EXPERIMENT NO. 2

Aim :- Implementation of Hamming code for Error Detection and correction.

Theory :-

Hamming code is a set of error-correction codes that can be used to detect and correct the errors that can occur when the data is moved or stored from the sender to the receiver. It is a technique developed by R.W. Hamming for error correction. Redundant bits are extra binary bits that are generated and added to information-carrying bits of data transfer to ensure that no bits were lost during the data transfer. The number of redundant bits can be calculated using the following formula:

$$2^r \geq m+r+1$$

where, r = redundant bits, m = data bits

Suppose number of data bits is 7, then number of redundant bits will be 4 ($2^4 \geq 7+4+1$)

Parity bits are used for error detection. There are two types of parity bits:

Even parity bit: In this, number of 1's are counted. If count is odd, parity bit value is set to 1, else set to 0.

Odd parity bit : In this, number of 1's are counted. if count is odd, parity bit value is set to 0. else set to 1.

General Algorithm of Hamming code :

1. Write the bit positions starting from 1 in binary form (1, 10, 11, etc).
2. All bit positions that are a power of 2 are marked as parity bits (1, 2, 4, 8, etc)
3. All the other bit positions are marked as data bits
4. Each data bit is included in a unique set of parity bits, as determined its bit position in binary form
   a. Parity bit 1 covers all bits positions whose binary representation includes a 1 in the least significant position (1, 3, 5, 7, 9, 11, etc).
   b. Parity bit 2 covers all bits positions whose binary representation includes a 1 in second position from least significant bit (2, 3, 6, 7, 10, 11) etc
   c. Parity bit 4 covers all bits positions whose binary representation includes a 1 in the third position from least significant bit (4-7, 12-15, 20-23, etc).
   d. Parity bit 8 covers all bits positions whose binary representation includes a 1 in fourth bit position from the least significant bit bits (8-15, 24-31, 40-47, etc).

e. In general, each parity bit covers all bits where the bitwise AND of the parity position and bit position is non-zero.

5. Since we check for even parity set a parity bit to 1 if total number of ones in the positions it checks is odd.

Hamming code can thereby detect and correct any single-bit error. If two data bits were flipped, it could detect it but not correct the error. Because the parity bits themselves do not have any parity data stored, if a data bit and a parity bit were flipped, it would be indistinguishable from a single-bit flip. Therefore, an additional overall parity bit is often added to reliably detect errors with 2 bits.

Conclusion :-

Hamming code is a widely used error-correction capable of correcting a single-bit error, but can only correct a limited number of multiple errors.

A^+ 5 4 22/8/23

```
op = int(input("Enter 1 for Hamming code generation\nEnter 2 for
error detection\n"))

if op == 1:
    m = list(map(int, input("Enter the data bits in binary:\n")))
    r = 0
    while (len(m) + r + 1) > (2 ** r):
        r += 1
    print("Total number of data bits m =", len(m))
    print("Total number of parity bits required r =", r)
    print("Total number of bits in the encoded data =", len(m) + r)
    print("The redundant bits are placed in the position", [2 ** x
for x in range(r)])

    m.reverse()
    c, ch, j, hamming = 0, 0, 0, []

    for i in range(0, (r + len(m))):
        p = (2 ** c)

        if p == (i + 1):
            hamming.append(0)
            c = c + 1
        else:
            hamming.append(int(m[j]))
            j = j + 1

    for parity in range(0, len(hamming)):
        ph = (2 ** ch)
        if ph == (parity + 1):
            startIndex = ph - 1
            i = startIndex
            y = []

            while i < len(hamming):
                block = hamming[i:i + ph]
                y.extend(block)
                i += 2 * ph

            for z in range(1, len(y)):
                hamming[startIndex] = hamming[startIndex] ^ y[z]
            ch += 1

    hamming.reverse()
    print('Hamming code generated would be:', end="")
    print(int(''.join(map(str, hamming))))

elif op == 2:
    print('Enter the received Hamming code')
    d = input()
    data = list(d)
    data.reverse()
    c, ch, h, h_copy = 0, 0, [], []
```

```
    for k in range(0, len(data)):
        p = (2 ** c)
        h.append(int(data[k]))
        h_copy.append(data[k])
        if p == (k + 1):
            c = c + 1

    parity_list = []

    for parity in range(0, len(h)):
        ph = (2 ** ch)
        if ph == (parity + 1):
            startIndex = ph - 1
            i = startIndex
            y = []

            while i < len(h):
                block = h[i:i + ph]
                y.extend(block)
                i += 2 * ph

            for z in range(1, len(y)):
                h[startIndex] = h[startIndex] ^ y[z]
            parity_list.append(h[parity])
            ch += 1
    parity_list.reverse()
    error = sum(int(parity_list) * (2 ** i) for i, parity_list in
enumerate(parity_list[::-1]))

    if error == 0:
        print('There is no error in the received Hamming code')
    elif error >= len(h_copy):
        print('Error cannot be detected')
    else:
        print('Error is in', error, 'bit')

        if h_copy[error - 1] == '0':
            h_copy[error - 1] = '1'
        elif h_copy[error - 1] == '1':
            h_copy[error - 1] = '0'
        print('After correction, Hamming code is:')
        h_copy.reverse()
        print(int(''.join(map(str, h_copy))))
else:
    print('Option entered does not exist')

'''
python -u "C:/Users/Rishab/OneDrive/Desktop/CN Experiments/import
hamm.py"
Enter 1 for Hamming code generation
Enter 2 for error detection
1
Enter the data bits in binary:
1101
Total number of data bits m = 4
Total number of parity bits required r = 3
Total number of bits in the encoded data = 7
The redundant bits are placed in the position [1, 2, 4]
Hamming code generated would be:1100110
```

```
python -u "C:/Users/Rishab/OneDrive/Desktop/CN Experiments/import
hamm.py"
Enter 1 for Hamming code generation
Enter 2 for error detection
2
Enter the received Hamming code
1100110
There is no error in the received Hamming code
python -u "C:/Users/Rishab/OneDrive/Desktop/CN Experiments/import
hamm.py"
Enter 1 for Hamming code generation
Enter 2 for error detection
2
Enter the received Hamming code
1100111
Error is in 1 bit
After correction, Hamming code is:
1100110
'''
```