VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Name: Apurba Koirala

Reg no: 22BCE3799

Subject Code: BCSE308P

Course Title: Computer Networks Lab

Lab Slot: L31 + L32

Guided By: Dr. Arivoli A

Apurba Koirala
22BCE3799

12th August 2024

LAB Assessment - 2

1. Implement forward error detection mechanism using Cyclic Redundancy Check (CRC). Get the input data from the user for performing CRC. Based on the generator calculate the CRC bits to be added to the data at the sender side. Append the redundant bit with the message. At the receiver side, check for errors in the message during transmission

Given the dataword 10100 1111 0 and the divisor 10111.

a. Show the generation of the code word at the sender side (using binary division).

b. Show the checking of the code word at the receiver site (assume no error).

c. Show the checking of the code word at the receiver site. (assume error in the leftmost 2nd and 4th bit).

a) Using binary division,

generation of code word at sender site,

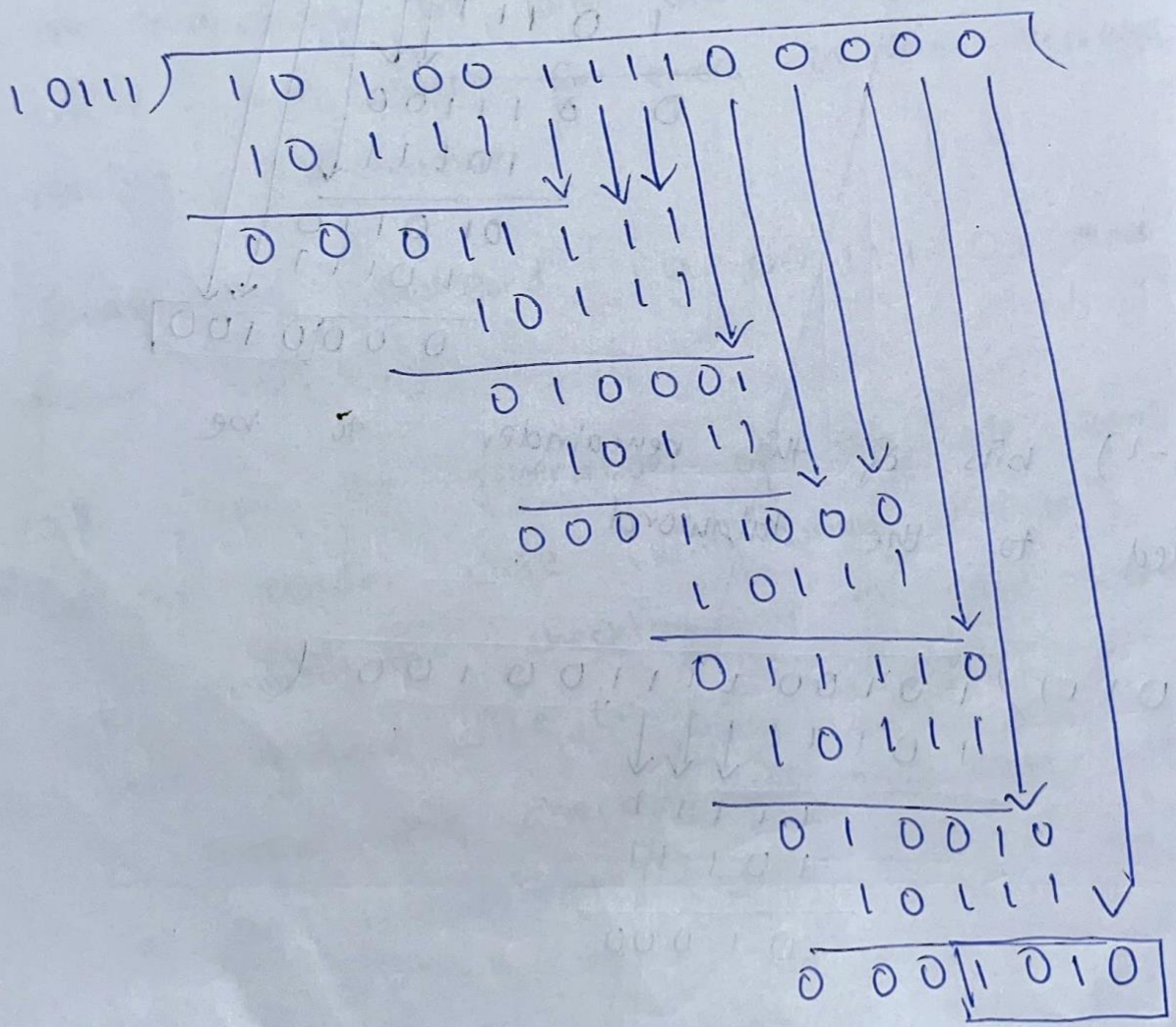
(n-1) 0s to be added to the dataword,

here,

n = 5

n-1 = 4-1 0s to be added.

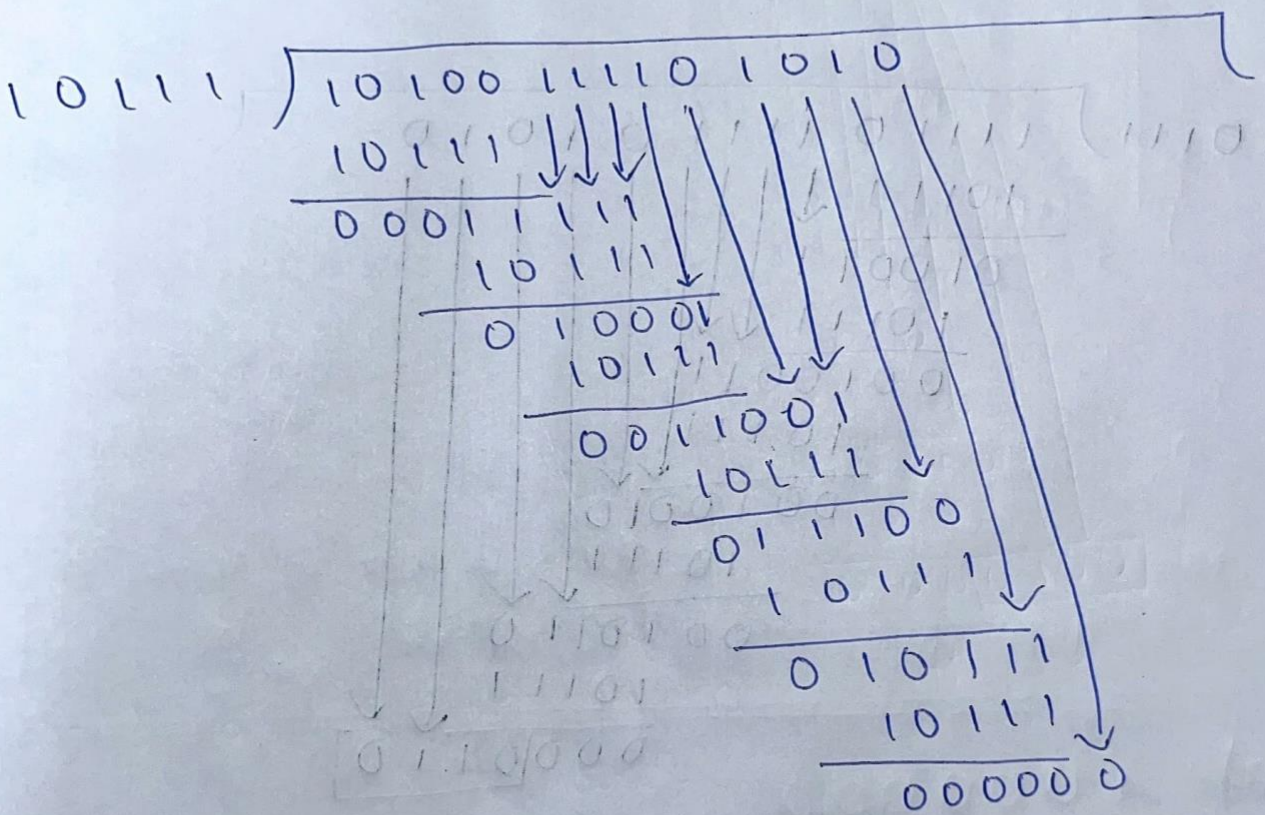
here,



∴ generated code word: 10100111101010

b) Checking the code word at receiver site, assuming there are no errors,

(n-1) bits of the remainder to be added to the dataword.



