Coding Concepts and Block Coding

It's hard to work in a noisy room as it makes it harder to think. Work done in such environment is likely to cause more mistakes. Electronic devices suffer much the same fate. Signals too are apt to be misinterpreted in an environment that is "noisy".

During a conversation how do you determine if you heard the other person correctly? We human *receivers* employ various forms of error detection and correction as we try to make sense of what we are hearing. First of all, we use context and if a word or phrase does not seem to fit the topic of conversation, we automatically replace the confusing section with what we expect to hear. If contextual confusion still exists, we ask the person to repeat what he has just said or we may ask the other person to speak louder. All of these are forms of error correction schemes employed during human communication.

Error detection and correction (EDC) to achieve good communication is also employed in electronic devices and in much the same conceptual way. In fact, without EDC most electronic communication would not be possible. The level of noise and interferences are just way too high in electronic medium. The amount of EDC required and its effectiveness depends on the signal to noise ratio, or SNR.

EDC can be implemented in the five following ways.

Increase signal power

This is same as speaking louder in a noisy room. We intuitively know that as the signal to noise ratio is increased, the errors decrease. Assuming that we can do nothing about the environment, the most obvious thing to do is to increase the transmitter power. But this is not always possible. Most communication devices have no ability to increase power. Examples of power-limited devices are telephone handsets and satellites. Both of these have a fixed amount of power and can not increase their transmitting power beyond a certain point. In fact most are designed to operate at their maximum power and have no spare power available.

There are actually some disadvantages to using this scheme. When amplifier characteristics are not linear, increasing the power means both signal and noise are amplified making the situation worse. You can see this mechanism at work in a radio. When you increase the volume, the noisiness of the signal goes from bad to intolerable

Decrease signal noise

We are at a party and the room is too noisy for conversation, we may move to a quieter corner where there is less noise. If you turn on the air conditioner in the room and the TV reception goes bad, you turn off the offending item to improve the reception. These are some means of noise reduction. In a communication device, the only noise that is under the designer's control is thermal and inter-modulation noise of the system. Assuming that the system is designed to minimize these, we do not have any way of reducing the noise. We are at the mercy of the environment and have to accept this as a given.

Introduce diversity

When listening to a radio, you find that the signal is waxing and waning. Your instinctive response is to move the radio to a different location. With a cordless phone, we may switch channels, or ask the caller to call us on a different line. In satellite communications, in wet and rainy areas, we often have two ground stations separated by some distance, both receiving the same signal so as to increase the SNR. Different polarizations which are used to expand the useable spectrum in satellite communications can also be used to send the same information as a means of error correction. All of these techniques fall under the category of diversity, the main purpose of these is to improve signal quality by taking advantage of redundancy. Another from of diversity comes to use from an unexpected source in the case of cell phones. The bouncing around of the signals causes amplitude reduction but with sophisticated signal processing we can actually use these low-power interfering signals to combine and improve the SNR.

Retransmission

Repeat retransmission is a special form of diversity. With the addition of a return acknowledgement that the signal was received correctly makes it a scheme called Acknowledgement Retransmit or ARQ. ARQ techniques require a duplex transmission. Each block is checked by the receiver, and in case of errors, retransmission is requested. If the SNR is high, there is very little retransmission and the throughput is high. If SNR dips, then overhead increases greatly. Here the tradeoff is between the overall bandwidth, and delay on one side and the receiver complexity (which is reduced greatly) on the other side. ARQ is used most commonly in wired and local area networks. Downloading files over the internet is one place where these techniques are used. You can monitor this occurring when waiting for a page to download on the web. The out-going blocks are the ARQ blocks. In most cases, we can see that the overhead is quite high (usually about 15%).

Forward Error Correction

When a duplex line is not available or is not practical, a form of error correction called Forward Error Coding (FEC) is used. The receiver has no real-time contact with the transmitter and can not verify if a block was received correctly. It must make a decision about the received data and do whatever it can to either fix it or declare an *alarm*.

