UNIT-3
DATA LINK LAYER

Syllabus:

Data link layer: Design issues, Framing: fixed size framing, variable size framing, flow control, error control, error detection and correction, CRC, Checksum: idea, one's complement internet checksum, services provided to Network Layer, Elementary Data Link Layer protocols: simplex protocol, Simplex stop and wait, Simplex protocol for Noisy Channel. Sliding window protocol: One bit, Go back N, Selective repeat-Stop and wait protocol, Data link layer in HDLC: configuration and transfer modes, frames, control field, point to point protocol (PPP): framing transition phase, multiplexing, multi-link PPP.

Design issues of data link layer:

- **Frame synchronization:** Data are sent in blocks called frames. The beginning and end of each frame must be recognizable.
- **Flow control:** The sending station must not send frames at a rate faster than the receiving station can absorb them.
- **Error control:** Bit errors introduced by the transmission system should be corrected.
- **Addressing:** On a shared link, such as a local area network (LAN), the identity of the two stations involved in a transmission must be specified.
- **Access Control:** It is usually not desirable to have a physically separate communications path for control information. Accordingly, the receiver must be able to distinguish control information from the data being transmitted.
- **Link management:** The initiation, maintenance, and termination of a sustained data exchange require a fair amount of coordination and cooperation among stations. Procedures for the management of this exchange are required.

Framing:

Framing in the data link layer separates a message from one source to a destination, or from other messages to other destinations, by adding a sender address and a destination address. The destination address defines where the packet is to go; the sender address helps the recipient acknowledge the receipt.

- Framing can be done in two ways:
    1. Fixed Size Framing
    2. Variable Size Framing
- **Fixed Size Framing:**
    - In fixed-size framing, there is no need for defining the boundaries of the frames; the size itself can be used as a delimiter.
      Example: ATM Wide Area Network.
- **Variable-Size Framing:**
    - In variable-size framing, we need a way to define the end of the frame and the beginning of the next.
    - There are two approaches were used for this purpose:
        1. **Character-Oriented Approach** and
        2. **Bit-Oriented Approach**.

1. *Character-Oriented Framing Protocols:*

- In a character-oriented protocol, data to be carried are 8-bit characters from a coding system such as ASCII.
- The header, which normally carries the source and destination addresses and other control information, and the trailer, which carries error detection or error correction redundant bits, are also multiples of 8 bits.
- To separate one frame from the next, an 8-bit (1-byte) flag is added at the beginning and the end of a frame.
- The flag, composed of protocol-dependent special characters, signals the start or end of a frame.
- Following figure shows the format of a frame in a character-oriented protocol.

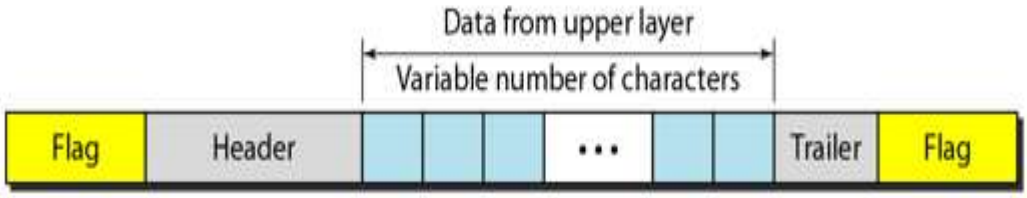


Figure: A frame in a character-oriented protocol

- Character-oriented framing was popular when only text was exchanged by the data link layers.
- The flag could be selected to be any character not used for text communication.
- **Byte stuffing** is a process of adding 1-extra byte whenever there is a flag or escape character in the text.

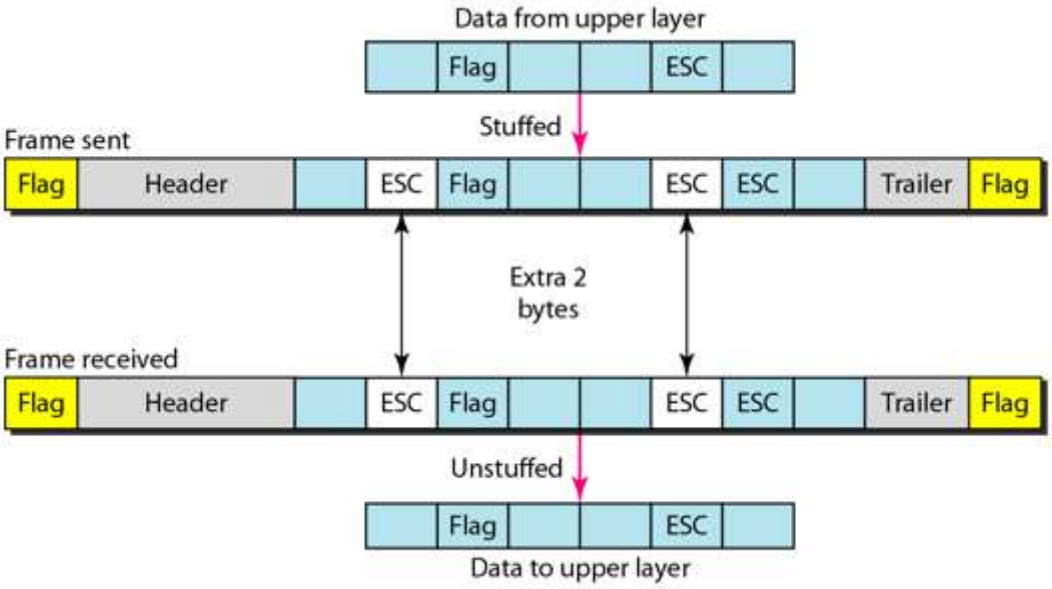


Figure: Byte stuffing and un-stuffing

2. Bit-Oriented Protocol Approach:

- A protocol in which the data frame is interpreted as a sequence of bits.
- In this method, each frame begins & ends with a special bit pattern 01111110 called ***Flags.***

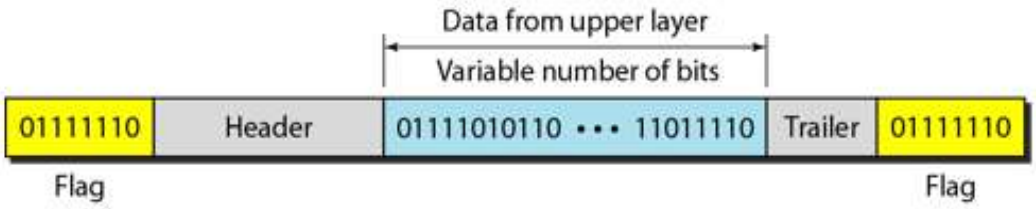


Figure: ***A frame in a bit-oriented protocol***

- **Bit stuffing** is a process of adding one extra bit **0** whenever five consecutive **1**'s follow a **0** in the data, so that the receiver does not mistake the pattern **01111110** for a flag.

