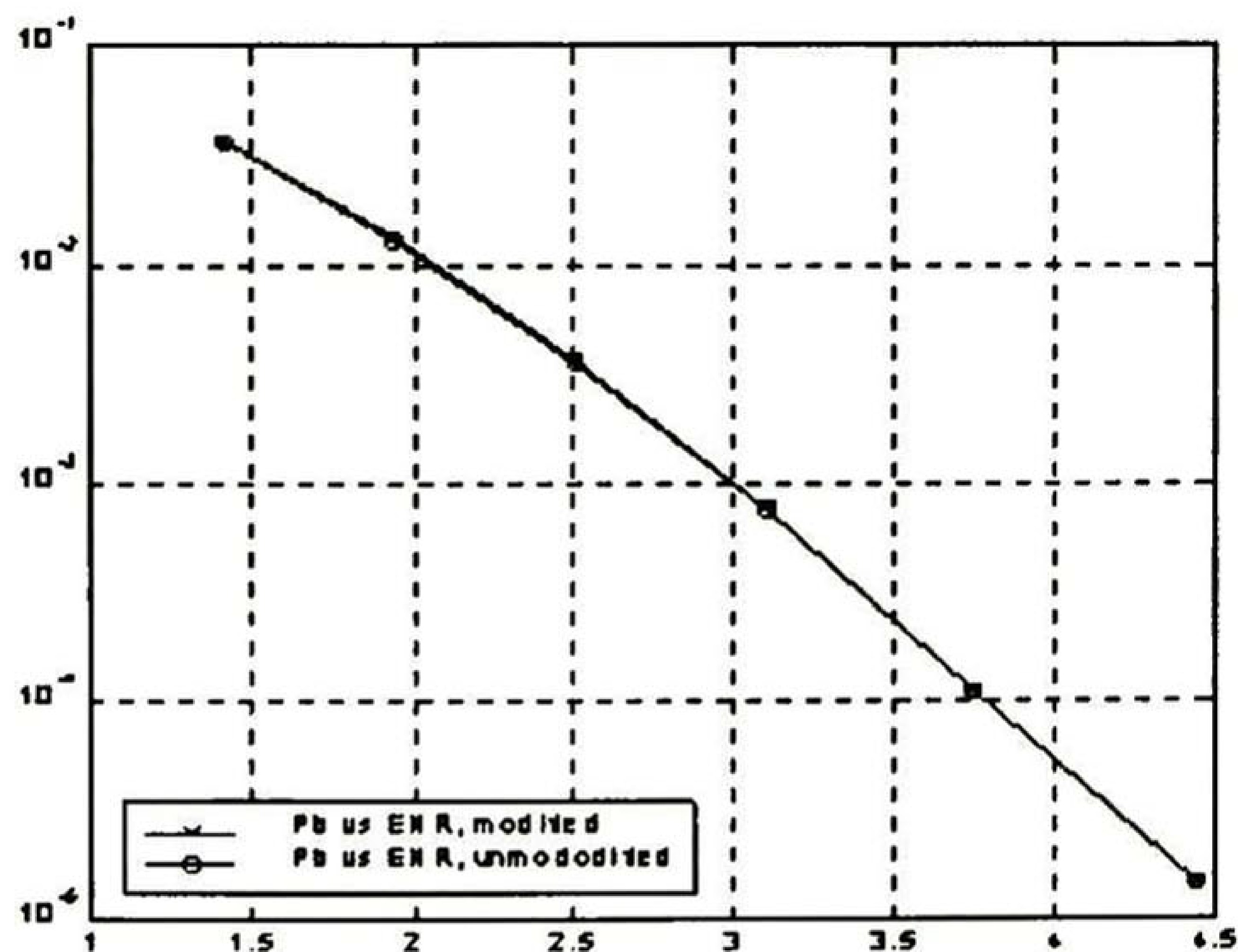
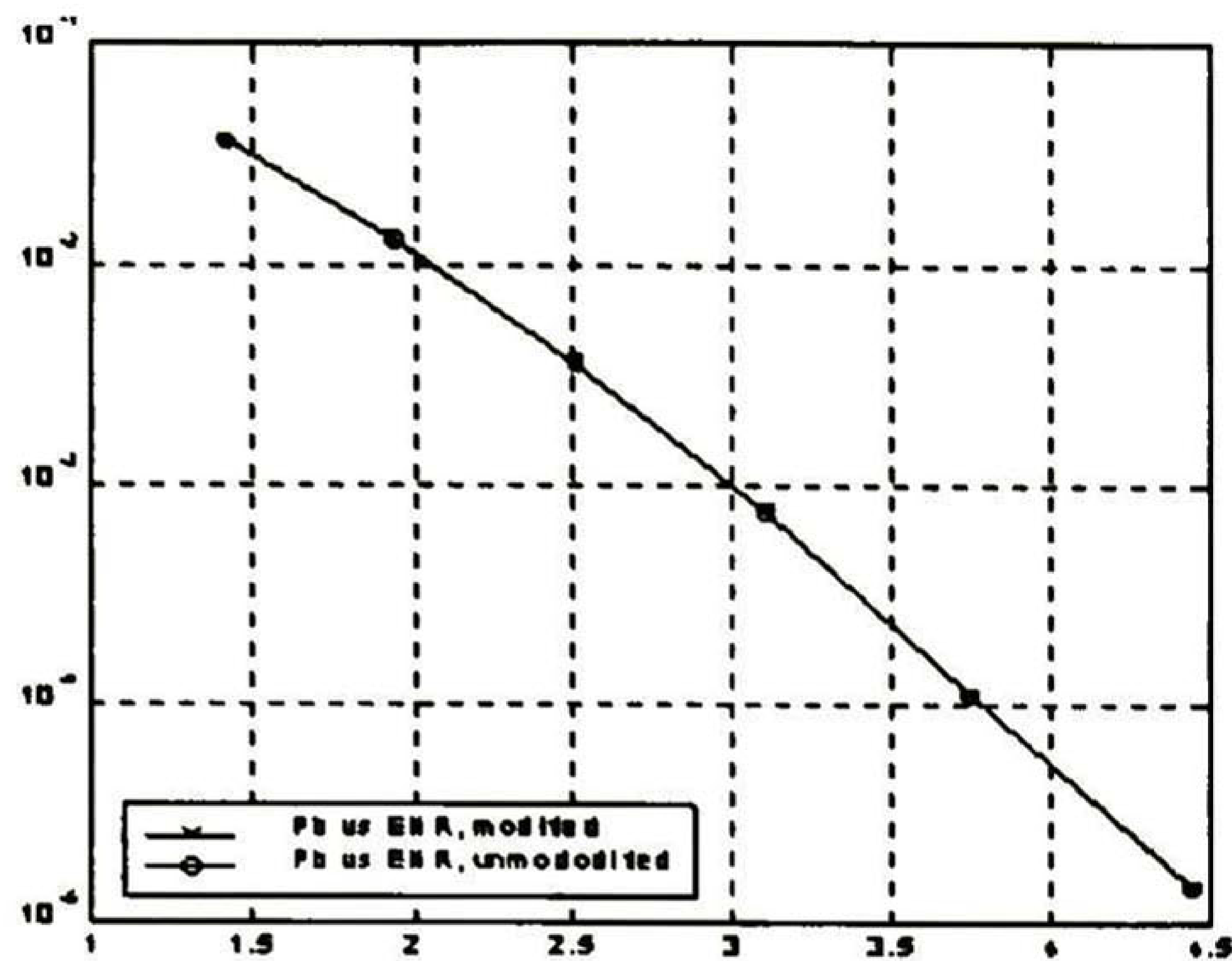


Figure 20. *E_b/N_o vs. Pb, SPL =70, Q=20*

ckp	*mem*	*alo*
327	68	321

Table 6. Number of operations required for decoding one bit, *SPL*=70, *Q*=20



ckp	*mem*	*alo*
326	67	321

Table 7. Number of operations required for decoding one bit, *SPL*=70, *Q*=30

Figure 21. *E_b/N_o vs. Pb, SPL =70, Q=30*

Figures 19 to 21 show that there is almost no deviation from the optimum in performance, especially at higher signal-to-noise ratios. The average number of memory accesses per decoded symbol, however, grows according to the formula:

$$mem = 64 + \left(\frac{SPL + q}{q} \right) \tag{5}$$

This formula tends to 64 as Q grows. 64 is the number of memory accesses required to store the previous state addresses in the trellis, because for $L=7$, there are 64 states.

3.5.3 Traceback method

So far the minimum *SPL* has been established. It has also been found that paths can be traced back and decoded with negligible loss in bit-error rate performance after this length. However, it remains to be determined whether a traceback architecture allows the decoding of one symbol per clock cycle, as required in the project objectives.

In fact, it is impossible to decode one symbol per clock cycle using traceback [9]. To see why, let us consider the diagram of the survivor path memory shown in figure 22.
The traceback algorithm effectively divides the survivor path memory in three blocks. The decode block (length Q) is where all paths have converged and data can be decoded. In the

merge block (length M) there are a multitude of possible paths. The idea is to use traceback in the merge block to find the initial state in the decode block, and then use traceback to decode the path. The write block (length W) is where new paths are written.

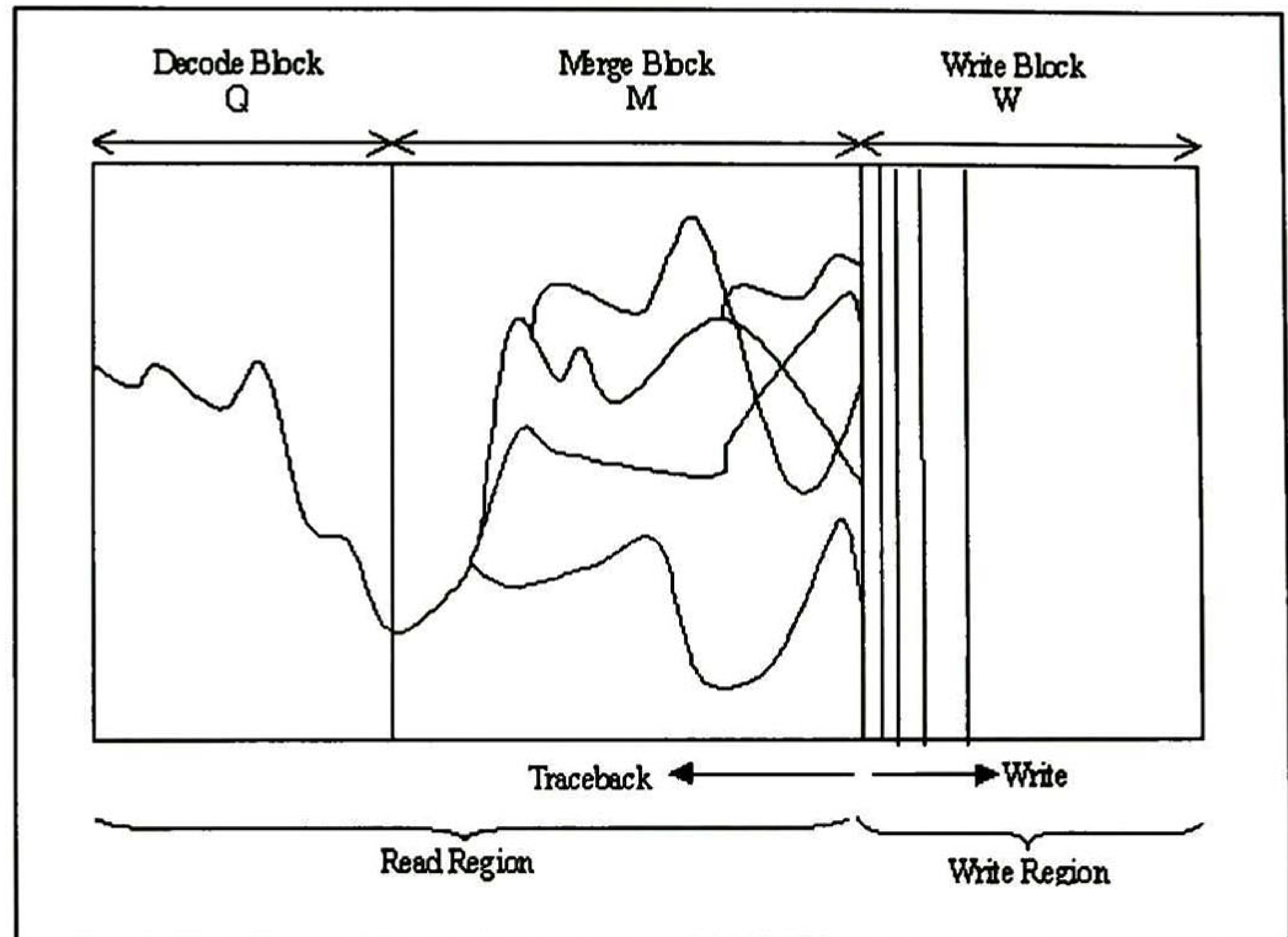


Figure 22. *Survivor Path Memory*

Once the path in the decode block has been decoded, this memory area is now used as the write block, a portion of the merge block turns into the new decode block, and so on. The memory is used in a circular, or wrapping, way. For this reason, the decode block and the write block must have the same length ($Q=W$).

The reason why it is impossible to decode one symbol per clock cycle with this scheme is that it requires the traceback of $M+Q$ branches, while it outputs only Q symbols. It is assumed that one branch can be read and another can be written per clock cycle.

Clearly, in the time it takes to read $M+Q$ branches, the same quantity of new branches must be written. The write area, however, has length of only Q. To avoid losing data a mismatch between read and write rates must exist, which prevents the circuit from decoding one symbol per clock cycle.

(To decode one symbol per clock cycle means that a single clock frequency is used, and that a symbol is admitted and another decoded per clock period).

If Q is made very large in comparison to M, then close to one symbol can be decoded per clock. However, expanding Q is very costly in terms of memory area required.

There are a number of modifications that can be made to the traceback algorithm to accommodate the decoding of one symbol per clock [5], [9], [10], [11]. All of them have one of the following disadvantages: they either require a larger survivor path memory, or they present extra complexity requiring a large amount of silicon area.

There are three main groups of techniques to improve the throughput of the traceback algorithm. They are briefly explained below.

***p*-pointer traceback.** This technique supports traceback of p paths concurrently, where each path occupies a different section of the survivor path memory, and is traced using an independent pointer. Since there is some overlap in the memory sections covered by each pointer, then the total amount of memory required is [5]:

$$D = M\left(2 + \frac{2}{p-1}\right) \quad for\ p > 1 \tag{6}$$

To this increase in memory length, it must be added the increase in logic overhead introduced by the multiple memory partitions and pointers.

Hybrid pre-traceback. This method works by applying look-ahead to the traceback. Instead of storing each branch in the survivor memory, it stores only every δ branch, in fact saving δ - 1 traceback steps. It uses a register-exchange network to calculate the branches.

Hybrid trace-forward. Trace-forward estimates the initial state of the decode block a priori, avoiding traceback of the merge block. To describe how it works, it is useful to define the tail of a survivor path. Given a time reference t, every trellis state at time $t+\delta$ has an associated survivor path that traces back to some state at time t. This state is referred to as $T^{S}_{m,\delta}$. For $\delta>M$, all tails should converge to the same state, and hence any tail can be used as an estimate of the initial state of the decode block.

The trace forward method is a recursion used to calculate the tails of survivor paths at the merge block at the same time they are being written to the memory. Associated with every state in the trellis is a register, which contains the tail state for that state. The tail

registers are initialized at time m, when the survivor paths are of length zero, and so the tail state and the current state are the same. The initial condition of the trace-forward method is then:

$$T^{S}_{m,0} = S$$

The trace-forward method consists updating the tail register of each state with the tail of its predecessor state, as follows:

$$T^{S}_{m,\Delta} = T^{S'}_{m,\Delta-1} \qquad (7)$$

As an example, let us consider the following trellis:

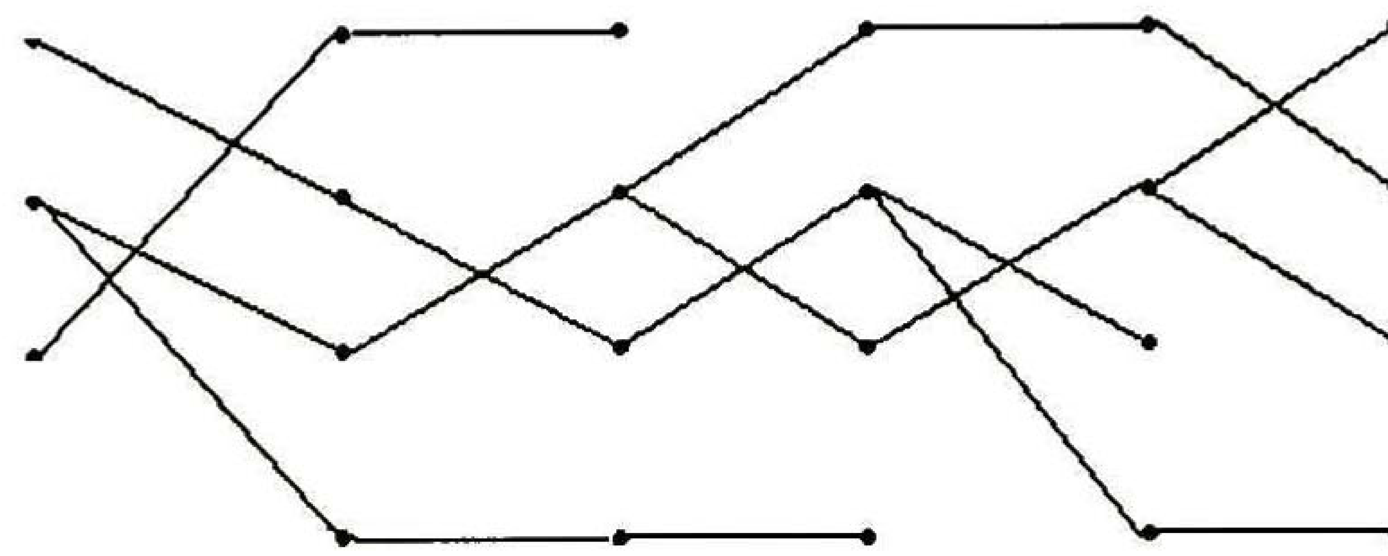


Figure 23. *An Example Trellis*

Now, let us calculate the evolution of the tail registers. At first, they are initialized with the current state, as follows:

$$T^{0}_{m,0} = 0$$
$$T^{1}_{m,0} = 1$$
$$T^{2}_{m,0} = 2$$
$$T^{3}_{m,0} = 3$$

Now, observing the surviving paths in the trellis, and according to recursion (7), the tail states for the next iteration should be:

$$T_{m,1}^{0} = 2$$
$$T_{m,1}^{1} = 0$$
$$T_{m,1}^{2} = 1$$
$$T_{m,1}^{3} = 1$$

The tail state for state 0 is now 2, because (as shown in the trellis) the branch connecting state 0 with state 2 has been chosen as the most likely. So, the tail state for the path that ends in state 0, and has length 1, is 2.

