**22BCE0476**

**AMAN CHAUHAN**

****

BCSE308P - Computer Networks Lab

LAB REPORT

**EDC AND ECC**

**2. EDC and ECC**

1. Single Bit error detection

2. Burst error detection

3. Block coding detect

4. Bock coding detect and correct

5. Hamming distance

6. Minimum Hamming distance

7. Simple parity

8. Two-dimension parity

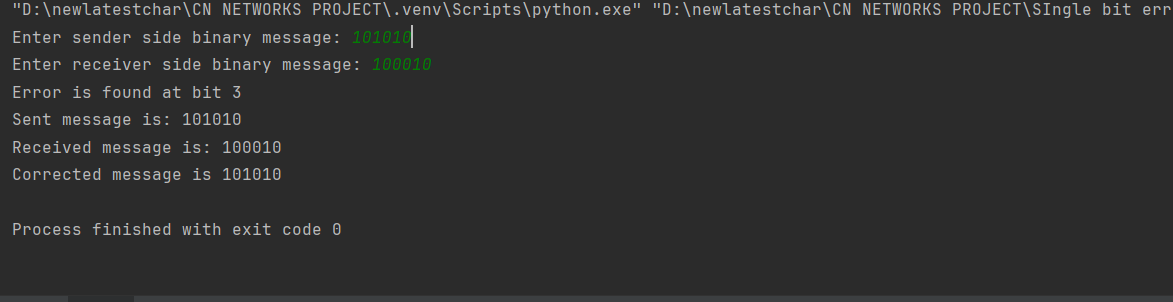
9. CRC

10. Check sum

**1. Single Bit error detection**

def single\_bit\_error\_detection(sender, receiver):  
 error\_found = False  
  
 # Checking each bit to identify the error position  
 for i in range(len(sender)):  
 if sender[i] != receiver[i]:  
 print(f"Error is found at bit {i + 1}")  
 error\_found = True  
 break  
  
 # Displaying the messages  
 print(f"Sent message is: {sender}")  
 print(f"Received message is: {receiver}")  
  
 # Output result  
 if not error\_found:  
 print("No error")  
 else:  
 print(f"Corrected message is {sender}")  
  
# Test case  
# Expected output:  
# Error is found at bit 3  
# Sent message is: 101010  
# Received message is: 100010  
# Corrected message is 101010  
  
sender\_message = input("Enter sender side binary message: ")  
receiver\_message = input("Enter receiver side binary message: ")  
single\_bit\_error\_detection(sender\_message, receiver\_message)

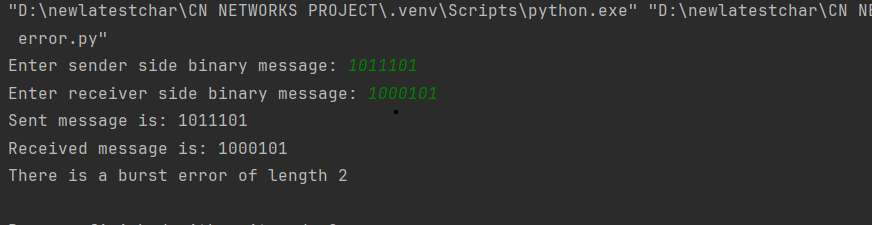
**OUTPUT:**

****

**2. Burst error detection**

def burst\_error\_detection(sender, receiver):  
 start, end = 0, 0  
 error\_found = False  
  
 # Identifying the start and end of the burst error  
 for i in range(len(sender)):  
 if sender[i] != receiver[i]:  
 if not error\_found:  
 start = i  
 error\_found = True  
 end = i  
  
 # Displaying the messages  
 print(f"Sent message is: {sender}")  
 print(f"Received message is: {receiver}")  
  
 # Output result  
 if not error\_found:  
 print("No error")  
 else:  
 error\_length = end - start + 1  
 if error\_length == 1:  
 print(f"There is an error of length {error\_length}")  
 else:  
 print(f"There is a burst error of length {error\_length}")  
  
