Batch: A1	**Roll No.: 16010123012**
Experiment / assignment / tutorial No.: 04	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Experiment No.: 04

TITLE: Implementation of Hamming Code & CheckSum for Computer Networks

AIM: To implement Layer 2 Error detection schemes: Hamming Code & CheckSum.

Expected Outcome of Experiment:

C02: Demonstrate Data Link Layer, MAC layer technologies & protocols and implement the functionalities like error control, flow control

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.

Hamming Code:
Hamming Code is an error-detection and error-correction code developed by Richard W. Hamming in 1950. It is widely used in digital communication and computer memory to detect and correct single-bit errors. The primary goal of Hamming Code is to ensure data integrity in systems where data corruption can occur.

Hamming Code adds redundant parity bits to data bits to create a code word. These parity bits are placed at specific positions in the code word and are calculated in such a way that if a single-bit error occurs during transmission or storage, it can be both detected and corrected.
Key Concepts

- Data bits (m): The original bits of information.
- Parity bits (r): Bits added to the data to check for errors.
- Code word: The combination of data and parity bits.

The position of each parity bit is a power of two (1, 2, 4, 8, ...), and each parity bit checks specific bits in the code word.

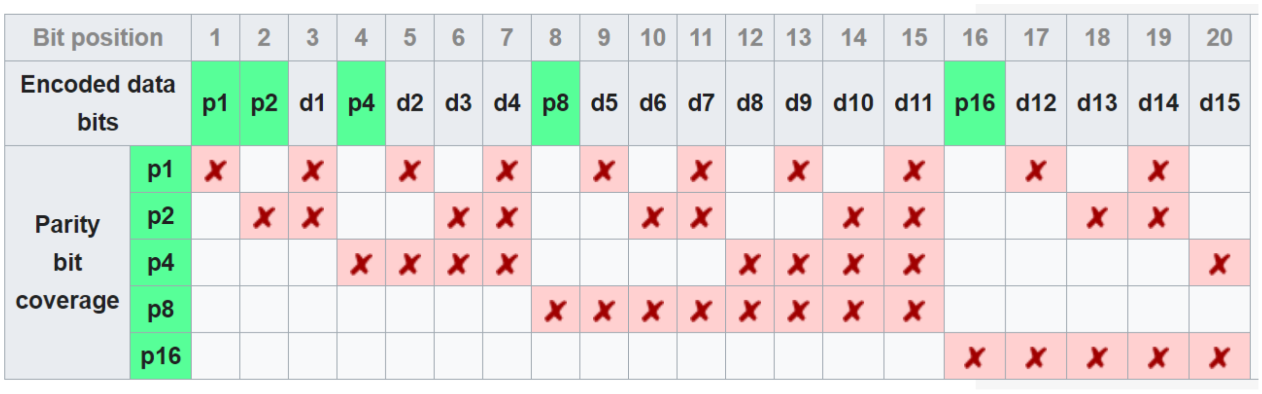
Determining the Number of Parity Bits
To determine the number of parity bits r required for m data bits, the following inequality must be satisfied:
$2^r \geq n+1$
Where,

- m data bits
- r parity bits
- n=m+r (coded bits)
- 1 bit for error detection (including no error)

Constructing a Hamming Code
Determine the number of parity bits needed.
Place the parity bits at positions 2^0, 2^1, 2^2, etc.
Insert the data bits in the remaining positions.
Calculate the parity bits using even (or odd) parity.
Transmit the full code word.

To Calculate Parity bits:
Step 1: Write the bit positions starting from 1 in binary form (1, 10, 11, 100, etc).
Step 2: All the bit positions that are a power of 2 are marked as parity bits (1, 2, 4, 8, etc).
Step 3: All the other bit positions are marked as data bits.
Step 4: Each data bit is included in a unique set of parity bits, as determined its bit position in binary form:
Parity bit 1 covers all the bits positions whose binary representation includes a 1 in the least significant position (1, 3, 5, 7, 9, 11, etc).
Parity bit 2 covers all the bits positions whose binary representation includes a 1 in the second position from the least significant bit (2, 3, 6, 7, 10, 11, etc).
Parity bit 4 covers all the bits positions whose binary representation includes a 1 in the third position from the least significant bit (4–7, 12–15, 20–23, etc).
Parity bit 8 covers all the bits positions whose binary representation includes a 1 in the fourth position from the least significant bit bits (8–15, 24–31, 40–47, etc).
In general, each parity bit covers all bits where the bitwise AND of the parity position and the bit position is non-zero.
Step 5: Since we check for even parity set a parity bit to 1 if the total number of ones in the positions it checks is odd. Set a parity bit to 0 if the total number of ones in the positions it checks is even.

