

MATH MODEL ON ERROR DETECTION AND CORRECTION

Mukunda D S¹, Poornachandra prakash², Manoj kumar P A³, Pavan kalyan V⁴, Prof.Pavan kumar E⁵

¹⁻⁵Department of Electronics and Communication, Sai Vidya Institute of Technology, Bengaluru, India

Abstract

Communication is a powerful tool that aids us to connect with people. In today's world digital communication has made our life much easier. But in the process of digital communication, the messages transmitted are often vulnerable to noise. In other words, the information signal sent from the transmitter gets corrupted even before receiving back at the receiver. Hence the receiver must be capable of detecting the random error bits added to the original data. These random error bits are due to environmental interferences and physical defects or even due to technology scaling and variations in parameters and noise levels.

This paper describes mathematical error detection/correction models (techniques) that can be utilized to protect the system against various types of errors. These detection/correction models have their own performance, complexity levels and requires a lot of effort to achieve the maximum efficiency in the digital transmission. This paper enumerates many error detection and correction math models that allow reliable delivery of digital data across unreliable communication channels.

Keywords: bit flip, errors memory, error detection and correction, parity, hamming code.

INTRODUCTION

In most cases, we assume that the digital circuits are more immune to noise and they process the information correctly. But there are interferences which affects occasionally the value of the digital signal. In digital system, the analog signals will manipulate themselves into digital bits they are often called as *Digital stream*. These interferences which arise will change the value of the sequence from logic 0 to 1 or from logic 1 to logic 0. We sometimes prosaically call this a *bit flip* (Ashenden, 2008, p. 58) . This change in the position of the bits leads to catastrophic *errors* in the output.

During transmission, when the input data is not matched with the output data, it is termed as *error*.

Fig below, illustrates the occurrence of error in the encoder and decoder of the digital system:

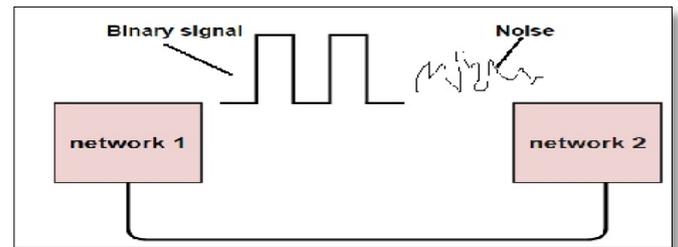


Fig 1: Error in digital system

In digital transmission these errors are called as *bit errors*. These bit errors are likely to occur in the memory of the digital circuits. Memories are the most important component for storing as well as retrieving any digital data that are needed at particular time (Shobha & Kavya, 2015, p. 1). Types of embedded memory include ROM, SRAM, DRAM, flash memory etc. Based on error impacting on memory, there are 2 types,

- Soft errors
- Hard errors

Soft errors are *transient* and consist of a bit flip in the memory without permanent effect on the storing capacity to store the data.

These errors are typically caused by high energy neutrons generated by collision of cosmic rays with atoms in the atmosphere. The error is soft because the circuit/device is not permanently damaged by this radiation. The frequency of soft error occurrence *soft error rate* depends on the way in which the device is manufactured and the location in which they operate.

Errors that persist in the memory are called hard errors. If the device/memory is affected by these types of hard errors, it loses all its capability to store the data and can no longer be used for read/write operation. A read operation will always yield to 0 or 1 value regardless the previous bit value written. These errors are caused by the manufacturing defects or from the electrical malfunctions after prolonged use.

If signal is carrying binary encoded data, any change, can alter the meaning of the data (Shobha & Kavya, 2015, p. 1). Based on this change of data during transmission, there are two types of errors,

- Single bit error
- Burst error

Single bit error means only one bit is changed from bit 0 to bit 1 or from bit 1 to bit 0 ("Computer Network | Error Detection - javatpoint," n.d.). Occurrence of these errors is parallel communication as data is transferred bitwise in single line and that single line can be corrupted.

Fig below shows occurrence of single burst error:



Fig 2: Single bit error

Burst error is the change in multiple/set of bits of data sequence from transmitter to receiver. These types of errors are common in both serial and parallel communication. Burst error is calculated from first bit change to last bit change (2019).

