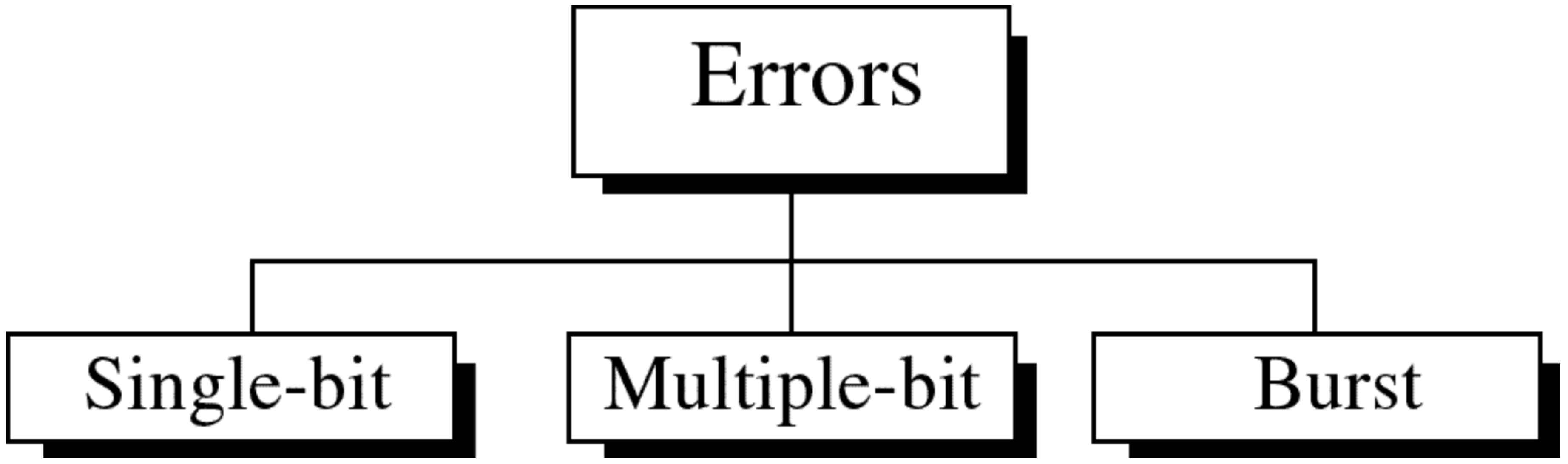
Lecture 10
Error Detection and Correction

- **Types of Errors**
- **Detection**
- **Correction**

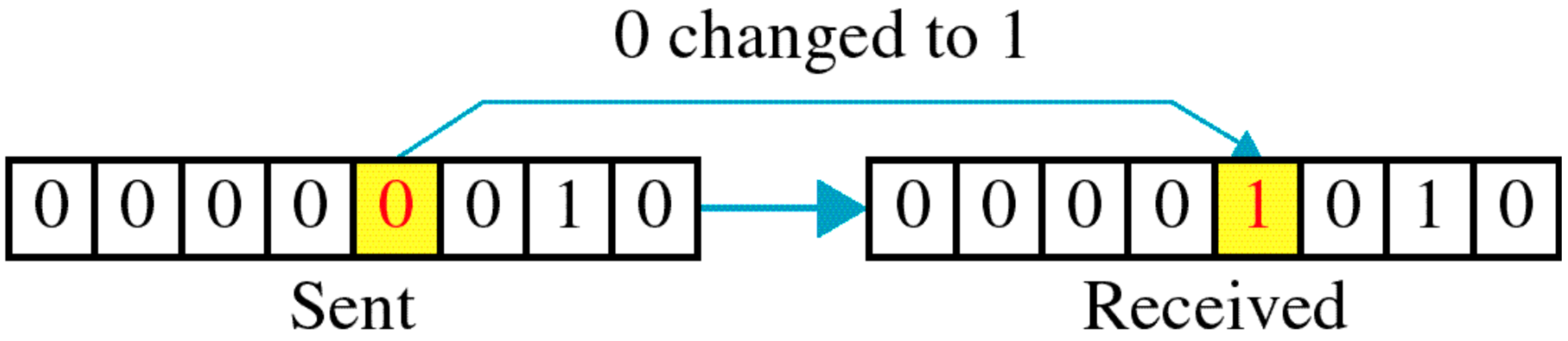
Basic concepts

- ★ Networks must be able to transfer data from one device to another with complete accuracy.
- ★ Data can be corrupted during transmission.
- ★ For reliable communication, errors must be detected and corrected.
- ★ **Error detection and correction** are implemented either at the **data link layer** or the **transport layer** of the OSI model.

Types of Errors



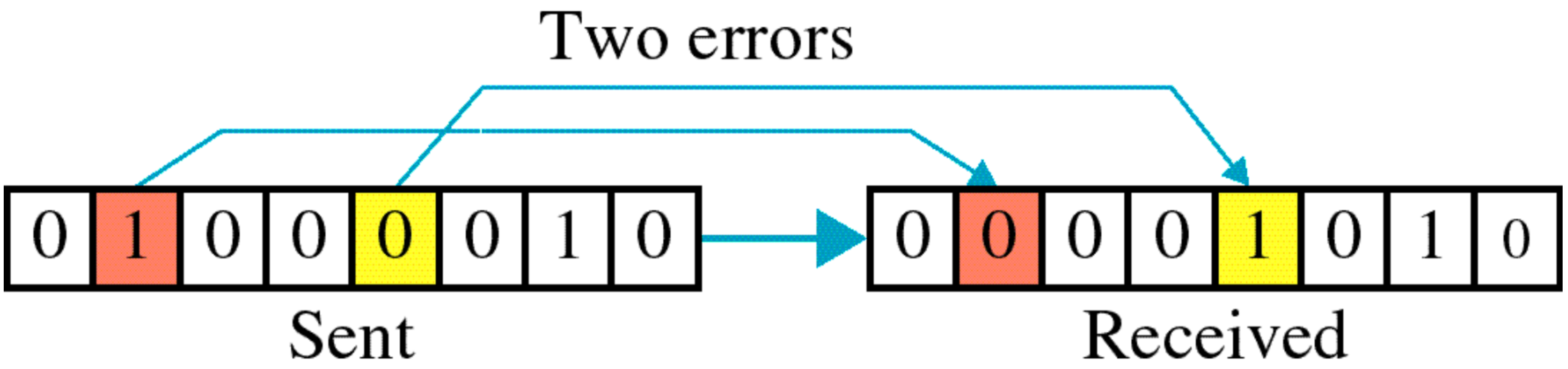
Single-bit error



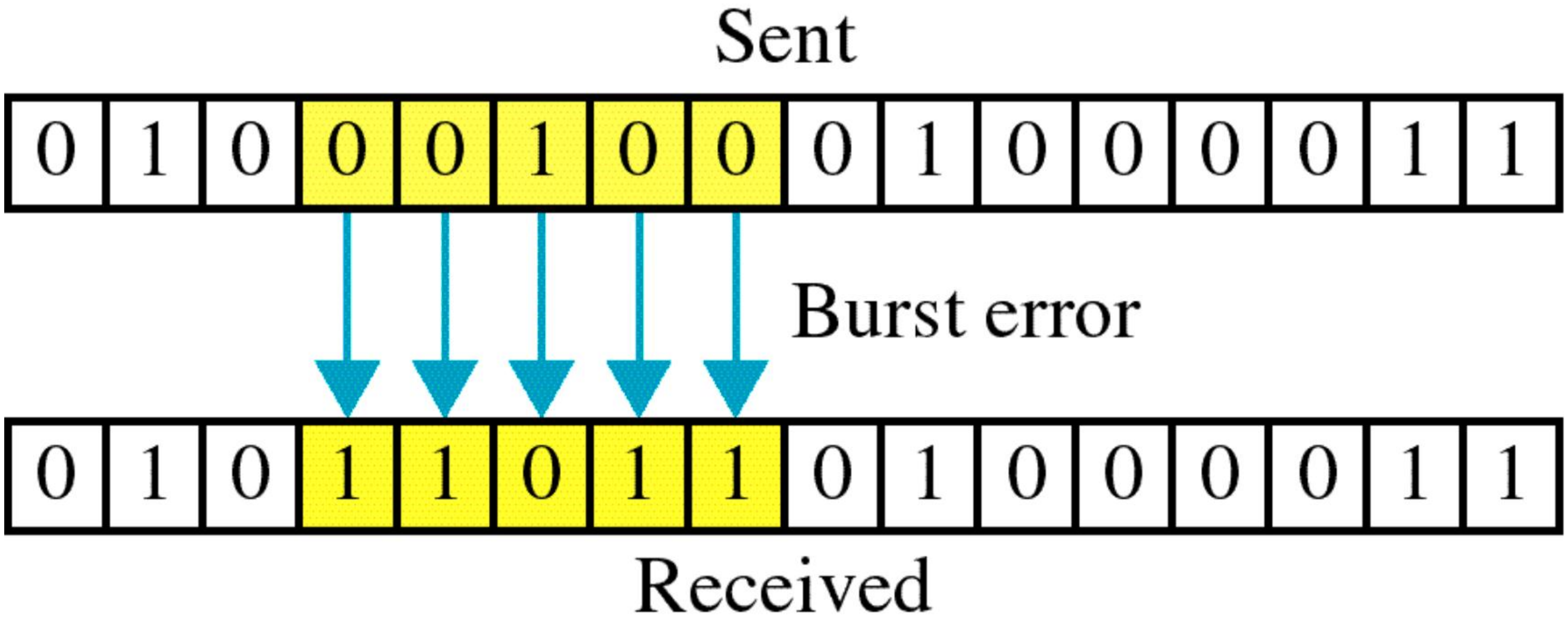
Single bit errors are the **least likely** type of errors in serial data transmission because the noise must have a very short duration which is very rare. However this kind of errors can happen in parallel transmission.