Following is the structure of Hamming code with the position of parity and data bits from LB to MSB.

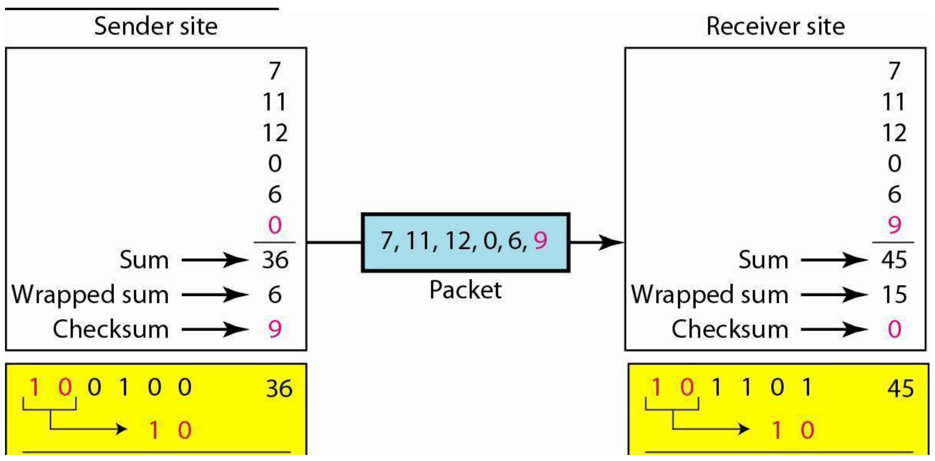


Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15
Parity bit coverage	p1	✗		✗		✗		✗		✗		✗		✗		✗		✗		✗	
	p2		✗	✗			✗	✗			✗	✗			✗	✗			✗	✗	
	p4				✗	✗	✗	✗					✗	✗	✗	✗					✗
	p8								✗	✗	✗	✗	✗	✗	✗	✗					
	p16																✗	✗	✗	✗	✗

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:



0 1 1 0 6 **1 1 1 1 15**

1 0 0 1 9 checkSum **0 0 0 0 0**

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.

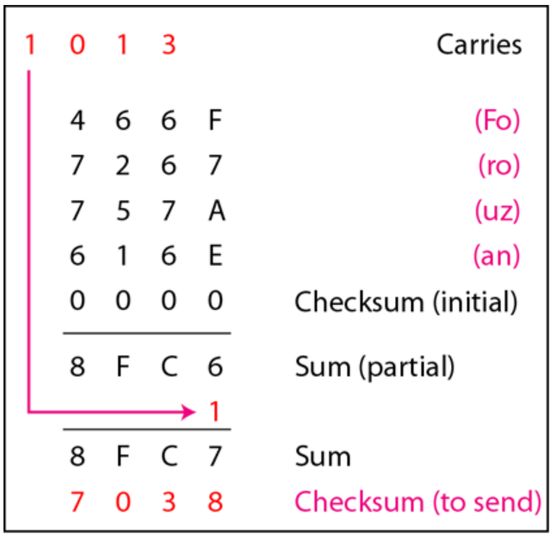
The first step is to form 16-bit words.
0100 F203 F4F5 F6F7 210E

The second step is to calculate the sum using 32-bits.
0100 + F203 + F4F5 + F6F7 + 210E = 0002 FFFD

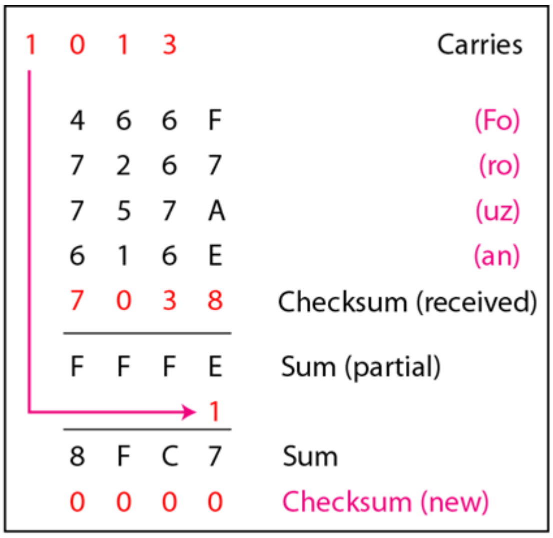
The third step is to add the carries (0002) to the 16-bit sum.
FFFD + 0002 = FFFF which means that no error was detected.

