Batch: D-2	**Roll No.: 16010123325**
Experiment / assignment / tutorial No.___4____	
Grade: AA / AB / BB / BC / CC / CD /DD	

Experiment No.:4

TITLE: Implementation of Hamming Code & Checksum for Computer Networks

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

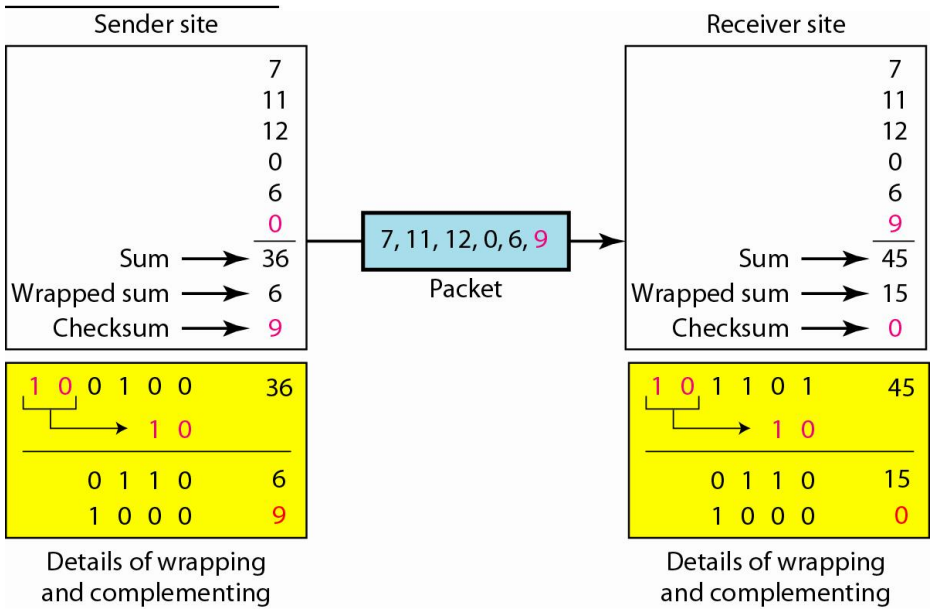
Following is the structure of Hamming code with the position of parity and data bits from LB to MSB.

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:



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:

1	0	1	3	Carries
4	6	6	F	(Fo)
7	2	6	7	(ro)
7	5	7	A	(uz)
6	1	6	E	(an)
0	0	0	0	Checksum (initial)
8	F	C	6	Sum (partial)
			1	
8	F	C	7	Sum
7	0	3	8	Checksum (to send)

a. Checksum at the sender site

1	0	1	3	Carries
4	6	6	F	(Fo)
7	2	6	7	(ro)
7	5	7	A	(uz)
6	1	6	E	(an)
7	0	3	8	Checksum (received)
F	F	F	E	Sum (partial)
			1	
8	F	C	7	Sum
0	0	0	0	Checksum (new)

a. Checksum at the receiver site

IMPLEMENTATION: (Attach the code and output)

