# CSE / T / 315A
# Data Communications
# Topic 7- Error Detection and Correction

Sarbani Roy
sarbani.roy@jadavpuruniversity.in
Office: CC-5-7
Cell: 9051639328

# Motivation

- Data can be corrupted during transmission.
- Some applications require that errors be detected and corrected.

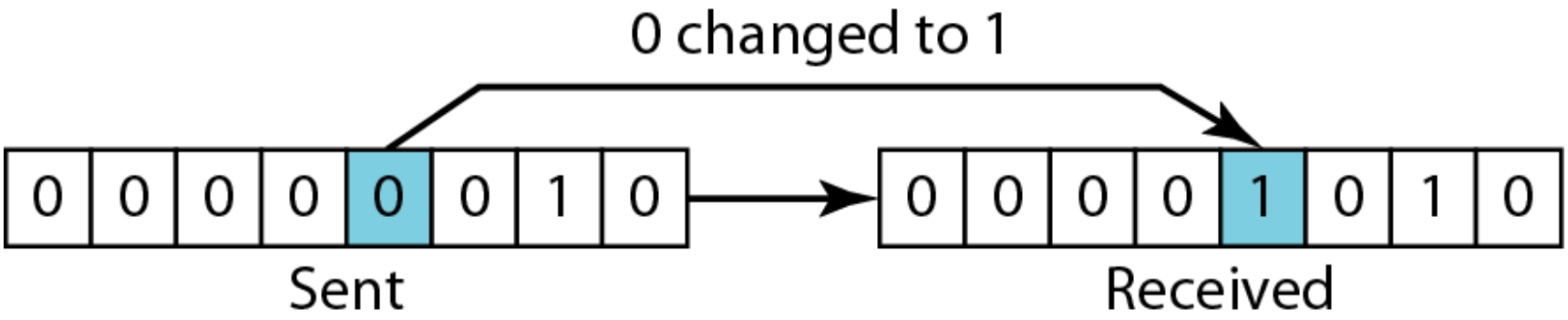
# Topics

Some issues related, directly or indirectly, to error detection and correction

- Types of Errors
- Redundancy
- Detection Versus Correction
- Popular techniques

