Practical 3 – Computer Networks Lab

Name: Neeraj Belsare

Roll No.: 79

Batch: A4

PRN: 202101040133

Title:

Detect and correct single bit.

Aim:

Write a program to detect and correct single bit error using

1. Parity Check
2. Hamming Code
3. Cyclic Redundancy Check

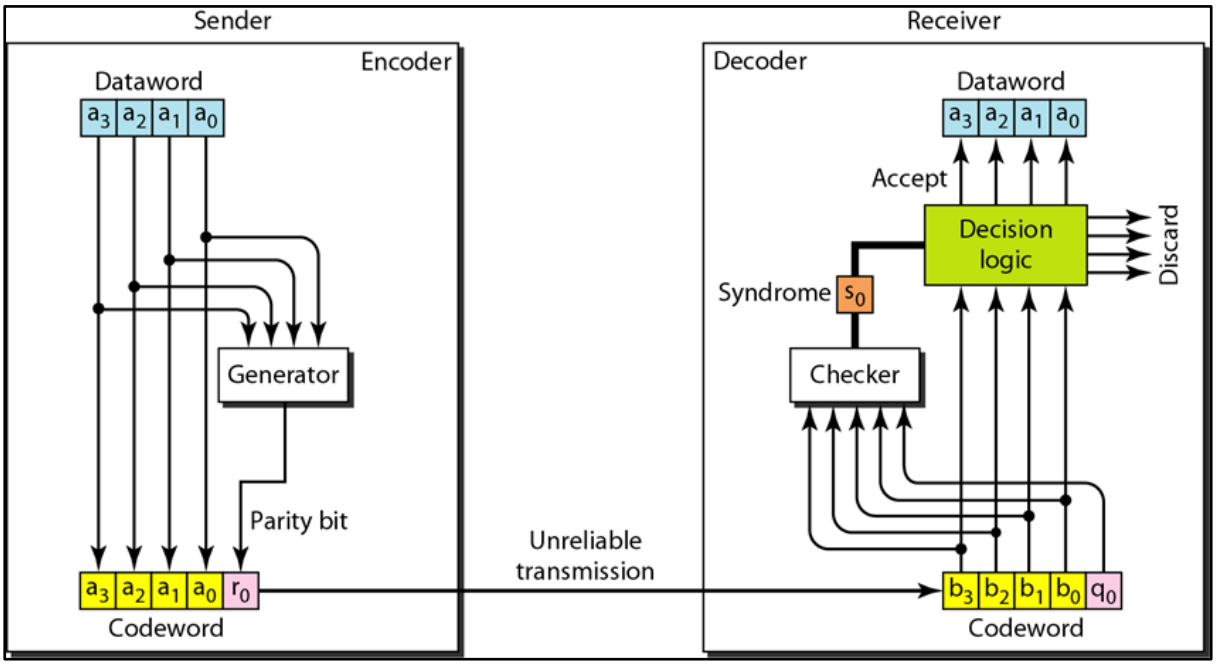
Theory:

Parity check

Parity check is a simple error detection method that uses an extra bit, called the parity bit, to detect errors in a block of data. The parity bit is set to 1 or 0 to make the total number of 1s in the block of data even or odd, depending on the type of parity check being used (even or odd parity).

A simple parity-check code is a single-bit error-detecting code in which n = k + 1 with $d_{min} = 2$. It cannot correct any error. A simple parity-check code can detect odd number of errors.