Example:

- ★ If data is sent at 1Mbps then each bit lasts only 1/1,000,000 sec. or 1 μs.
- ★ For a single-bit error to occur, the noise must have a duration of only 1 μs, which is very rare.


Two errors
0 1 0 0 0 0 1 0
0 0 0 0 1 0 1 0
Sent
Received

Burst error


Sent
0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
Burst error
0 1 0 1 1 0 1 1 0 1 0 0 0 0 1 1
Received

The term **burst error** means that two or more bits in the data unit have changed from 1 to 0 or from 0 to 1.

Burst errors does not necessarily **mean that the errors occur in consecutive bits**, the length of the burst is measured from the first corrupted bit to the last corrupted bit. Some bits in between may not have been corrupted.

- ★ **Burst error is most likely to happen in serial transmission** since the duration of noise is normally longer than the duration of a bit.
- ★ The number of bits affected depends on the data rate and duration of noise.

Example:

- ➔ If data is sent at rate = 1Kbps then a noise of 1/100 sec can affect 10 bits.(1/100*1000)

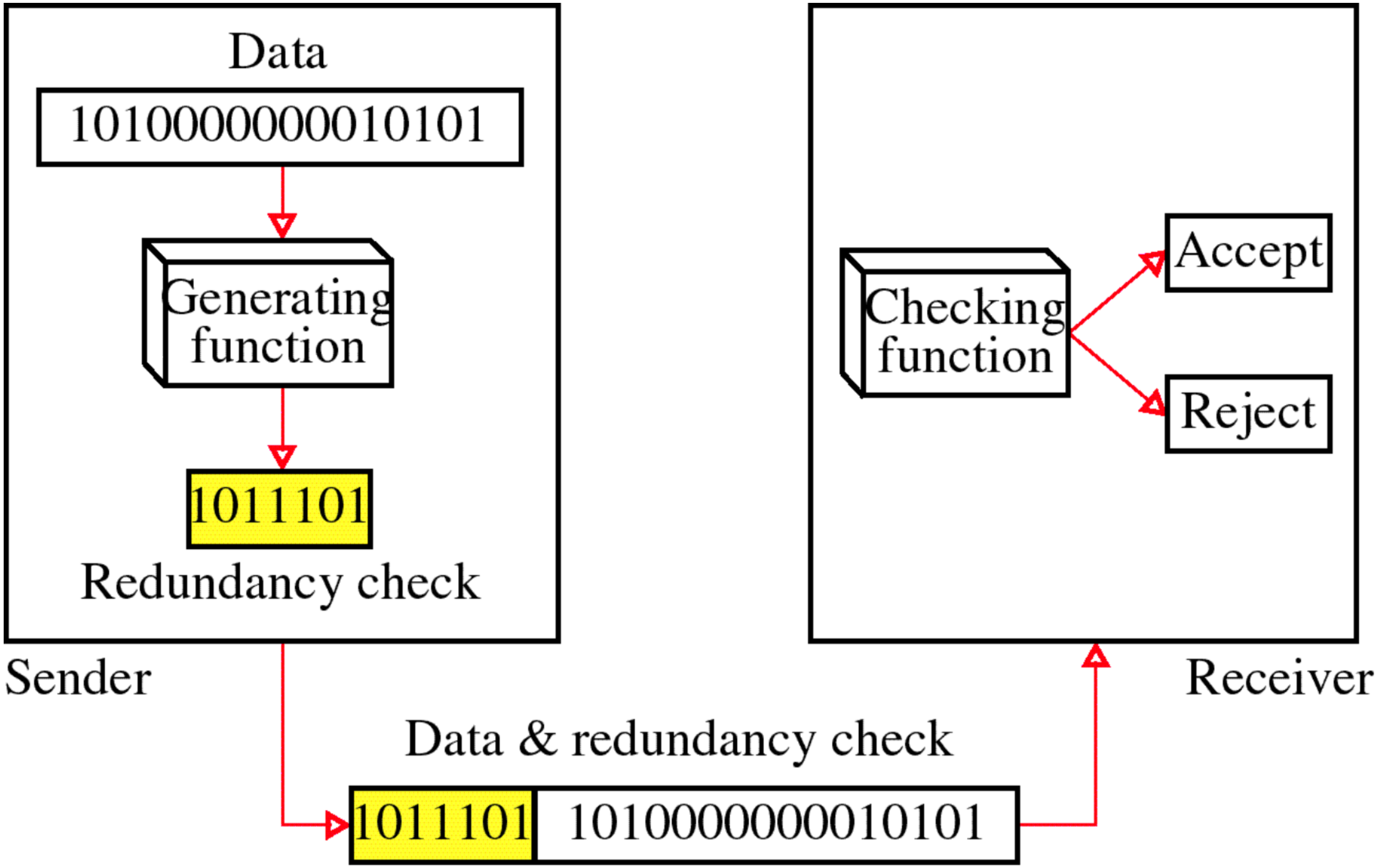
- ➔ If same data is sent at rate = 1Mbps then a noise of 1/100 sec can affect 10,000 bits.(1/100*10^6)

Error detection

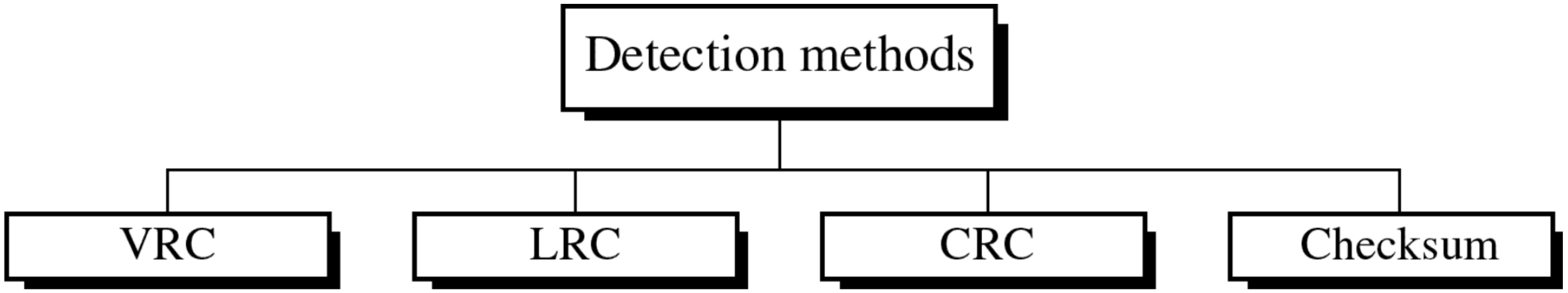
Error detection means to decide whether the received data is correct or not without having a copy of the original message.

Error detection **uses the concept of redundancy**, **which means** adding extra bits for detecting errors at the destination.