For the rest of the iterations, the evolution of the tail registers is as follows:

$$T_{m,2}^{0} = 2 \quad T_{m,3}^{0} = 1 \quad T_{m,4}^{0} = 1 \quad T_{m,5}^{0} = 1$$
$$T_{m,2}^{1} = 1 \quad T_{m,3}^{1} = 0 \quad T_{m,4}^{1} = 1 \quad T_{m,5}^{1} = 1$$
$$T_{m,2}^{2} = 0 \quad T_{m,3}^{2} = 1 \quad T_{m,4}^{2} = 0 \quad T_{m,5}^{2} = 1$$
$$T_{m,2}^{3} = 1 \quad T_{m,3}^{3} = 1 \quad T_{m,4}^{3} = 0 \quad T_{m,5}^{3} = 1$$

It is seen how, for this simple example, all tails have converged to the initial state 1. This is correct: it can be observed in the trellis that all surviving paths start, in fact, in state 1. So, state 1 can be used as an estimate to traceback and decode whatever paths are behind it, with confidence.

The area and memory requirements for these methods for enhancing traceback are summarized in table 8 [5].

Architecture	Memory Size (*SPL*)	Relative Area
2-pointer traceback	4M	1
3-pointer traceback	3M	1
hybrid pre-traceback	2M	0.66
hybrid trace forward	2M	0.59

Table 8. *Area estimates for traceback architectures*

Given that it uses less area and is conceptually very simple, the hybrid trace forward

architecture is proposed for implementation.

3.6 Memory Organization

There are two important issues to consider when planning the memory organization of the decoder. One is how the paths are going to be represented in memory; the other is how the memory will be built to allow the trace forward algorithm to work.

3.6.1 Path representation

Survivor path memory requires a very large amount of silicon area. For a 64-state decoder, with $SPL+Q=100$, the memory requirements are 200 branches times 64 states, times 6 bits, or 76,800 bits. (6 bits are needed to address the previous state in the trellis, and the memory required is $2M$ as per Table 8). This amount prevents all FPGA, and all but the most expensive ASIC implementations. It is clear that, in order to achieve a VLSI implementation at a reasonable cost, a reduction in the amount of memory required is indispensable.

(The other possibility, to use external RAM for storing the trellis, is expensive and slow, and does not comply with the requirements.)

The traceback algorithm can indeed be modified to use less memory for traceback. If the code rate is assumed to be 1/2, then it can be observed from the resulting trellis (see Figure 5) that each state leads to exactly two states, and therefore may have as predecessors only two definite states. It would seem that only one bit should be needed to address the previous state, because there are only two possible previous states. This bit is called **decision bit**

Given the structure of convolutional encoders, it is straightforward to calculate the previous state, given the current state and the decision bit. The previous state can be calculated as follows:

$$previous\ state = (current\ state << 1) || decision\ bit \qquad (8)$$

where "<<1" means left-shift, discarding the leftmost bit, and "||" is the concatenation operation as already defined above.

With this modification, the memory requirements are reduced from 76,800 bits to

12,800 bits, which makes implementation in an FPGA or small ASIC feasible. The bit-error probability is not increased because of this change, because the change only affects the way the survivor path memory is addressed.

3.6.2 Survivor path memory blocks

In order to implement the hybrid trace forward architecture, the memory is divided in two blocks (see Fig. 24).

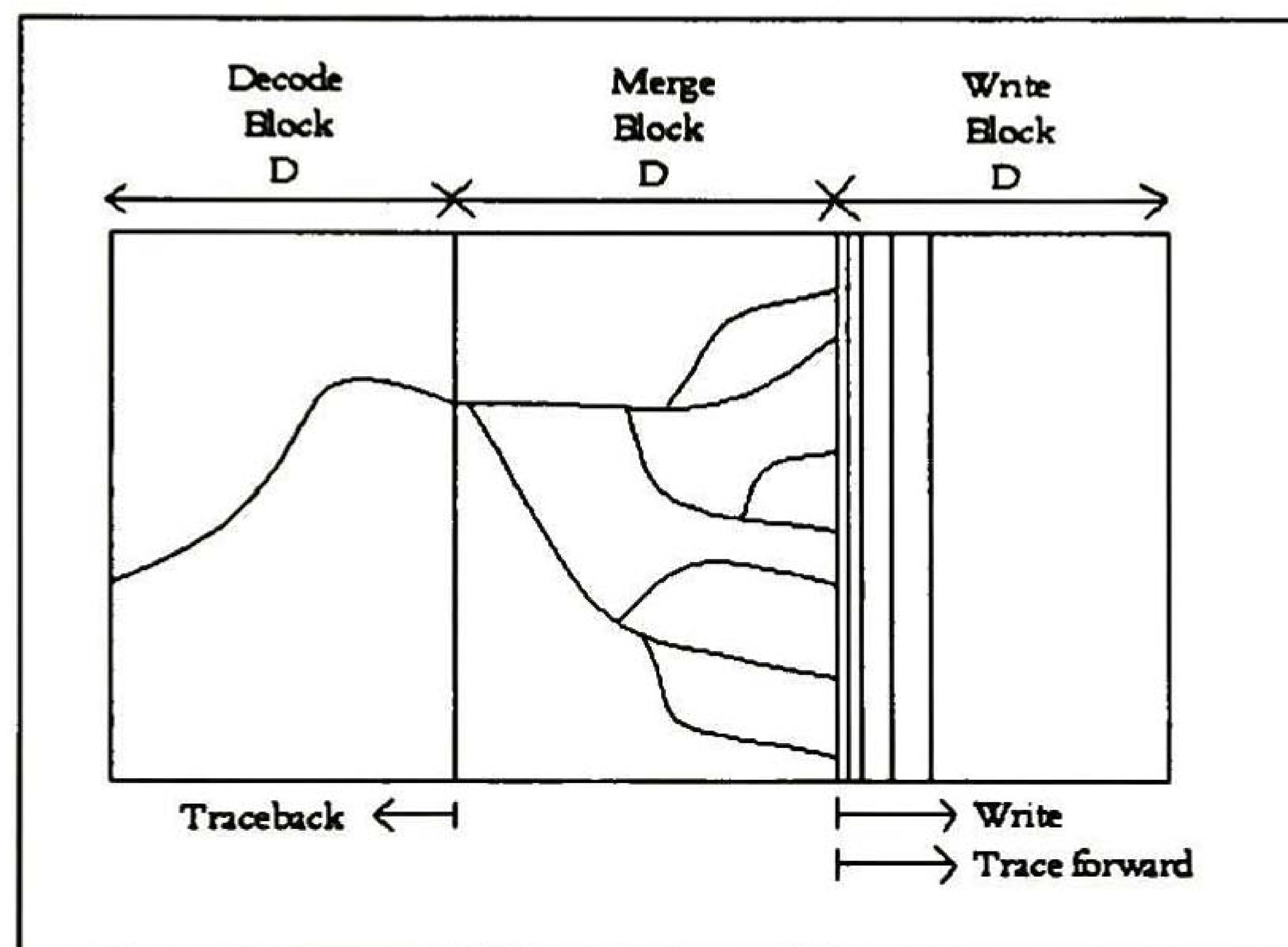


Figure 24. *Trace Forward Memory Organization*

For clarity, the memory is presented in Fig. 24 as having three blocks; however, because the read and write rates are the same, the write block can be folded onto the decode block, and thus the memory requirement is $2D$.

The sequence of events in the memory is as described in table 10. For simplicity, the two memory blocks have been simply labeled $B1$ and $B2$. Each row in the table corresponds to a change in the function of the memory blocks. The direction in which data is written or read from the memory is indicated with arrows.

It is observed that states 4 through 7 repeat continuously; that is, time step 8 is identical to time step 4, time step 9 to time step 5, and so on. This suggests the implementation of the memory controller as a 4-state FSM, which will cycle the memory

functions as described in table 9. Time steps are used to indicate the sequence of operations. The task of each memory block, *B1* and *B2*, is indicated. Arrows are used to indicate the order in which memory locations are addressed; an arrow pointing left indicates increasing addresses, and an arrow pointing right indicates decreasing addresses.

Time Step	B1	B2
1	writing data ---> calculating tail —>	–empty--
2	data ---> –full--	writing data —> calculating tail —>
3	data ---> traceback <--- writing data <--- calculating tail <---	data ---> –full--
4	data <--- --full--	data ---> traceback <--- writing data <--- calculating tail <---
5	data <--- traceback —> writing data —> calculating tail —>	data <--- --full--
6	data —> –full--	data <--- traceback —> writing data —> calculating tail —>
7	data —> traceback <--- writing data <--- calculating tail <---	data —> –full--

Table 9. *Sequence of events in survivor path memory*

The same memory block can be used to both trace back one symbol, and write the next symbol at the same time. This requires a memory that can be read and written to at the

same time (in the same memory location). Most memories, though certainly not all, have this capability. In particular, synchronous memories can be read and written to the same location in the same clock cycle.

Note that data is sometimes ordered in an increasing order, and sometimes in a decreasing order. This adds a small amount of complexity to the memory controller.

Of course, the value of D must be chosen accordingly to what has been found about the minimum value of SPL in section 3.5.1.

With this memory architecture, while decision bits are being written in the write block, the tail of the paths is being calculated. At the same time, the decode block is being traced back, with the previously calculated tail as starting state, and data bits are being decoded.

3.7 Summary of results

Several modifications to the Viterbi algorithm have been proposed. These modifications address the areas of register normalization, survivor path storage, survivor path memory organization, and traceback method.

Figure 20 shows a comparison between results for the algorithm without any modifications (figure 7) and the optimized algorithm with $SPL=70$ and $Q=20$. Table 10 summarizes the algorithm complexity for each case (the optimized algorithm includes 64 extra memory accesses to account for the tail calculation). The decoding speed in the unoptimized case has been calculated as approximately 2.8kbps, in the case when memory accesses and arithmetic-logic operations take a clock cycle to complete. The decoding speed of the fully optimized algorithm is approximately 54.5kbps. This represents an increase by a factor of more than 19, obtained by manipulation of the algorithm, with negligible degradation in bit-error probability.

algorithm	*ckp*	*mem*	*alo*
unmodified	8678	164	8513
optimized	326	132	321

Table 10. *Number of operations required for decoding one bit, for the unmodified and the optimized algorithms.*

Of course, this decoding speed is still far from the goal of one symbol per clock cycle. The remaining speed will have to be obtained not from changes to the algorithm per se, but by exploiting parallelism and performing several operations concurrently. This is done in chapter 4.

3.8 Fully optimized algorithm

The fully optimized algorithm, including all modifications made in the previous section, is presented. This version of the algorithm is specific for code rate r equal to 1/2.

The algorithm does not include the trace forward method. However, it should be taken into account that accesses to memory D occur as described in section 3.6.2.

Definitions

Let:

$R = r_1, r_2, r_3, \ldots$ the sequence of received, noisy symbols that are to be decoded
U the set of states of the convolutional coder
T a trellis, defined as the pair (S, g_T) where
$S \subseteq U \times U$ is a set of ordered pairs (x, y), specifying that state x is connected to state y. That is, (x, y) is a branch in the trellis
g_T: $S \rightarrow \mathbf{R}^r$ is a function that takes an element of S and returns the vector that corresponds to the output of the coder for that element
M a vector, where the element in position x is the accumulated distance for the path that ends in state x. Elements of M may not be larger than $2D_{max}$. M is initialized to zero
M_temp a vector used to store temporal values, before storing them in M
P a vector used to store state tails. It has a row for each state in U.

D a finite array where the algorithm stores the paths it creates. Each column is a time iteration, and each row is a state. Element (w, z) contains the previous state for state w at time iteration z. It is initialized to zero

O is the output (decoded) sequence

$||$ is the concatenation operator. $x_3 \,||\, (x_2, x_1) = x_3, x_2, x_1$

Q a number, which indicates how many branches are to be decoded in a single traceback

SPL the number of columns of *D*. The number of columns correspond to the number of time iterations that can be stored in *D*

mod(x, y) a function that returns x *modulus* y

MSB(x) a function that returns the most significant bit of binary number x

sum2c(x, y) a function that returns $x+y$, in 2's-complement arithmetic

h: $(U, B) \rightarrow U$, where $B=\{0,1\}$, is a function that takes a state of the trellis, and a binary number, and returns the previous state in the trellis

Algorithm: Viterbi_Optimized(*R*, *U*, *T*);
Inputs: A sequence of noisy symbols
Output: A sequence of uncoded symbols

```
Viterbi_Optimized(R, U, T)
{
   start = 0;                   // pointer to start (oldest branch) of structure D
   t = 0;                       // t counts number of symbols in R
   while(! is_empty(R)) {       // repeat for all symbols in R, taken in sequence
      t = mod(t + 1, SPL);      // update iteration counter
      r = getNext(R);           // get next element from R and remove it
      for (x : x e U) {         // find minimumdistance between each state in coder
                                // trellis and received symbol
         n =0;
         for (y : (y, x) e S) { // find minimum distance between each branch that
                                // ends in state x and received symbol
            d_n = mod(distance(r, g_T(y, x)) + M(y), 2D_max);
            n = n + 1;
         }
         if (MSB(sum2c(d_0, d_1)) = 1) {  // if d0<d1
            M_temp(x) = d_0;              // new path distance is m
            D(x, t) = 0;                  // store previous state in the trajectory
         }
         else {
            M_temp(x) = d_1;              // new path distance is m
            D(x, t) = 1;                  // store previous state in the trajectory
         }
         current_state = D(minimum(M_temp), t);      // find shortest path
      }
      M = M_temp;                         // update path distances
      if (full(D))   {                    // if decoder memory is full
         i = SPL;                         // i counts number of branches processed
         j = t;                           // j points to end of D (earliest branch)
         while(TRUE) {                         // repeat until start of D is reached
            next_state = D(current_state, j);  // find next state in path
            if (i <Q) {                        // check if Q or less branches remain
               O = O || decode(next_state, current_state);    // decode symbol that
                                                              // corresponds to current
                                                              // branch in path
            }
            if (j = start) {                   // if oldest branch in trellis reached
               start = start + Q;              // update start of D and exit
               break;
            }
            current_state = next_state;        // move back one step in D
            j = mod(j - 1, Dsize);             // update pointer to D
            i = i  1;                                 // update branch counter
         }
      }
   }
}
```

4 Implementation

In this chapter an architecture for implementing the Viterbi algorithm is proposed. This architecture is simulated in a computer system and then implemented and verified in a VLSI circuit.

The architecture proposed strives to meet the requirements set forth in section 1.2. The architecture seeks to exploit parallelism whenever possible; throughout the whole chapter, this search for parallelism will be evident.

4.1 Architecture

The architecture proposed divides the Viterbi algorithm in three modules. One module calculates the path metrics and selects the shortest one; it is called the add-compare-select (ACS) module. The second is the survivor path memory; all paths are stored here. The third module performs the trace forward and traceback, and decodes the information.

This partition follows directly from the algorithm. It can be observed that first survivor paths are selected, then stored, then processed.

A block diagram of these modules is presented in figure 25.

As seen in section 3.1, it is assumed that the communication system uses QAM-4 modulation. **X** and **Y** in figure 25 are 3-bit vectors that come from the system's demodulator, which makes 3-bit soft decisions on the received QAM-4 vectors (see section 2.3) It is also assumed that a code of rate 1/2 is used.

