Computer Networks Lab
Assignment 1

Name : **Shivam Singh** Roll : **002010501040** Class : **BCSE III**

Group : **A2**

no_reply@example.com

Problem Statement:

Design and implement an error detection module which has four schemes namely LRC, VRC, Checksum and CRC.

Please note that you may need to use these schemes separately for other applications (assignments). You can write the program in any language. The Sender program should accept the name of a test file (contains a sequence of 0,1) from the command line. Then it will prepare the data frame (decide the size of the frame) from the input. Based on the schemes, codeword will be prepared. Sender will send the codeword to the Receiver. Receiver will extract the dataword from codeword and show if there is any error detected. Test the same program to produce a PASS/FAIL result for following cases (not limited to).

(a) Error is detected by all four schemes. Use a suitable CRC polynomial (list is given in the next page).

(b) Error is detected by checksum but not by CRC.

(c) Error is detected by VRC but not by CRC. [Note: Inject error in random positions in the input data frame. Write a separate method for that.]

Design:

A bit, on travel, is subjected to electromagnetic (optical) interference due to noise signals (light sources). Thus, the data transmitted may be prone to errors. In particular, a bit '0' (bit '1') sent by the sender may be delivered as a bit '1' (bit '0') at the receiver. This happens because the voltage present in noise signals either has a direct impact on the voltage present in the data signal or creates a distortion leading to mis-interpretation of bits while decoding the signals at the receiver. This calls for a study on error detection and error correction. The receiver must be intelligent enough to detect the error and ask the sender to transmit the data packet or must have the ability to detect and correct the errors. In this assignment, we shall discuss the following error detection techniques in detail.

1. Vertical Redundancy check

2. Longitudinal Redundancy check

3. Checksum

4. Cyclic Redundancy check.

To simulate noisy channels, I have injected errors by flipping the bits at certain bit positions of the data being transmitted.

I have implemented 3 programs to implement the error-detections scheme:

- sender.py : Sender program
    - ★ a. The input file is read, which contains a sequence of 0 and 1.
    - ★ b. The message sequence is divided into datawords on the basis of frame size taken as the input from the user.
    - ★ c. According to the four schemes namely VRC, LRC, Checksum and CRC, redundant bits/dataword are added along with the datawords to form codewords.
    - ★ d. The datawords and codewords which are to be sent are displayed.
    - ★ e. These encoded codewords are then sent to the receiver.

- receiver.py : Receiver program
  - ★ a. The codewords are received from the sender.
  - ★ b. The received codewords are then decoded according to the four schemes namely VRC, LRC, Checksum and CRC.
  - ★ c. The codewords and extracted datawords are displayed. If any error is detected, it is displayed.

- medium.py : Program to simulate noisy channels.
  - ★ a. Errors are injected at random/specific bit positions, as instructed by the user.

- helper.py : Helper program to provide utility functions.
  - ★ a. Contains utility functions like add(), xor(), divide() . It is imported into other programs to help with numeric calculations.

- main.py : Acts as an interface for all the programs.
  - ★ a. A file name is taken as input, which consists of the binary sequence and is treated as the input file. The classes Sender and Receiver in sender.py and receiver.py respectively are instantiated to send and receive the data read from the input file.
  - ★ b. The function invocations of Sender class stored in sender.py and Receiver class stored in receiver.py are done in this file to send and receive the data.
  - ★ c. The sent data are injected with errors using the above mentioned functions.
  - ★ d. For all the following three cases, three different functions have been created to execute a specific case at a moment. i. Error is detected by all four schemes. ii. Error is detected by checksum but not CRC. iii. Error is detected by VRC but not CRC.

Implementation :

Code snippet of sender.py:

```python
from unittest import result

from numpy import result_type

import helper


class Sender:

    #Initialize all the data members of class

    def __init__(self, size):

        self.codewords = []

        self.dataSize = size

        self.parity = ""

        self.checksum = ""


    #Function to generate codewordss to be sent

    def createData(self, filename, type, poly=""):

        fileinput = open(filename, "r")

        packet = fileinput.readline()

        fileinput.close()


#VRC

        if type == 1:
            self.VRC(packet)
#LRC
        elif type == 2:
            self.LRC(packet)
#checksum

        elif type == 3:
            self.checkSum(packet)


#CRC
        elif type == 4:
            self.CRC(packet,poly)


    #Function to display the sent codewordss
    def displayData(self, type):
        datawords = []
        for x in self.codewords:
            datawords.append(x[:self.dataSize])
        print("Datawords to be sent:")
        print(datawords)
        print("codewordss sent by sender:")
        print(self.codewords)
        if type == 2:
            print(self.parity, " - parity")
        elif type == 3:
            print(self.checksum, " - checksum")
        print("\n")

    #Helper function for VRC Even Parity generator
    def rowEvenParityGenerator(self):
        for i in range(len(self.codewords)):
            countOnes = 0
            for j in range(len(self.codewords[i])):
                if self.codewords[i][j] == '1':
                    countOnes += 1
            if countOnes%2==1:
                self.codewords[i] += '1'
            else:
                self.codewords[i] += '0'

    #Helper function for LRC Even Parity generator
    def columnEvenParityGenerator(self):
        i=0
        while i<self.dataSize:
            countOnes = 0
            j=0
            while j<len(self.codewords):
                if self.codewords[j][i] == '1':
                    countOnes += 1
                j+=1
            if countOnes%2==1:
                self.parity += '1'
            else:
                self.parity += '0'
            i+=1
    def calcheckSum(self):
        result = self.codewords[0]
        csum = ""

        for i in range(1,len(self.codewords)):
            result = helper.add(result, self.codewords[i])
            while len(result) > self.dataSize:
                t1 = result[:len(result)-self.dataSize]
                t2 = result[len(result)-self.dataSize:]
                result = helper.add(t1, t2)
        for i in range(len(result)):
                if result[i]=='0':
                    csum += '1'
                else:
                    csum += '0'
        return csum
    def VRC(self,packet):

        tempword=""

        for i in range(len(packet)):

                if i>0 and i%self.dataSize==0:

                    self.codewords.append(tempword)

                    tempword = ""

                tempword += packet[i]

        self.codewords.append(tempword)

        self.rowEvenParityGenerator()


    def LRC(self,packet):

        tempword=""

        for i in range(len(packet)):

                if i>0 and i%self.dataSize==0:

                    self.codewords.append(tempword)

                    tempword = ""

                tempword += packet[i]

        self.codewords.append(tempword)

        self.columnEvenParityGenerator()


    def checkSum(self,packet):

        tempword=""

        for i in range(len(packet)):

                if i>0 and i%self.dataSize==0:
                    self.codewords.append(tempword)
                    tempword = ""
                tempword += packet[i]
        self.codewords.append(tempword)
        self.checksum = self.calcheckSum()

    def CRC(self,packet,poly):
        tempword=""
        tempsize = self.dataSize #- (len(poly)-1)
        for i in range(len(packet)):
                if i>0 and i%tempsize==0:
                    tempword += '0'*(len(poly)-1)
                    remainder = helper.divide(tempword, poly)
                    remainder = remainder[len(remainder)-(len(poly)-1):]
                    tempword = tempword[:tempsize]
                    tempword += remainder
                    self.codewords.append(tempword)
                    tempword = ""
                tempword += packet[i]

        tempword += '0'*(len(poly)-1)
        remainder = helper.divide(tempword, poly)
        remainder = remainder[len(remainder)-(len(poly)-1):]
        tempword = tempword[:tempsize]
        tempword += remainder

        self.codewords.append(tempword)
```