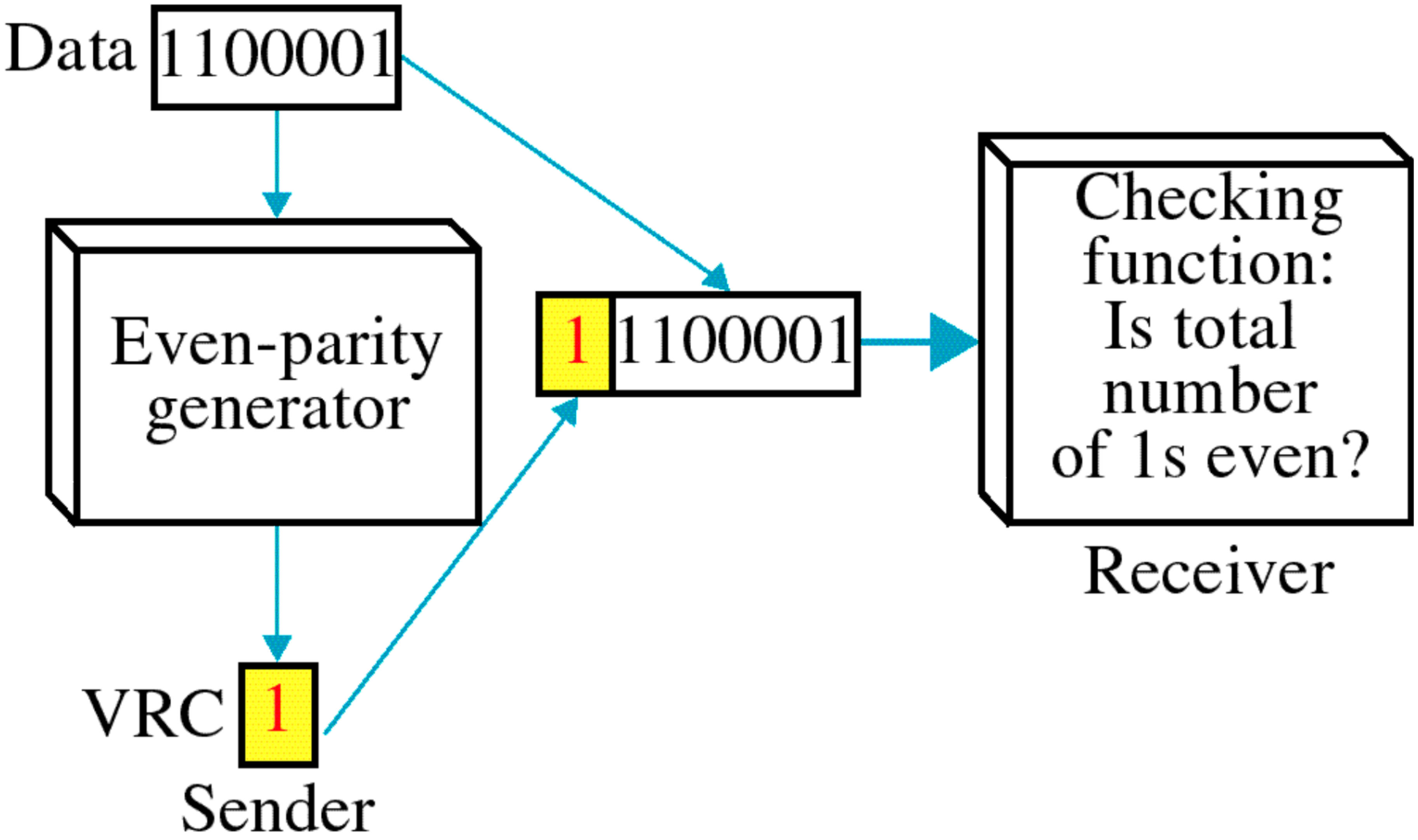
Redundancy

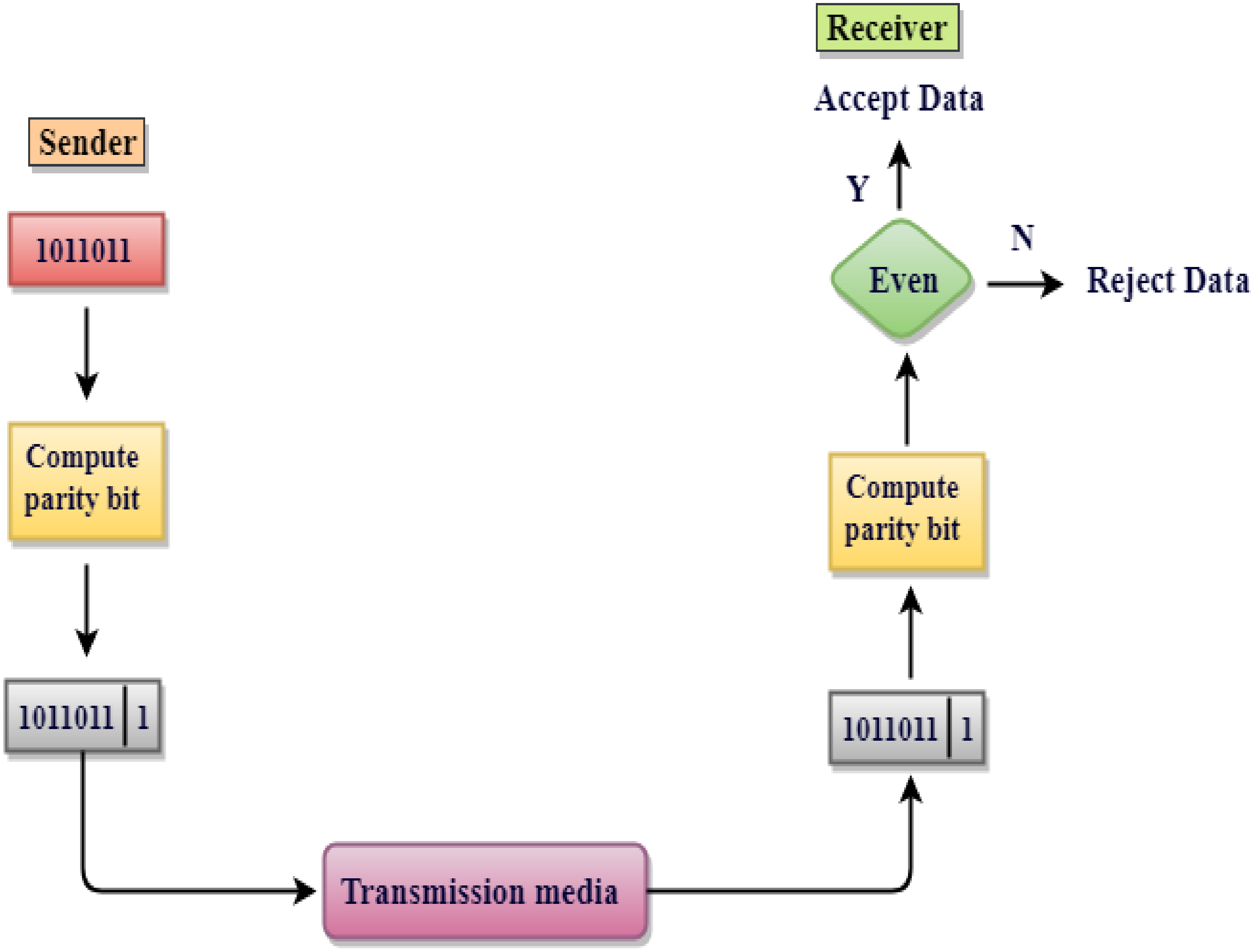

Data
1010000000010101
Generating function
1011101
Redundancy check
Sender
Checking function
Accept
Reject
Receiver
Data & redundancy check
1011101
1010000000010101

Four types of redundancy checks are used in data communications



Vertical Redundancy Check
VRC

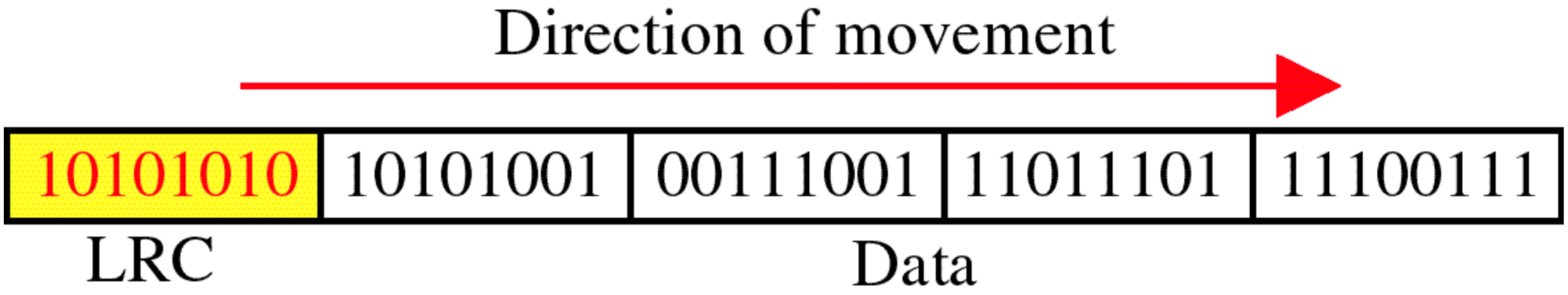



Receiver
Accept Data
Y
N
Even
Reject Data
Sender
1011011
Compute parity bit
1011011 | 1
Compute parity bit
1011011 | 1
Transmission media

Performance

- It can detect single bit error
- It can detect burst errors only if the total number of errors is odd.