# Test case  
# Expected output:  
# Sent message is: 1011101  
# Received message is: 1000101  
# There is a burst error of length 3  
  
sender\_message = input("Enter sender side binary message: ")  
receiver\_message = input("Enter receiver side binary message: ")  
burst\_error\_detection(sender\_message, receiver\_message)

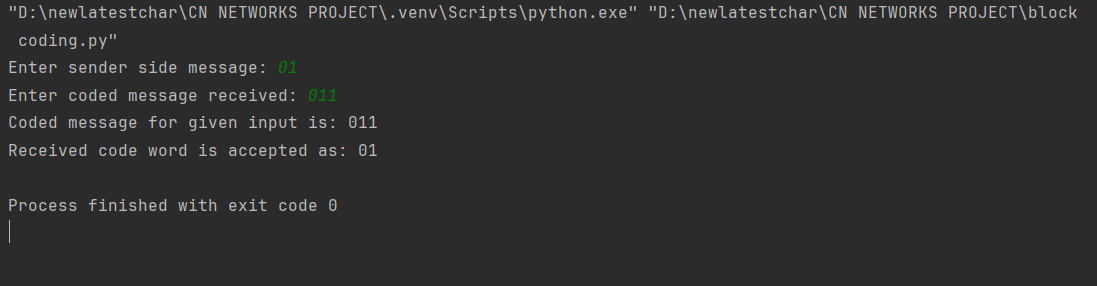
**OUTPUT:**

****

**3. Block coding detect**

# Define the block coding mappings  
encoding\_map = {  
 "00": "000",  
 "01": "011",  
 "10": "101",  
 "11": "110"  
}  
  
decoding\_map = {  
 "000": "00",  
 "011": "01",  
 "101": "10",  
 "110": "11"  
}  
  
  
def block\_coding\_detect(sender, receiver):  
 # Encode the sender message  
 encoded\_message = encoding\_map.get(sender, None)  
 print(f"Coded message for given input is: {encoded\_message}")  
  
 # Decode the received message and check validity  
 if receiver in decoding\_map:  
 decoded\_message = decoding\_map[receiver]  
 print(f"Received code word is accepted as: {decoded\_message}")  
 else:  
 print("Received code word is discarded.")  
  
  
# Test Case  
# Expected Output:  
# Coded message for given input is: 011  
# Received code word is accepted as: 01  
  
sender\_message = input("Enter sender side message: ")  
receiver\_message = input("Enter coded message received: ")  
block\_coding\_detect(sender\_message, receiver\_message)

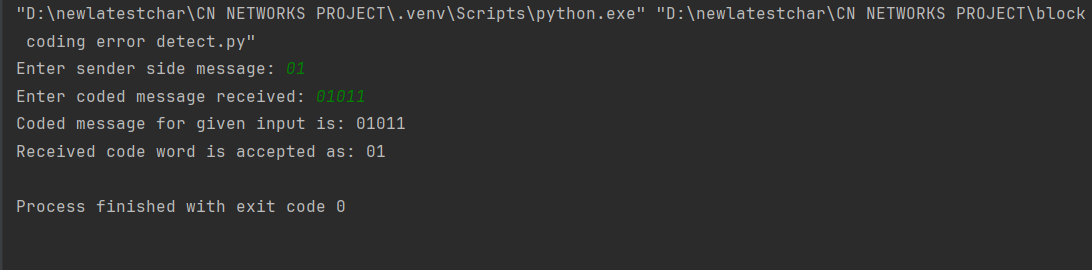
**OUTPUT:**

****

**4. Bock coding detect and correct**

# Define encoding and decoding maps for block coding  
encoding\_map = {  
 "00": "00000",  
 "01": "01011",  
 "10": "10101",  
 "11": "11110"  
}  
  
decoding\_map = {  
 "00000": "00",  
 "01011": "01",  
 "10101": "10",  
 "11110": "11"  
}  
  
  
def block\_coding\_detect\_correct(sender, receiver):  
 # Encode the sender message  
 encoded\_message = encoding\_map.get(sender, None)  
 print(f"Coded message for given input is: {encoded\_message}")  
  
 # Decode the received message and check validity  
 if receiver in decoding\_map:  
 decoded\_message = decoding\_map[receiver]  
 print(f"Received code word is accepted as: {decoded\_message}")  
 else:  
 print("Received code word is discarded.")  
  
