Sant Dnyaneshwar Shikshan Sanstha's
Annasaheb Dange College of Engineering and Technology (ADCET), Ashta
An Autonomous Institute, Affiliated to Shivaji University, Kolhapur, Approved By AICTE, New Delhi & Govt. of Maharashtra, Accredited by NAAC 'A++' Grade, Bangalore
Department of CSE (IOT and Cyber Security including Blockchain Technology)

Class: SY B.Tech Sem VI

AY: 2023-2024

Course: Information Theory for Cyber Security (Laboratory)

Couse Code: 1ICPC210

Experiment No. 3

Title: Implement Hamming code and block code c/ c++ /python.

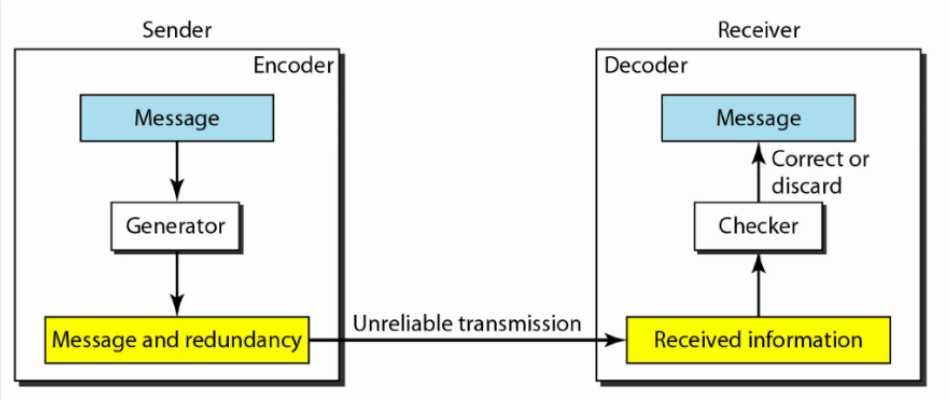
Objectives:

1. Understanding Error Detection and Correction
2. Understanding Block Code
3. Implementing Block Code
4. Analyzing Performance of Hamming code
5. Comparative Analysis of different types of block codes.

Block Code:

Block coding is a technique of encoding data into a specific format. It is mainly used to detect and correct errors occurred in the information during transmission and storage. This is done by adding a block code of redundant information to the main data.

Block coding is mainly employed to create a robust method of data transmission and storage. In the block coding, data is encoded by splitting it into multiple blocks of a fixed size and applying encoding techniques to each of these blocks separately.



In block coding, the input data is taken and transformed into a longer block of encoded data by adding some redundant data to it. This addition redundant data helps to detect and correct errors that occur during transmission and storage.

Block coding method generally works on binary data which is represented in the form of 0s and 1s. To perform block coding, there are various types of techniques are available, such as parity check codes, Hamming codes, Reed-Solomon codes, BCH codes, etc. Where, the parity check codes is the simplest technique to perform block coding. However, this technique has some limitations, such as it can detect only single-bit errors. The other block coding technique are much advanced and can detect as well as correct the errors.

Advantages of Block Coding in Digital Electronics

Block coding offers several benefits in the field digital electronics. Some key advantages of block coding in digital electronics are listed below:

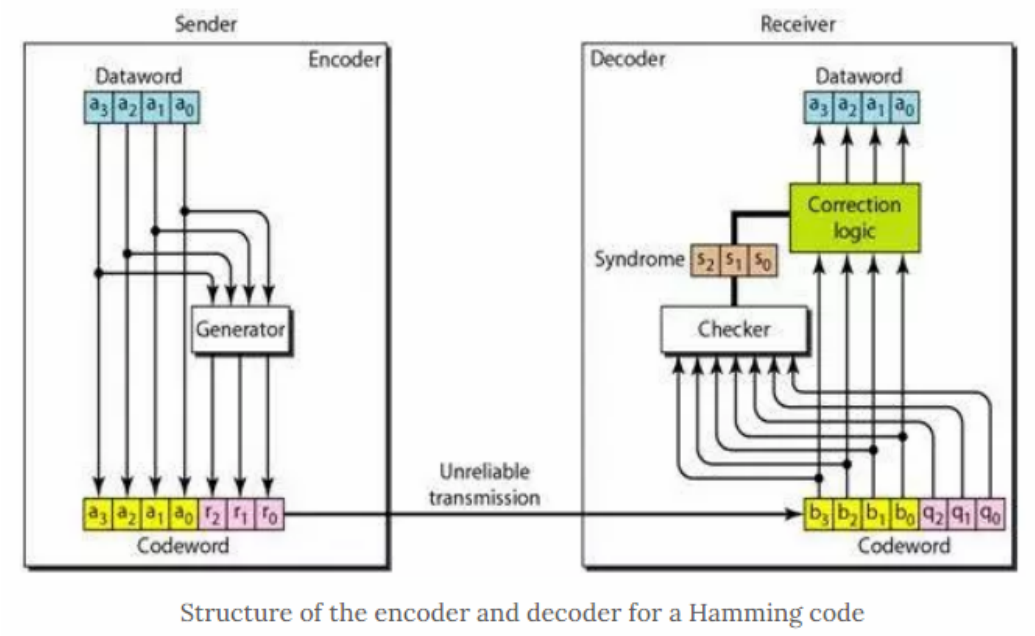
- Block coding improves the integrity of the received data by error detection and correction occurred during transmission and storage.
- Block coding improves overall reliability of the data transmission.
- Block coding increases immunity of the communication channel against noise and interference.
- Block coding allows for efficient utilization of storage space and channel bandwidth through the error correction.

Disadvantages of Block Coding in Digital Electronics

Apart from various advantages, block coding also has some disadvantages which are given below:

- Block coding increases redundancy in the data due to addition of extra bits for error correction.
- Block coding increases the overall data size of the block code, which consumes extra storage space or channel bandwidth.
- Block coding can reduce overall performance of the system, due to additional encoding and decoding processes.
- Block coding can cause delays in data transmission.
- Block coding involves the utilization of complex algorithms and hardware resources that introduce in its implementation.

Hamming Code:



Structure of the encoder and decoder for a Hamming code

Encoding a message by Hamming Code

The procedure used by the sender to encode the message encompasses the following steps −

Step 1 − Calculation of the number of redundant bits.

If the message contains mmnumber of data bits, rrnumber of redundant bits are added to it so that mr is able to indicate at least $(m + r + 1)$ different states. Here, $(m + r)$ indicates location of an error in each of $(m + r)$ bit positions and one additional state indicates no error. Since, rr bits can indicate $2^r r$ states, $2^r r$ must be at least equal to $(m + r + 1)$. Thus the following equation should hold $2^r \geq m+r+1$

Step 2 − Positioning the redundant bits.

The r redundant bits placed at bit positions of powers of 2, i.e. 1, 2, 4, 8, 16 etc. They are referred in the rest of this text as r_1 (at position 1), r_2 (at position 2), r_3 (at position 4), r_4 (at position 8) and so on.

Step 3 − Calculating the values of each redundant bit.

The redundant bits are parity bits. A parity bit is an extra bit that makes the number of 1s either even or odd. The two types of parity are −

- **Even Parity** − Here the total number of bits in the message is made even.
- **Odd Parity** − Here the total number of bits in the message is made odd.