Fig below illustrates burst error occurrence:

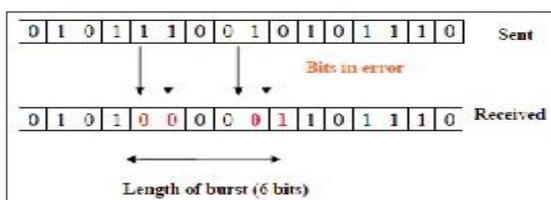


Fig 3: Burst error

With this knowledge, the errors can exist in multiple ways and affect the process of transmission. Especially the memory is more susceptible to bit errors than logic circuits using flip-flops, registers for storage.

At last it becomes more common and important to include some error detection and correction techniques both in memory and logic circuits to overcome these errors.

ERROR DETECTION TECHNIQUES

Memories are more vulnerable to bit errors than using flip-flops and registers for storage, due to the storage density and how long the data stored in memories, it is more obvious to include some techniques of error detection in memory circuits than in flip-flops and registers.

The process of identifying the errors that are present in the transmitted data is called error detection. We use some redundancy bits to find these errors, by adding to the data before it is transmitted. These redundancy bits

constitute to a code and are called Error detecting codes and the technique is called Error detection.

Some popular techniques for error detection are:

1. One dimensional Parity check
2. Two-dimensional Parity check
3. Cyclic redundancy check

1. One dimensional Parity check

One technique that is commonly used for detecting error is parity, which deals with counting the number 1's present in a code word. Parity error checking deals with increasing the code length by adding one extra bit, called the parity bit. In the even parity scheme, the parity bit in each code word is set to 0 or 1 to make sure that the total number of 1 bits is even (For example, if the original code word is 1110, the augmented code word is 11101) whereas odd parity scheme sets the parity bit to ensure that the total number of 1 bits is odd, for example, if the original code word is 1010, the augmented code word is 10101 (Ashenden, 2008, p. 60), (Subiya Yaseen, Hussain Ahmed, & Kehkeshan Jallal, 2013, pp. 1-3).

In an even parity scheme, valid code words have even parity, and invalid code words have odd parity. If the error causes a 0 bit to change to 1, the number of 1 bit is increased by one, and the parity becomes odd. Similarly, if error changes a 1 bit to 0, the number of 1 bit is decreased by one, and parity becomes odd. So in order to check whether a bit has flipped or not, we simply count the number of 1 bits, including the parity bit. If the count obtained is odd, parity has been reversed (Ashenden, 2008, p. 60). Indicating that an error has occurred.

This scheme below shows the illustration of parity checking:

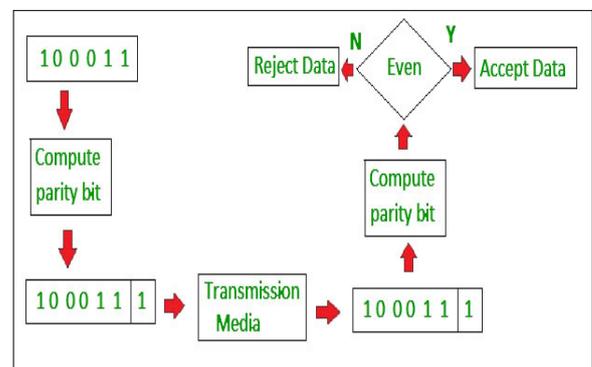


Fig 4: One dimensional parity check

The circuit which adds a parity bit is called Parity generator. The circuit which verifies the parity is called Parity checker.

Here are some of the example code words illustrating even and odd parity:

Message with even parity		Message with odd parity	
Message	Parity	Message	Parity
000	0	000	1
001	1	001	0
010	1	010	0
011	0	011	1
100	1	100	0
101	0	101	1
110	0	110	1
111	1	111	0

