**Batch: B - 2 Roll No.: 16014022050**

**Experiment / ~~assignment~~ / ~~tutorial~~ No.: 8**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

**Experiment No.: 8**

|  |
| --- |
| **TITLE:** Implementation of CRC & Checksum for Computer Networks |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**AIM:** To implement Layer 2 Error detection schemes: CRC & Checksum.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

**C02:**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. A. S. Tanenbaum, “Computer Networks”, Pearson Education, Fourth Edition
2. B. A. Forouzan, “Data Communications and Networking”, TMH, Fourth Edition

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Pre Lab/ Prior Concepts:**

Data Link Layer, Error Correction/Detection, Types of Errors

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**New Concepts to be learned:** Checksum

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**CRC (Cyclic Redundancy Check):**

Cyclic Redundancy Check (CRC) is another error detection technique to detect errors in data that has been transmitted on a communications link. A sending device applies a 16 or 32 bit polynomial to a block of data that is to be transmitted and appends the resulting cyclic redundancy check (CRC) to the block. The receiving end applies the same polynomial to the data and compares its result with the result appended by the sender. If they agree, the data has been received successfully. If not, the sender can be notified to resend the block of data.

**At Sender Side:**

* Sender has a generator G(x) polynomial.
* Sender appends (n-1) zero bits to the data.

Where, n= no of bits in generator

* Dividend appends the data with generator G(x) using modulo 2 division (arithmetic).
* Remainder of (n-1) bits will be CRC.

**Codeword:** It is combined form of Data bits and CRC bits i.e. Codeword = Data bits + CRC bits.

**Example**

Assume that –

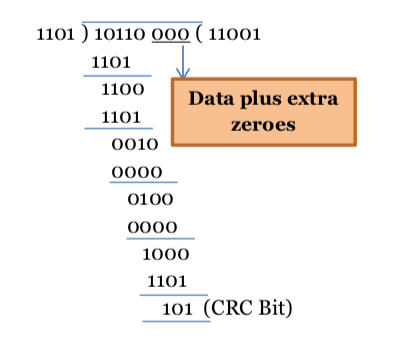
(a) data is 10110.

(b) code generator is 1101.

(Code generator can also be mentioned in polynomial: x3+x2+1 )

**Calculate CRC Bits:** While calculating the CRC bits, we pad (n-1) 0’s to the message bits, where ‘n’ = no of bits in the code generator.

Cyclic Redundancy check will be generated as shown below –



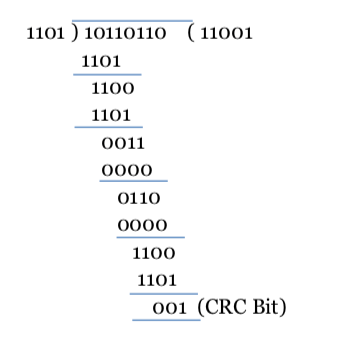
**Figure 1: CRC calculation by sender**

**At Receiver Side**

* Receiver has same generator G(x).
* Receiver divides received data (data + CRC) with generator.
* If remainder is zero, data is correctly received.
* Else, there is error.

Assume the received message is 10110110.

**Calculate CRC Bits:** It does not add any padding bits, rather calculates from the entire received code word.



**Figure 2: CRC calculation by receiver**

The CRC bits are calculated to be different. Thus, there is an error detected.