  
# Test Case  
# Expected Output:  
# Coded message for given input is: 01011  
# Received code word is accepted as: 01  
  
sender\_message = input("Enter sender side message: ")  
receiver\_message = input("Enter coded message received: ")  
block\_coding\_detect\_correct(sender\_message, receiver\_message)

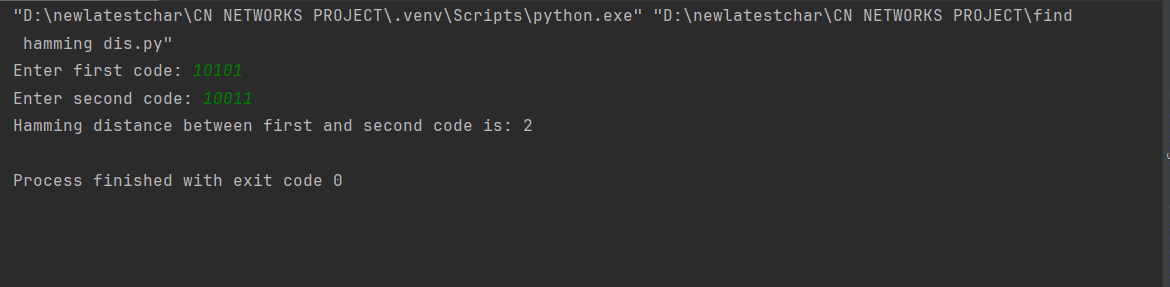
**OUTPUT:**

****

**5. Hamming distance**

def hamming\_distance(s1, s2):  
 # Calculate the Hamming distance  
 hamming\_dist = 0  
 for i in range(len(s1)):  
 if s1[i] != s2[i]:  
 hamming\_dist += 1  
 return hamming\_dist  
  
# Get input from user  
s1 = input("Enter first code: ")  
s2 = input("Enter second code: ")  
  
# Ensure the codes are of equal length  
if len(s1) == len(s2):  
 print("Hamming distance between first and second code is:", hamming\_distance(s1, s2))  
else:  
 print("Error: Codes must be of the same length.")  
  
# Test Case  
# Input:  
# s1 = "10101"  
# s2 = "10011"  
# Expected Output:  
# Hamming distance between first and second code is: 2

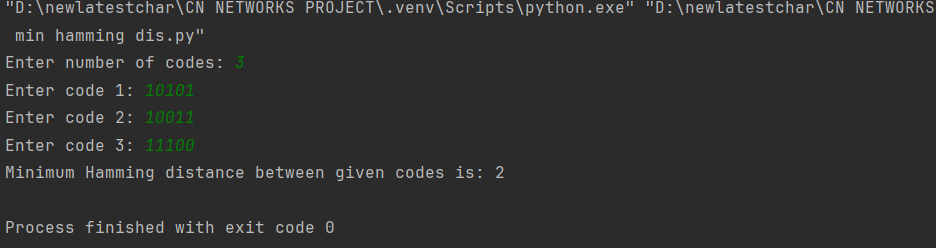
**OUTPUT:**

****

**6. Minimum Hamming distance**

def hamming\_distance(s1, s2):  
 # Calculate the Hamming distance between two strings  
 hamming\_dist = 0  
 for i in range(len(s1)):  
 if s1[i] != s2[i]:  
 hamming\_dist += 1  
 return hamming\_dist  
  
def find\_min\_hamming\_distance(codes):  
 # Initialize the minimum Hamming distance with a large number  
 hmin = float('inf')  
 n = len(codes)  
  
 # Compare each code with every other code  
 for i in range(n):  
 for j in range(i + 1, n):  
 dist = hamming\_distance(codes[i], codes[j])  
 if dist < hmin:  
 hmin = dist  
 return hmin  
  
# Get input from user  
n = int(input("Enter number of codes: "))  
codes = []  
  
# Read the codes  
for i in range(n):  
 code = input(f"Enter code {i+1}: ")  
 codes.append(code)  
  
# Ensure all codes have the same length  
if all(len(code) == len(codes[0]) for code in codes):  
 print("Minimum Hamming distance between given codes is:", find\_min\_hamming\_distance(codes))  
else:  
 print("Error: All codes must be of the same length.")  
  
