Coding Beyond the Computational Cutoff Rate

Thesis by

Oliver Collins

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1989

(submitted May 11, 1989)

Acknowledgement

The author wishes to thank his advisor, Dr. Robert J. McEliece, for many things, especially co-authoring the paper that is Chapter 5 of this thesis.

Abstract

This thesis presents a collection of new codes, algorithms, and hardware, which can all be used to reduce the required energy per information bit to noise spectral density ratio on the Gaussian channel. First comes a feedback technique from an outer to an inner code. The basic idea is to perform a second maximum likelihood decoding operation of the inner code that incorporates side information. Next comes a new kind of algebraic outer code which we get from combining Reed Solomon codes with themselves. The most important results, however, deal with the construction of long constraint length Viterbi decoders. One chapter presents a hardware design of a constraint length 15, rate 1/6 decoder. The last chapter gives some results on the partitioning of a deBruijn graph which make the number of interconnections in the design physically realizable.

Table of Contents

Figures

Table

To

Abigail Anne Collins (1949-1972)

Introduction

Traditionally people view coding as the sacrifice of bits that could be used to carry information in order to achieve greater reliability, e.g., a Hamming code in a computer memory. Coding is, however, a general way to exchange complexity at the transmitter and receiver for reduced medium cost per information bit at any given error rate. Coding's benefits become greater as the allowable error rate approaches zero where the medium cost of an uncoded system becomes infinite, whereas that of a coded system approaches a constant called channel capacity.

For magnetic tape the medium price is just the cost of a reel divided by the number of bits it can hold. For a deep space probe, it is the transmitter power divided by the data rate; i.e., it is measured in Joules per bit of information sent at whatever bandwidth expansion factor the equipment allows. As the complexity increase falls disproportionately on the ground-based receiving equipment and the medium cost is astronomical (approximately fifteen million dollars for a twenty-percent increase in power), the deep space channel is an ideal place to apply coding, and many of the examples in this thesis use it. The deep space channel (Reference 1) is an analog, white Gaussian channel with two-level signalling; i.e. the transmitter sends out symbols that can have either of two values (say -1 or +1), and the output of the radio frequency equipment on Earth is a voltage proportional to the transmitted symbol plus an independent Gaussian random variable whose variance is proportional to the spectral density of the noise, N_0.

The goal of coding is to minimize $E_s/\mathrm{R}N_0$, where E_s is the energy per symbol and 1/R is the average number of symbols devoted to each information bit. The capacity of this channel, the minimum ratio of the energy per information bit to the noise spectral density, is upper-bounded by and for small R very close to

$$\frac{2^{2R}-1}{2R}. \tag{1}$$

This is the capacity of the channel if the input is not restricted to only two levels and comes from the more usual

$$C = \frac{1}{2}\log(1+2E_s/N_o) \tag{2}$$

when E_s is set equal to $\mathrm{R}E_b$.

The conversion of discrete to continuous values greatly complicates the problem of designing a coding system that approaches channel capacity. The transmitter outputs discrete binary symbols, but what arrives at the decoding equipment is an analog estimate of whether a $+1$ is more likely to have been sent than a -1. The problem is that algebraic decoding methods, the only known ones that allow an exponential reduction in decoded error rate with only polynomial complexity, cannot incorporate the fuzzy information produced by the channel. This problem has led to the use of concatenated codes when error rates must be very low, where an inner code that can incorporate soft decision information is concatenated with an algebraic code. The original channel combined with the inner code forms another channel over which the outer code operates. Unfortunately, this outer channel must have a lower capacity than the original channel; no matter how good the outer code, the loss cannot be recovered. Said differently, the potential performance of the concatenated code viewed as a whole is much greater than the serial, concatenated decoding algorithm manages to exploit.

Chapter two deals with a modification of the classic Viterbi algorithm for maximum likelihood decoding of convolutional codes, which allows some feedback from the outer algebraic code to the inner code. The combined scheme is superior to convolutional coding alone even at the high allowable error rates characteristic of uncompressed images. Chapter three tells about improved algebraic codes formed by combining Reed-Solomon codes with themselves. The combination of these two completely debunks the idea of a computational cutoff rate by coming within 1dB of channel capacity on the analog Gaussian channel. With only twenty percent left until the ultimate limit trying to save the idea of a hard limit imposed by decoder complexity limitations would be not only futile but boring.

Chapter four describes the hardware and algorithmic details required to build powerful, i.e., long constraint-length, convolutional decoders. The principal problem, as with many large machines, turns out to be internal communication, and the final solution to it is a problem in graph theory, which is presented in Chapter five. The basic structure of maximum likelihood decoding of convolutional codes comes from the state diagram of the shift register encoder. This graph, called a de Bruijn graph, appears in many other contexts, e.g., as the constant dimensional FFT. Hence the results in the last chapter should have application beyond coding theory.

CHAPTER II

Decoding with Side Information

A convolutional encoder consists of a shift register (whose length is called the *constraint length* of the code), a block of linear combinational logic, and a multiplexer. As Figure 1 shows, the information to be encoded is clocked into the shift register and the output of the multiplexer goes to the radio frequency modulator. The multiplexer makes one complete sweep every time the shift register is clocked. The number of outputs of the combinational logic (inputs to the multiplexer) is 1/R where R is the rate of the code. The combinational logic could be made nonlinear or even time varying without changing the decoding algorithm much and we will see later that time varying codes may have some advantages.

Before talking about the maximum likelihood decoding of these codes, we must define in what sense we mean maximum likelihood. We could for instance ask for maximum likelihood on a bit-by-bit basis. This scheme would minimize the decoder bit error probability; however, it is extremely difficult to implement and is only slightly superior to another much simpler decoding technique, Viterbi decoding. Viterbi decoding finds the most likely sequence of bits between known starting and ending states of the encoder shift register. Figure 2 shows how it works. Each column of dots represents the four possible states of a two bit shift register encoder. The lines connecting successive columns represent the possible state transitions as a new bit is shifted into the register. Each line corresponds to a use of the channel, e.g., the modulation

of the radio frequency carrier by the sequence of symbols produced by the combinational logic.

The decoder sees the symbols produced by the encoder after the channel noise has corrupted them. If each of the branches of the graph is given a length equal to the logarithm of the probability that the symbol sequence which it calls for was sent, then the shortest path through the graph that connects two points will correspond to the most likely sequence of transmitted symbols between these two fixed states of the encoder. For the Gaussian channel some simplification is possible, and these lengths can just be the received vector of voltages multiplied by the vector of symbols which the encoder would generate if it took the corresponding branch.

Finding the shortest path through the graph is quite easy because none of the lines jumps over any of the columns of points. Thus, one simply has to find the shortest path to every point of each column in turn. This technique is called Viterbi decoding. Viterbi decoding can even be made free of the need to operate between fixed encoder states, because when proceeding backward through the trellis the number of possible shortest paths can be no more than the number of states and at each step there is always some chance that two will merge. This convergence means that beyond a certain distance all paths will be the same with arbitrarily high probability and makes continuous encoding and decoding possible; i.e., whichever starting state is chosen, the decoding result will be the same.

The whole purpose of this chapter is to point out that the decoding algorithm can use information other than the received symbols. Its effect is to prune the trellis of state transitions if it is hard or modify their lengths if it is soft. The lower half of Figure 2 shows what happens to the trellis when one of the information bits is known, and Table 1 presents the results

of simulations of the constraint length 15 rate 1/4 convolutional code to be used on the Galileo spacecraft first with no side information and then with different types. These simulations come from a hardware decoder using 254 levels of input quantization and a traceback depth of 170. Each line in the table required a run of 450,000 bits. The table gives the bit and symbol error rates as a function of signal to noise ratio expressed in dB. A symbol here is just a group of eight consecutive bits; i.e., it refers to the basic unit of an outer code that processes bytes. A Reed-Solomon code over GF(256) is such a code. This dichotomy is confusing but entrenched. (For ten-bit symbols the error rates would be about ten percent higher and for twelve-bit symbols they would be about thirteen percent higher.) The signal to noise ratios in the table must be converted to information bit signal to noise ratios in order to be meaningful; i.e. some of the bits going into the shift register carry no information; they are forced to a predetermined value. To account for this loss, one has to subtract the fraction of bits carrying information (expressed in dB) from the SNR figure in the table. This number appears at the top of each subtable. It is also necessary to divide the error rates by the same fraction to compare the code with known bits to the original. These quotients are the corrected error rates in the table.

Decoding with some bits known appears to be a losing game when corrected bit error rates are compared at the same information bit signal to noise ratio. However, two simple observations show why the process is worthwhile. The first is that if the known bits are separated by a distance less than the constraint length, then the decoder complexity is reduced. Consider the data for every tenth bit known. The decoder complexity will be comparable to a constraint length 14 code because no more than 14 unknown bits ever appear in the encoder shift register at the same time. There are 2^{14} instead of 2^{15}

states in one column of the trellis. The performance is, however, very close to the original constraint length 15 code.

The second observation is that the known bits can come from the decoding of an outer code whose parameters are chosen so as to make the probability of failure to decode negligibly small, say less than one in a million. If the bits going into the shift register are protected by interleaved outer codes, then the decoding of one of the outer codewords followed by a second convolutional decoding operation using this information will greatly decrease the error rates of the symbols in the remaining outer codewords. The power that was used by the symbols of the first outer codeword is now available to the the symbols of the others.

As an example consider what happens when every other 8 bit symbol is known. Looking at Table 1 we see that the error rate of the remaining symbols has dropped to a very small value. We may either choose to accept this under one in a thousand error rate on half the symbols or clean it up with another outer code. The redundancy of the code required to clean up these errors will be much less than one percent. The complexity of the second convolutional decoding operation will be very small because the machine used to perform it needs only 128 states. Thus, with very little hardware we can achieve the equivalent of perfect erasure declaration.

Suppose we do not require extremely low error rates; e.g., we want to transmit uncompressed images. We can still benefit from redecoding by choosing 0dB as the inner-code operating point and protecting every fifth symbol with one of the outer codes described in Chapter III (or, for that matter, an ordinary Reed Solomon code). To achieve a one-in-a-million chance of failing to decode, a Chapter III code will require a redundancy of 15 percent. Since only 1/5 of the data is protected, the average redundancy

will be .03. Thus, the information bit signal to noise ratio of the combined code with redecoding is .132dB, and 4/5 of the data will experience a bit error rate of four in a thousand, while 1/5 will have an error rate lower than one in a million. This compares with a .5dB signal to noise ratio for a five-in-a-thousand bit error rate. Thus we are saving almost .4dB by adding a block code and making a second pass. The complexity increase is almost the same as that of going to a constraint length 16 code which yields less than a .1dB improvement.

We can, of course, perform multiple re-decoding operations. However, returns diminish, and the whole arrangement becomes quite baroque. Making two passes we can achieve an operating point of .205 dB at an error rate below one in a million. The capacity of the Gaussian channel at this rate is only -.875 dB; thus, we are within 1 dB of channel capacity.

Figure 1
Convolutional Encoder

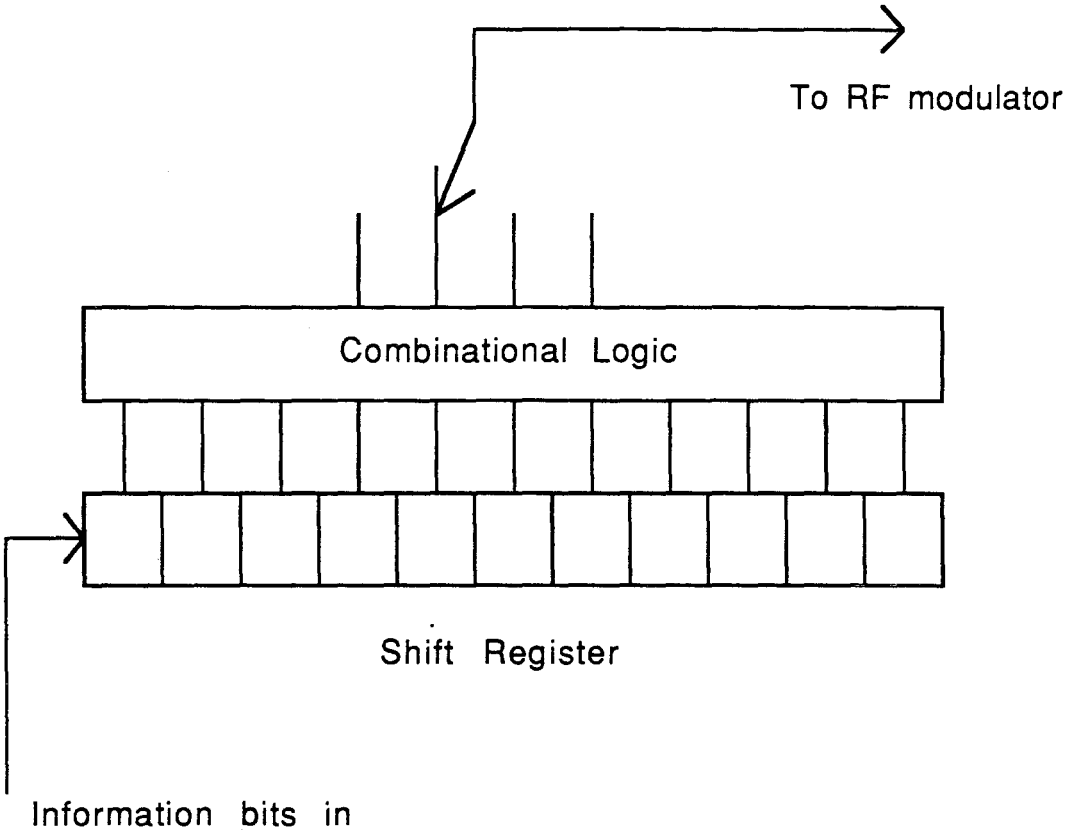
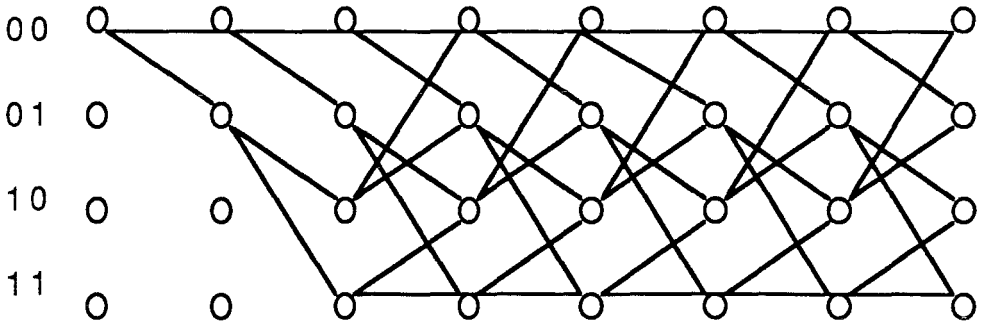