Fig 5: Even and Odd parity

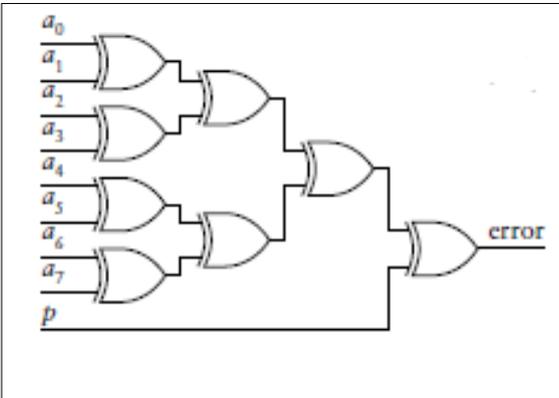
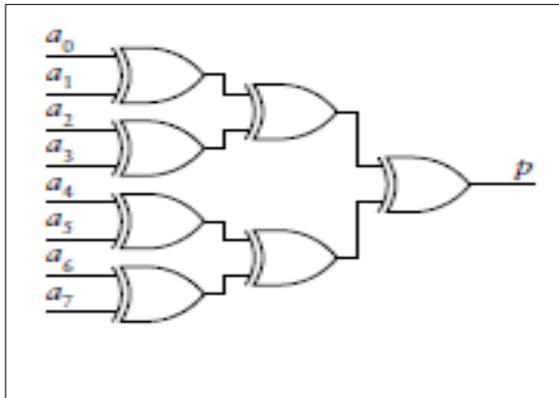


Fig 6: Parity Checker [1], Fig 7: Parity generator [1]

In DRAM memory systems the use of parity memory is to provide single-bit error detection capability.

Figure below illustrates the generation of a parity bit for data written to DRAM: parity, indicating either an even or an odd number of '1' bits in the data [2]

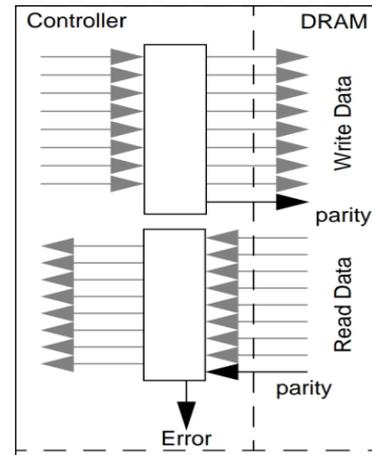


Fig 8: Parity checking in DRAM

2. Two-dimensional Parity check

In a simple parity check, the parity check bits are calculated only for each row. In two-dimensional parity check technique we calculate parity bits for both rows and also for columns then, both are sent along with the data. At the receiver we compare the sent and received parity bits.

The figure below illustrates the same:

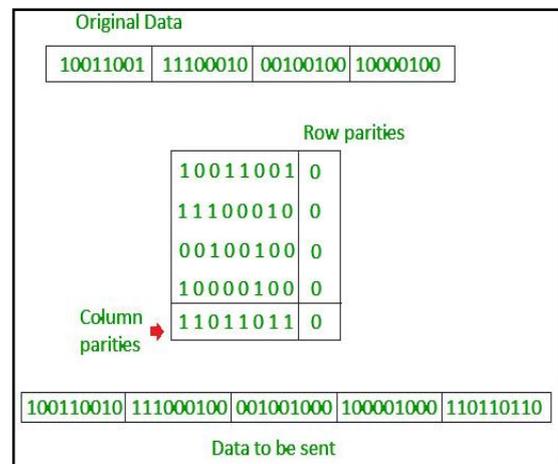


Fig 9: Two dimensional parity check

Drawbacks of parity

The problem of using parity technique to check for errors is that it only allows us to detect a *single bit flip* in a code word, it does not allow us to identify which bit is flipped, nor does it allow us to detect an even number of bit flips (Ashenden, 2008, p. 240). If the interference (error) flips two, four, six, or any even number of bits parity is preserved, so we miss that error (Ashenden, 2008, p. 240). However, in practical applications, the probability of changing of multiple bits is extremely less. If the application requires only detecting the error, parity method is most suitable.

3. Cyclic Redundancy Check (CRC)

These types of codes are used for *error detection* and error correction and also for *encoding* (2019). Consider a linear block code (n, K) , k is the length of the message (the number of information bits). n indicates the total length of the transmitted message after adding check bits. They can be easily designed using shift-registers with feedback connections. With every cyclic shift we can generate a new code word.

CRC Code Generation

- CRC is based on binary division (“Error Detection in Computer Networks,” 2019).
- CRC involves the process of adding *redundancy bits* called ‘*redundancy check bits*’ to the end of the data and make the resulting data exactly divisible without any remainder.