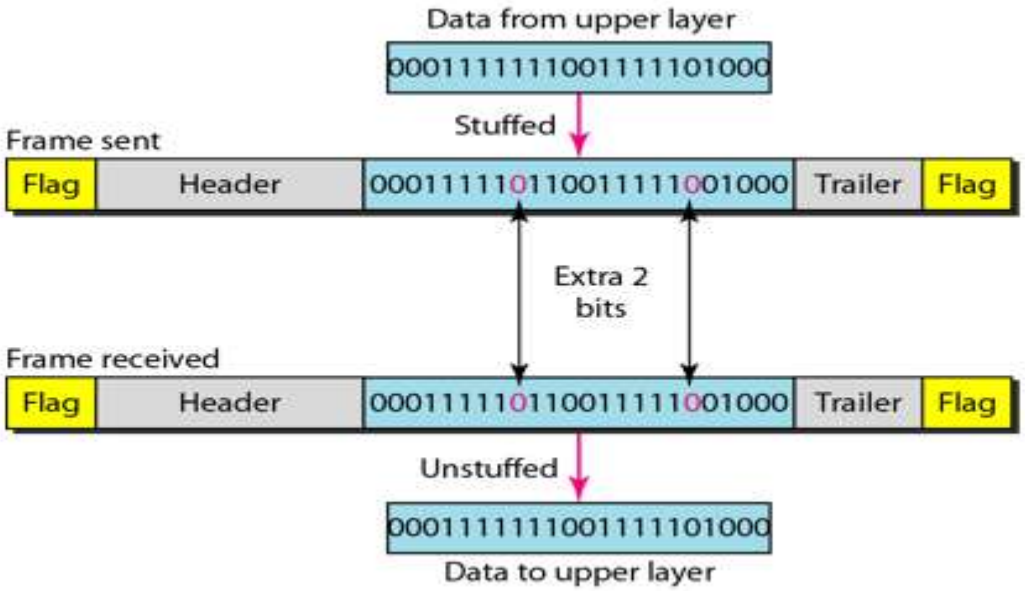
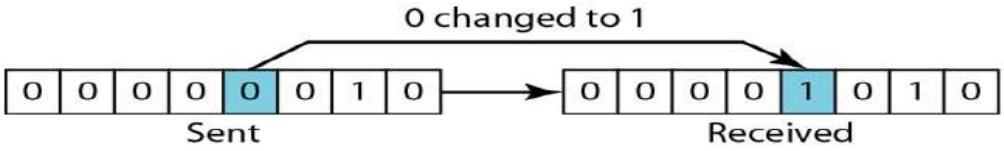


Figure: Bit stuffing and un-stuffing

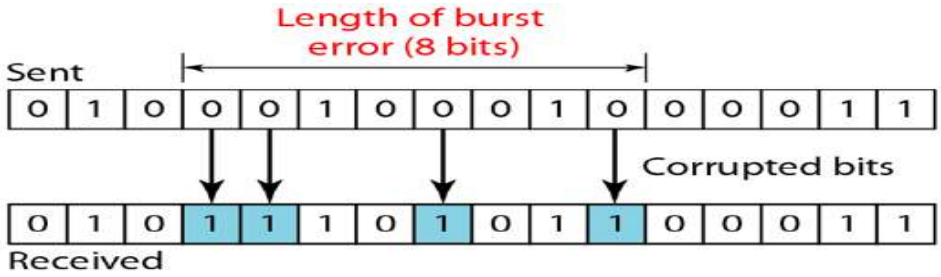
Error Control:

- The bit stream transmitted by the physical layer is not guaranteed to be error free.
- The data link layer is responsible for error detection and correction.
- The most common error control method is to compute and append some form of a checksum to each outgoing frame at the sender's data link layer and to recomputed the checksum and verify it with the received checksum at the receiver's side. If both of them match, then the frame is correctly received; else it is erroneous.
- Error control is both ***Error Detection*** and ***Error Correction.***

- **Types of Errors:**
  - **Single-Bit error:** The term ***single-bit error*** means that only 1 bit of a given data unit (such as a byte, character, or packet) is changed from 1 to 0 or from 0 to 1.



  - **Burst Error:** A burst error means that 2 or more bits in the data unit have changed.



Redundancy

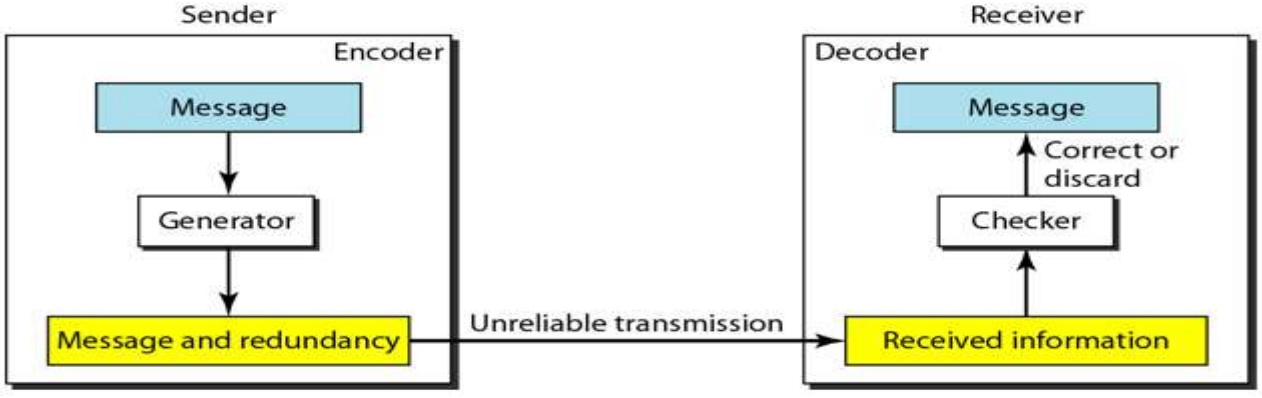
The central concept in detecting or correcting errors is **redundancy.** To be able to detect or correct errors, we need to send some extra bits with our data. These redundant bits are added by the sender and removed by the receiver. Their presence allows the receiver to detect or correct corrupted bits.

