LAB REPORT I

NAME: Rohit Sadhu

ROLL: 002010501074

BCSE III, 5-th SEM

GROUP A3, 2022-23

ASSIGNMENT – 1

PROBLEM STATEMENT

Design and implement an error detection module.

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 scheme, codeword will be prepared. Sender will send the codeword to the Receiver. Receiver will extract the data word from codeword and show if there is any error detected. Test the same program to produce a PASS/FAIL result for following cases.

(a) Error is detected by all four schemes. Use a suitable CRC polynomial.

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

(c) Error is detected by VRC but not by CRC.

ERROR DETECTION MECHANISMS:

1. Vertical Redundancy Check (VRC):

   Encoding - An extra ‘0’ bit is appended to the dataword and parity

   (bitwise-XOR of all bits) is calculated. (dataword + parity bit = codeword)

   Decoding - Received codeword is taken and parity bit (bitwise-XOR of all bits) is calculated. If the parity bit is ‘0’, no error is detected.

   Code snippet:

```python
# VRC PARITY
def vrcEncoder(data):
    n = len(data)
    cnt = 0
    for i in range(0, n):
        if(data[i] == '1'):
            cnt += 1

    if(cnt % 2 == 0):
        data = data + "0"
    else:
        data = data + "1"
    return data


def VRCParityCheck(data):
    n = len(data)
    cnt = 0
    for i in range(0, n):
        if (data[i] == '1'):
            cnt += 1
    if (cnt % 2 == 0):
        return True
    else:
        return False
```

2. Longitudinal Redundancy Check (LRC):

Encoding - The dataword is divided into some words of word_size < dataword_size each. Another word of the same word_size containing only ‘0’s is taken. Then we carry out bitwise-XOR of all words. The resulting word consists of the redundant bits has size=word_size. (dataword + redundant bits = codeword)

Decoding - Received codeword is divided into words of size = word_size. Again the redundant bit word is calculated in the same manner. If the resulting word contains all ‘0’ bits, no error is detected.

Code snippet:

```
# LRC 2D PARITY CHECK
def lrcEncoder(data):
    temp = data.split()
    n = len(temp)
    redundant = ""
    for i in range(0, 8):
        cnt = 0
        for j in range(0, 4):
            if(temp[j][i] == '1'):
                cnt += 1

        if(cnt % 2 == 0):
            redundant += "0"
        else:
            redundant += "1"

    return redundant


def lrcHelper(data):

    str_modified = data[:8]+' '+data[8:16]+' '+data[16:24]+'
'+data[24:]
    redundant = lrcEncoder(str_modified)
    return redundant
```

3. <u>Checksum:</u>

Encoding - The datword is divided into some words of word_size < dataword_size each. Another word of the same word_size containing only ‘0’s is taken. Then we carry out 1’s complement addition of all words. The resulting sum, known as checksum, has size=word_size. (dataword + checksum = codeword)

Decoding - Received codeword is divided into words of size = word_size. Again checksum is calculated in the same manner. If the resulting checksum contains all ‘0’ bits, no error is detected.

Code snippet:

```
# CHECKSUM IMPLEMENTATION
def checksumEncoder(SentMessage, k):
    # Dividing sent message in packets of k bits.
    c1 = SentMessage[0:k]
    c2 = SentMessage[k:2 * k]
    c3 = SentMessage[2 * k:3 * k]
    c4 = SentMessage[3 * k:4 * k]

    # CALCULATING THE BINARY SUM OF PACKETS
    Sum = bin(int(c1, 2) + int(c2, 2) + int(c3, 2) + int(c4, 
2))[2:]

    # ADDING THE BITS THAT OVERFLOWED
    if (len(Sum) > k):
        x = len(Sum) - k
        Sum = bin(int(Sum[0:x], 2) + int(Sum[x:], 2))[2:]
    if (len(Sum) < k):
        Sum = '0' * (k - len(Sum)) + Sum

    # CALCULATING THE COMPLEMENT
    Checksum = ''
    for i in Sum:
        if (i == '1'):
            Checksum += '0'
        else:
            Checksum += '1'
    return Checksum
```

```
def checkReceiverChecksum(errorMessage, k, prevChecksum):
    # Dividing sent message in packets of k bits.
    c1 = errorMessage[0:k]
    c2 = errorMessage[k:2 * k]
    c3 = errorMessage[2 * k:3 * k]
    c4 = errorMessage[3 * k:4 * k]

    # CALCULATING BINARY SUM OF PACKETS + CHECKSUM
    ReceiverSum = bin(int(c1, 2) + int(c2, 2) + int(prevChecksum,
2) +
                      int(c3, 2) + int(c4, 2))[2:]

    # ADDING THE BITS THAT OVERFLOWED
    if (len(ReceiverSum) > k):
        x = len(ReceiverSum) - k
        ReceiverSum = bin(
            int(ReceiverSum[0:x], 2) + int(ReceiverSum[x:],
2))[2:]

    # CALCULATING THE COMPLEMENT
    ReceiverChecksum = ''
    for i in ReceiverSum:
        if (i == '1'):
            ReceiverChecksum += '0'
        else:
            ReceiverChecksum += '1'
    return ReceiverChecksum
```