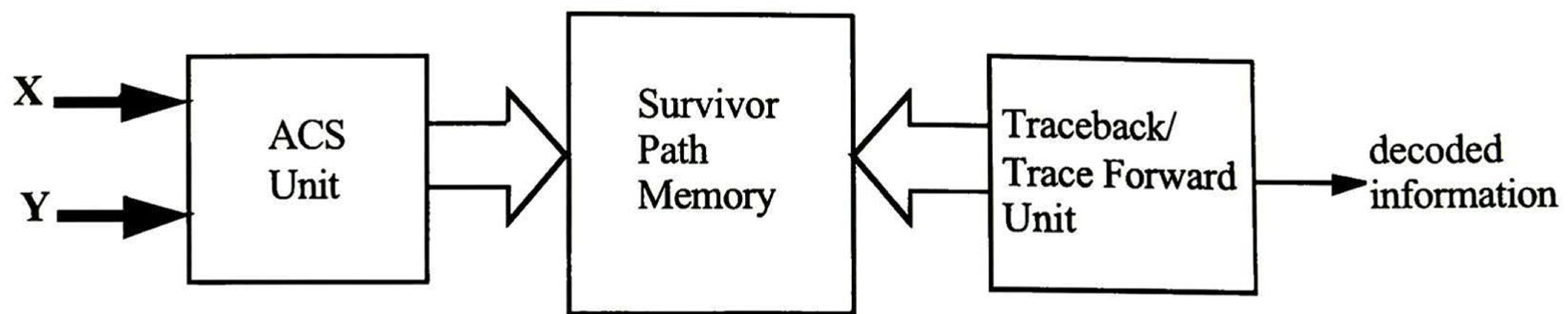


Figure 25. *Block Diagram of the Architecture*

4.1.1 The ACS module

The function of the ACS module is to select survivor paths and keep path metrics updated. It produces decision bits that are stored in the survivor path memory.

The ACS module is regarded as one of the main obstacles to achieve an efficient VLSI implementation, and much study has been devoted to it [14], [15]. The problem is that the ACS operation is slow, requires many gates, and needs to be executed on a large number of states. In this section, it is explored how to design an efficient ACS module.

The Viterbi algorithm presented in chapter 3 selects, for each received symbol, and for each trellis state, the branch that belongs to the shortest path that ends in that state. The branches that end in each state depend on the particular convolutional coder selected. However, it has been observed that trellises used in convolutional coding present certain of the properties of **shuffle-exchange graphs** [12]. One of these properties is that a n-state trellis can be divided in $n/2$ 2-state trellises, known as **butterflies**. These structures are very similar to those found in Fast Fourier Transform algorithms.

For example, the trellis shown in figure 5 can be divided in two butterflies as shown in figure 26.

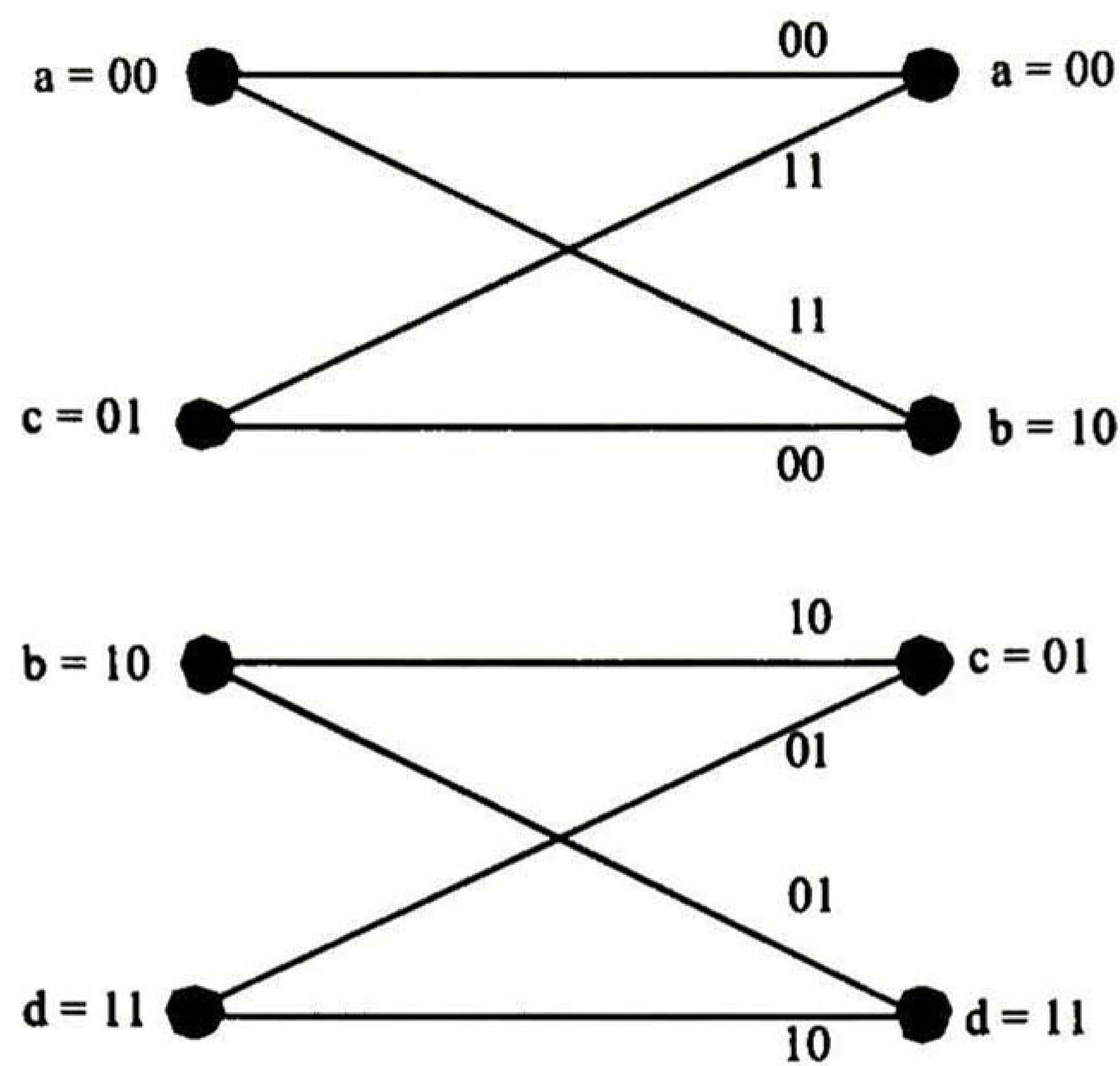


Figure 26. *Two butterflies*

For each state, the distance between the label associated with each branch and the received symbol (called branch metric; see 2.3.2) is calculated. Then, each distance is added to the path metric for that state, and the smallest one is selected. For this reason, this operation is known as **add-compare-select** (ACS).

Each destination state has two sources, and the ACS operation selects one of them and discards the other. This decision can be represented by a "0" when one path is selected, and by a "1" when the other is selected; the output of the ACS operation is a single bit, called **decision bit**, as described in section 3.6.1. The decision bit can be made equal to the less-significant bit of the selected source state.

The circuit that performs this operation on a single state of the trellis is called an **ACS calculating circuit** The circuit that operates on the whole trellis is called the **ACS unit**

There are several ways to implement the ACS unit, with respect to the level of parallelism desired [13], [14], [15]. In one extreme, the serial implementation of the ACS unit comprises a single ACS calculating circuit that operates on each state in a serial manner. This approach requires the less area, but is the slowest. On the other hand, the parallel implementation has as many ACS calculating circuits as there are states in the trellis, and each state is processed independently and concurrently. This approach requires a large amount of area, but is the fastest. There are intermediate solutions where there are fewer calculating circuits than states, achieving a compromise in area versus performance.

Since the objective of this work is to concentrate on performance, the fully parallel implementation has been chosen; so, there is one ACS calculating circuit per trellis state. The question remains on what is the most efficient way to design the ACS calculating circuits.

The trellis butterflies have many characteristics that help in the design of the ACS calculating circuit. One is that the four branches of a butterfly share only two different labels; that is, branch labels are repeated in each butterfly. This means that only two branch metrics must be calculated per butterfly. Likewise, the same two path metrics are used to calculate the shortest distance to each state. In conclusion, it can be seen that the same information is needed in both states of the butterfly; it is only the way the information is used which varies per state. For this reason, it is proposed that the trellis is divided up in butterflies and one ACS calculating circuit is used per butterfly.

It can also be seen that there is no need to calculate the branch metrics every time a new symbol arrives. There are only four different branch labels in the trellis. Likewise, there are only 64 different symbols that might be received: there are 64 different values for two vectors of three bits each. So, all possible branch metrics can be calculated in advance, and the proper result presented to each ACS calculating circuit for each received symbol. This approach reduces complicated arithmetic circuitry to a lookup table.

As shown in Appendix A, the Euclidean metric can be substituted by the absolute value metric in the maximum likelihood calculations of the Viterbi algorithm, without affecting performance. The absolute value metric is much easier to calculate than the Euclidean, and thus it is chosen for the proposed architecture. The absolute value metric defines the distance between the received vector (x_1, x_2) and the branch label (y_1, y_2) as:

$$D = |x_1 - y_1| + |x_2 - y_2| \tag{9}$$

Since both the Euclidean and the absolute value metrics can be calculated via a lookup table approach, the most important advantage of the absolute value metric is that it always returns an integer value, so it is not necessary to design circuitry to handle floating-point numbers.

The lookup table approach requires a memory of 2^{12} x 4 bits: there are 2^{12} combinations of x_1, x_2, y_1, y_2, and D is a four-bit number. Such a large memory might be

unfeasible to implement; an alternative is to use a smaller lookup table to realize the subtractions in Eq. (9), and implement an adder to carry out the sum.

Furthermore, it can be observed that the values being subtracted, (y_1, y_2), are always either "000" or "111" The values x_1 and x_2 are 3-bit quantized samples from the demodulator. Table 11 shows the possible results of the subtractions of Eq. (9).

As it is seen in table 11, the possible result of each of the subtractions in Eq. (9) is either the received vector, or its logical negation.

x	|x - 000"|	|x- 111"|
0	0	111
1	1	110
10	10	101
11	11	100
100	100	11
101	101	10
110	110	1
111	111	0

Table 11. *All possible results of subtractions in Eq. (9)*

The adds and comparisons needed in the ACS operation are potentially slow. For this reason, it is proposed that the ACS calculating circuit is implemented as a pipeline, where each (slow) operation is performed in each stage. A pipelined circuit allows the processing of one symbol per clock cycle, which is one of the requirements set forth in section 1.2.

Figure 27 shows the architecture proposed for the ACS calculating circuit. In it, the adders with the "+1" symbol add 1 to the result, and a single inverter symbol is used to indicate that every bit in the corresponding signal is inverted. These operations are needed to implement the two's complement method of distance comparison described in section 3.4. The box labeled "MSB" outputs only the most significant bit of its input.

The subtractions being input into the circuit are those described above, in table 11. The b_n quantity indicates whether the subtraction corresponds to the first or the second branch in the butterfly. The quantities labeled *Previous State Metric Top* and *Bottom* correspond to the current path metric of the top and bottom states of the butterfly, respectively.

The decision bits that the circuit calculates are those described in section 3.6.1. The decision bit is 0 when the branch that originates in the top state has been selected, and 1 otherwise. The *New Path Metric* quantities are the new path metrics for the top and bottom states of the butterfly, respectively. The numbers in parenthesis indicate the path that corresponds to the new metric; the first number indicates the state where the path originates (that is, on the left-hand side of the butterfly) and the second the state where it ends (that is, on the right-hand side). The number 1 corresponds, as before, to the top state, and 2 to the bottom state.

The figure also indicates how the circuit operations are pipelined in 3 stages. Each clock cycle, the results calculated in one stage are passed to the next. The adders and multiplexors of stage 3 operate in parallel. It can be seen that the ACS operation involves a relatively large number of sums; if no pipelining was used, then this circuit would impose severe restrictions to the maximum speed attainable.

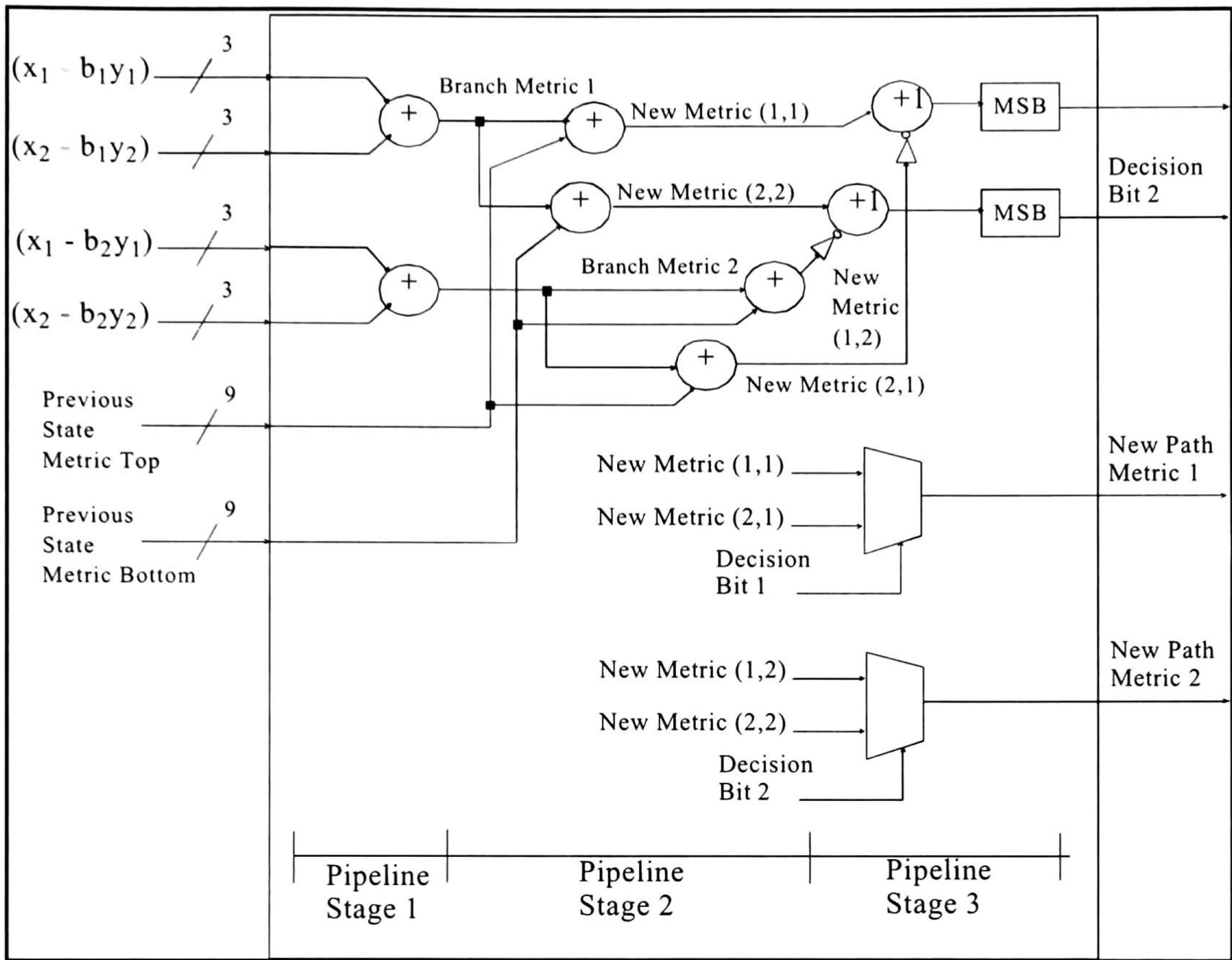


Figure 27. *ACS Calculating Circuit Architecture*

Figure 28 shows a block diagram for the ACS unit. Note that the output of this circuit is a 64-bit vector, comprising one decision bit per state. These bits are stored in the path memory and used to reconstruct the most likely path traversed by the convolutional coder, as described in the next section.

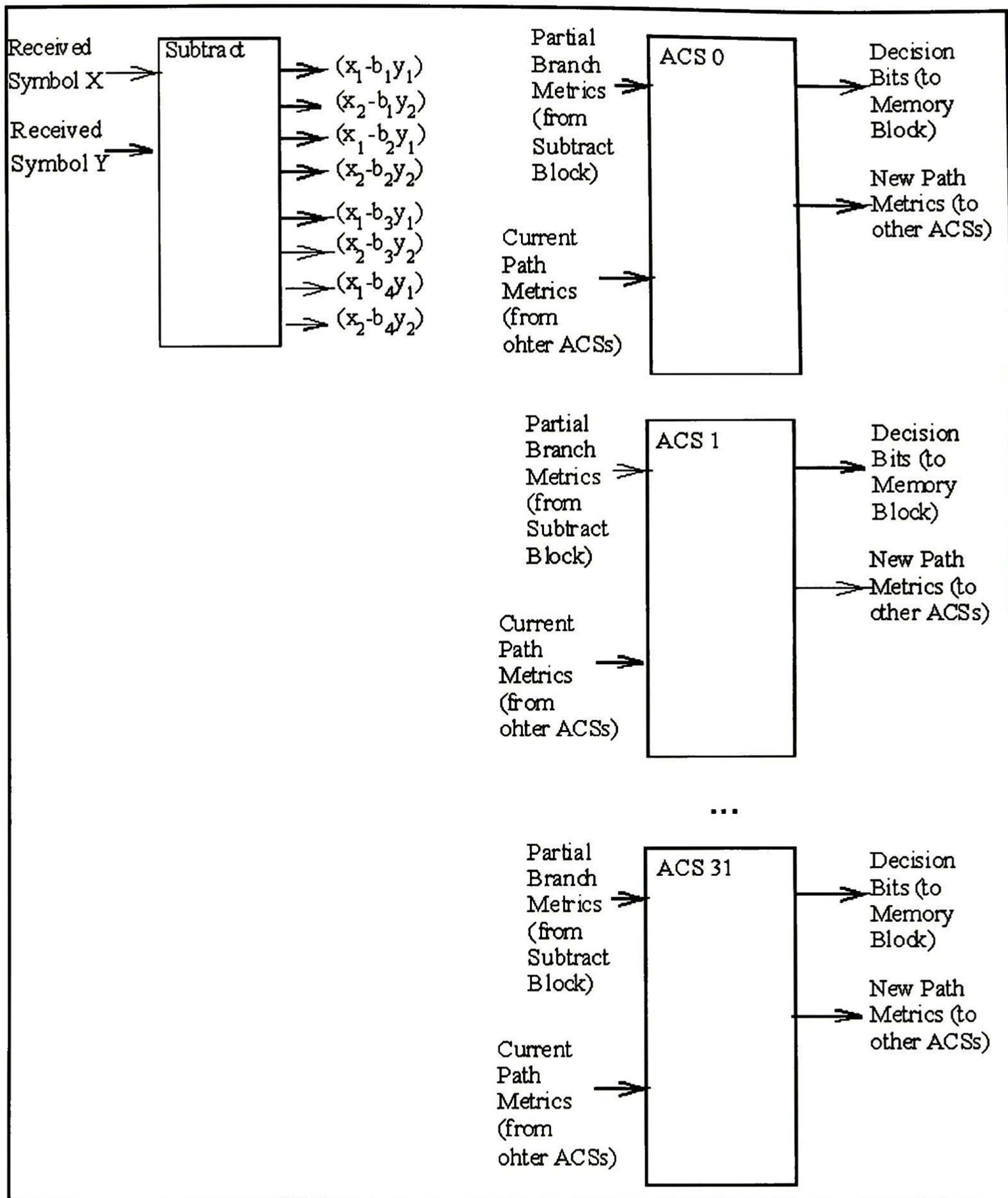


Figure 28. *ACS Unit Architecture*

The way the different blocks are connected depends on the particular trellis chosen for the convolutional coder. The trellis defines how the states are interconnected, and the ACS

unit follows this structure. It is interesting to note two things about this architecture. First, decoders for different trellises can be accomplished just by changing the interconnections, thus suggesting the idea of an ACS unit that uses a switching network of sorts to implement a decoder that is suited to any trellis.