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

Example:



a. Checksum at the sender site



a. Checksum at the receiver site

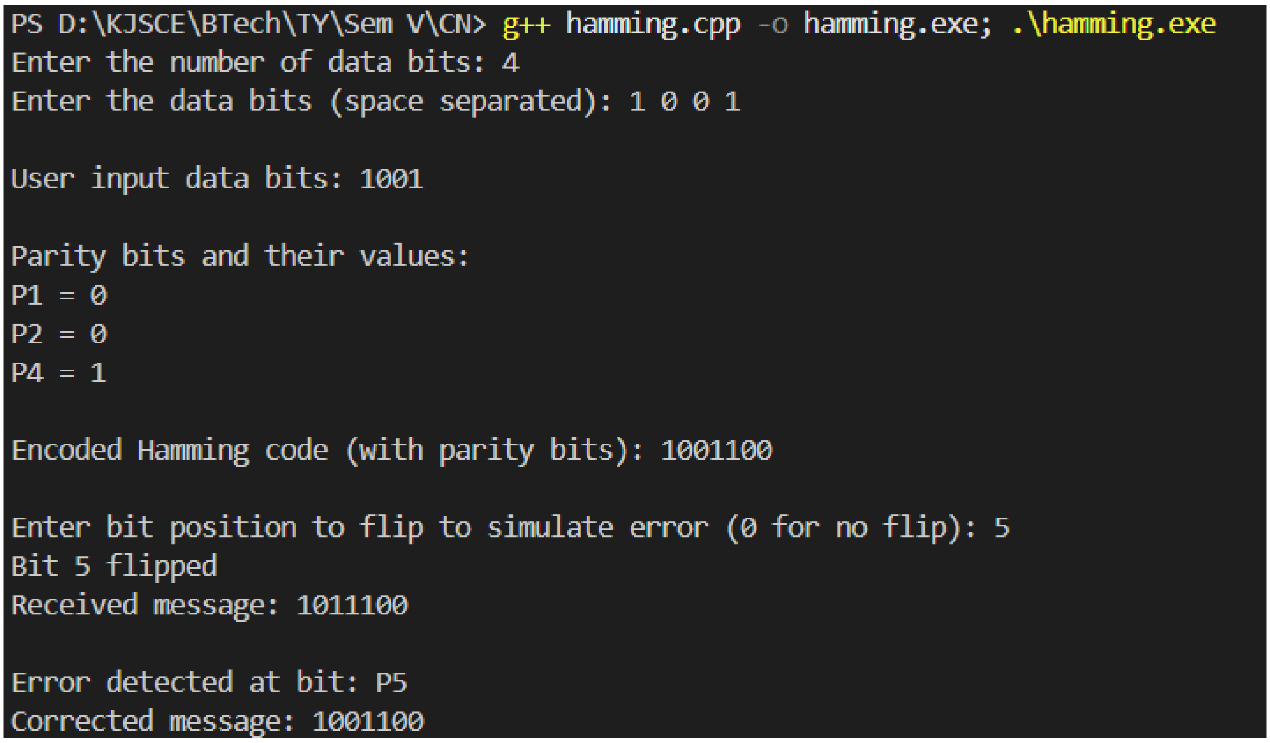
<u>IMPLEMENTATION</u>: (Attach the code and output)

a) Hamming code generation and Reception with error. Perform error detection and correction.

```
#include <bits/stdc++.h>
using namespace std;

int Parity(int m)
{
  int r = 0;
  while ((1 << r) < (m + r + 1))
  {
    r++;
  }
  return r;
}

bool isPowerOfTwo(int x)
{
  return (x & (x - 1)) == 0;
}

int main()
{
  cout << "Enter the number of data bits: ";
  int m;
  cin >> m;
```