Figure 2
Trellis Diagrams



State diagrams for two bit shift register

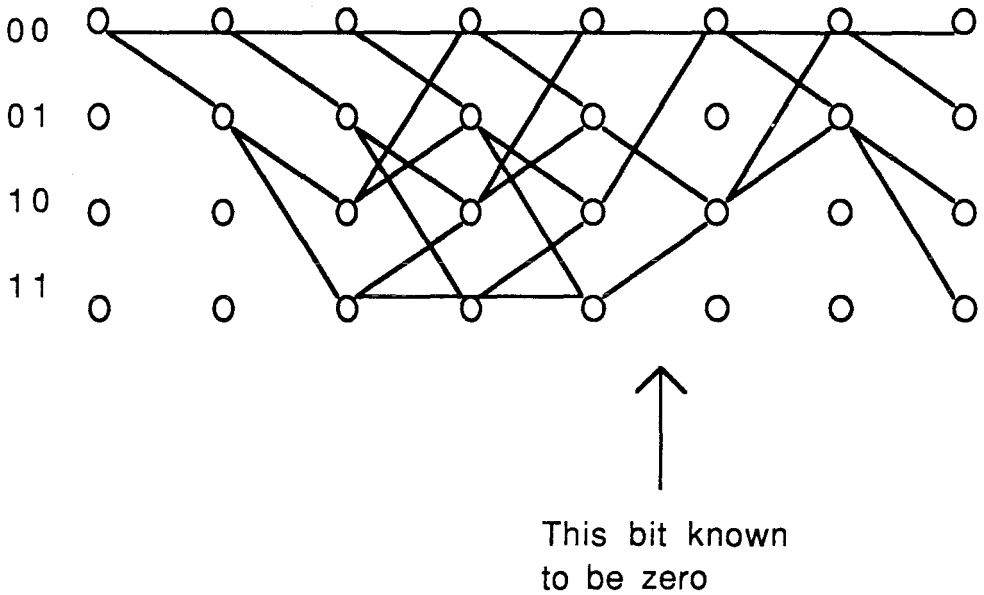


Table 1

No Bits Known
(Conventional Viterbi Decoding)

SNR in dB	Bit error rate	8 bit symbol error rate
.6dB	.00359	.00838
.5dB	.00534	.0124
.3dB	.0110	.0247
0dB	.0268	.0589
-.2dB	.0474	.1020

One Bit Out of Ten Known
(10log(.9)=-.4576)

SNR in dB	Bit error rate	8 bit symbol error rate	Corrected bit error rate
.3dB	.00214	.00575	.00238
0dB	.00669	.0166	.00743
-.2dB	.0124	.0304	.0138

Table 1
(Continued)

Every Other Symbol Known
(10log(.5)=-3.0103)

SNR in dB	Bit error rate	8 bit symbol error rate	Corrected bit error rate	Corrected symbol error rate
.3dB				
0dB	.000177	.000350	.000354	.000700
-.2dB	.000177	.000350	.000354	.000700

Every Third 8 Bit Symbol Known
(10log(.666)=-1.765)

SNR in dB	Bit error rate	8 bit symbol error rate	Corrected bit error rate	Corrected symbol error rate
.3dB				
0dB	.000868	.00230	.00130	.00345
-.2dB				

Table 1
(Continued)

Every Fourth 8 Bit Symbol Known
(10log(.75)=-1.249)

SNR in dB	Bit error rate	8 bit symbol error rate	Corrected bit error rate	Corrected symbol error rate
.3dB				
0dB	.00191	.00472	.00255	.00629
-.2dB	.00341	.00826	.00455	.0110

Every Fifth 8 Bit Symbol Known
(10log(.8)=-.969)

SNR in dB	Bit error rate	8 bit symbol error rate	Corrected bit error rate	Corrected symbol error rate
.3dB	.00134	.00341	.00168	.00426
0dB	.00322	.00785	.00403	.00981
-.2dB	.00466	.0114	.00583	.0143

Table 1

(Continued)

Every sixth 8 Bit Symbol Known
(10log(.8333)=-.7918)

SNR in dB	Bit error rate	8 bit symbol error rate	Corrected bit error rate	Corrected symbol error rate
.3dB				
0dB	.00351	.00784	.00421	.00941
-.2dB	.00800	.0183	.00960	.0220

Every Seventh 8 Bit Symbol Known
(10log(.8571)=-.6695)

SNR in dB	Bit error rate	8 bit symbol error rate	Corrected bit error rate	Corrected symbol error rate
.3dB				
0dB	.00584	.0124	.00681	.0145
-.2dB	.0117	.00251	.0137	.00293

CHAPTER III

Alegraic Codes

A Reed-Solomon (Reference 2) code has symbols in some Galois field and one fewer symbol in a codeword than there are elements in the field; e.g., 8 bit Reed-Solomon codes have 255 symbols. Any number of these symbols can be parity checks; the rest will be information. If the number of parity check symbols is P, then the code can correct E symbol errors and R symbol erasures if and only if $2E + R <= P$. These codes are fairly well suited to and often used on the independent symbol error channel (possibly, also, with erasures) and other channels which can be converted into the independent symbol error channel with only a small loss of capacity. The channel formed by a convolutional encoder and Viterbi decoder operating over the Gaussian channel is such a beast because the errors come in moderately long bursts. The first step in converting this bursty channel into the independent symbol error channel is to group consecutive groups of bits to form symbols. The second is to group the symbols into codewords. We do not, however, want to choose consecutive symbols to be members of the same codeword. The spacing needs to be wide to achieve independence. The loss of capacity will be small only if the burst length is comparable to the symbol size; e.g., a very small burst will destroy the whole symbol in which it resides.

A Reed-Solomon code that experiences symbol error probability e will always have to have a rate less than 1-2e if the probability of its failing to decode must be low. Very big Reed-Solomon codes will approach this value.

However, big Reed-Solomon codes demand big symbols and we saw, above, an instance where an increase in the symbol size also caused an increase in e. For the Viterbi decoder channel, the increase in the symbol error rate with symbol size is gradual. For other channels such as the photon counting optical channel, the symbol size is dictated by the modulation technique, and any deviation drastically increases the error rate. The growth of symbol error rate with symbol size is not the only thing that keeps us from using large Reed-Solomon codes. The decoder complexity is quadratic in the codeword length. What we want is to combine the good features of short and long Reed-Solomon codes.

This chapter describes a new type of code formed by combining Reed-Solomon codes with themselves. If a fairly short (say 8 bit) Reed-Solomon code is designed to have a one-in-a-million chance of failing to decode at some symbol error rate, then there is still a good chance that it will decode successfully even if many of the parity check symbols are erased; i.e., many of the parity check symbols will probably not be needed to achieve succesful decoding. The idea of this chapter is to share these parity checks among many codewords by using a second set of Reed-Solomon codes.

Figure 3 illustrates the concept. The data are first encoded conventionally into Reed-Solomon codewords. These codewords could be sent over the channel as they stand. Instead, we take all the parity check symbols to the right of some position and make them symbols in another Reed-Solomon codeword. These symbols are never sent over the channel at all. The parity of the second Reed-Solomon codeword goes in their place. The decoding algorithm first attempts to decode all of the primary codewords, treating those symbols not sent as erasures. Most of these codewords will decode successfully, and the probability of an erroneous decoding will be orders

of magnitude lower. The decoding of most of the primary codewords provides the information symbols of the secondary codewords. When a primary codeword fails, an erasure is declared in all the secondary codewords. The secondary codewords thus see the original channel error probability on some of their symbols and the rest experience an erasure rate determined by the design of the code. When the secondary codewords decode, the failed primary codewords get all their parity checks. The failed primary codewords will now decode. The increase in decoder complexity is small because only a few of the primary codewords require two decoding attempts, and the number of secondary codewords is small. Figures 4, 5, and 6 present results for different error rates. All that is necessary to verify them is a set of tables of the binomial distribution and a simple result:

Suppose that we have a set of coins each of which has a probability p of turning up heads, and that each coin landing heads up counts one point. If the probability of getting more than A points is very small, then it will drop still farther if one or more coins in the original set is each replaced by two coins whose probability of heads is p/2 and whose values are 1/2 point.

This result is essentially just a specialized form of the law of large numbers. A distribution will pull in towards its mean as the number of independant events wich compose it increases. Here we have kept the expected value of A constant but more independant coin tosses go into determining its actual value in a particular game.

An obvious generalization of these techniques is to split the secondary codewords into two groups and play the game twice. Another is to let the primary and secondary codewords have elements in different fields.

Figure 3

Symbol error rate=.04
Redundancy of normal 8 bit Reed Solomon Code=.2196
at 10^-6 failure to decode
dB adder=1.077dB

Redundancy of new code=.1476
dB adder=.694dB

Redundancy of 10 bit Reed-Solomon code=.1447
at 10^-6 failure to decode
dB adder =.679dB

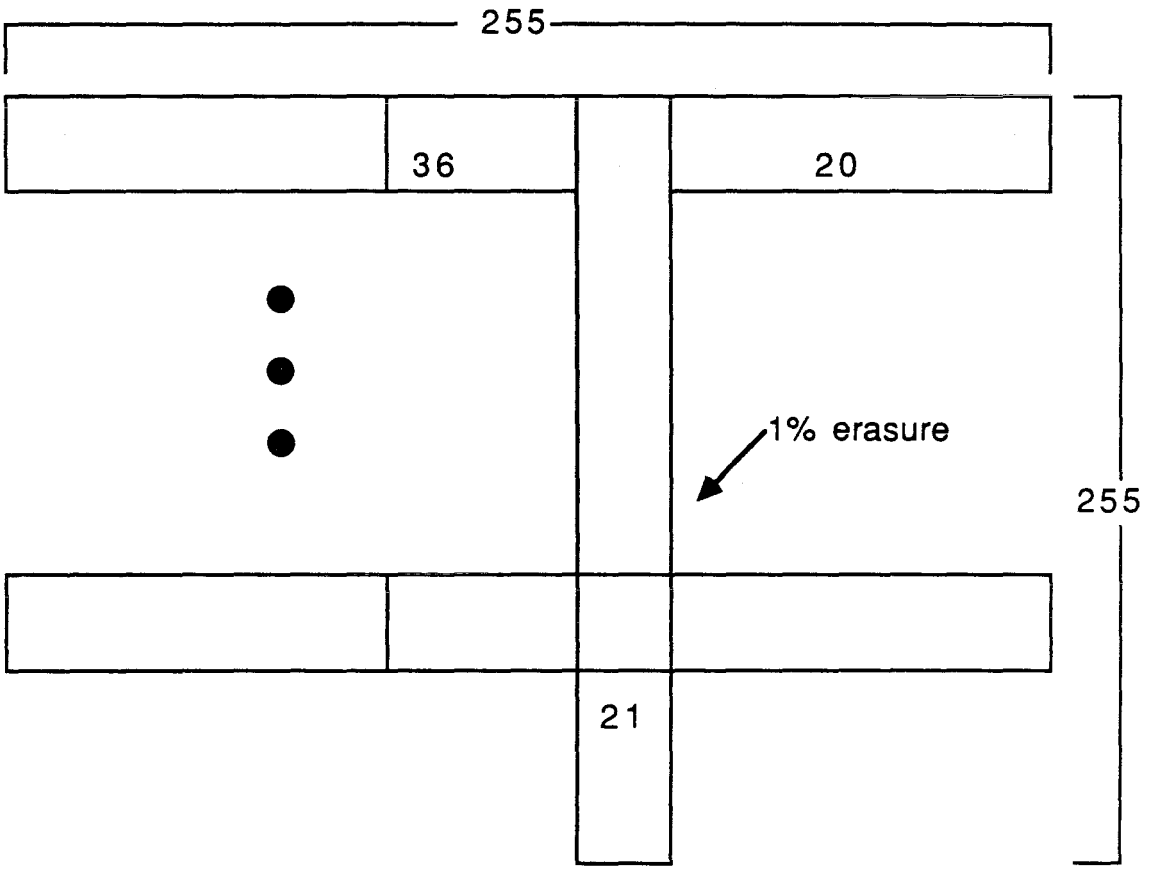


Figure 4

Symbol error rate=.02
Redundancy of normal 8 bit Reed Solomon Code=.1490
at 10^-6 failure to decode
dB adder=.7007dB

Redundancy of new code=.0889
dB adder=.4046dB

Redundancy of 10 bit Reed-Solomon code=.0880
at 10^-6 failure to decode
dB adder =.400dB