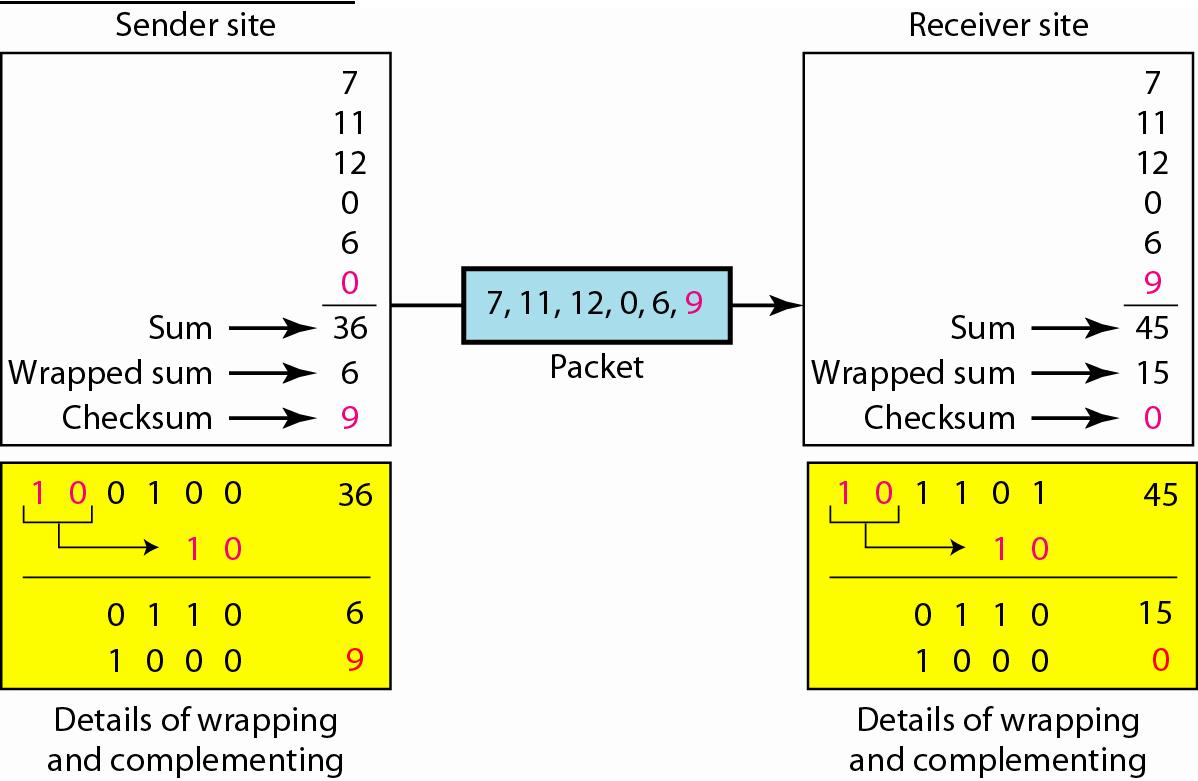
**Internet Checksum:**

A checksum is a simple type of redundancy check that is used to detect errors in data.

Errors frequently occur in data when it is written to a disk, transmitted across a network or otherwise manipulated. The errors are typically very small, for example, a single incorrect bit, but even such small errors can greatly affect the quality of data, and even make it useless.

In its simplest form, a checksum is created by calculating the binary values in a packet or other block of data using some algorithm and storing the results with the data. When the data is retrieved from memory or received at the other end of a network, a new checksum is calculated and compared with the existing checksum. A non-match indicates an error; a match does not necessarily mean the absence of errors, but only that the simple algorithm was not able to detect any.

**Simple Checksum:**



**Internet Checksum:**

The following process generates Internet Checksum

Assume the packet header is: 01 00 F2 03 F4 F5 F6 F7 00 00

(00 00 is the checksum to be calculated)

The first step is to form 16-bit words.

0100 F203 F4F5 F6F7

The second step is to calculate the sum using 32-bits.

0100 + F203 + F4F5 + F6F7 = 0002 DEEF

The third step is to add the carries (0002) to the 16-bit sum.

DEEF + 002 = DEF1

The fourth step is to take the complement. (1s becomes 0s and 0s become 1s)

~DEF1 = 210E

So the checksum is 21 0E.

The packet header is sent as: 01 00 F2 03 F4 F5 F6 F7 21 0E

\* At the receiver, the steps are repeated.

The first step is to form 16-bit words.