Method descriptions of sender.py:

createData(self, filename, type, poly="") : This method is used to take input of sequence of 0,1 from a given file, accessing it using the filename passed as argument. Then the packet is divided into datawords of fixed size. According to the type (scheme) of error detection, namely VRC, LRC, Checksum, CRC, the datawords are appended with appropriate redundant bits.

displayData(self,type) : This method has been used for displaying the datawords and codewords which are to be sent.

rowEvenParityGenerator(self) : Generates the codewords by computing and adding the parity bit in accordance with the VRC scheme.

columnEvenParityGenerator(self) : Generates the codewords by computing and adding the parity bit in accordance with the LRC scheme.

calcheckSum( ): Calculates the checksum of the codewords according to module-2 arithmetic.

VRC(self,packet): Used to generate the codewords based on input data and calculate the parity bits for VRC .

LRC(self,packet): Used to generate the codewords based on input data and calculate the parity bits for LRC .

CRC(self,packet,poly): Takes as input the generator polynomial and generates the codewords and CRC based on input data and appends the CRC to the end of each codeword .

Code snippet of receiver.py:

```
import helper


class Receiver:

    #Initialize all the data members of class

    def __init__(self, s):

        self.codewords = s.codewords

        self.dataSize = s.dataSize

        self.sentparity = s.parity

        self.receivedparity = ""

        self.sentchecksum = s.checksum

        self.sum = ""

        self.addchecksum = ""

        self.complement = ""


    #Function to decode and check for error in codewords
```

```
def checkError(self, type, poly=""):

    if type == 1:

        for i in range(len(self.codewords)):

            countOnes = 0

            for j in range(len(self.codewords[i])):

                if self.codewords[i][j]=='1':

                    countOnes += 1

            if countOnes%2==0:

                print("Parity is even.", end=' ')

                print("NO ERROR DETECTED")

            else:

                print("Parity is odd.", end=' ')

                print("ERROR DETECTED")

    elif type == 2:

        error = False

        i=0

        while i<self.dataSize:

            countOnes = 0

            j=0

            while j<len(self.codewords):

                if self.codewords[j][i] == '1':

                    countOnes += 1

                j+=1

            if countOnes%2==1:
                ch = '1'
            else:
                ch = '0'
            self.receivedparity += ch
            if ch != self.sentparity[i]:
                error = True
            i+=1
        if error:
            print("Parity did not match.",end='  ')
            print("ERROR DETECTED")
        else:
            print("Parity matched.",end='  ')
            print("NO ERROR DETECTED")
    elif type == 3:
        self.sum = self.calculateSum()
        result = helper.add(self.sum, self.sentchecksum)
        while len(result) > self.dataSize:
            t1 = result[:len(result)-self.dataSize]
            t2 = result[len(result)-self.dataSize:]
            result = helper.add(t1, t2)
        self.addchecksum = result


        #finding complement and checking error
        error = False
        for ch in self.addchecksum:
            if ch == '0':
                self.complement += '1'
                error = True
            else:
                self.complement += '0'
        if error:
            print("Complement is not zero.",end=' ')
            print("ERROR DETECTED")
        else:
            print("Complement is zero.",end=' ')
            print("NO ERROR DETECTED")
    elif type == 4:
        for i in range(len(self.codewords)):
            remainder = helper.divide(self.codewords[i], poly)
            error = False
            for j in range(len(remainder)):
                if remainder[j] == '1':
                    error = True
            print("Remainder:",remainder,end=' ')
            if error:
                print("ERROR DETECTED")
            else:
                print("NO ERROR DETECTED")
    print()


#Function to display the received codeword
def displayData(self, type):
    print("codewords received by receiver:")
    print(self.codewords)
    if type == 2:
        print(self.receivedparity, " - parity")
    elif type == 3:
        print(self.sum, " - sum")
        print(self.addchecksum, " - sum + checksum")
        print(self.complement, " - complement")
    for i in range(len(self.codewords)):
        self.codewords[i] = self.codewords[i][:self.dataSize]
    print("Extracting data words from codewords:")
    print(self.codewords)
    print()



def calculateSum(self):
    result = self.codewords[0]
    for i in range(1,len(self.codewords)):
        result = helper.add(result, self.codewords[i])
        while len(result) > self.dataSize:
            t1 = result[:len(result)-self.dataSize]
            t2 = result[len(result)-self.dataSize:]
            result = helper.add(t1, t2)
    return result
```

Method descriptions of receiver.py :

checkError(self,type,poly="") : This method takes the type (scheme) of the error detection method as one of its arguments and the other is a polynomial (its error-detection scheme is CRC) . On the basis of the scheme the codewords received are checked for error. If there is an error, an appropriate message is displayed on the screen.

displayData( ): Shows the received codewords and the datawords after decoding the codewords.

calculateSum( ): Calculates the sum of codewords according to module-2 arithmetic.

Code snippet of medium.py:

```
import random
import sys


#function to inject errors in random positions
def injectRandomError(frames):
    for i in range(len(frames)):
        pos = random.randint(0, len(frames[i])-1)
        frames[i] = frames[i][:pos]+'1'+frames[i][pos+1:]
    return frames

#function to inject errors in specific positions
def injectAtPosError(frames, zeropos, onepos):
    for i in range(len(zeropos)):
        for j in range(len(zeropos[i])):
            pos = zeropos[i][j]
            frames[i] = frames[i][:pos]+'0'+frames[i][pos+1:]
    for i in range(len(onepos)):
        for j in range(len(onepos[i])):
            pos = onepos[i][j]
            frames[i] = frames[i][:pos]+'1'+frames[i][pos+1:]
    return frames
```

Method descriptions of medium.py :

injectRandomError(frames) : This method is used to inject errors in random positions of the codewords which are sent by the sender program. The randint() function of the random python module has been used for generating random positions within the size of the codewords.

injectAtPosError(frames,zeropos,onepos) : This method is used to inject errors in specific positions of the codewords. The positions in which the error is to be inserted is passed as arguments, as zeropos and onepos. Zeropos contains the list of positions in which the value of those positions will be made 0, whereas Onepos contains the list of positions in which the value of those positions will be made 1.