4. Cyclic Redundancy Check (CRC):

Encoding - A key is decided. Length(key) - 1 ‘0’ bits are appended to the dataword and fed into the CRC check bit generator. Here modulo 2 binary division is performed on the augmented dataword with the key as divisor. The remainder consists of the check bits which are appended to the message to get the codeword.

Decoding - Received codeword is fed into the CRC check bit generator. If the remainder contains all ‘0’ bits, no error is detected.

Code snippet:

```python
# CRC IMPLEMENTATION
def xor(a, b):
    # initialize result
    result = []

    # Traverse all bits, if bits are
    # same, then XOR is 0, else 1
    for i in range(1, len(b)):
        if a[i] == b[i]:
            result.append('0')
        else:
            result.append('1')

    return ''.join(result)


def remainderGenerator(dividend,  divisor):
    l = len(divisor)
    f = dividend[0: l]
    while (l < len(dividend)):
        if f[0] == '1':
            f = xor(divisor, f) + dividend[l]
            # print("if, length = ", len(f))

        else:
            f = xor('0' * l, f) + dividend[l]
            # print("else, length = ", len(f))
        l += 1
    if f[0] == '1':
        f = xor(divisor, f)
    else:
        f = xor('0' * l, f)
    return f

def crcEncoder(data, divisor):
    appended_data = data + '0' * (len(divisor)-1)
    remainder = remainderGenerator(appended_data, divisor)
    codeword = data + remainder
    return codeword
```