FEC techniques add a heavy burden on the link either in adding redundant data and adding delay. Also most FEC techniques are not very responsive to the actual environment and the overhead is there whether needed or not. Another big disadvantage is the lower information rate. However at the same time, these techniques reduce the need to vary power. For the same power, we can now achieve a lower error rate. The communication in this case remains simplex and all the burden of detecting and correcting errors falls on the receiver. The transmitter complexity is avoided but is now placed on the receiver instead.

Digital signals use FEC in these three well known methods:

1. Block Codes
2. Convolutional Codes
3. Interleaving

Block codes operate on a block of bits. Using a preset algorithm, we take a group of bits and add a coded part to make a larger block. This block is checked at the receiver. The receiver then makes a decision about the validity of the received sequence.

Convolutional codes are referred to as continuous codes as they operate on a certain number of bits continuously.

Interleaving has mitigating properties for fading channels and works well in conjunction with these two types of coding. In fact all three techniques are used together in many EDC suites such as Digital Video

Broadcast, satellite communications, radio and cell phones and baseband systems such as PCs and CD players.

In this Tutorial we will cover the workings of block codes in detail.

Block Codes

.

Block codes are referred to as (n, k) codes. A block of k information bits are coded to become a block of n bits.

But before we go any further with the details, let's look at an important concept in coding called Hamming distance.

Let's say that we want to code the 10 integers, 0 to 9 by a digital sequence. Sixteen unique sequences can be obtained from four bit words. We assign the first ten of these, one to each integer. Each integer is now identified by its own unique sequence of bits as in Table below. The remaining six possible sequences are unassigned.

Table I – Assigning 4-bit sequences to the 10 integers

Sequence	Integer	Sequence	Integer
0000	0	0001	1
0010	2	0011	3
0100	4	0101	5
0110	6	0111	7
1000	8	1001	9
1010	Unassigned	1011	Unassigned
1100	Unassigned	1101	Unassigned
1110	Unassigned	1111	Unassigned

The universe of all possible sequences is called the code space. For 4-bit sequences, the code space consists of 16 sequences. Of these we have used only ten. These are the valid words. The ones not used or unassigned are invalid code words. They would never be sent, so if one of these is received then the receiver assumes that an error has occurred.

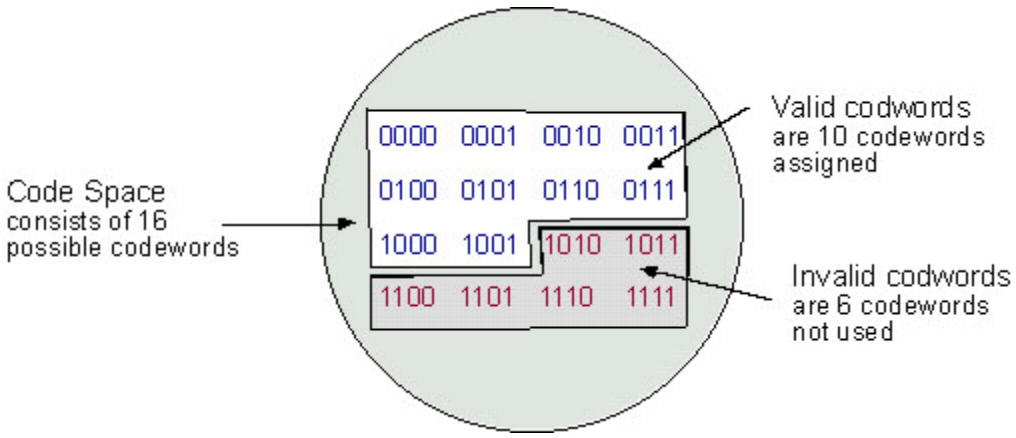


Figure 1 – Codeword space is a list of all words possible form a particular alphabet size such as 4-bit

words shown. The valid codewords are a subset of these codewords that are mapped with actual data such as the 10 codewords that are each assigned to one of the 10 digits.