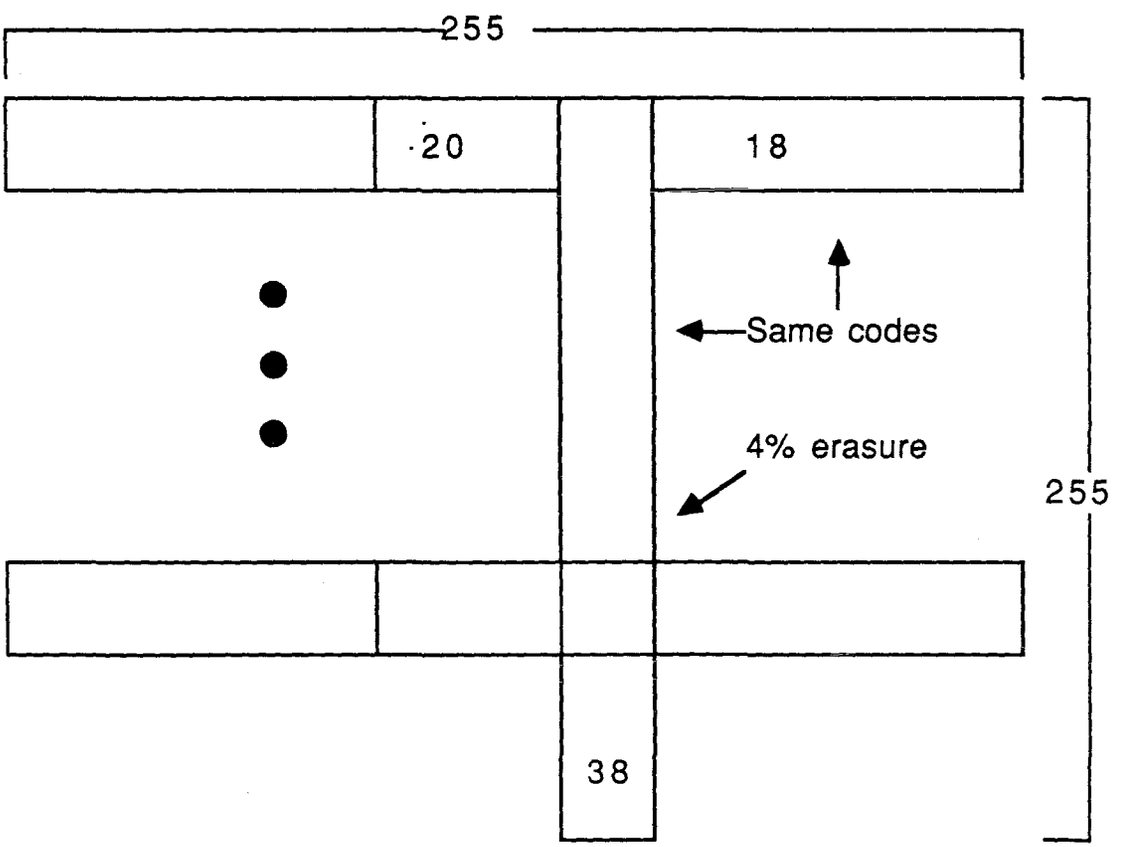


Figure 5

Symbol error rate=.01
Redundancy of normal 8 bit Reed Solomon
Code=.1020
at 10^-6 failure to decode
dB adder=.467dB

Redundancy of new code=.0527
dB adder=.235dB

Redundancy of 10 bit Reed-Solomon code=.0567
at 10^-6 failure to decode
dB adder =.235dB

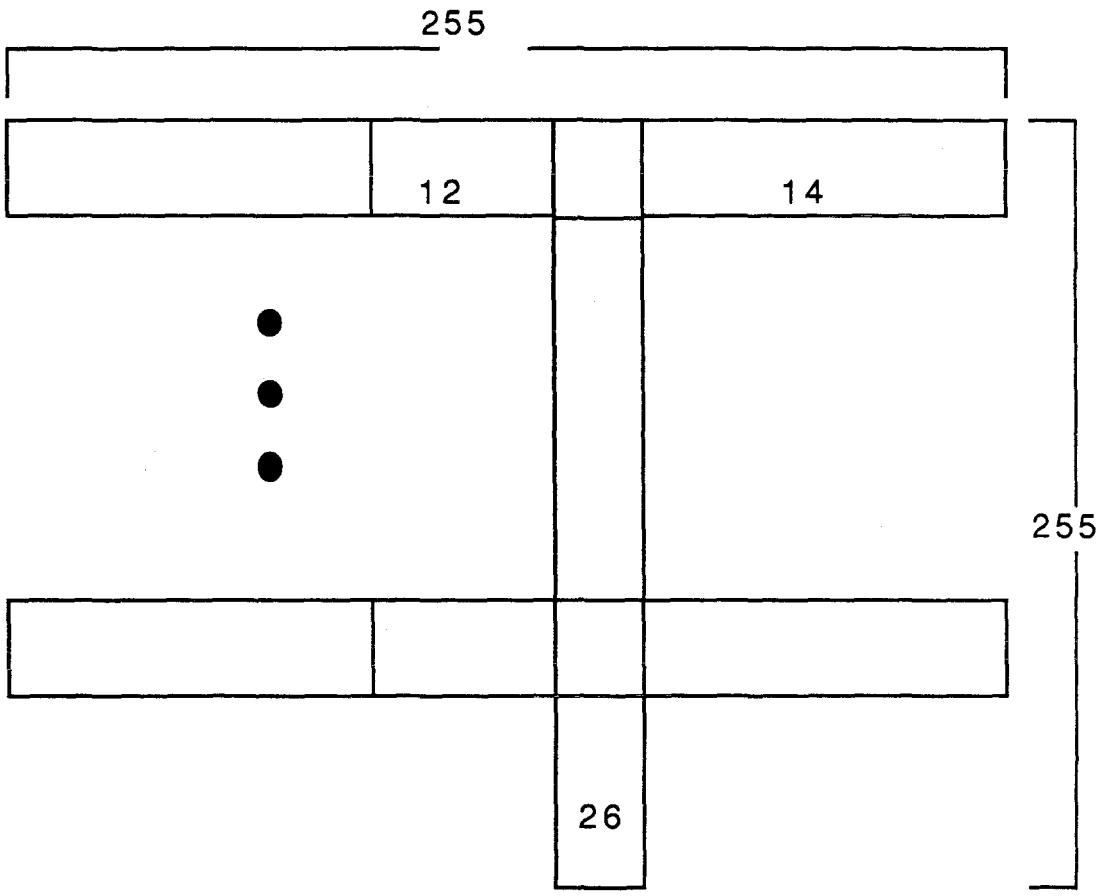
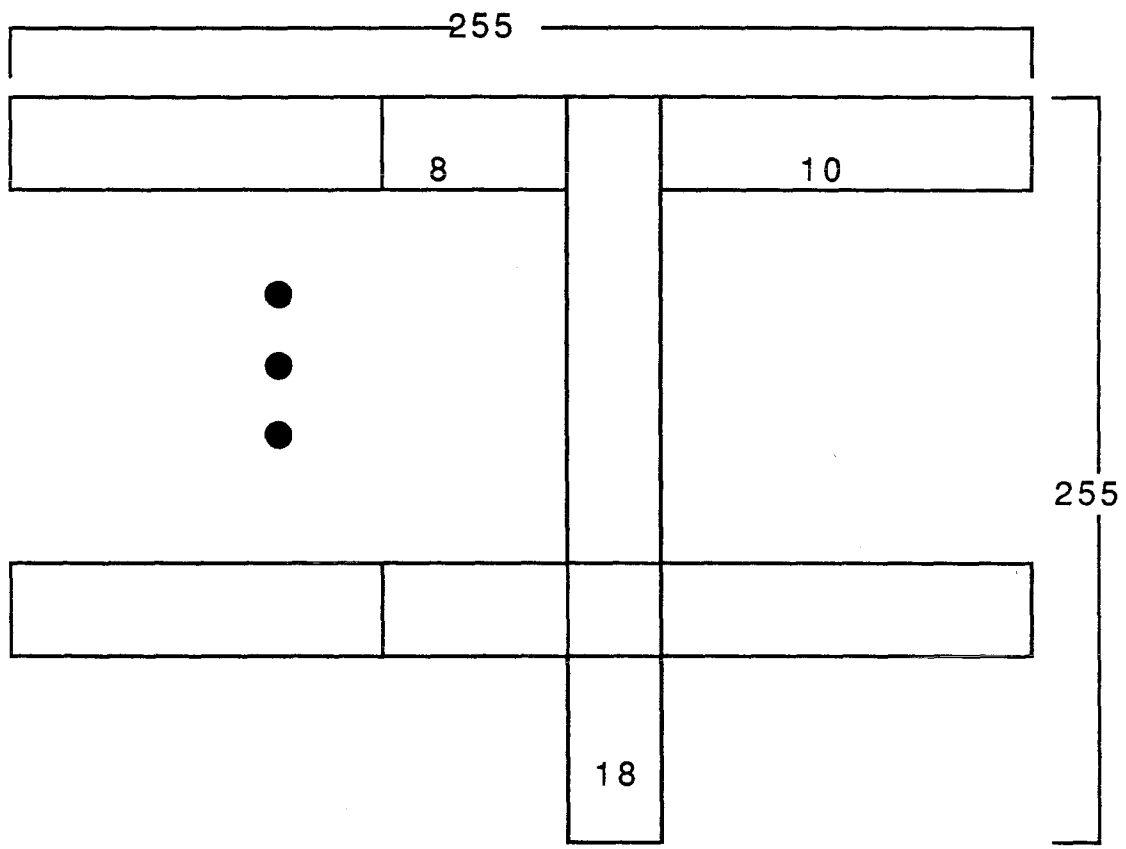


Figure 6

Symbol error rate=.005
Redundancy of normal 8 bit Reed Solomon
Code=.0706
at 10^-6 failure to decode
dB adder=.318dB

Redundancy of new code=.0341
dB adder=.151dB

Redundancy of 10 bit Reed-Solomon code=.0371
at 10^-6 failure to decode
dB adder =.164dB



CHAPTER IV

Hardware

This chapter presents a bit serial implementation of a long constraint length convolutional decoder at a level of detail just above final schematic drawings. The diagrams show a constraint length 15 (or less), rate 1/6, traceback depth 170 decoder, but the design is quite plastic. The machine can only decode a constraint length 15 code if it has leading and trailing ones in the generator polynomials. This restriction causes the branch lables to occurr in antipodal pairs and allows the simplified branch metric computer shown in Figure 7. This metric computer generates the two branch metrics (labelled q and r in Figure 7) for a pair of states whose labels differ only in the rightmost (least significant) bit. Such a pair is called a *butterfly*. The branch metric computer takes as inputs the six symbols that correspond to one information bit as well as the sum of their absolute values.

The symbol magnitues and signs arrive on separate sets of lines. Each of the six symbol signs is compared with a bit stored in the butterfly. If the two disagree then the corresponding symbol magnitude is added into the sum that is branch metric q. The six bits stored in each butterfly depend on the particular code and are reloaded every time it is changed through one long shift register. This shift register is the initialization chain in Figure 11.

The second output, r, is just the difference between q and the sum of all the symbol magnitudes. Here is where the antipodal property of the branch metrics simplifies the design. The bit serial subtractor that computes this

difference is shown in Figure 8. Note its similarity to the bit serial adder above it; the "D" latch becomes a borrow rather than a carry register. All the "D" latches in all of the drawings with one exception in Figure 9 have a single, common clock. They store data on the rising edge of this clock. The inputs to the branch metric computer and all other arithmetic elements of the butterfly are of such a form as to guarantee that carries (or borrows) will always clear. Thus, none of these "D" latches requires a set or reset.

The outputs of the branch metric computer go to the body of the butterfly, shown in Figure 10. Here they are added to the accumulated metric inputs of the butterfly in all possible ways. These inputs and the outputs in the upper left of the figure get connected into a de Bruijn Graph. Accomplishing this connection for 8,192 butterflies is the subject of the next chapter. For the moment, note only that the wires between many butterflies will be long. Thus, the inputs and outputs need "D" latches to allow a full clock period for a signal to propagate between butterflies.

The constraint length control mutiplexer in Figure 10 allows the decoding of codes with constraint length less than 15. We can (at least for a channel whose symbols are independent) left justify all the generator polynomials so that they all have a leading 1 without affecting the code's properties. For codes with $K < 15$ this will make the two states of a butterfly indistinguishable; hence, the mutiplexer. Thus, for $K < 15$ there are still restrictions on the generator polynomials but not on the code.

The only remaining elements in Figure 12 are the compare/select units that are identical and shown in detail in Figure 9. A compare/select unit compares two number, X and Y, that it receives in bit serial form and outputs the lesser of the two. The word clock marks the arrival of the most significant bits and, hence, the time when the selection of the smaller of the two can be

made. If one or more bits of data in the information stream are known, then the selection can be predetermined by using the force lines shown in Figure 9. These lines and their associated multiplexer are all that is necessary to implement the redecoding scheme of Chapter II. The two N bit shift registers store the two numbers coming in on the X and Y wires until their most significant bits arrive and the comparison can be made. The greater number is just thrown away.

The length of these shift registers, N, is a very important parameter of the design both because the decoding speed is inversely proportional to N+2 and because the area of the shift registerss is a significant fraction of the total chip area required for the butterfly. The dynamic range of the accumulated metrics is limited to the product of the reciprocal of the rate, the contraint length minus one, and the largest symbol magnitude. This product is 6*14*127=10,668 in this design. Renormalization consists of removing the leading one from all the accumulated metrics. Thus, if we could always locate the smallest metric and check whether its leading bit is a one, we would need N equal to one plus the greates integer of the base two logarithm of 10,668. Looking at all the 16,384 metrics in a constraint length 15 machine is, however, extremely difficult. Thus, it may pay to make N bigger than 15 if it means we do not have to look at all the metrics. We can get by with looking at only a single randomly chosen metric if we add one more bit to N. Looking at only a single metric means that the spread can occur on either side of the renormaliztion threshold. If we look at two metrics whose symbol sequences are antipodal, e.g., the all zero state and the state with a single leading one, and always compare the larger of the two with the threshold then the potential spread of the metrics about the threshold is only 1.5 instead of 2. For our design this reduction allows us to

get by with $N = 15$. N, however, is taken to be 16 for two reasons. First, the system is so large that bringing two particular metrics to a sensing point is more difficult than just looking at whatever metric happens to be closest. Second, with $N = 16$ we can use a traceback chain length of 16 as shown in Figure 11, which reduces the required traceback memory bandwidth.

A single line controls both traceback and renormalization as shown in Figure 9. A single width pulse on this line loads the traceback chain shift register, and a double width pulse also removes the leading bit from all the accumulated metrics. The communication pipeline registers in Figure 10, which have already been mentioned, make this sharing possible. When the metrics are in the shift registers these registers contain dummy bits which only affect the comparisons if their is a tie. Hence, alway setting one of them to zero does not affect the operation of the decoder.