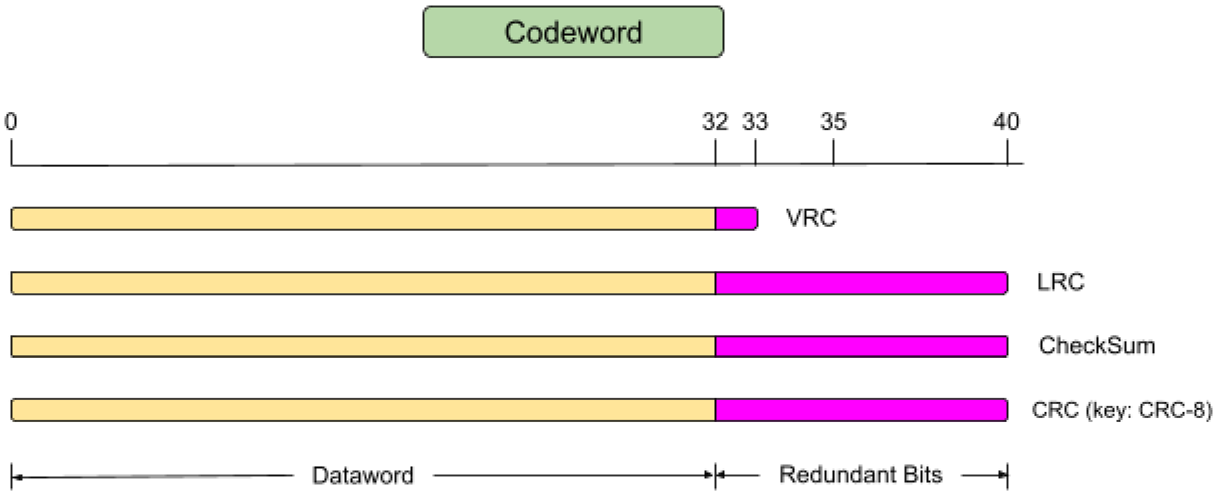
ERROR INJECTION STRATEGIES

1. Single Bit Error: A random bit from the codeword is chosen and flipped.
2. Burst Error: An error window of size (provided as argument) < codeword is chosen randomly. Within this window, some bits are chosen randomly and flipped.
3. Random Error: Few bits are randomly chosen and flipped.

Code Snippet:

```
def errorGenerator(data):
    n = len(data)

    pos = 4

    # print(temp)
    while pos > 0:
        index = random.randint(0, n-1)

        if(data[index] == '1'):

            data = data[:index]+'0'+data[index+1:]
        else:

            data = data[:index]+'1'+data[index+1:]

        pos -= 1
    redundant = lrcHelper(data)
    return data, redundant
```

DESIGN



ACK Word = {ith bit = **0** if ERROR NOT DETECTED, **1** if ERROR DETECTED, i = 0, 1, 2, 3}

MAIN LOGIC:

We take a number as input and convert it into a 32-bit binary string and then apply the logic.

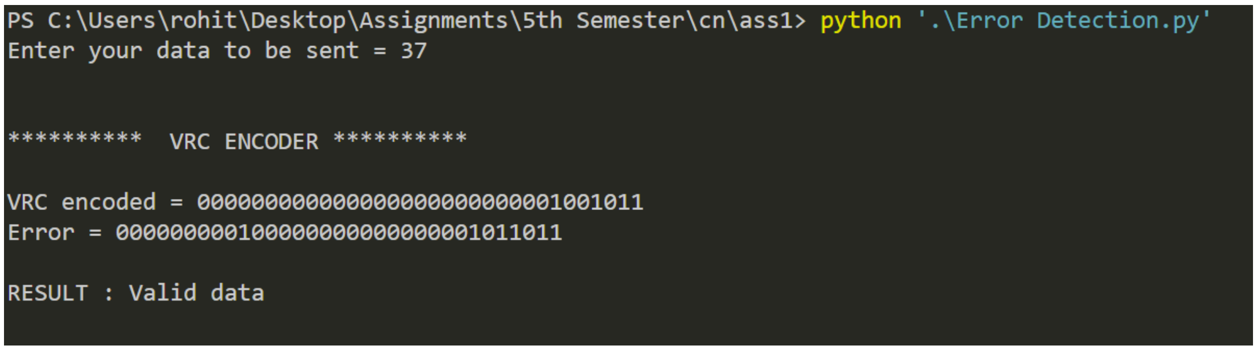
```python
if __name__ == '__main__':
    dataWord = int(input("Enter your data to be sent = "))
    data = '{:032b}'.format(dataWord)

    error, errored_redundant = errorGenerator(data)

    # 1. VRC
    print()
    print()
    print("**********  VRC ENCODER **********")
    print()
    encoded = vrcEncoder(data)
    encoded_error, _ = errorGenerator(encoded)
    print('VRC encoded = ' + encoded)
    print('Error = ' + encoded_error)
    print()
    print('RESULT : ', end='')
    if(VRCParityCheck(encoded_error)):
        print("Valid data")
    else:
        print("Errored Data")
    print()
    print()

    # 2. LRC
    print()
    print()
    print("**********  LRC ENCODER **********")
    print()
    lrc_redundant = lrcHelper(data)
    print('LRC encoded redundant = ' + lrc_redundant)
    print('Errored redundant = ' + errored_redundant)
    print()
    print('RESULT : ', end='')
    if(lrc_redundant == errored_redundant):
        print("Valid data")
    else:
        print("Errored Data")
    print()
    print()

    # 3. CHECKSUM
    print()
    print()
    print("**********  CHECKSUM ENCODER **********")
    print()
    k = int(input("Enter the packet length for CheckSum = "))
    encoded = checksumEncoder(data, k)
    decoded_after_error = checkReceiverChecksum(error, k, encoded)
    ans = int(decoded_after_error, 2)

    print('CHECKSUM encoded = ' + encoded)
    print('Error = ' + decoded_after_error)
    print()
    print('RESULT : ', end='')
    if(ans == 0):
        print("Valid Data")
    else:
        print("Errored Data")
    print()
    print()

    # 4. CRC
    print()
    print()
    print("**********  CRC ENCODER **********")
    print()
    divisor = int(input("Enter the divisor = "))
    divisor = bin(divisor)[2:]

    encoded = crcEncoder(data, divisor)
    error, _ = errorGenerator(encoded)
    decoded_remainder_after_error = remainderGenerator(error, divisor)
    ans = int(decoded_remainder_after_error, 2)

    print('CRC encoded = ' + encoded)
    print('Data after error insertion = ' + error)
    print('Decoded remainder after error = ' +
decoded_remainder_after_error)
    print()
    print('RESULT : ', end='')
    if(ans == 0):
        print("Valid data")
    else:
        print("Errored Data")
    print()
    print()
```

