# Experiment No.:4



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



## Expected Outcome of Experiment: CO:



**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

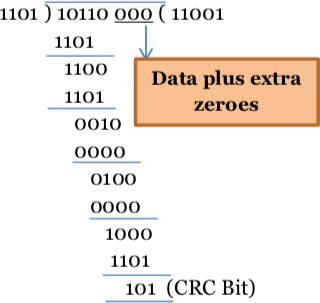
Assume that –

1. data is 10110.
2. The 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**

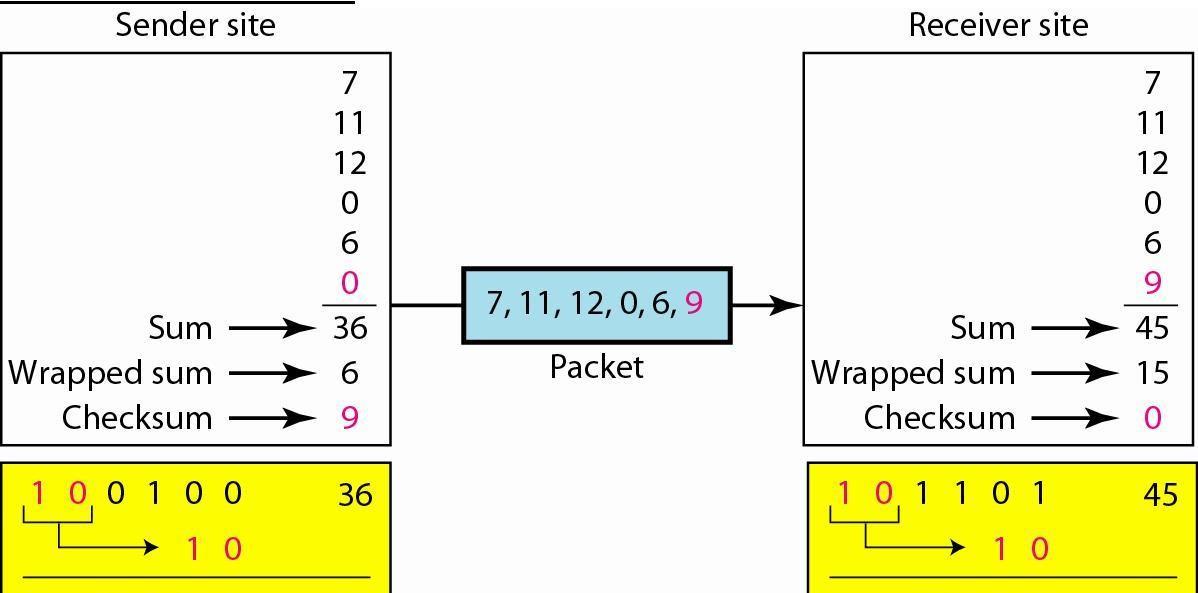
The CRC bits are calculated to be different. Thus, there is an error detected.

# 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:

Page | mester V July-Dec 2024

## 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.

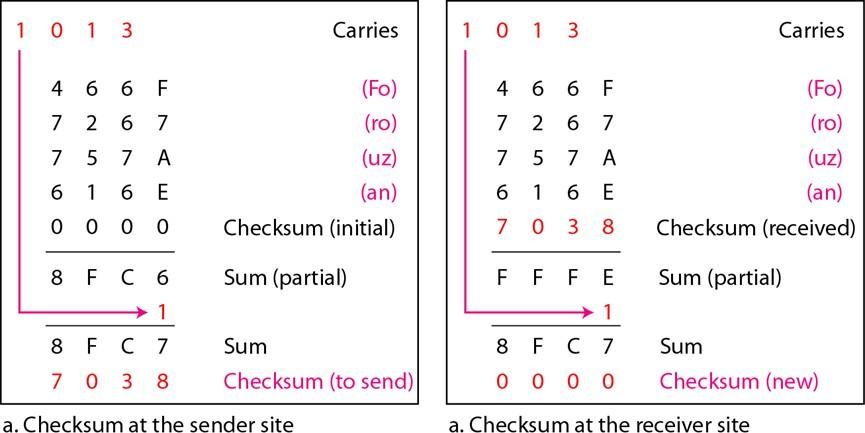
0100 F203 F4F5 F6F7 210E

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:**

(#include <iostream>

#include <string>

using namespace std;

string Xor(string a, string b) { string result = "";

int n = b.length();

for (int i = 1; i < n; i++) { if (a[i] == b[i])

result += "0"; else

result += "1";

}

return result;

}

string Mod2Div(string dividend, string divisor) { int pick = divisor.length();

string tmp = dividend.substr(0, pick); int n = dividend.length();

while (pick < n) {

if (tmp[0] == '1')

tmp = Xor(divisor, tmp) + dividend[pick]; else

tmp = Xor(string(pick, '0'), tmp) + dividend[pick];

pick += 1;

}

if (tmp[0] == '1')

tmp = Xor(divisor, tmp); else

tmp = Xor(string(pick, '0'), tmp);

return tmp;