Hamming Weight: The Hamming weight of this code scheme is the largest number of 1's in a valid codeword. This number is 3 among the 10 codewords we have chosen. (the ones in the while space)

Concept of Hamming Distance

In continuous variables, we measure distance by Euclidean concepts such as lengths, angles and vectors. In the binary world, distances are measured between two binary words by something called the Hamming distance. The Hamming distance is the number of disagreements between two binary sequences of the same size. It is a measure of how apart binary objects are.

The Hamming distance between sequences 001 and 101 is = 1
The Hamming distance between sequences 0011001 and 1010100 is = 4

Hamming ***distance*** and ***weight*** are very important and useful concepts in coding. The knowledge of Hamming distance is used to determine the capability of a code to detect and correct errors.

Although the Hamming ***weight*** of our chosen code set is 3, the minimum Hamming ***distance*** is 1. The codeword 1011 and codeword 1010 differ by only one bit. A single bit error can turn 1011 into 1010. The receiver interpreting the received sequence of 1010 will see it as a valid codeword will never know that actually the sent sequence was 1011. So it only takes one bit error to turn one valid codeword into another valid codeword such that no error is apparent. Number 2 can then turn into Number 8 with one bit error in the first bit.

Now extend the chosen code set by one parity bit. Here is what we do, if the number of 1's is even then we extend the sequence by appending a 1, and if the number of 1's is odd, then we add a zero at the end of the sequence.. The codespace now doubles. It goes from 16 possible codewords to 32 codewords since codespace is equal to 2^N, where N is the length of the sequence. We can extend the sequence yet again using the same process and now the code space increases to 64. In the first case the number of valid codewords is 10 out of 32 and in the second case it is still 10 but out of 64, as shown in Column 4.

Table II – Codespace vs. valid codewords, extending the sequence by adding a parity bit

Integer	4 bit Seq.	Extended by 1 bit	Extended again by 1 bit
0	0000	00000	000000
1	0001	00011	000110
2	0010	00101	001010
3	0011	00110	001100
4	0100	01001	010010
5	0101	01010	010100
6	0110	01100	011000
7	0111	01111	011110
8	1001	10010	100100
9	1010	10100	101000
	6 more not used	22 more not used	52 more not used

We now have three code choices, with 4, 5 or 6-bit words. The Hamming weight of the first set is 3, 4 for

the second set and 5 for the last set. The minimum Hamming distance for the first set is 1. The minimum Hamming distance for the second set is 2 bits and 3 for the last set.

In the first set, even a single bit error cannot be tolerated (because it cannot be detected). In the second set, a single error can be detected because a single bit error does not turn one code word into another one. So its error detection ability is 1. For the last set, we can detect up to 2 errors. So its error detection capability is 2 bits.

We can generalize this to say that the maximum number of error bits that can be detected is

$t = d_{min} - 1$

Where d_{min} is Hamming distance of the codewords. For a code with $d_{min} = 3$, we can both detect 1 and 2 bit errors.

So we want to have a code set with as large a Hamming distance as possible since this directly effects our ability to detect errors. It also says that in order to have a large Hamming distance, we need to make our codewords larger. This adds overhead, reduces information bit rate for a given bandwidth and proves the adage that you cannot get something for nothing even in signal processing.

Number of errors we can correct

Detecting is one thing but correcting is another. For a sequence of 6 bits, there are 6 different ways to get a 1 bit error and there are 15 different ways we can get 2-bit errors.

No. of 2-bit combinations = $\frac{6!}{2!\quad 4!} = 15$

What is the most likely error event given that this is a Gaussian channel? Most likely event is a 1 bit error then 2 bit errors and so on. There is also a possibility that 3 errors can occur, but for a code of 6 bits, the maximum we can detect is two bits. This is an acceptable risk since 3 bit errors are highly unlikely because if the transition probability p is small (<<1), the probability of getting three errors is cube of the channel error rate,

$P_E(N) = p^N$

Of course it is indeed possible to get as many as 6 errors in a 6 bit sequence but since we have no way of detecting this scenario, we get what is called a decoding failure. This is also termed the probability of undetected errors and is the limitation on what can be corrected.