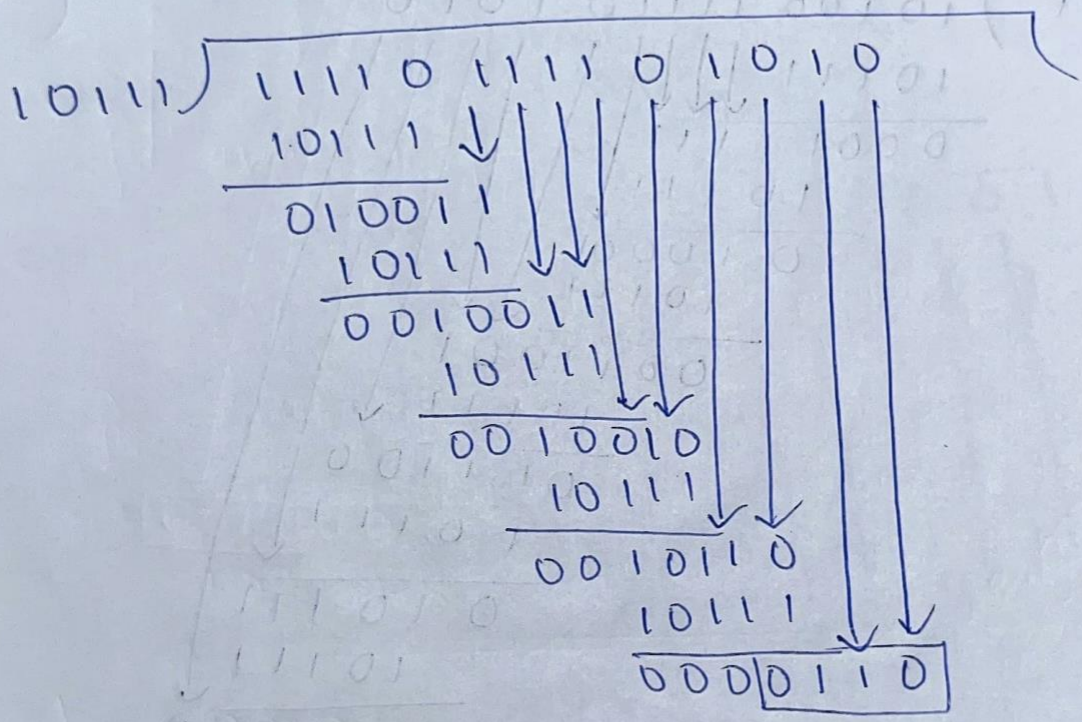
Here, as the remainder is 0, there is no error.

c. assuming error at the leftmost 2nd & 4th bit

the received data word is.

1111 0 1111 0 1010

dividing,



The remainder is a non-zero value which indicates that error has occured during data transmission.

CODE:

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

void performXOR(char* input, const char* poly) {
    int polyLen = strlen(poly);
    for (int i = 0; i <= strlen(input) - polyLen; i++) {
        if (input[i] == '1') {
            for (int j = 0; j < polyLen; j++) {
                input[i + j] = (input[i + j] == poly[j]) ? '0' : '1';
            }
        }
    }
}

void generateCodeword(const char* message, const char* poly, char* codeword) {
    int polyLen = strlen(poly);
    char temp[100];

    strcpy(temp, message);
    for (int i = 0; i < polyLen - 1; i++) {
        strcat(temp, "0");
    }

    performXOR(temp, poly);

    strcpy(codeword, message);
    strcat(codeword, &temp[strlen(temp) - polyLen + 1]);
}

