**Experiment No. 6 Date:**

**TITLE: ERROR DETECTION AND CORRECTION**

**AIM**: To study error detection and correction methods like Hamming code.

**Theory:**

Hamming Codes:

Hamming codes are error detecting and self-correcting code developed by Richard Hamming. It is used to detect and correct single-bit errors and detect (but not correct) two-bit errors. The code uses parity bits placed at specific positions within the data bits to create a codeword. If a data bit is represented by positions that are powers of 2 (1, 2, 4, 8, etc.), then parity bits are calculated for overlapping groups of data bits. When a bit error occurs, the parity bits indicate the position of the corrupted bit, which can then be corrected.

They are commonly used in memory systems, satellite communication, and digital communication where single-bit errors are likely to occur.

Issues:

* Ineffective for burst errors (multiple-bit errors).
* Overhead increases with the number of parity bits added for larger data block.

Efficiency: It is eficient for single-bit error detection and correction but not suited for high-error environments.

### Cyclic Redundancy Check (CRC)

### CRC is an error-detecting technique used to detect errors during data transmission. It treats data as a binary number and divides it by a predetermined polynomial, resulting in a remainder (CRC value) that is appended to the data. During reception, the same division is performed to check if the data was altered.

### It is used in data transmission protocols like Ethernet, USB, and file transfer protocols to ensure data integrity.

Issues:

* It can detect burst errors but error correction is not possible.
* The efficiency depends on the choice of the polynomial used.

Efficiency: It is highly efficient for detecting errors, especially burst errors. It is commonly used in network communication and data storage systems.

Real-Time Use: It is used in network protocols (Ethernet, TCP/IP), storage devices (Hard Disks, SSDs), and file compression formats (ZIP, RAR).

Checksum:

A checksum is a basic error-detecting technique where the data is divided into equal-sized blocks. The blocks are summed using binary addition, and the resulting value is sent along with the data. The receiver repeats the addition and compares the result with the transmitted checksum.

It is used in simpler data transfer methods like UDP (User Datagram Protocol) and file transmission systems where high reliability is not required.

Issues:

* Ineffective against specific types of errors (e.g., if two bits are altered in a way that maintains the sum).
* Not suitable for high-reliability systems.

Efficiency: It has low computational overhead but limited reliability. It is useful for detecting random errors but not burst errors.

Real-Time Use: It is commonly used in UDP, TCP, and simple file transmission systems for basic error detection.

**CODE**  
**a) Hamming code**

#include <stdio.h>

// Function to calculate Hamming Code (7,4)

void generateHammingCode(int data[4], int code[7]) {

code[2] = data[0]; // Data bit 1

code[4] = data[1]; // Data bit 2

code[5] = data[2]; // Data bit 3

code[6] = data[3]; // Data bit 4

// Parity bits

code[0] = code[2] ^ code[4] ^ code[6]; // P1

code[1] = code[2] ^ code[5] ^ code[6]; // P2

code[3] = code[4] ^ code[5] ^ code[6]; // P4

}

// Function to check received Hamming Code

void checkHammingCode(int received[7]) {

int p1 = received[0] ^ received[2] ^ received[4] ^ received[6];

int p2 = received[1] ^ received[2] ^ received[5] ^ received[6];

int p4 = received[3] ^ received[4] ^ received[5] ^ received[6];

int errorPosition = p1 \* 1 + p2 \* 2 + p4 \* 4;

if (errorPosition == 0) {

printf("No error in received data.\n");

} else {

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

received[errorPosition - 1] ^= 1; // Correct the error

printf("Corrected data: ");

for (int i = 0; i < 7; i++) {

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

}

printf("\n");

}

}

int main() {

int data[4], code[7], received[7];

printf("Enter 4-bit data one by one: ");

for (int i = 0; i < 4; i++) {

scanf("%d", &data[i]);

}

generateHammingCode(data, code);

printf("Encoded Hamming Code: ");

for (int i = 0; i < 7; i++) {

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

}

printf("\n");

printf("Enter received 7-bit data one by one: ");

for (int i = 0; i < 7; i++) {

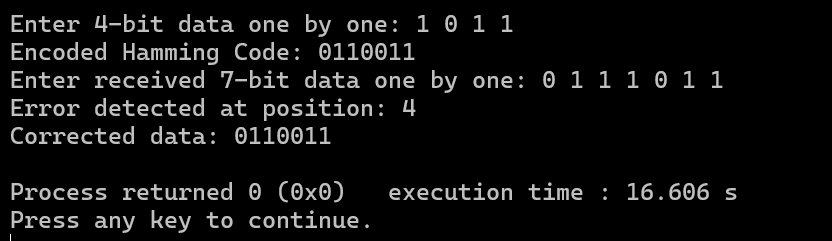
scanf("%d", &received[i]);