The 16 bit long traceback chain contains the results of 16 comparisons made by 8 butterflies after it is loaded. These 16 bits are written into traceback memory during the next 16 clock cycles, leaving 2 clock cycles two read the memory in the overall 18 clock master cycle. Every single traceback chain writes to memory during the 16 write clocks. However, only a single bit from the whole traceback memory is read during one of the two read cycles. The physical memory arrangement for one board having 32 16-butterfly chips is shown in Figure 12. The multiplexer is continued at the board level, and the whole arrangement is controlled by the circuit of Figure 13. The large ROM in Figure 13 handles the mapping from logical to physical addresses; the small one creates three independent regions of memory from each of the 16,384 states. The handling of these regions is shown in Figure 14. The pattern repeats and each of the two active regions requires one of the two reads available in each master cycle.

Figure 7
Metric Computer

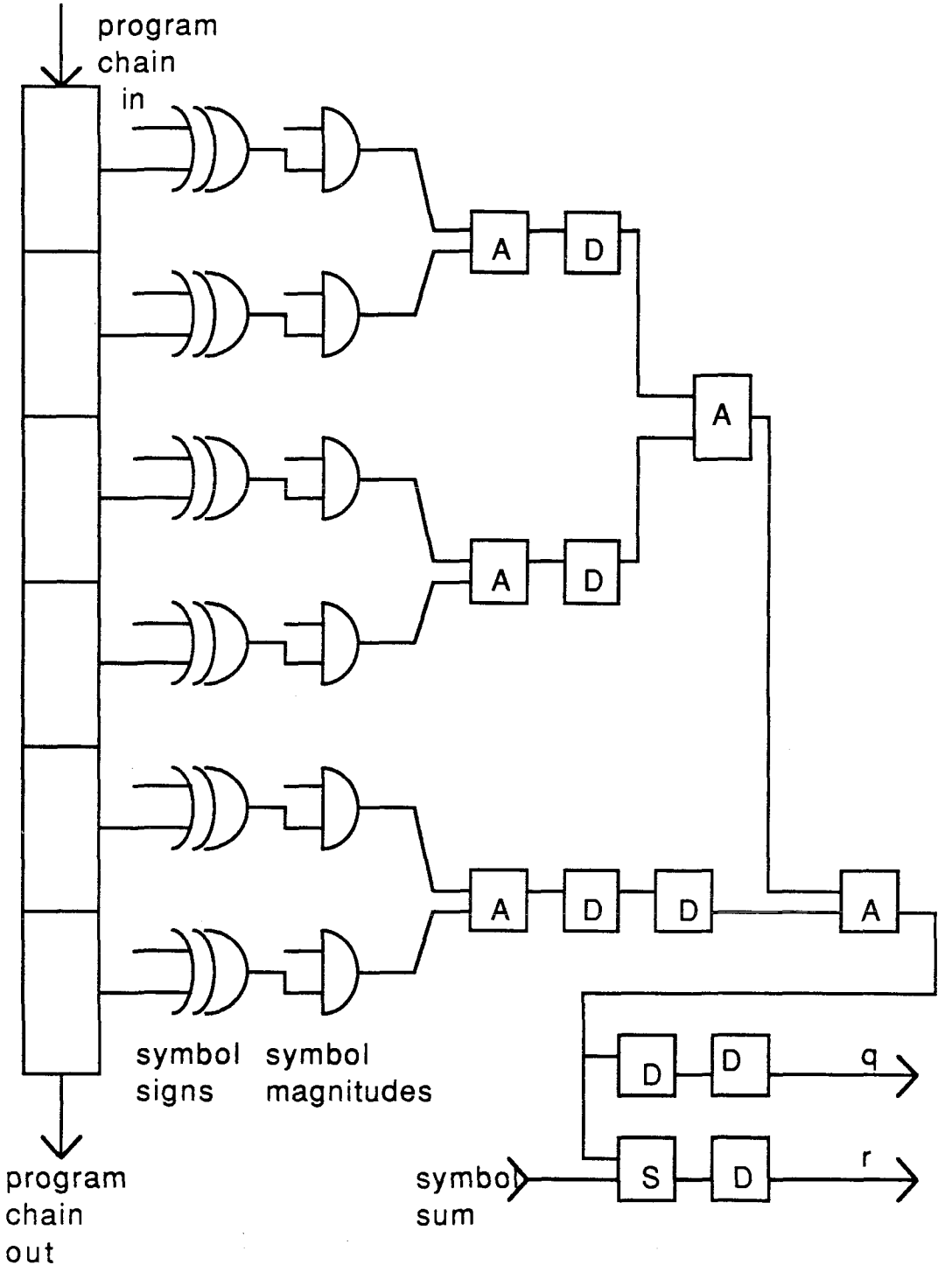


Figure 8

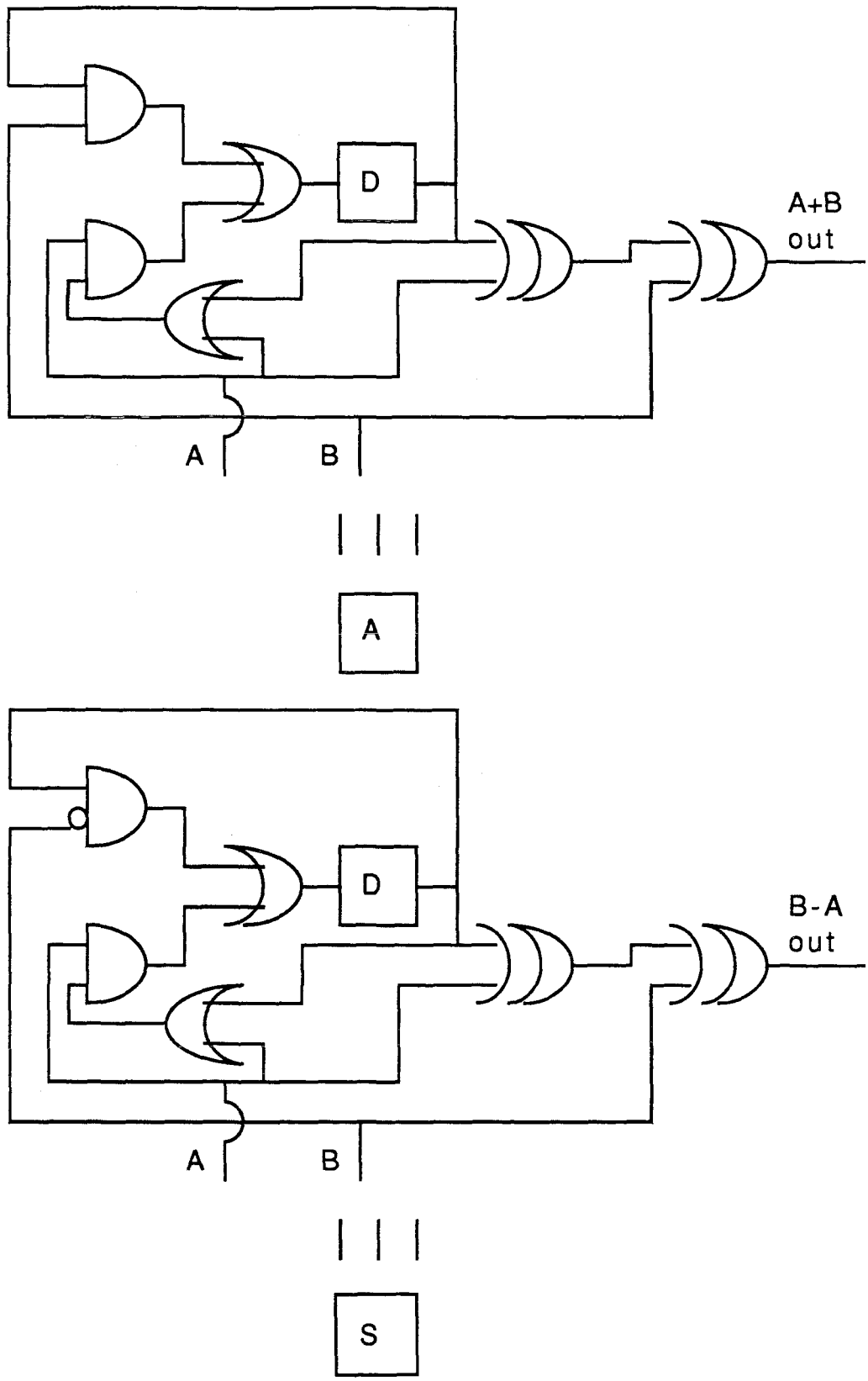


Figure 9

Compare/Select

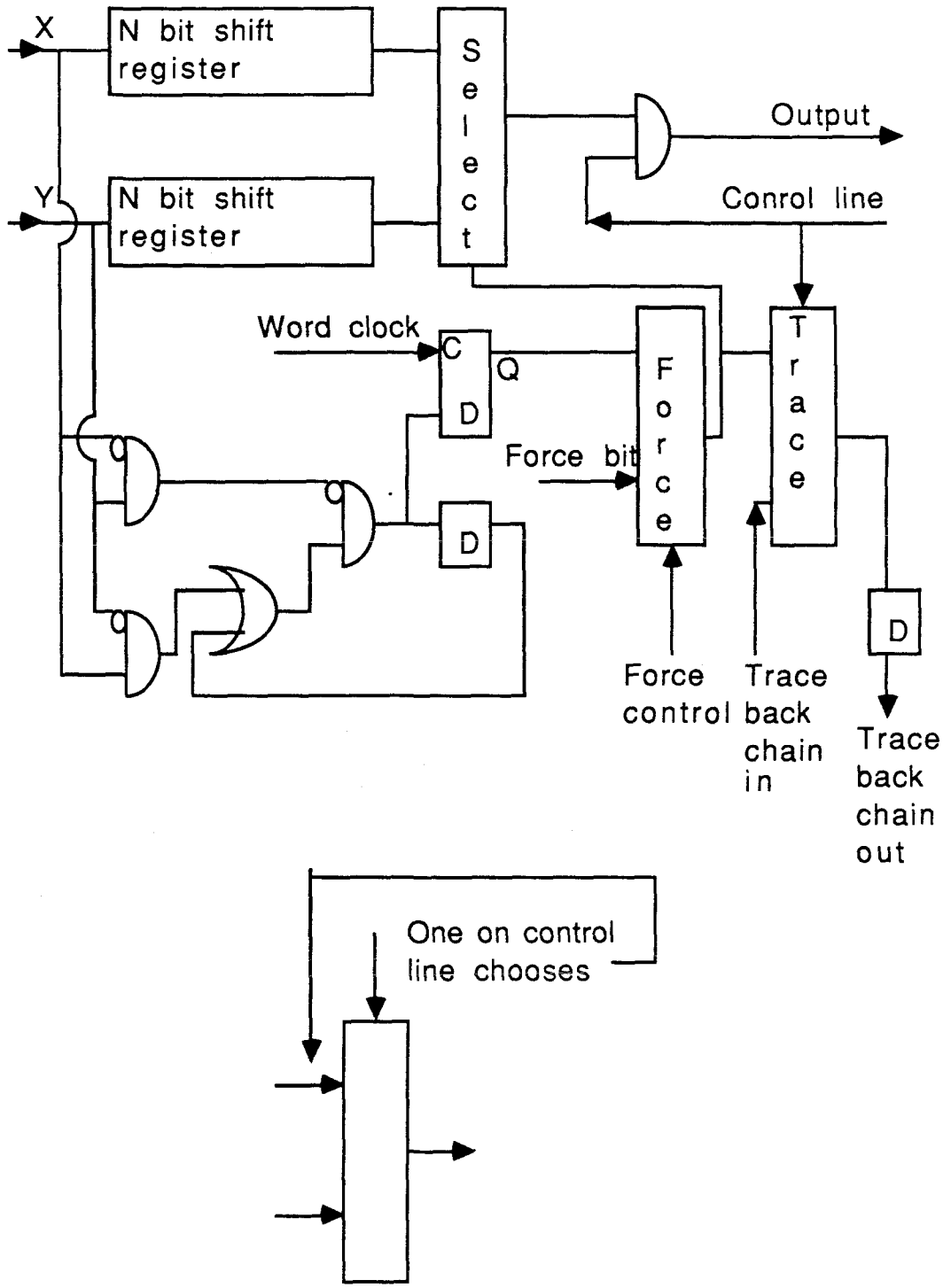


Figure 10
Butterfly

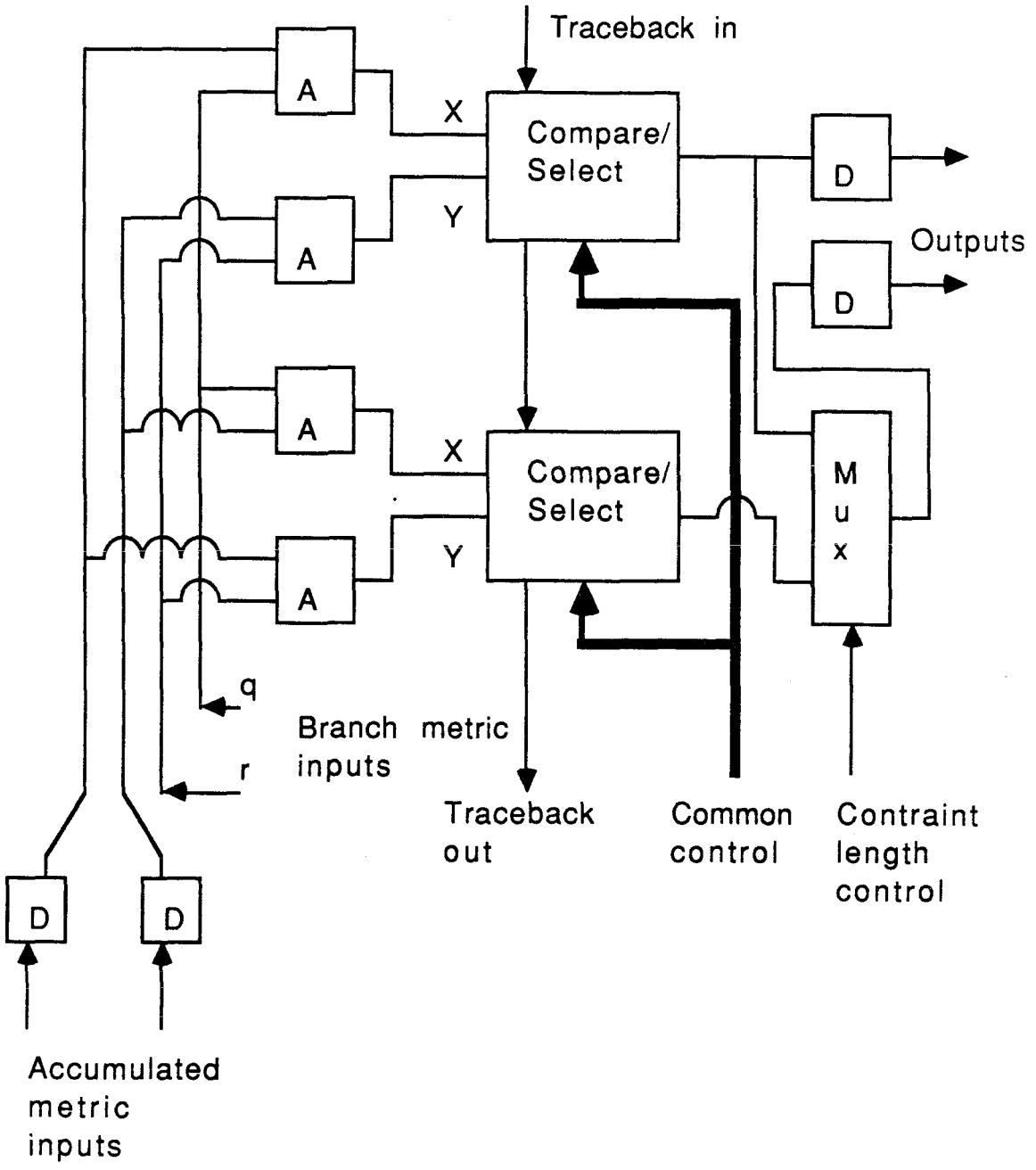