void validateCodeword(char* codeword, const char* poly, char* result) {
    strcpy(result, codeword);
    performXOR(result, poly);
}

int main() {
    char message[] = "1010011110";
```

```
    char poly[] = "10111";
    char codeword[100];
    char result[100];

    generateCodeword(message, poly, codeword);
    printf("Generated codeword (Sender): %s\n", codeword);

    validateCodeword(codeword, poly, result);
    if (strchr(result, '1') == NULL) {
        printf("No error detected in received codeword (No error case).\n");
    } else {
        printf("Error detected in received codeword (No error case).\n");
    }
    printf("Remainder (No error case): %s\n", &result[strlen(result) - strlen(poly) + 1]);

    char alteredCodeword[100];
    strcpy(alteredCodeword, codeword);
    alteredCodeword[1] = (alteredCodeword[1] == '0') ? '1' : '0';
    alteredCodeword[3] = (alteredCodeword[3] == '0') ? '1' : '0';
    printf("Erroneous codeword (Receiver): %s\n", alteredCodeword);

    validateCodeword(alteredCodeword, poly, result);
    if (strchr(result, '1') == NULL) {
        printf("No error detected in received codeword (Error case).\n");
    } else {
        printf("Error detected in received codeword (Error case).\n");
    }
    printf("Remainder (Error case): %s\n", &result[strlen(result) - strlen(poly) + 1]);

    return 0;
}
```

Output:

```
Generated codeword (Sender): 10100111101010
No error detected in received codeword (No error case).
Remainder (No error case): 0000
Erroneous codeword (Receiver): 11110111101010
Error detected in received codeword (Error case).
Remainder (Error case): 0110
```

2. Assuming even parity (odd Parity), find the parity bit for each of the following data units:

a. 100 1011
b. 0001100
c. 1000000
d. 1110111

Answer;

a) 1001011

Soln;

here,

in the given 7-bit data

1 0 0 1 0 1 1

no. of ones(1s) = 4

for even parity,

Pev = 0, as number of 1s is already even.

for odd parity

Pod = 1, as number of 1s is not odd and ~~the~~ 1as parity bit should be added.

b) 0001100

Soln;

here,

in the given 7 bit data

0 0 0 1 1 0 0

no. of ones = 2

for even parity,

$P_{ev} = 0$, as the number of 1s is even.

for odd parity,

$P_{od} = 1$, as the number of 1s is to be made odd with the parity bit.

---

c) 1000000

Soln;

here,

in the given 7 bit data

1 0 0 0 0 0 0

no. of ones = 1

for even parity

$P_{ev} = 1$, as the number of 1's is to be made even with the extra parity bit.

for odd parity

$P_{od} = 0$, as the number of 1s is already odd.

d) 1 1 1 0 1 1 1

Soln;

here,

in the given 7 bit dat

1 1 1, 0 1 1 1

no. of ones = 6.

for even parity,

Pev = 0 as the number of 1s is already even.

for odd parity

P_{OD} = 1 as the number of 1s is not odd and is to be mad odd.

CODE:

```c
#include <stdio.h>
#include <string.h>

void calculateParity(char *data) {
    int count = 0;
    int i;

    for (i = 0; i < strlen(data); i++) {
        if (data[i] == '1') {
            count++;
        }
    }

    int even_parity = (count % 2 == 0) ? 0 : 1;
    int odd_parity = (count % 2 == 0) ? 1 : 0;

    printf("Even Parity Bit: %d\n", even_parity);
    printf("Odd Parity Bit: %d\n", odd_parity);
}

int main() {
    char data[100];

    printf("Enter the binary sequence: ");
    scanf("%s", data);

    calculateParity(data);

    return 0;
}
```

OUTPUT:

```
Enter the binary sequence: 1110111
Even Parity Bit (Pev): 0
Odd Parity Bit (Pod): 1
```

3. Hamming code: Error detection and Error Correction.

a) Data word - 0100 - Error in the leftmost 2nd bit

b) Data word - 0111 - Error in the rightmost 5th bit

c) Data Word - 1111 - Error in the leftmost 4th bit.

Solution;

here

m (Length of dataword) = 4

so, calculating no. of redundant bits (r)

$2^r >= m + r + 1$

Taking different values of r

r=0, $2^0 = 1 \not\geq 4+0+1 = 5$ false

$2^1 = 2 \not\geq 4+1+1 = 6$ false

$2^2 = 4 \not\geq 4+2+1 = 7$ false

$2^3 = 8 \geq 4+3+1 = 8$ true

r = 3

total no. of bits = 4+3 = 7

a) 0100

7	6	5	4	3	2	1
0	1	0	P_3	0	P_2	P_1

	P_3	P_2	P_1
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

$P_1 \rightarrow 1, 3, 5, 7$

$P_2 \rightarrow 2, 3, 6, 7$

$P_3 \rightarrow 4, 5, 6, 7$

Now,

when even parity

a) $P_1 = P_1\ 0\ 0\ 0$

P_1 be 0 for even parity

$P_2 = P_2\ 0\ 1\ 0$

P_2 be 1 for even parity

$P_3 = P_3\ 0\ 1\ 0$

P_3 be 1 for even parity

data = 0 1 0 1 0 0

Assuming error in leftmost 2nd bit

transmitted data = 0 0 0 1 0 1 0

Now, checking error,

$P_1 = 0000$ (even) - 0

$P_2 = 1000$ (odd) - 1

$P_3 = 1000$ (odd) - 1

here, $P_3 P_2 P_1 = 110$ represents 6.

6th bit contains an error.

The bit to be changed is the 6th bit

∴ 6th bit to be changed from 0 to 1.

∴ The corrected word is 0101010

b) Data word 0111, error in the rightmost 5th bit.

7	6	5	4	3	2	1
0	1	1	P_4	1	P_2	P_1

where,

$P_1 \rightarrow$ 1, 3, 5, 7

$P_2 \rightarrow$ 2, 3, 6, 7

$P_3 \rightarrow$ 4, 5, 6, 7

b)

when even parity

$P_1 = P_1 \,11\,0$

$P_1 = 0$

$P_2 = P_2 \,11\,0$

$P_2 = 0$

$P_3 = P_3 \,11\,0$

$P_3 = 0$

∴ The hamming code = 0110100

Assuming error in the rightmost 5th bit.

the received word = 0100100

calculating parity bits for received word,

$P_1 = 0100$ (odd) - 1
$P_2 = 0110$ (even) - 0
$P_3 = 0010$ (odd) - 1

$P_3P_2P_1 = 101$, which represents 5.

Meaning the 5th bit should be flipped from 0 to 1.

corrected word = 0110100

c) Data word - 1111 - Error in the leftmost 4th bit.

7	6	5	4	3	2	1
1	1	1	P_3	1	P_2	P_1

where,

$P_1 \rightarrow 1, 3, 5, 7$
$P_2 \rightarrow 2, 3, 6, 7$
$P_3 \rightarrow 4, 5, 6, 7$

c) when even parity

$P_1 = P_1\ 1\ 1$
$P_1 = 1$
$P_2 = P_2\ 1\ 1$
$P_2 = 1$
$P_3 = P_3\ 1\ 1$
$P_3 = 1$

Thus,

complete hamming code = 1 1 1 1 1 1 1

Error in the leftmost 4th bit,

the received word = 1 1 1 0 1 1 1

checking parity bits for received word,

P_1 : 1, 1, 1, 1 (even) - 0

P_2 : 1, 1, 1, 1 (even) - 0

P_3 : 0, 1, 1, 1 (odd) - 1

$P_3 P_2 P_1$: 1 0 0 which is 4th bit.

Meaning, the 4th bit should be flipped.

∴ corrected word = 1 1 1 1 1 1 1

CODE:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Define a structure to represent the Hamming code
typedef struct {
    char* data;
    int m, r;
    char* msg;
} Hamming;

// Function to reverse a string
void reverse(char* str, int length) {
    int start = 0;
    int end = length - 1;
    while (start < end) {
        char temp = str[start];
