Name :- Rushikesh Karbhari Palve
Roll No. :- 31258

: Date :	Page No :
/ /20	

Assignment No. 3

Title :- Hamming codes and CRC.

Problem statement :-

Write a program for error detection and correction for 7/8 bits ASCII codes using Hamming codes ~~or~~ or CRC.

Objectives :-

To learn error detection and correction for 7/8 bits ASCII codes using Hamming codes or CRC.

Outcomes :-

After completing the assignment students will be able to write program for error detection and correction using Hamming codes and CRC.

Theory :-

Hamming Code :-

Hamming code is a set of error-correction codes that can be used to detect and correct the errors that can occur when the data is moved or stored from the sender to the receiver. It is technique developed by R.W. Hamming for error correction.

Redundant bits :-

Redundant bits are extra binary bits that are generated and added to the information-carrying bits of data transfer to ensure that no bits were lost during the data transfer. The number of redundant bits can be calculated using the following formula :

$2^r \geq m+r+1$

where,

r = redundant bits

m = data bits.

Suppose the number of data bits is 7, then the number of redundant bits can be calculated using :

$= 2^4 \geq 7+4+1$

Thus, the number of redundant bits = 4.

Parity bits :-

A parity bit is a bit appended to a data of binary bits to ensure that the total number of 1's in the data is even or odd. Parity bits are used for error detection. There are two types of parity bits :-

1.] Even parity bit :-

In the case of even parity, for a given set of bits, the number of 1's are counted. If that count is odd, the parity bit value is set to 1, making the total count of occurences of 1's an even number. If the total number of 1's in a given set of bits is already even, the parity bit's value is 0.

2.] Odd parity bit :-

In the case of odd parity, for a given set of bits, the number of 1's are counted. If that count is even, the parity bit value is set to 1, making the total count of occurences of 1's an odd number. If the total number of 1's in a given set of bits is already odd, the parity bit's value is 0.

General Algorithm of Hamming Code :-

The Hamming Code is simply the use of extra parity bits to allow the identification of an error.

1. Write the bit positions starting from 1 in binary form (1, 10, 11, 100, etc).

2. All the bit positions that are powers of 2 are marked as parity bits (1, 2, 4, 8, etc.)

3. All the other bit positions are marked as data bits.

4. Each data bit is included in a unique set of parity bits, as determined its bit position in binary form.

a) Parity bit 1 covers all the bit positions whose binary representation includes a 1 in the least significant position (1, 3, 5, 7, 9, 11, etc)

b) parity bit 2 covers all the bits positions whose binary representation includes a 1 in the second position from the least significant bit (2, 3, 6, 7, 10, 11, etc)

c) Parity bit 4 covers all the bits positions whose binary representation includes a 1 in the third position from the least significant bit (4-7, 12-15, 20-23, etc)

d) parity bit 8 covers all the bits positions whose binary representation includes a 1 in the fourth position from the least significant bit bits (8-15, 24-31, 40-47, etc).

e) In general, each parity bit covers all bits where the bitwise AND of the parity position and the bit position is non-zero.

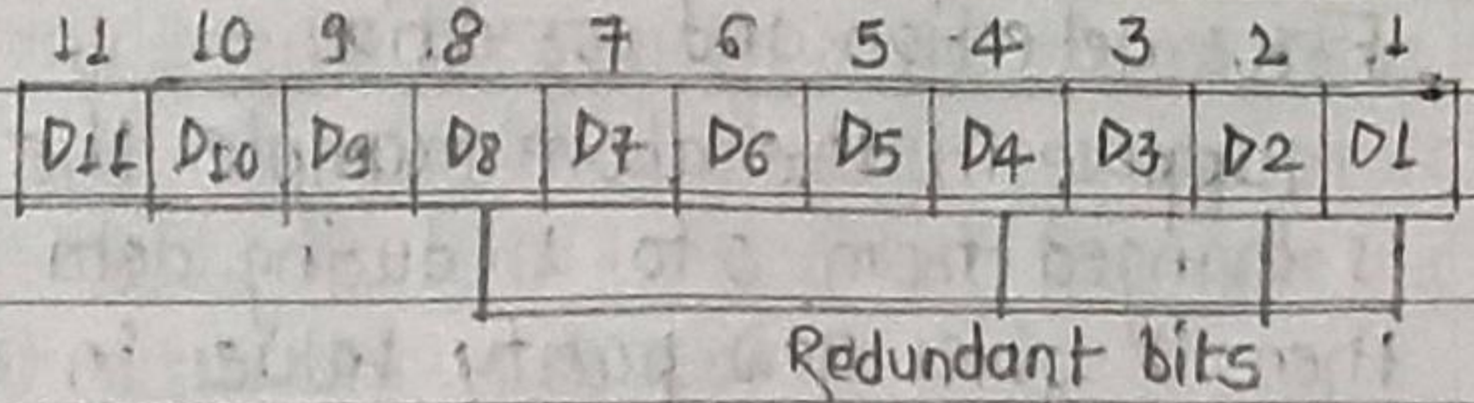
5. Since we check for even parity set a parity bit to 1 if the total number of ones in the positions it checks is odd.