We can add some more zeros to the original data based on the requirement of check bits. The new binary data sequence is divided (modulo 2- division) with a new code of the length $n + 1$, here n represents the number of check bits which is added to the original data. If any remainder, we add it to the dividend to get a new cyclic code. The generated code word is completely divisible that is used in generation of code (2019).

- If we do not get any remainder, the data unit is presumed to be *correct* and is accepted.
- If any remainder occurs, we conclude that the data is *corrupted* and it is rejected.

The methodology is described as below:

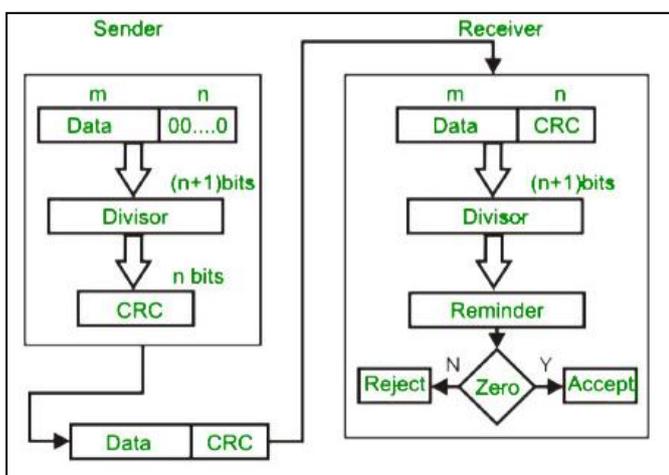
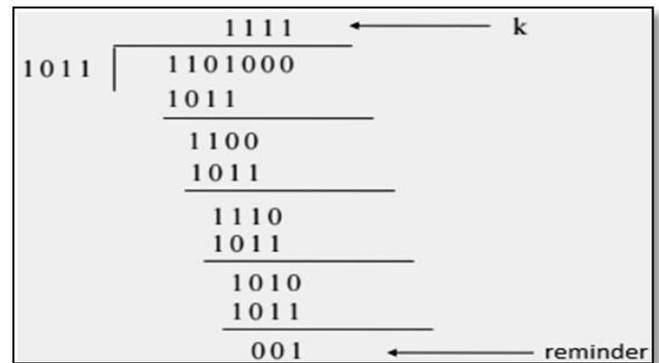


Fig 10: Cyclic Redundancy Check

Example for CRC



This data unit has occurred with a remainder and therefore rejected.

These CRC techniques are exclusively in digital communication systems. CRC codes are capable of providing high and effective level of protection.

For one bit and two bit errors, the probability of CRC is 100 %, burst error of length $n - 1$, has the probability of CRC error detecting 100 %, burst error of length $n + 1$, has the probability of error detecting $1 - (1/2)^{n-1}$, burst error of length greater than $n - 1$, the probability of error detecting is $1 - (1/2)^n$ (2019).

ERROR CORRECTION TECHNIQUES

The basic error correction techniques consists of two types (2019), they are:

1. Single bit error correction
2. Burst error correction

Single bit error correction is a method of correction single bit error.

Burst error correction is a method of correction of burst error.

I. Alpha Numeric Codes

These codes represent alphabets, numbers, punctuation mark and mathematical symbol. They are implemented in input output devices like printers, keyboard and monitor etc.

The types of alphanumeric codes (2019) are:

1. Morse Code
2. Baudot code
3. Hollerith code
4. ASCII Code
5. EBCDIC Code
6. Unicode Code

1. Morse code

Even before computer and digital electronics era, Morse code was very popular code. Morse code is used in telecommunication as a *telegraphic* code.

2. Baudot code

This code is invented in 1870 by *Emile Baudot*; this code uses 5 elements to represent the alphabet. This is most commonly used in *teleprinters*.

3. Hollerith code

This was developed by *Herman Hollerith* in 1896. It is also called as punched card codes, because the 12-bit code is punched on card for transmitting information.

4. ASCII Code

American Standard Code for Information Interchange (1&1 IONOS Inc., 2020) is most commonly used alphanumeric code. This was invented in 1967. ASCII code is 7-bit code, it has 128 characters. They are 26 lower case letter, 26 upper case letter, 33 special characters and symbols, 33 control characters and 10 digits.