Second, the same architecture can be used to implement decoders for any constraint length. Since one ACS calculating circuit is used per two states of the trellis, it can be seen that 2^L-1 blocks are needed, where L is the constraint length of the code. Naturally, the area requirements grow exponentially with L; the circuit's structure, however, remains the same.

The resources needed for each ACS calculating circuit consist of:

- 2 3-bit adders,
- 8 9-bit adders,
- 2 18:9 multiplexers,
- 40 flip-flops to implement the pipeline.

These requirements are not slight, especially since each of these circuits must be repeated many times. Since, for this particular implementation, 32 ACS calculating circuits are needed, the total resources needed for the ACS unit are:

- 64 3-bit adders,
- 256 9-bit adders,
- 64 18:9 multiplexers,
- 1280 flip-flops,
- and a significant amount of routing between elements.

The target device for this implementation, the Altera FLEX10K100, contains 5000 flip-flops. The ACS calculating circuit is expected, then, to require about 25-30% of the available area.

This circuit takes advantage of many characteristics of the trellis structure of convolutional codes to decrease its area requirements. The main characteristics, in this sense, of this architecture, are the use of the absolute value metric, and the avoidance of repeated calculation of values that are required in different blocks. The area savings in this respect allow a fully parallel, pipelined architecture that is not inherently small, but which is very

fast.

4.1.2 Survivor Path Memory

The survivor path memory is very simple. The main aspect that deserves careful consideration in a VLSI implementation is that the memory requires a word length of 64 bits, which is not available in any current FPGA architecture, and can potentially be expensive in an ASIC implementation.

The word length problem is easily solved by grouping together blocks of memory of shorter word length. In an FPGA, however, this can be impossible because the number of memory blocks is usually limited to a few.

One essential property that the memory must have is that it can be read and written to in a single clock cycle. Without this capability, it is impossible to decode one bit per clock cycle, which is a requirement of this implementation.

In section 3.6.2 a memory organization scheme was proposed that requires two blocks of *SPL*x64 bits each. This memory organization can be used to decode one bit per clock cycle. In this section, it is shown how this memory was implemented within the capabilities of the Altera FLEX10K100 FPGA.

The memory of the Altera device is not suited to implement the memory organization of section 3.6.2. It is not possible to read and write to the same memory location in the same clock cycle; two clock cycles are needed in the best case.

How can this problem be solved? The short answer is that it can't be solved within the capabilities of the target device. There are other memory organizations possible, but they do not allow the decoding of one bit per clock cycle. It is necessary to make a compromise, choose the second-best memory organization, and use it to demonstrate the validity of the algorithm proposed.

In section 3.6.2, it was shown how the structure of figure 24, which has three memory blocks, can be reduced to a structure with only two blocks. It is proposed to go back to the scheme of figure 24, which uses three memory blocks. This scheme avoids having to read and write to the same memory location at the same time; always one block is being read while other is being written to. This does not prevent one-cycle decoding *per se*; however, further

limitations in Altera's memory organization do.

The Altera device has 12 memory blocks, called "embedded array blocks (EAB)", of 2048 bits each. Each EAB can be configured to have 2048x1, 1024x2, 512x4, or 256x8 words [16]. In order to accommodate 64-bit words, it is necessary to use 8 EABs (in the 256x8 configuration). To build the three-block structure described in section 3.6.2, 24 memory blocks would be needed; however, as mentioned, the device has only 12.

Then, the only way to implement the memory is to use eight 256x8 EABs as if they were a large, 256x64 EAB, and to create the three memory blocks of figure 24 (*decode*, *merge*, and *write*) by partitioning the address space. Figure 29 shows how to implement this memory organization.

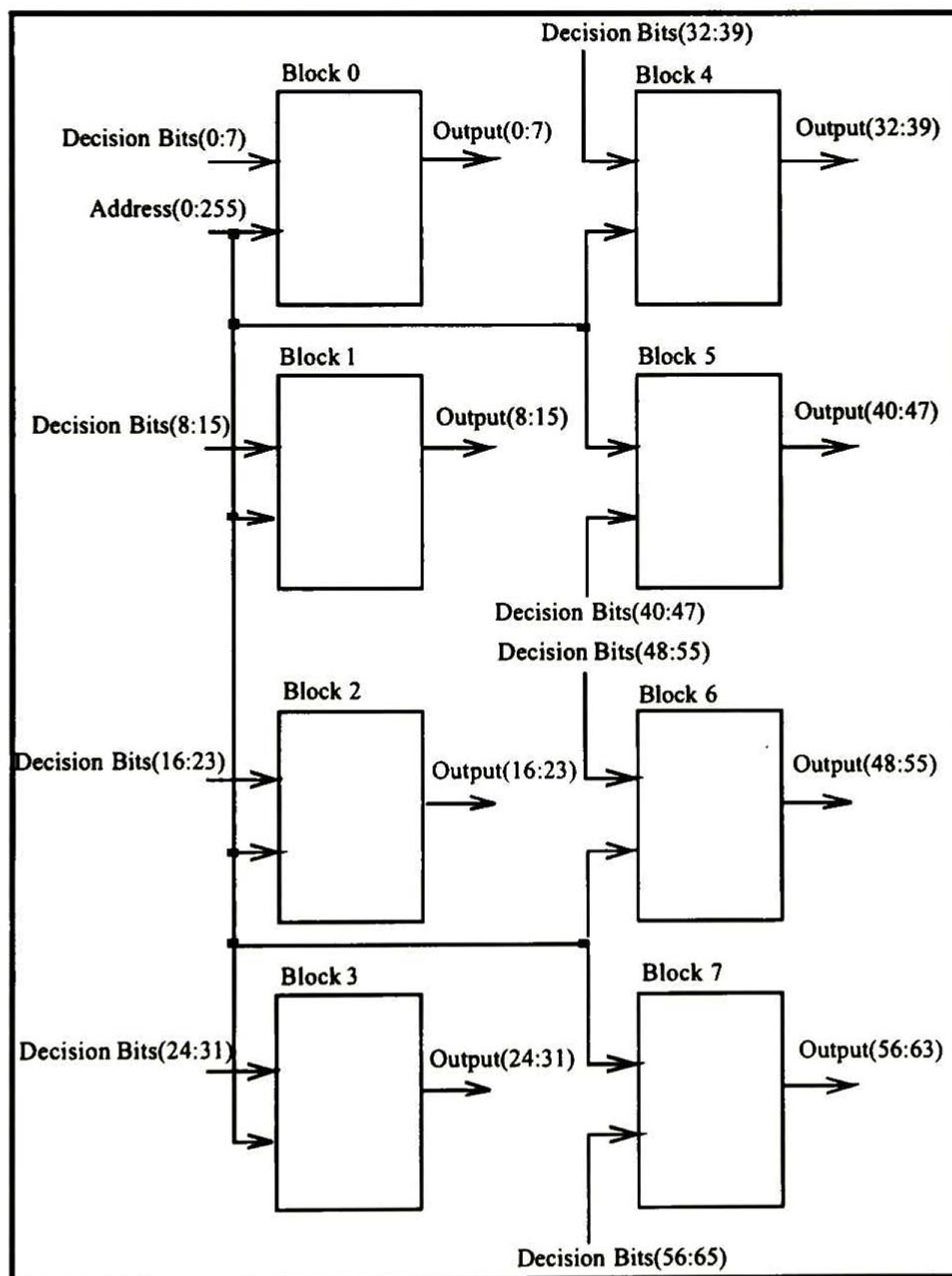


Figure 29. *Memory Organization*

As explained above, each memory block in the figure is configured as 256x8. Thus, the address space of this memory comprises 256 words of 64 bits each. This memory is then divided in three blocks; each has 80 words. A memory controller alternates the function of each block between *decode*, *merge*, and *write*, according to the scheme presented in table 12, where blocks have been named *B1*, *B2*, and *B3*. The notation is that of table 9, where arrows are used to indicate the ordering of data within the memory. Tail operations are done concurrently with memory accesses, and are discussed in the next section.

The reasons to choose *SPL* to be 80 are that in this way, the memory can be neatly divided in three equal blocks, and 80 has been determined in section 3.5.1 to be a good enough number to use.

Time Step	B1	B2	B3
1	writing data --->	–empty--	--empty--
2	data ---> –full--	writing data —> calculating tail of B1 —>	--empty--
3	data ---> traceback <---	data ---> –full--	writing data —> calculating tail of B2 --->
4	writing data --- > calculating tail of B3 —>	data ---> traceback <---	data —> –full--
5	data —> –full--	writing data —> calculating tail of B1 —>	data —> traceback <---
6	data —> traceback <---	data —> –full--	writing data —> calculating tail of B2 —>

Table 12. *Sequence of operations using a 3-block memory*

Note that in this memory architecture data is always ordered in increasing order, as

opposed to the two-block architecture, where the ordering alternates between increasing and decreasing.

Note also that time steps 4, 5, and 6 repeat indefinitely. Thus, a 3-state controller is needed.

In conclusion, because of device limitations, it is impossible to implement a survivor path memory that allows the decoding of one bit per clock cycle. A 3-block architecture that is feasible given the target device has been presented. This memory has to run with a clock that is twice as fast as the rest of the circuit, because it has to execute both a read and a write in one symbol period.

4.1.3 Traceback Unit

The traceback unit uses the data stored in the survivor path memory to reconstruct the original path traversed by the convolutional coder in an optimal way.

As described in section 3.5.3, the actual traceback method implemented is a hybrid traceforward method that employs tails to calculate the state where the traceback will start.

From table 12, it can be seen that while data is being written to a block, the tail that corresponds to that block is being calculated concurrently. The tail that is thus obtained is then used as the starting state to traceback the data in the previous block.

The traceback is then done in two, concurrent steps. First calculate the tail, then use it to traceback the previous block. Thus, there are two circuits in the traceback unit, one to calculate tails and another to realize the actual traceback.

Tail calculator. This circuit is actually very straightforward, although very large in area. There are 64 6-bit registers, numbered from 0 to 63. Each must be initialized to its assigned number at the start of the recursion. At the end of the recursion, all registers will contain the same number with very high probability; this number is the state where traceback of the previous block should start from.

During the recursion, each register is loaded with the value of one of the two registers that are connected to it. Which value to use depends on the decision bit being read from the survivor path memory. The connections between registers are defined by the trellis.

The tail initialization, as well as the data provided to it from the survivor path memory, should be controlled by the survivor path memory controller.

The architecture of the tail calculator is presented in figure 30.

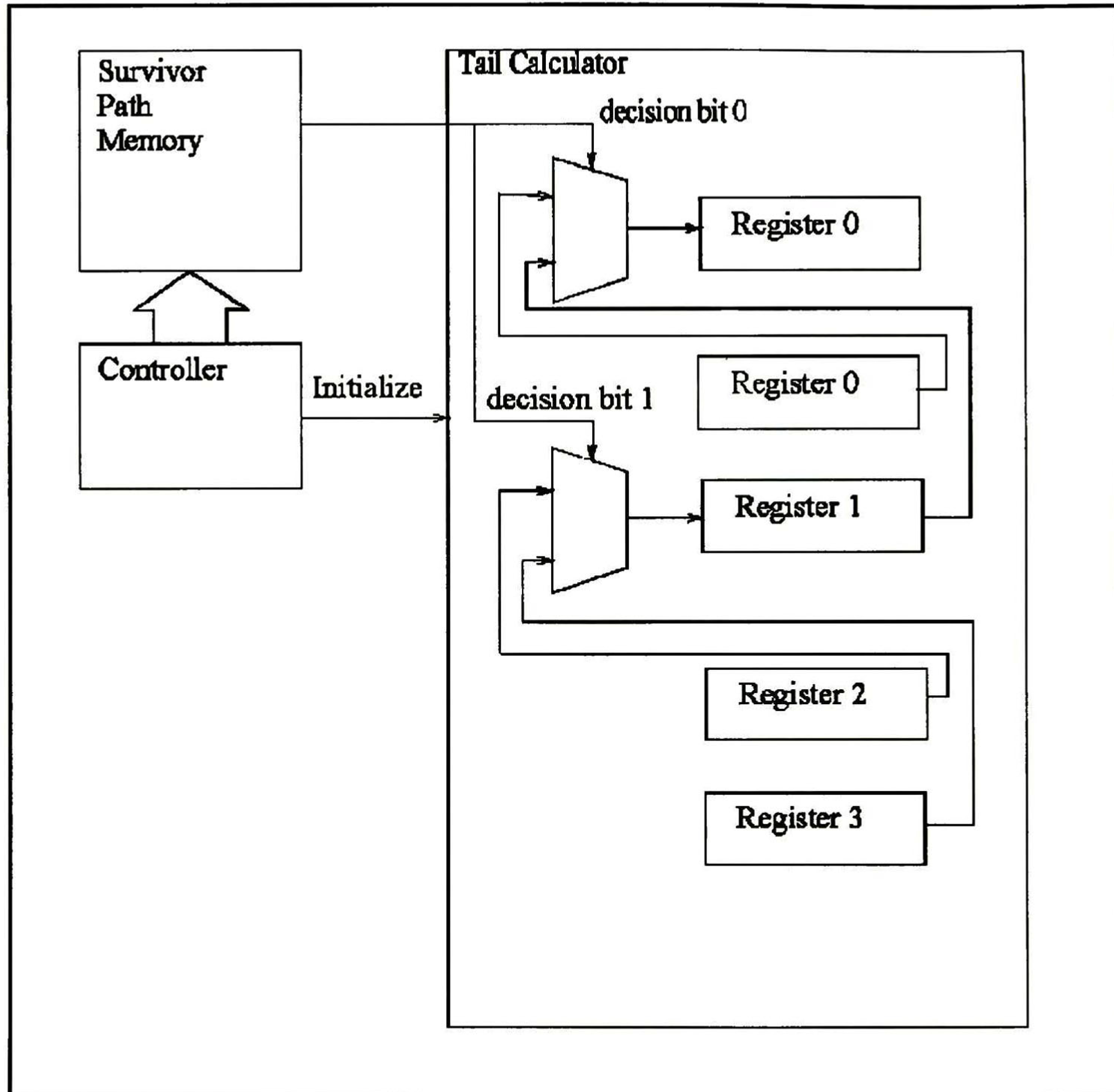


Figure 30. *Tail calculator architecture*

Note that *register 0*is connected to itself; this happens when, for a given input, the next state of the trellis is equal to the present state.

It can be seen that this circuit is very simple, although, as said above, it can be very large. The resources needed for this circuit are 64 registers of 6 bits each, for a total of 384 flip flops, in addition to 64 2-bit multiplexers. The Altera device has around 5,000 flip flips, which means that the tail calculator can take up between 5% and 10% of the total circuit area.

Traceback circuit. The traceback circuit needs to load the calculated tail when it starts in a register called "*current state*", and then use the decision bit associated with it to decide which state it is connected to (the "*previous state*").

$$previous\ state = (current\ state << 1) || decision\ bit$$

Since the trellis of a convolutional coder is a shuffle-exchange graph (section 3.6.1), the previous state is easily calculated by equation 8, repeated here for convenience:

The traceback unit must also decide what was the input to the convolutional coder that produced the transition that has been selected. That is, it is known that the convolutional coder made a transition from "*previous state*" to "*current state*", but what the convolutional decoder must ultimately calculate is what input would have produced such a transition, which is equivalent to decoding. This operation is called **decode** in the algorithm in section 3.8.

It happens that, precisely because of the shuffle-exchange structure of the trellis, the decoded bit in each traceback iteration is equal to the most-significant bit of "*current state*". That property can be easily inferred also from the schematic representation of a convolutional encoder (figure 1): the only way that the state of a coder of Y bits is more than $Y/2$ is because the bit that just entered the coder is a "1" Naturally, this is the same as saying that the decision bit that corresponds to the "*current state*" can be used as the decoded bit.