Code snippet of helper.py:

```
def add(a, b):
        result = ""
        s = 0
        i = len(a)-1
        j = len(b)-1
        while i>=0 or j>=0 or s==1:
            if i>=0:
                s+=int(a[i])
            if j>=0:
                s+=int(b[j])
            result = str(s%2) + result
            s //= 2
            i-=1
            j-=1
        return result


def xor(a, b):

        result = ""

        for i in range(1, len(b)):

            if a[i]==b[i]:

                result += '0'

            else:

                result += '1'

        return result


def divide(dividend, divisor):

        xorlen = len(divisor)

        temp = dividend[:xorlen]

        while len(dividend) > xorlen:

            if temp[0]=='1':

                temp=xor(divisor,temp)+dividend[xorlen]

            else:

                temp=xor('0'*xorlen,temp)+dividend[xorlen]

            xorlen += 1

        if temp[0]== '1':

            temp=xor(divisor,temp)

        else:

            temp=xor('0'*xorlen,temp)
        return temp
```

Method descriptions of helper.py :

add(self,a,b) : Takes parameter 2 binary sequences, a and b. Returns a binary sequence after adding a and b according to module-2 arithmetic.

xor(a, b): Takes parameter 2 binary sequences, a and b. Returns a binary sequence after performing the bitwise XOR operation between a and b.

divide(self,dividend,divisor) : Takes parameter 2 binary sequences, dividend and divisor. Returns a binary sequence which is the remainder after dividing the dividend by the divisor according to module-2 arithmetic.

Code Snippet of main.py:

```
import random

import sys

from helper import *

from sender import *

from receiver import *

from medium import *


#function to generate random sequence of 0,1 and store it in a file

def generateRandomInput(length, filename):

    fileout = open(filename, "w")
    for i in range(length):

        fileout.write(str(random.randint(0,1)))

    fileout.close()


def case1(size, filename):

    puterror = True

    print("Error is detected by all four schemes.")

    print("_______________Vertical Redundancy Check___________________")

    type = 1

    s = Sender(size)

    s.createData(filename,type)

    s.displayData(type)

    if puterror:

        s.codewords = injectRandomError(s.codewords)

    r = Receiver(s)

    r.checkError(type)

    r.displayData(type)

    print("_______________Longitudinal Redundancy Check _______________-")

    type = 2

    s = Sender(size)

    s.createData(filename,type)

    s.displayData(type)

    if puterror:

        s.codewords = injectRandomError(s.codewords)
```

```
r = Receiver(s)
r.checkError(type)
r.displayData(type)
print("__________________Checksum_______________________")
type = 3
s = Sender(size)
s.createData(filename,type)
s.displayData(type)
if puterror:
    s.codewords = injectRandomError(s.codewords)
r = Receiver(s)
r.checkError(type)
r.displayData(type)
print("_________________Cyclic Redundancy Check_________________")
type = 4
#poly = "1001"
#print("Generator polynomial:",poly)
poly = input("Enter Generator Polynomial: ")
s = Sender(size)
s.createData(filename,type,poly)
s.displayData(type)
if puterror:
    s.codewords = injectRandomError(s.codewords)
r = Receiver(s)
```

```
    r.checkError(type,poly)

    r.displayData(type)


def case2(size, filename):

    print("--------------------------- CASE 2 -------------------------")

    print("Error is detected by checksum but not by CRC.")

    print("-------------- Checksum ------------------------------------")

    type = 3

    s = Sender(size)

    s.createData(filename,type)

    s.displayData(type)

    zeropos = []

    onepos = [[5]]

    s.codewords = injectAtPosError(s.codewords,zeropos,onepos)

    r = Receiver(s)

    r.checkError(type)

    r.displayData(type)

    print("-------------- Cyclic Redundancy Check ---------------------")

    type = 4

    poly = "1000"

    print("Generator Polynomial:",poly)

    s = Sender(size)

    s.createData(filename,type,poly)

    s.displayData(type)
    s.codewords = injectAtPosError(s.codewords,zeropos,onepos)

    r = Receiver(s)

    r.checkError(type,poly)

    r.displayData(type)


print("------------------------------------------------------------\n")



#function to execute Case 3

def case3(size, filename):

    print("--------------------------- CASE 3 -------------------------")

    print("Error is detected by VRC but not by CRC.")

    print("-------------- Vertical Redundancy Check -------------------")

    type = 1

    s = Sender(size)

    s.createData(filename,type)

    s.displayData(type)

    zeropos = []

    onepos = [[5]]

    s.codewords = injectAtPosError(s.codewords,zeropos,onepos)

    r = Receiver(s)

    r.checkError(type)

    r.displayData(type)

    print("-------------- Cyclic Redundancy Check ---------------------")

    type = 4
    poly = "100"

    print("Generator Polynomial:",poly)

    s = Sender(size)

    s.createData(filename,type,poly)

    s.displayData(type)

    s.codewords = injectAtPosError(s.codewords,zeropos,onepos)

    r = Receiver(s)

    r.checkError(type,poly)

    r.displayData(type)

print("------------------------------------------------------------\n")


#insert case 2 and case 3

#driver function to run the error detection module

if __name__ == "__main__":

    print("------------------------------------------------------------")


    case = int(input("Enter Case Number : "))

    if case == 1:

        size = int(input("Enter length of the dataword: "))

        generateRandomInput(size*4, sys.argv[1])

        case1(size, sys.argv[1])

    elif case == 2:

        size = 8
        case2(size, sys.argv[1])

    elif case == 3:

        size = 6

        case3(size, sys.argv[1])

    else:

        print("You entered invalid choice.")
```

Method descriptions of main.py:

generateRandomInput(length,filename) : Writes a binary sequence of given length in given file.