# Test Case  
# Input:  
# Enter number of codes: 3  
# Enter code 1: 10101  
# Enter code 2: 10011  
# Enter code 3: 11100  
# Expected Output:  
# Minimum Hamming distance between given codes is: 2

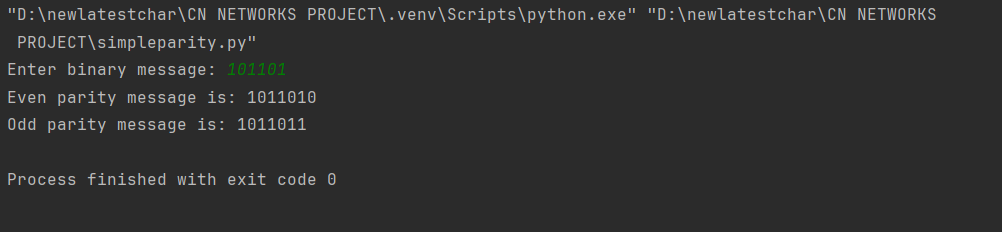
**OUTPUT:**

****

**7. Simple parity**

def simple\_parity(message):  
 # Count the number of 1's in the message  
 count\_of\_ones = message.count('1')  
  
 # Check for even or odd parity  
 if count\_of\_ones % 2 == 0:  
 even\_parity\_message = message + "0"  
 odd\_parity\_message = message + "1"  
 else:  
 even\_parity\_message = message + "1"  
 odd\_parity\_message = message + "0"  
  
 return even\_parity\_message, odd\_parity\_message  
  
  
# Get input from the user  
message = input("Enter binary message: ")  
  
# Calculate the parity  
even\_parity\_message, odd\_parity\_message = simple\_parity(message)  
  
# Display the results  
print("Even parity message is:", even\_parity\_message)  
print("Odd parity message is:", odd\_parity\_message)  
  
# Test Case  
# Input:  
# Enter binary message: 101101  
# Expected Output:  
# Even parity message is: 1011011  
# Odd parity message is: 1011010

**OUTPUT:**

****

**8. Two-dimension parity**

def two\_dimensional\_parity(rows, columns, binary\_matrix):  
 # Initialize row and column parity arrays  
 row\_parity = [0] \* (rows + 1)  
 col\_parity = [0] \* columns  
  
 # Calculate row parity  
 for i in range(rows):  
 count\_of\_ones = sum(1 for bit in binary\_matrix[i] if bit == '1')  
 row\_parity[i] = 1 if count\_of\_ones % 2 != 0 else 0  
  
 # Calculate column parity  
 for j in range(columns):  
 count\_of\_ones = sum(1 for i in range(rows) if binary\_matrix[i][j] == '1')  
 col\_parity[j] = 1 if count\_of\_ones % 2 != 0 else 0  
  
 # Calculate overall parity bit  
 total\_count\_of\_ones = sum(row\_parity[:rows]) + sum(col\_parity)  
 row\_parity[rows] = 1 if total\_count\_of\_ones % 2 != 0 else 0  
  
 # Display the original matrix with row parity bits  
 print("Two-dimensional even parity along with message is:")  
 for i in range(rows):  
 print(binary\_matrix[i] + str(row\_parity[i]))  
  
 # Display column parity bits and overall parity bit  
 print("".join(map(str, col\_parity)) + str(row\_parity[rows]))  
  
# Input the matrix size and binary values  
rows = int(input("Enter number of rows: "))  
columns = int(input("Enter number of columns: "))  
  