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

```cpp
// Function to calculate number of parity bits
int parityBits(int k) {
    int r = 0;
    while (pow(2, r) < (k + r + 1))
        r++;
    return r;
}

// Function to calculate and insert parity bit at position pb
void put(vector<char>& w, int pb) {
    int ans = 0;
    cout << "Calculating parity bit at position " << pb << " → covers positions: ";
    for (int i = 0; i < w.size(); i++) {
        if ((i + 1) & pb) != 0) {
            ans ^= (w[i] - '0');
            cout << (i + 1) << " ";
        }
    }
    w[pb - 1] = (char)(ans + '0');
    cout << " | Parity value: " << w[pb - 1] << endl;
}

// Function to encode data word
void encode() {
    string x;
    cout << "Enter data word: ";
    cin >> x;

    reverse(x.begin(), x.end()); // reverse for easier bit indexing
    vector<char> cw(x.begin(), x.end());

    int k = cw.size();
    int r = parityBits(k);
    int n = k + r;

    cout << "Data bits (reversed for processing): " << x << endl;
    cout << "Number of parity bits needed: " << r << endl;
    cout << "Total length (n): " << n << endl;

    vector<char> w(n, '0');

    // Mark parity bit positions
    int c = 0;
    for (int i = 0; i < n; i++) {
        int pb = pow(2, c);
        if (pb - 1 == i) {
            cout << "Placing parity bit at position " << (i + 1) << endl;
            w[i] = '0'; // placeholder
            c++;
        }
    }

    // Fill data bits
    c = 0;
    for (int i = 0; i < n; i++) {
        int logVal = (int) log2(i + 1);
        int pb = pow(2, logVal);
        if ((i + 1) != pb) {
            w[i] = cw[c];
            cout << "Placing data bit " << cw[c] << " at position " << (i + 1) << endl;
            c++;
        }
    }

    // Calculate parity bits
    for (int i = 0; i < r; i++) {
        int pb = pow(2, i);
```

```cpp
    cout << "\nFinal Hamming code: ";
    for (int i = n - 1; i >= 0; i--) cout << w[i];
    cout << endl;
}

// Function to check for errors in received Hamming code
void checkError() {
    string code;
    cout << "Enter received Hamming code: ";
    cin >> code;

    reverse(code.begin(), code.end());
    vector<char> w(code.begin(), code.end());

    int n = w.size();
    int r = 0;
    while (pow(2, r) < n + 1) r++;

    cout << "Checking code of length " << n << " with " << r << " parity bits..." << endl;

    int errorPos = 0;
    for (int i = 0; i < r; i++) {
        int pb = pow(2, i);
        int parity = 0;
        cout << "Checking parity bit at position " << pb << " → covers positions: ";
        for (int j = 0; j < n; j++) {
            if ((j + 1) & pb) != 0) {
                parity ^= (w[j] - '0');
                cout << (j + 1) << " ";
            }
        }
        cout << " | Result: " << parity << endl;
        if (parity != 0)
            errorPos += pb;
    }

    if (errorPos == 0) {
        cout << "\nNo error detected in the code." << endl;
    } else if (errorPos > n) {
        cout << "\nError detected outside code length – possibly multiple errors." << endl;
    } else {
        cout << "\nError detected at position: " << errorPos << endl;
        w[errorPos - 1] = (w[errorPos - 1] == '0') ? '1' : '0';
        cout << "Corrected code: ";
        for (int i = n - 1; i >= 0; i--)
            cout << w[i];
        cout << endl;
    }
}

int main() {
    int choice;
    cout << "Choose operation:\n1. Encode data word\n2. Check Hamming code for errors\n";
    cin >> choice;

    switch (choice) {
        case 1:
            encode();
            break;
        case 2:
            checkError();
            break;
        default:
            cout << "Invalid choice" << endl;
            break;
    }
```

Output-

```
Choose operation:
1. Encode data word
2. Check Hamming code for errors
1
Enter data word: 110
Data bits (reversed for processing): 011
Number of parity bits needed: 3
Total length (n): 6
Placing parity bit at position 1
Placing parity bit at position 2
Placing parity bit at position 4
Placing data bit 0 at position 3
Placing data bit 1 at position 5
Placing data bit 1 at position 6
Calculating parity bit at position 1 → covers positions: 1 3 5   | Parity value: 1
Calculating parity bit at position 2 → covers positions: 2 3 6   | Parity value: 1
Calculating parity bit at position 4 → covers positions: 4 5 6   | Parity value: 0

Final Hamming code: 110011
```

```
Choose operation:
1. Encode data word
2. Check Hamming code for errors
2
Enter received Hamming code: 110010
Checking code of length 6 with 3 parity bits...
Checking parity bit at position 1 → covers positions: 1 3 5   | Result: 1
Checking parity bit at position 2 → covers positions: 2 3 6   | Result: 0
Checking parity bit at position 4 → covers positions: 4 5 6   | Result: 0

Error detected at position: 1
Corrected code: 110011
```