Figure 11
Chip

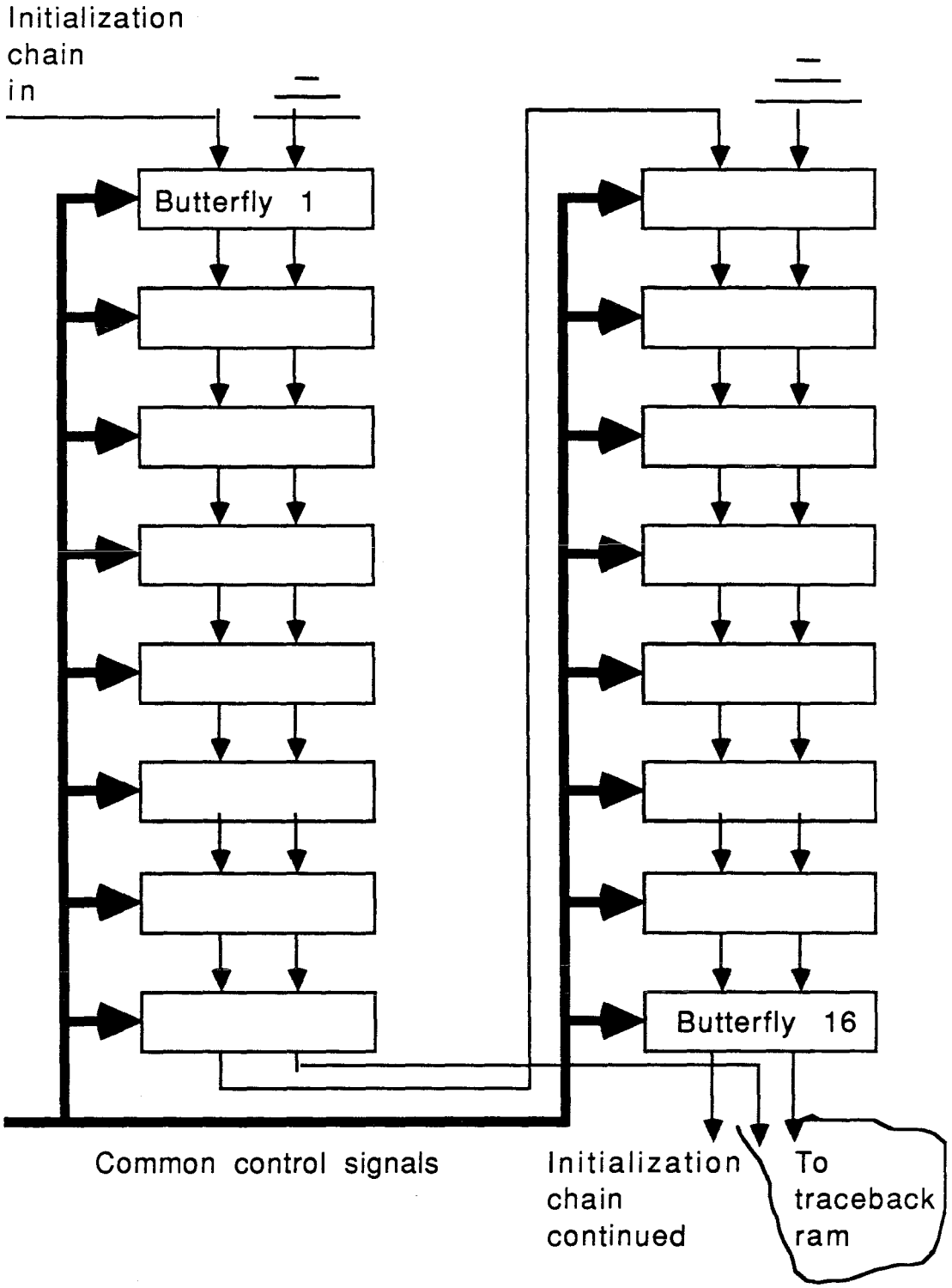


Figure 12
Board

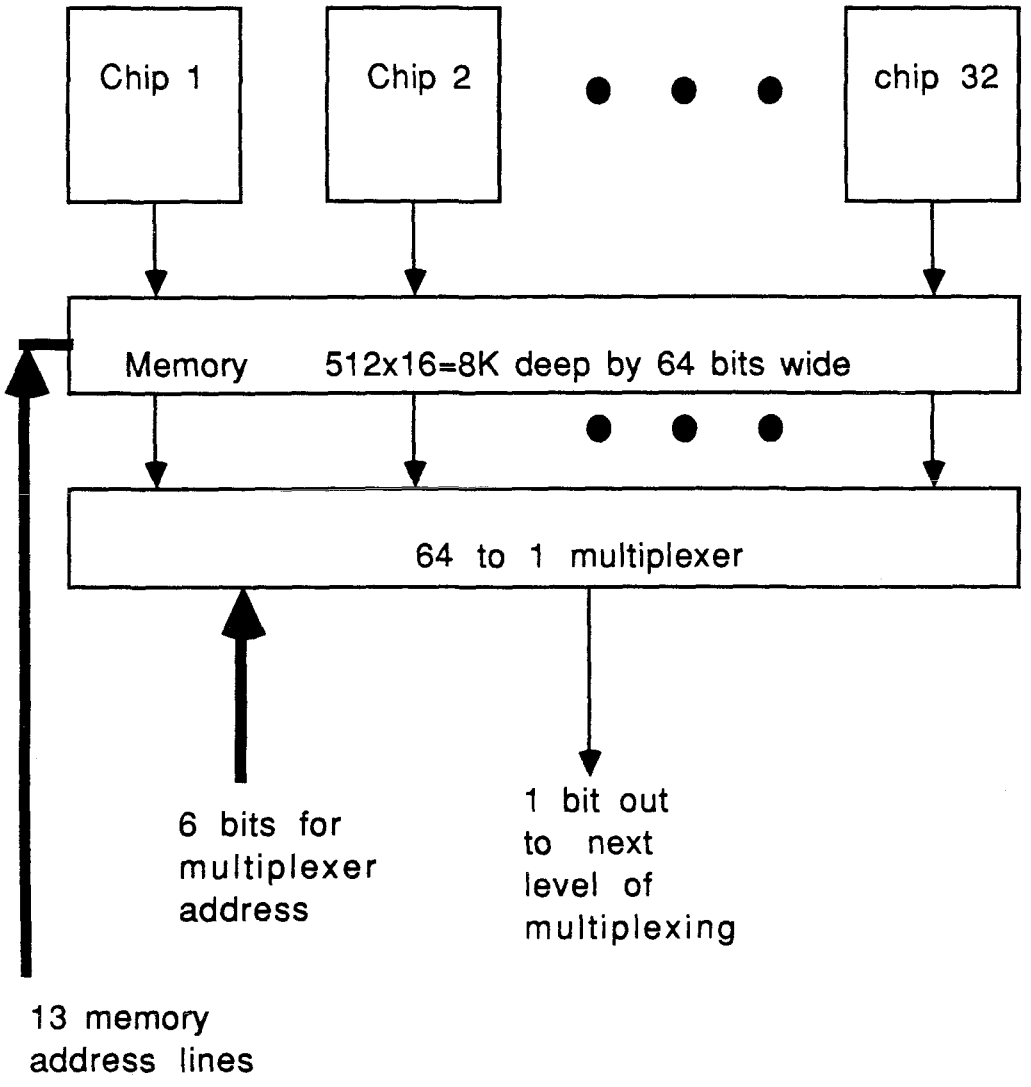
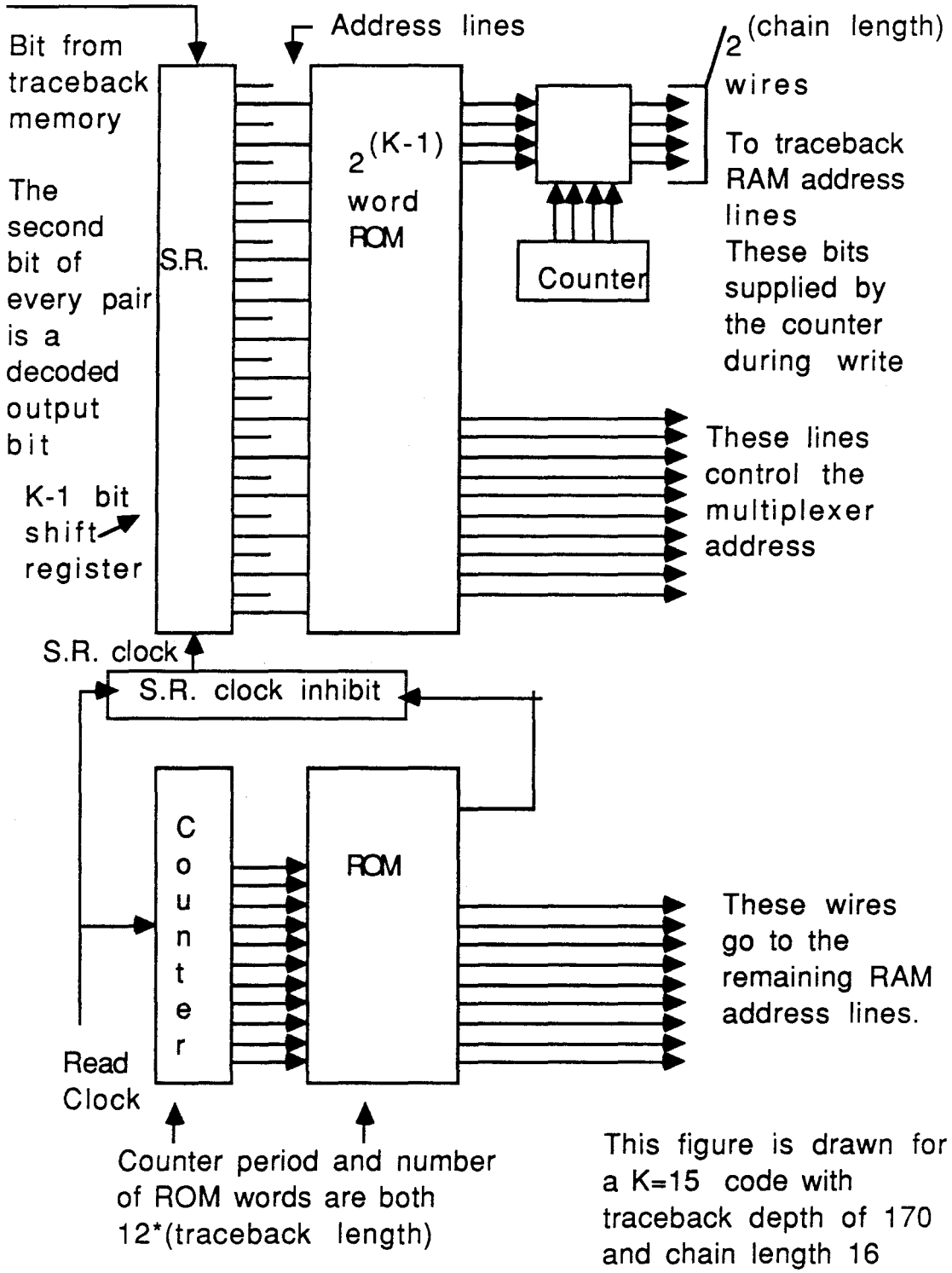


Figure 13



This figure is drawn for a K=15 code with traceback depth of 170 and chain length 16

Figure 14

Decoding left	Inactive	Traceback left
Traceback right	Decoding right	Inactive
Inactive	Traceback left	Decoding left
Decoding right	Inactive	Traceback right
Traceback left	Decoding left	Inactive
Inactive	Traceback right	Decoding right

CHAPTER VI

Graphs

This chapter consists of a paper which has been submitted to the Journal of the Association for Computing Machinery.

1. Introduction and Summary.

The nth order *deBruijn graph* B_n is the state diagram for an n-stage binary shift register. It is a directed graph with 2^n vertices, each labelled with an n-bit binary string, and 2^{n+1} edges, each labelled with an $(n+1)$-bit binary string. The vertex labels represent the contents of the shift register at a given point of time. The label on an edge connecting one vertex to another represents the contents of the shift register preceeded by the bit that is being input to the shift register, as it changes from one state to the next. In Figure 1 we see a representation of B_3.