Longitudinal Redundancy Check
LRC

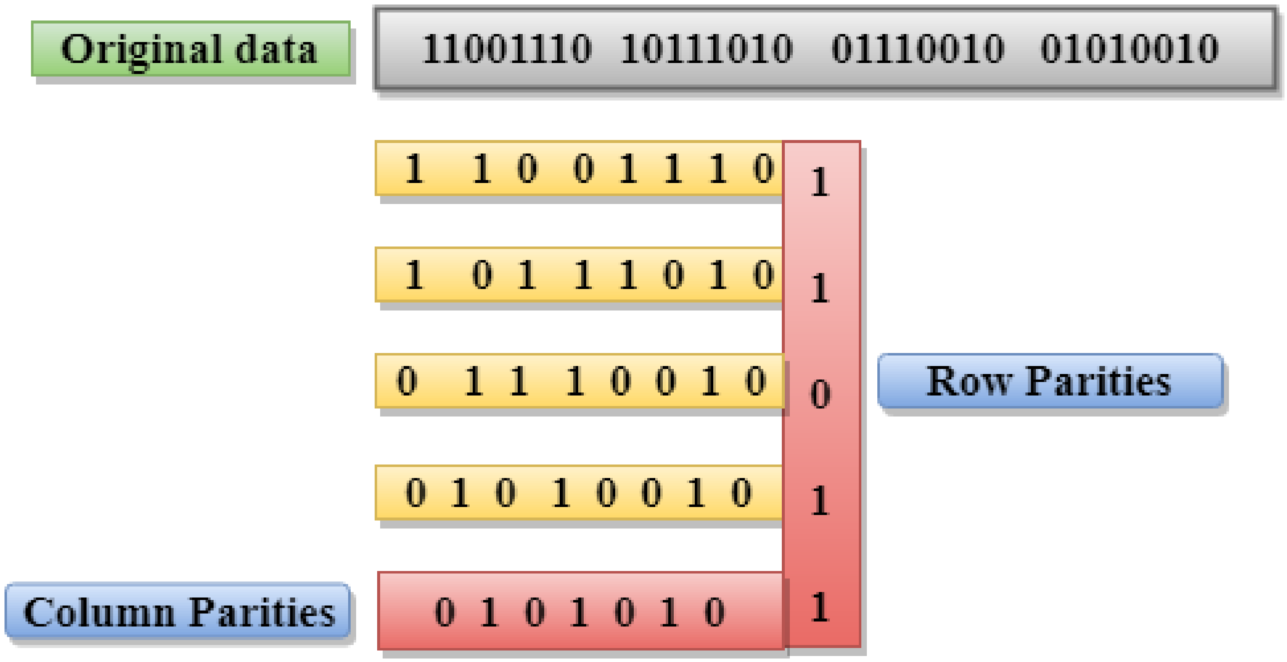


Performance

- ➔LCR increases the likelihood of detecting burst errors.
- ➔If two bits in one data units are damaged and two bits in exactly the same positions in another data unit are also damaged, the LRC checker will not detect an error.

Two-Dimensional Parity Check

- Performance can be improved by using **Two-Dimensional Parity Check** which organizes the data in the form of a table.
- Parity check bits are computed for each row, which is equivalent to the single-parity check.
- In Two-Dimensional Parity check, a block of bits is divided into rows, and the redundant row of bits is added to the whole block.
- At the receiving end, the parity bits are compared with the parity bits computed from the received data.



Drawbacks Of 2D Parity Check

- If two bits in one data unit are corrupted and two bits exactly the same position in another data unit are also corrupted, then 2D Parity checker will not be able to detect the error.
- This technique cannot be used to detect the 4-bit errors or more in some cases.

Cyclic Redundancy Check (CRC)

CRC is a redundancy error technique used to determine the error.

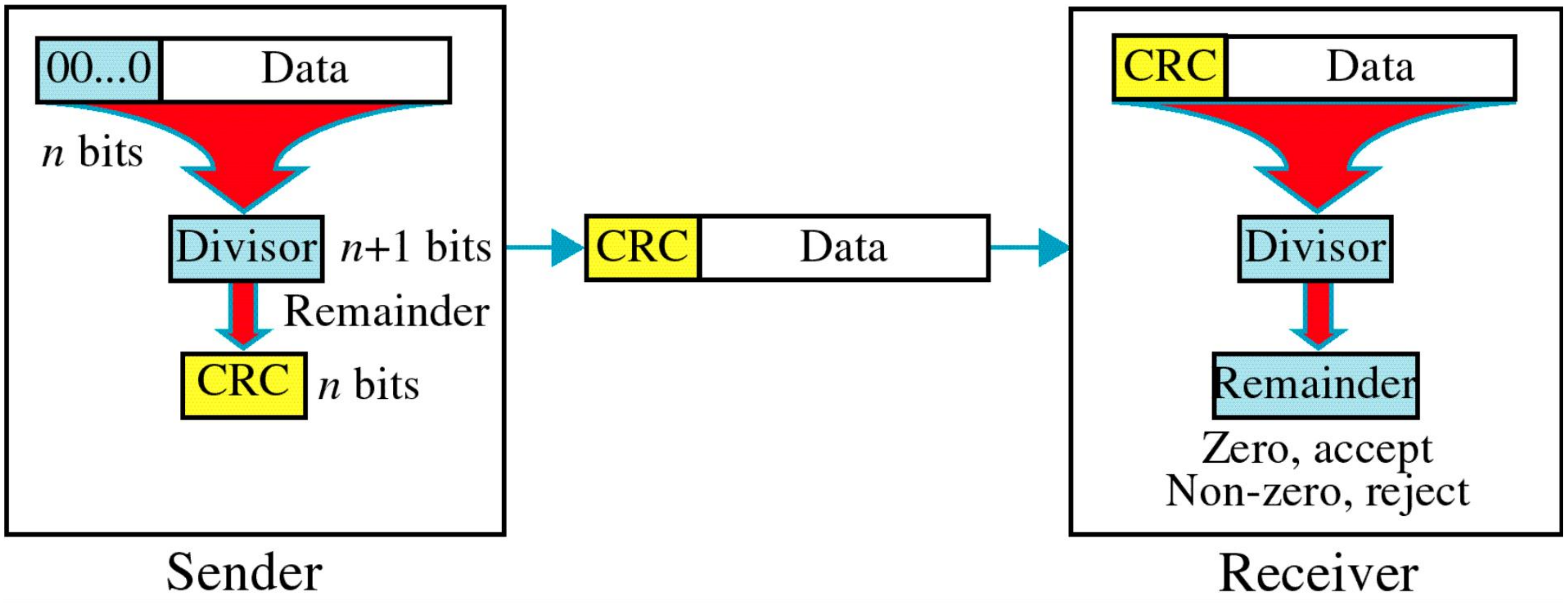
Following are the steps used in CRC for error detection:

- In CRC technique, a string of n 0s is appended to the data unit, and this n number is less than the number of bits in a predetermined number, known as division which is n+1 bits.
- Secondly, the newly extended data is divided by a divisor using a process is known as binary division. The remainder generated from this division is known as CRC remainder.
- Thirdly, the CRC remainder replaces the appended 0s at the end of the original data. This newly generated unit is sent to the receiver.
- The receiver receives the data followed by the CRC remainder. The receiver will treat this whole unit as a single unit, and it is divided by the same divisor that was used to find the CRC remainder.

If the resultant of this division is zero which means that it has no error, and the data is accepted.

If the resultant of this division is not zero which means that the data consists of an error. Therefore, the data is discarded.