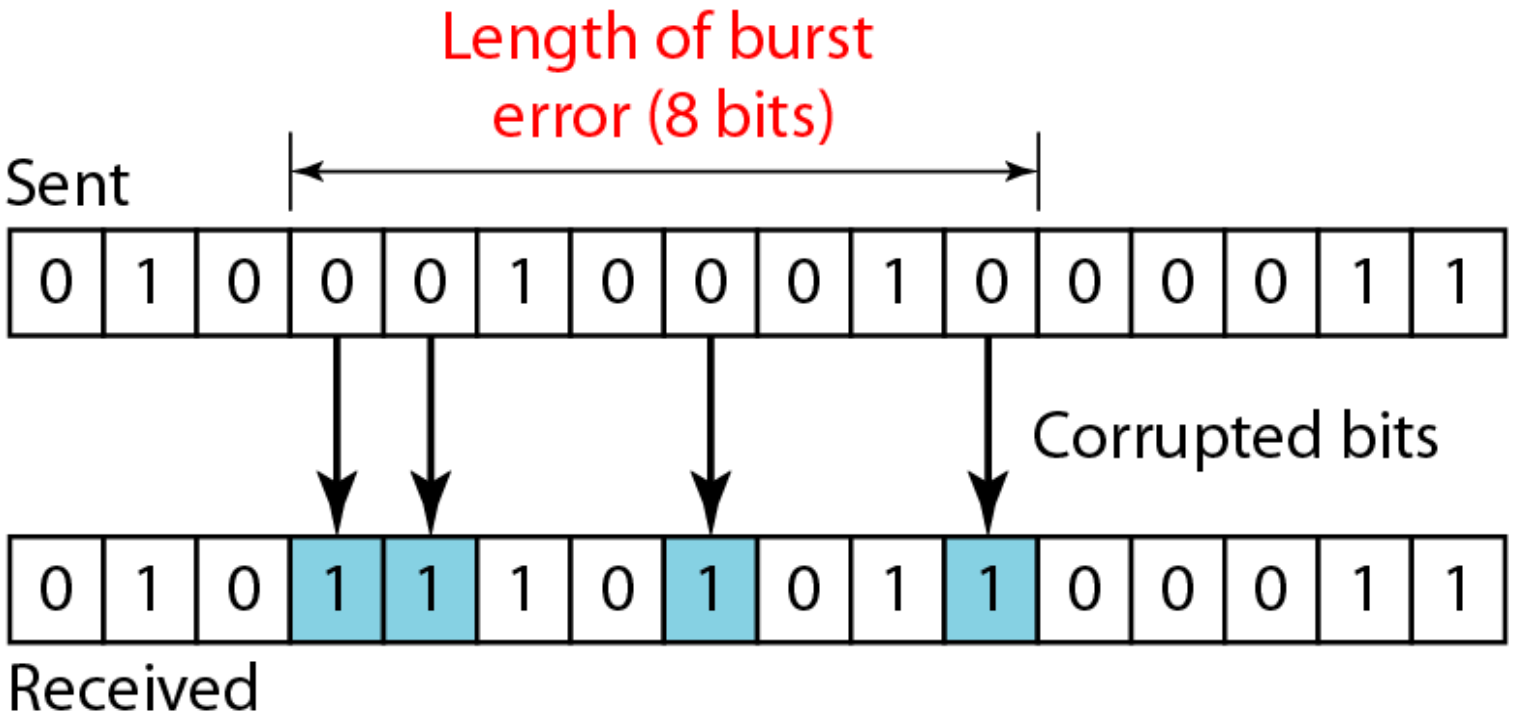
# Single bit error

- In a single-bit error, only 1 bit in the data unit has changed.



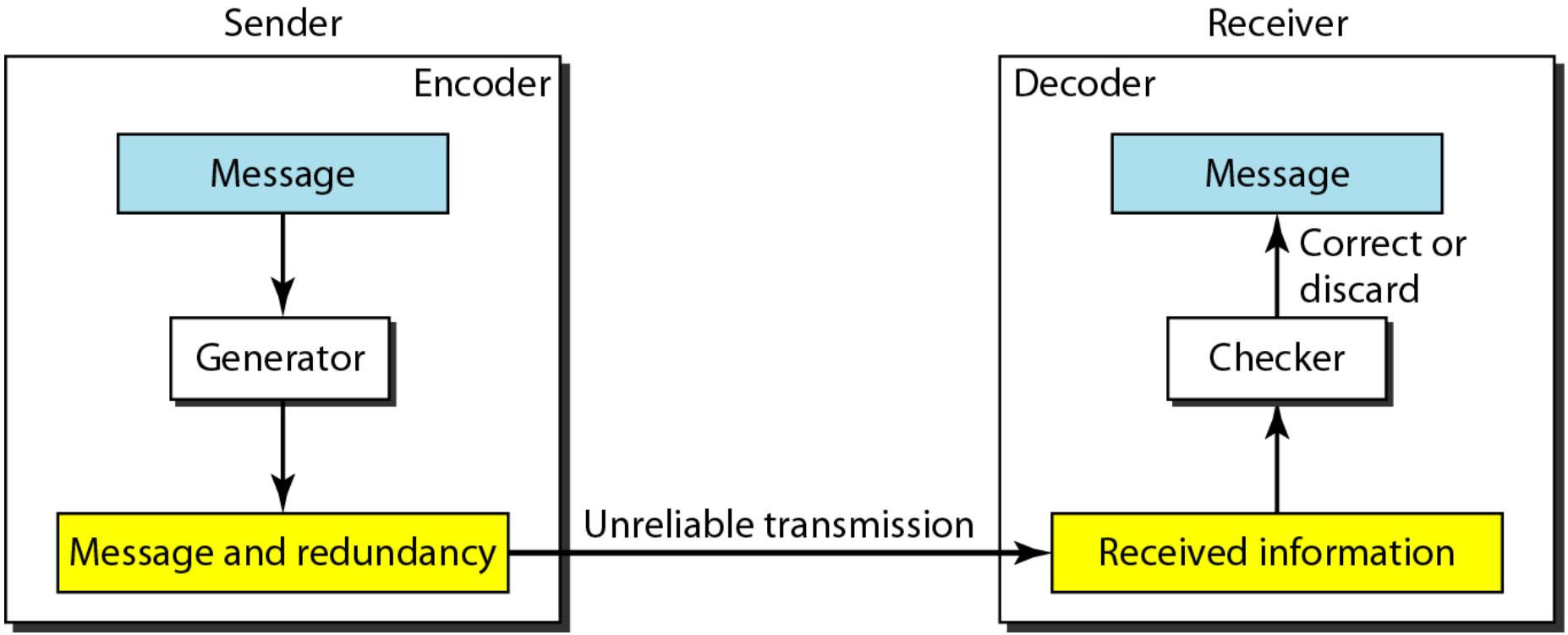
# Burst error

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



# A solution approach: redundancy

- To detect or correct errors, we need to send extra (redundant) bits with data.
- Figure below shows the structure of encoder and decoder.