The structure of the encoder and decoder for simple parity-check code



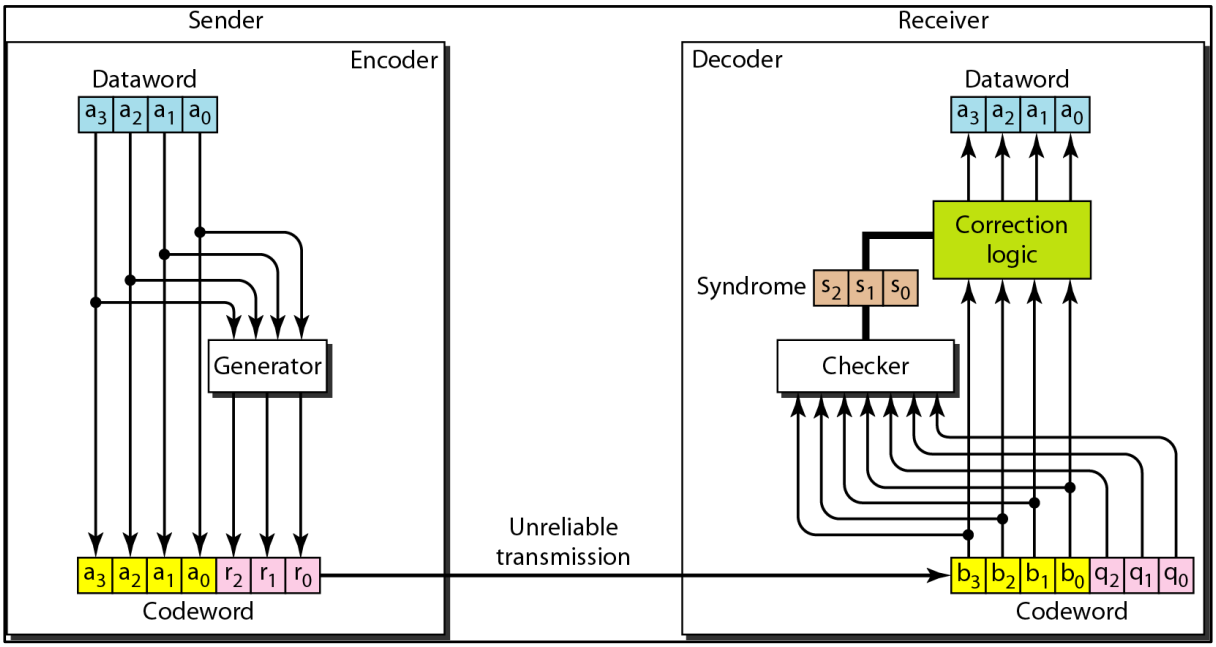
Hamming Code:

Hamming code is a more sophisticated error detection and correction code that uses multiple parity bits to detect and correct errors in a block of data. Hamming codes are more complex than parity checks, but they offer better error detection and correction capabilities.

Hamming codes work by dividing the block of data into two parts: the data bits and the parity bits. The parity bits are calculated using a complex mathematical formula, and they are added to the block of data before it is transmitted.

The receiver then calculates the parity bits for the received block of data and compares them to the received parity bits. If the two sets of parity bits are the same, then the block of data is assumed to be error-free. Otherwise, an error is detected, and the receiver can use the parity bits to correct the error.

The structure of the encoder and decoder for a Hamming code



Cyclic Redundancy Check:

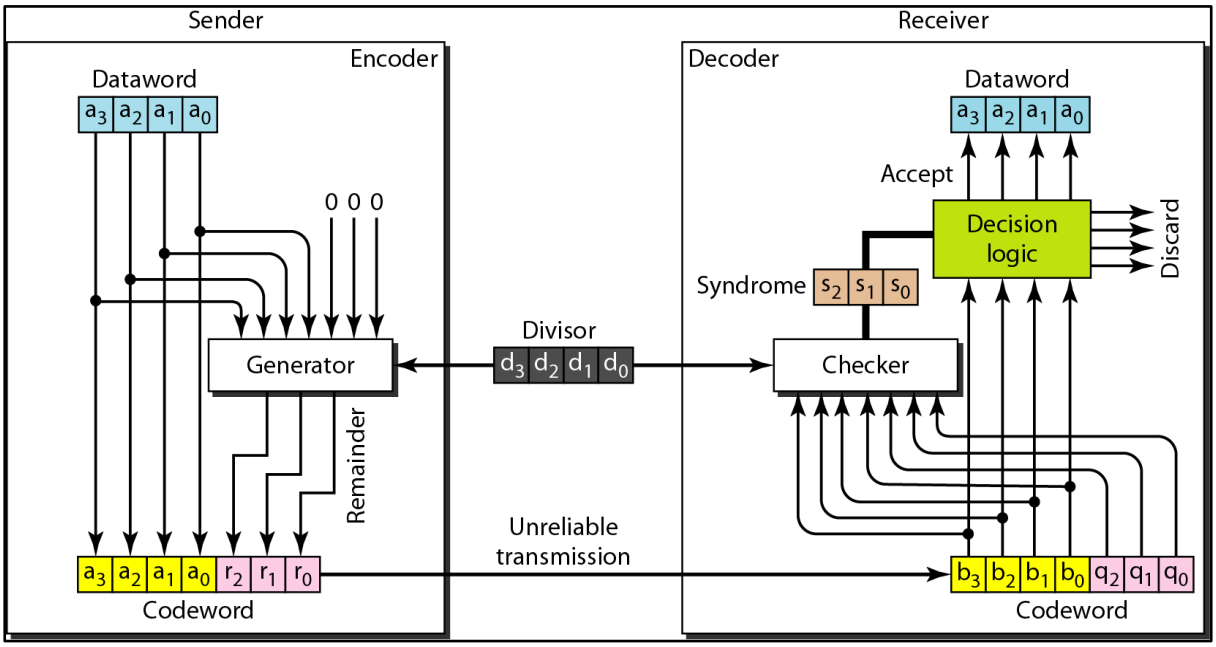
CRC is a powerful error detection code that is widely used in data communication systems. CRC works by appending a sequence of check bits to the end of a block of data. The check bits are calculated using a polynomial division operation. The receiver then divides the incoming block of data by the same polynomial. If there is no remainder, the block of data is assumed to be error-free. Otherwise, an error is detected.

In the encoder, the dataword has k bits; the codeword has n bits. We add n−k zeros to the right side of each dataword. The result is then fed into the generator. The generator divides the augmented dataword by a divisor of n–k + 1 bits (modulo-2 division). The quotient of the division is discarded; the remainder (r2r1r0) is added to the dataword to create the codeword.

The decoder receives the codeword (possibly corrupted in transition). It is fed to the checker, which is a replica of the generator. The remainder produced is a syndrome of n − k bits, which is fed to the decision logic analyzer.

If the syndrome bits are all 0s, the 4 leftmost bits of the codeword are accepted as the dataword (interpreted as no error); otherwise, the 4 bits are discarded (error).

The structure of the encoder and decoder for a CRC



Procedure/code:

1. Parity Check

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string data;
    char parityChoice;

    cout << "Enter binary data: ";
    cin >> data;

    cout << "Choose parity type (O for Odd Parity, E for Even Parity): ";
    cin >> parityChoice;
```

```
    // Count the number of 1s in the input data
    int onesCount = 0;
    for (char bit : data) {
        if (bit == '1') {
            onesCount++;
        }
    }

    // Add the appropriate parity bit (0 or 1) based on user choice
    char parityBit;
    if ((parityChoice == 'O' || parityChoice == 'o') && onesCount % 2 == 0) {
        parityBit = '1'; // Add 1 for Odd Parity
    } else if ((parityChoice == 'E' || parityChoice == 'e') && onesCount % 2 != 0)
{
        parityBit = '1'; // Add 1 for Even Parity
    } else {
        parityBit = '0'; // Add 0 for no parity or correct parity
    }

    // Display the data with the parity bit
    string dataWithParity = data + parityBit;
    cout << "Data with parity bit: " << dataWithParity << endl;

    // Simulate an error by flipping one bit
    // For demonstration purposes
    if (dataWithParity.size() >= 2) {
```