The number of errors that we can correct is given by

$$t(\text{int}) = \frac{d_{\min} - 1}{2}$$

For a code with $d_{min} = 3$, the correction capability is just 1 bit.

Creating block codes

The block codes are specified by (n.k). The code takes k information bits and computes (n-k) parity bits from the code generator matrix. Most block codes are systematic in that the information bits remain unchanged with parity bits attached either to the front or to the back of the information sequence.

Hamming code, a simple linear block code

Hamming codes are most widely used linear block codes. A Hamming code is generally specified as (2^n-1, 2^n-n-1). The size of the block is equal to 2^n-1. The number of information bits in the block is equal to 2^n-n-1 and the number of overhead bits is equal to n. All Hamming codes are able to detect three errors and correct one. Common sizes of Hamming codes are (7, 4), (15,11) and (31, 26). All of these have the same Hamming distance.

Lets take a look the 7,4 Hamming code. In this code, the length of the information sequence is 4 bits. So there are only 16 possible sequences. The table below lists all 16 of these sequences.

Table III- 16 possible sequences from a 4 bit word

Seq No.	i0	i1	i2	i3
1	0	0	1	
2	0	1	0	
3	0	1	1	
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	0	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	0	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1
16	0	0	0	0

We would like to add three parity bits to the above information sequences Eight different combinations of 3 bits are possible. We ignore the all-zero combination and write down the remaining seven combinations in Table IV.

Table IV – Creating a parity matrix

Sequence			
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Make a note of rows 1, 2 and 4. All other rows are linear combinations of these three basis vectors. This property comes in handy when you are proving why the process I have outlined here works. But we are not going to dwell too much on the theory here, just the practical aspect of what this simple construct can do for us.

Let's call these seven rows matrix H. It is called the parity matrix of the code. It has n rows and n-k columns. For a code of (7, 4), it has 7 rows and 3 columns. We can reorder the rows of this matrix any which way we want. The only thing to keep in mind is that we want the basis rows to be together either on top or at the bottom. Each reordering will give us a new Hamming code. Following are just two ways we can order the rows of H, each of these will result in a different code.

$$H = \left[\begin{array}{ccc} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \qquad \text{A different ordering of} \qquad H = \left[\begin{array}{ccc} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ \hline 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{array}\right]$$

Now we take the upper part of this matrix (under the line) and create a new matrix called G

$$G = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}\right]$$

Take the upper part of the H matrix (the one on the left) and add a unit matrix to the right of it, making a new matrix of size (4 x 7). This is our generator matrix, the one that tells us how the 16 info sequences will be mapped to the 16 valid codewords.

Rearrangement of the rows of H will lead to a different G matrix and hence will change the mapping. The essential point to this is that two block codes of the same size may create an entirely different mapping depending on how the rows are ordered, but each mapping has the same Hamming weight and has the same error correction/detection capability.

Now all we have to do is multiply the information vector d with matrix G to get a codeword c.

c = d . G

For information sequence [0, 1, 1, 0], we get the following transmitted codeword of length 7.

$$c = [0 \quad 1 \quad 1 \quad 0] \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} = [0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0]$$

Coder architecture

The coder utilizes a Linear Feedback Shift Register circuit (LSFR). Using the LSFR, The figure below shows how the H matrix is produced using only shifts and XOR. We can produce the H matrix containing all seven rows. First five of these are shown in the figure.