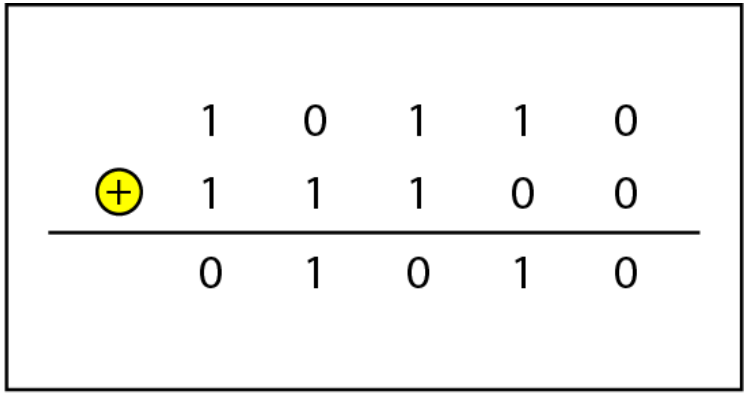


# XORing

- In modulo-N arithmetic, we use only the integers in the range 0 to N −1, inclusive.
- XORing of two single bits or two words


a. Two bits are the same, the result is 0.


b. Two bits are different, the result is 1.

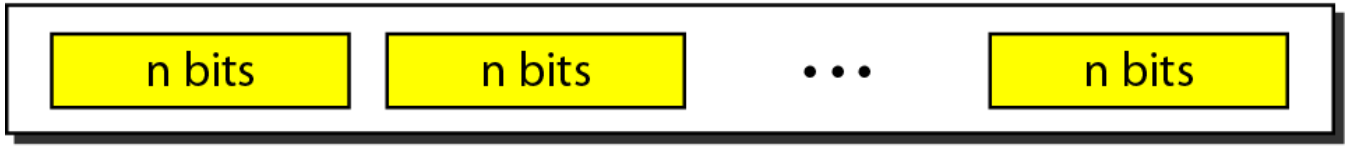



c. Result of XORing two patterns

# Block Coding

- In block coding, we divide our message into blocks, each of k bits, called datawords. We add r redundant bits to each block to make the length n = k + r. The resulting n-bit blocks are called codewords.
- *The 4B/5B block coding is a good example of this type of coding. In this coding scheme, k = 4 and n = 5. As we saw, we have $2^k = 16$ datawords and $2^n = 32$ codewords. We saw that 16 out of 32 codewords are used for message transfer and the rest are either used for other purposes or unused.*