```
        dataWithParity[dataWithParity.size() - 2] =
(dataWithParity[dataWithParity.size() - 2] == '0') ? '1' : '0';
        cout << "Simulated error in received data: " << dataWithParity << endl;
    }

    // Check for errors by counting the number of 1s in the received data
    int receivedOnesCount = 0;
    for (char bit : dataWithParity) {
        if (bit == '1') {
            receivedOnesCount++;
        }
    }

    // If the total number of 1s matches the chosen parity, no error, otherwise,
there is an error
    if ((parityChoice == 'O' || parityChoice == 'o') && receivedOnesCount % 2 !=
0) {
        cout << "No error detected. Data received successfully with Odd Parity."
<< endl;
    } else if ((parityChoice == 'E' || parityChoice == 'e') && receivedOnesCount
% 2 == 0) {
        cout << "No error detected. Data received successfully with Even Parity."
<< endl;
    } else {
        cout << "Error detected in received data." << endl;
    }

    return 0;
}
```

2. Hamming Code

```
#include <iostream>
#include <cmath>
#include <string>
#include <unistd.h>

using namespace std;
class Hamming{

    public:
        string message;
        int codeword[100], temp[100];
        int n, check;
        char parity;

        Hamming(){
            parity = 'E';
            message = "";
            n = check = 0;

            for(int i = 0; i < 100; i++){
                temp[i] = codeword[i] = 0;
            }
        }

        int findr(){
            for(int i = 1 ;; i++){
```

```
            if((n + i + 1) <= pow(2, i))
                return i;
        }
    }
};

class Transmitter : public Hamming{

  public:

    void generate(){
        do{
            cout << "Enter the message to be transmitted in binary (LSB to
MSB): ";
            cin >> message;
        }while(message.find_first_not_of("01") != string::npos);

        n = message.size();
        cout << "Enter the Parity Type - Odd(O) / Even(E): ";
        cin >> parity;

        for(unsigned int i = 0; i < message.size(); i++){
            if(message[i] == '1'){
                temp[i+1] = 1;
            }
            else{
                temp[i+1] = 0;
            }
```

```
        }
        computeCode();
    }

    void computeCode(){
        check = findr();
        cout << "\nThe number of parity bits are: " << check << endl;
        cout << "The total number of bits to be transmitted are: " << (n+check)
<< endl << endl;

        for(int i = (n+check), j = n; i > 0; i--){
            if((i & (i - 1)) != 0){
                codeword[i] = temp[j--];
            }
            else{
                codeword[i] = setParity(i);
            }
        }

        cout << "PARITY BITS: \n";

        for(int i = 0; i < check; i++){
            cout << "P" << pow(2, i) << " : " << codeword[(int)pow(2, i)] <<
endl;
        }

        cout << "\nHamming Encoded Output: " << endl;
```

```
    for(int i = 1; i <= (n+check); i++){
        cout << codeword[i];
    }

    cout << endl;
}

int setParity(int x){
    bool flag = true;
    int bit;

    if(x == 1){
        bit = codeword[x+2];
        for(int j = (x + 3); j <= (n+check); j++){
            if(j % 2){
                bit ^= codeword[j];
            }
        }
    }

    else{
        bit = codeword[x+1];
        for(int i = x; i <= (n + check); i++){
            if(flag){
                if((i == x) || (i == (x+1))){
                    bit = codeword[x+1];
                }
```

```
                else{
                    bit ^= codeword[i];
                }
            }
            if(((i+1) % x) == 0){
                flag = !flag;
            }
        }
    }

    if(parity == 'O' || parity == 'o'){
        return (!(bit));
    }
    else{
        return (bit);
    }
  }
};

class Receiver : public Hamming{

  public:

    void correct(){
      do{
        cout << "Enter the Received Encoded Data (LSB to MSB): ";
        cin >> message;
```