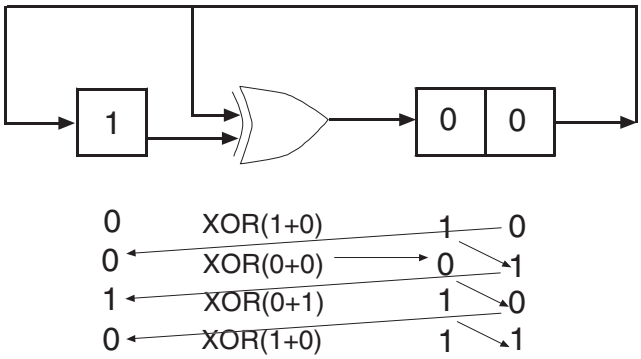
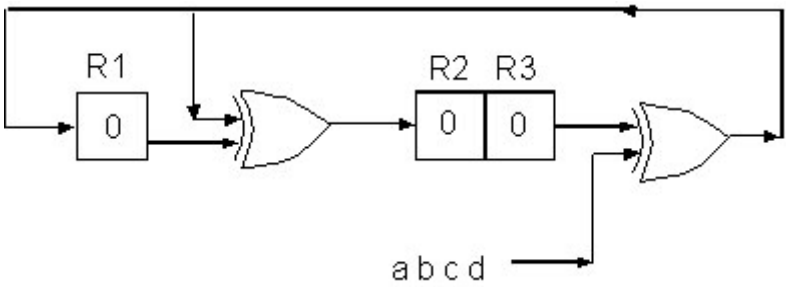


Fig 2 – Using an LSFR to create the H matrix

The following encoder architecture, proposed by Benjamin Arazi in Ref. 1 (a great book), and based on the above LSFR produces the codeword with ingenuity and little fuss. Taking the LSFR in the Figure 2 which produces the rows of H matrix, we feed it an information vector, a b c d. Sequential shift of these bits through this LSFR give us the following contents in the three shift registers.



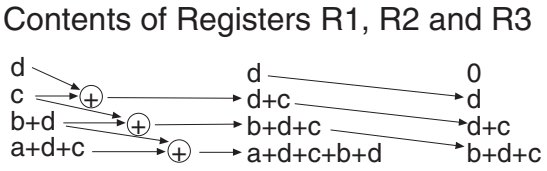


Fig 3 – The encoder architecture using an LSFR

The last row gives the three parity bits that are then attached as tail bits to the information sequence to create the valid codeword.

We can also then say that the equation for the parity bits is

P1 = a + d + c
P2 = a + d + c + b + d
P3 = b + d + c

For sequence of 0 1 1 0 , we get

P1 = 0 + 0 + 1 = 1
P2 = 0 + 0 + 1 + 1 + 0 = 0
P3 = 1 + 0 + 1 = 0

The codeword is 1 0 0 0 1 1. This codeword is different from the one calculated above because here the starting point of the registers was different.

Decoding

The main concept in decoding is to determine which of the 2^k valid codewords was transmitted given that one of the 2^n has been received. We will continue with our example of the (7, 4) block code.

Decoding of block codes is pretty easy. We compute something called a **Syndrome** of an incoming codeword. We do that by multiplying the incoming vector by the transposed parity matrix H. (Remember we multiply with matrix G on the encoder side and by H on the decoder side.)

$s = H^T . c$

The size of the syndrome will be equal to (n-k) bit. For our example of the (7, 4) code, it will be three bits long. Without proving, let me state that, an all-zero syndrome means no error has occurred. So we want the syndrome to be zero.

Let's multiply the received code vector [0 1 1 0 1 1 0] with the H^T matrix, to see if we get all zeros since we know that this is a valid codeword.

$$s = \left[\begin{array}{cccc|ccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

s = [0 0 0]

The results is indeed [0 0 0], an all zero syndrome.

Let's see what syndrome we get when an error is made. The received vector is now [**1** 1 1 0 1 1 0], the left most bit is received inverted. Now we compute the syndrome again and get

$$s = \left[\begin{array}{cccc|ccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

s = 1 1 1

Now compute the ordinal number from this sequence, we get 7. This tells us that the 7th bit is in error. Knowing this, the decoder corrects the 7th bit, which is the left most bit, from 1 back to 0)

A syndrome table can be pre-created in the decoder so that a quick mapping to the bit location where the error has occurred can be made. This is done by changing a valid code vector one bit at a time and looking at the computed syndrome. We get the following syndrome table for this code.