5. EBCDIC Code

Extended Binary Coded Decimal Interchange code was first developed by the company called *IBM*. It is 8-bit code, it can represent 256 characters. They are 26 lower case letter, 26 upper case letter, 33 special characters and symbols, 33 control characters and 10 digits. In EBCDIC code numbers are represented by 8421.

6. UNICODE

The drawback of *ASCII* code and *EBCDI* codes are, they do not suit to all languages and they do not have enough characters to represent each and every data. This is overcome by *UNICODE*. *UNICODE* uses different characters to represent every number and it is compatible with all languages and it can represent all types of data. It is 16-bit code; it can represent 65536 different characters.

II. Hamming Code

Hamming code is used detect the error and correct it in the data. This code uses the parity bits in the code word that depends on number of information bits. The extra bits added to the data sequence are called as *redundancy bits*. Redundancy bits are placed at particular position to remove the error.

Hamming code can correct *single- and double-bit errors*. Basic Structure of Hamming Encoder and Decoder (Chauhan, 2016) are as shown below:

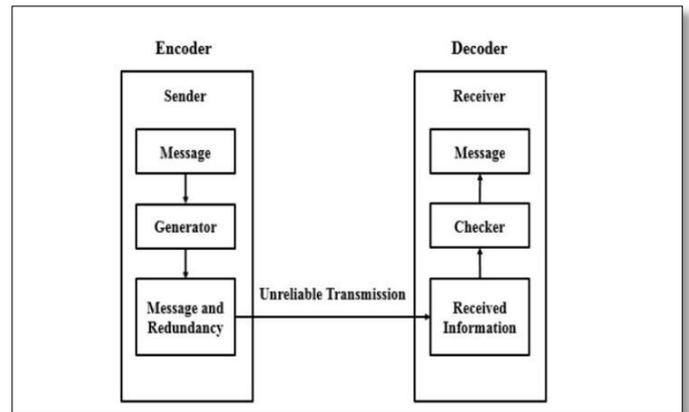


Fig 12: Hamming encoder and Decoder

Hamming codes are calculated using matrices. These matrices are code *generator matrix (G)*, *parity matrix (P)* and *parity check matrix (H)*.

Generator matrix (G): It is a matrix whose rows form basics for a linear code (Hamming code).

Parity matrix (P): It is a matrix which describes the linear relations that the code must satisfy.

Parity check matrix (H): The parity matrix along with the Identity matrix is termed as parity check matrix.

The computation of Hamming codes using code generator matrix and parity check matrix is as illustrated in the below example: Ex: *The parity check bits of a (7, 4) LBC are given below, the data bits to be sent are 1101. Find the hamming code constructed at the encoder and decoder.* [3]

$$\begin{aligned}
 c_1 &\rightarrow d_1 \oplus d_3 \oplus d_4 \\
 c_2 &\rightarrow d_1 \oplus d_3 \oplus d_4 \\
 c_3 &\rightarrow d_2 \oplus d_3 \oplus d_4 \\
 c_4 &\rightarrow d_1 \\
 c_5 &\rightarrow d_2 \\
 c_6 &\rightarrow d_3 \\
 c_7 &\rightarrow d_4
 \end{aligned}$$

Sol: Given, $(n, K) = (7, 4)$
 n = code word length;
 K = message bits;
 $n - K$ = parity bits = $7 - 4 = 3$;

From above data we can get Generator Matrix as:

$$\text{Codeword, } [C] = [D] * [G]$$

Where D = data

G = generator matrix

$$[c] = [D][G]$$

$$[d_1 \ d_2 \ d_3 \ d_4] \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S = r * H^T$$

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

The general form of Generator matrix is,

$$[G] = [I_k \ | \ P]$$

Modulo 2 addition between r and h^T

Where I is Identity matrix

$$S = [1 \ 0 \ 0]$$

From the generator matrix, the parity matrix is obtained as below,

The syndrome pattern is in the 1st row of the H transpose, which indicates that the 1st bit in received code is corrupted. Corrected code word is [0 1 0 0 1 0 1]. Thus, data obtained is same as the data transmitted using hamming code.

$$P = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}; \quad P^T = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}; \quad I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