# Input each row of the matrix  
binary\_matrix = []  
for i in range(rows):  
 binary\_matrix.append(input(f"Enter row {i + 1}: "))  
  
# Calculate and display two-dimensional parity  
two\_dimensional\_parity(rows, columns, binary\_matrix)  
  
# Test Case  
# Input:  
# Enter number of rows: 3  
# Enter number of columns: 3  
# Enter row 1: 101  
# Enter row 2: 110  
# Enter row 3: 011  
# Expected Output:  
# Two-dimensional even parity along with message is:  
# 1011  
# 1101  
# 0110  
# 111

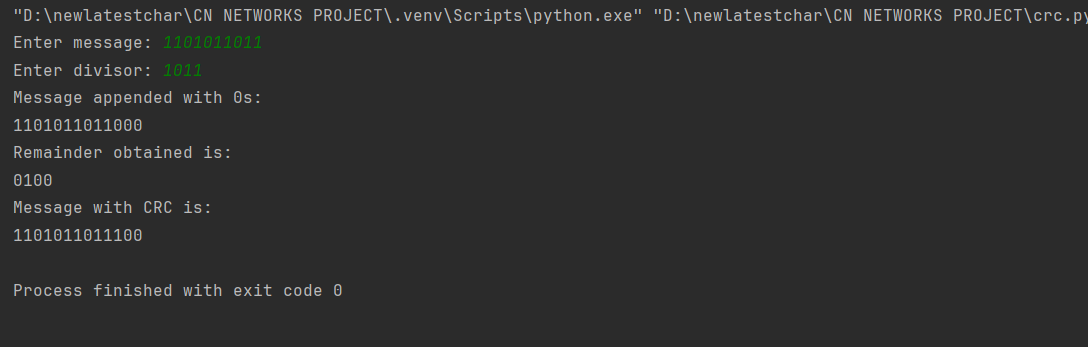
**OUTPUT:**

****

**9. CRC**

def crc():  
 # Get inputs  
 message = input("Enter message: ")  
 divisor = input("Enter divisor: ")  
  
 # Append zeros to the message based on the length of the divisor  
 omsg = message  
 n = len(message)  
 l = len(divisor)  
  
 message = message + '0' \* (l - 1)  
 print("Message appended with 0s: ")  
 print(message)  
  
 # Perform the division (XOR operation)  
 t = ['0'] \* l  
 for i in range(l):  
 t[i] = '1' if message[i] != divisor[i] else '0'  
  
 for j in range(l, len(message)):  
 for i in range(l - 1):  
 t[i] = t[i + 1]  
 t[l - 1] = message[j]  
  
 if t[0] == '0':  
 continue  
  
 for i in range(l):  
 t[i] = '1' if t[i] != divisor[i] else '0'  
  
 # Display remainder  
 print("Remainder obtained is: ")  
 print("".join(t))  
  
 # Append the CRC code to the original message  
 omsg = omsg + "".join(t[1:])  
 print("Message with CRC is: ")  
 print(omsg)  
  