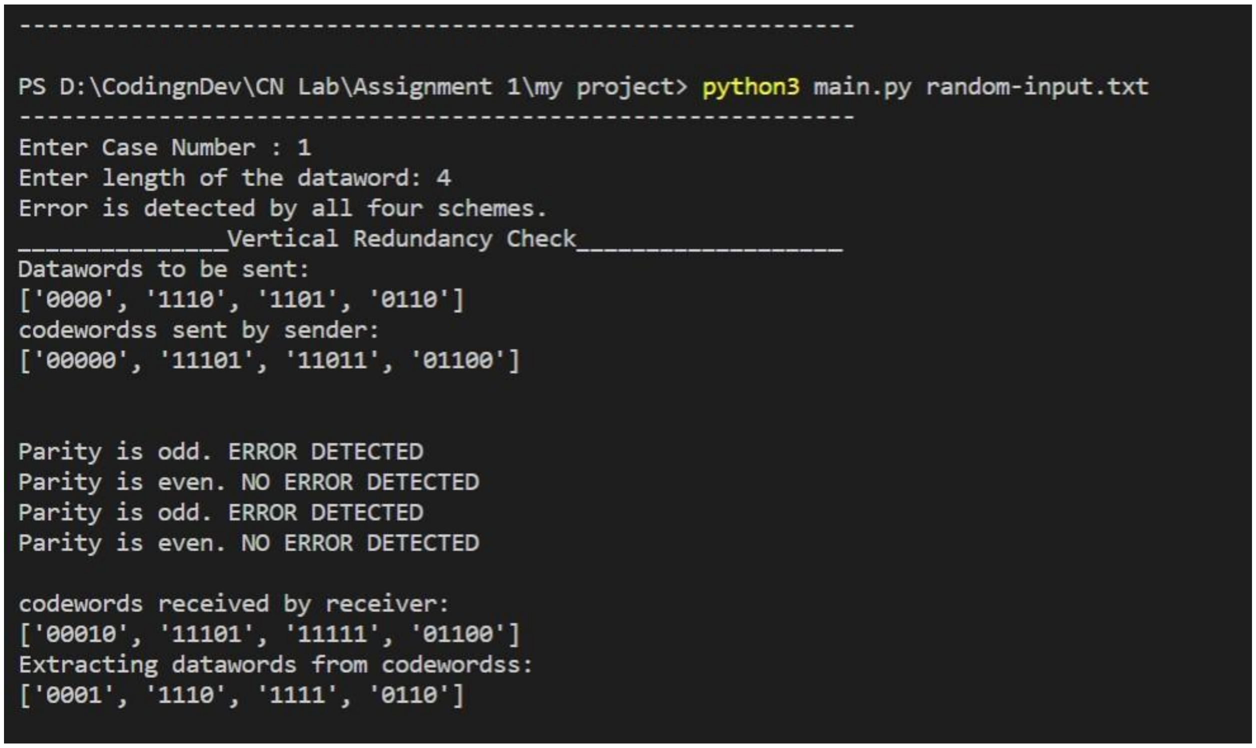
Case1(frame_size, file_name): Method to execute Case 1 of the given assignment.

Case2(frame_size, file_name): Method to execute Case 2 of the given assignment.

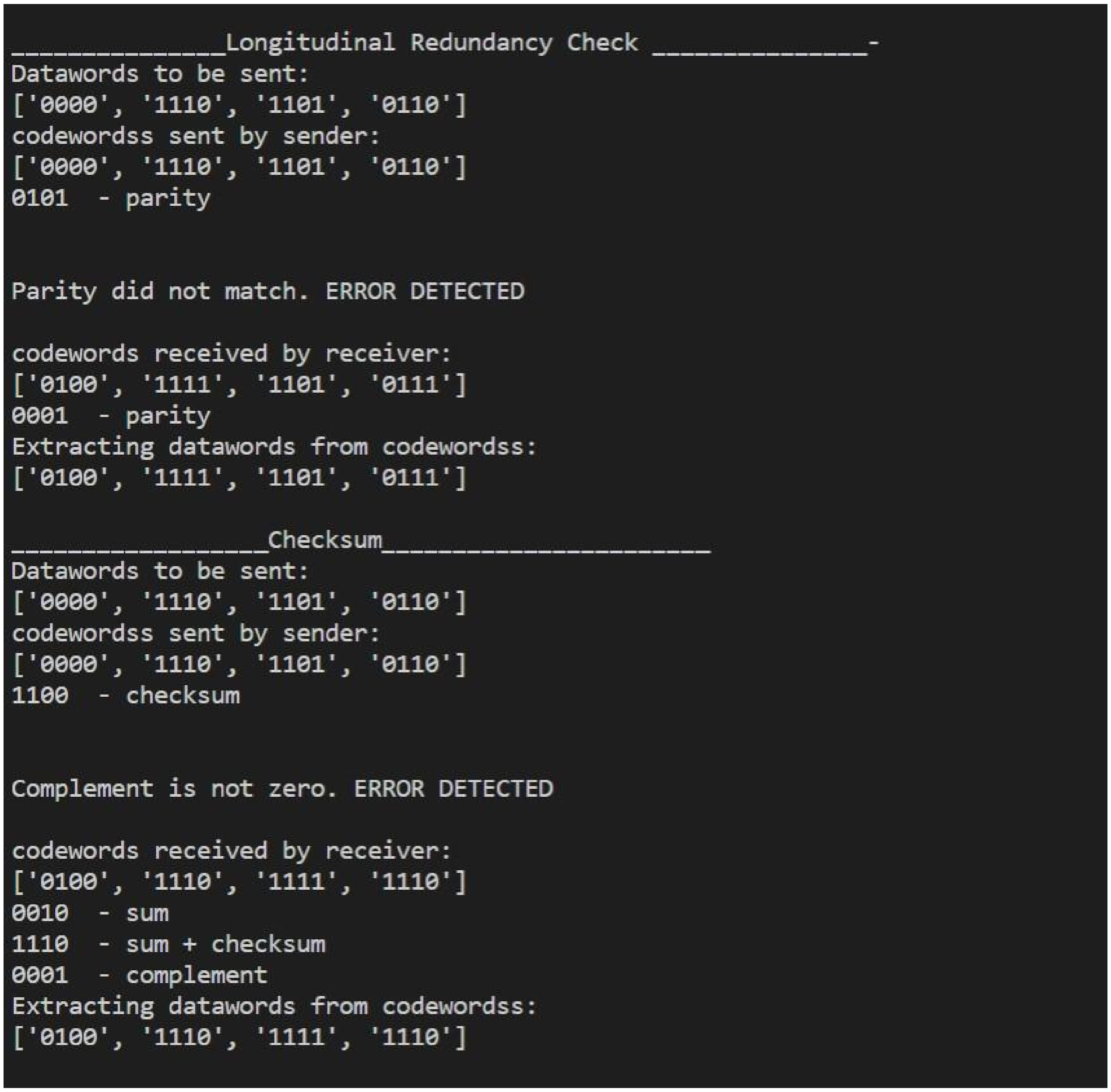
Case3(frame_size, file_name): Method to execute Case 3 of the given assignment.