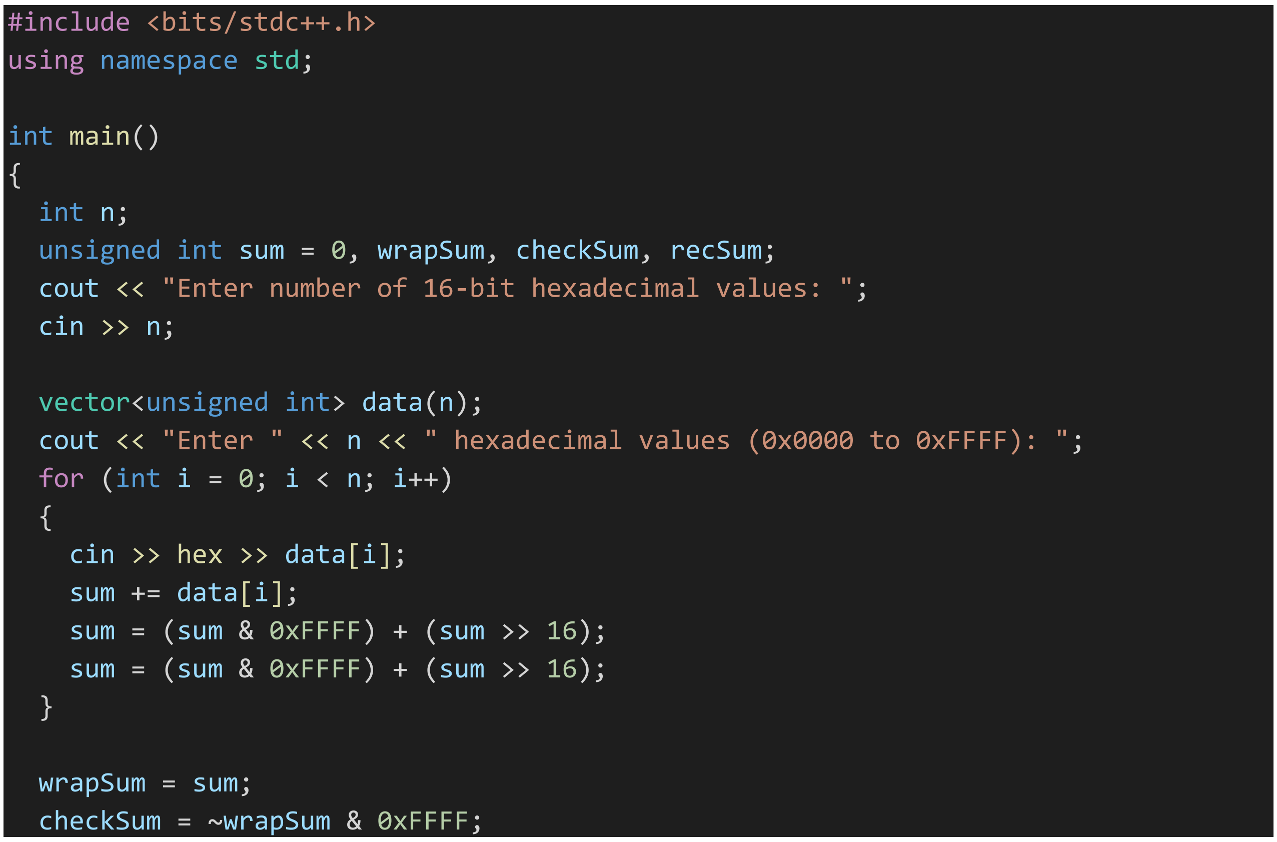
```
vector<int> data(m);
cout << "Enter the data bits (space separated): ";
for (int i = 0; i < m; i++)
{
  cin >> data[i];
}

cout << "\nUser input data bits: ";
for (int i = 0; i < m; i++)
{
  cout << data[i];
}
cout << endl;

int r = Parity(m);
int n = m + r;

vector<int> encoded(n + 1, 0);
for (int i = 1, j = 0; i <= n; i++)
{
  if (isPowerOfTwo(i))
  {
    encoded[i] = 0;
  }
  else
  {
    encoded[i] = data[j++];
  }
}

vector<int> parityValues(r);
for (int i = 0; i < r; i++)
{
  int parityPos = 1 << i;
  int parity = 0;
  for (int j = parityPos; j <= n; j++)
  {
    if (j & parityPos)
    {
      parity ^= encoded[j];
    }
  }
  encoded[parityPos] = parity;
  parityValues[i] = parity;
}

cout << "\nParity bits and their values:\n";
for (int i = 0; i < r; i++)
{
  cout << "P" << (1 << i) << " = " << parityValues[i] << endl;
}

cout << "\nEncoded Hamming code (with parity bits): ";
for (int i = n; i >= 1; i--)
{
  cout << encoded[i];
}
cout << endl;

int flipBit;
cout << "\nEnter bit position to flip to simulate error (0 for no flip): ";
cin >> flipBit;

vector<int> received = encoded;
if (flipBit != 0)
{
  if (flipBit >= 1 && flipBit <= n)
  {
    received[flipBit] ^= 1;
    cout << "Bit " << flipBit << " flipped\n";
  }
  else
  {
    cout << "Invalid bit position to flip, No bit flipped\n";
  }
}
else
{
  cout << "No bit flipped\n";
}

cout << "Received message: ";
for (int i = n; i >= 1; i--)
{
  cout << received[i];
}
```

```cpp
  cout << endl;

  int errorPos = 0;
  for (int i = 0; i < r; i++)
  {
    int parityPos = 1 << i;
    int parity = 0;
    for (int j = parityPos; j <= n; j++)
    {
      if (j & parityPos)
      {
        parity ^= received[j];
      }
    }
    if (parity != 0)
    {
      errorPos += parityPos;
    }
  }

  if (errorPos == 0)
  {
    cout << "\nNo error detected in the received message\n";
  }
  else
  {
    cout << "\nError detected at bit: P" << errorPos << endl;
    received[errorPos] ^= 1;

    cout << "Corrected message: ";
    for (int i = n; i >= 1; i--)
    {
      cout << received[i];
    }
    cout << endl;
  }

  return 0;
}
```