6. Set a parity bit to 0 if the total number of ones in the positions it checks is even.

<u>Determining the position of redundant bits :-</u>

These redundancy bits are placed at the positions which correspond to the power of 2.

Roll NO: 31258



Example :-

Data to be transmitted : 1011001

No. of redundant bits : 4

11	10	9	8	7	6	5	4	3	2	1
1	0	1	R8	1	0	0	R4	1	R2	R1

Determining the parity bits - for even parity -

(i) R1 :

$D_3 D_5 D_7 D_9 D_{11} = 10111$

$\therefore R_1 = 0$

(ii) R_2 :

$D_3 D_6 D_7 D_{10} D_{11} = 10101$

$\therefore R_2 = 1$

(iii) R_4 :

$D_5 D_6 D_7 = 001$

$\therefore R_4 = 1$

(iv) R_8 :

$D_9 D_{10} D_{11} = 101$

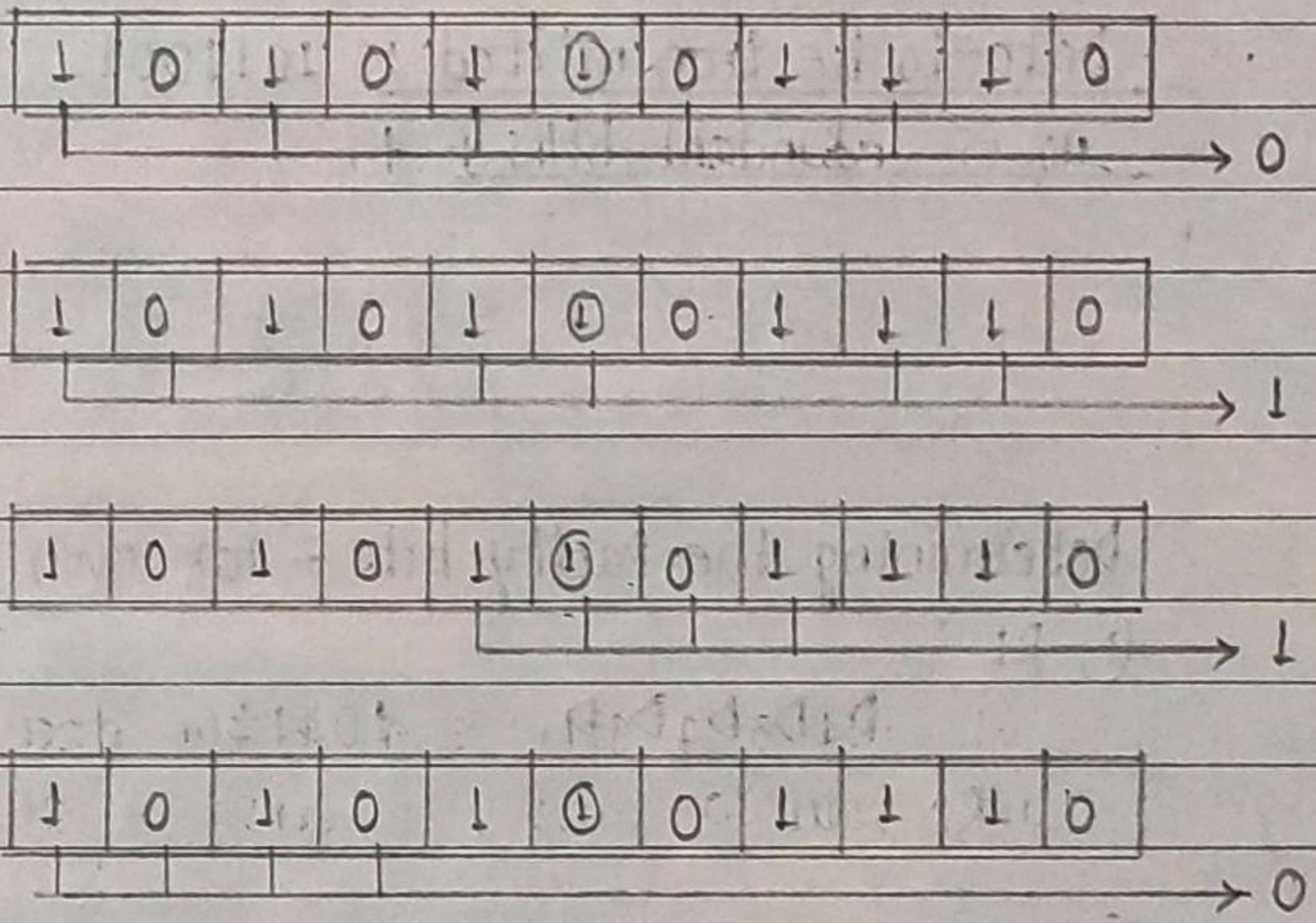
$\therefore R_8 = 0$

Thus data transferred is :-

11	10	9	8	7	6	5	4	3	2	1
1	0	1	0	1	0	0	1	1	1	0

Error detection and correction -

suppose in the above example the 6th bit is changed from 0 to 1 during data transmission then it gives new parity values in the binary number.



The bits give the binary number as 0110 whose decimal representation is 6. Thus the bit 6 contains an error. To correct the error the 6th bit is changed from 1 to 0.

Cyclic Redundancy check (CRC):-

CRC or Cyclic Redundancy check is a method of detecting accidental changes/errors in the communication channel.

CRC uses Generator Polynomial which is available on both sender and receiver side. An example generator polynomial is of the form like $x^3 + x + 1$. This generator polynomial represents key 1011. Another example is x^2+1 that represents key 101.

n : Number of bits in data to be sent from sender side.

k : Number of bits in the key obtained from generator polynomial.

Sender Side (Generation of Encoded Data from Data and Generator Polynomial (or key)):

1. The binary data is first augmented by adding (k-1) zero's in the end of the data.
2. Use modulo-2 binary division to divide binary data by the key and store remainder of division.
3. Append the remainder at the end of the data to form the encoded data and send the same.

Receiver Side (check if there are errors introduced in transmission)

Perform modulo-2 division again and ~~again~~ if the remainder is 0, then there are no errors.

Modulo-2 Division :-

The process of modulo-2 binary division is the same as the familiar division process we use for decimal numbers. Just that instead of subtraction, we use XOR here.