TEST CASES:

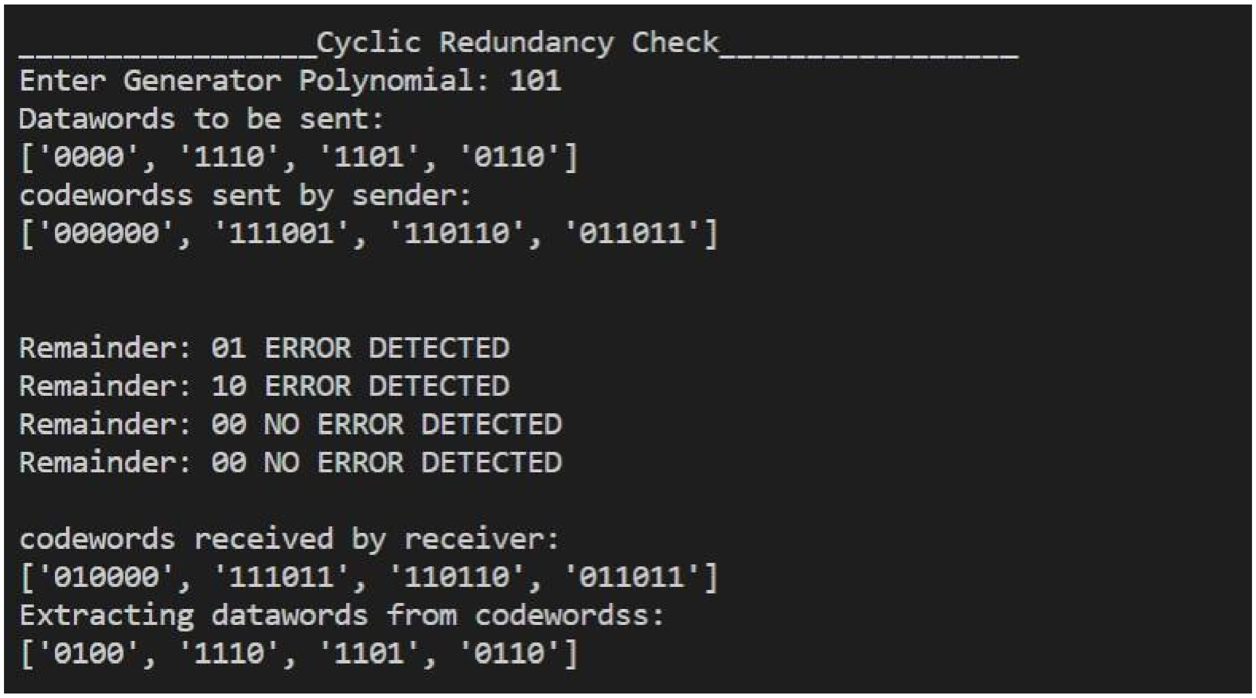
CASE 1:



```
------------------------------------------------------------

PS D:\CodingnDev\CN Lab\Assignment 1\my project> python3 main.py random-input.txt
------------------------------------------------------------
Enter Case Number : 1
Enter length of the dataword: 4
Error is detected by all four schemes.
_______________Vertical Redundancy Check__________________
Datawords to be sent:
['0000', '1110', '1101', '0110']
codewordss sent by sender:
['00000', '11101', '11011', '01100']


Parity is odd. ERROR DETECTED
Parity is even. NO ERROR DETECTED
Parity is odd. ERROR DETECTED
Parity is even. NO ERROR DETECTED

codewords received by receiver:
['00010', '11101', '11111', '01100']
Extracting datawords from codewordss:
['0001', '1110', '1111', '0110']
```

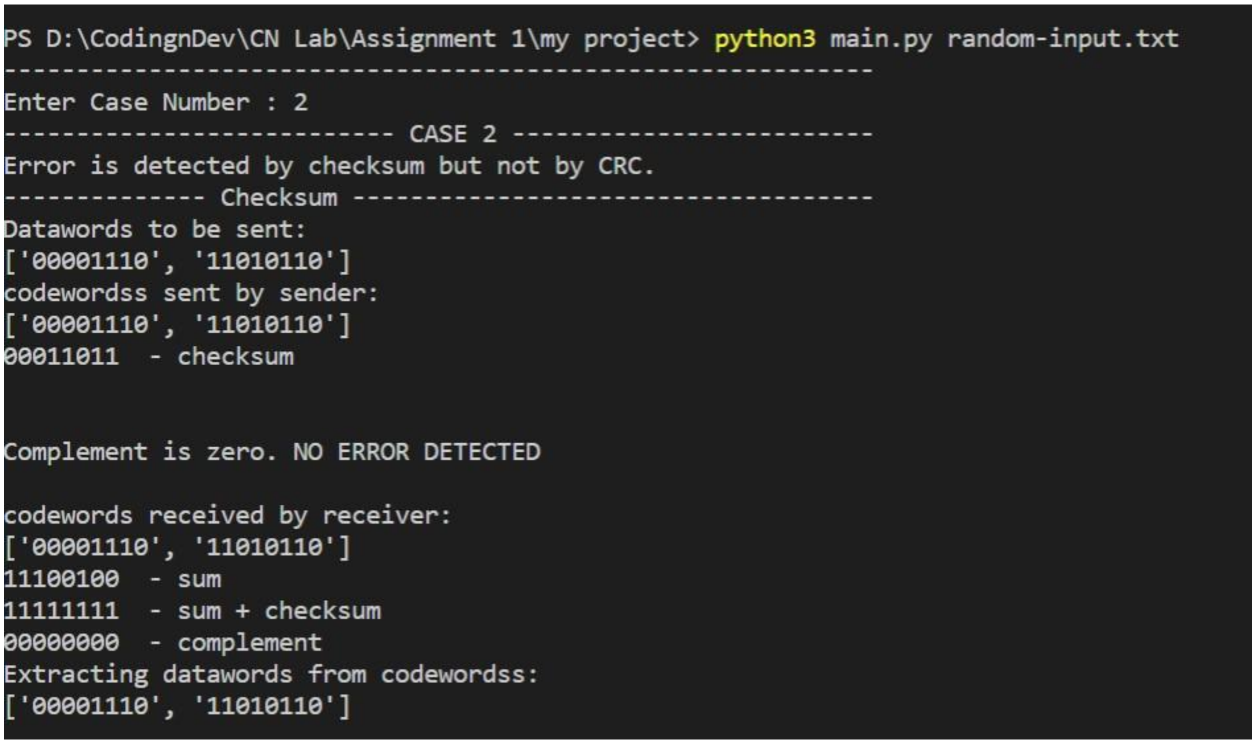


```
_______________Longitudinal Redundancy Check _______________-
Datawords to be sent:
['0000', '1110', '1101', '0110']
codewordss sent by sender:
['0000', '1110', '1101', '0110']
0101  - parity


Parity did not match. ERROR DETECTED

codewords received by receiver:
['0100', '1111', '1101', '0111']
0001  - parity
Extracting datawords from codewordss:
['0100', '1111', '1101', '0111']

__________________Checksum_______________________
Datawords to be sent:
['0000', '1110', '1101', '0110']
codewordss sent by sender:
['0000', '1110', '1101', '0110']
1100  - checksum


Complement is not zero. ERROR DETECTED

codewords received by receiver:
['0100', '1110', '1111', '1110']
0010  - sum
1110  - sum + checksum
0001  - complement
Extracting datawords from codewordss:
['0100', '1110', '1111', '1110']
```