0100 F203 F4F5 F6F7 210E

The second step is to calculate the sum using 32-bits.

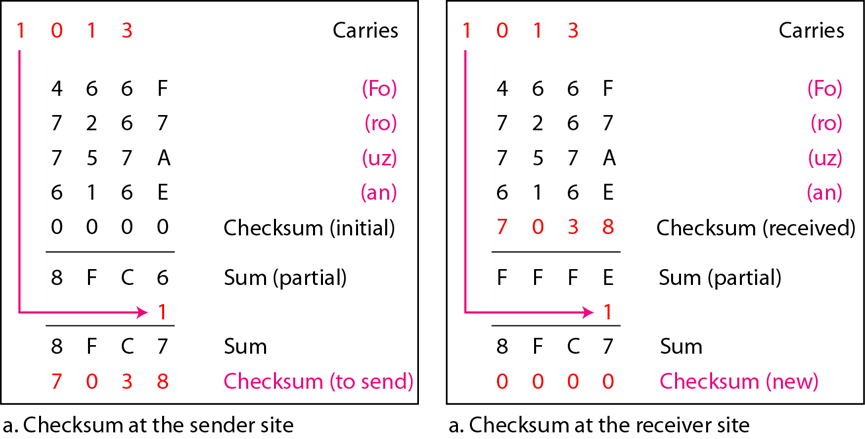
0100 + F203 + F4F5 + F6F7 + 210E = 0002 FFFD

The third step is to add the carries (0002) to the 16-bit sum.

FFFD + 0002 = FFFF which means that no error was detected.

(In 1s complement, zero is 0000 or FFFF.)

**Example:**

****

**IMPLEMENTATION:** (printout of codes)

* + - 1. **CheckSum:**

# input: 10011001111000100010010010000100

def calculate\_checksum(binary\_strings, bit\_size):

    integers = [int(binary, 2) for binary in binary\_strings]

    total\_sum = sum(integers)

    mask = (1 << bit\_size) - 1

    wrapped\_sum = total\_sum & mask

    checksum\_value = wrapped\_sum ^ mask

    checksum\_binary = bin(checksum\_value)[2:].zfill(bit\_size)

    return checksum\_binary

def verify\_checksum(data\_with\_checksum, bit\_size):

    integers = [int(binary, 2) for binary in data\_with\_checksum]

    total\_sum = sum(integers)

    mask = (1 << bit\_size) - 1

    wrapped\_sum = total\_sum & mask

    expected\_result = mask

    return wrapped\_sum == expected\_result, bin(wrapped\_sum)[2:].zfill(bit\_size)

def divide\_into\_chunks(binary\_string, chunk\_size):

    if len(binary\_string) % chunk\_size != 0:

        padding\_needed = chunk\_size - (len(binary\_string) % chunk\_size)

        binary\_string = '0' \* padding\_needed + binary\_string

    chunks = [binary\_string[i:i+chunk\_size] for i in range(0, len(binary\_string), chunk\_size)]

    return chunks

def main():

    binary\_string = input("Enter the binary string: ").strip()

    bit\_size = int(input("Enter the bit size (8, 16, etc.): "))

    binary\_string = ''.join([ch for ch in binary\_string if ch in '01'])

    print("\n--- SENDER SIDE ---")

    chunks = divide\_into\_chunks(binary\_string, bit\_size)

    print(f"Data divided into {bit\_size}-bit chunks:")

    for i, chunk in enumerate(chunks):

        print(f"Chunk {i+1}: {chunk}")

    checksum = calculate\_checksum(chunks, bit\_size)

    print(f"\nSender Checksum: {checksum}")

    sender\_data = chunks + [checksum]

    print("\nSender Data (with checksum):")

    for i, data in enumerate(sender\_data):

        if i < len(chunks):

            print(f"Chunk {i+1}: {data}")

        else:

            print(f"Checksum: {data}")

    print("\n--- RECEIVER SIDE ---")