It must be taken into account that the traceback operation produces decoded bits in the inverse order from which they were generated. It is thus necessary to invert them again, and this can be accomplished with a pair of shift registers. Two are needed because one must be used to store the output of the traceback circuit, while another outputs the inverted data stream to the receiver.

Given these considerations, the architecture of the traceback circuit is presented in figure 31.

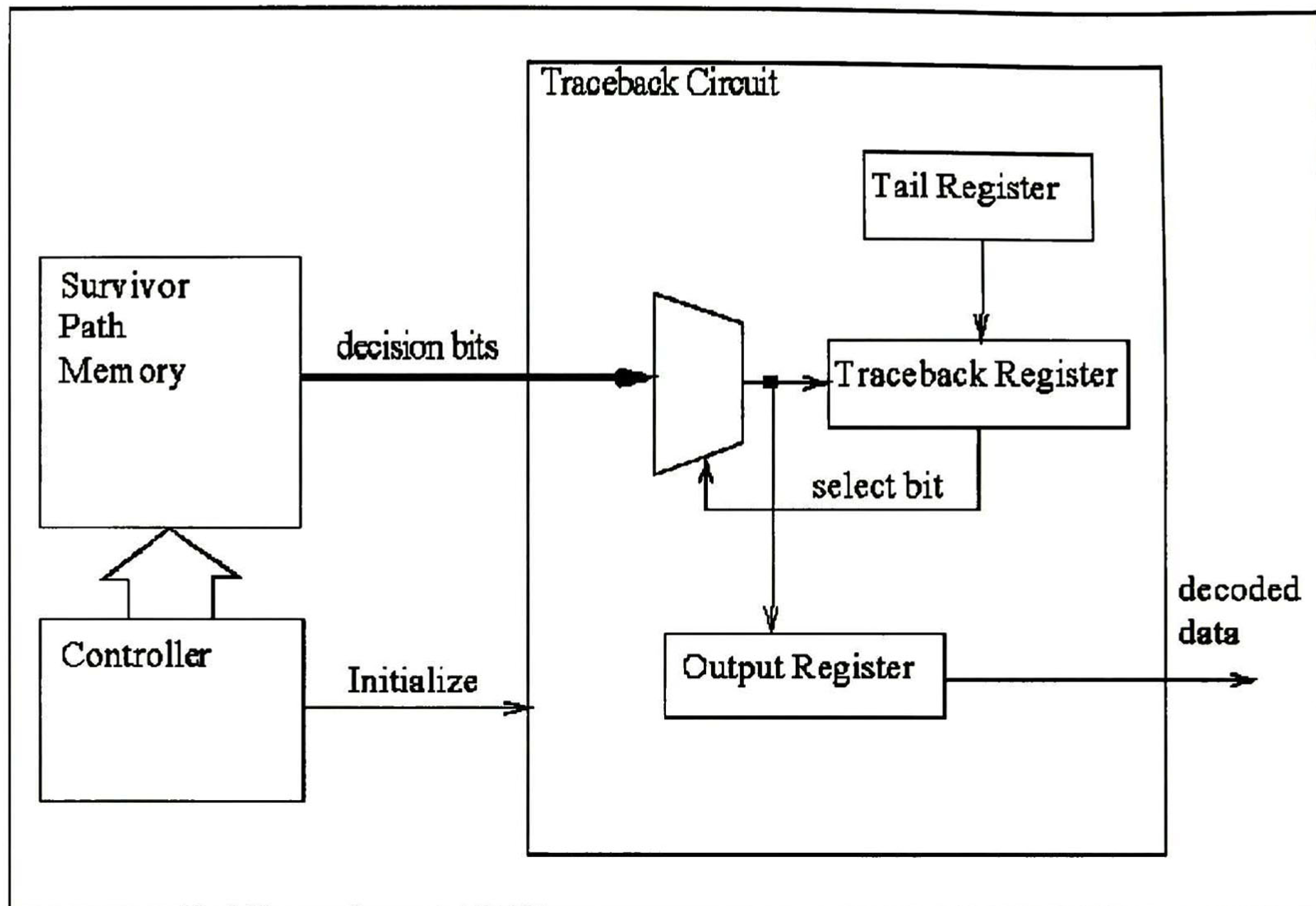


Figure 31. *Traceback circuit architecture*

The traceback circuit requires a 64 to 1 multiplexer and 160 (80x2) flip flops for the output register. It is half the size of the tail calculator.

In conclusion, it is evident how the properties of the trellis structure of convolutional codes can result in very simple and fast, if large, decoding circuits.

4.1.3 Other considerations

This architecture is fully synchronous, and thus it has all the advantages of synchronous design [27].

This implementation needs a signal to tell it when to start processing the incoming data stream. Another signal is used to indicate the circuit that receives the decoded output when it can be considered valid. These have been implemented as active-high signals.

5 Results

This chapter presents and evaluates simulation and experimental results obtained with the architecture proposed in chapter 4. In simulation, it is verified that a circuit based on the proposed architecture actually works. Experimentally, the areas where the architecture is evaluated are: silicon area, decoding rate, clock period, and bit-error rate performance.

5.1 Simulation results

Two circuits based in the architecture proposed above were designed and simulated. One circuit employed the two-block memory architecture of figure 24. As explained in chapter 4, it is not possible to implement this memory in the target device; however, it was important to simulate it to demonstrate, as far as possible, the validity of the concept. The simulation results obtained with this model are exactly equal to those obtained with the implementable 3-block memory, except for the difference in clock rate. The second circuit designed uses a 3-block memory architecture, and thus it fits in the target device. Since this is the circuit that was implemented, the rest of the chapter focuses on it.

The circuit was written in VHDL, synthesized and simulated with Synopsys tools, and physical design was done with Altera's own MaxPlusII. The circuit was verified using a simulation test plan, which is presented in Appendix C. By definition, the circuit is declared functional when it passes all cases in the test plan.

The most important test is the bit-error rate test. It is desired that the circuit presents a bit-error rate performance that is very close to the theoretical results, and at worst similar to those found in the C simulations of the algorithm and presented in section 3, figures 19 to 21. The circuit's simulated bit-error rate performance is presented in figure 32, where it is compared to the unmodified algorithm as presented in section 2.3.5.

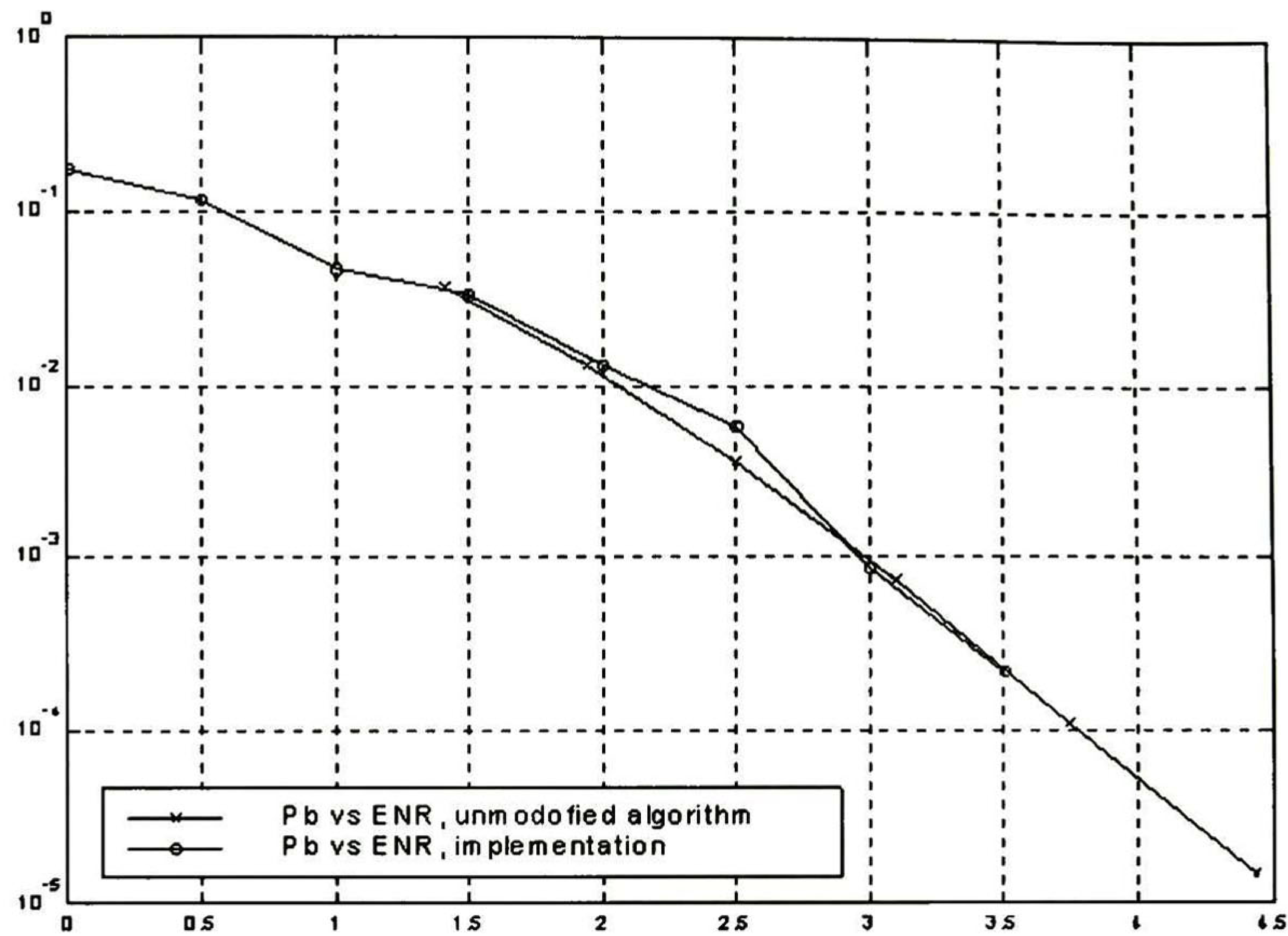


Figure 32. *Comparison between unmodified and implemented algorithms*

Figure 32 shows comparable performance between the unmodified algorithm and the implemented version, especially when E_b/N_o is larger than 3dB. The difference seen between 2dB and 3dB could be due to the fact that the hardware architecture was simulated for fewer bits than the original algorithm, which was coded in C, because VHDL simulation is much slower; another possibility is that, as seen in figure 14, an *SPL* of 80, as implemented, is not the best choice for an E_b/N_o of 2.5dB, being in one extreme of the knee. As explained in section 3.5.1, however, communication systems rarely operate below at $P_b < 10^{-3}$, which is achieved by the proposed architecture at E_b/N_o approximately equal to 3dB. Results are comparable to those reported in the literature [17].

5.2 Experimental results

In this section, it is described how the circuit was tested in the laboratory, and the area and speed results obtained. These results are compared to those of other reported Viterbi implementations.

5.2.1 Lab verification

In order to test the circuit in the lab, a scheme called *built-in self verification* was used. It is similar to *built-in self test*[27], but it exercises the circuit in functional mode instead of test mode. That is, it is not intended to verify that the circuit has no manufacturing errors, but rather than it is functional.

The Viterbi architecture presented here is rather simple in the sense that it has only one mode of operation. It cannot be configured to do different things. For this reason, it is easy to design a circuit that can be used to test the decoder. The scheme proposed is shown in figure 33.

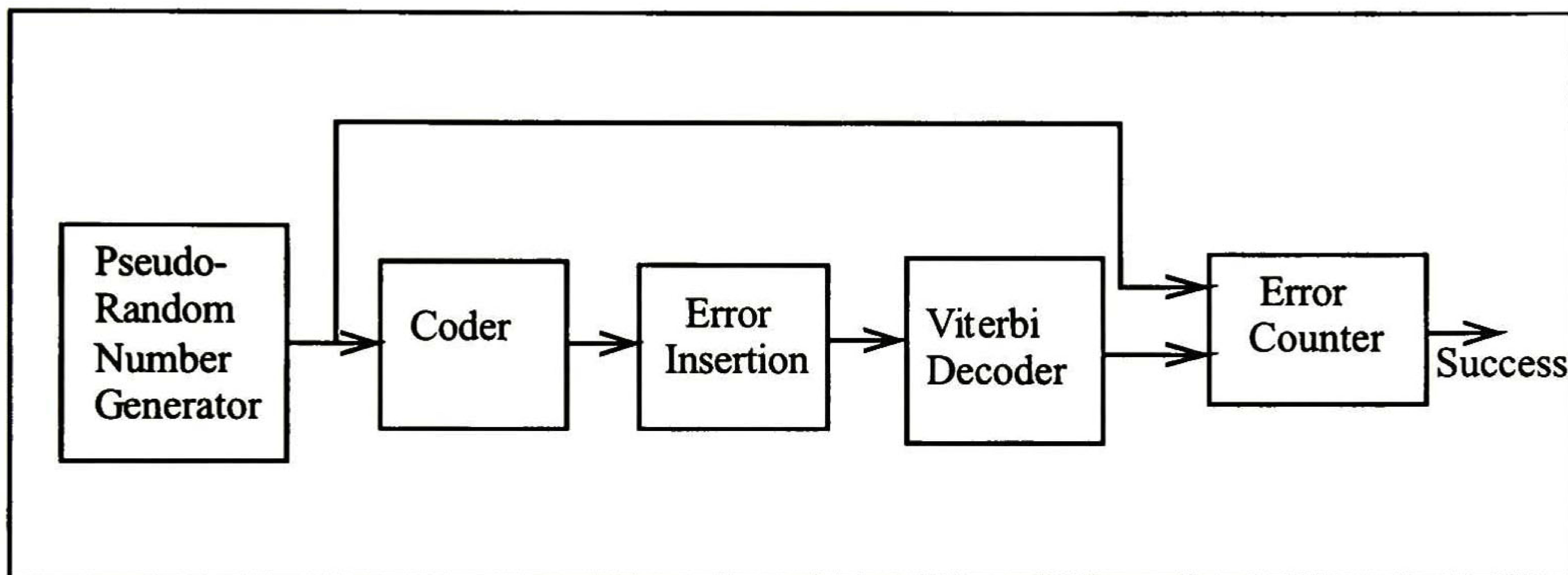


***Figure* 33.** *Built-In Self Verification Scheme*

In this scheme, a pseudorandom number generator is used as a data source. Another block inserts errors at a controlled rate, simulating a channel, and feeds the corrupted data to the Viterbi decoder. The decoder will correct some of the errors; how many depends on the insertion rate, and can be determined by simulation. An error counter is used to check that the error rate at the output of the decoder is exactly that predicted by simulation, in which case it determines that the circuit is functionally correct. The proposed architecture was tested successfully, using this scheme, in the laboratory, at the clock rate described below.

5.2.2 Area resources used

The implementation proposed fits in the target device, where it uses 92% of the

available resources. The numbers reported by Altera's EDA tools are summarized in table 13.

Memory Bits Used	% Memory Used	Logic Cells Used	% Logic Cells Used
15360	62%	4603	92%

Table 13. *Resources used by the circuit*

These numbers include the testing circuit described above, which take up about 10% of the area. The circuit had to be synthesized with area optimizations in order to make it fit in the FPGA. This had the side-effect of reducing the operating speed. In the end, however, the objective of fitting it in the desired FPGA was achieved.

5.2.3 Decoding rate and clock period

With the circuit optimized for area, as described above, the maximum clock frequency achieved was 8MHz. Given the FPGA memory limitation described in section 4.1.2, the maximum bit rate achieved is 4Mbps. Given that the memory limitation is inherent to the FPGA, and not to the proposed architecture, the objective of decoding one bit per clock cycle is considered achieved.

5.2.4 Decoding delay

Decoding delay is defined as the number of clock cycles that it takes to decode one symbol, from the moment it is sampled at the input of the circuit until the decoded symbol appears in the circuit's outputs. The circuit's delay is 384 clock cycles, or 96us at 4MHz. There was not a goal to achieve regarding decoding delay; however, the delay obtained is very good for most applications.

5.3 Comparison with other implementations

The implementation proposed in this thesis is compared with the architectures offered by Qualcomm [17], by Motorola [18], by Mentor Graphics [19], and with the architecture reported by the Jet Propulsion Laboratory [28].

Qualcomm offers a chip tailored for satellite communications, and it has been available since 1998. Motorola offers a high-performance vector parallel processing expansion to their PowerPC microprocessor (called AltiVec), and they offer a computer program that implements the Viterbi algorithm on that architecture. Mentor Graphics offers an intellectual-property core, which can be implemented in any technology. IBM Corp. offers a chip based on this core [20]. Finally, the JPL Viterbi decoder is used to allow reliable communications with space probes carrying out missions in deep space.

Unfortunately, not all vendors provide complete information about their products, making the comparison difficult. Table 14 summarizes the information available.

Code Rate. Most Viterbi decoders have a code rate of 1/2. The reasons are, most probably, that it is the easier code rate to implement, and at the same time allows for decoding of a large number of punctured codes. Only the JPL implementation has a different code rate, 1/6, given the enormous error-correcting requirements of deep space communications.

Constraint length. Most Viterbi decoders have a code rate of 7. Only the Motorola implementation has a smaller constraint length, 5, and the JPL implementation has a constraint length of 15.

Decoding Rate. Decoding rate varies widely between different Viterbi decoders, ranging from 4Mbps to 85Mbps. The architecture proposed in this thesis is severely hampered by the technology used; the Altera Flex10K100 FPGA is more than six years old, and, in synthesis, the design had to be heavily optimized for area instead of speed to make it fit in the device. It is probable that, with adequate technology (for example, 0.35u CMOS) the proposed architecture would reach between 300Mbps and 500Mbps.