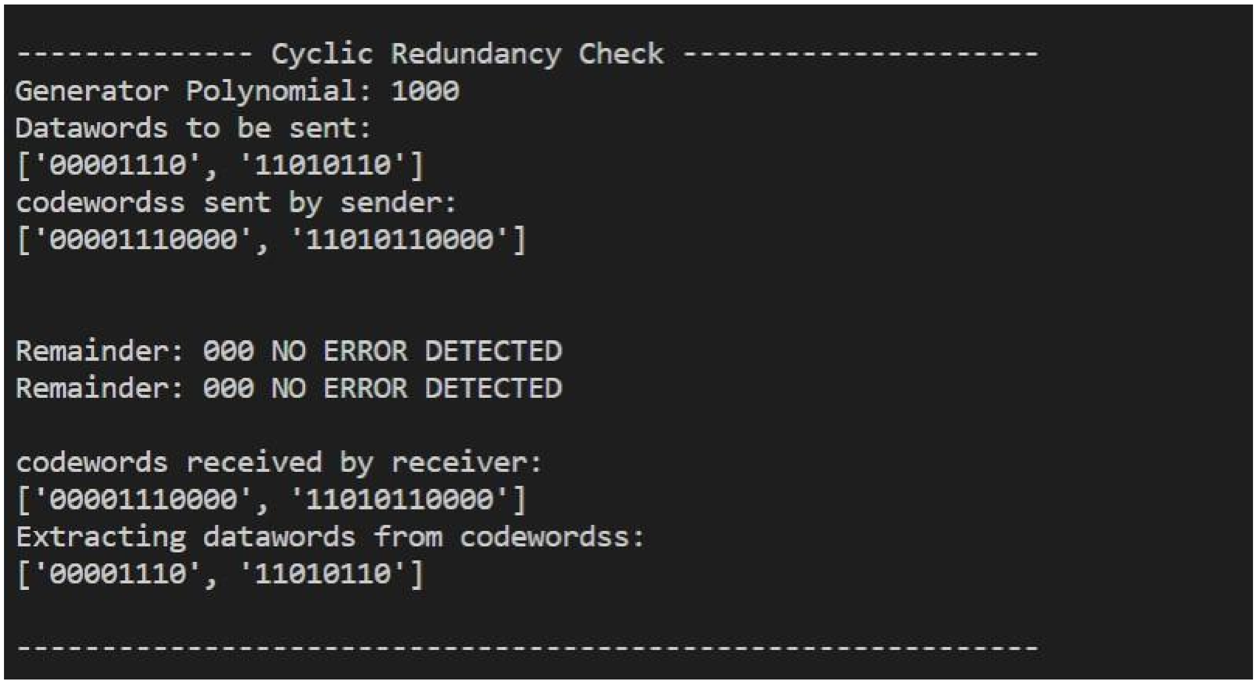


```
_________________Cyclic Redundancy Check_________________
Enter Generator Polynomial: 101
Datawords to be sent:
['0000', '1110', '1101', '0110']
codewordss sent by sender:
['000000', '111001', '110110', '011011']


Remainder: 01 ERROR DETECTED
Remainder: 10 ERROR DETECTED
Remainder: 00 NO ERROR DETECTED
Remainder: 00 NO ERROR DETECTED

codewords received by receiver:
['010000', '111011', '110110', '011011']
Extracting datawords from codewordss:
['0100', '1110', '1101', '0110']
```

CASE 2:

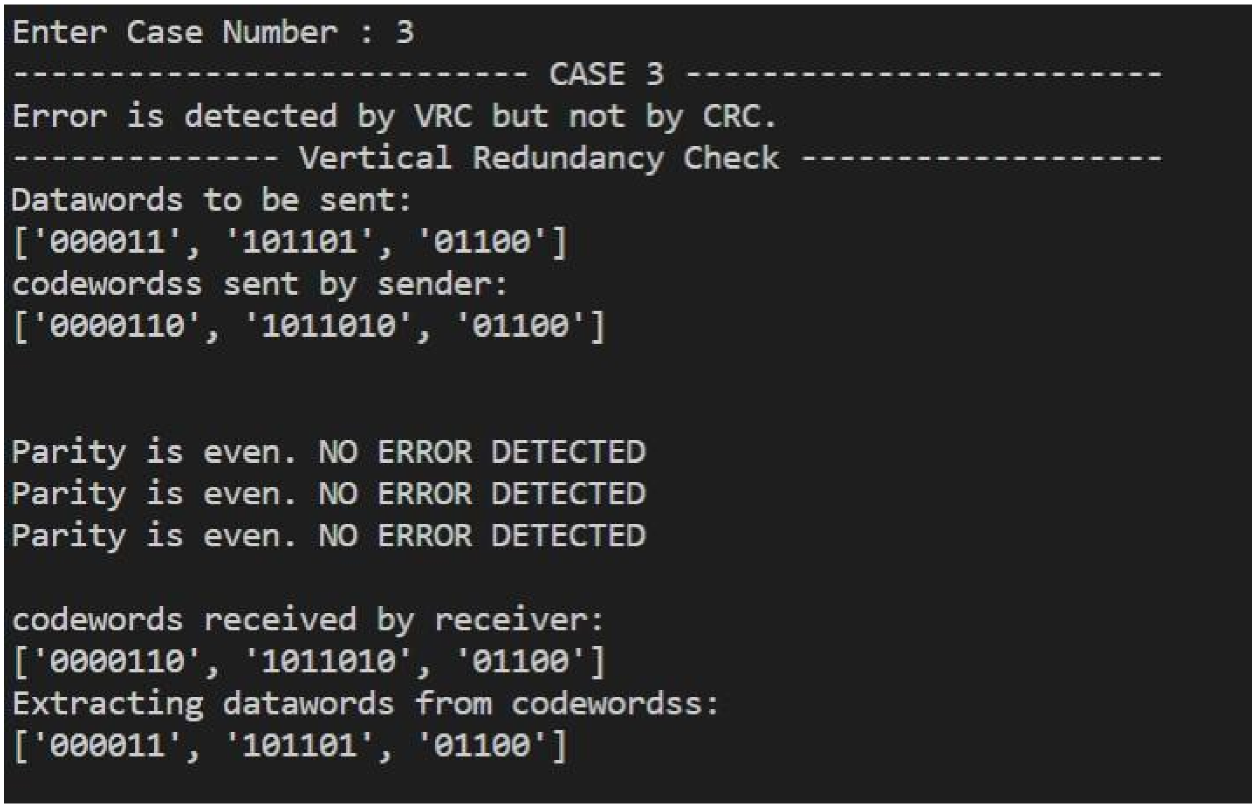


```
PS D:\CodingnDev\CN Lab\Assignment 1\my project> python3 main.py random-input.txt
------------------------------------------------------------
Enter Case Number : 2
--------------------------- CASE 2 -------------------------
Error is detected by checksum but not by CRC.
-------------- Checksum -----------------------------------
Datawords to be sent:
['00001110', '11010110']
codewordss sent by sender:
['00001110', '11010110']
00011011  - checksum


Complement is zero. NO ERROR DETECTED

codewords received by receiver:
['00001110', '11010110']
11100100  - sum
11111111  - sum + checksum
00000000  - complement
Extracting datawords from codewordss:
['00001110', '11010110']
```