- In each step, a copy of the divisor (or data) is XORed with the k-bits of the dividend.
- The result of the XOR operation (remainder) is (n-1) bits, which is used for the next step after 1 extra bit is pulled down to make it n bits long.
- When there are no bits left to pull down, we have a result. The (n-1)-bit remainder which is appended at the sender side.

Example 1 (No error in transmission) :-

Data word to be sent - 100100
Key - 1101 (or generator polynomial x^3+x^2+1)

Sender side :-

```
           111101
    1101 ) 100100000
           1101
           ----
            1000
            1101
            ----
             1010
             1101
             ----
              1110
              1101
              ----
               0110
               0000
               ----
                1100
                1101
                ----
                 001
```

Therefore, the remainder is 001 and hence the encoded data sent is 100100001.

Receiver side :-

Code word received at the receiver side 100100001

```
          111101
  1101 | 100100001
         1101
          1000
          1101
          01010
           1101
            1110
            1101
            0110
             0000
              1101
              1101
              0000
```

Therefore, the remainder is all zeros. Hence, the data received has no error.

Example 2 (Error in transmission) :-

Data word to be sent - 100100
key - 1101

Sender side :-

```
          111101
     1101|100100000
          1101
           1000
           1101
            1010
            1101
             1110
             1101
              0110
              0000
               1100
               1101
                001
```

Therefore, the remainder is 001 and hence the code word sent is 100100001.

Receiver side :-

Let there be an error in transmission media code word received at the receiver side –

100000001

```
              111010
        1101 |100000001
              1101
               1010
               1101
                1110
                1101
                 0110
                 0000
                  1100
                  1101
                   0011
                   1101
                    011
```

since, the remainder is not all zeros, the error is detected at the receiver side.

Conclusion :-

Thus, we have studied the error detection and correction techniques for 7/8 bits ASCII codes using Hamming codes & CRC.