    introduce\_error = input("\nDo you want to introduce an error? (y/n): ").lower().strip()

    receiver\_data = sender\_data.copy()

    if introduce\_error == 'y':

        chunk\_index = int(input("Enter chunk index to modify (1 to {}): ".format(len(chunks)))) - 1

        bit\_position = int(input(f"Enter bit position to flip (1 to {bit\_size}): ")) - 1

        chunk\_list = list(receiver\_data[chunk\_index])

        chunk\_list[bit\_position] = '1' if chunk\_list[bit\_position] == '0' else '0'

        receiver\_data[chunk\_index] = ''.join(chunk\_list)

        print(f"\nModified Chunk {chunk\_index+1}: {receiver\_data[chunk\_index]}")

    print("\nReceiver Data (with checksum):")

    for i, data in enumerate(receiver\_data):

        if i < len(chunks):

            print(f"Chunk {i+1}: {data}")

        else:

            print(f"Checksum: {data}")

    is\_valid, result = verify\_checksum(receiver\_data, bit\_size)

    print(f"\nSum at receiver (including checksum): {result}")

    if is\_valid:

        print("\nRESULT: all 1's, no errors detected")

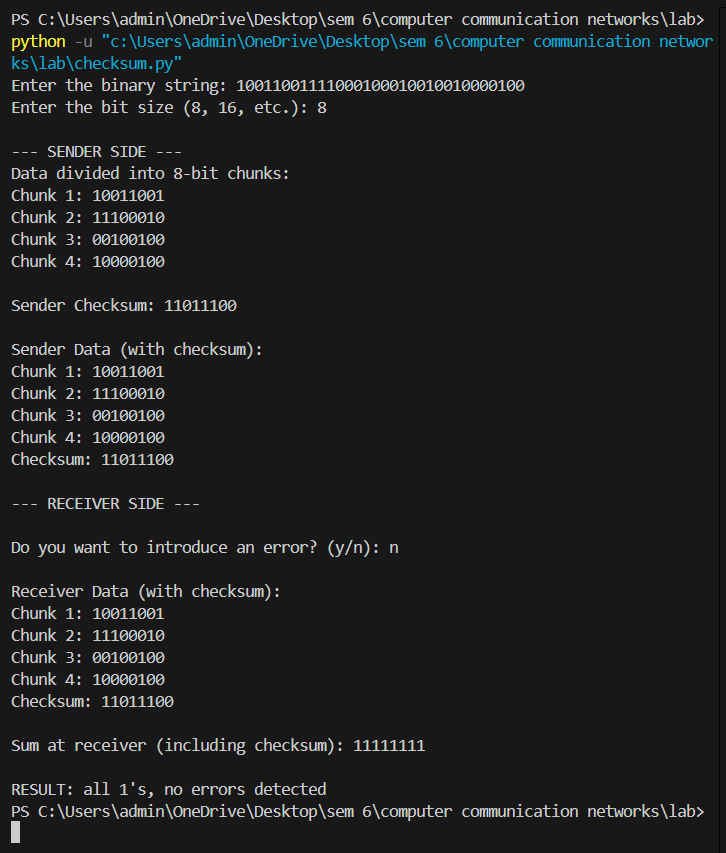
    else:

        print(f"\nRESULT: not all 1's, errors detected")

