**10. WAP to Implement Error Correction: Hamming Code**

**Steps to Implement Hamming Code**

1. **Add Parity Bits**:
   * Place parity bits at positions that are powers of 2 (1, 2, 4, 8, ...).
2. **Calculate Parity Bits**:
   * Each parity bit checks a specific set of data bits. The parity bit is calculated such that the number of 1s in the set (including the parity bit) is even.
3. **Transmit Data**:
   * The sender transmits the original data bits along with the calculated parity bits.
4. **Error Detection and Correction**:
   * At the receiver's end, the receiver recalculates the parity and checks whether the data is error-free. If the parity doesn't match, it can locate the erroneous bit and correct it.

**Hamming Code Implementation in C**

This program implements both the generation (sender) and correction (receiver) of a Hamming Code.

**Steps:**

* **Sender Side**:
  + Add parity bits to the data.
  + Calculate the parity bits using XOR operation.
* **Receiver Side**:
  + Recalculate the parity bits and check for errors.
  + If an error is found, the receiver can correct it.

**C Program: Hamming Code for Error Correction**

c

Copy code

#include <stdio.h>

#include <math.h>

#include <string.h>

#define MAX\_BITS 32

// Function to calculate parity bit for a given position (using XOR)

int calculateParityBit(int data[], int n, int pos) {

int parity = 0;

for (int i = 0; i < n; i++) {

if ((i + 1) & (1 << (pos - 1))) { // Check if bit i is in the parity group

parity ^= data[i]; // XOR operation for parity calculation

}

}

return parity;

}

// Function to encode the data with Hamming code (generate parity bits)

void encodeHammingCode(char \*data, int n, int \*encoded) {

int i, j, k = 0;

int m = n + ceil(log2(n)) + 1; // Total number of bits (data + parity)

int parityPos = 1;

// Initialize encoded array with data and placeholders for parity bits

for (i = 0; i < m; i++) {

if (i == parityPos - 1) {

encoded[i] = -1; // Placeholder for parity bit

parityPos \*= 2;

} else {

encoded[i] = data[k++] - '0'; // Copy data bits

}

}

// Calculate parity bits

for (i = 0; i < log2(m); i++) {

encoded[(1 << i) - 1] = calculateParityBit(encoded, m, i + 1);

}

}

// Function to detect and correct errors in received data using Hamming Code

int decodeHammingCode(int \*encoded, int n) {

int i, j, errorPos = 0;

int m = n + ceil(log2(n)) + 1;

// Check each parity bit

for (i = 0; i < log2(m); i++) {

int parity = calculateParityBit(encoded, m, i + 1);

if (parity != encoded[(1 << i) - 1]) {

errorPos += (1 << i); // Find the position of the error

}

}

// Correct the error if detected

if (errorPos) {

printf("Error detected at position: %d\n", errorPos);

encoded[errorPos - 1] = (encoded[errorPos - 1] == 1) ? 0 : 1;

printf("Data after correction: ");

for (i = 0; i < m; i++) {

printf("%d", encoded[i]);

}

printf("\n");

} else {

printf("No errors detected.\n");

}

// Return the decoded data (after removing parity bits)

printf("Decoded data: ");

for (i = 0; i < m; i++) {

if ((i + 1) != (1 << i)) { // Skip parity bits

printf("%d", encoded[i]);

}

}

printf("\n");

}

int main() {

char data[MAX\_BITS];

int n;

// Input data to be transmitted (binary string)

printf("Enter the binary data to be encoded: ");

scanf("%s", data);

// Length of data

n = strlen(data);

// Encoded data (with parity bits)

int encoded[MAX\_BITS] = {0};

// Encode the data using Hamming Code

encodeHammingCode(data, n, encoded);

// Print encoded data with parity bits

printf("Encoded data with parity bits: ");

for (int i = 0; i < n + ceil(log2(n)) + 1; i++) {

printf("%d", encoded[i]);

}

printf("\n");

// Simulate a bit error (optional for testing)

// Example: Change one of the data bits to simulate an error

// encoded[4] = (encoded[4] == 0) ? 1 : 0;

// Decode and check for errors

decodeHammingCode(encoded, n);

return 0;

}

**How It Works:**

**Sender Side (Encoding):**

1. **Input**: The binary data to be transmitted.
2. **Add Parity Bits**: Parity bits are placed at positions that are powers of 2 (1, 2, 4, 8, ...).
3. **Calculate Parity**: Each parity bit is calculated based on the XOR of the relevant data bits.
4. **Transmit Data**: The encoded data (data + parity bits) is transmitted.

**Receiver Side (Decoding and Error Correction):**

1. **Recalculate Parity**: The receiver recalculates the parity for the received data.
2. **Error Detection**: If the recalculated parity does not match the received parity, an error is detected.
3. **Error Correction**: If the error is detected, the position of the erroneous bit is identified and corrected.

**Compilation and Execution**

**Compilation:**

bash

Copy code

gcc -o hamming hamming.c

**Execution:**

bash

Copy code

./hamming

**Sample Input and Output**

**Input:**

plaintext

Copy code

Enter the binary data to be encoded: 1101001

**Output (Sender Side):**

plaintext

Copy code

Encoded data with parity bits: 111010010111

**Output (Receiver Side):**

plaintext

Copy code

Decoded data: 1101001

No errors detected.

**Key Points**

1. **Error Detection and Correction**:
   * Hamming Code can detect and correct single-bit errors.
   * It can detect, but not correct, two-bit errors.
2. **Parity Bits**:
   * The number of parity bits is determined by the formula p≥log⁡2(n+p+1)p \geq \log\_2(n + p + 1)p≥log2​(n+p+1), where n is the number of data bits.
3. **Error Location**:
   * If an error is detected, the receiver can find the position of the corrupted bit and correct it.