CODE :-

```
/*
 * Problem Statement :-
 *  Write a program for error detection and correction for 7/8 bits
ASCII codes using
 *  Hamming Codes or CRC.
 */


#include <bits/stdc++.h>
using namespace std;

class CRC
{
    private:
        string data, key;

    public:
        void input();
        string xor1(string, string);
        string mod2div(string, string);
        string encodedData();
        string remainder(string);
};


void CRC::input()
{
    cout<<"\n\t\t Enter Data (string) : ";
    cin>>data;

    cout<<"\n\t\t Enter Key (string) : ";
    cin>>key;
}

string CRC::xor1(string a, string b)
{
    string result = "";
    int n = b.length();

    for(int i = 1; i < n; i++)
    {
        if (a[i] == b[i])
            result += "0";
        else
            result += "1";
    }
    return result;
}

string CRC::mod2div(string dividend, string divisor)
{
    int pick = divisor.length();
    string tmp = dividend.substr(0, pick);

    int n = dividend.length();

    while (pick < n)
    {
        if (tmp[0] == '1')
            tmp = xor1(divisor, tmp) + dividend[pick];
        else
            tmp = xor1(string(pick, '0'), tmp) + dividend[pick];

        pick += 1;
    }

    if (tmp[0] == '1')
        tmp = xor1(divisor, tmp);
    else
        tmp = xor1(string(pick, '0'), tmp);

    return tmp;
}

string CRC::encodedData()
{
    int l_key = key.length();

    string appended_data = (data +string(l_key - 1, '0'));

    string remainder = mod2div(appended_data, key);

    string codeword = data + remainder;

    cout << "\n\t\t Remainder : "<< remainder << "\n";
    cout << "\n\t\t Encoded Data (Data + Remainder) :"<< codeword <<
 "\n";

    return codeword;
}

class HammingCode
{
    private:
        vector<int> msgBit;
        char p;
    public:
        void input();
        vector<int> generateHammingCode(int, int);
        void findHammingCode();
        void checkError(vector<int>&, int);
};

void HammingCode::input()
{
```

```cpp
    int m;
    cout<<"\n\t\t Enter number of bits in Message : ";
    cin>>m;
    msgBit.resize(m);
    cout<<"\n\t\t Enter Message (space separated) : ";
    for(int i=0; i<m; i++)
    {
        cin>>msgBit[i];
    }
    cout<<"\n\t\t Enter Parity (e/o) : ";
    cin>>p;
}

vector<int> HammingCode::generateHammingCode(int m, int r )
{
    vector<int> hammingCode(r + m);

    for (int i = 0; i < r; ++i)
    {
        hammingCode[pow(2, i) - 1] = -1;
    }

    int j = 0;

    for (int i = 0; i < (r + m); i++)
    {
        if (hammingCode[i] != -1)
        {
            hammingCode[i] = msgBit[j];
            j++;
        }
    }

    for (int i = 0; i < (r + m); i++)
    {
        if (hammingCode[i] != -1)
            continue;

        int x = log2(i + 1);
        int one_count = 0;

        for (int pos = i + 2; pos <= (r + m); ++pos)
        {
            if (pos & (1 << x))
            {
                if (hammingCode[pos - 1] == 1)
                {
                    one_count++;
                }
            }
        }

        if (one_count % 2 == 0)
        {
            if(p == 'e')
                hammingCode[i] = 0;
            else
                hammingCode[i] = 1;
        }
        else
        {
            if(p == 'e')
                hammingCode[i] = 1;
            else
                hammingCode[i] = 0;
        }
    }

    return hammingCode;
}

void HammingCode::checkError(vector<int>& receivedCode, int r)
{
    vector<int> pos;
    vector<int> parity;
    for(int i=0; i<r; i++)
    {
        pos.push_back(pow(2,i)-1);
    }
    for (unsigned int i = 0; i < pos.size(); i++)
    {
        int x = log2(pos[i] + 1);
        int one_count = 0;

        for (unsigned int j = pos[i]; j <= receivedCode.size(); j++)
        {
            if (j & (1 << x))
            {
                if (receivedCode[j - 1] == 1)
                {
                    one_count++;
                }
            }
        }

        if (one_count % 2 == 0)
        {
            if(p == 'e')
                parity.push_back(0);
            else
                parity.push_back(1);
        }
        else
        {
            if(p == 'e')
                parity.push_back(1);
            else
                parity.push_back(0);
        }
    }
    int cnt = 0;
    for(unsigned int i=0; i<parity.size(); i++)
    {
        cnt += parity[i]*pow(2,i);
    }
    if(cnt == 0)
    {
        cout<<"\n\t\t No Error...!!"<<endl;
    }
    else
    {
        cout<<"\n\t\t Error...!! Error present at position "<<cnt<<e
ndl;

        receivedCode[cnt-1] = receivedCode[cnt-1] == 1 ? 0:1;
        cout<<"\n\t\t Corrected Received Code : ";
        for(unsigned int i=0; i<receivedCode.size(); i++)
        {
            cout<<receivedCode[i]<<" ";
        }
    }
}

void HammingCode::findHammingCode()
{
    int m = msgBit.size();

    int r = 1;

    while (pow(2, r) < (m + r + 1))
    {
        r++;
    }

    vector<int> ans = generateHammingCode(m, r);

    cout << "\n\t\t Message bits are : ";
    for (unsigned int i = 0; i < msgBit.size(); i++)
        cout << msgBit[i] << " ";

    cout << "\n\n\t\t Receiver side Hamming code is : ";
    for (unsigned int i = 0; i < ans.size(); i++)
        cout << ans[i] << " ";

    cout<<"\n\n\t\t Enter Received Code of length("<<ans.size()<<")
: ";

    vector<int> receivedData(ans.size());
    for(unsigned int i=0; i<ans.size(); i++)
    {
        cin>>receivedData[i];
    }

    checkError(receivedData, r);
}

int main()
{
    int choice;

    while(true)
    {
        cout<<"\n === Main-
Menu ===  \n\t 1. CRC  \n\t 2. Hamming Code  \n\t 3. Exit \n";
        cout<<"\n\t Enter Your Choice : ";
        cin>>choice;

        if(choice == 1)
        {
            CRC c1;
            string str;

            c1.input();

            string encodedString = c1.encodedData();

            cout<<"\n\t\t Enter received data (string): ";
            cin>>str;

            bool flag = true;
            if(str.length() != encodedString.length()) flag = false;
            else if (flag)
            {
                for(unsigned int i=0; i<str.length(); i++)
                {
                    if(str[i] != encodedString[i])
                    {
                        flag = false;
                        break;
                    }
                }
            }

            if(flag)
            {
                cout<<"\n\t\t Received Data is valid."<<endl;
            }
            else
            {
                cout<<"\n\t\t Error!! Received Data is Invalid."<<en
dl;
            }
        }
        else if(choice == 2)
        {
            HammingCode h1;
            h1.input();
            h1.findHammingCode();
        }
        else if(choice == 3)
        {
            cout<<"\n\n\t\t\t === Thank You ===";
            exit(0);
        }
        else
        {
            cout<<"\n\t Enter correct choice...!!"<<endl;
        }
    }

    return 0;
}
```