To detect or correct errors, we need to send extra (redundant) bits with data.

Coding

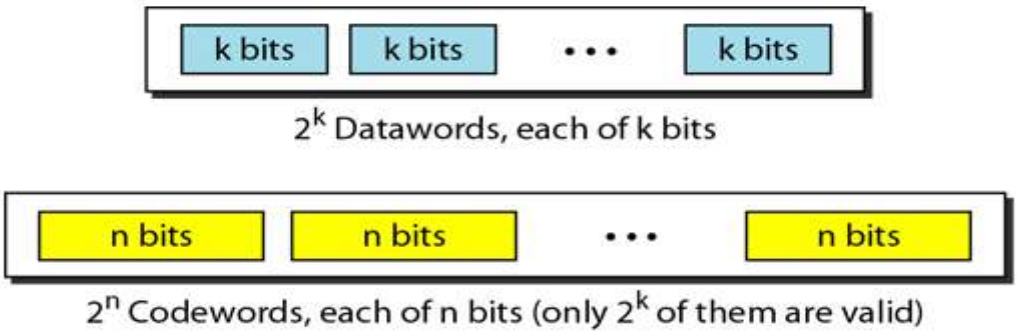
Redundancy is achieved through various coding schemes. The sender adds redundant bits through a process that creates a relationship between the redundant bits and the actual data bits. The receiver checks the relationships between the two sets of bits to detect or correct the errors. The ratio of redundant bits to the data bits and the robustness of the process are important factors in any coding scheme. Figure shows the general idea of coding.

We can divide coding schemes into two broad categories: **block coding** and **convolution coding**(more complex thats why we use block coding).



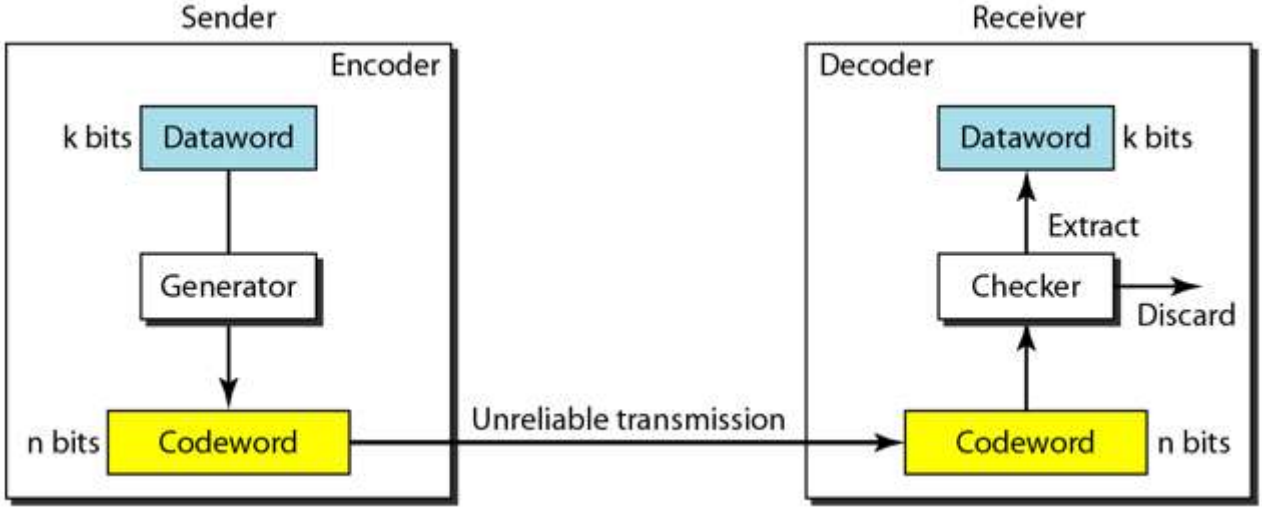
Block Coding:

- In block coding, we divide our message into blocks, each of k bits, called data words. We add r redundant bits to each block to make the length n = k + r. The resulting n-bit blocks are called codewords.

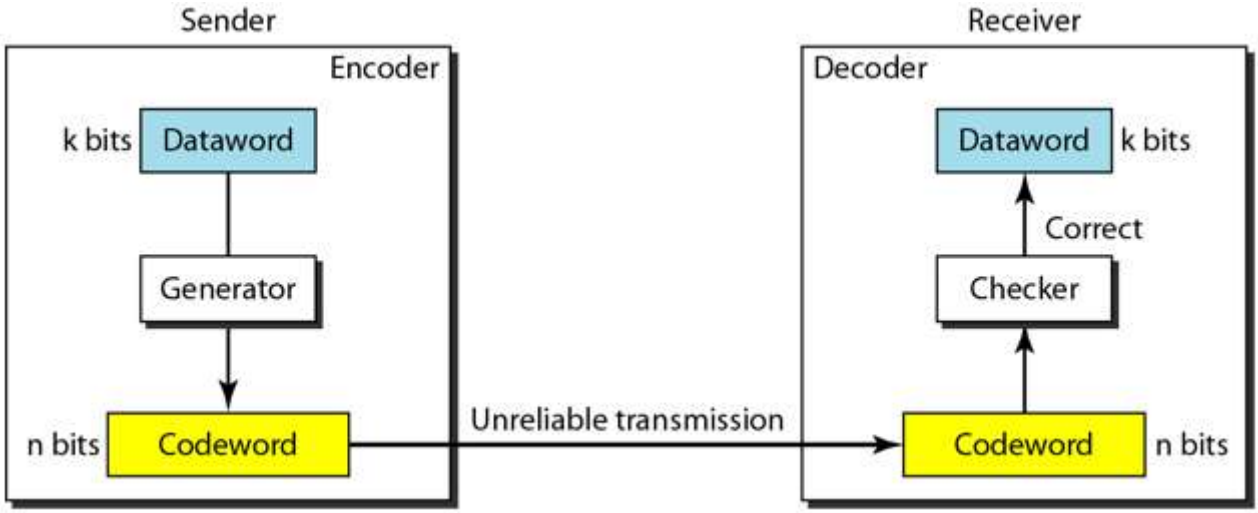


- Error detection and correction process in block coding.