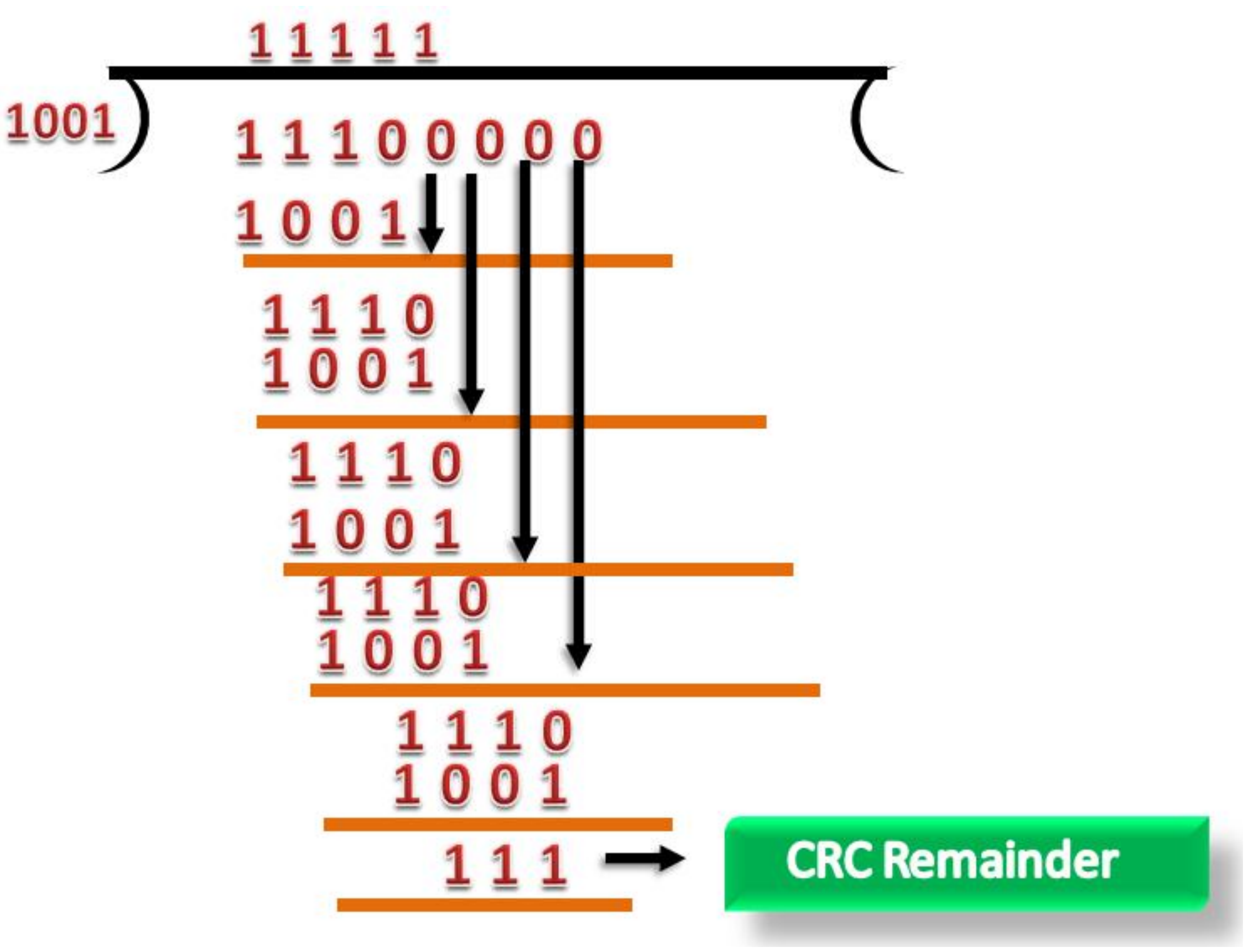
Cyclic Redundancy Check
CRC



Suppose the original data is 11100 and divisor is 1001.

CRC Generator

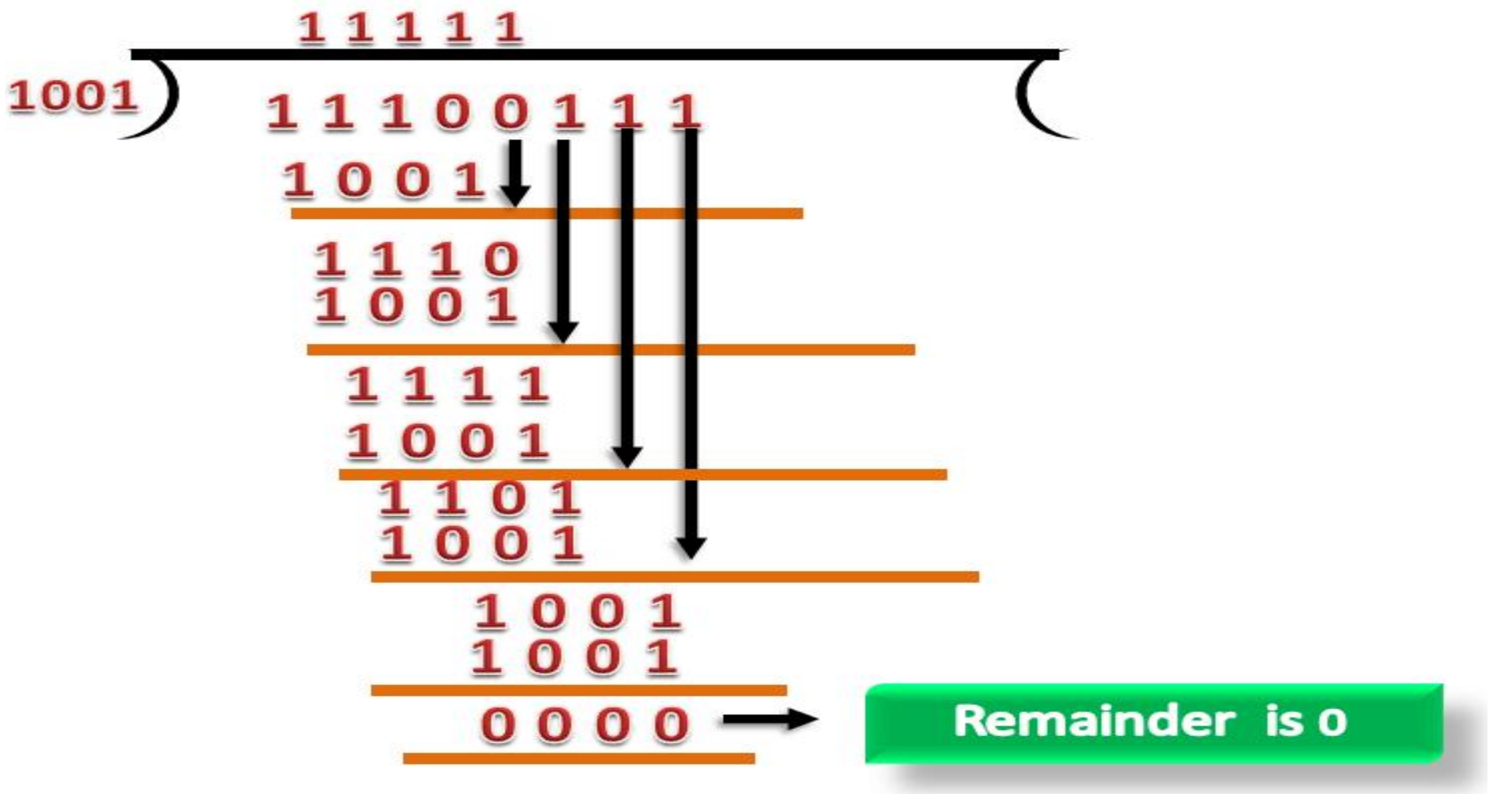
- A CRC generator uses a modulo-2 division. Firstly, three zeroes are appended at the end of the data as the length of the divisor is 4 and we know that the length of the string 0s to be appended is always one less than the length of the divisor.
- Now, the string becomes 11100000, and the resultant string is divided by the divisor 1001.
- The remainder generated from the binary division is known as CRC remainder. The generated value of the CRC remainder is 111.
- CRC remainder replaces the appended string of 0s at the end of the data unit, and the final string would be 11100111 which is sent across the network.



11111
1001
11100000
1001
1110
1001
1110
1001
1110
1001
1110
1001
111
CRC Remainder

CRC Checker

- The functionality of the CRC checker is similar to the CRC generator.
- When the string 11100111 is received at the receiving end, then CRC checker performs the modulo-2 division.
- A string is divided by the same divisor, i.e., 1001.
- In this case, CRC checker generates the remainder of zero. Therefore, the data is accepted.