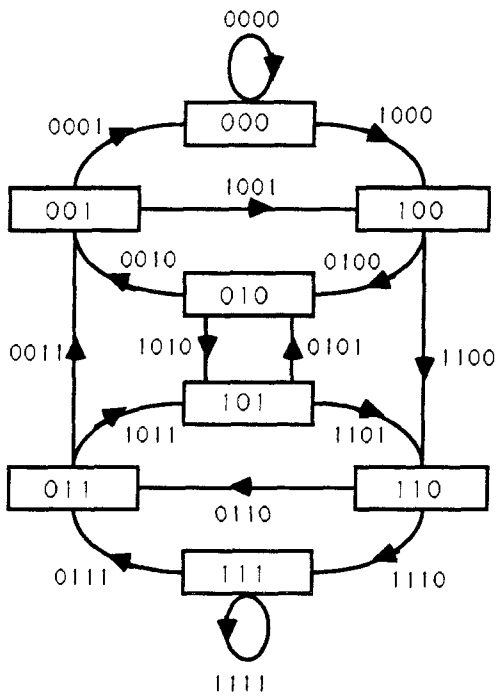


Figure 1. The deBruijn graph B_3.

We are interested in the deBruijn graph B_n because it gives the topology for a fully parallel Viterbi decoder for any rate $1/\nu$ convolutional code with constraint length $n+2$ ([3], Chapter 7). In such a decoder, a "butterfly" (a pair of add-compare-select units) must be located at each node of the graph, and all communication between butterflies takes place along the edges of the graph. In fact, Caltech's Jet Propulsion Laboratory is currently developing such a decoder, called the Big Viterbi Decoder, for a constraint length 15,

rate 1/4 convolutional code, for the *Galileo* mission. The BVD has 2^{13} butterflies, connected according to the topology of B_{13}. It is constructed from 256 identical semi-custom VLSI chips, each containing 32 butterflies. These chips are arranged on 16 identical printed-circuit boards, each containing 16 chips. Of the 2^{14} "wires" (butterfly interconnections) in the decoder, 56% are internal to the chips, another 17% are internal to the boards, and 27% are inter-board, or "backplane" connections. Furthermore, these chips and boards are universal, in the sense that any deBruijn graph B_n with $n \geq 5$ can be built from copies of these same chips, and any B_n with $n \geq 9$ can be built from copies of these same boards. In this paper, we will give the theoretical background which led to the design of these chips and boards. See [1] and [5] for further details. (We will return to the BVD at the end of the paper — see Example 6.4.)

2. Preliminaries About Strings.

In this section we introduce some notation and establish a few elementary facts about binary strings, which will be needed throughout the rest of the paper.

2.1. Definitions. A binary string is a sequence of 0s and 1s. The *length* of a binary string x, denoted by $|x|$, is the number of symbols in x. The *empty string* ϵ is the string with no symbols. Thus $|\epsilon| = 0$. The set of all strings of length n is denoted by $\{0,1\}^n$. If x and y are two strings, the *concatenation* of x and y, denoted by xy or $x*y$, is the string obtained by following the bits of x by the bits of y. If $x = a*b*c$, then a is called a *prefix*, b is a *substring*, and c is a *suffix* of x. If $b*c$ isn't empty, then a is called a *proper prefix* of x; if either a or c is nonempty, b is called a *proper substring* of x; and if $a*b$ isn't empty, c is called a *proper suffix* of x. If x is a nonempty binary string, then the symbol x^L (the *left part* of x) denotes the string obtained by removing the rightmost bit of x; similarly, x^R (the *right part* of x) denotes the string obtained by removing the leftmost bit of x. If S and T are sets of binary strings, we say that S *covers* T if every string in T has a substring in S. Similarly, we say that S *prefixes* T if every string in T has a prefix in S. We say that S is *irreducible* if no string in S covers any other. Finally, we define the *cost* of a set of strings S as $\text{cost}(S) = \sum_{s \in S} 2^{-|s|}$, where $|s|$ denotes the length of the string s. ❐

2.2. Examples. If $x = 1011$, then $|x| = 4$, $x^L = 101$ and $x^R = 011$. The set $S = \{10, 111\}$ covers $\{010, 100, 101, 110, 111\}$, and $\{1, 0000\}$ covers $\{0,1\}^n$ for all $n \geq 4$. Similarly, $\{1, 000\}$ prefixes $\{1, 00000\}$, and $\{0, 10, 110, 111\}$ prefixes $\{0,1\}^n$ for all $n \geq 3$. Also, $\{1, 000\}$ is irreducible, but $\{1, 001\}$ is not. The cost of the set $\{10, 111\}$ is 3/8, $\text{cost}(\{1, 000\}) = 5/8$, and $\text{cost}(\{0,1\}^n) = 1$, for all $n \geq 1$. ❐

2.3. Theorem. If S is an irreducible set of strings, then every string x covered by S can be factored uniquely in the form $x = \lambda s \rho$, where $s \in S$, λ

and ρ are (possibly empty) strings, and $(\lambda s)^L$ has no substring from S. We will call this factorization the *S-factorization* of x.

Proof: Since x is covered by S, x will have one or more substrings from S. Among these S-substrings, there will be a unique *leftmost* one, since no string in S covers any other. Call this unique leftmost S-substring s. Then plainly $x = \lambda s \rho$ is the desired unique factorization. ❐

2.4. Examples. As noted above, $S = \{1, 0000\}$ is irreducible and covers all strings of length 4. The S-factorization of 1010 is $\epsilon * 1 * 010$, the S-factorization of 0101 is $0 * 1 * 01$, and the S-factorization of 0000 is $\epsilon * 0000 * \epsilon$. ❐

2.5. Lemma. If S covers $\{0,1\}^n$, then every string x of length n or greater will have a unique S-factorization, and if $x = \lambda s \rho$ is this factorization, then $|\lambda s| \leq n$.

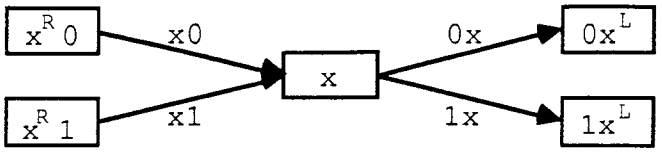
Proof: Every string of length n or greater will have a substring of length n. This substring will be covered by S, and hence so will x. Now let x be a string of length $\geq n$, and let $x = \lambda s \rho$ be its S-factorization, as described in Theorem 2.3. By definition of the S-factorization, $(\lambda s)^L$ is not covered by S. However, if $|\lambda s| > n$, then $|(\lambda s)^L| \geq n$, which would imply that $(\lambda s)^L$ is covered by S, a contradiction. ❐

3. DeBruijn Graphs and Subgraphs.

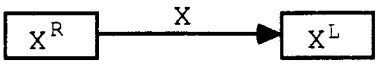
The deBruijn graph B_n, which is the state diagram for an n-stage shift register, can be described as follows. There are 2^n vertices, each labelled with an n-bit binary string x. There is a directed edge from the vertex with label x to exactly two other vertices, viz. those with labels $0x^L$ and $1x^L$. The edge from x to $0x^L$ is labelled $0x$ and the edge from x to $1x^L$ is labelled $1x$. Similarly, there are exactly two edges directed into x, from $x^R 0$ and $x^R 1$, which are labelled $x0$ and $x1$. This definition is summarized in Figure 2a. For example, in Figure 1, from the vertex 101 there are edges leading to $0(101)^L = 010$ and to $1(101)^L = 110$. Equivalently, we can define B_n by saying that it has 2^{n+1} edges, each labelled with an $(n+1)$-bit binary string, and that the edge labelled X connects the vertices with n-bit labels X^R and X^L. This alternative definition is summarized in Figure 2b. For example, in Figure 1, the edge labelled 1011 is a directed edge from $(1011)^R = 011$ to $(1011)^L = 101$.

We next need to define the notion of a *subgraph* of a deBruijn graph. In this definition, and later, the symbols $E(G)$ and $V(G)$ stand for the number of edges and vertices in the graph G. Thus for example, $E(B_n) = 2^{n+1}$ and $V(B_n) = 2^n$.

3.1. Definition. If H and G are graphs, H is called a *G-subgraph*, written $H \subseteq G$, if H has the same vertex set as G, and an edge set which is a subset



(a)



(b)

Figure 2. Two Equivalent definitions of the DeBruijn Graph B_n. In (a), we see the four vertices connected to the vertex labelled x (x is an n-bit string); in (b), we see the vertices at the left and right ends of the edge labelled X (X is an $(n+1)$-bit string).

of the edge set of G. The *density* of a G-subgraph H, denoted by $\mathrm{den}(H : G)$, is the fraction of the edges in H present in G, i.e., $\mathrm{den}(H : G) = E(H)/E(G)$. ❐

3.2. Examples. A B_n-subgraph of density 0 consists of 2^n isolated vertices, and a B_n-subgraph of density 1 is B_n itself. Figure 3 shows a B_3-subgraph of density 6/16, consisting of the eight vertices of B_3 and the six edges labelled $\{0010, 0011, 0100, 0101, 0101, 0111\}$. ❐

Our goal is to build a large deBruijn graph B_N by connecting together multiple copies of a smaller graph, called a "building block." If we think of the building blocks as VLSI chips, it is natural to want to minimize the number of edges needed to connect the building blocks together. This goal leads to the following definition.

3.3. Definition. A graph H is a *building block* for a graph G if there exists G-subgraph $\overline{H}$ which is the disjoint union of several copies of H. The *efficiency* of H as a building block for G, denoted by $\mathrm{eff}(H : G)$, is defined

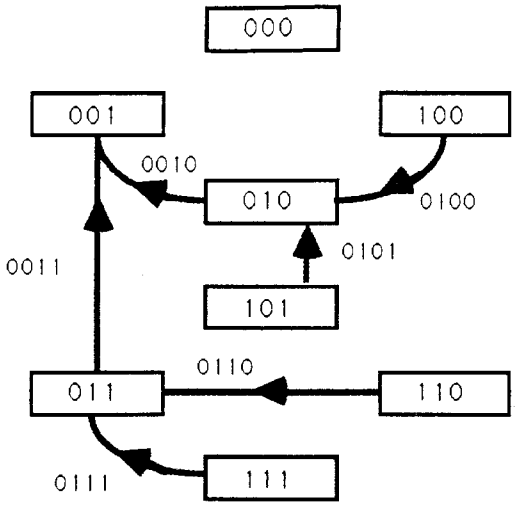


Figure 3. A B_3 subgraph with density 6/16. (In the notation of Section 4, this is the graph $B_3(\{1,000\})$. In the notation of Section 5, it is the graph $\overline{B}_3(\{1\})$. It is a universal deBruijn subgraph of efficiency 3/8.)

to be $\text{den}(\overline{H} : G)$. In other words, $\text{eff}(H : G)$ represents the fraction of the edges of G which are accounted for by the edges in the building blocks. ❐

3.4. Theorem. If H is a building block for G, then

$$\text{eff}(H : G) = \frac{V(G)E(H)}{V(H)E(G)}.$$

Proof: The G-subgraph $\overline{H}$ has the same number of vertices as G and is the disjoint union of several disjoint copies of H. Since G has $V(G)$ vertices, and H has $V(H)$ vertices, this means that $\overline{H}$ is the union of exactly $V(G)/V(H)$ copies of H. Since each of of these copies of H has $E(H)$ edges, $E(\overline{H}) = E(H)(V(G)/V(H))$. Thus $\text{eff}(H : G) = \text{den}(\overline{H} : G) = E(\overline{H})/E(G) = (E(H)(V(G)/V(H))/E(G) = (V(G)E(H)/V(H)E(G))$. ❐

3.5. Examples. Any B_n-subgraph H is a building block of efficiency $\text{den}(H : B_n)$ for B_n. A B_n-subgraph of density 0 is a building block of efficiency 0 for any B_N with $N \geq n$. A B_n-subgraph of density 1 (i.e., B_n itself) cannot be a building block for any larger deBruijn graph, since the disjoint union of 2^{N-n} copies of B_n is a disconnected graph with the same number of edges as B_N, which is connected. In Figure 4 we see two copies of the graph H in Figure 3 relabelled and wired together with 20 new edges to

form a graph isomorphic to B_4. Since B_4 has 32 edges, and the two copies of H together have 12 edges, it follows that H is a building block for B_4 of efficiency $12/32 = 37.5\%$. In fact, the graph in Figure 3 is a building block of efficiency $3/8$ for *any* B_N with $N \geq 3$, as we will see in Example 4.5, below. The building block of Figure 3 is an example of what we call a *universal deBruijn building block.* ❐

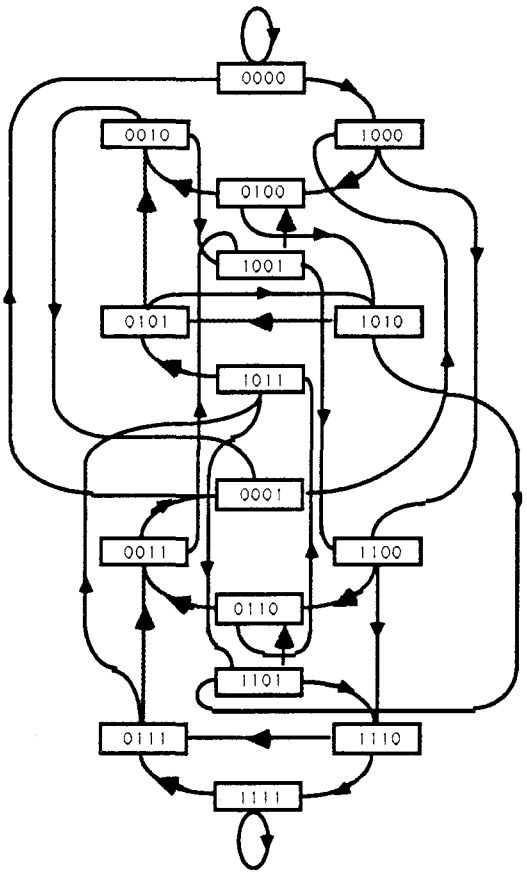


Figure 4. Two copies of the B_3 subgraph H from Figure 3, relabelled and wired together to make B_4 (edge labels omitted).

3.6. Definition. A *universal deBruijn building block of order* n is a B_n-subgraph which is a building block for any deBruijn graph B_N with $N \geq n$. ❐

The following theorem shows that it is easy to compute the efficiency of any univeral deBruijn building block.