SIMULATION USING XILINX ISE DESIGN SUIT.14

From generator matrix we get parity check matrix (H):

```

13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ////////////////////////////////////////////////////////////////////
21 module ham_enc(
22     input clk,
23     input [3:0] d,
24     output [6:0] c
25 );
26 reg [6:0] c;
27 always @(posedge clk)
28 begin
29     c[6]=d[3];
30     c[5]=d[2];
31     c[4]=d[1];
32     c[3]=d[1]^d[2]^d[3];
33     c[2]=d[0];
34     c[1]=d[0]^d[2]^d[3];
35     c[0]=d[0]^d[1]^d[3];
36 end
37
38
39 endmodule
40

```

$$H = [I \ | \ P^T]$$

$$= \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Encoding:

$$\text{Codeword (C)} = [1 \ 1 \ 0 \ 1] * \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$= [0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1]$$

The hamming codeword transmitted is [0 1 0 0 1 0 1].

Fig 13: Hamming Encoder

Decoding:

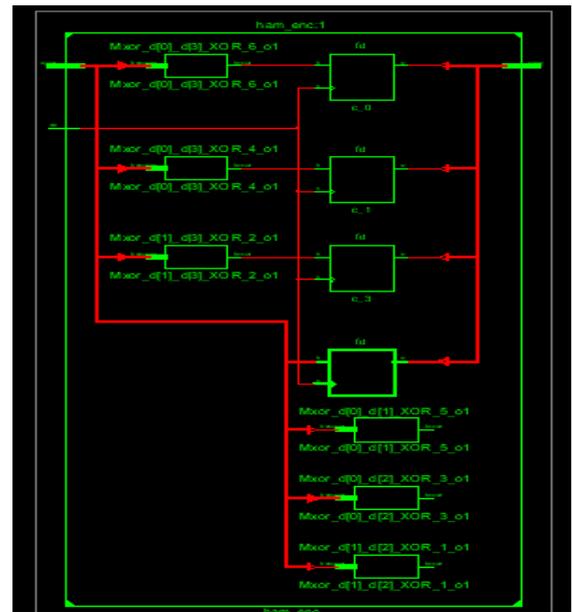
If no error during transmission then the [received code=transmitted code.] If there is an error, then we multiply r and H^T to obtain syndrome pattern.

$$\text{Let } r = [1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1]$$

```

24 module hamenc_tb;
25
26
27 // Inputs
28 reg clk;
29 reg [3:0] d;
30
31 // Outputs
32 wire [6:0] c;
33
34 // Instantiate the Unit Under Test (UUT)
35 ham_enc uut (
36     .clk(clk),
37     .d(d),
38     .c(c)
39 );
40
41 initial begin
42     forever
43     begin
44         clk=1;
45         #50 clk=0;
46         #50 clk=1;
47     end
48 end
49 initial begin
50     d=4'b0000;
51     #100 d=4'b0001;

```



```

52 #100 d=4'b0010;
53 #100 d=4'b0011;
54 #100 d=4'b0100;
55 #100 d=4'b0101;
56 #100 d=4'b0110;
57 #100 d=4'b0111;
58 #100 d=4'b1000;
59 #100 d=4'b1001;
60 #100 d=4'b1010;
61 #100 d=4'b1011;
62 #100 d=4'b1100;
63 #100 d=4'b1101;
64 #100 d=4'b1110;
65 #100 d=4'b1111;
66 end
67 initial
68 begin
69     $monitor ($time,"clk=%b,d=%b,c=%b",clk,d,c);
70 end
71 endmodule
72
73

```

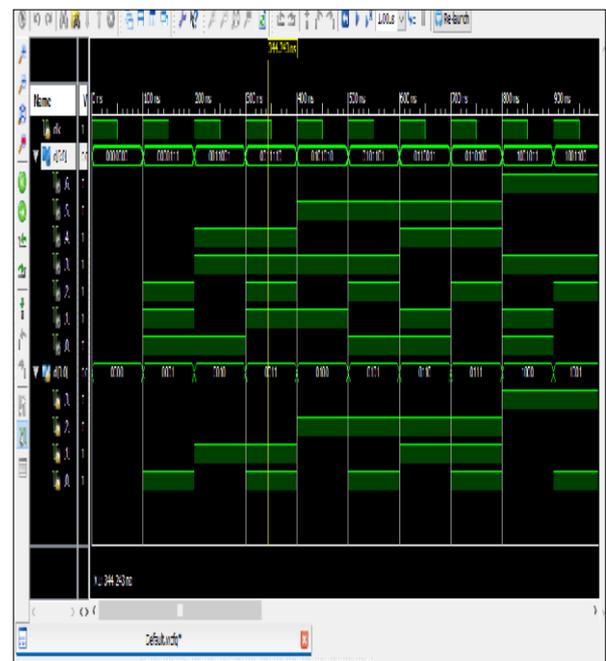


Fig 14: Test bench Code (encoder)