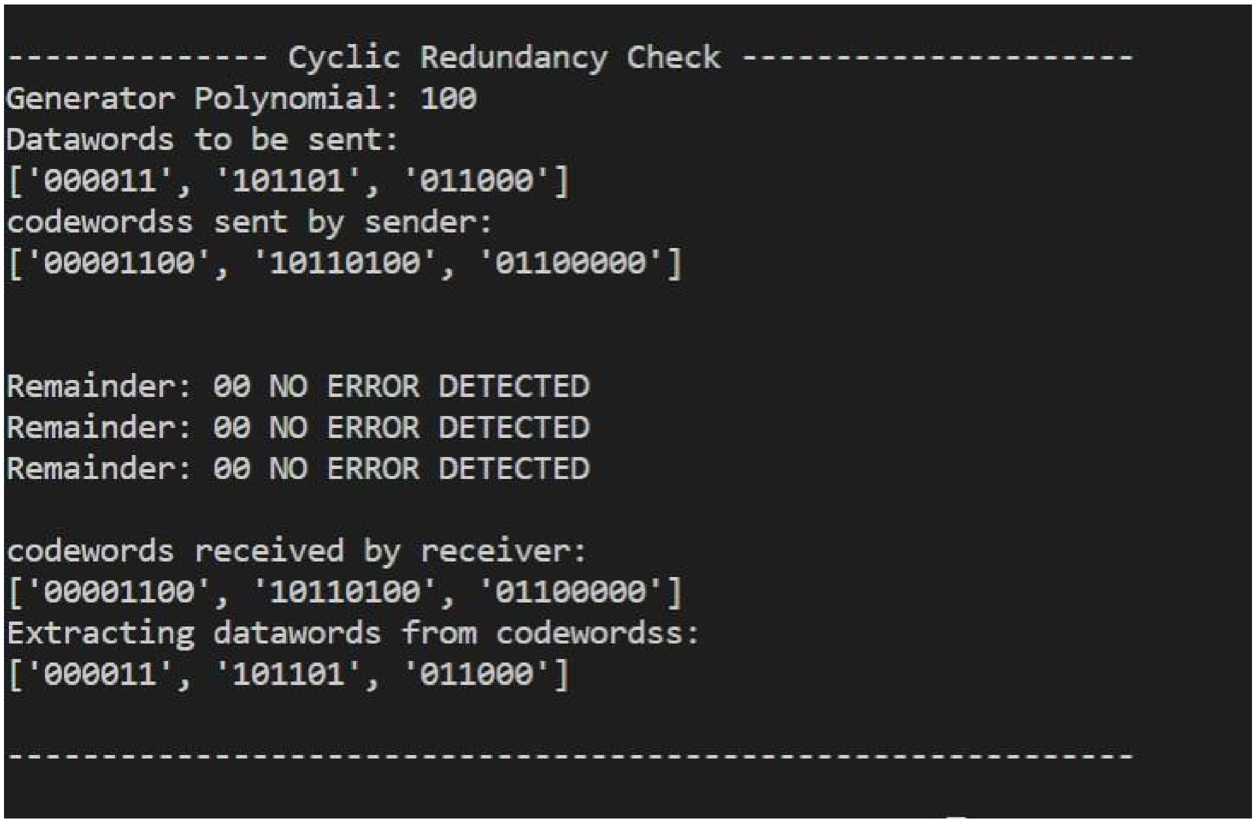


```
-------------- Cyclic Redundancy Check ----------------------
Generator Polynomial: 1000
Datawords to be sent:
['00001110', '11010110']
codewordss sent by sender:
['00001110000', '11010110000']


Remainder: 000 NO ERROR DETECTED
Remainder: 000 NO ERROR DETECTED

codewords received by receiver:
['00001110000', '11010110000']
Extracting datawords from codewordss:
['00001110', '11010110']

------------------------------------------------------------
```

CASE 3:



```
Enter Case Number : 3
--------------------------- CASE 3 -------------------------
Error is detected by VRC but not by CRC.
-------------- Vertical Redundancy Check -------------------
Datawords to be sent:
['000011', '101101', '01100']
codewordss sent by sender:
['0000110', '1011010', '01100']


Parity is even. NO ERROR DETECTED
Parity is even. NO ERROR DETECTED
Parity is even. NO ERROR DETECTED

codewords received by receiver:
['0000110', '1011010', '01100']
Extracting datawords from codewordss:
['000011', '101101', '01100']
```



```
-------------- Cyclic Redundancy Check ---------------------
Generator Polynomial: 100
Datawords to be sent:
['000011', '101101', '011000']
codewordss sent by sender:
['00001100', '10110100', '01100000']


Remainder: 00 NO ERROR DETECTED
Remainder: 00 NO ERROR DETECTED
Remainder: 00 NO ERROR DETECTED

codewords received by receiver:
['00001100', '10110100', '01100000']
Extracting datawords from codewordss:
['000011', '101101', '011000']

------------------------------------------------------------
```

RESULTS:

VERTICAL REDUNDANCY CHECK (VRC)

Vertical redundancy check is also known as Parity check. In this method, a redundant bit is added to a dataword in order to maintain the parity of set bits in the dataword. It is of 2 types: odd parity (a bit is added such that there is an odd number of set bits in the dataword) and even parity (a bit is added such that there is an even number of set bits in the dataword). Here , even parity has been implemented.

Dataword: 1 0 1 0 1

Number of set bits: 3

Parity bit: 1

Codeword: 1 0 1 0 1 0

Dataword: 1 0 1 1 1

Number of set bits: 4

Parity bit: 0

Codeword: 1 0 1 0 1 1

LONGITUDINAL REDUNDANCY CHECK (LRC)

Longitudinal redundancy check is also known as 2D parity check. We can imagine the data to be sent is organized in the form of a table consisting of rows and columns. We attach a parity bit (adhering to either odd or even parity scheme) for every column of the table. Here , even parity has been implemented.

Original data : 1 0 1 1 1 0 0 0 1 0 0 1

1 0 1 1

1 0 0 0

1 0 0 1

Parity: 1 0 1 0

CHECKSUM

According to this scheme, we calculate the sum of all datawords (as per modulo-2 arithmetic) at the sender side. We calculate the 1's complement of this sum (called the checksum) and send it to the receiver. At the receiver side, we calculate the sum of the received codewords along with the checksum. If the result of this addition is 0, it means that the scheme couldn't detect any errors during transmission. Otherwise, there was an error.