Checksum

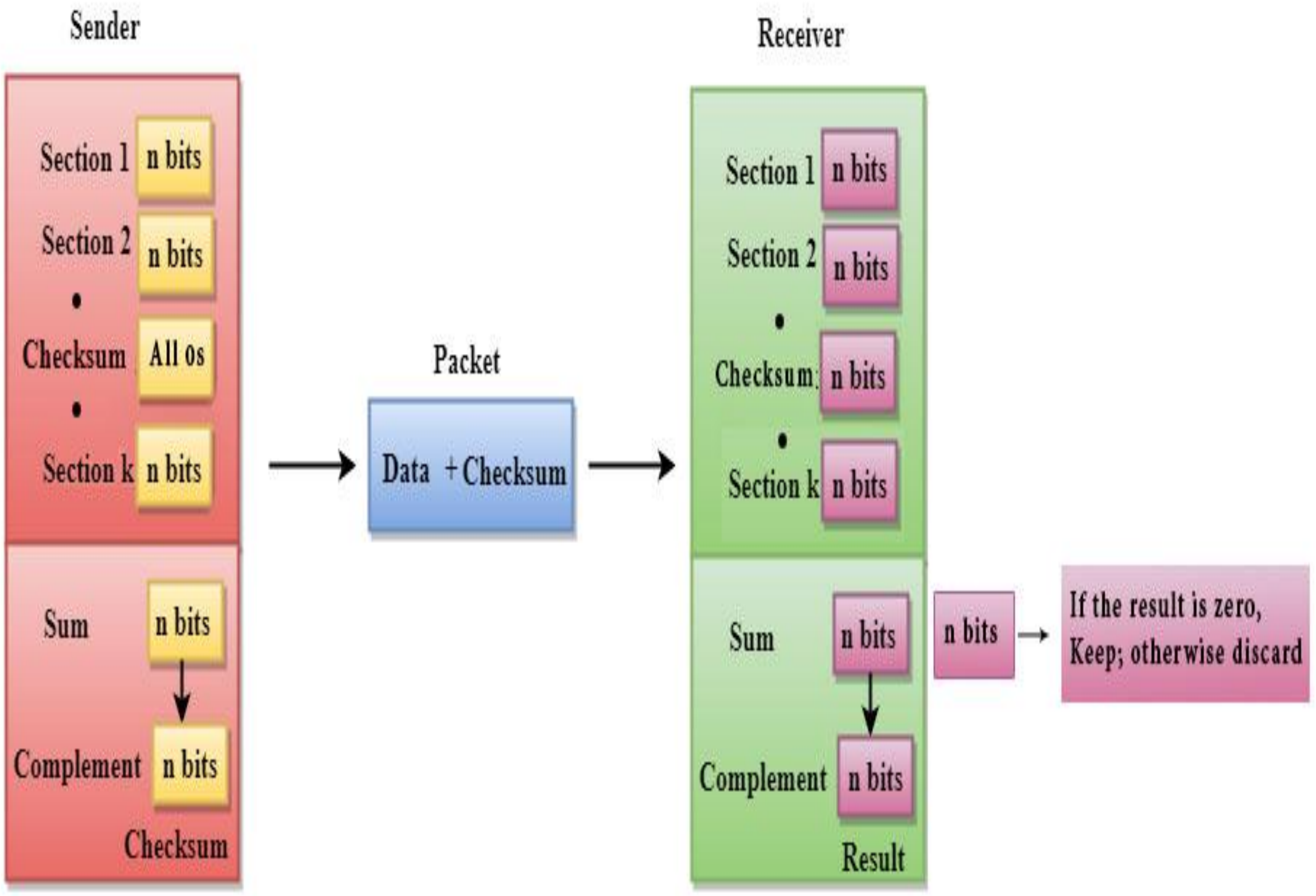
A Checksum is an error detection technique based on the concept of redundancy.

It is divided into two parts:

Checksum Generator

A Checksum is generated at the sending side. Checksum generator subdivides the data into equal segments of n bits each, and all these segments are added together by using one's complement arithmetic. The sum is complemented and appended to the original data, known as checksum field. The extended data is transmitted across the network.

Suppose L is the total sum of the data segments, then the checksum would be ?L


Sender
Section 1
n bits
Section 2
n bits
Checksum
All 0s
Section k
n bits
Sum
n bits
Complement
n bits
Checksum
Packet
Data + Checksum
Receiver
Section 1
n bits
Section 2
n bits
Checksum
n bits
Section k
n bits
Sum
n bits
Complement
n bits
Result
n bits
If the result is zero,
Keep; otherwise discard

At the sender

- The unit is divided into *k* sections, each of *n* bits.
- All sections are added together using one's complement to get the sum.
- The sum is complemented and becomes the checksum.
- The checksum is sent with the data

At the receiver

- The unit is divided into *k* sections, each of *n* bits.
- All sections are added together using one’s complement to get the sum.
- The sum is complemented.
- If the result is zero, the data are accepted: otherwise, they are rejected.

Performance

- The checksum detects all errors involving an odd number of bits.
- It detects most errors involving an even number of bits.
- If one or more bits of a segment are damaged and the corresponding bit or bits of opposite value in a second segment are also damaged, the sums of those columns will not change and the receiver will not detect a problem.

Error Correction

Error Correction codes are used to detect and correct the errors when data is transmitted from the sender to the receiver.

Error Correction can be handled in two ways:

- **Backward error correction:** Once the error is discovered, the receiver requests the sender to retransmit the entire data unit.

- **Forward error correction:** In this case, the receiver uses the error-correcting code which automatically corrects the errors.

A single additional bit can detect the error, but cannot correct it.

Error correction

For correcting the errors, one has to know the exact position of the error. For example, If we want to calculate a single-bit error, the error correction code will determine which one of seven bits is in error. To achieve this, we have to add some additional redundant bits.

Suppose r is the number of redundant bits and d is the total number of the data bits. The number of redundant bits r can be calculated by using the formula:

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

The value of r is calculated by using the above formula. For example, if the value of d is 4, then the possible smallest value that satisfies the above relation would be 3.

To determine the position of the bit which is in error, a technique developed by R.W Hamming is Hamming code which can be applied to any length of the data unit and uses the relationship between data units and redundant units.

Hamming Code

Parity bits: The bit which is appended to the original data of binary bits so that the total number of 1s is even or odd.
Even parity: To check for even parity, if the total number of 1s is even, then the value of the parity bit is 0. If the total number of 1s occurrences is odd, then the value of the parity bit is 1.
Odd Parity: To check for odd parity, if the total number of 1s is even, then the value of parity bit is 1. If the total number of 1s is odd, then the value of parity bit is 0.

Algorithm of Hamming code:

An information of 'd' bits are added to the redundant bits 'r' to form d+r.
The location of each of the (d+r) digits is assigned a decimal value.
The 'r' bits are placed in the positions 1,2,.....2^{k-1}.
At the receiving end, the parity bits are recalculated. The decimal value of the parity bits determines the position of an error.

Ex.1. Suppose the original data is 1010 which is to be sent.

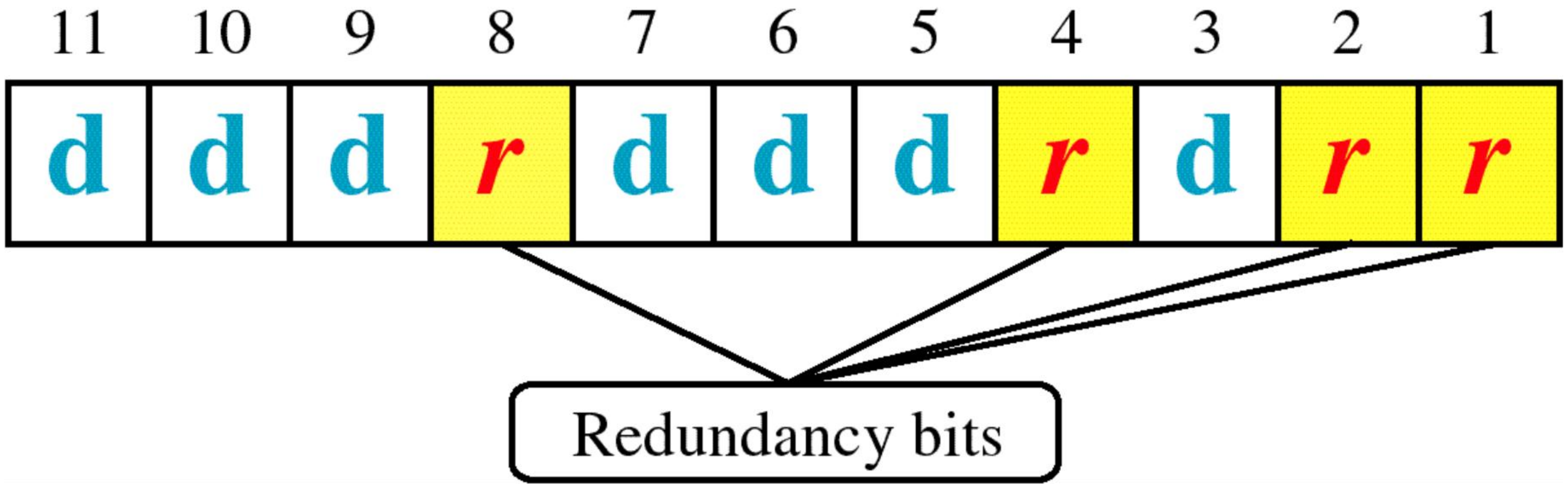
Total number of data bits 'd' = 4
Number of redundant bits r :
$2^r >= d+r+1$
$2^r >= 4+r+1$
Therefore, the value of r is 3 that satisfies the above relation. **To total number of bits = d+r = 4+3 = 7;**

Ex.2. Suppose the original data is 1011010 which is to be sent.