```
    }while(message.find_first_not_of("01") != string::npos);

    n = message.size();
    cout << "Enter the Parity Type - Odd(O) / Even(E): ";
    cin >> parity;

    for(unsigned int i = 0; i < message.size(); i++){
        if(message[i] == '1'){
            codeword[i+1] = 1;
        }
        else{
            codeword[i+1] = 0;
        }
    }
    detect();
}

void detect(){
    int position = 0;
    check = findr();

    for(int i = 0; i < (n-check); i++){
        bool flag = true;
        int x = pow(2, i);
        int bit = codeword[x];

        if(x == 1){
```

```
        for(int j = (x+1); j <= (n+check); j++){
            if(j % 2){
                bit ^= codeword[j];
            }
        }
    }

    else{
        for(int k = (x+1); k <= (n+check); k++){
            if(flag){
                bit ^= codeword[k];
            }
            if((k+1) % x == 0){
                flag = !flag;
            }
        }
    }

    if((parity == 'E' || parity == 'e') && (bit == 1)){
        position += x;
    }

    if((parity == 'O' || parity == 'o') && (bit == 0)){
        position += x;
    }
}
```

```
        cout << "\nPARITY BITS: \n";

        for(int i = 0; i < check; i++){
            cout << "P" << pow(2,i) << " : " << codeword[(int)pow(2,i)] << endl;
        }

        cout << "\nReceived Data: " << endl;

        for(int i = 1; i <= (n); i++){
            cout << codeword[i];
        }

        cout << endl;

        if(position != 0){
            cout << "\nError detected in the transmitted data at position: " <<
position << endl;
            codeword[position] = !codeword[position];

            cout << "\nReceived data sequence after correction: " << endl;
            for(int i = 1; i <= (n); i++){
                cout << codeword[i];
            }

            cout << endl;
        }

        else{
```

```
            cout << "\nNo error in the transmitted data." << endl;
        }

        cout << "\nHamming Decoded Output: ";
        for(int i = 1; i <= (n); i++){
            if((i & (i - 1)) != 0){
                cout << codeword[i];
            }
        }
        cout << endl;
    }
};

int main(){

    string choice;

    Transmitter t;
    Receiver r;

    cout << "HAMMING ERROR DETECTION AND CORRECTION\n\n";

    cout << "\nAre you the Transmitter or the Receiver? ";
    cin >> choice;

    if ((choice == "transmitter")||(choice == "Transmitter")||(choice ==
"t")||(choice == "T")){
        cout << endl;
```

```
    t.generate();
  }

  if ((choice == "receiver")||(choice == "Receiver")||(choice == "r")||(choice == "R")){
    cout << endl;
    r.correct();
  }