Clocks per decoded symbol. The ability to decode one symbol per clock is a good measure of the efficiency of an architecture, because it means that it cannot be optimized further, except to increase its clock period or to decrease its area. Of the architectures being compared, only Qualcomm and the one proposed here achieve this goal.

Area. Area requirements for the different architectures vary widely. All the hardware-based implementations, except the JPL one, fit very comfortably in any current gate-array

Architecture	Code Rate	Constraint Length	Decoding Rate	One clock per symbol	Area	Fabrication Technology	Decoding Delay*	Single Chip Solution
This thesis	1/2	7	4Mbps	yes	~90Kgates	Altera Flex10K100	. 384	yes
Qualcomm	1/2	7	30Mbps	yes	unknown	ASIC/ unknown	183	yes
Motorola	1/2	5	24Mbps	no	Software	Software + PowerPC CPU	unknown	no
Mentor Graphics	1/2	7	85Mbps	no	~40Kgates	0.5u CMOS	unknown	yes
JPL	1/6	15	4.4Mbps	no	~11Mgates + 4Mbits of memory	ASIC/ unknown	unknown	no (64 ASICs)

Table 14. *Architecture comparison*

* Decoding delay is expressed in clock periods

ASIC.

Fabrication Technology. Fabrication technology is a very important factor when comparing architecture implementations, because an inferior architecture can perform better than a superior one given sufficiently better technology. The ideal would be to compare all the architectures when implemented with the same technology; however, this is not possible.

Decoding Delay. The decoding delay can play an important role in communication systems that depend on latency, such as voice communication. The decoding delay of the architectures being compared is unknown except for the Qualcomm decoder, which appears to be more than two times better than the proposed architecture. The decoding delay depends on the length of the path memory, because the traceback operation generates a sequence of bits that is in inverse order with respect to the original sequence; this sequence must be stored and then output in reverse order. The reason the observed delay is so high in the proposed architecture is the memory limitation described in section 4.1.2; when used with synchronous memory the delay is reduced to 192, which compares well with the Qualcomm delay, especially since the Qualcomm device has a shorter path memory (96 states vs. 120 states).

Single chip solution. For cost reasons, it could be desirable to have a solution that fits in a single chip. Here, the JPL and Motorola implementations do not fare very well, and their solutions can be very costly.

As can be seen, it is very difficult to draw definitive conclusions on which architecture is best. There are, however, two reasons why it might be argued that the architecture proposed in this work is, in several ways, superior to other offerings.

One reason is the simplified and heavily pipelined ACS calculating circuit. This circuit alone might explain the area difference between Mentor Graphics' architecture and the one proposed. Also, it is evident from the descriptions in [18] and [19] that Motorola and Mentor use path normalization (section 3.4) instead of the 2's-complement method. In chapter 3 it was shown how the 2's-complement method is vastly superior to path normalization.

The second reason is that the competition seems to be using traceback instead of the hybrid trace-forward method proposed here. As has been shown, it is not possible to achieve maximum efficiency (one decoded symbol per cycle) with conventional traceback. (The

Qualcomm chip does decode one symbol per cycle, but it is not clear from the information available why they are limited to 30Mbps).

Other advantages of the proposed architecture are that changing it to be used for larger constraint lengths is trivial, and implies a change in size, but not a change in decoding speed; also, it is suitable to be used as a core, integrated in a more complex circuit, because it would use only about $1mm^2$ of area in current fabrication technologies.

6 Conclusions

6.1 Review of objectives

The objectives of this thesis were met. Summarizing,

- the Viterbi algorithm was simplified and adapted to a VLSI implementation with negligible loss of decoding performance,
- the ACS operation was pipelined and simplified, which resulted in better circuit performance,
- the hybrid traceforward method was implemented and tested, offering the possibility of one-clock per symbol performance,
- the architecture was tested successfully both in simulation and in the laboratory, meeting all area and speed requirements, and
- the results are better, or comparable to, other reported results.

Stating the Viterbi algorithm as was done in chapters 2 and 3 of this thesis can be considered a relevant contribution, since it had not been reported in this way before. All the opportunities to do parallel operations, and to manipulate the algorithm to better suit a VLSI implementation, are revealed in this way.

6.2 Future work

There are many aspects of channel decoding that must be addressed in a communication system; this architecture deals with symbol decoding, but neglects other problems such as symbol synchronization, bit-error rate monitoring, and configurability. Punctured codes are an especially important application of convolutional codes, and the architecture proposed here can be used to decode them, but not without extra functionality.

To offer a truly complete, useful solution for channel decoding, a circuit must include

all of these functions.

The existence of better decoding techniques, of further refinements in the simplification or the parallelization of the Viterbi algorithm, are of course not ruled out. It is possible, for example, to increase the data rate by using several decoders in parallel. These, and others not yet foreseen, routes remain to be explored.

6.3 Final remarks

It has shown in this thesis how a complex algorithm can be broken down into parts, simplified, adapted, parallelized, and in general converted to a form that is suitable for VLSI implementation, fast, and very close in performance to the original one.

There are two apparently unrelated fields in communications engineering. One field, the scientific, is more academic. Here new algorithms and theories are invented. However, as can be corroborated by reading most current text books, people working in this field are satisfied by showing, at some abstract level, that their new algorithm, equation or theory is correct, in some sense.

The technical field, however, is more preoccupied with bringing solutions to those who need them. Technical engineers work at a more concrete level, where ideas are valued by very concrete parameters like time-to-market, economic and technical feasibility, and resources needed to bring the idea to reality.

This thesis happily founds itself, not by accident, between these two fields, showing that, in fact, they are interrelated and interdependent. An algorithm can be beautiful, elegant, and efficient when seen from a certain perspective; however, if it can't be used to bring better solutions to those in need for them, then, from this different perspective, it is a very poor algorithm indeed.

Both perspectives are valid and in some way the same: it is often seen that the algorithms that the human sense of aesthetics finds most beautiful are those with better and more efficient implementations. How to go from one side - the beautiful idea - to the other - a good implementation - is not always trivial, however, and those working on this problem need to have research papers and textbooks in one hand and a workstation, soldering iron and

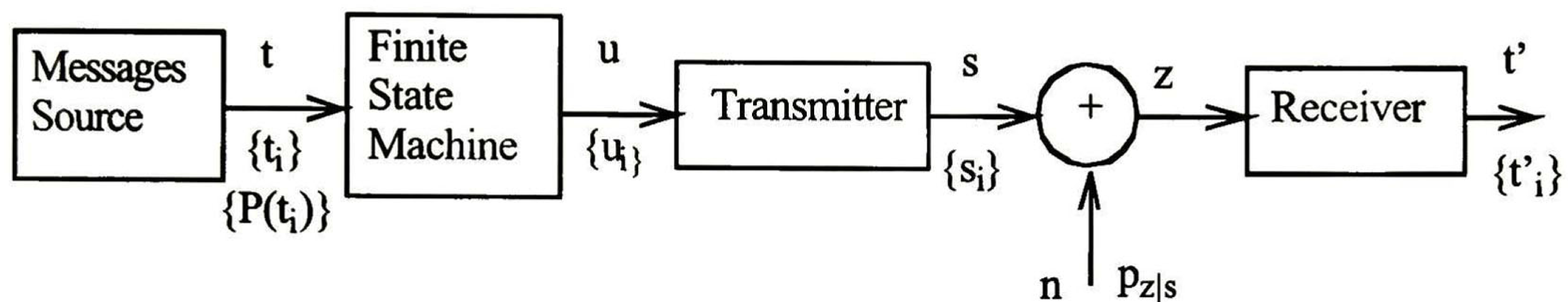
wires in the other. One foot in the academia and another where products are made and sold.

For those who work as interpreters between these two fields, as this thesis shows, there are some very pleasant rewards waiting.

Appendix A

In this appendix, the role of Euclidean distance in the calculation of the likelihood ratios of equations (2) and (3) is explained, justifying equation (4). It is also why the Euclidean distance has been substituted by the absolute value distance.

The following argument follows [25] closely. Consider the following communications system:



The message source produces messages t, out of a possible set $\{t_i\}$, each with probability $P(t_i)$. Each message is independent of all others. The message source produces a sequence of messages called T. Each message enters a finite state machine, producing a transition, labeled u, out of a finite set of transitions $\{u_i\}$. The sequence of transitions produced by the sequence T is called U. The sequence U is a set of dependent symbols; the dependency has been introduced by the state machine. Then, a transmitter (modulator) maps each symbol u to a vector $\mathbf{s}$, out of a finite set of vectors $\{\mathbf{s}_i\}$. The sequence of transmitted vectors is called S. These vectors are contaminated by additive noise. Noise has a density function p_n, and each noise sample n is independent. The sequence of noise samples is called N. The channel is defined by the set of conditional density functions $\{p_{\mathbf{z}}(\mathbf{z}\,|\,\mathbf{s}=\mathbf{s}_i)\}$, denoted, for brevity, $p_{\mathbf{z}|\mathbf{s}}$.

The receiver observes a sequence Z of vectors with noise, and from it, must estimate the sequence T that was originally transmitted. The estimated sequence is called T' To estimate T, the receiver first makes an estimate of the sequence of transmitted signals, S', and from it the estimates U' and T' can be univocally (and trivially) recovered. A rule must be devised for the receiver to assign to each received sequence Z one of the possible transmitted sequences S_i with minimum probability of error. Such a receiver is called the optimum

receiver.

Let it be assumed that the received sequence Z equals R.The probability of correct decision is maximized by mapping R into the sequence S_i for which the a posteriori probability is maximum; that is, if $Z=R$ is observed, then the receiver estimate S' should be set to S_m if and only if

$$P(S_m|Z=R) > P(S_j|Z=R) \quad \forall j \neq m \tag{A.1}$$

That is, the probability of every possible sequence S_i, given that $Z=R$, is calculated, and the S_i with largest probability is chosen. If several sequences have the same probability, one is chosen at random.

Using the mixed form of the Bayes rule, Eq. A.1 can be expressed as

$$\frac{p_{\mathbf{z}}(R|S_m)P(S_m)}{p_{\mathbf{z}}(R)} > \frac{p_{\mathbf{z}}(R|S_j)P(S_j)}{p_{\mathbf{z}}(R)} \quad \forall j \neq m$$

Since the denominator is common to both terms, the inequality can be reduced to

$$p_{\mathbf{z}}(R|S_m)P(S_m) > p_{\mathbf{z}}(R|S_j)P(S_j) \quad \forall j \neq m \tag{A.2}$$

In communication theory, it is often assumed that the message source is source-encoded, which yields a sequence T of independent and equally probable messages. In convolutional coding the finite-state machine does not have an arbitrary structure, but rather regular structure called de Bruijn. This causes that all sequences S are equally probable.

The decision rule of the receiver is, then, to assign $S' = S_m$ when

$$p_{\mathbf{z}}(R|S_m) \tag{A.3}$$

is maximum. Eq. A.3 is commonly called a likelihood function.

It is observed that the received sequence Z is equal to the noise sequence N plus the transmitted vector sequence S; that is,

$$\mathbf{z}_j = \mathbf{s}_j + \mathbf{n}_j$$

where $\mathbf{z}_j$, $\mathbf{s}_j$, and $\mathbf{n}_j$ are the j-th elements of the sequences Z, S, and N, respectively. This implies that $Z=R$ when S_i was transmitted if and only if $N=R-S_i$. Then, the conditional density functions $p_{\mathbf{z}|\mathbf{s}}$ are given by

$$p_{\mathbf{z}}(R|S_i) = p_{\mathbf{n}}(R - S_i|S_i)$$

In most cases, specially those where convolutional codes are used, the noise is statistically independent of the transmitted vectors; that is,

$$p_{\mathbf{n}|\mathbf{s}} = p_{\mathbf{n}}$$

and

$$p_{\mathbf{n}}(R - S_i|S_i) = p_{\mathbf{n}}(R - S_i)$$

The expression $p_{\mathbf{n}}(R - S_i)$ is the likelihood function that must be maximimized over i

In the case studied in this thesis, the noise is assumed to be zero-mean white Gaussian noise with probability density function

$$p_{\mathbf{n}}(\mathbf{a}) = \frac{1}{(2\pi\sigma^2)^{k/2}} \exp\left(-\frac{1}{2\sigma^2} \sum_{j=1}^{k} a_j^2 \right) \tag{A.4}$$

where k is the number of elements in each vector (that is, their dimension) and a_j are the elements of vector $\mathbf{a}$. Since the square of the length of a vector is given by

$$|\mathbf{a}^2| = \mathbf{a} \cdot \mathbf{a} = \sum_{j=1}^{k} a_j^2$$

then Eq. A.4 can be written as

$$p_{\mathbf{n}}(\mathbf{a}) = \frac{1}{(2\pi\sigma_2)^{k/2}} \exp\left(\frac{-|\mathbf{a}|^2}{2\sigma^2} \right) \tag{A.5}$$

Substituting the likelihood function in Eq. A.5, it can be seen that the optimum receiver sets $S'=S_m$ when

$$e^{\frac{-|R-S_i|^2}{2\sigma^2}} \tag{A.6}$$

is maximum for i=m. Maximizing Eq. A.6 is equivalent to minimizing

$$\left|R - S_i\right|^2 \tag{A.7}$$

R and S_i are sequences with elements $\{\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_k\}$ and $\{\mathbf{s}_{i1}, \mathbf{s}_{i2}, ..., \mathbf{s}_{ik}\}$, respectively, and corresponding elements are subtracted, so the minimization rule can be expressed as choosing S_m as the receiver estimate if and only if

$$\sum \left|\mathbf{r}_j - \mathbf{s}_{mj}\right|^2 \tag{A.8}$$

over the length of the sequences R and S_i, is minimum. It is recognized that $|\mathbf{r}_j\text{-}\mathbf{s}_{mj}|$ is the Euclidean distance between vectors $\mathbf{r}_j$ and $\mathbf{s}_{mj}$.

It is interesting to observe how the optimum decision rule as expressed in Eq. A.8 is different from the case when independent vectors are transmitted. In the case of independent vectors, the vector that is closer in Euclidean distance to the one received is chosen as estimate. In the case of dependent vectors, the aggregated distance of a sequence of vectors must be minimized. This means that, for particular elements of the sequence, the optimum rule for independent vectors might not apply, that is, vectors that are not the closest to the received one might be chosen. The overall result, however, is improved error performance.

It was desired to investigate if the Euclidean metric could be replaced by a metric that is easier to calculate in hardware. The absolute-value metric, also known as the Manhattan, city-block, or taxi-driver metric, was the ideal candidate, given its simplicity. As shown in chapter 4, the use of this metric leads to simplification of the ACS circuit.

The absolute-value distance between two vectors, $\mathbf{x}$ and $\mathbf{y}$, with k elements each, is defined as

$$d_v(x,y) = \sum_{j=1}^{k} \left\|(x_j - y_j)\right\|$$

where the double bars indicate absolute value.

In order to change the metric used, it must be shown that the change will not affect the outcome of the minimization in Eq. A.8; that is, if a certain vector is chosen when the

Euclidean metric is used, the same vector should be chosen when using the absolute-value metric; otherwise, a different sequence S_i will be chosen as most likely.

Extensive testing was performed to show that the change of metric did not affect the performance of the decoder. Simulations were done with decoders of constraint length $L=4$ and $L=7$, with E_b/N_o ranging from 1dB to 6dB. In no case a difference in performance between decoders using the two metrics was found.

This, however, does not constitute a rigorous proof that the metrics are interchangeable for the Viterbi decoders investigated. A proof is needed because it is known that, given vectors **x**, **y**, and **r**, the distance between **x** and **r** could be smaller than that between **y** and **r** under one metric, but larger under the other.

A rigorous proof is not complete at this time; a strategy that might lead to it is given. The problem can be formulated in the following way. First, the trellis is divided in sections

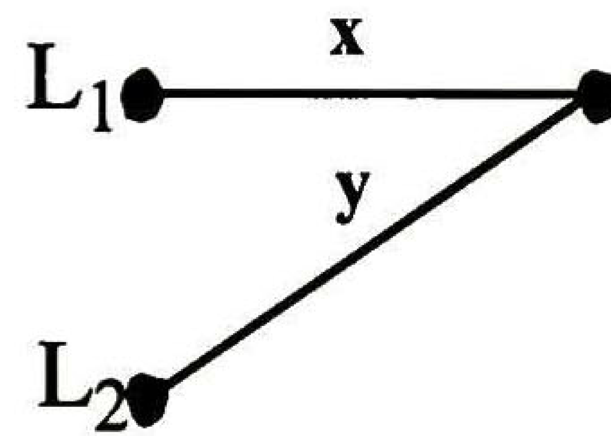


Figure A.1. A section of the trellis

as depicted in figure A.1; the path lengths up to each state are labeled L_1 and L_2, and the vectors associated with each branch are **x** and **y**. For each section of the trellis, and for each received vector **r**, the Viterbi algorithm calculates the distances

$$P_1 = L_1 + (r_1 - x_1)^2 + (r_2 - x_2)^2$$
$$P_2 = L_2 + (r_1 - y_1)^2 + (r_2 - y_2)^2$$

and chooses the smaller. Here, the received vector **r** is compared against the vectors associated with each branch, **x** and **y**. What needs to be proven is that, when replacing the squared Euclidean distance with the absolute-value metric, the same branch will be chosen; this can be expressed as requiring proof that:

$$L_1 + (r_1 - x_1)^2 + (r_2 - x_2)^2 >$$
$$L_2 + (r_1 - y_1)^2 + (r_2 - y_2)^2$$
$$\Leftrightarrow \tag{A.9a}$$
$$L_1 + \|r_1 - x_1\| + \|r_2 - x_2\| >$$
$$L_2 + \|r_1 - y_1\| + \|r_2 - y_2\|$$

and

$$L_1 + (r_1 - x_1)^2 + (r_2 - x_2)^2 =$$
$$L_2 + (r_1 - y_1)^2 + (r_2 - y_2)^2$$
$$\Leftrightarrow \tag{A.9b}$$
$$L_1 + \|r_1 - x_1\| + \|r_2 - x_2\| =$$
$$L_2 + \|r_1 - y_1\| + \|r_2 - y_2\|$$

where the double bars indicate absolute value.