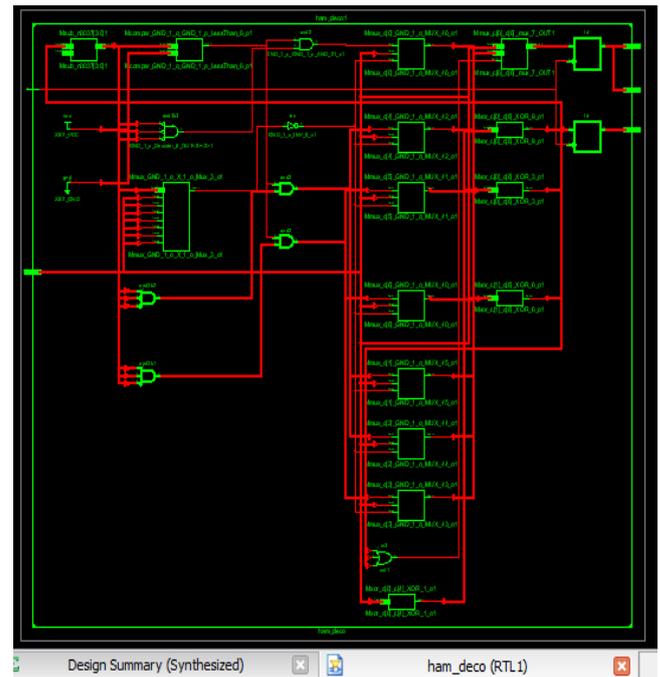
Fig 15: RTL Schematic and Simulation Output (encoder)

```

21 module ham_deco (
22     input clk,
23     input [6:0] c,
24     output [2:0] s,
25     output [6:0] c2,
26     output [3:0] d
27 );
28 reg[2:0] s;
29 reg [6:0] c2;
30 reg [3:0] d;
31 always@(posedge clk)
32 begin
33     s[2]= c[0]^c[4]^c[5]^c[6];
34     s[1]= c[1]^c[2]^c[5]^c[6];
35     s[0]= c[0]^c[4]^c[4]^c[6];
36     c2=c;
37     if (s)
38         c2[s-1]=~c[s-1];
39     end
40     always@ (c2)
41     begin
42         d[0]=c2[2];
43         d[1]=c2[4];
44         d[2]=c2[5];
45         d[3]=c2[6];
46     end
47 endmodule
48

```

Fig 16: Hamming decoder



```

23 //////////////////////////////////////////////////
24
25 module hamdec_tb;
26
27     // Inputs
28     reg clk;
29     reg [6:0] c;
30
31     // Outputs
32     wire [2:0] s;
33     wire [6:0] c2;
34     wire [3:0] d;
35
36     // Instantiate the Unit Under Test (UUT)
37     ham_deco uut (
38         .clk(clk),
39         .c(c),
40         .s(s),
41         .c2(c2),
42         .d(d)
43     );
44
45     initial begin
46         forever
47         begin
48             clk=1;
49             #50 clk=0;
50             #50 clk=1;

```

```

51     end
52 end
53 initial
54 begin
55     c=7'b0000000;
56     #100 c=7'b0000111;
57     #100 c=7'b0011001;
58     #100 c=7'b0011110;
59     #100 c=7'b0101010;
60     #100 c=7'b0101101;
61     #100 c=7'b0110011;
62     #100 c=7'b0110100;
63     #100 c=7'b1001011;
64     #100 c=7'b1001100;
65     #100 c=7'b1010010;
66     #100 c=7'b1010101;
67     #100 c=7'b1100001;
68     #100 c=7'b1100110;
69     #100 c=7'b1111000;
70     #100 c=7'b1111111;
71 end
72 initial
73 begin
74     $monitor($time,"c=%b, clk=%b, s=%b, c2=%b, d=%b", c, clk, s, c2, d);
75 end
76
77 endmodule
78