  return 0;
}
```

3. Cyclic Redundancy Check

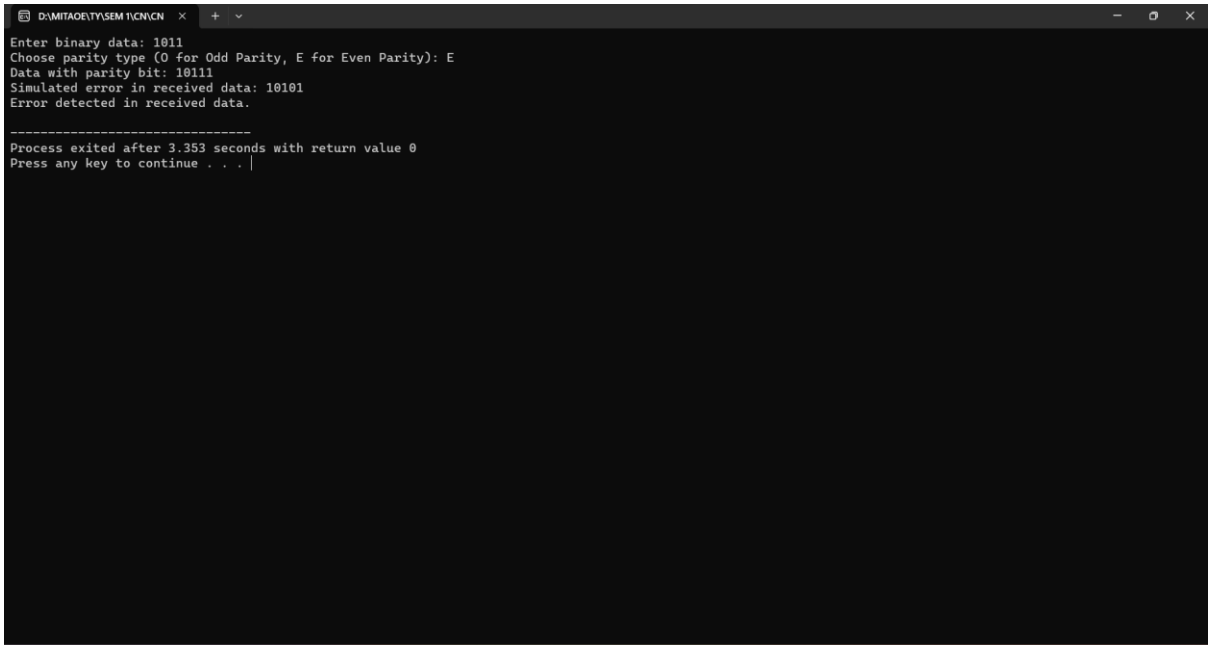
```
#include<iostream>

using namespace std;

string xorfun( string encoded, string crc) {
    //Bitwise XOR operation
    int crclen = crc.length();

    for ( int i = 0 ; i <= (encoded.length() - crclen) ; ) {
        for( int j=0 ; j < crclen ; j++) {
            encoded[i+j] = encoded[i+j] == crc[j] ? '0' : '1' ;
        }
        for( ; i< encoded.length() && encoded[i] != '1' ; i++) ;

    }
```

```
    return encoded;
}

int main() {
    string data, crc, encoded = "";
    cout << "SENDER SIDE" << endl;