Total number of data bits 'd' = 7
Number of redundant bits r :
$2^r >= d+r+1$
$2^r >= 7+r+1$
Therefore, the value of r is 4 that satisfies the above relation. **To total number of bits = d+r = 7+4= 11;**

Determining the position of the redundant bits

The number of redundant bits is 4. The three bits are represented by r1, r2, r4. The position of the redundant bits is calculated with corresponds to the raised power of 2. Therefore, their corresponding positions are **1, 2^1, 2^2**, **2^3**



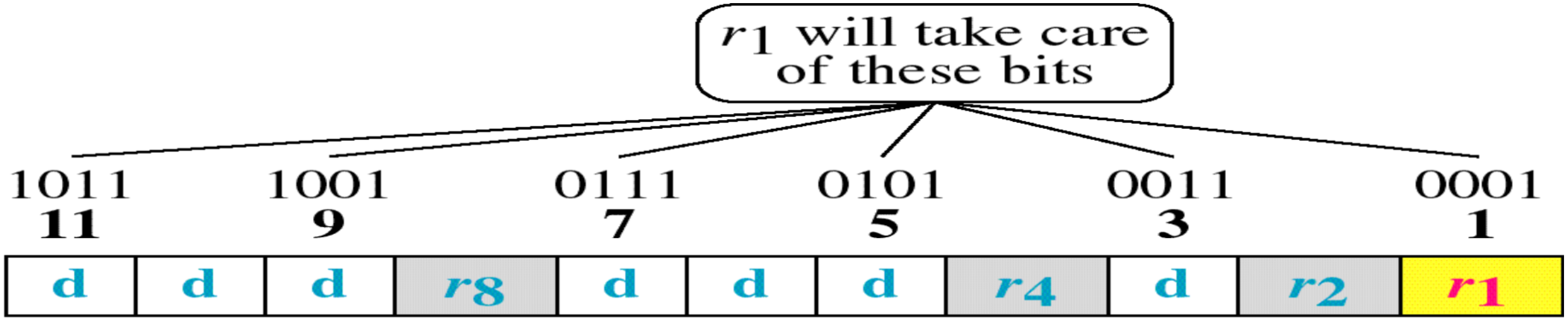
Determining the Parity bits

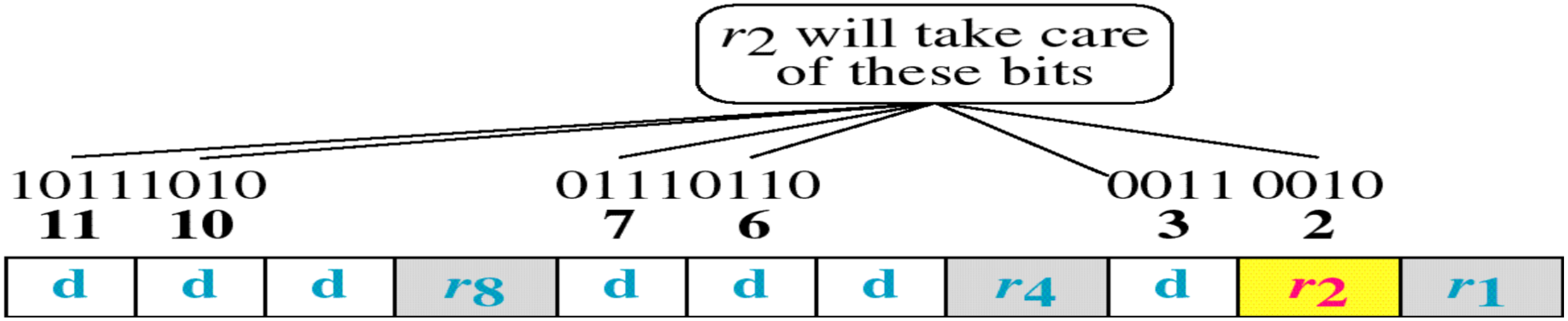
Determining the r1 bit

The r1 bit is calculated by performing a parity check on the bit positions whose binary representation includes 1 in the first position. i.e. r1,d3,d5,d7,d9,d11.

Determining r2 bit

The r2 bit is calculated by performing a parity check on the bit positions whose binary representation includes 1 in the second position. i.e. r2,d3,d6,d7,d10,d11.





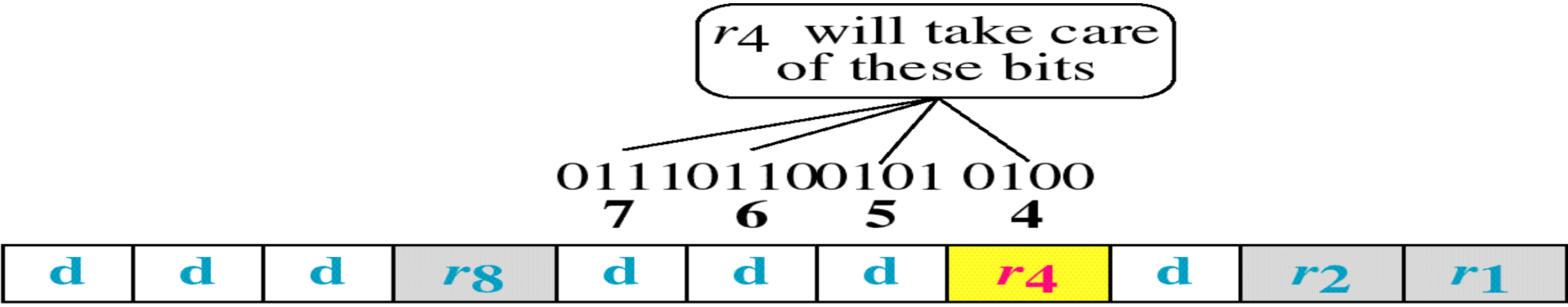
Determining the Parity bits

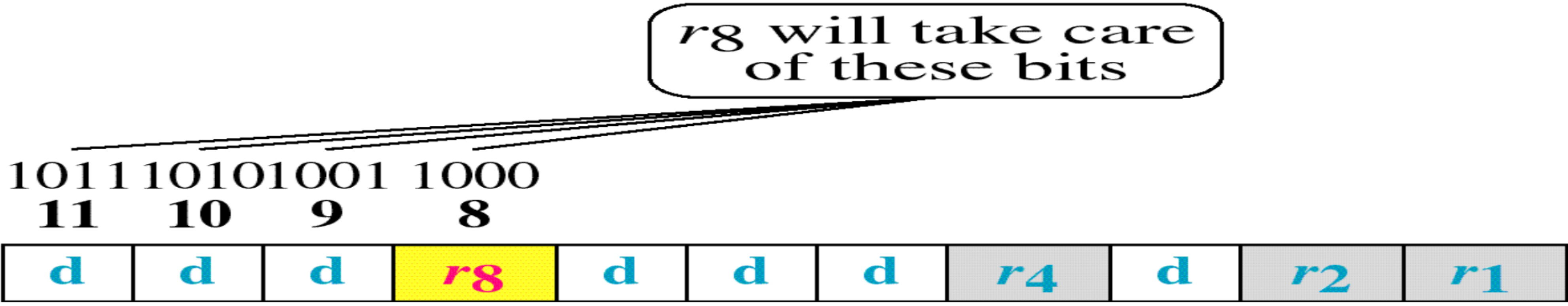
Determining r4 bit

The r4 bit is calculated by performing a parity check on the bit positions whose binary representation includes 1 in the third position. i.e. r4,d5,d6,d7.