}

void EncodeData(string data, string key) {



int keylen = key.length();

string appended\_data = data + string(keylen - 1, '0');

string remainder = Mod2Div(appended\_data, key);

string codeword = data + remainder;

cout << "Remainder: " << remainder << endl;

cout << "Encoded Data (Data + Remainder): " << codeword << endl << endl;

}

|

void Receiver(string data, string key) {

string currxor = Mod2Div(data.substr(0, key.length()), key); int curr = key.length();

while (curr != data.length()) {

if (currxor.length() != key.length()) { currxor += data[curr++];

} else {

currxor = Mod2Div(currxor, key);

}

}

if (currxor.length() == key.length()) { currxor = Mod2Div(currxor, key);

}

if (currxor.find('1') != string::npos) {

cout << "There is some error in data." << endl;

} else {

cout << "Correct message received." << endl;

}

}

int main() {

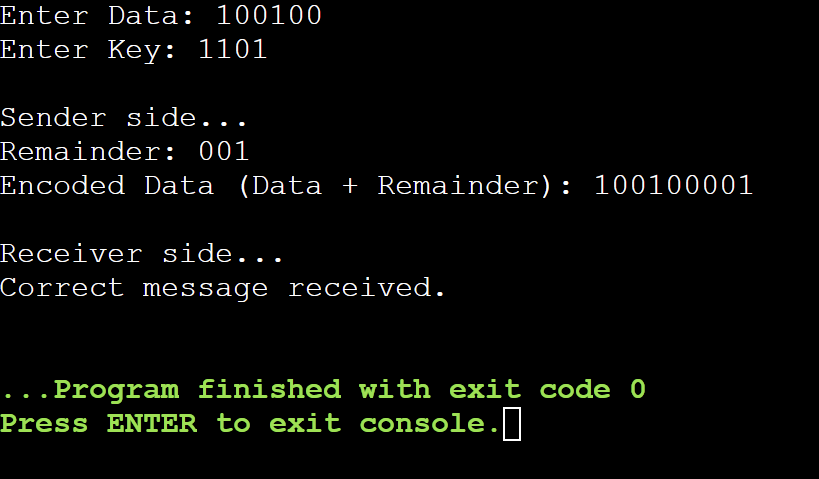
string data, key;

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

cout << "Enter Key: "; cin >> key;



## Output







using namespace std;

int generateComplement(int checkSum) { checkSum = 0xFFFF - checkSum; return checkSum;

}

int generateCheckSum(string s) { string hexadecimalValue; int x, checkSum = 0;

for (int i = 0; i < s.length() - 2; i += 2) { x = (int)(s[i]);

stringstream stream; stream << hex << x;

hexadecimalValue = stream.str();

x = (int)(s[i + 1]);

stream.str(""); stream.clear(); stream << hex << x;

hexadecimalValue += stream.str();

cout << s[i] << s[i + 1] << " : " << hexadecimalValue << endl;

x = stoi(hexadecimalValue, nullptr, 16); checkSum += x;

}

if (s.length() % 2 == 0) {

x = (int)(s[s.length() - 2]); stringstream stream;

stream << hex << x; hexadecimalValue = stream.str();

x = (int)(s[s.length() - 1]); stream.str(""); stream.clear();

stream << hex << x; hexadecimalValue += stream.str();



cout << s[s.length() - 2] << s[s.length() - 1] << " : " << hexadecimalValue << endl;

x = stoi(hexadecimalValue, nullptr, 16);

} else {

x = (int)(s[s.length() - 1]); stringstream stream;

stream << hex << x;

hexadecimalValue = "00" + stream.str();

cout << s[s.length() - 1] << " : " << hexadecimalValue << endl;

x = stoi(hexadecimalValue, nullptr, 16);

}

checkSum += x;

hexadecimalValue = to\_string(checkSum);

if (hexadecimalValue.length() > 4) {

int carry = stoi(hexadecimalValue.substr(0, 1), nullptr, 16); hexadecimalValue = hexadecimalValue.substr(1, 4);

checkSum = stoi(hexadecimalValue, nullptr, 16); checkSum += carry;

}

checkSum = generateComplement(checkSum);

return checkSum;

}

void receive(string s, int checkSum) {

int generatedChecksum = generateCheckSum(s); generatedChecksum = generateComplement(generatedChecksum);

int syndrome = generatedChecksum + checkSum; syndrome = generateComplement(syndrome);