OUTPUT :-

```
 === Main-Menu ===
    1. CRC
    2. Hamming Code
    3. Exit

    Enter Your Choice : 1

       Enter Data (string) : 100100

       Enter Key (string) : 1101

       Remainder : 001

       Encoded Data (Data + Remainder) :100100001

       Enter received data (string): 100100001

       Received Data is valid.

 === Main-Menu ===
    1. CRC
    2. Hamming Code
    3. Exit

    Enter Your Choice : 1
```

```
      Enter Data (string) : 100100

      Enter Key (string) : 1101

      Remainder : 001

      Encoded Data (Data + Remainder) :100100001

      Enter received data (string): 100000001

      Error!! Received Data is Invalid.

=== Main-Menu ===
   1. CRC
   2. Hamming Code
   3. Exit

   Enter Your Choice : 2

      Enter number of bits in Message : 7

      Enter Message (space separated) : 1 0 1 1 0 0 1

      Enter Parity (e/o) : e

      Message bits are : 1 0 1 1 0 0 1

      Receiver side Hamming code is : 1 0 1 0 0 1 1 1 0 0 1

      Enter Received Code of length(11) : 1 0 1 0 0 1 1 1 0 0 1

      No Error...!!

=== Main-Menu ===
   1. CRC
   2. Hamming Code
   3. Exit

   Enter Your Choice : 2

      Enter number of bits in Message : 7

      Enter Message (space separated) : 1 0 1 1 0 0 1

      Enter Parity (e/o) : e

      Message bits are : 1 0 1 1 0 0 1

      Receiver side Hamming code is : 1 0 1 0 0 1 1 1 0 0 1

      Enter Received Code of length(11) : 1 0 1 0 0 0 1 1 0 0 1

      Error...!! Error present at position 6
      Corrected Received Code : 1 0 1 0 0 1 1 1 0 0 1

=== Main-Menu ===
   1. CRC
   2. Hamming Code
   3. Exit

   Enter Your Choice : 3


         === Thank You ===
```