Error Detection in Block Coding:



Error Correction Block Coding:



Error Handling Techniques:

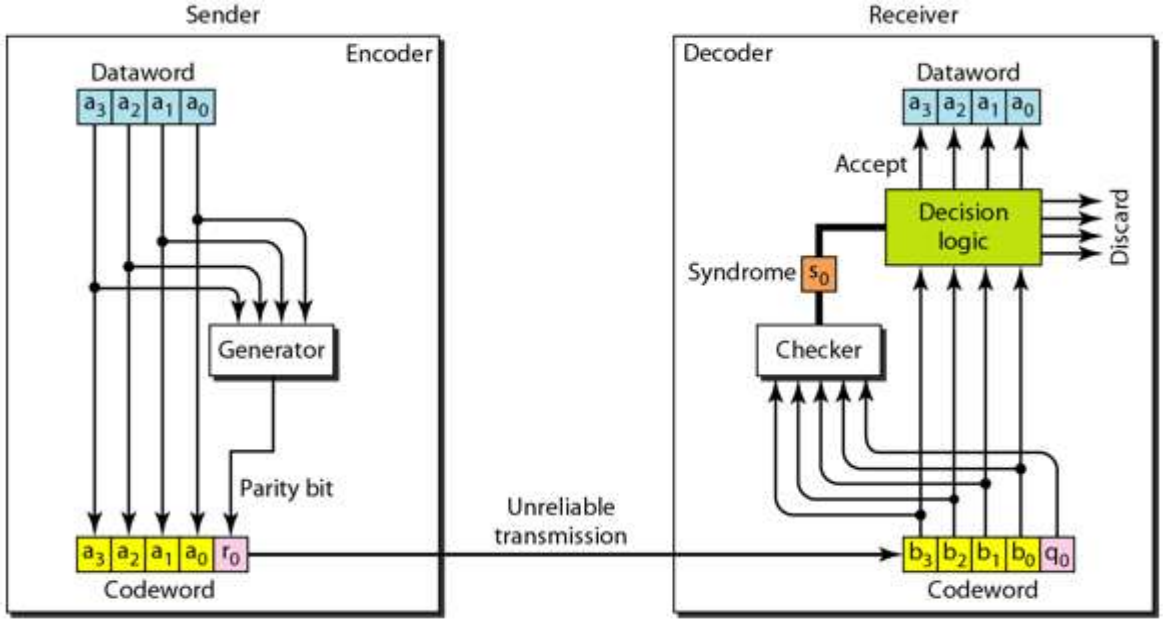
- Following are some of the error detection techniques
  - Parity Checking - used for linear-codes
  - Cyclic Redundancy Check (CRC) - used for Cyclic-codes
  - Checksum
  - Hamming Code (used for both error detection and correction)

Parity checking:

- Parity checking is one of the error detection technique.
- In parity checking process while transmission of data additionally we append one extra bit called parity bit.

Data bits	Parity bit

- Parity bit is decided based on number of binary 1's in data section.
- **Types of parity:**
  Even parity: If number of 1's in data is **even** then parity bit is '0', otherwise parity bit is '1'.
  Odd parity: If number of 1's in data is **odd** then parity bit is '0', otherwise parity bit is '1'.
- Encoding and decoding process in parity checking (mostly we use even parity).



The encoder uses a generator that takes a copy of a 4-bit dataword (a_0, a_1, a_2, and a_3) and generates a parity bit r_0. The dataword bits and the **parity bit** create the 5-bit codeword. The parity bit that is added makes the number of 1s in the codeword even. This is normally done by adding the 4 bits of the dataword (modulo-2); the result is the parity bit. In other words,

$$r_0 = a_3 + a_2 + a_1 + a_0 \qquad \text{(modulo-2)}$$

If the number of 1s is even, the result is 0; if the number of 1s is odd, the result is 1. In both cases, the total number of 1s in the codeword is even.

The sender sends the codeword which may be corrupted during transmission. The receiver receives a 5-bit word. The checker at the receiver does the same thing as the generator in the sender with one exception: The addition is done over all 5 bits. The result, which is called the **syndrome,** is just 1 bit. The syndrome is 0 when the number of 1s in the received codeword is even; otherwise, it is 1.

$$s_0 = b_3 + b_2 + b_1 + b_0 + q_0 \quad \text{(modulo-2)}$$

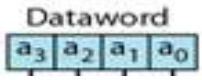
The syndrome is passed to the decision logic analyzer. If the syndrome is 0, there is no error in the received codeword; the data portion of the received codeword is accepted as the dataword; if the syndrome is 1, the data portion of the received codeword is discarded. The dataword is not created.

- **Example:**

Data	Even Parity	Odd Parity
1001	1001 | **0** Number of 1's in data is even that's why parity bit is **0**	1001 | **1** Number of 1's in data is even that's why parity bit is **1**
10101110	10101110 | **1** Number of 1's in data is odd that's why parity bit is **1**	10101110 | **0** Number of 1's in data is odd that's why parity bit is **0**

Procedure of Encoding and decoding process in parity checking:

Encoding at sender site: (consider even parity)



Consider 4-bit Data word: 1001

Parity bit $r_0 = (a_3+a_2+a_1+a_0)$ modulo 2

Here $a_3=1$, $a_2=0$, $a_1=0$, $a_0=1$

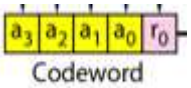
Therefore $r_0 = (1+0+0+1) \% 2$

$= 2 \% 2$

$= 0$

So resultant parity bit **r_0=0,**

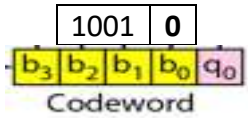
Final **code word** is



1001 | **0**

Decoding process at receiver site:

Code word from sender is



b_3=1, b_2=0, b_1=0, b_0=1, r_0

In Decoding process, **Checker** takes the code word and performs the following,

$S_0 = (b_3+b_2+b_1+b_0+ r_0)$ modulo 2 = (1+0+0+1+0) % 2

= 2 % 2 = 0

Resultant **Syndrome S_0** is **0**

Therefore, **No Error in data so receiver accepts the data**.