}

checkHammingCode(received);

return 0;

}

**OUTPUT**  


**b)CRC**

#include <stdio.h>

#include <string.h>

void xor1(char \*result, const char \*a, const char \*b) {

int n = strlen(b);

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

result[i - 1] = (a[i] == b[i]) ? '0' : '1';

}

result[n - 1] = '\0';

}

void mod2div(char \*remainder, const char \*dividend, const char \*divisor) {

int pick = strlen(divisor);

int n = strlen(dividend);

char tmp[100], zero\_str[100];

strncpy(tmp, dividend, pick);

tmp[pick] = '\0';

while (pick < n) {

char temp\_res[100];

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

xor1(temp\_res, divisor, tmp);

} else {

memset(zero\_str, '0', pick);

zero\_str[pick] = '\0';

xor1(temp\_res, zero\_str, tmp);

}

snprintf(tmp, sizeof(tmp), "%s%c", temp\_res, dividend[pick]);

pick++;

}

char temp\_res[100];

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

xor1(temp\_res, divisor, tmp);

} else {

memset(zero\_str, '0', pick);

zero\_str[pick] = '\0';

xor1(temp\_res, zero\_str, tmp);

}

strcpy(remainder, temp\_res);

}

void transmitter() {

char data[100], key[100], code[100], remainder[100];

printf("Enter data to be transmitted: ");

scanf("%s", data);

printf("Enter the Generating polynomial: ");

scanf("%s", key);

int l\_key = strlen(key);

char appended\_data[100];

strcpy(appended\_data, data);

memset(appended\_data + strlen(data), '0', l\_key - 1);

appended\_data[strlen(data) + l\_key - 1] = '\0';

mod2div(remainder, appended\_data, key);

strcpy(code, data);

strcat(code, remainder);

printf("Data padded with n-1 zeros: %s\n", appended\_data);

printf("CRC or Check value is: %s\n", remainder);

printf("Final data to be sent: %s\n", code);

}

void receiver() {

char received\_data[100], key[100], remainder[100];

printf("Enter the received data: ");

scanf("%s", received\_data);

printf("Enter the Generating polynomial: ");

scanf("%s", key);

mod2div(remainder, received\_data, key);

if (strchr(remainder, '1') == NULL) {

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

printf("Data sent is %.\*s\n", (int) (strlen(received\_data) - strlen(key) + 1), received\_data);

} else {

printf("Error detected.\n");

}

}

int main() {

int choice;

while (1) {

printf("\nMenu\n1. Transmitter\n2. Receiver\n3. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

transmitter();

break;

case 2:

receiver();

break;

case 3:

return 0;

default:

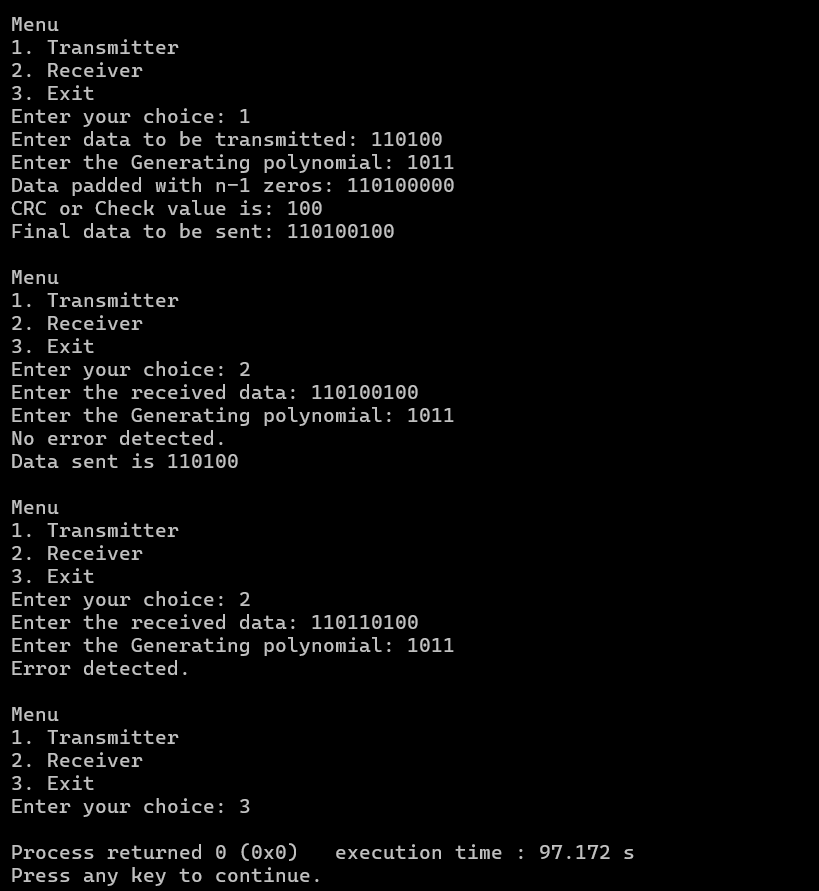
printf("Invalid choice! Please enter 1, 2, or 3.\n");