b) Implement Internet Checksum.

```cpp
unsigned short addOnesComplement(unsigned short a, unsigned short b) {
    unsigned int sum = a + b;
    while (sum >> 16) { // if carry exists
        sum = (sum & 0xFFFF) + (sum >> 16);
    }
    return sum & 0xFFFF;
}

// Calculate sender checksum
unsigned short calculateSenderChecksum(const vector<unsigned short>& data) {
    unsigned short sum = 0;
    for (auto word : data) {
        sum = addOnesComplement(sum, word);
    }
    return ~sum & 0xFFFF;
}

// Calculate receiver checksum
unsigned short calculateReceiverChecksum(const vector<unsigned short>& data) {
    unsigned short sum = 0;
    for (auto word : data) {
        sum = addOnesComplement(sum, word);
    }
    return ~sum & 0xFFFF;
}

// Print 16-bit value as 4-digit hex
string hex16(unsigned short x) {
    stringstream ss;
    ss << uppercase << hex << setw(4) << setfill('0') << x;
    return ss.str();
}

int main() {
    int n;
    cout << "Enter number of 16-bit words: ";
    cin >> n;
    if (n <= 0) {
        cout << "Number of words must be positive." << endl;
        return 0;
    }

    vector<unsigned short> data(n);
    cout << "Enter " << n << " words in hex (e.g., ABCD), separated by space or newline:\n";
    for (int i = 0; i < n; i++) {
        string token;
        cin >> token;
        if (token.size() >= 2 && (token.substr(0,2) == "0x" || token.substr(0,2) == "0X")) token = token.substr(2);
        data[i] = static_cast<unsigned short>(stoul(token, nullptr, 16));
    }

    // Sender side
    unsigned short checksum = calculateSenderChecksum(data);
    cout << "\n--- Sender Side ---\n";
    cout << "Calculated Checksum: " << hex16(checksum) << endl;

    cout << "Transmitted Codeword: ";
    for (auto w : data) cout << hex16(w) << " ";
    cout << hex16(checksum) << endl;

    // Receiver side (No Error)
    cout << "\n--- Receiver Side (No Error Case) ---\n";
    vector<unsigned short> recvData = data;
    recvData.push_back(checksum);

    unsigned short recvChecksum = calculateReceiverChecksum(recvData);
    cout << "Receiver Calculated Checksum: " << hex16(recvChecksum) << endl;
    cout << ((recvChecksum == 0x0000) ? "No Error Detected\n" : "Error Detected!\n");

    // Receiver side (Error Introduced)
    cout << "\n--- Receiver Side (Error Introduced) ---\n";
```

```cpp
    recvData[0] ^= 0x0001; // flip LSB of first word

    cout << "Corrupted Codeword: ";
    for (auto w : recvData) cout << hex16(w) << " ";
    cout << endl;

    recvChecksum = calculateReceiverChecksum(recvData);
    cout << "Receiver Calculated Checksum: " << hex16(recvChecksum) << endl;
    cout << ((recvChecksum == 0x0000) ? "No Error Detected\n" : "Error Detected!\n");

    return 0;
}
```

Output-

CONCLUSION:

The above experiment highlights error detection and error correction mechanisms using methods like Checksum and Hamming Code.

Post Lab Questions

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

A **Cyclic Redundancy Check (CRC)** generator polynomial (also called the **divisor**) is a key factor in detecting transmission errors. The choice of generator polynomial follows certain rules to ensure good error detection capabilities:

Rules:

1. **The generator polynomial (G(x)) must have at least two terms**
   → Ensures detection of all single-bit errors.
   (Example: $x^3 + 1$ is invalid; $x^3 + x + 1$ is valid)
2. **The coefficient of the highest and lowest degree terms must be 1**
   → Guarantees that both the first and last bits of the codeword are involved in error checking.
3. **G(x) should not divide ($x^n + 1$) for small n**
   → Prevents the generator from missing certain error patterns (like repetitive errors).
4. **For good detection properties:**
   - Detect **all single-bit errors**, if G(x) has more than one term.
   - Detect **all double-bit errors**, if G(x) does not divide ($x^k + 1$) for any $k \leq n$.
   - Detect **any odd number of errors**, if G(x) contains the factor (x + 1).
   - Detect **burst errors** up to the degree of the polynomial.

2. State the advantages and disadvantages of Internet Checksum.

Advantages:

1. **Simple and fast:** Uses only basic addition and 1's complement arithmetic, making it easy to implement in software.
2. **Low overhead:** Adds only a small 16-bit checksum field to the message.
3. **Widely supported:** Commonly used in Internet protocols like IP, TCP, and UDP.

Disadvantages:

1. **Limited error detection:** Ineffective for many multi-bit or burst errors.
2. **Unsuitable for noisy channels:** Less reliable than CRC in detecting frequent transmission errors.
3. **May miss certain error patterns:** Some bit errors can cancel each other out during 1's complement addition, leading to undetected errors.

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