        str[start] = str[end];
        str[end] = temp;
        start++;
        end--;
    }
}

// Function to set the redundant bits in the Hamming code
void setRedundantBits(Hamming* h) {
    int bit = 0;
    for (int i = 1; i <= h->m + h->r; i *= 2) {
        int count = 0;
        for (int j = i + 1; j <= h->m + h->r; j++) {
            if (j & (1 << bit)) {
                if (h->msg[j] == '1') count++;
            }
        }
        if (count % 2 == 0) h->msg[i] = '0';
        else h->msg[i] = '1';
        bit++;
```

```
    }
}

// Function to display the message
void showMsg(Hamming* h) {
    printf("The data packet to be sent is: ");
    for (int i = h->m + h->r; i >= 1; i--) {
        printf("%c ", h->msg[i]);
    }
    printf("\n");
}

// Function to flip a bit at the specified position
void flipBit(Hamming* h, int bitPos) {
    if (bitPos >= 1 && bitPos <= h->m + h->r) {
        h->msg[bitPos] = (h->msg[bitPos] == '0') ? '1' : '0';
        printf("Bit position %d flipped. Updated code: ", bitPos);
        for (int i = h->m + h->r; i >= 1; i--) {
            printf("%c ", h->msg[i]);
        }
        printf("\n");
    } else {
        printf("Invalid bit position!\n");
    }
}

// Function to simulate the receiver's actions
void receiver(Hamming* h) {
    char ans[20] = "";
    int bit = 0;
    for (int i = 1; i <= h->m + h->r; i *= 2) {
        int count = 0;
        for (int j = i + 1; j <= h->m + h->r; j++) {
            if (j & (1 << bit)) {
                if (h->msg[j] == '1') count++;
            }
        }
        if (count % 2 == 0) {
            if (h->msg[i] == '0') strcat(ans, "0");
```

```
            else strcat(ans, "1");
        } else {
            if (h->msg[i] == '1') strcat(ans, "0");
            else strcat(ans, "1");
        }
        bit++;
    }

    if (strchr(ans, '1') != NULL) {
        int power = 1;
        int wrongBit = 0;
        for (int i = 0; i < strlen(ans); i++) {
            if (ans[i] == '1') wrongBit += power;
            power *= 2;
        }
        printf("Bit number %d is wrong and has an error.\n", wrongBit);
    } else {
        printf("Correct data packet received.\n");
    }
}

// Main function
int main() {
    char data[20];
    printf("Enter data:\n");
    scanf("%s", data);

    Hamming h;
    h.r = 0;
    reverse(data, strlen(data));
    h.m = strlen(data);

    int power = 1;
    while (power < (h.m + h.r + 1)) {
        h.r++;
        power *= 2;
    }

    h.msg = (char*)malloc((h.m + h.r + 1) * sizeof(char));
```

```
    int curr = 0;
    for (int i = 1; i <= h.m + h.r; i++) {
        if (i & (i - 1)) {
            h.msg[i] = data[curr++];
        } else {
            h.msg[i] = 'n';
        }
    }

    setRedundantBits(&h);
    showMsg(&h);

    int flipPosition;
    printf("Enter the bit position to flip (1-based index):\n");
    scanf("%d", &flipPosition);
    flipBit(&h, flipPosition);

    receiver(&h);

    free(h.msg);
    return 0;
}
```