}

}

}

**OUTPUT:**



**c) checksum**

#include <stdio.h>

#include <string.h>

// Function to validate 4-bit binary input

int isValidBinary(char bin[]) {

if (strlen(bin) != 4) return 0;

for (int i = 0; i < 4; i++) {

if (bin[i] != '0' && bin[i] != '1')

return 0;

}

return 1;

}

// Function to compute binary sum with carry handling

void binarySum(char a[], char b[], char sum[]) {

int carry = 0;

int tempSum[5] = {0}; // Temporary 5-bit sum to handle overflow

for (int i = 3; i >= 0; i--) {

int bitSum = (a[i] - '0') + (b[i] - '0') + carry;

tempSum[i + 1] = bitSum % 2; // Store result

carry = bitSum / 2; // Update carry

}

tempSum[0] = carry; // Store carry in the MSB

// Handle carry wrapping (if there is an overflow carry)

if (carry) {

for (int i = 3; i >= 0; i--) {

int bitSum = tempSum[i + 1] + carry;

sum[i] = (bitSum % 2) + '0';

carry = bitSum / 2;

}

} else {

for (int i = 0; i < 4; i++) {

sum[i] = tempSum[i + 1] + '0';

}

}

sum[4] = '\0'; // Null-terminate string

}

// Function to compute one's complement checksum

void computeChecksum(char sum[], char checksum[]) {

for (int i = 0; i < 4; i++) {

checksum[i] = (sum[i] == '0') ? '1' : '0';

}

checksum[4] = '\0';

}

// Function to verify checksum

void verifyChecksum(char receivedSum[], char checksum[], char num1[], char num2[]) {

char totalSum[5];

binarySum(receivedSum, checksum, totalSum);

// Check if total sum is "1111" (indicating no error)

if (strcmp(totalSum, "1111") == 0) {

printf("\nThere is no error in the received data.");

printf("\nDecoded data: %s %s\n", num1, num2);

} else {

printf("\nError detected in received data!\n");

}

}

int main() {

int choice;

char num1[5], num2[5], sum[5], checksum[5], receivedSum[5];

do {

printf("\nChecksum Menu:\n");

printf("1. Transmitter\n");

printf("2. Receiver\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: // Transmitter

do {

printf("\nEnter the first 4-bit binary number: ");

scanf("%s", num1);

} while (!isValidBinary(num1));

do {

printf("Enter the second 4-bit binary number: ");

scanf("%s", num2);

} while (!isValidBinary(num2));

// Compute binary sum and checksum

binarySum(num1, num2, sum);

computeChecksum(sum, checksum);

// Display results

printf("\nThe sum of the two binary numbers is: %s", sum);

printf("\nChecksum (1's complement of sum): %s", checksum);

printf("\nEncoded data to send: %s %s %s\n", num1, num2, checksum);

break;

case 2: // Receiver

do {

printf("\nEnter the first 4-bit binary number: ");

scanf("%s", num1);

} while (!isValidBinary(num1));

do {

printf("Enter the second 4-bit binary number: ");

scanf("%s", num2);

} while (!isValidBinary(num2));

do {

printf("Enter the checksum values: ");

scanf("%s", receivedSum);

} while (!isValidBinary(receivedSum));

binarySum(num1, num2, sum);

verifyChecksum(sum, receivedSum, num1, num2);

break;

case 3: // Exit

printf("\nExiting program...\n");

break;

default:

printf("\nInvalid choice! Please enter 1, 2, or 3.\n");

}

} while (choice != 3);

return 0;

}

**OUTPUT:**

A screenshot of a computer program

AI-generated content may be incorrect.

***CONCLUSION:***

This experiment implemented and performed Hamming Code, CRC, and Checksum for error detection and correction methods successfully. Hamming Code corrects single-bit errors, CRC detects errors in networks and storage, and Checksum provides a simple but less reliable error check. Each method balances accuracy and efficiency based on system needs.