```

Fig 17: Test bench code (Decoder)

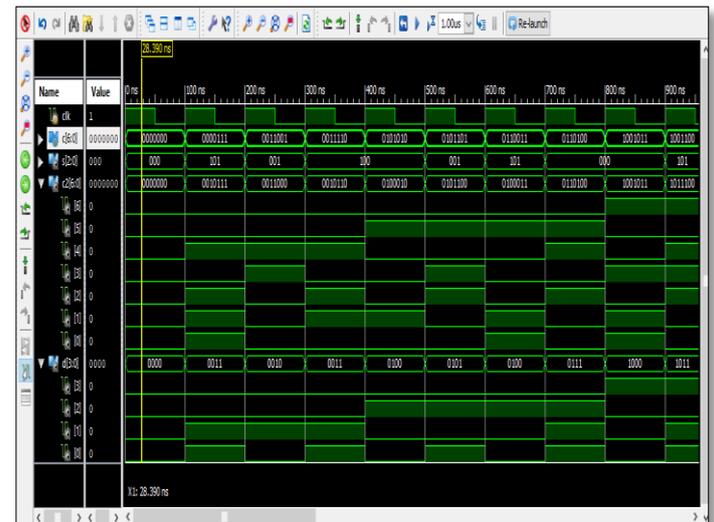


Fig 18: RTL schematic and Simulation Output (decoder)

CONCLUSION

Coding techniques gives us the required protection for the information against a noise. This Error *detection and correction* mechanisms are widely used today and one of the effective techniques which exist in the world. It reduces the effect of errors caused due to bit change. And it always keeps the receiver in the error free version. In this paper we have tried to illustrate all possible error detection, correction controlling mechanisms.

ACKNOWLEDGMENT

I cannot express enough thanks to my college *Sai Vidya Institute of Technology* (SVIT) for their continued support and encouragement and also I offer my sincere appreciation for the learning opportunities provided by *Visveshwaraya Technological University (VTU)*.

REFERENCES

1. Shobha, B. N., & Kavya, B. S. (2015). <https://www.ijser.org/researchpaper/SURVEY-ON-ERROR-DETECTION-AND-CORRECTION-SCHEMES-FOR-MEMORY-APPLICATIONS>. *International Journal of Scientific & Engineering Research*, 6(2). Retrieved from <https://www.ijser.org/researchpaper/SURVEY-ON-ERROR-DETECTION-AND-CORRECTION-SCHEMES-FOR-MEMORY-APPLICATIONS.pdf>
2. Subiya Yaseen , Hussain Ahmed, & Kehkeshan Jallal , S. (2013). ASM Based Parity Detection Algorithm for Communication and Networking. *International Journal of Engineering Science Invention*, 2(4). Retrieved from [http://ijesi.org/papers/Vol%202\(4\)/Version-4/A240106.pdf](http://ijesi.org/papers/Vol%202(4)/Version-4/A240106.pdf)
3. Ashenden, P. J. (2008). Binary Coding: Bit Errors. In *Digital Design: An Embedded Systems Approach Using Verilog* (p. 58). Retrieved from https://textbooks.elsevier.com/web/product_details.aspx?isbn=9780123695277
4. Computer Network | Error Detection - javatpoint. (n.d.). Retrieved May 8, 2020, from <https://www.javatpoint.com/computer-network-error-detection>
5. (2019, July 31). Error Correction and Detection Codes | CRC, Hamming, Parity. Retrieved from <https://www.electronicshub.org/error-correction-and-detection-codes/>
6. Error Detection in Computer Networks. (2019, October 10). Retrieved May 8, 2020, from <https://www.geeksforgeeks.org/error-detection-in-computer-networks/>
7. 1&1 IONOS Inc. (2020, May 4). ASCII-Codes | Overview of all characters on the ASCII table. Retrieved May 8, 2020, from <https://www.ionos.com/digitalguide/server/known-how/ascii-codes-overview-of-all-characters-on-the-ascii-table/>
8. Chauhan, H. (2016, December 16). Hamming Encoder and Decoder. Retrieved May 8, 2020, from <https://www.slideshare.net/HetangChauhan/hamming-encoder-and-decoder>