    cout << "Enter the data: " << endl;
    cin>>data;

    cout<<"Enter the generator: "<<endl;
    cin>>crc;

    encoded += data;

    int datalen = data.length();
    int crclen = crc.length();

    for(int i=1 ; i <= (crclen - 1) ; i++)
        encoded += '0';

    encoded = xorfun(encoded, crc);

    cout<<"The checkbits generated are: ";
    cout<<encoded.substr(encoded.length() - crclen + 1)<<endl<<endl;
    cout<<"The message to be transmitted is: ";
    cout<<data + encoded.substr(encoded.length() - crclen + 1);
```

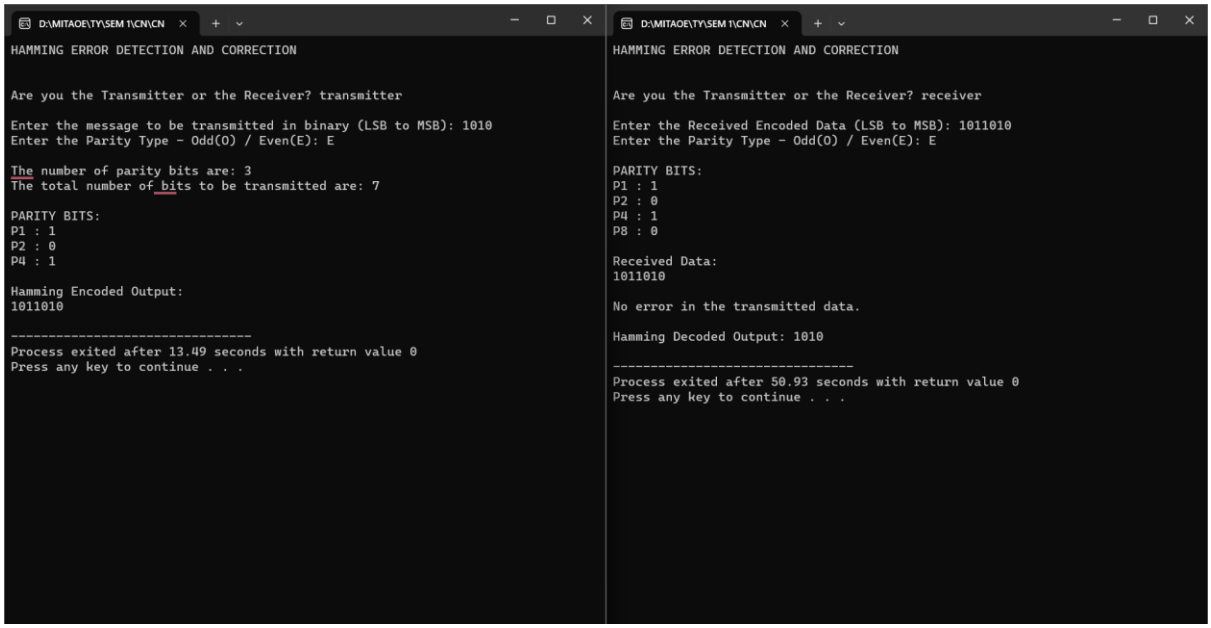
```
	cout<<endl<<"RECEIVER SIDE"<<endl;