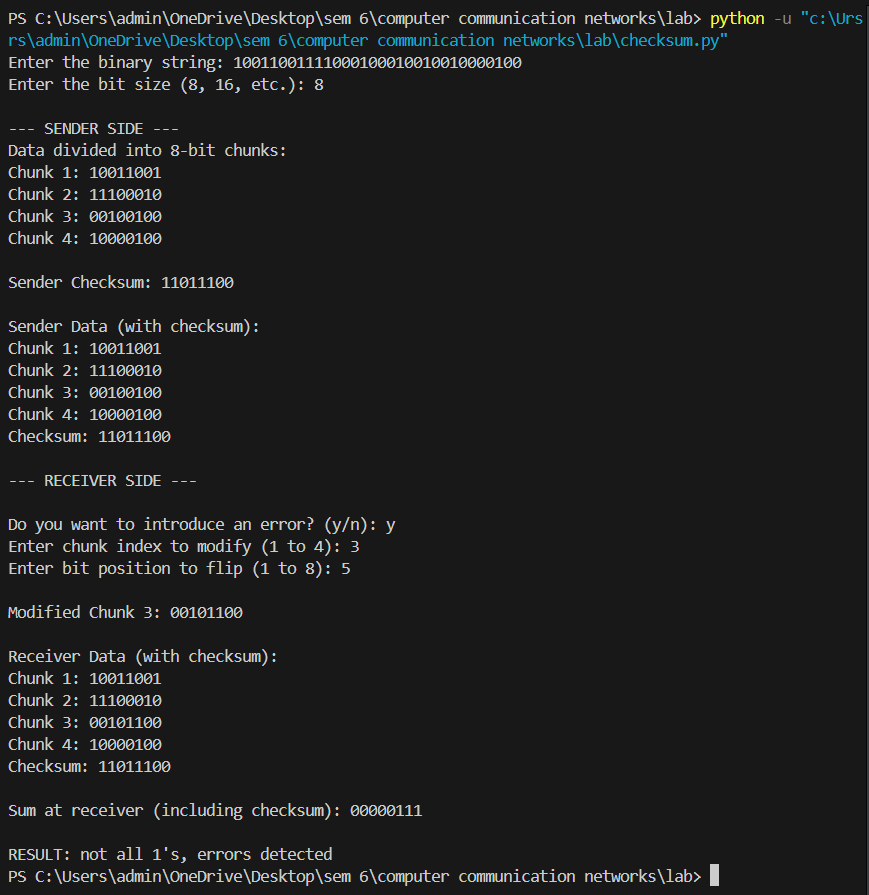
if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Case 1: No Error**

****

**Case 2: Error in Transmission**

****

* + - 1. **CRC:**

def binary\_division(dividend, divisor):

    dividend\_bits = [int(bit) for bit in dividend]

    divisor\_bits = [int(bit) for bit in divisor]

    for i in range(len(dividend\_bits) - len(divisor\_bits) + 1):

        if dividend\_bits[i] == 1:

            for j in range(len(divisor\_bits)):

                dividend\_bits[i+j] ^= divisor\_bits[j]

    remainder = dividend\_bits[-(len(divisor\_bits)-1):]

    return ''.join(map(str, remainder))

def calculate\_crc(data, generator, pad\_length):

    padded\_data = data + '0' \* pad\_length

    remainder = binary\_division(padded\_data, generator)

    return remainder

def verify\_crc(data\_with\_crc, generator):

    remainder = binary\_division(data\_with\_crc, generator)

    return all(bit == '0' for bit in remainder), remainder

def main():

    data = input("Enter the binary data: ").strip()

    generator = input("Enter the generator polynomial (in binary): ").strip()

    data = ''.join([ch for ch in data if ch in '01'])

    generator = ''.join([ch for ch in generator if ch in '01'])

    pad\_length = len(generator) - 1

    print("\n--- SENDER SIDE ---")

    print(f"Original Data: {data}")

    print(f"Generator Polynomial: {generator}")

    print(f"Degree of Polynomial: {pad\_length}")

    crc = calculate\_crc(data, generator, pad\_length)

    print(f"CRC: {crc}")

    transmitted\_data = data + crc

    print(f"Transmitted Data (Data + CRC): {transmitted\_data}")

    print("\n--- RECEIVER SIDE ---")

    introduce\_error = input("\nDo you want to introduce an error? (y/n): ").lower().strip()

    received\_data = transmitted\_data

    if introduce\_error == 'y':

        bit\_position = int(input(f"Enter bit position to flip (1 to {len(transmitted\_data)}): ")) - 1