  
# Call the crc function  
crc()

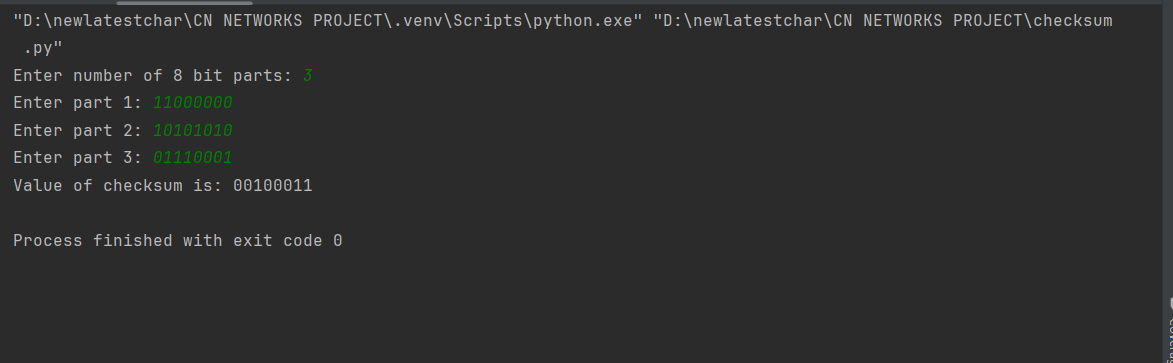
**OUTPUT:**

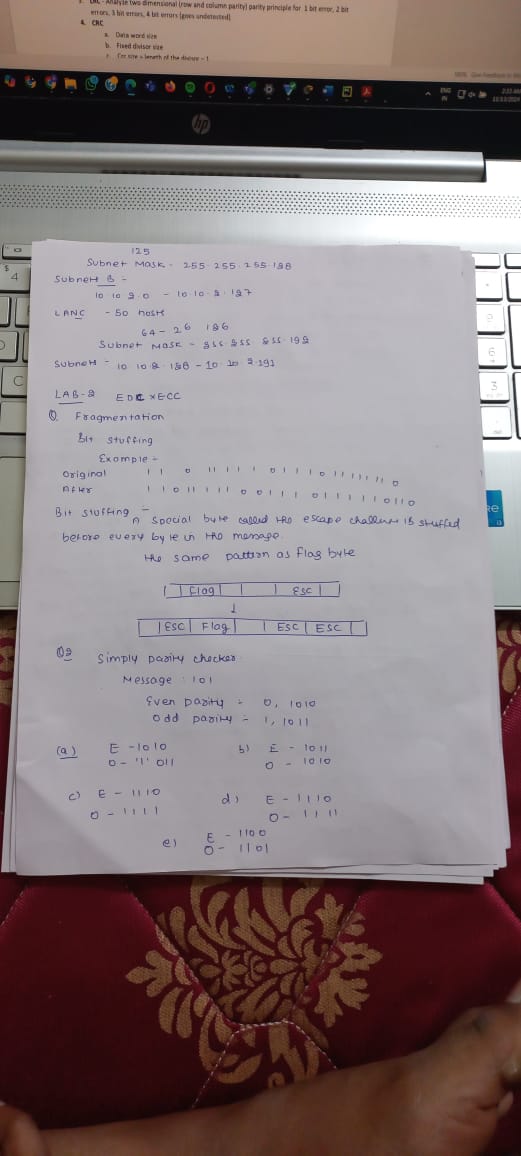
****

**10. Check sum**

def badd(s1, s2):  
 carry = '0'  
 result = ''  
 for j in range(7, -1, -1):  
 if s1[j] == '0' and s2[j] == '0':  
 if carry == '0':  
 bit = '0'  
 carry = '0'  
 else:  
 bit = '1'  
 carry = '0'  
 elif s1[j] == '0' and s2[j] == '1':  
 if carry == '0':  
 bit = '1'  
 carry = '0'  
 else:  
 bit = '0'  
 carry = '1'  
 elif s1[j] == '1' and s2[j] == '0':  
 if carry == '0':  
 bit = '1'  
 carry = '0'  
 else:  
 bit = '0'  
 carry = '1'  
 elif s1[j] == '1' and s2[j] == '1':  
 if carry == '0':  
 bit = '0'  
 carry = '1'  
 else:  
 bit = '1'  
 carry = '1'  
  
 result = bit + result  
  
 if carry == '1':  
 result = badd(result, "00000001")  
  
 return result  
  
  
def main():  
 n = int(input("Enter number of 8 bit parts: "))  
 arr = []  
 for i in range(n):  
 arr.append(input(f"Enter part {i + 1}: "))  
  
 checksum = arr[0]  
 for i in range(1, n):  
 checksum = badd(checksum, arr[i])  
  
 ccs = ''  
 for i in range(8):  
 if checksum[i] == '0':  
 ccs += '1'  
 else:  
 ccs += '0'  
  
 print(f"Value of checksum is: {ccs}")  
  
  
# Call the main function  
main()

**OUTPUT:**

****



A piece of paper with writing on it

Description automatically generated