Table V – Syndrome to error location mapping

Error pattern	Syndrome
0 0 0 0 0 0 0	0 0 0
1 0 0 0 0 0 0	1 1 1
0 1 0 0 0 0 0	0 1 1
0 0 1 0 0 0 0	1 0 1
0 0 0 1 0 0 0	1 1 0
0 0 0 0 1 0 0	1 0 0
0 0 0 0 0 1 0	0 1 0
0 0 0 0 0 0 1	0 0 1

Here is an another example, here bit no. 4 as been received in error.

$$s = \left[\begin{array}{cccc|ccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

s = 1 1 0

We see from Table V that bit number 4 has been received in error. The syndrome tells us not only that an error has occurred, but which bit (its location) has been received in error so it can be corrected. All single bits can be detected and corrected in this manner. Since the probability of a single error is much larger than t + 1 error, this method of decoding is called the Maximum Likelihood Decoding. MLD is the most efficient method even though other method such as nearest neighbor would lead to somewhat better results at the expense of decoding speed and memory.

When two errors occur, as in the case of the vector below, we get a non-zero syndrome. For non-zero syndrome, we correct the error based on the syndrome table but this lets an error pass through unbeknownst.

c_{sent} = 0 1 1 0 1 1 0

$c_{sentrcvd}$ = 0 1 1 0 0 0 0

Bits no. 2 and 3 are received inverted.

The syndrome for this vector is

$$s = \left[\begin{array}{cccc|ccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 01 & 0 \end{bmatrix}$$

s = 1 1 0

This causes us to change this vector to

d = 0 1 1 1 0 0 0

which is obviously wrong.

What happens when we have 3 error or 4. Up to 3 erros, the syndrome will be non-zero but any number of errors larger than 3 may give a 0 syndrome so that the errored block goes through undetected.

Decoding structure

The following circuit automatically corrects one bit errors. It utilizes the LSFR circuits of Fig 2 and 3 with addition of an AND gate. As the message is shifted in, the rows of H matrix are produced by R2 register. After n shifts, the register again goes back to 000. As long as the AND gate does not produce a 1, the incoming message is shifted out as is, with no correction. It is left to as an exercise for the proverbial student to work out the details.

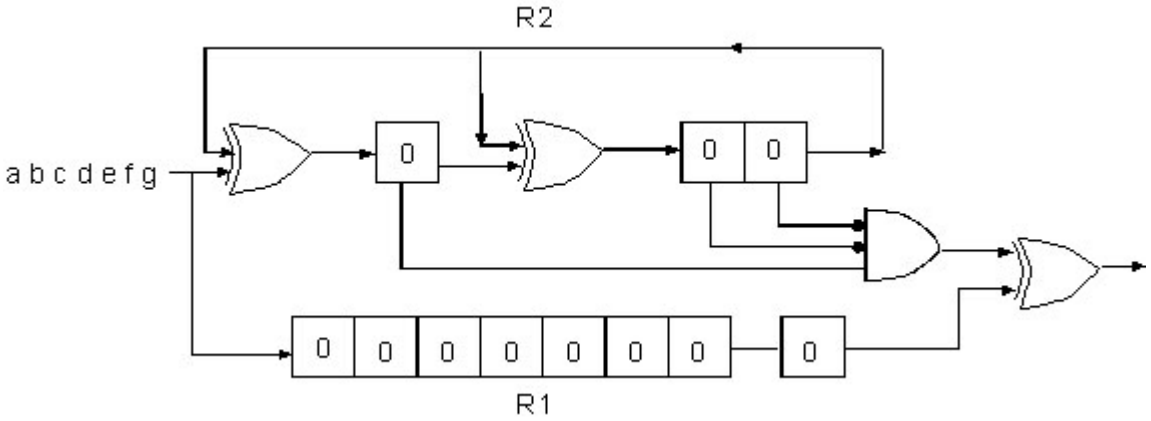


Fig 4 – Decoder that automatically corrects of one bit errors

References

1: A Common sense approach to the theory of error correcting codes
by Benjamin Arazi, MIT press, 1988

2. Digital Communications, I. A. Glover and P M Grant, Prentice Hall

mntcastle@earthlink.net

Other tutorials at:
www.complextoreal.com