        received\_bits = list(received\_data)

        received\_bits[bit\_position] = '1' if received\_bits[bit\_position] == '0' else '0'

        received\_data = ''.join(received\_bits)

        print(f"\nOriginal Transmitted Data: {transmitted\_data}")

        print(f"Modified Received Data: {received\_data}")

    else:

        print(f"Received Data: {received\_data}")

    is\_valid, remainder = verify\_crc(received\_data, generator)

    print(f"\nRemainder after division: {remainder}")

    if is\_valid:

        print("\nRESULT: remainder is all zeros, no errors detected")

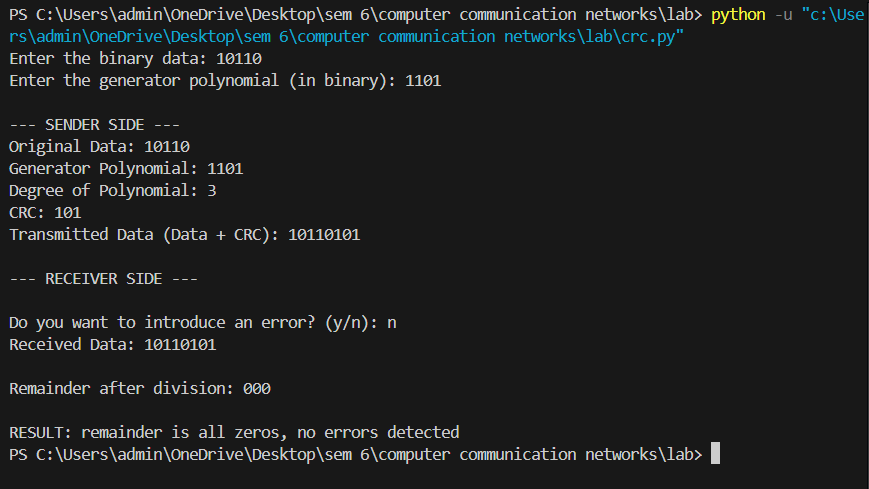
    else:

        print(f"\nRESULT: remainder is not all zeros, errors detected")

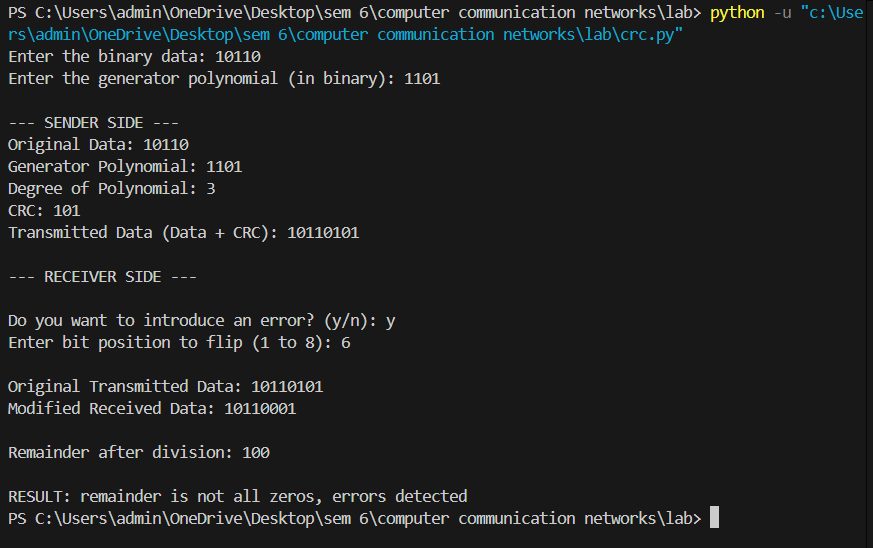
if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Case 1: No Error**