Consider the case when error Code word is transmitted to receiver:

Problem: Consider the code word that is transmitted to the receiver is '10011', find out the code word has error or not. by considering even parity.

Solution: given code word is 10011

1001	**1**

In data section given data bits respectively b_3=1, b_2=0, b_1=0, b_0=1

In Decoding process, **Checker** takes the code word and performs the following,

$S_0 = (b_3+b_2+b_1+b_0+ r_0)$ modulo 2

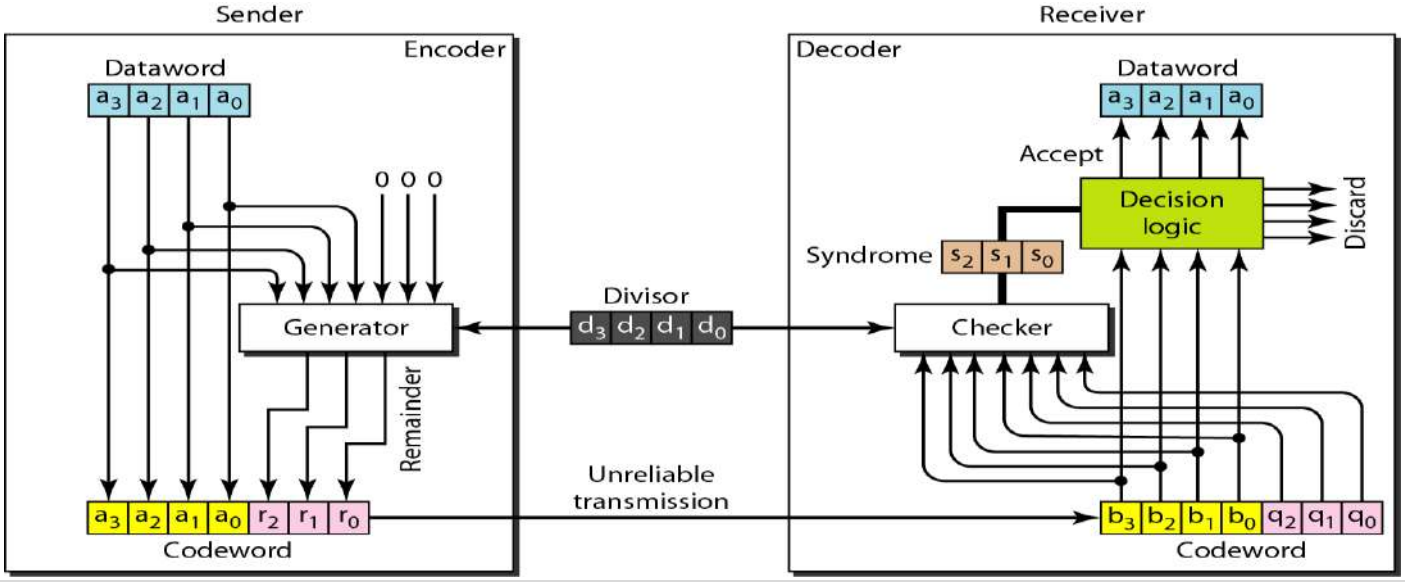
= (1+0+0+1+1) % 2 = 3 % 2 = 1

Resultant **Syndrome S_0** is **1**

Therefore, **Error in data So receiver discards the data.**

Cyclic Redundancy Checking(CRC): Cyclic redundancy checking is a method of checking for errors in data that has been transmitted on a communications link. A sending device applies a generator polynomial to a block of data that is to be transmitted and appends the resulting cyclic redundancy code (CRC) to the block. The receiving end applies the same polynomial to the data and compares its result with the result appended by the sender. If they agree, the data has been received successfully. If not, the sender can be notified to resend the block of data.

Encoding and Decoding process in CRC:

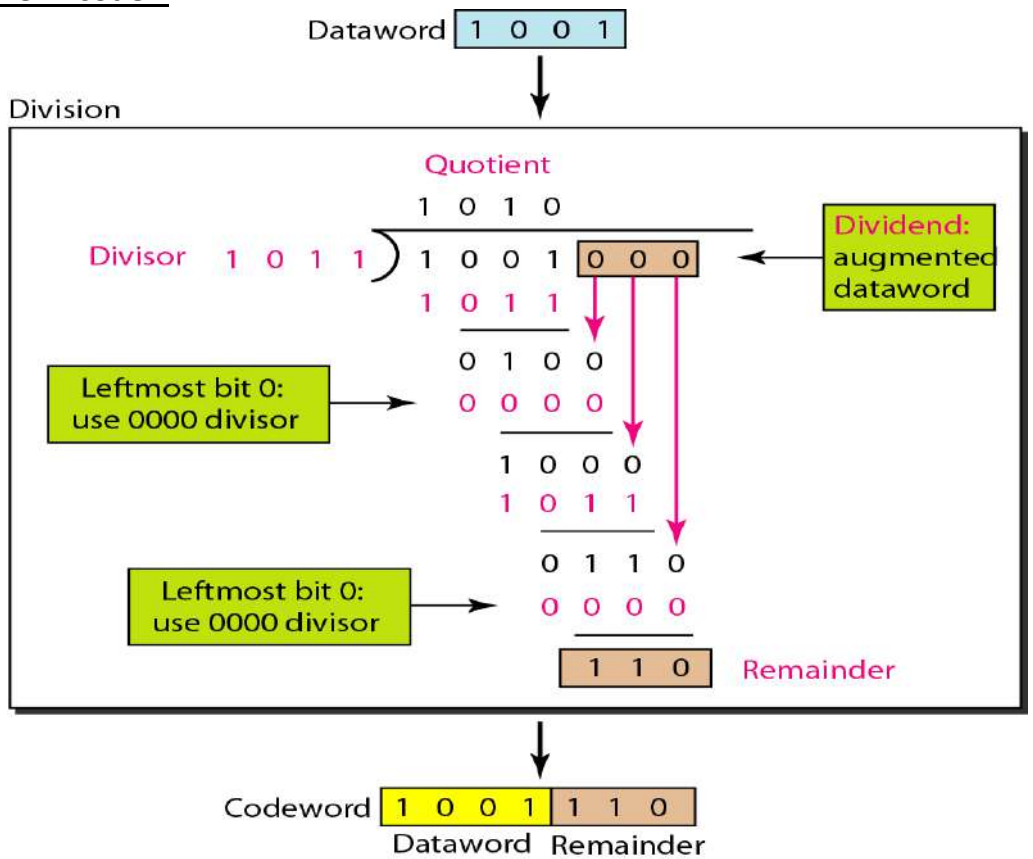


- In CRC we use Binary division, binary division is nothing but EX-OR operation

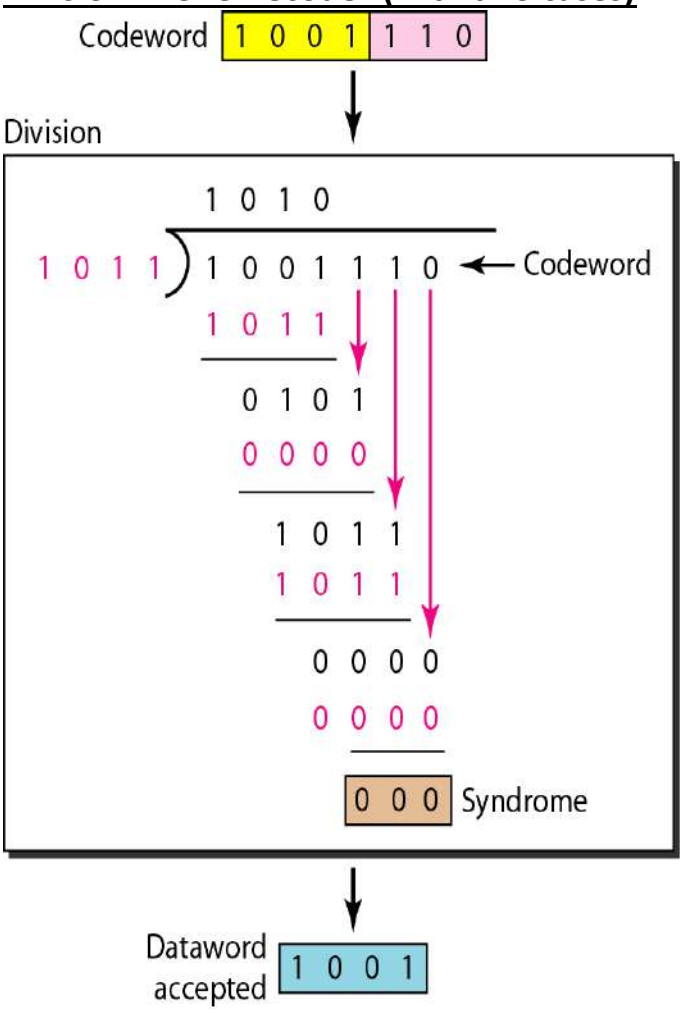
A	B	Z=A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

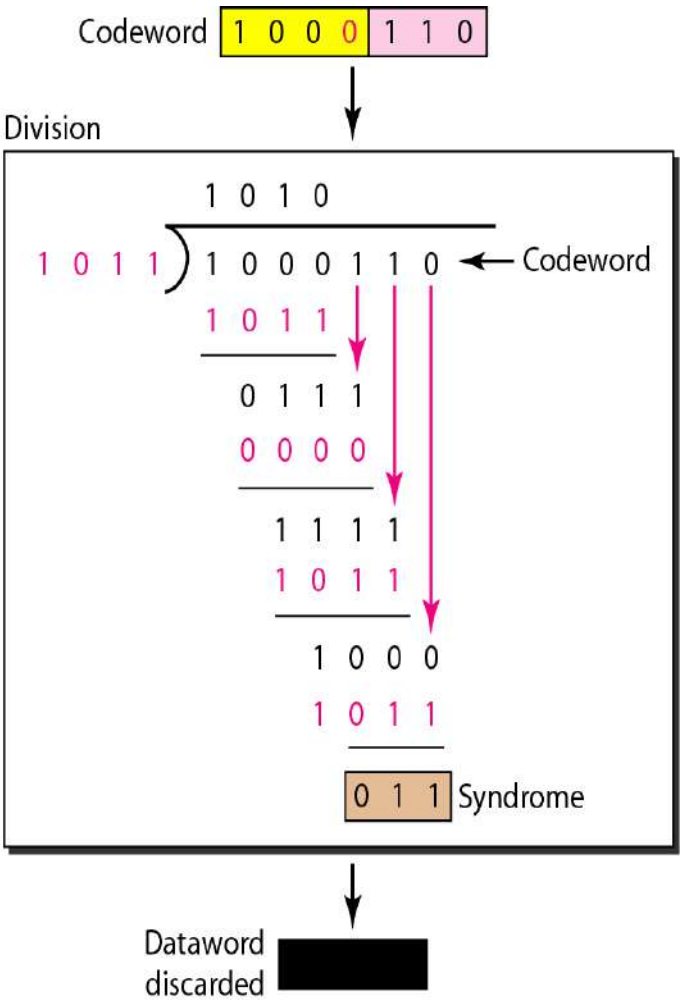
- Before performing division, we have to append some redundant bits to data word, number of redundant bits decided based on number of bits in generator (Divisor).
- If **number of bits in generator is "n"** then we have to **append "n-1" 0's** to data word.
- After appending zeros generated word is called as **augmented data word**.
- Now perform binary division at sender site i.e. **encoding** process with following data
  Divisor: CRC generator.
  Dividend: Augmented data word.
- After division we get remainder i.e. CRC (Cyclic Redundancy Code).
- With the help of remainder, generate a **Code word** by replacing appended 0's in augmented data word by CRC and this code word is transmitted to receiver for decoding process.
- In **decoding** process again perform division with following data
  Divisor: CRC generator
  Dividend: Code word
- After division we get remainder. This remainder is called as Syndrome.
- If Syndrome contains all 0's, then accept the data, otherwise discard the data and ask for retransmission.