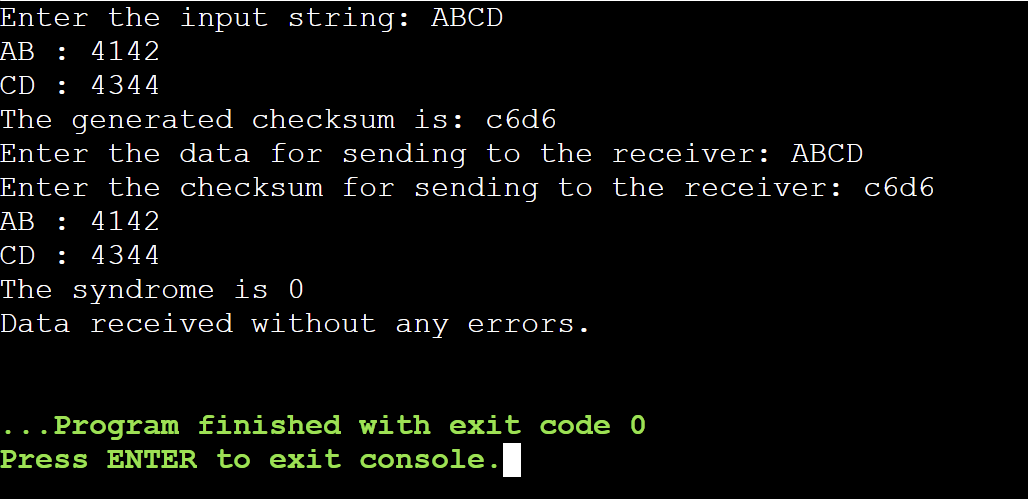
cout << "The syndrome is " << hex << syndrome << endl;

if (syndrome == 0) {

cout << "Data received without any errors." << endl;



Output



## CONCLUSION:

We were successfully able to implement the CRC code using XOR division in a Java code and were able to get correct output. We understood the concept, the uses and the advantages and disadvantages of CRC’s.

**Post Lab Questions**

1. Discuss about the rules for choosing a CRC generator

## Rules for Choosing a CRC Generator

Choosing an effective Cyclic Redundancy Check (CRC) generator polynomial is crucial for reliable error detection in data transmission. Here are the key considerations:

* 1. **Degree of the Polynomial**: The polynomial's degree determines the number of CRC bits. Common degrees are CRC-8, CRC-16, and CRC-32. Higher degrees improve error detection.
  2. **Constant Term**: The polynomial must include a constant term (i.e., the lowest-order term must be 1). This ensures the detection of certain errors, especially those involving all zeroes.
  3. **Odd Number of Terms**: Polynomials with an odd number of terms are more effective in detecting both single-bit and odd-numbered errors.
  4. **Error Detection**: The chosen polynomial should detect all single-bit and most double-bit errors, and should effectively handle burst errors up to its degree plus 1.
  5. **Burst Error Length**: The polynomial should be able to detect burst errors up to the length of the polynomial (e.g., a degree-32 polynomial detects burst errors of up to 33 bits).
  6. **Application-Specific Choices**: Different communication protocols use standardized polynomials, such as CRC-16 for Bluetooth and CRC-32 for Ethernet.
  7. **Primitive Polynomials**: Primitive polynomials are preferred for maximum error detection capabilities since they cover a wider range of error patterns.
  8. **Efficiency**: Simpler polynomials with fewer terms may offer faster computations but must still provide adequate error detection for the given application.

## Example Polynomials

* + - **CRC-8**: G(x)=x8+x2+x+1G(x) = x^8 + x^2 + x + 1G(x)=x8+x2+x+1 (used in SMBus)
    - **CRC-16**: G(x)=x16+x15+x2+1G(x) = x^{16} + x^{15} + x^2 + 1G(x)=x16+x15+x2+1

(used in Modbus)

* + - **CRC-32**: G(x)=x32+x26+⋯+1G(x) = x^{32} + x^{26} + \dots + 1G(x)=x32+x26+⋯+1

(used in Ethernet)

The right polynomial balances error detection with computational efficiency, depending on the specific application and its error characteristics.

.

1. State the advantages and disadvantages of Internet Checksum.

## Advantages of Internet Checksum

* 1. **Simplicity**: The Internet Checksum algorithm is easy to implement and computationally inexpensive, making it suitable for low-resource environments.
  2. **Fast Execution**: Due to its simple addition-based operation, checksum calculations can be performed quickly, minimizing overhead in real-time systems.
  3. **Low Resource Requirements**: It requires minimal processing power and memory, making it ideal for devices with limited computational capabilities.
  4. **Wide Adoption**: Internet Checksum is widely used in networking protocols like TCP/IP, ensuring compatibility across different systems and networks.
  5. **Basic Error Detection**: It effectively detects common types of data corruption, such as single-bit errors and simple patterns of bit errors.

## Disadvantages of Internet Checksum

1. **Limited Error Detection**: Internet Checksum is less robust compared to more advanced error detection techniques, such as CRC. It may fail to detect more complex error patterns like multi-bit errors.
2. **No Correction Capability**: The checksum only identifies errors and does not provide any mechanism to correct them.
3. **Vulnerability to Burst Errors**: It is less effective at detecting burst errors, where multiple bits are altered in sequence.
4. **Collision Risk**: Two different blocks of data can produce the same checksum, leading to undetected errors (known as "collisions").

In summary, while Internet Checksum is lightweight and widely used, its error detection capabilities are limited compared to more sophisticated techniques.

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