Department of
Computer Science and Engineering

Title: Implementing of Error Detection & Correction Mechanism using Hamming Code.

Data Communication Lab

CSE 208

Green University of Bangladesh

1 Objective(s)

- To attain knowledge on the Hamming code and how it works.
- To implement Hamming code error detection and correction algorithms.

2 Problem Analysis

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 technique developed by R.W. Hamming for error correction.

Redundant bits are extra binary bits that are generated and added to the 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:

$$2^r \geq m + r + 1 \tag{1}$$

where, r = redundant bit, m = data bit.

Suppose the number of data bits is 7, then the number of redundant bits can be calculated using:

$$r = 2^4 \geq 7 + 4 + 1 \tag{2}$$

Thus, the number of redundant bits= 4

2.0.1 A General Algorithm of Hamming code:

The Hamming Code is simply the use of extra parity bits to allow the identification of an error.

1. Write the bit positions starting from 1 in binary form (1, 10, 11, 100, etc).
2. All the 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 the 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 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).
   (c) 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).
   (d) 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).
   (e) In general, each parity bit covers all bits where the bit-wise AND of the parity position and the bit position is non-zero.
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.
6. Set a parity bit to 0 if the total number of ones in the positions it checks is even.

2.0.2 Approach and Example of Hamming code:

Hamming code uses redundant bits (extra bits) which are calculated according to the below formula:-

$$2r \geq m + r + 1 \tag{3}$$

Where r is the number of redundant bits required and m is the number of data bits.
R is calculated by putting r = 1, 2, 3 . . . until the above equation becomes true.
R1 bit is appended at position 2^0

R2 bit is appended at position 2^1

R3 bit is appended at position 2^2 and so on.

These redundant bits are then added to the original data for the calculation of error at receiver's end. At receiver's end with the help of even parity (generally) the erroneous bit position is identified and since data is in binary we take complement of the erroneous bit position to correct received data.

Respective index parity is calculated for r1, r2, r3, r4 and so on.