Division in CRC Encoder:

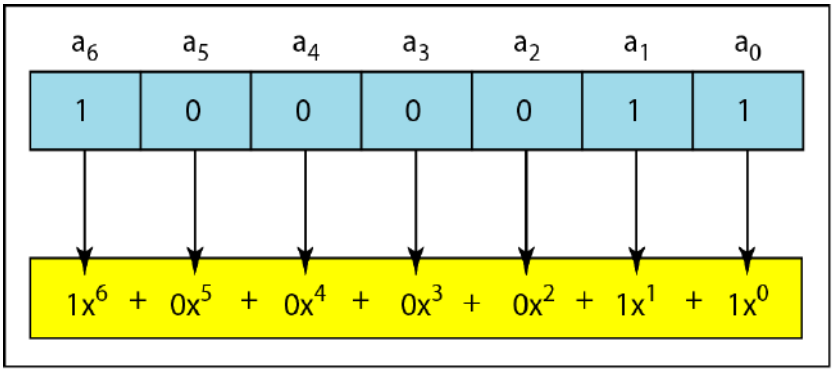


Division in CRC Decoder:(with two cases)

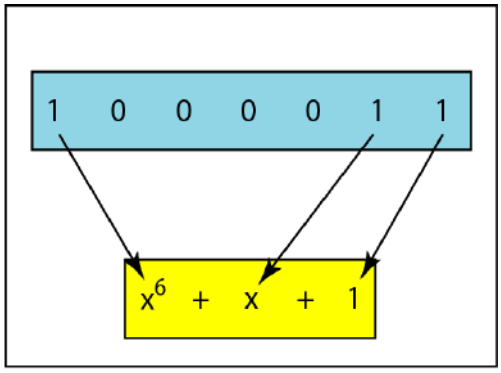




Polynomial to represent a binary word:



a. Binary pattern and polynomial



b. Short form

Checksum:

- A checksum is an error-detection method, the transmitter computes a numerical value according to the number of set or unset bits in a message and sends it along with each message frame.
- At the receiver end, the same checksum function (formula) is applied to the message frame to retrieve the numerical value.
- If the received checksum value matches the sent value, the transmission is considered to be successful and error-free.
- A checksum may also be known as a hash sum.
- Both TCP and UDP communication layers provide a checksum count and verification as one of their services.
- **By sending sum along with data:**

  Suppose our data is a list of five 4-bit numbers that we want to send to a destination. In addition to sending these numbers, we send the sum of the numbers. For example, if the set of numbers is (7, 11, 12, 0, 6), we send (7, 11, 12, 0, 6, **36**), where 36 is the sum of the original numbers. The receiver adds the five numbers and compares the result with the sum. If the two are the same, the receiver assumes no error, accepts the five numbers, and discards the sum. Otherwise, there is an error somewhere and the data are not accepted.

- **By sending negative sum along with data:**

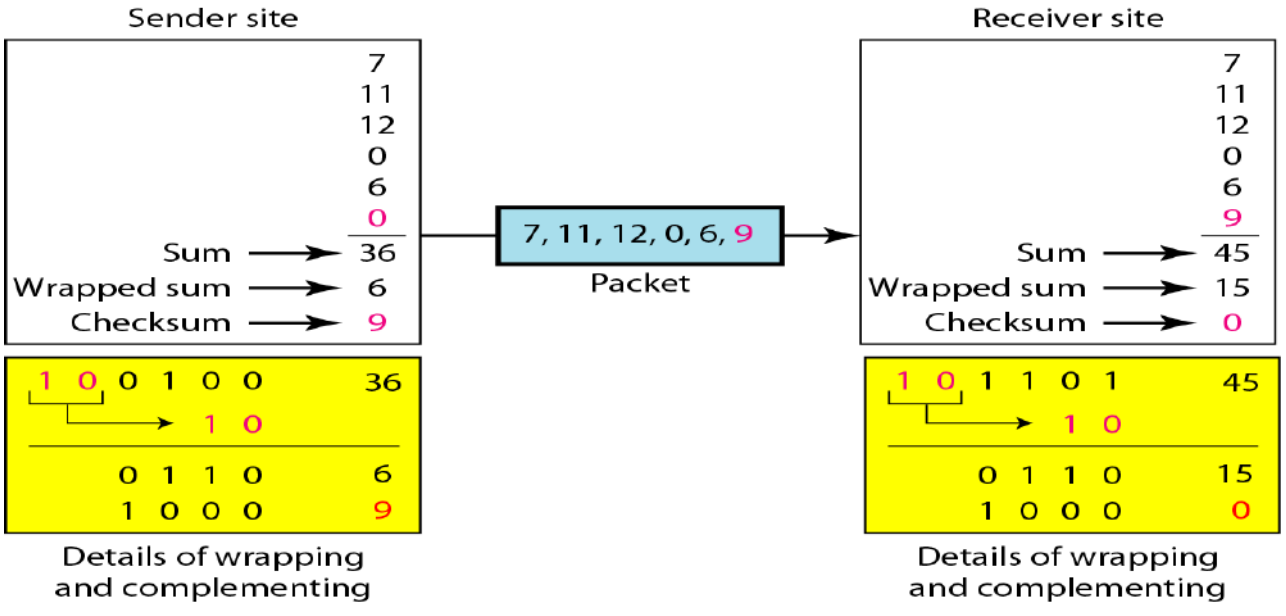
  We can make the job of the receiver easier if we send the negative (complement) of the sum, called the *checksum*. In this case, we send (7, 11, 12, 0, 6, **−36**). The receiver can add all the numbers received (including the checksum). If the result is 0, it assumes no error; otherwise, there is an error.

- **One's complement arithmetic:**

  In one's complement arithmetic if we take some n-bit numbers, after addition we may get a resultant sum that can exceed n-bits, in that case we wrap the resultant sum into n-bit number.

  Consider The number 21 in binary is 10101 (it needs five bits). We can wrap the leftmost bit and add it to the four rightmost bits. We have (0101 + 1) = 0110 or 6.

  Example:



Internet Checksum:

Traditionally, the Internet has been using a 16-bit checksum. The sender calculates the checksum by following these steps.