****

**Case 2: Error in Transmission**

****

**CONCLUSION:**

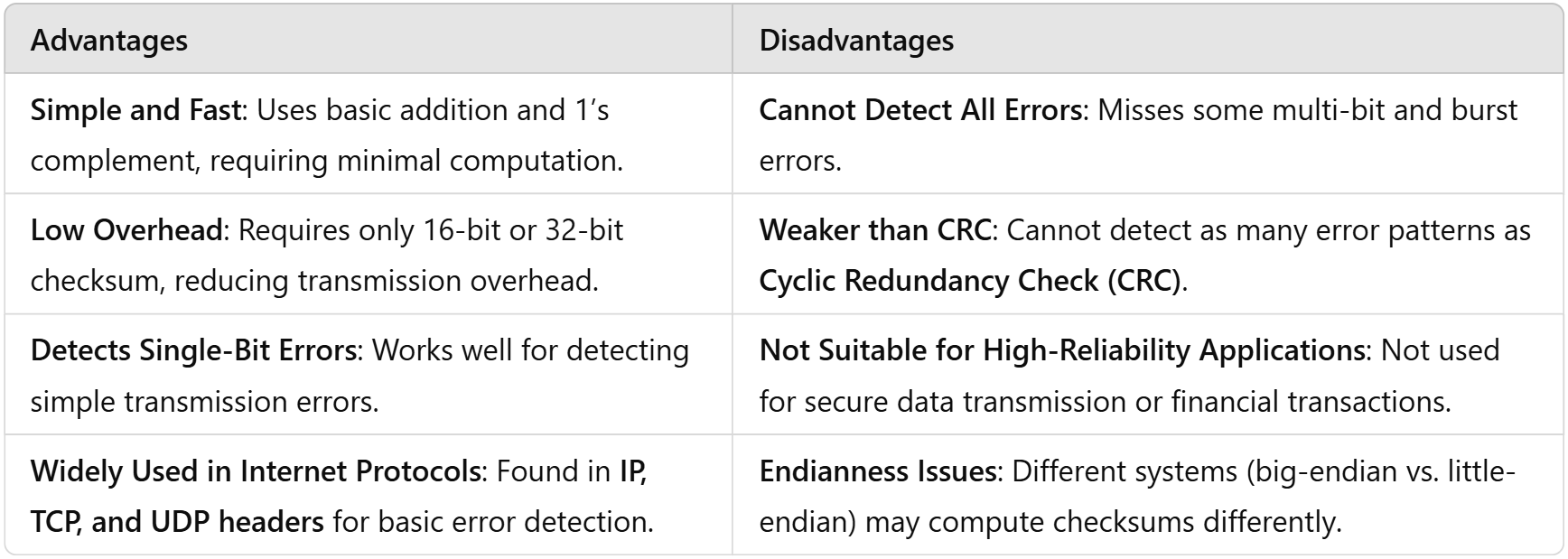
In conclusion, our experiment with checksum and CRC implementations demonstrates that while both methods provide error detection capabilities, CRC offers superior error detection performance, particularly for burst errors and specific bit patterns, due to its polynomial division approach rather than simple addition. The practical implementation confirms theoretical advantages of CRC over checksum, with CRC being able to detect all burst errors of length up to the degree of the generator polynomial and all odd-numbered bit errors.

**Post Lab Questions:**

1. **Discuss about the rules for choosing a CRC generator.**

* Must be of Degree n – The generator polynomial must match the CRC bit size.
* Must be a Prime Polynomial – It should not be factored into smaller polynomials.
* Must Include the Highest-Order Term (xn) and the Constant Term (1) – Ensures all bits are involved.
* Should Detect Common Error Patterns – Must detect single-bit, double-bit, and burst errors efficiently.
* Avoid Simple Patterns (xn+1) – These fail to detect certain error types.
* Must be Standardized for Compatibility – Should be widely used in communication protocols.

1. **State the advantages and disadvantages of Internet Checksum.**

****

**Date: \_\_\_\_\_\_\_\_\_\_ Signature of Faculty In-charge**