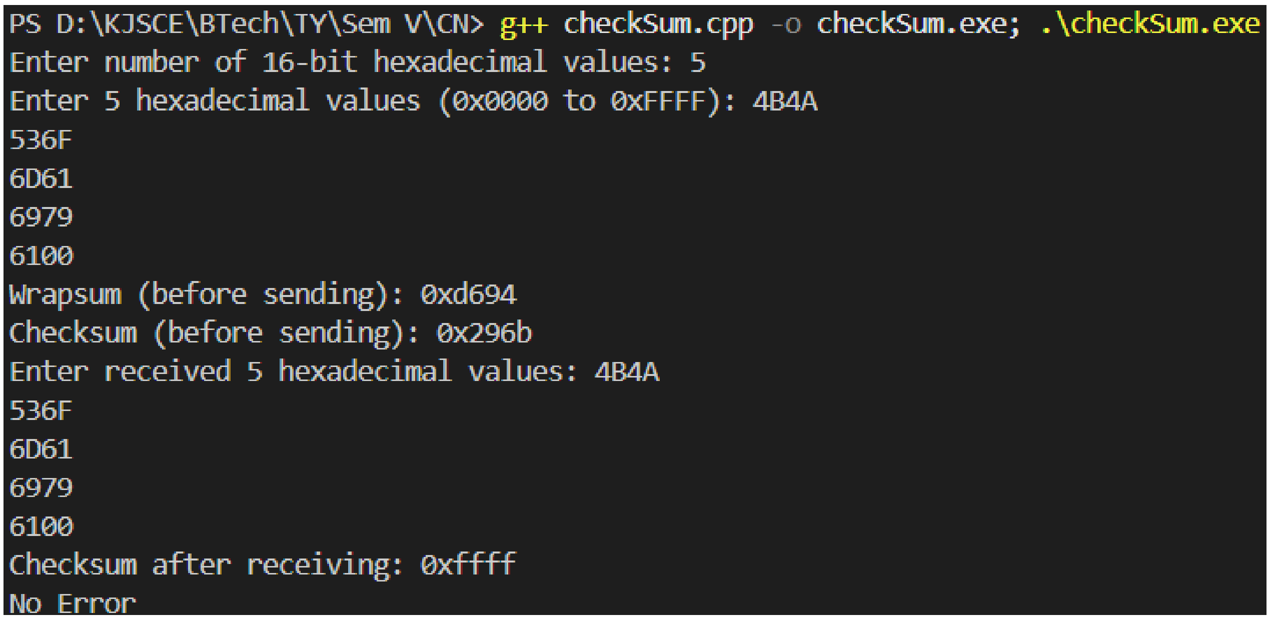


```
PS D:\KJSCE\BTech\TY\Sem V\CN> g++ hamming.cpp -o hamming.exe; .\hamming.exe
Enter the number of data bits: 4
Enter the data bits (space separated): 1 0 0 1

User input data bits: 1001

Parity bits and their values:
P1 = 0
P2 = 0
P4 = 1

Encoded Hamming code (with parity bits): 1001100

Enter bit position to flip to simulate error (0 for no flip): 5
Bit 5 flipped
Received message: 1011100

Error detected at bit: P5
Corrected message: 1001100
```