	cout<<"Enter the message received: "<<endl;
	string msg;
	cin>>msg;


	msg = xorfun( msg, crc);
				//bitwise xor is performed between received bits and
the generator crc bits


	for( char i : msg.substr(msg.length() - crclen + 1))
	//after performing xor , if the last few bits are zero then there's no error in
transmission
		if( i != '0' ) {
			cout<<"Error in communication."<<endl;
			//if bits not zero ; ERROR IN TRANSMISSION
			return 0;
		}


	cout<<"No Error in transmission!"<<endl;
	return 0;
}
```

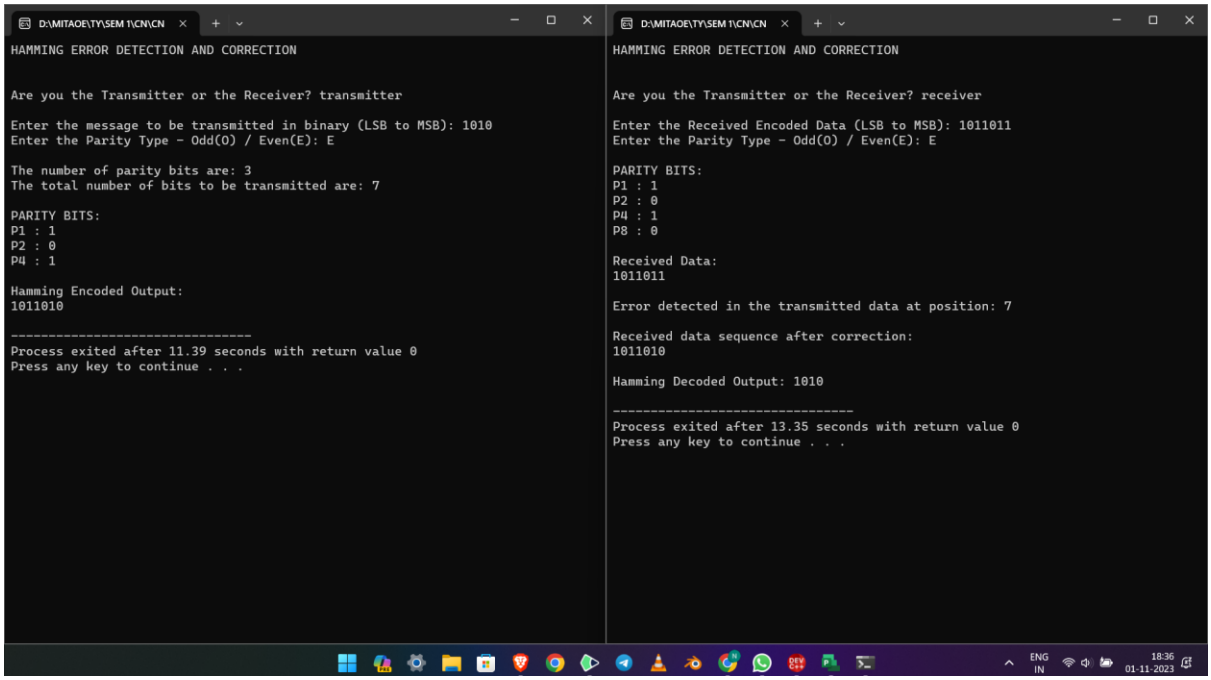
Output:

1. Parity Check

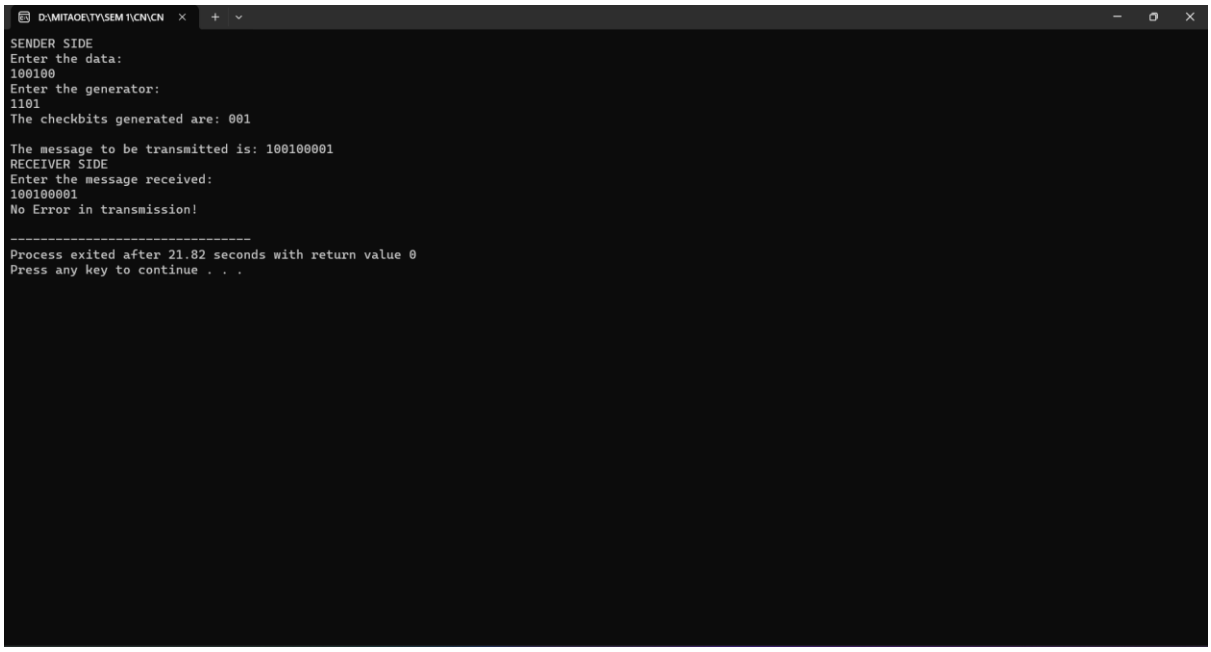


2. Hamming Code





3. Cyclic Redundancy Check



Conclusion:

In this practical, important error detection methods: Hamming code, CRC, and parity bit checks have been explored and implemented. Hamming code is great for fixing single-bit errors, ensuring data accuracy. CRC creates unique codes to detect different types of errors, enhancing reliability. Parity bit checks are simple and effective for spotting odd errors within data. By using these

techniques together, we can secure data transmission, ensuring that information sent over digital systems remains reliable and intact.