OUTPUT:

```
Enter data:
0100
The data packet to be sent is: 0 1 0 1 0 1 0
Enter the bit position to flip (1-based index):
6
Bit position 6 flipped. Updated code: 0 0 0 1 0 1 0
Bit number 6 is wrong and has an error.
```

4. Implement forward error detection mechanism using checksum method. Get the input data from the user for performing Checksum. Calculate the Checksum bits to be added to the data at the sender site. Append the redundant bit with the message. Check whether the data has reached correctly at receiver or not?

Consider the data unit to be transmitted is -

0 1 1 1 1 0 0 1 1 0 0 0 0 0 1 0 1 1 0 0 1 0 0 1 0 0 0
0 1 0 1

Consider 8 bit checksum is used.

Solution;

dividing the data to 8-bits:

0 1 1 1 1 0 0 1
1 0 0 0 0 0 1 0
1 1 1 0 0 1 0 0
1 0 0 0 0 1 0 1

we perform one's complement binary addition between the first and second bit:

$$\begin{array}{r} 0\ 1\ 1\ 1\ 1\ 0\ 0\ 1 \\ +\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \\ \hline 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1 \end{array}$$

adding the 3rd bit to the previous sum,

```
  1 1
  1 1 1 1 1 0 1 1
+ 1 1 1 0 0 1 0 0
-----------------
  1 1 0 1 1 1 1 1
+               1
-----------------
1 1 1 0 0 0 0 0 0
```

adding the 4th bit,

```
  1 1 1 0 0 0 0 0
+ 1 0 0 0 0 1 0 1
-----------------
  0 1 1 0 0 1 0 1
+               1
-----------------
  0 1 1 0 0 1 1 0
```

taking 1's complement of the final sum

= 1 0 0 1 1 0 0 1

The data sent from the sender side

0 1 1 1 1 0 0 1 1 0 0 0 0 0 1 0 1 1 1 0 0 1 0 0 1 0 0 0 0 1 0 1

Checking at the receiver side, adding the first four 8 bit chunks together we get

0 1 1 0 0 1 1 0

adding 1 0 0 1 1 0 0 1 too the sum:

```
  0 1 1 0 0 1 1 0
+ 1 0 0 1 1 0 0 1
-----------------
  1 1 1 1 1 1 1 1
```

taking 1's complement,

0 0 0 0 0 0 0 0.

As the final sum after 1's complement is 0, the receiver side understands that there is no error in the data received.

Apurva Kurala
22BCE3799

CODE:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define BUFFER_SIZE 1024

unsigned char compute_checksum(const unsigned char *buffer, int size) {
    unsigned int total = 0;
    for (int i = 0; i < size; ++i) {
        total += buffer[i];
        if (total > 0xFF) {
            total = (total & 0xFF) + 1;
        }
    }
    return ~((unsigned char)total);
}

int validate_checksum(const unsigned char *buffer, int size, unsigned char checksum) {
    unsigned int total = 0;
    for (int i = 0; i < size; ++i) {
        total += buffer[i];
        if (total > 0xFF) {
            total = (total & 0xFF) + 1;
        }
    }
    total += checksum;
    if (total > 0xFF) {
        total = (total & 0xFF) + 1;
    }
    return (total == 0xFF);
}

void display_binary(unsigned char byte) {
    for (int i = 7; i >= 0; --i) {
        putchar((byte & (1 << i)) ? '1' : '0');
    }
}
```

```c
int main() {
    char input[BUFFER_SIZE];
    unsigned char buffer[BUFFER_SIZE / 8];
    int size = 0;

    printf("Input binary data: ");
    fgets(input, sizeof(input), stdin);

    char *segment = strtok(input, " \n");
    while (segment != NULL) {
        if (strlen(segment) == 8) {
            unsigned char byte = (unsigned char)strtol(segment, NULL, 2);
            buffer[size++] = byte;
        }
        segment = strtok(NULL, " \n");
    }

    unsigned char checksum = compute_checksum(buffer, size);

    printf("Data Segments:\n");
    for (int i = 0; i < size; ++i) {
        display_binary(buffer[i]);
        putchar('\n');
    }

    printf("Checksum:\n");
    display_binary(checksum);
    putchar('\n');

    unsigned char transmitted_data[size + 1];
    memcpy(transmitted_data, buffer, size);
    transmitted_data[size] = checksum;

    int valid = validate_checksum(transmitted_data, size, checksum);

    printf("Validation Result:\n");
    if (valid) {
        printf("Data verified.\n");
```

```
    } else {
        printf("Data corrupted hence error.\n");
    }

    return 0;
}
```

OUTPUT:

```
Input binary data: 01111001100000101110010010000101
Data Segments:
Checksum:
11111111
Validation Result:
Data verified.
```