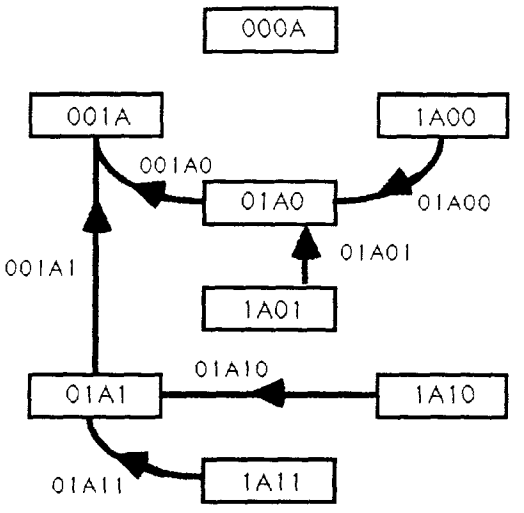


Figure 5. The graph $B_3(S, A)$, obtained by inserting the symbol A immediately after the first occurrence of a substring from $S = \{1, 000\}$ in the corresponding label in $B_3(S)$ (cf. Figure 3).

3.7. Theorem. Let H be a a universal deBruijn building block of order n. Then for all $N \geq n$, $\text{eff}(H : B_N) = \text{den}(H : B_n)$. We will call this common value the *efficiency* of H as a deBruijn building block.

Proof: By Theorem 3.4, the efficiency of H as a building block for B_N is

$$\begin{aligned}\text{eff}(H : B_N) &= \frac{V(B_N)}{E(B_N)} \cdot \frac{E(H)}{V(H)} \\ &= \frac{2^N}{2^{N+1}} \cdot \frac{2^{n+1}\,\text{den}(H : B_n)}{2^n} \\ &= \text{den}(H : B_n). \quad \square\end{aligned}$$

In the next section, we will describe a general construction for universal deBruijn subgraphs (Theorem 4.3), and in Section 5 (Theorem 5.9), we will see that there exist universal deBruijn subgraphs whose efficiency approaches 1 as n approaches infinity.

4. A General Construction for Universal DeBruijn Subgraphs.

In this section, we will present our main theorem (Theorem 4.3), which gives a general construction for universal deBruijn building blocks. The key to this construction is a way of constructing a B_n subgraph from a set of strings of length $\leq n$.

4.1. Definition. If S is a set of strings, $B_n(S)$ is defined to be the B_n-subgraph obtained by deleting from B_n all edges whose labels have a prefix in S. ❐

4.2. Lemma. If S is irreducible, $E(B_n(S)) = 2^{n+1}(1 - \text{cost}(S))$; equivalently, $\text{den}(B_n(S) : B_n) = 1 - \text{cost}(S)$.

Proof: The 2^{n+1} edges in B_n are labelled with the $(n+1)$-bit strings, and there are 2^{n+1} of them. For each $s \in S$, there are exactly $2^{n+1-|s|}$ $(n+1)$-bit strings with s as a prefix. Since no $(n+1)$-bit string can have two prefixes in S (no string in S is a prefix of any other, since S is irreducible), the subgraph $B_n(S)$ will have exactly $2^{n+1} - \sum_{s \in S} 2^{n+1-|s|} = 2^{n+1}(1 - \text{cost}(S))$ edges. ❐

The main theorem in this paper is the following.

4.3. Theorem. If S is irreducible and covers $\{0,1\}^n$, then $B_n(S)$ is a universal deBruijn building block of order n with efficiency $1 - \text{cost}(S)$.

We postpone the proof of Theorem 4.3 until we have given several examples, and stated and proved a stronger but more technical result (Theorem 4.6).

4.4. Example. The set $S = \{0,1\}$, is irreducible, has cost 1, and covers $\{0,1\}^n$ for any $n \geq 1$. The corresponding subgraph $B_n(S)$ is a B_n-subgraph of density zero, and is plainly a building block of efficiency zero for any deBruijn graph with $N \geq n$. ❐

4.5. Example. The set $S = \{1,000\}$ is irreducible, has cost $5/8$, and covers $\{0,1\}^3$. In this case, $B_3(S)$ is identical to the B_3-subgraph in Figure 3. Thus Theorem 4.3 implies that the graph $B_3(\{1,000\})$ is a universal deBruijn building block with efficiency $3/8$, as asserted in Example 3.5. ❐

The next theorem concerns a family of relabelled copies of the graph $B_n(S)$. If A is any binary string, we construct the graph $B_n(S,A)$ from $B_n(S)$ by inserting the string A into each vertex or edge label just after the first (leftmost) occurrence of a substring from S. In Figure 5, we illustrate this construction for the graph $B_3(\{1,000\})$.

4.6. Theorem. For all $N \geq n$, we have

$$\bigcup_{|A|=N-n} B_n(S,A) = B_N(S).$$

4.7. Example. In Figure 6 we have illlustrated Theorem 4.6, by showing the two graphs $B_3(S,0)$ and $B_3(S,1)$, for $S = \{1,000\}$. When taken together, these two graphs form the graph $B_4(S)$, which is a subgraph of B_4. Thus by adding the 20 edges whose labels have a prefix from S (16 with prefix 1 and 4 with prefix 000), we will obtain B_4, and indeed this is how we arrived at Figure 4. ❐

Proof of Theorem 4.6: Let us call the the graph $\bigcup_{|A|=N-n} B_n(S,A)$ appearing in the statement of the theorem the *union graph.* To prove Theorem 4.6, we need to show that the union graph is a B_N-subgraph, and that its edges are exactly those whose labels have no prefix from S. To do this, we will prove the following four assertions (always A denotes a string of length $N-n$):

(1) Every N-bit string occurs as a vertex label in some $B_n(S,A)$;

(2) Every edge in $B_n(S,A)$ is an edge of the deBruijn graph B_N;

(3) No edge label in $B_n(S,A)$ has an S-prefix;

(4) Every $(N+1)$-bit string without an S-prefix appears as an edge label on some $B_n(S,A)$.

Taken together, (1) and (2) show that the union graph is a B_N-subgraph; and (3) and (4) show that the edge labels in the union graph are the $(N+1)$-bit strings without an S-prefix.

Proof of (1): Let X be an arbitrary N-bit string, and let $X = \lambda s \rho$ be its S-factorization. By Lemma 2.5, $|\lambda s| \leq n$, so that $|\rho| \geq N-n$. If we denote the leftmost $N-n$ bits of ρ by A, then we have $\rho = A\rho'$, and hence $X = \lambda s A \rho'$. It follows that X appears as the label of the vertex $\lambda s \rho'$ in $B_n(S,A)$. For example, if $n=3$, $S=\{1,000\}$ and $N=8$, the 8-bit string $X = 01011110$ has S-factorization $0*1*011110$. The first five bits of 011110 are 01111 and so 01011110 appears as the label on vertex 010 in $B_3(S, 01111)$.

Proof of (2): If an $(n+1)$-bit edge label x in $B_n(S)$ has S-factorization $x = \lambda s \rho$, then neither λ nor ρ is empty: λ isn't empty, because no edge label in $B_n(S)$ has an S-prefix; ρ isn't empty, since $|x| = n+1$, and by Theorem 2.5, any S factorization has $|\lambda s| \leq n$. Thus in $B_n(S)$, an edge with S-factorization $\lambda s \rho$ connects the vertices with labels $(\lambda s \rho)^R = \lambda^R s \rho$ and $(\lambda s \rho)^L = \lambda s \rho^L$. Furthermore, this representation of the vertex labels must in fact be the S-factorization of them, since an earlier occurrence of a substring from S in either $\lambda^R s \rho$ or $\lambda s \rho^L$ would imply an earlier occurence of an S-string in $\lambda s \rho$. This means that in the graph $B_n(S,A)$, the edge $\lambda s A \rho$ connects the vertices $\lambda^R s A \rho = (\lambda s A \rho)^R$ and $\lambda s \rho^L = (\lambda s A \rho)^R$. In other words, an edge with label X in $B_n(S,A)$ connects X^R to X^L, and by the definition in Figure 2b, this is an edge in the deBruijn graph B_n.

For example, in Figure 5, we see that the edge with label $001A0$ connects $001A0^R = 01A0$ to $001A0^L = 001A$, and this is an edge in the deBruijn graph $B_{|A|+3}$, for any string A.

Proof of (3): Let X be an edge label in the graph $B_n(S, A)$. Then by definition, X has the form $X = \lambda sA\rho$, where $\lambda s\rho$ is the S-factorization of an edge label in $B_n(S)$ with λ nonempty. If X had an S prefix, say s', then either s' would be a prefix of $\lambda s\rho$, or s would be a proper prefix of s'. But both of these alternatives are impossible: s' cannot be a prefix of $\lambda s\rho$, since $\lambda s\rho$, being an edge of $B_n(S)$, has no S-prefix; and s cannot be a proper substring of s', since no string in S is a proper substring of any other. Thus every edge in $B_n(S, A)$ is an edge in $B_N(S)$, as asserted.

Proof of (4): To prove that every $(N + 1)$-bit string with no S-prefix occurs as an edge label in $B_n(S, A)$ for some A, let X be such a string, and let $X = \lambda s\rho$ be its S-factorization, in which necessarily λ is nonempty. If A denotes the leftmost $N-n$ bits of ρ, then as above, we have $X = \lambda sA\rho'$. The string $\lambda s\rho'$ cannot have a prefix in S, for if s' were such a prefix, then either s' would be a prefix of X, or else s would be a proper substring of s' (since λ is nonempty), and both of these alternatives ae impossible. Thus $\lambda s\rho'$ is the label on an edge of $B_n(S)$, and so $X = \lambda sA\rho'$ appears as the label corresponding to that edge in the graph $B_n(S, A)$. For example, let $S = \{1, 000\}$, $n = 3$, and $N = 8$, and consider the 9-bit string $X = 001011100$, which has no S-prefix. The S-factorization of X is $X = 00 * 1 * 011100$. The first 5 bits of 011100 are 01110, and so X appears as the label on the edge 001 in the graph $B_3(S, 01110)$.

This completes the proof of Theorem 4.6. ❐

We conclude this section with the proof of Theorem 4.3.

Proof of Theorem 4.3: Theorem 4.6 explicitly shows that the union of 2^{N-n} copies of $B_n(S)$ forms a subgraph (namely, $B_N(S)$) of the big deBruijn graph B_N, and so B_N can be constructed simply by adding the edges missing from B_N to this union. Thus B_n is a universal deBruijn building block. According to Theorem 3.7, the efficiency of a universal deBruijn building block is the same as its density, and by Lemma 4.2, the density of $B_n(S)$ is $1 - \text{cost}(S)$. ❐

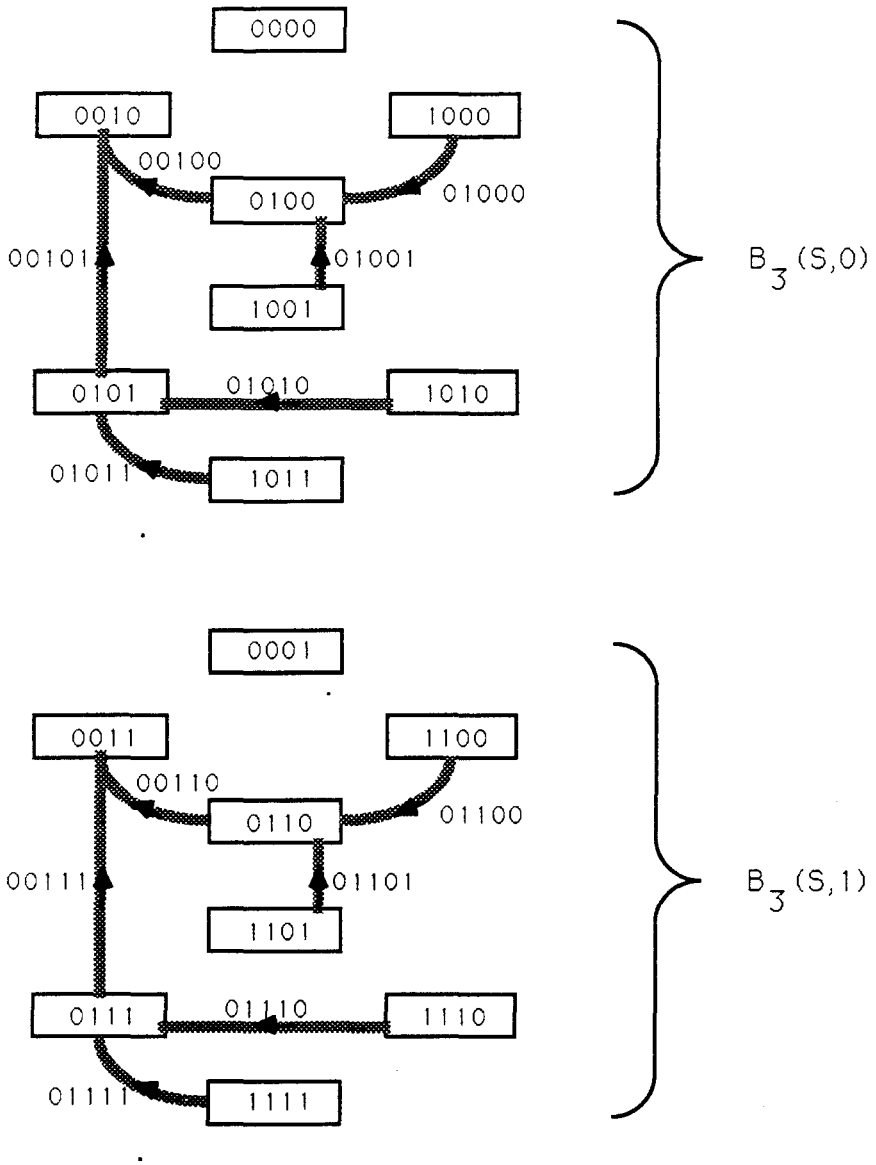


Figure 6. Illustrating Theorem 4.6:
$B_3(S,0) \cup B_3(S,1) = B_4(S)$ (compare with Figure 4).

5. Construction of Low-Cost Covers for $\{0,1\}^n$.

In Theorem 4.3, we showed how to construct universal deBruijn building blocks from covering sets for $\{0,1\}^n$ of small cost, but we were able to cite

only a few examples. In this section, we will give several general constructions for low-cost covers for $\{0,1\}^n$, thereby automatically producing (via Theorem 4.3) efficient universal deBruijn building blocks.