It should be realized that, in the Viterbi implementation used, vectors **x**, **y**, and **r** cannot take arbitrary values. They are two-dimensional vectors, and the only values they can take are:

$$\mathbf{x}, \mathbf{y} \in \{(0,0), (0,7), (7,0), (7,7)\}$$
$$0 \leq r_i \leq 7$$

The strategy suggested is to prove equation A.9 holds for $L_1 = L_2 = 0$, and then use induction to prove they hold for all values of L_1 and L_2.

That Eq. A.9 holds for $L_1 = L_2 = 0$ has been proven by exhaustively testing every possible combination of **x**, **y**, and **r**. There are 64*4=256 possible combinations, and Eq. A.9 holds for all of them.

The proof for general L_1 and L_2 remains to be done.

Appendix B

What is the meaning of E_b/N_0? How is it calculated? What is it used for? These questions are answered in this appendix.

In the study of channel codes, it is often assumed that the channel disturbs the transmitted signal through additive white Gaussian noise. Such a channel is usually referred to as the AWGN channel. The average power of white Gaussian noise is infinite; its power spectral density is a constant, usually called $N_o/2$, and it has zero mean [24].

Since the average power of noise in the AWGN channel is infinite, it might be assumed that it would be impossible to transmit any signal at all. However, although the average power of white Gaussian noise is indeed infinite, only the portion of the noise that is found in the same frequency band as the transmitted signal affects it. From a geometric perspective [1, 25], white Gaussian noise has infinite dimensions; however, when affecting a signal, only those dimensions that correspond to the signal being affected are taken into account. All the other noise dimensions are irrelevant, which explains why transmission in the AWGN channel is possible.

Digital modulation can be very concisely described as the mapping of binary words to signal pulses. Digital receivers operate by trying to detect if one of the signal pulses known to be produced by the transmitter is present at their input. This is different to the approach in analog transmission, where the shape of the signal must be recovered. The reception problem consists in designing a receiver that maximizes the signal-to-noise ratio at its output (or, equivalently [24, 2], that minimizes the average probability of error).

Such a receiver is called the optimum receiver. The optimum receiver is divided in two parts: the demodulator, whose function is to try to find the signal pulse in the received signal, and a decision device that, according to certain criteria, decides whether the signal pulse was present or not.

There are two solutions to this problem, and both are equivalent, in the sense that the

average probability of error is the same for both. When trying to maximize the signal-to-noise ratio at the output of the receiver, the optimum receiver takes the form of a matched filter plus a decision device; when trying to minimize the average probability of error, the correlation receiver is found. Both are shown in figure B.1.

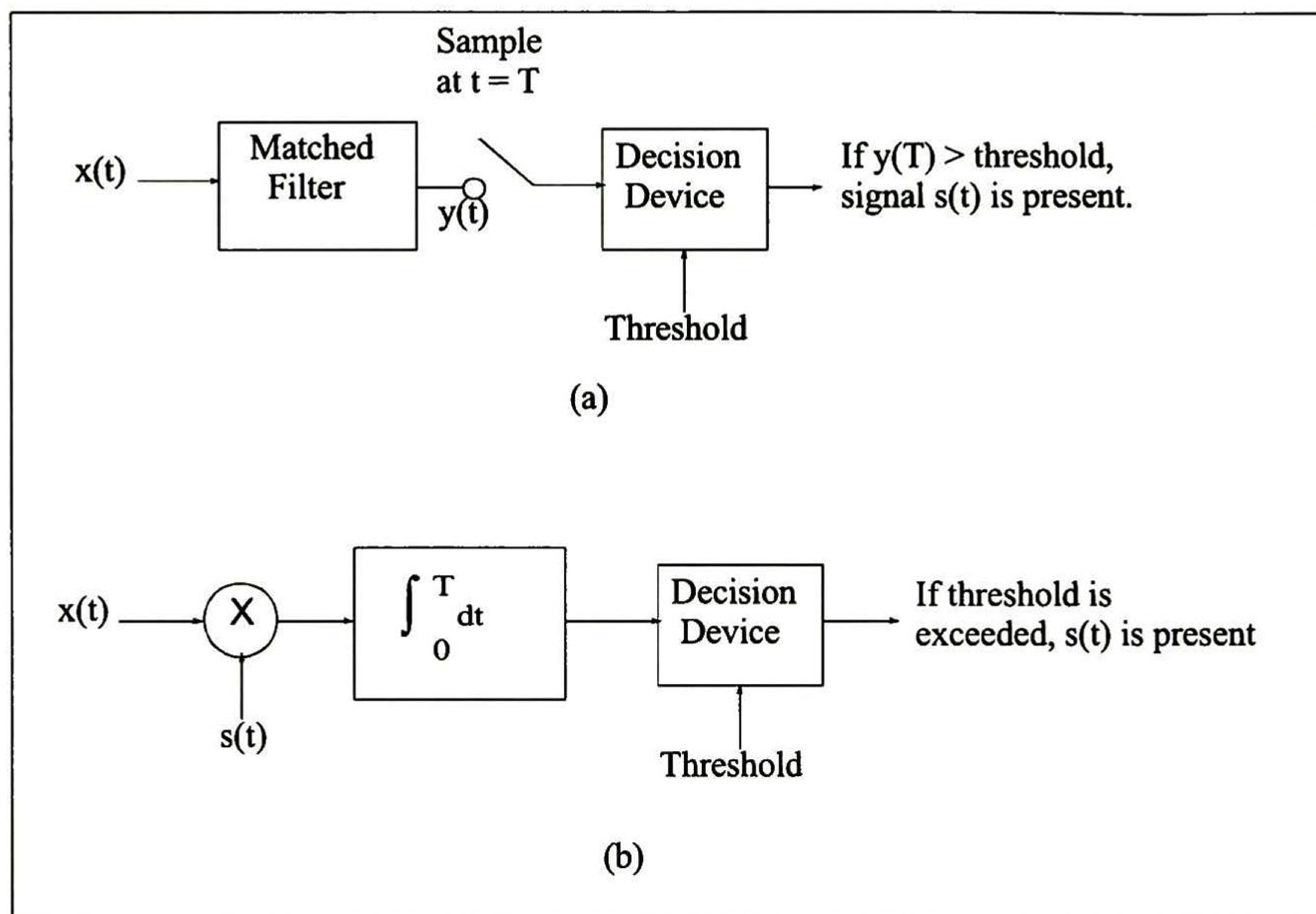


Figure B.1. *(a) The matched filter receiver; (b) the correlation receiver*

The matched-filter implementation can be described as follows. Let $h(t)$ be the impulse response of a linear time-invariant filter with transfer function $H(f)$, and $s_o(t)$ and $n_o(t)$ the signal and noise components of the filter output produced by the input signal component $s(t)$ and the input noise component $w(t)$, respectively, as shown in figure B.2. It can be shown that the filter impulse response $h_{opt}(t)$ that maximizes the signal-to-noise ratio at its output is

$$h_{opt}(t) = \begin{cases} s(T - t), & 0 \leq t \leq T \\ 0, & otherwise \end{cases}$$

where T is the period of the signal pulse $s(t)$. A filter with impulse response $h_{opt}(t)$ is known

as a matched filter to the signal $s(t)$, and its output at time $t = T$ has maximum signal-to-noise ratio.

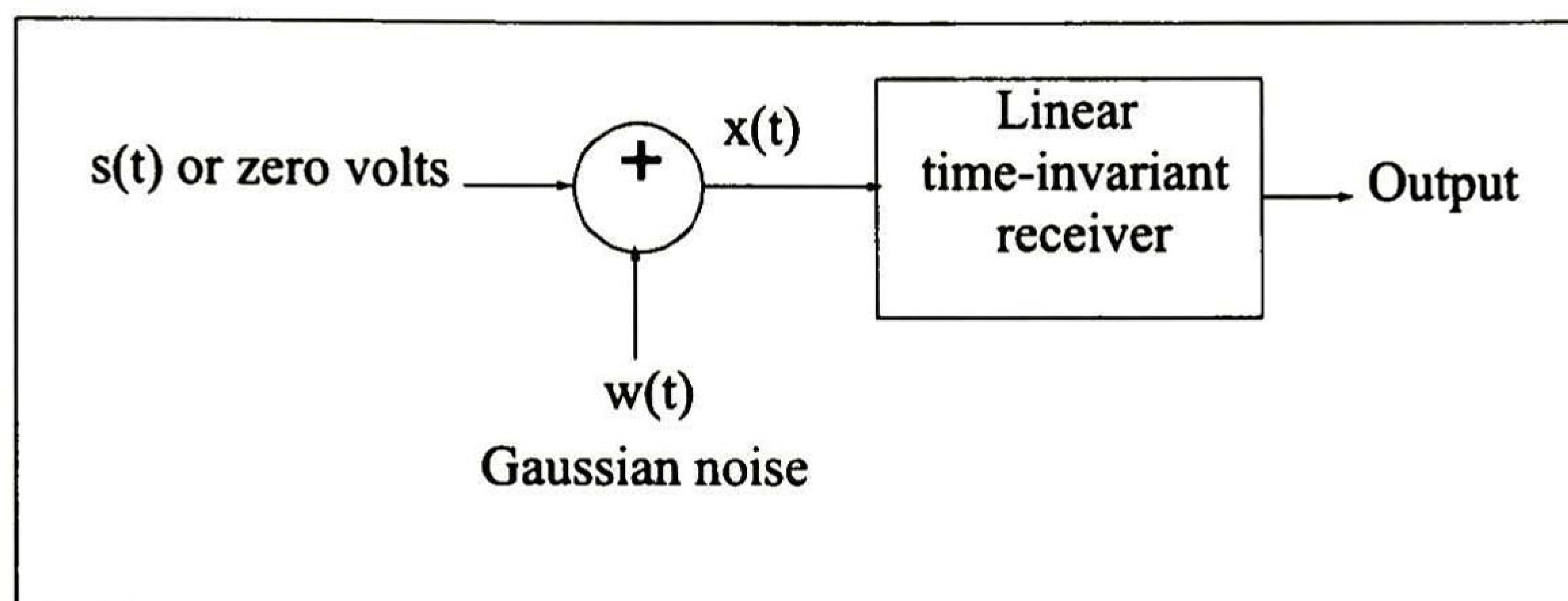


Figure B.2 Signal and noise components of the input of a receiver

The signal-to-noise ratio at the output of a matched filter can be found to be

$$SNR_o = \frac{2E}{N_o}$$

where E is the energy of the signal pulse, or

$$E = \int_0^T s^2(t)dt$$

This result is very important because it establishes that the specific waveform of the signal pulse $s(t)$ is irrelevant to the SNR obtained; only its energy is important. That is, all signals that have the same energy are equally effective at combating the effect of white noise with an optimum receiver. It is also important to note that energy signal and power spectral amplitude of noise are the only two factors that are involved in evaluating the bit-error-rate performance of a receiver.

After the matched filter, a decision device decides whether the signal pulse was actually present or not, based on the amplitude of the matched filter output at time $t = T$. When several different signal pulses are used to transmit different binary words, it is necessary to use a matched filter per pulse, operating in parallel. The decision device then

proceeds to select the signal pulse that corresponds to the matched filter with larger output at time $t=T$, after weighing all outputs by a factor that depends on the probability of each binary word [2].

When using convolutional codes, the optimum decision device takes the form of the Viterbi decoder [22].

Different modulation schemes have different average probabilities of error for a given signal energy. For M-ary modulation, where a single signal pulse carries 2^M bits of information, it's better to consider the energy carried per bit instead of per pulse. The energy per bit is defined as

$$E_b = \frac{E}{M}$$

The E_b/N_o needed to achieve a given probability of error is a figure commonly used to compare the probability of bit error of different modulation schemes or the error-correcting capabilities of different channel codes [1], [2]. This is because

> ... two systems that use an unequal number of symbols may be compared meaningfully only if they use the same amount of energy to transmit each bit of information. It is the total amount of energy needed to transmit the complete message that represents the cost of the transmission, not the amount of energy needed to transmit a particular symbol satisfactorily. Accordingly, in comparing the different data transmission systems, we will use, as the basis of our comparison, the probability of symbol error expressed as a function of the signal energy per bit-to-average noise power per unit bandwidth ratio; that is E_b/N_o [24, p. 586].

It is very common to study and simulate communication systems using the geometric representation of signals, instead of waveforms [1],[2], [23], [25]. It is possible to find the energy per bit from the signal vectors just as it is possible to find it from the pulse waveform. If a signal pulse $s(t)$ is represented by vector $[a_1, a_2, ..., a_n]$ in a n-dimensional orthonormal

signal space, then its energy is given by

$$Es = \sum_{1}^{n} a_j^2$$

This allows the calculation of the signal pulse energy without knowledge of the signal waveform. In the specific case of the QAM modulation used in this thesis, where the transmitted signal is represented by the four vectors [1,1], [1,-1], [-1,1] and [-1,-1], the signal energy is equal to 2, which means that the energy per bit E_b equals 1.

Appendix C

This appendix contains the test plans used to verify that the implementations presented in this thesis actually work as required.

The test methodology for the implementation was to test each module separately, then test its interfaces, and then test blocks composed of several modules, until the higher-level module was tested. Then, a black-box approach to testing was used, where the top-level module was tested without regard for its internal building blocks (except when an error was found).

For brevity, only the black-box tests are included in this appendix. These tests are what ultimately determines if the circuit as a whole works or not. They are, therefore, the most important tests.

The methodology is to run a predefined set of tests, and check that the circuit's output is correct. For simulation, a test bench capable of executing the test cases was used. For testing the physical implementation, functional test circuitry was included in the device, and the tests were run automatically. In this way, the circuit itself can determine if it is performing satisfactorily or not.

The circuit implemented has only one function: to decode a 1/2, $L=7$ convolutional code. It has no configuration options, or gives any status, or is programmable in any way. Because of this, verification concentrates on two aspects: one, that the circuit can decode a variety of different patterns, and two, that the measured P_b vs. E_b/N_o is comparable to that reported in the literature.

Simulation Test Plan

The simulation test plan is a table of cases that must be run. Columns represent different E_b/N_o levels, while rows represent different patterns. In each table cell the expected P_b is listed. The test plan is complete when the circuit presents the expected P_b for all cases listed (see Table C.1).

		E_b/N_o (dB)						
		$N_o = 0$	1.41	1.94	2.5	3.1	3.74	4.44
Pattern	All zeros	0					110×10^{-9}	
	All Ones	0				756×10^{-9}		
	Alternating 1-0	0			3.6×10^{-3}			
	11110000	0						
	2^{20} - 1	0	36×10^{-3}	12.8×10^{-3}	3.6×10^{-3}	756×10^{-9}	110×10^{-9}	14×10^{-9}

Table C.1. *Expected P_b for each pattern run and for each E_b/N_o*

There are two important things to consider when running the test cases described above: how long to run the simulation, and how to compare the results obtained with those in the literature.

Normally, a simulation needs to be run for approximately $10*1/P_b$ bits before results can be considered reliable [23]. While testing the Viterbi implementation, it was found that this number of bits is insufficient to obtain a figure for P_b that can be considered reliable, because a small change in the length of the simulation produced a relatively large change in the measured probability of error. For this reason, the decision was made to run the simulation for at least $100*1/P_b$, which produces more reliable results. The consequence of doing this is that simulation time grows very large for large signal-to-noise ratios.

Most books on digital communications report E_b/N_o vs. P_b for L=7, r=1/2 convolutional codes. However, E_b/N_o vs. P_b is presented as a graph, which is fine for getting a feel for the behavior of the code or for comparison with other codes, but not for getting accurate P_b figures for a given E_b/N_o. This makes it extremely difficult to compare the results presented in this thesis with those reported in the literature with a high degree of precision. To make matters worse, it is frequently not reported what modulation scheme was used to obtain the results claimed (in this thesis, QAM modulation has been used because it suits codes of rate 1/2 very well, and because it makes good use of the available bandwidth). The consequence of this situation is that results presented are considered valid because they are very similar to those found in the literature.

Physical implementation test plan

To test the physical implementation of the Viterbi algorithm, a circuit was designed to generate a data sequence with errors in it, and another circuit to count how many errors the Viterbi circuit was able to correct. Figure C.2 shows a block diagram of how this is done.

The physical implementation has several pins used to control the testing process. The "test" input pin forces a circuit reset, and connects the inputs of the Viterbi decoder to the test data sequence. During the test, the output pin "test_in_progress" is set to 1. When the test is complete, the pin "test_complete" is set to 1, and, if the test was successful, the pin "test_successful" is also set to 1.

The test produces a data sequence of 524,000 bits, and 694 errors are expected. If exactly 694 errors are detected after 524,000 bits have been decoded, then the test is declared successful.

The data sequence used is a 2^{20}-1 sequence like the one used in simulation. In order to insert errors more or less independently, whenever the seven less significant bits of the data generator are equal to zero the output of the generator is inverted. This converts the additive white Gaussian noise channel assumed in the theoretical development of maximum-likelihood decoding to an approximately equivalent binary channel.