PACKET_SIZE = k units.

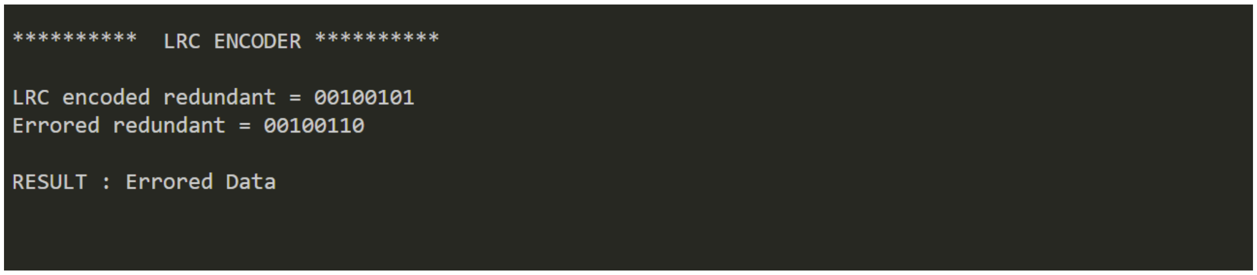
The string is divided into 32 / k (=PACKET_COUNT) datawords.

For each dataword, codewords are obtained using 4 schemes: VRC, LRC, Checksum, CRC-8

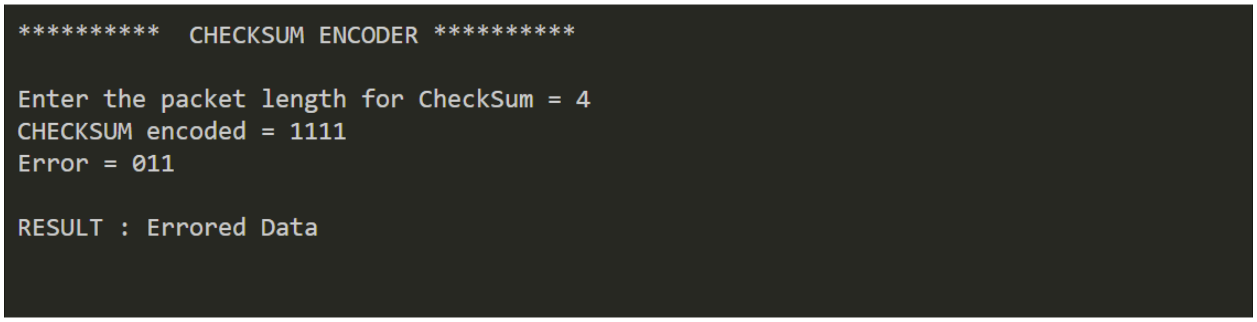
Output for a case with random error injected:



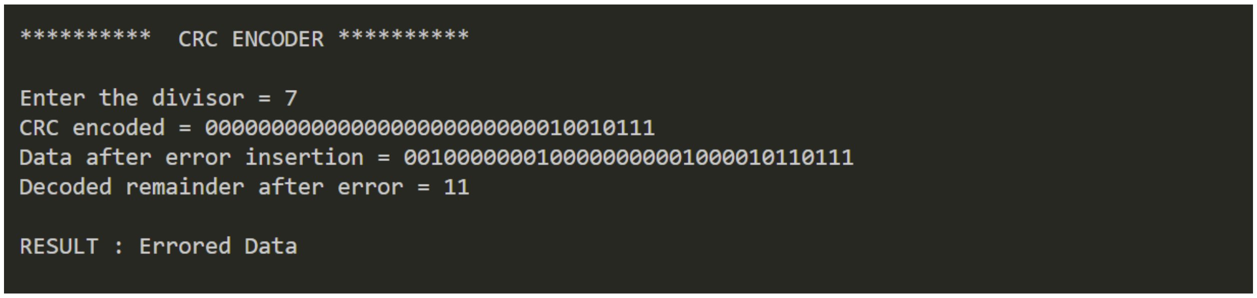
```
PS C:\Users\rohit\Desktop\Assignments\5th Semester\cn\ass1> python '.\Error Detection.py'
Enter your data to be sent = 37


**********  VRC ENCODER **********

VRC encoded = 000000000000000000000000001001011
Error = 000000000100000000000000001011011

RESULT : Valid data
```



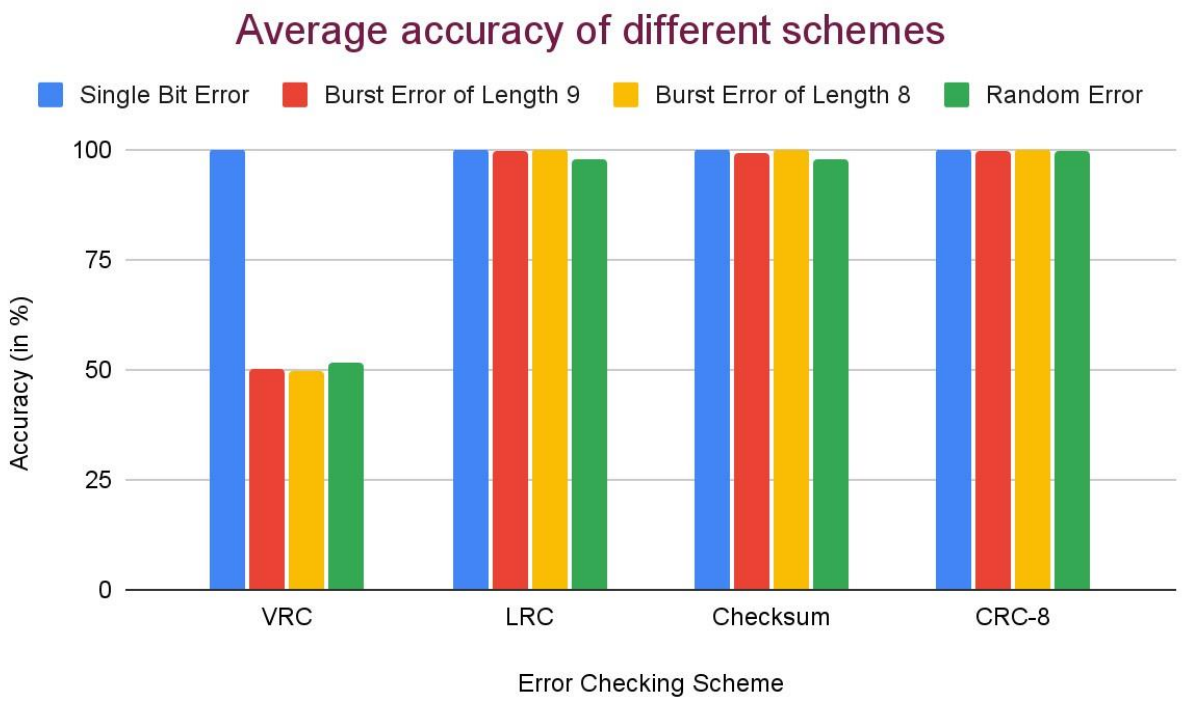
```
**********  LRC ENCODER **********

LRC encoded redundant = 00100101
Errored redundant = 00100110

RESULT : Errored Data
```



```
**********  CHECKSUM ENCODER **********

Enter the packet length for CheckSum = 4
CHECKSUM encoded = 1111
Error = 011

RESULT : Errored Data
```



```
**********  CRC ENCODER **********

Enter the divisor = 7
CRC encoded = 0000000000000000000000000010010111
Data after error insertion = 0010000000010000000001000010110111
Decoded remainder after error = 11

RESULT : Errored Data
```