Sender site:

1 0 1 1

1 0 0 0

1 0 0 1

+ 1 (carry from the addition)

---

1 1 0 1

Checksum= 0010 (1’s Complement)

Receiver site:

1 0 1 1

1 0 0 0

1 0 0 1

0 0 1 0

---

1 1 1 1

0 0 0 0

Since the complement is 0, we can say that no error occurred during the transmission.

CYCLIC REDUNDANCY CHECK (CRC)

CRC uses a generator polynomial, which is known at both the sender and receiver site. A polynomial can be written in the form

P(x) = ∑ Cr * x^r , where r is a whole number (r = 0, 1, 2, ...) We can denote this polynomial using a binary sequence: if the bit at the r-th position (0- based indexing) is 1, the coefficient of xr is 1, otherwise the coefficient is 0.

CRC performs mod 2 arithmetic (exclusive-OR) on the message using a divisor polynomial. Firstly, the message to be transmitted is appended with CRC bits and the number of such bits is the degree of the divisor polynomial. The divisor polynomial 1 1 0 1 corresponds to the x^3 + x^2 + 1. For example, for the message 1 0 0 1 0 0 with the divisor polynomial 1 1 0 1, the message after appending CRC bits is 1 0 0 1 0 0 0 0 0. We compute CRC on the modified message M.

ANALYSIS :

Vertical Redundancy Check (VRC):

1. This scheme detects all single bit errors. Further, it detects all multiple errors as long as the number of bits corrupted is odd (referred to as odd bit errors). Suppose the message to be transmitted is

10111 10001 10010 and the message received at the receiver is

1 0 0* 1 1 1 0 0 0 1 1 0 0 1 0

2. 0* represents that this bit is in error. For the above example, when the receiver performs the parity check, it detects that there was an error in the first nibble during transmission as there is a mismatch between the

parity bit and the data in the nibble. However, the receiver does not know which bit in that nibble is in error. Similarly, the following error is also detected by the receiver.

The message transmitted is

10111 10001 10010

and the message received at the receiver is

0*1*1 1 1 1 0 0 1*1 1 0 0 1 0

Three bits are corrupted by the transmission medium and this is detected by the receiver as there is a mismatch between the 2nd nibble and the 2nd parity bit. It is important to note that the error in the first nibble is unnoticed as there is no mismatch between the data and the parity bit.

3. The above example also suggests that not alleven bit errors (multiple bit error with the number of bits corrupted even) are detected by this scheme. If even bit error is such that the even number of bits are corrupted in each nibble, then such error is unnoticed at the receiver. However, if an even bit is such that at least one of the nibble has an odd number of bits in error, then such error is detected by VRC.

Longitudinal Redundancy Check (LRC):

1. Similar to VRC, LRC detects all single bit and odd bit errors. Some even bit errors are detected and the rest is unnoticed by the receiver.
2. The following error is detected by LRC but not by VRC. For the message 1011 1000 1001, suppose the received message is

1101 1000 1001 1010

3. **3.** In the above example, there is mismatch between the number of 1's and the parity bit in Columns 2 and 3.
4. **4.** The following error is detected by VRC but not by LRC. For the message 1011 1000 1001, suppose the received message is

   1 1 0 0* . The above error is unnoticed by the receiver if we follow the VRC scheme whereas it is detected by LRC as illustrated below.

1 0 1 0*1 1 0 0 1*1 1 0 0 1 0

5. Here again, not all even bit errors are detected by this scheme. If the error is such that each column has an even number of bits in error, then such error is undetected. However, if the distribution is such that at least one column contains an odd number of bits in error, then such errors are always detected at the receiver.

Checksum:

1. If multiple bit error is such that in each column, a bit '0' is flipped to bit '1', then such an error is undetected by this scheme. Essentially, the message received at the receiver has lost the value 1 1 1 1 1 with respect to the sum. Although it loses this value, this error is unnoticed at the receiver.
2. Also, multiple bit error is such that the difference between the sum of the sender's data and the sum of receiver's data is 1 1 1 1, then this error is unnoticed by the receiver.
3. The above two errors are undetected by the receiver due to the following interesting observations.
4. For any binary data such that a!=0000,the value of a+(1111)in 1's complement arithmetic is a. On the similar line, if a1,...,an are nibbles, then a1+...+ak+(1 1 1 1) = a1+...+ak This is true because, a + 1 1 1 1 gives a – 1 with a carry '1' and in turn this '1' is added with a – 1 as part of 1's complement addition, yielding a.

5. Consider the scenario in which the sender transmits a1,...,ak along with the checksum and the transmission line corrupts multiple bits due to which we lose 1 1 1 1 on the sum. Although the receiver received a1+...+ak-(1 1 1 1), it follows from the above observation that a1+...+ak-(1 1 1 1) is still a1+...+ak, thus error is undetected.
6. Similar to other error detection schemes, checksum detects all odd bit errors and most of even bit errors.

Cyclic Redundancy Check (CRC):

1. The answer to whether CRC detects all errors depends on the divisor polynomial used as a part of CRC computation. Consider the divisor polynomial x^2 which corresponds to 1 0 0 and the message to be transmitted is 1 0 1 0 1 0. After appending CRC bits (in this case 0 0) we get 1 0 1 0 1 0 0 0. Assuming during the data transmission, the 3rd bit is in error and the message received at the receiver is 1 0 1 0 1 1*0 0.

On performing CRC check on 1 0 1 0 1 1 0 0, we see that the remainder is zero and the receiver wrongly concludes that there is no error in transmission. The reason this error is undetected by the receiver is that the message received is perfectly divisible by the divisor 1 0 0. More appropriately, if we visualize the received message as the xor of the original message and the error polynomial, then we see that the error polynomial is divisible by 1 0 0.

2. Message received=10101100
   Message received = message transmitted + error polynomial, i.e.,

10101100=10101000+00000100

Clearly both are divisible by the divisor 1 0 0. In general, if the error polynomial is divisible by the divisor, then such errors are undetected by the receiver. The receiver wrongly concludes that there is no error in transmission. For example, if the error polynomial is 0 0 0 0 1 1 0 0 (3rd and 4th bits in the message in error), then the receiver is unaware of this error.

However, if the divisor is 1 0 1, then both errors are detected by the receiver. Thus, choosing an appropriate divisor is crucial in detecting errors at the receiver if any during the transmission.

COMMENTS:

The assignment has given me the opportunity to actually implement common error-detection schemes and has enhanced my prior knowledge of the schemes. It has also helped in understanding the demerits of a detection scheme, and how such demerits are overcome by other detection schemes. I thoroughly enjoyed researching the concepts and implementing them in code.