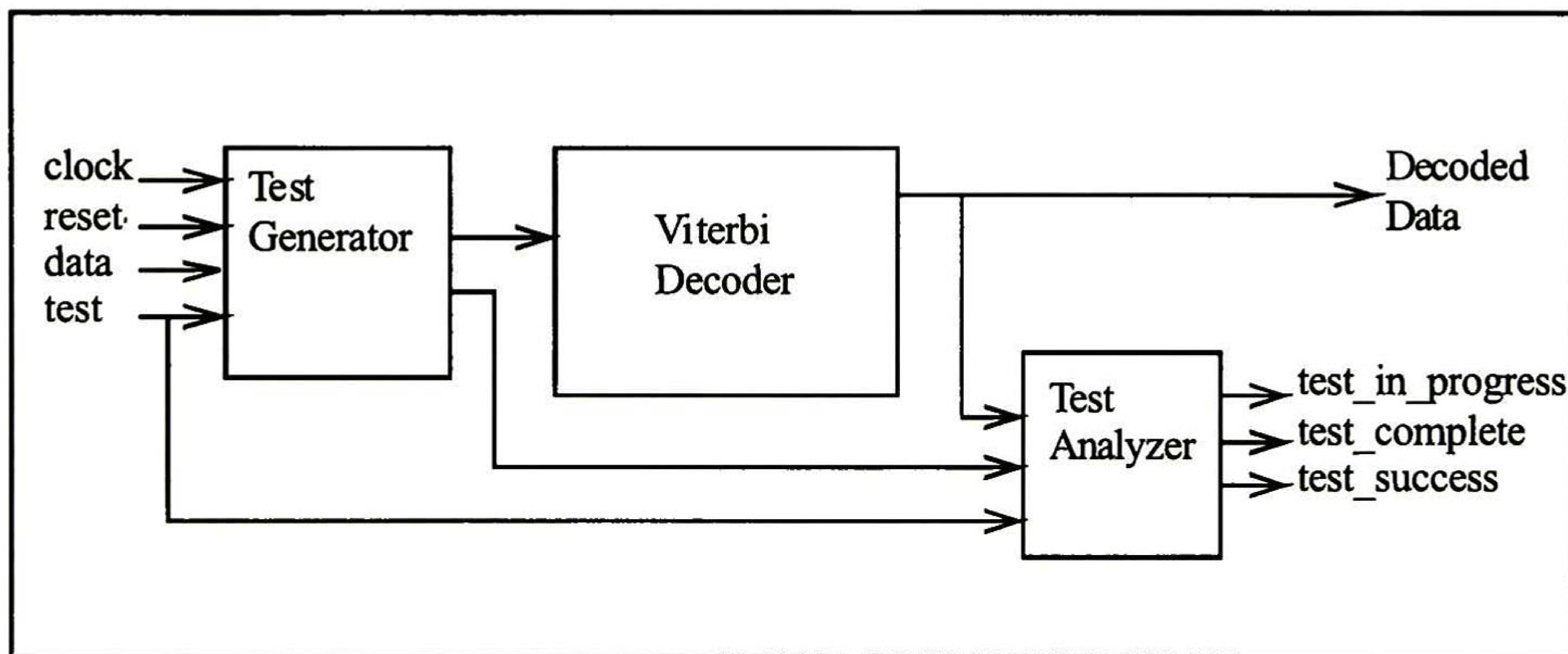


Figure C.2 *Test Circuits*

The Test Generator block in figure C.2 produces an encoded, noisy sequence for the Viterbi decoder; it also reproduces the original data sequence, which is compared with the decoded sequence by the Test Analyzer block.

Appendix D

This appendix describes the contents of the accompanying CD-ROM. The directory structure of the CD-ROM is:

```
\-------- C Code
    |---- VHDL Code
        |---- Testbench
    |---- Other Implementations
      |---- source_ff
      |---- source_altera_1block
```

The C Code directory contains the C code used to simulate the Viterbi algorithm in a PC. The main file is called viterbi.c, which includes two other files, definic.c and funcextr.c. viterbi.c allows running the algorithm several times in a single run, with varying noise power and SPL, and storing the results in a file in Matlab format. definic.c includes all variable definitions. funcextr.c performs several extra functions, like calculating the Euclidean distance between two vectors. These files were compiled with Borland C++ compiler version 4.

The VHDL Code directory contains the actual circuit that was implemented and tested. The design's hierarchy is shown in figure D.1.

The Testbench directory contains three VHDL files that were used to simulate the Viterbi circuit. The file infosource.vhd simulates an information source, a convolutional coder, a modulator, the additive white Gaussian noise channel, and a soft demodulator. The sink.vhd file compares the output of the Viterbi decoder with the original data source and calculates how many errors there are. The file testbed.vhd connects all circuits together for simulation.

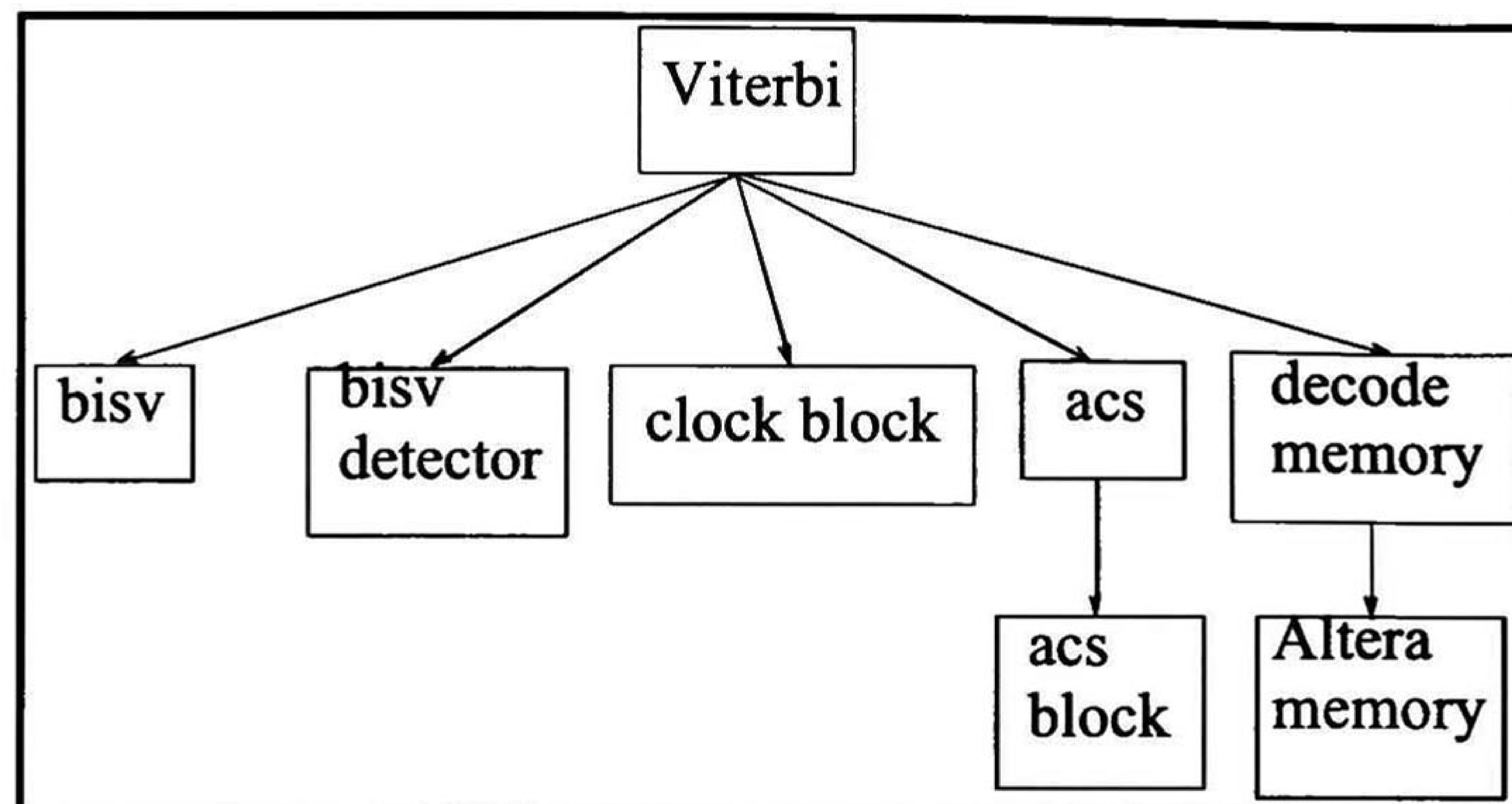


Figure D.1. *Design Hierarchy*

Each block in the figure above has a corresponding VHDL file.

The Other Implementations directory contains two alternative implementations of the algorithm that cannot be fit into the Altera device used, but are interesting and worth presenting. The source_ff directory contains an implementation that uses flip-flops instead of RAM memory. This implementation is very fast, but takes up a large amount of area. If area is not a concern, this implementation should be chosen over the one presented above.

The source_altera_1block directory contains an implementation using a hypothetical RAM block that is truly synchronous (not like the ones currently offered by Altera; see Chapter 4). If such a memory ever becomes available from Altera, this source should be used, as it twice as fast as the one presented above. This implementation should be easy to convert to other FPGA technologies that offer this kind of memory.

References

[1] B. Sklar, "Digital Communications", Prentice Hall, 1998.

[2] J. Proakis, "Digital Communications", 3rd Edition, McGraw-Hill, 1995.

[3] A. Viterbi and J. Omura, "Principles of Digital Communication and Coding", McGraw-Hill, 1979.

[4] G. D. Forney, "The Viterbi Algorithm", Proceedings of the IEEE, Vol. 61, No. 3, March 1973.

[5] R. Cypher and C. B. Shung, "Generalized Trace Back Techniques for Survivor Memory Management in the Viterbi Algorithm", 1990.

[6] H. L. Lou, "Implementing the Viterbi Algorithm", IEEE Signal Processing Magazine, Sept. 1995.

[7] A. Vitayev and P. H.Siegel, "On Viterbi Detector Path Metric Differences", IEEE Trans. on Comm., Vol. 46, No. 12, Dec. 1998.

[8] A. J. Viterbi, "Convolutional Codes and Their Performance in Communications Systems", IEEE Trans. on Telecomm. Tech., Oct. 1971.

[9] P.J. Black and T.H.Y. Meng, "Hybrid Survivor Path Architectures for Viterbi Decoders", Proceedings CASSP, pp. I-433-I-436, 1993.

[10] G. Feygin and P. G. Gulak, "Survivor Sequence Memory Management in Viterbi Decoders", Technical Report CSRI-252, University of Toronto, Jan. 1991.

[11] C. M. Rader, "Memory Management in a Viterbi Decoder", IEEE Trans. on Comm., Vol. COM-29, No. 9, Sept. 1981.

[12] P. G. Gulak and E. Shwedyk, "VLSI Structures for Viterbi Receivers: Part I General Theory and Applications", IEEE J. on Select. Areas in Comm., Vol. SAC-4, No. 1, Jan. 1986.

[13] C. B. Shung, H.-D. Lin, R. Cypher, P. H. Siegel, and H. K. Thapar, "Area-Efficient Architectures for the Viterbi Algorithm - Part I: Theory", IEEE Trans. on Comm., Vol. 41, No. 4, April 1993

[14] G. Fettweis, H. Meyr, "High-Speed Parallel Viterbi Decoding: Algorithm and VLSI-Architecture", IEEE Comm. Mag., May 1991

[15] M. Bóo, F. Argüello, J. D. Bruguera, R. Doallo, and E. Zapata, "High Performance VLSI Architecture for the Viterbi Algorithm", IEEE Trans. on Comm., Vol. 45, No. 2, Feb. 1997

[16] -----, Altera FLEX10K Data Sheet Version 4.01, Altera Corporation, June 1999

[17] -----, Qualcomm Corp., Q1900 Viterbi Encoder/Decoder Data Sheet, 1998

[18] http://www.mot.com/SPS/PowerPC/AltiVec/

[19] http://www.mentorg.com/inventra/

[20] http://www.chips.ibm.com/products/asics/products/cores/briefs/viterbi_decoder.html

[21] C. Thomas, "Elements of Information Theory", John Wiley, 1991

[22] J. Omura, "On the Viterbi Decoding Algorithm", IEEE Transactions on Information Theory, Vol. IT15, Jan. 1969, pp. 177-179.

[23] M. Jeruchim, P. Balaban, K. Shanmugan, Simulation of Communication Systems, Plenum Press,1992.

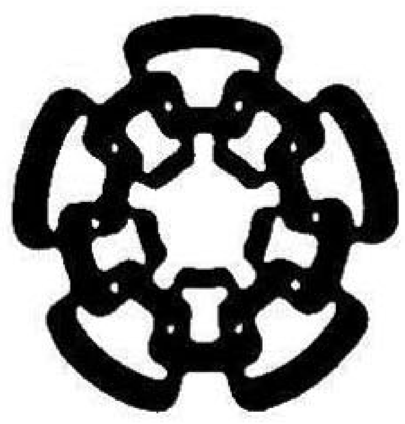
[24] S. Haykin, An Introduction to Digital and Analog Communications, Wiley, 1989

[25] J. Wozencraft, I. Jacobs, Principles of Communication Engineering, Waveland Press, 1965

[26] J. B. Cain,G. C. Clark and J. Geist, "Punctured Convolutional Codes of Rate (n-1)/n and simplified Maximum Likelihood Decoding", IEEE Trans. On Inf. Theory, vol. IT-25, January 1979, pp. 97-100.

[27] E. J. McCluskey, Logic Design Principles with Empahsis on Testable Semicustom Circuits, Prentice Hall, 1986.

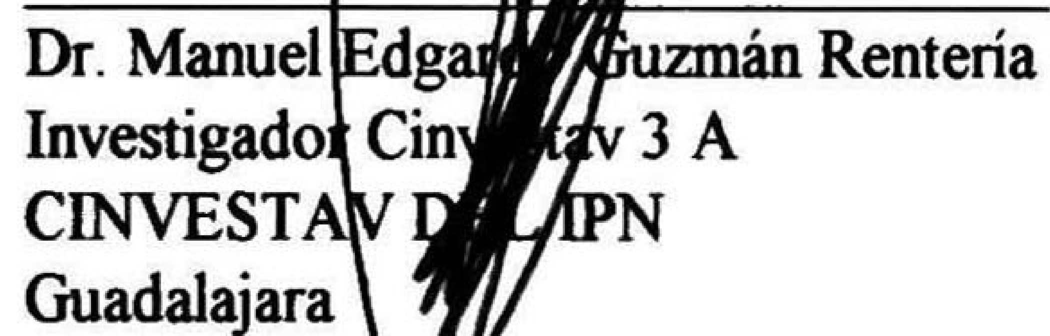
[28] G. R. Burke, "Case Study of the Design of a Viterbi Decoder", Design SuperCon Conference Proceedings, 1997.



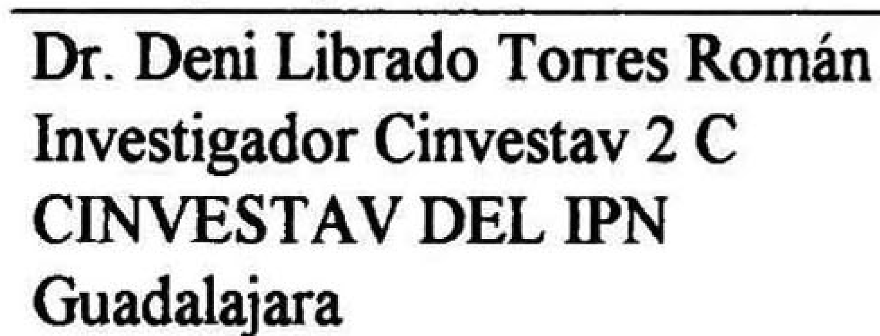
CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL IPN

UNIDAD GUADALAJARA

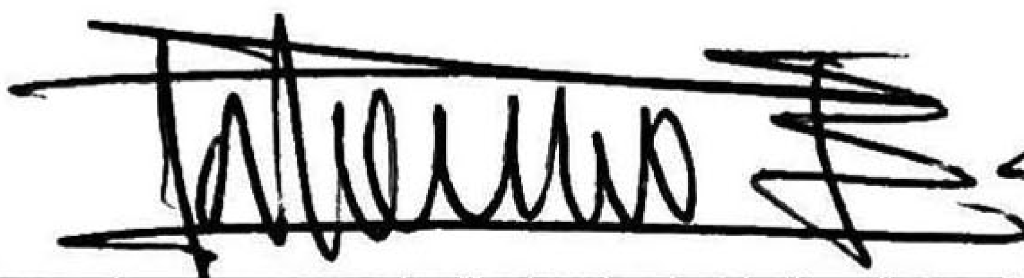
El Jurado designado Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, aprobó la tesis: "A VLSI Implementation of the Viterbi Algorithm" que presenta el Ing. Luis Miguel Bazdresch Sierra el día 28 de Julio de 2000.



Dr. Manuel Edgardo Guzmán Rentería
Investigador Cinvestav 3 A
CINVESTAV DEL IPN
Guadalajara



Dr. Deni Librado Torres Román
Investigador Cinvestav 2 C
CINVESTAV DEL IPN
Guadalajara



M. en C. Jesús Palomino Echartea
Director General
TDCOM, S.A. de C.V.
Guadalajara.

PROL. LOPEZ MATEOS SUR 590 APARTADO POSTAL 31-438 GUADALAJARA, JAL. 45090 TEL. (3) 684-1580 FAX (3) 684-1708

CINVESTAV
BIBLIOTECA CENTRAL

SSIT000003850