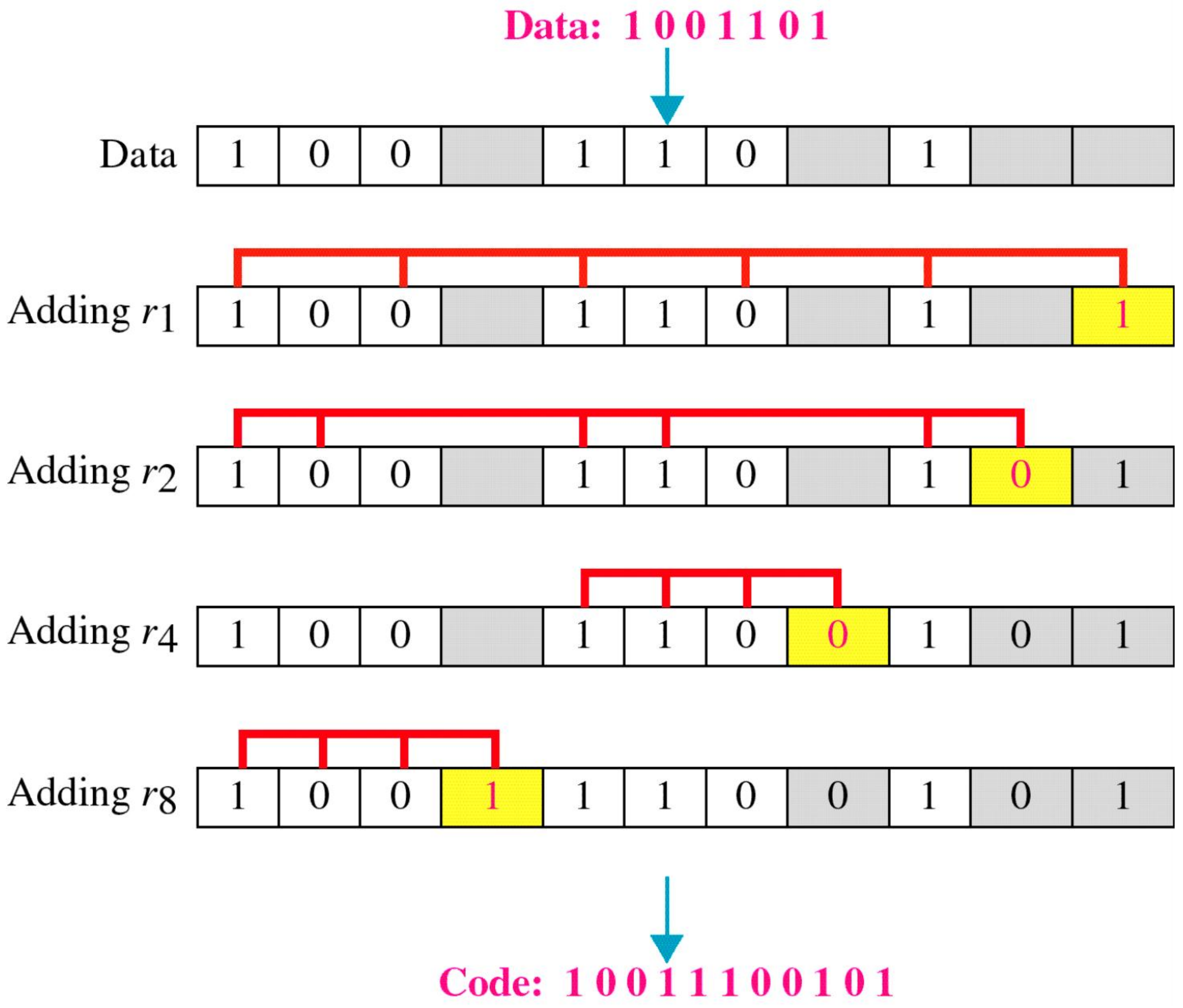
Determining r8 bit

The r8 bit is calculated by performing a parity check on the bit positions whose binary representation includes 1 in the fourth position. i.e. r8,d9,d10,d11.

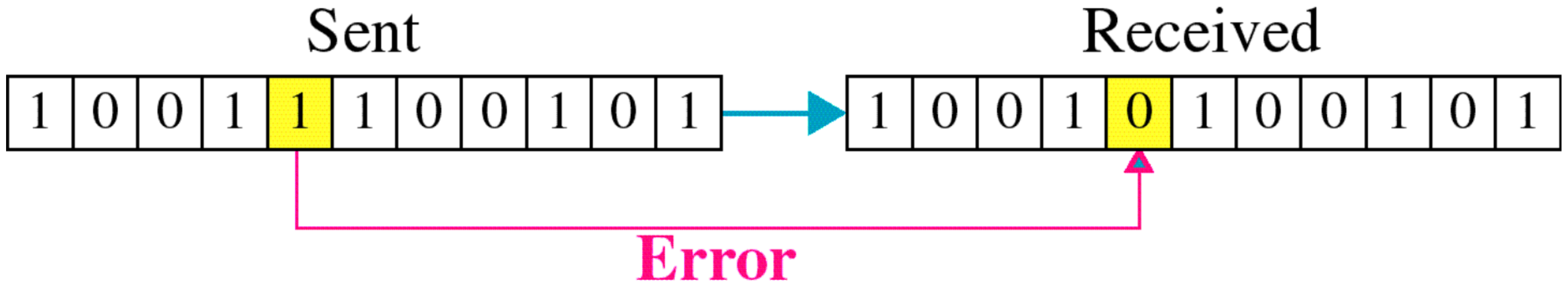




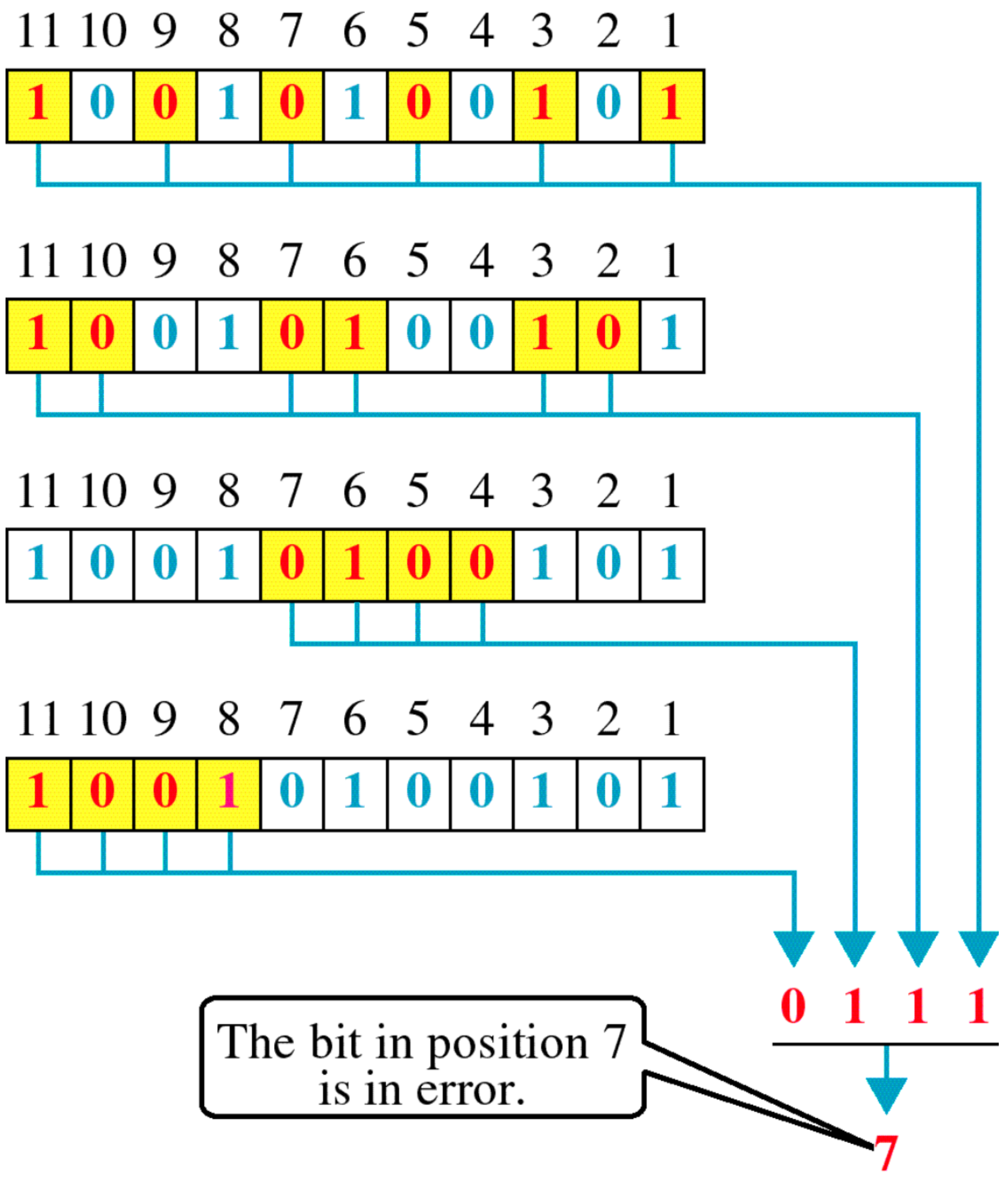
Example of Hamming Code


Data: 1 0 0 1 1 0 1
Data | 1 | 0 | 0 | | 1 | 1 | 0 | | 1 | |
Adding r_1 | 1 | 0 | 0 | | 1 | 1 | 0 | | 1 | | 1
Adding r_2 | 1 | 0 | 0 | | 1 | 1 | 0 | | 1 | 0 | 1
Adding r_4 | 1 | 0 | 0 | | 1 | 1 | 0 | 0 | 1 | 0 | 1
Adding r_8 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1
Code: 1 0 0 1 1 1 0 0 1 0 1

Single-bit error



Error Detection



11 10 9 8 7 6 5 4 3 2 1
1 0 0 1 0 1 0 0 1 0 1
11 10 9 8 7 6 5 4 3 2 1
1 0 0 1 0 1 0 0 1 0 1
11 10 9 8 7 6 5 4 3 2 1
1 0 0 1 0 1 0 0 1 0 1
11 10 9 8 7 6 5 4 3 2 1
1 0 0 1 0 1 0 0 1 0 1
0 1 1 1
7
The bit in position 7 is in error.