Each redundant bit, r_i, is calculated as the parity, generally even parity, based upon its bit position. It covers all bit positions whose binary representation includes a 1 in the i^{th} position except the position of r_i. Thus −

- r_1 is the parity bit for all data bits in positions whose binary representation includes a 1 in the least significant position excluding 1 (3, 5, 7, 9, 11 and so on)
- r_2 is the parity bit for all data bits in positions whose binary representation includes a 1 in the position 2 from right except 2 (3, 6, 7, 10, 11 and so on)
- r_3 is the parity bit for all data bits in positions whose binary representation includes a 1 in the position 3 from right except 4 (5-7, 12-15, 20-23 and so on)

Decoding a message in Hamming Code

Once the receiver gets an incoming message, it performs recalculations to detect errors and correct them. The steps for recalculation are −

Step 1 − Calculation of the number of redundant bits

Using the same formula as in encoding, the number of redundant bits are ascertained.

$2^r \geq m + r + 1$ where *m* is the number of data bits and *r* is the number of redundant bits.

Step 2 − Positioning the redundant bits

The *r* redundant bits placed at bit positions of powers of 2, i.e. 1, 2, 4, 8, 16 etc.

Step 3 − Parity checking

Parity bits are calculated based upon the data bits and the redundant bits using the same rule as during generation of c_1,c_2 ,c_3 ,c_4 etc. Thus

c_1 = parity(1, 3, 5, 7, 9, 11 and so on)

c_2 = parity(2, 3, 6, 7, 10, 11 and so on)

c_3 = parity(4-7, 12-15, 20-23 and so on)

Step 4 − Error detection and correction

The decimal equivalent of the parity bits binary values is calculated. If it is 0, there is no error. Otherwise, the decimal value gives the bit position which has error. For example, if $c_1c_2c_3c_4$ = 1001, it implies that the data bit at position 9, decimal equivalent of 1001, has error. The bit is flipped to get the correct message.

Sample Code: (Block Code, Parity Check)

```
// C++ program to find parity
// of an integer
# include<bits/stdc++.h>
# define bool int
using namespace std;

// Function to get parity of number n. It returns 1
// if n has odd parity, and returns 0 if n has even
// parity
bool getParity(unsigned int n)
{
```

```
    bool parity = 0;
    while (n)
    {
        parity = !parity;
        n      = n & (n - 1);
    }
    return parity;
}

/* Driver program to test getParity() */
int main()
{
    unsigned int n = 7;
    cout<<"Parity of no "<<n<<" = "<<(getParity(n)? "odd": "even");

    getchar();
    return 0;
}
```

Sample Code: (Hamming Code)

```
#include<stdio.h>

void main() {
    int data[10];
    int dataatrec[10],c,c1,c2,c3,i;

    printf("Enter 4 bits of data one by one\n");
    scanf("%d",&data[0]);
    scanf("%d",&data[1]);
    scanf("%d",&data[2]);
    scanf("%d",&data[4]);

    //Calculation of even parity
    data[6]=data[0]^data[2]^data[4];
data[5]=data[0]^data[1]^data[4];
data[3]=data[0]^data[1]^data[2];

printf("\nEncoded data is\n");
for(i=0;i<7;i++)
        printf("%d",data[i]);
```

```
   printf("\n\nEnter received data bits one by one\n");
   for(i=0;i<7;i++)
      scanf("%d",&dataatrec[i]);

   c1=dataatrec[6]^dataatrec[4]^dataatrec[2]^dataatrec[0];
c2=dataatrec[5]^dataatrec[4]^dataatrec[1]^dataatrec[0];
c3=dataatrec[3]^dataatrec[2]^dataatrec[1]^dataatrec[0];
c=c3*4+c2*2+c1 ;

   if(c==0) {
printf("\nNo error while transmission of data\n");
   }
else {
printf("\nError on position %d",c);

printf("\nData sent : ");
       for(i=0;i<7;i++)
        printf("%d",data[i]);

printf("\nData received : ");
       for(i=0;i<7;i++)
        printf("%d",dataatrec[i]);
printf("\nCorrect message is\n");

//if errorneous bit is 0 we complement it else vice versa
if(dataatrec[7-c]==0)
dataatrec[7-c]=1;
       else
dataatrec[7-c]=0;
for (i=0;i<7;i++) {
printf("%d",dataatrec[i]);
}
}
}
```