To produce a cover for $\{0,1\}^n$, we begin with an arbitrary irreducible set S of strings of length $\leq n$, which we call a *precover* for $\{0,1\}^n$. In general, S will fail to cover a certain subset of $\{0,1\}^n$, which we call $\text{Omit}_n(S)$. We denote the number of strings in $\text{Omit}_n(S)$ by $\text{omit}_n(S)$. Plainly, if we adjoin $\text{Omit}_n(S)$ to S, the resulting set, which we denote by $C_n(S)$, will still be irreducible, will cover $\{0,1\}^n$, and its cost will be $\text{cost}(S) + 2^{-n}\text{omit}_n(S)$. We summarize this simple but useful construction in the following theorem.

5.1. Theorem. For any irreducible set S of strings of length $\leq n$, the set $C_n(S)$ is an irreducible cover of $\{0,1\}^n$ of cost $(\text{cost}(S)+2^{-n}\text{omit}_n(S))$. ❐

5.2. Example. If $S = \{1\}$, then $\text{Omit}_n(S) = \{00\cdots0\}$, and $\text{omit}_n(S) = 1$ for all $n \geq 1$. Thus $C_n(S) = \{1, 00\cdots0\}$ is a cover for $\{0,1\}^n$ of cost $1/2 + 2^{-n}$, for all $n \geq 1$. ❐

5.3. Example. If $S = \{10\}$, then for $n \geq 2$, $\text{Omit}_n(S)$ consists of the $n+1$ strings of the form $0^k * 1^{n-k}$ for $0 \leq k \leq n$. Thus $\text{omit}_n(S) = n + 1$, and $C_n(S)$ is a cover for $\{0,1\}^n$ of cost $1/4 + (n+1)/2^n$, for all $n \geq 2$. ❐

5.4. Example. If $S = \{100, 1101\}$, then for $n \geq 4$, it can be shown that $\text{omit}_n(S) = 1 + \binom{n}{1} + \binom{n}{2}$, and thus $C_n(S)$ is a cover for $\{0,1\}^n$ of cost $3/16 + (1 + \binom{n}{1} + \binom{n}{2})/2^n$, for all $n \geq 4$. ❐

If we combine Theorems 4.3 and 5.1, we find that if S is an irreducible set of strings of length $\leq n$, then $B_n(C_n(S))$ is a universal deBruijn building block of order n and efficiency $1 - \text{cost}(S) - 2^{-n}\text{omit}_n(S)$. For simplicity, we denote $B_n(C_n(S))$ by $\overline{B}_n(S)$, and display this fact as a Theorem.

5.5. Theorem. If S is an irreducible set of strings of length $\leq n$, then the graph $\overline{B}_n(S)$ is a universal deBruijn building block of order n and efficiency $1 - \text{cost}(S) - 2^{-n}\text{omit}_n(S)$. ❐

The following theorem is a partial generalization of examples 5.2 and 5.3.

5.6. Theorem. Fix $m \geq 1$. If $S_m = \{10^{m-1}\}$, then as $n \to \infty$, $\text{omit}_n(S_m) = O(\alpha_m^n)$, where α_m is the largest positive root of the equation $z^m - 2z^{m-1} + 1 = 0$, which is strictly less than 2. Thus for all $n \geq m$, $C_n(S)$ is a cover for $\{0,1\}^n$ of cost $2^{-m} + O(\alpha_m/2)^n$, which approaches 2^{-m} as $n \to \infty$.

Proof: According to [2], if m is fixed, the generating function $f_m(z) = \sum_{n\geq 0} \text{omit}_n(S_m)z^n$ is given in closed form by $f_m(z) = 1/(1 - 2z + z^m)$. It follows then from the general theory of rational generating functions [4, Theorem 4.1.1], that $\text{omit}_n = O(\beta^n)$, where β is the reciprocal of the smallest positive root of the equation $1 - 2z + z^m = 0$, which is also the largest positive

root of the "reciprocal" polynomial $P_m(z) = z^m - 2z^{m-1} + 1$. The largest root of $P_m(z)$ is strictly less than 2, since $P_m(1) = 0$, $P_m(2) = 1$ and $P'_m(z) > 0$ for $z > 2$. ❐

5.7. Corollary. If c_n denotes the minimum cost for a cover for $\{0,1\}^n$, then $\lim_{n\to\infty} c_n = 0$.

Proof: Theorem 5.6 implies that for any $m \geq 1$, $\lim_{n\to\infty} c_n \leq 2^{-m}$. ❐

5.8. Remarks. We conjecture, but cannot prove, that $c_n = \Theta(1/n)$. However, McEliece and Swanson, in a forthcoming paper, will show that $c_n = \Omega(1/n)$ and $c_n = O(\log n/n)$. (The latter result is based on a more careful analysis of the type given in Theorem 5.6.) ❐

In view of the connection between covers for $\{0,1\}^n$ and universal deBruijn building blocks, Corollary 5.7 implies the following.

5.9. Theorem. There exist universal deBruijn building blocks whose efficiency is arbitrarily close to 1.

Although Theorem 5.6 gives an infinite family of reasonably cheap covers for $\{0,1\}^n$, it does not produce the cheapest possible covers for all values of n. Indeed, below is a table giving the cheapest covers of $\{0,1\}^n$, for $1 \leq n \leq 10$, that we know, and therefore also the most efficient universal deBruijn building blocks we know, for orders up to 10. In every case, we give only the "precover" S, it being understood that the actual cover is the larger set $C_n(S)$. We notice that for $n \leq 7$, the "$10\cdots0$" construction of Theorem 5.6 gives the best cover we know of, while for $8 \leq n \leq 10$ (and presumably for all larger values of n, too) the best cover is considerably more complicated. For $1 \leq n \leq 5$, we believe that the values in the table above are the best possible. For larger values of n, however, improvements may be possible. For $n \geq 8$, the covers described in the table are based on the general "$\{100, 1101\}$"-cover described in Example 5.4. For example, for $n = 8$, $\text{omit}(\{100, 1101\}) = 37$, so that $C_8(\{100, 1101\})$ is a cover for $\{0,1\}^8$ of cost $1/8 + 1/16 + 37/256 = 85/256$. However, by trial and error, we find that of the 37 strings in $\text{Omit}(\{100, 1101\})$, all but six are covered by $\{010101, 010111, 011111, 0000001, 0000101, 0000111\}$. Thus if we use $\{100, 1101\} \cup \{010101, 010111, 011111, 0000001, 0000101, 0000111\}$ as a precover, Theorem 5.1 guarantees a cover of cost $1/8 + 1/16 + 1/64 + 1/64 + 1/64 + 1/128 + 1/128 + 1/128 + 6/256 = 72/256$, as shown in the table. (Using $\{10\}$ as a precover for $n = 8$ results in a cover of cost $73/256$.)

n	$cost \cdot 2^n$	*efficiency*	S (*precover*)
1	2	0.000	{1}
2	3	0.250	{1}
3	5	0.375	{1}
4	9	0.438	{1} or {10}
5	14	0.563	{10}
6	23	0.641	{10}
7	40	0.688	{10}
8	72	0.719	{100, 1101, 010101, 010111, 011111, 0000001, 0000101, 0000111}
9	127	0.752	{100, 1101, 0000001, 0101011, 0101111, 0111111, 00001011, 00001111, 01010101}
10	229	0.776	{100, 1101, 01010101, 01010111, 01111111, 000000001, 000000101, 000000111, 000010101, 000010111, 000011111}

6. Hierarchical Building Blocks.

We have seen that the universal deBruijn building blocks described in Theorem 4.3 can be used to build deBruijn graphs of any size. Surprisingly, however, they can also be used as building blocks for larger universal deBruijn building blocks! This is useful in practice, when many chips must be put on several boards, and the boards are then wired together to make the deBruijn graph. The main theorem here is the following.

6.1. Theorem. Suppose $k \leq n$. If S is irreducible and covers $\{0,1\}^k$, and T is irreducible and covers $\{0,1\}^n$, and if S prefixes T, then $B_k(S)$ is a building block for $B_n(T)$. Furthermore,

$$\text{eff}(B_k(S) : B_n(T)) = \frac{1 - \text{cost}(S)}{1 - \text{cost}(T)}.$$

Proof: Theorem 4.6 says that $B_n(S)$ is equal to $\bigcup_{A:|A|=n-k} B_k(S, A)$, and so $B_k(S)$ is a building block for $B_n(S)$. But since every string in T has a prefix in S, then $B_n(S) \subseteq B_n(T)$, so that $B_k(S)$ is also a building block for $B_n(T)$. To calculate $\text{eff}(B_k(S) : B_n(T))$, we use Theorems 3.4 and 4.3:

$$\begin{aligned}
\text{eff}(B_k(S) : B_n(T)) &= \frac{V(B_n(T))E(B_k(S))}{V(B_k(S))E(B_n(T))} \\
&= \frac{2^n 2^{k+1}(1 - \text{cost}(S))}{2^k 2^{n+1}(1 - \text{cost}(T))} \\
&= \frac{1 - \text{cost}(S)}{1 - \text{cost}(T)}. \qquad \square
\end{aligned}$$

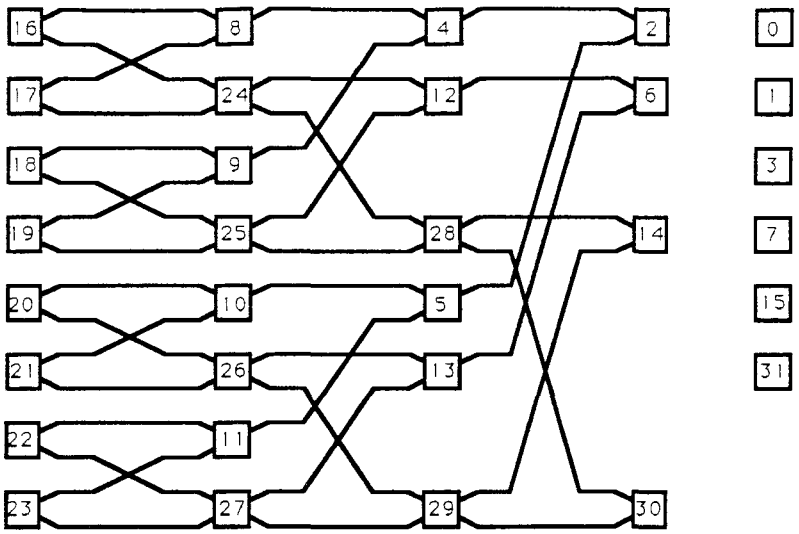


Figure 7. One possible layout of the $\overline{B}_5(\{10\})$ chip used to build the BVD. (All edges are directed from left to right. The vertex labels shown are the decimal equivalents of the actual five-bit binary labels, and the edge labels have been omitted.)

6.2. Lemma. If S is any set of strings, and if $k \leq n$, then $C_k(S)$ prefixes $C_n(S)$.

Proof: By definition, $C_k(S) = S \cup \text{Omit}_k(S)$ and $C_n(S) = S \cup \text{Omit}_n(S)$. A string $s \in \text{Omit}_n(S)$ is a string of length n with no substring from S. The k-bit prefix of s is a string of length k which also has no substring form S, and so this prefix is in $\text{Omit}_n(S)$. Thus every string in $C_n(S)$ is either in S, or has a prefix in $\text{Omit}_k(S)$. ❐

6.3. Theorem. If $k \leq n$, then $\overline{B}_k(S)$ is a building block for $\overline{B}_n(S)$, and

$$\text{eff}(\overline{B}_k(S) : \overline{B}_n(S)) = \frac{1 - \text{cost}(S) - 2^{-k}\text{omit}_k(S)}{1 - \text{cost}(S) - 2^{-n}\text{omit}_n(S)}.$$

Proof: Follows from Theorem 6.1 and Lemma 6.2. ❐

6.4. Example. We return to the Big Viterbi Decoder mentioned briefly in Section 1. The BVD requires the construction of the deBruijn graph B_{13}. The actual construction used at JPL is based on a VLSI chip realization of the graph $\overline{B}_5(\{10\})$, which, by Theorem 5.5 and Example 5.3, is a universal deBruijn building block of efficiency 18/32 (see Figure 7), and so it contains exactly $64 \cdot \frac{18}{32} = 36$ edges.

According to Theorem 6.3, $\overline{B}_5(\{10\})$ is a building block for $\overline{B}_9(\{10\})$, and in the BVD, 16 of the $\overline{B}_5(\{10\})$-chips are wired together on a printed-circuit board to make a $\overline{B}_9(\{10\})$-board. Now $\overline{B}_9(\{10\})$ is a universal deBruijn building block of efficiency 374/512, and so it contains exactly $512 \cdot \frac{374}{512} = 748$ edges. However, $16 \cdot 36 = 576$ of these edges are internal to the component chips, so that each $\overline{B}_9(\{10\})$-board actually has only $748 - 576 = 172$ printed wires. Finally, since $\overline{B}_9(\{10\})$ has efficiency 374/512 as a deBruijn building block, in order to build B_{13}, there will be $2^{14} \cdot (1 - \frac{374}{512}) = 4416$ wires external to the board, or "backplane" wires. In summary:

unit type	*number of units*	*wires/unit*	*total wires*
chip	256	36	9216
board	16	172	2752
backplane	1	4416	4416
			16384

❐

References.

[1] O. Collins, F. Pollara, S. Dolinar, and J. Statman, "Wiring Viterbi Decoders (Splitting Debruijn Graphs)," *JPL TDA Progress Report 42-96 (February 1989), in press.*

[2] L. J. Guibas and A. M. Odlyzko, *J. Comb. Theory* A (30) (1981), pp. 183–208.

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

[4] R. Stanley, *Enumerative Combinatorics, Vol. 1.* Monterey, California: Wadsworth and Brooks-Cole, 1986.

[5] J. Statman, G. Zimmerman, F. Pollara, and O. Collins, "A Long Constraint VLSI Viterbi Decoder for the DSN," *JPL TDA Progress Report 42-95 (November 1988), pp. 134-142.*

Bibliography

1) Introduction to Communication Science and Systems

Pierce and Posner, Plenum Publishing
New York, New York 1980

2) The Theory of Information and Coding Robert J. McEliece

Addison-Wesley Publishing Company
Reading, Massachusettes 1977