ANALYSIS

The following conclusions can be drawn:

1. VRC can detect all odd numbered errors, and cannot detect any even numbered errors as it simply takes a xor of all the bits in the code word. Thus, VRC can detect all single bit errors as shown in the first block of rows.

2. LRC can detect the errors which get distributed in such a way that all the columns have even numbered bit flips. In such a situation the xor will remain unchanged and error will go undetected. This is because the LRC divides the code word in packets of equal bits and then calculates bitwise XOR of the packets, so an odd number of bit flips in a particular position will generate '1' for that position and the resultant check bits will be non-zero. So single bit errors and burst errors of length 8 can be detected as in these cases, only 1 bit per column is changed. However in case of burst error of length 9, the first and last error bit will overlap, and if the bits in between do not flip, no error is detected. In general, if word_size is odd, LRC can detect all even-numbered errors and if word_size is even, LRC can detect all odd-numbered errors.

3. Checksum works by dividing the codeword into words and then calculating 1's complement binary addition of these words. It is unable to detect any error if the net change in sum is 0 or a multiple of 2^(size of each word).

4. In CRC-8, the coefficient of x^0 is 1 and 6 high bits are present, so it can detect all single bit errors. If the difference between 2 error bits is less than the number of check bits, the error is always detected. This CRC module hence can detect all burst errors of burst length less/equal to the degree of polynomial. Since

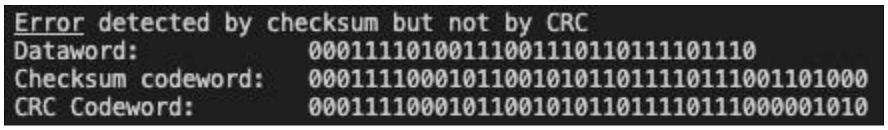
the degree of the divisor polynomial is 8, the remainder in case of burst errors of length <= 8 will always be non-zero and hence these errors will be detected with 100% accuracy.

This is clearly visible that CRC with key CRC-8 outperforms all other mechanisms with a slight margin in detecting any kind of errors. VRC is the least powerful but is easy to implement. LRC and Checksum are pretty much the same in their ability to detect errors.

To summarize: ability to detect errors - **VRC << Checksum ≈ LRC < CRC**

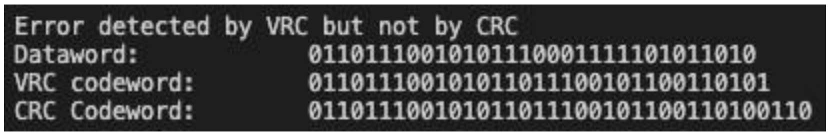
Some special cases (for comparison, error is induced in only dataword):

1. Error is detected by Checksum but not by CRC (key CRC-8 “111010101”)



```
Error detected by checksum but not by CRC
Dataword:            00011110100111001110110111101110
Checksum codeword:   0001111000101100101011011110111001101000
CRC Codeword:        0001111000101100101011011110111000001010
```

2. Error is detected by VRC but not by CRC - For this, CRC-4 (“10101”) is taken as the key because CRC-8 will detect all odd-numbered errors as it has a factor x + 1



```
Error detected by VRC but not by CRC
Dataword:            01101110010101110001111101011010
VRC codeword:        011011100101011011100101100110101
CRC Codeword:        011011100101011011100101100110100110
```

COMMENTS

The assignment helps to gain in-depth understanding of different error checking schemes, how they work, their advantages and disadvantages. This will enable me to deal with possible bugs. For example, a good choice of CRC polynomial can eliminate errors to a much larger extent. The project is simulated using a client-server socket model which gives a sense of real-word scenario, and can be improved through a module simulating the channel, instead of inducing errors on the sender side itself.