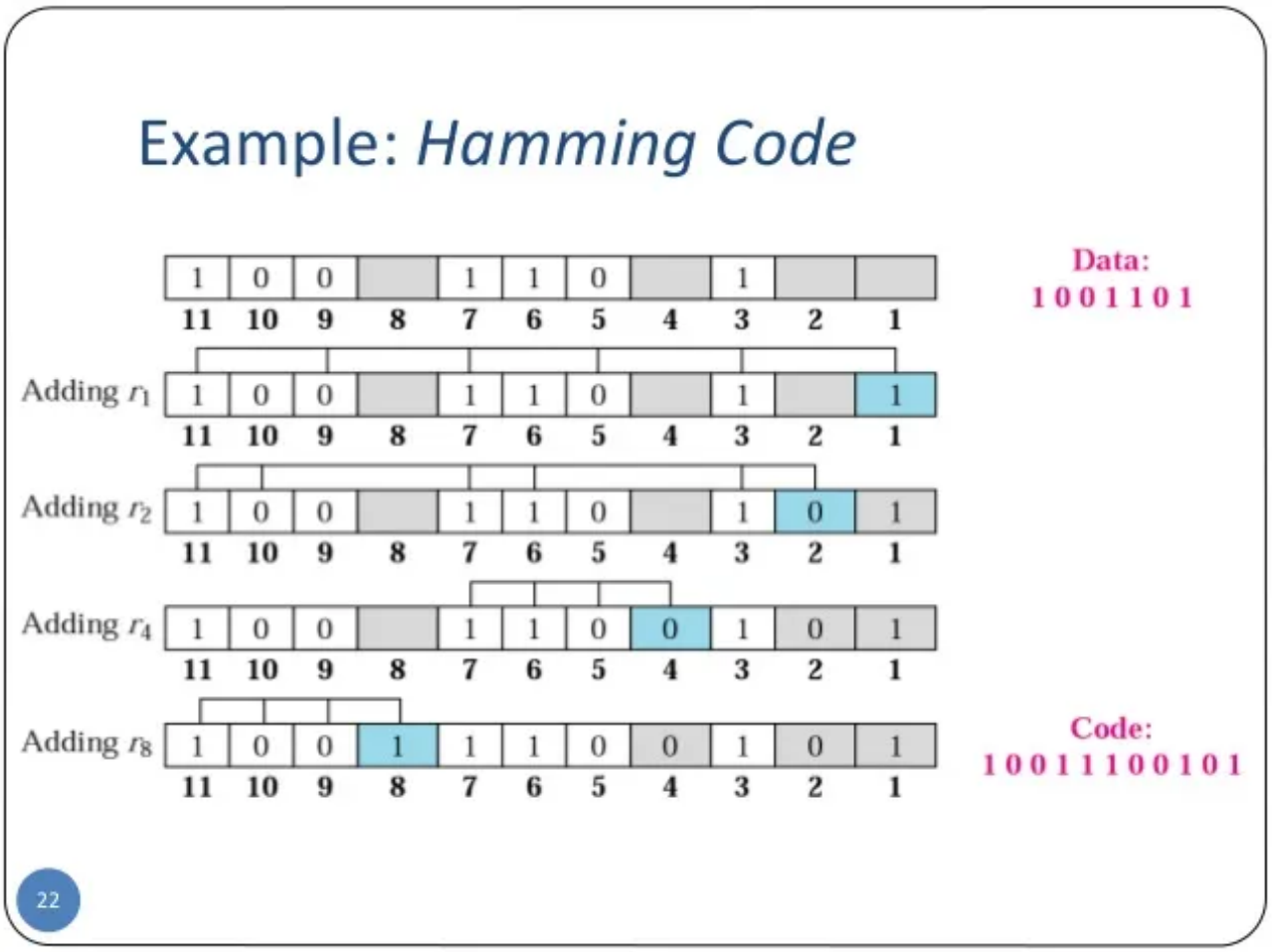

Figure 1: Example of Hamming Code.

2.0.3 Implementation of Even Parity Checking:

```
#include <math.h>
#include <stdio.h>

// Store input bits
int input[32];

// Store hamming code
int code[32];

int ham_calc(int, int);
void solve(int input[], int);

// Function to calculate bit for
// ith position
int ham_calc(int position, int c_l)
```

```
{
    int count = 0, i, j;
    i = position - 1;

    // Traverse to store Hamming Code
    while (i < c_l) {
        for (j = i; j < i + position; j++) {

            // If current boit is 1
            if (code[j] == 1)
                count++;
        }

        // Update i
        i = i + 2 * position;
    }
    if (count % 2 == 0)
        return 0;
    else
        return 1;
}

// Function to calculate hamming code
void solve(int input[], int n)
{
    int i, p_n = 0, c_l, j, k;
    i = 0;

    // Find msg bits having set bit
    // at x'th position of number
    while (n > (int)pow(2, i) - (i + 1)) {
        p_n++;
        i++;
    }

    c_l = p_n + n;
    j = k = 0;

    // Traverse the msgBits
    for (i = 0; i < c_l; i++) {

        // Update the code
        if (i == ((int)pow(2, k) - 1)) {
            code[i] = 0;
            k++;
        }

        // Update the code[i] to the
        // input character at index j
        else {
            code[i] = input[j];
            j++;
        }
    }

    // Traverse and update the
    // hamming code
    for (i = 0; i < p_n; i++) {
```

```
        // Find current position
        int position = (int)pow(2, i);

        // Find value at current position
        int value = ham_calc(position, c_l);

        // Update the code
        code[position - 1] = value;
    }

    // Print the Hamming Code
    printf("\nThe generated Code Word is: ");
    for (i = 0; i < c_l; i++) {
        printf("%d", code[i]);
    }
}

// Driver Code
void main()
{
    // Given input message Bit 0111
    input[0] = 0;
    input[1] = 1;
    input[2] = 1;
    input[3] = 1;

    int N = 4;

    // Function Call
    solve(input, N);
}
```

2.0.4 Output for the code of Even Parity Checking:

```
The generated Code Word is: 0001111
```

3 Discussion & Conclusion

Based on the focused objective(s) to understand about the Hamming code, the additional lab exercise made me more confident towards the fulfilment of the objectives(s).

4 Lab Task (Please implement yourself and show the output to the instructor)

1. Implement Hamming coder where user input will be the bit stream.

5 Lab Exercise (Submit as a report)

Implement of Hamming code correction approach.

6 Policy

Copying from internet, classmate, seniors, or from any other source is strongly prohibited. 100% marks will be *deducted* if any such copying is detected.