b) Implement Internet CheckSum.



```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
  int n;
  unsigned int sum = 0, wrapSum, checkSum, recSum;
  cout << "Enter number of 16-bit hexadecimal values: ";
  cin >> n;

  vector<unsigned int> data(n);
  cout << "Enter " << n << " hexadecimal values (0x0000 to 0xFFFF): ";
  for (int i = 0; i < n; i++)
  {
    cin >> hex >> data[i];
    sum += data[i];
    sum = (sum & 0xFFFF) + (sum >> 16);
    sum = (sum & 0xFFFF) + (sum >> 16);
  }

  wrapSum = sum;
  checkSum = ~wrapSum & 0xFFFF;
  cout << "Wrapsum (before sending): 0x" << hex << setw(4) << setfill('0') <<
wrapSum << endl;
  cout << "CheckSum (before sending): 0x" << hex << setw(4) << setfill('0') <<
checkSum << endl;

  recSum = 0;
  cout << "Enter received " << n << " hexadecimal values: ";
  for (int i = 0; i < n; i++)
  {
    unsigned int val;
    cin >> hex >> val;
    recSum += val;
    recSum = (recSum & 0xFFFF) + (recSum >> 16);
    recSum = (recSum & 0xFFFF) + (recSum >> 16);
  }
  recSum += checkSum;
  recSum = (recSum & 0xFFFF) + (recSum >> 16);
  recSum = (recSum & 0xFFFF) + (recSum >> 16);

  cout << "CheckSum after receiving: 0x" << hex << setw(4) << setfill('0') <<
recSum << endl;
  if (recSum == 0xFFFF)
  {
    cout << "No Error" << endl;
  }
  else
  {
    cout << "Error Detected" << endl;
  }
}
```



```
PS D:\KJSCE\BTech\TY\Sem V\CN> g++ checkSum.cpp -o checkSum.exe; .\checkSum.exe
Enter number of 16-bit hexadecimal values: 5
Enter 5 hexadecimal values (0x0000 to 0xFFFF): 4B4A
536F
6D61
6979
6100
Wrapsum (before sending): 0xd694
Checksum (before sending): 0x296b
Enter received 5 hexadecimal values: 4B4A
536F
6D61
6979
6100
Checksum after receiving: 0xffff
No Error
```

CONCLUSION:

I have successfully completed the experiment on the implementation of Hamming Code and CheckSum for error detection and correction in computer networks. Through this experiment, I understood how Hamming Code can detect and correct single-bit errors by using parity bits, thereby ensuring reliable data transmission. I also learned how the Internet CheckSum helps in detecting errors during data transfer by verifying packet integrity.

Post Lab Questions

1. **Discuss about the rules for choosing a CRC generator.**
   When selecting a generator polynomial (G(x)) for Cyclic Redundancy Check (CRC), certain rules should be followed to maximize error detection capability:
   - The polynomial should have at least two terms. It must include both the highest-degree term and the constant term (to detect single-bit errors).
   - It should be able to detect all single-bit, double-bit errors, provided the generator does not divide xk+1x^k + 1xk+1.
   - It should detect all odd numbers of errors if it contains the factor (x+1)(x + 1)(x+1).
   - It should detect burst errors up to the length of the generator polynomial.

2. **State the advantages and disadvantages of Internet CheckSum.**
   The Internet CheckSum has the advantage of being simple, fast, and easy to implement, which is why it is widely used in TCP/IP protocols. However, its limitation is that it can only detect errors, not correct them, and it may fail to identify some complex error patterns, making it less reliable than CRC.

Date : 21 / 08 / 2025 **Signature of Faculty In-charge**