Sender site:

1. **The message is divided into 16-bit words.**
2. **The value of the checksum word is set to 0.**
3. **All words including the checksum are added using one's complement addition.**
4. **The sum is complemented and becomes the checksum.**
5. **The checksum is sent with the data.**

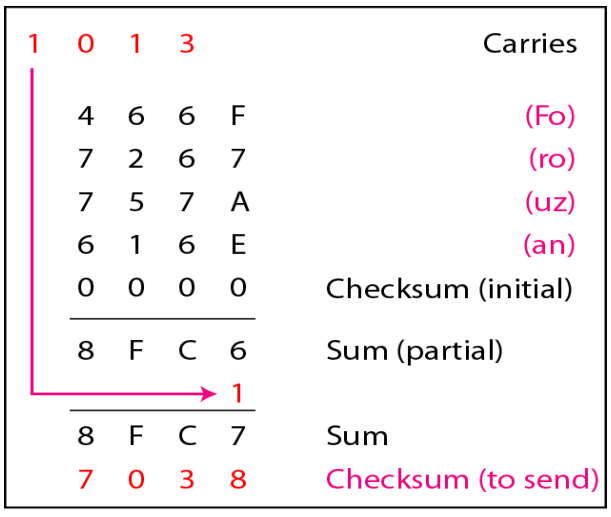
The receiver uses the following steps for error detection.

Receiver site:

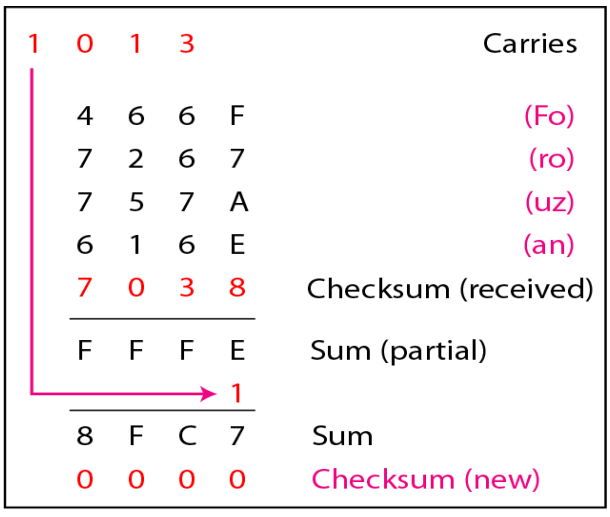
1. **The message (including checksum) is divided into 16-bit words.**
2. **All words are added using one's complement addition.**
3. **The sum is complemented and becomes the new checksum.**
4. **If the value of checksum is 0, the message is accepted; otherwise, it is rejected.**

Example:

- Let us calculate the checksum for a text of 8 characters ("Forouzan").
- The text needs to be divided into 2-byte (16-bit) words. We use ASCII to change each byte to a 2-digit hexadecimal number.
- For example, **F** is represented as **46** and **o** is represented as **6F**. Bellow figure shows how the checksum is calculated at the sender and receiver sites.
- In part a of the figure, the value of partial sum for the first column is 36. We keep the rightmost digit (6) and insert the leftmost digit (3) as the carry in the second column. The process is repeated for each column.



a. Checksum at the sender site



a. Checksum at the receiver site

Note: For detailed explanation of Internet checksum calculation refer the class notes

Hamming Code:

- Hamming is given by Richard W. Hamming, it is used for both error-detection and correction (single-bit).
- Mostly we use a 7-bit hamming code.
- A 7-bit hamming code is the combination of 4-Data bits and 3- Parity bits.
- To construct hamming code, first find out the positions of parity bits using following formula, rest of the positions are allotted for data bits.

Position of parity bit= 2^n (n=0,1,2,3, ….)

Positions of parity bits:

When n=0→2^0=1, parity bit Position is 1, i.e. P_1

When n=1→2^1=2, parity bit Position is 2, i.e. P_2

When n=2→2^2=4, parity bit Position is 1, i.e. P_4

Therefore the 7-bit hamming code structure is

D_7	D_6	D_5	P_4	D_3	P_2	P_1

Data bit combinations for parity bits:

For P_1: Starting from 1st position, check 1-bit and skip 1-bit, check 1-bit and skip 1-bit and so on

P_1 D_3 D_5 D_7

For P_2: Starting from 2nd position, check 2-bits and skip 2-bits, check 2-bits and skip 2-bits and so on

P_2 D_3 D_6 D_7

For P_4: Starting from 4th position, check 4-bits and skip 4-bits, check 4-bits and skip 4-bits and so on

P_4 D_5 D_6 D_7

Example:

Hamming code for the following 4-bit data 1001

D_7	D_6	D_5	P_4	D_3	P_2	P_1

First place given data bits from left to right

1	0	0	P_4	1	P_2	P_1

Now find out the Parity bits based on data bit combinations by considering even parity

For P_1: P_1 D_3 D_5 D_7 → 0 1 0 1 So P_1= 0

For P_2: P_2 D_3 D_6 D_7 → 0 1 0 1 So P_2= 0

For P_4: P_4 D_5 D_6 D_7 → 1 0 0 1 So P_4= 1

Therefore, hamming code is

1	0	0	1	1	0	0

Error detection and correction in hamming code:

Example: Let us consider the following hamming code that is transmitted to the receiver,

1	0	1	1	1	0	0

In above hamming code, lets try to find out the error is there or not. If error found correct it.

Note: If error found make parity bit as '1', otherwise make parity bit as '0'

D_7	D_6	D_5	P_4	D_3	P_2	P_1

First take the data combinations for each parity bit.

For P_1: P_1 D_3 D_5 D_7 → 0 1 1 1

Here parity bit is not matched with data so there is an error, make parity bit **P_1= 1**

For P_2: P_2 D_3 D_6 D_7 → 0 1 0 1

Here parity bit is matched with data so no error, make parity bit **P_2= 0**

For P_4: P_4 D_5 D_6 D_7 → 1 1 0 1

Here parity bit is not matched with data so there is an error, make parity bit **P_4= 1**

Now for correcting error, generate an error code with new parity bits,

P_4 P_2 P_1= 1 0 1 → Equivalent decimal value is **5**

Therefore, **error exist at 5th position** in given hamming code, for correcting **invert the bit in 5th position**.

After correction new hamming code is,

1	0	0	1	1	0	0