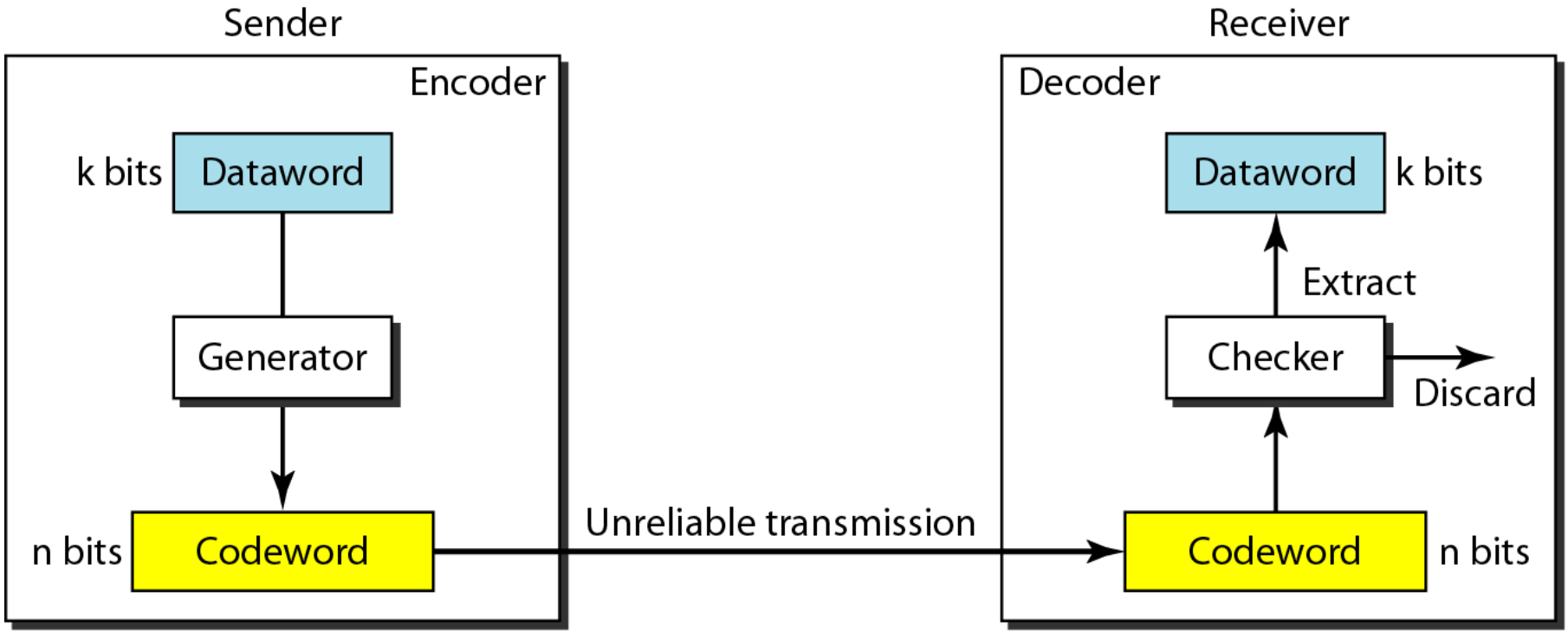

$2^k$ Datawords, each of k bits




$2^n$ Codewords, each of n bits (only $2^k$ of them are valid)

# Error Detection

- Enough redundancy is added to detect an error.
- The receiver knows an error occurred but does not know which bit(s) is(are) in error.
- Has less overhead than error correction.



# Example

- Let us assume that k = 2 and n = 3. Table below shows the list of datawords and codewords. Later, we will see how to derive a codeword from a dataword.

| *Datawords* | *Codewords* |
|---|---|
| 00 | 000 |
| 01 | 011 |
| 10 | 101 |
| 11 | 110 |

- Assume the sender encodes the dataword 01 as 011 and sends it to the receiver.

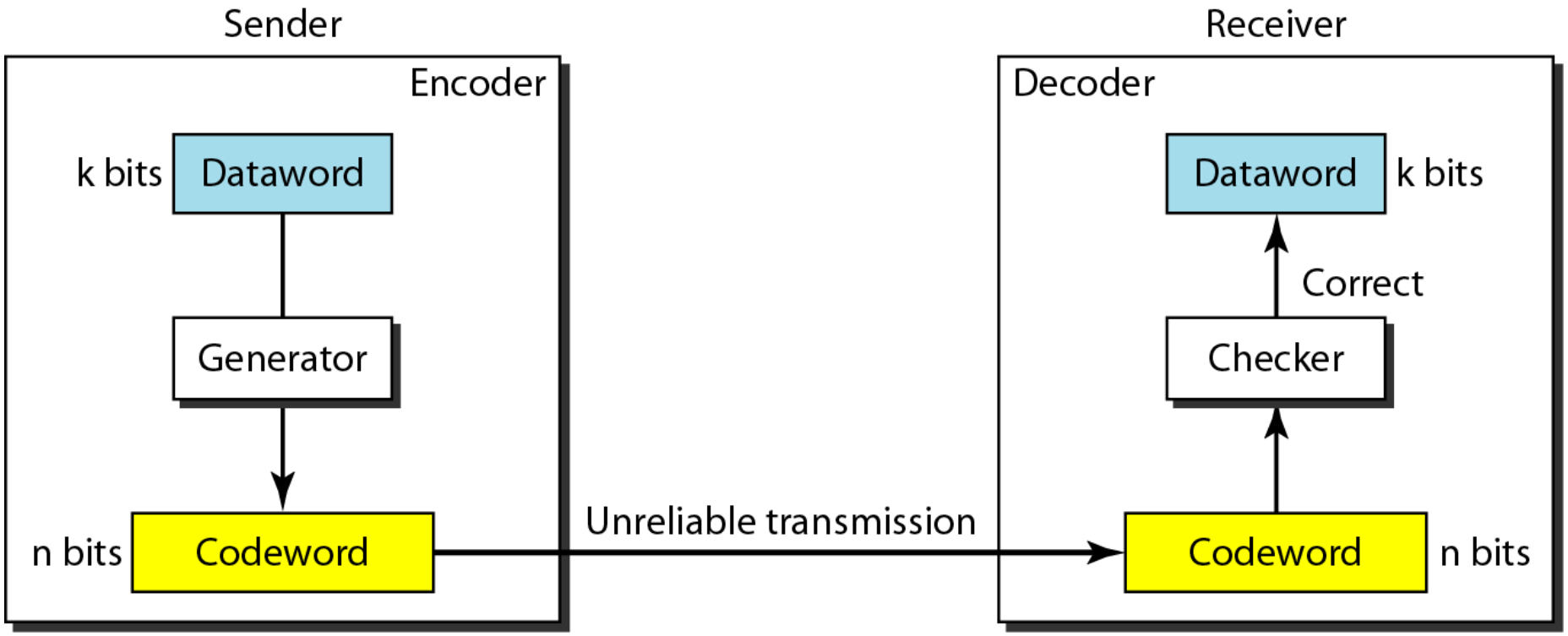
# Consider the following cases:

1. The receiver receives 011. It is a valid codeword. The receiver extracts the dataword 01 from it.
2. The codeword is corrupted during transmission, and 111 is received. This is not a valid codeword and is discarded.
3. The codeword is corrupted during transmission, and 000 is received. This is a valid codeword. The receiver incorrectly extracts the dataword 00. Two corrupted bits have made the error undetectable.

# Motivation for Error Correction

- An error-detecting code can detect only the types of errors for which it is designed; other types of errors may remain undetected.

# Structure of encoder and decoder in error correction



# Redundant bits

- Let us add more redundant bits to previous example to see if the receiver can correct an error without knowing what was actually sent.
- We add 3 redundant bits to the 2-bit dataword to make 5-bit codewords.
- Assume the dataword is 01. The sender creates the codeword 01011. The codeword is corrupted during transmission, and 01001 is received. First, the receiver finds that the received codeword is not in the table. This means an error has occurred. The receiver, assuming that there is only 1 bit corrupted, uses the following strategy to guess the correct dataword.

| *Datawords* | *Codewords* |
|---|---|
| 00 | 000 |
| 01 | 011 |
| 10 | 101 |
| 11 | 110 |

| *Dataword* | *Codeword* |
|---|---|
| 00 | 00000 |
| 01 | 01011 |
| 10 | 10101 |
| 11 | 11110 |

# Error Correction (Simple approach!)

1. Comparing the received codeword with the first codeword in the table (01001 versus 00000), the receiver decides that the first codeword is not the one that was sent because there are two different bits.
2. By the same reasoning, the original codeword cannot be the third or fourth one in the table.
3. The original codeword must be the second one in the table because this is the only one that differs from the received codeword by 1 bit. The receiver replaces 01001 with 01011 and consults the table to find the dataword 01.

| *Dataword* | *Codeword* |
|---|---|
| 00 | 00000 |
| 01 | 01011 |
| 10 | 10101 |
| 11 | 11110 |

# Hamming Distance

- The Hamming distance between two words is the number of differences between corresponding bits.
- Let us find the Hamming distance between two pairs of words.
  - The Hamming distance d(000, 011) is 2 because

$$000 \oplus 011 \text{ is } 011 \text{ (two 1s)}$$

  - The Hamming distance d(10101, 11110) is 3 because

$$10101 \oplus 11110 \text{ is } 01011 \text{ (three 1s)}$$

# Minimum Hamming Distance

- The minimum Hamming distance is the smallest Hamming distance ($d_{min}$) between all possible pairs in a set of words.

| *Datawords* | *Codewords* |
|---|---|
| 00 | 000 |
| 01 | 011 |
| 10 | 101 |
| 11 | 110 |

$d(000, 011) = 2$ $d(000, 101) = 2$ $d(000, 110) = 2$ $d(011, 101) = 2$
$d(011, 110) = 2$ $d(101, 110) = 2$

***The $d_{min}$ in this case is 2***

| *Dataword* | *Codeword* |
|---|---|
| 00 | 00000 |
| 01 | 01011 |
| 10 | 10101 |
| 11 | 11110 |

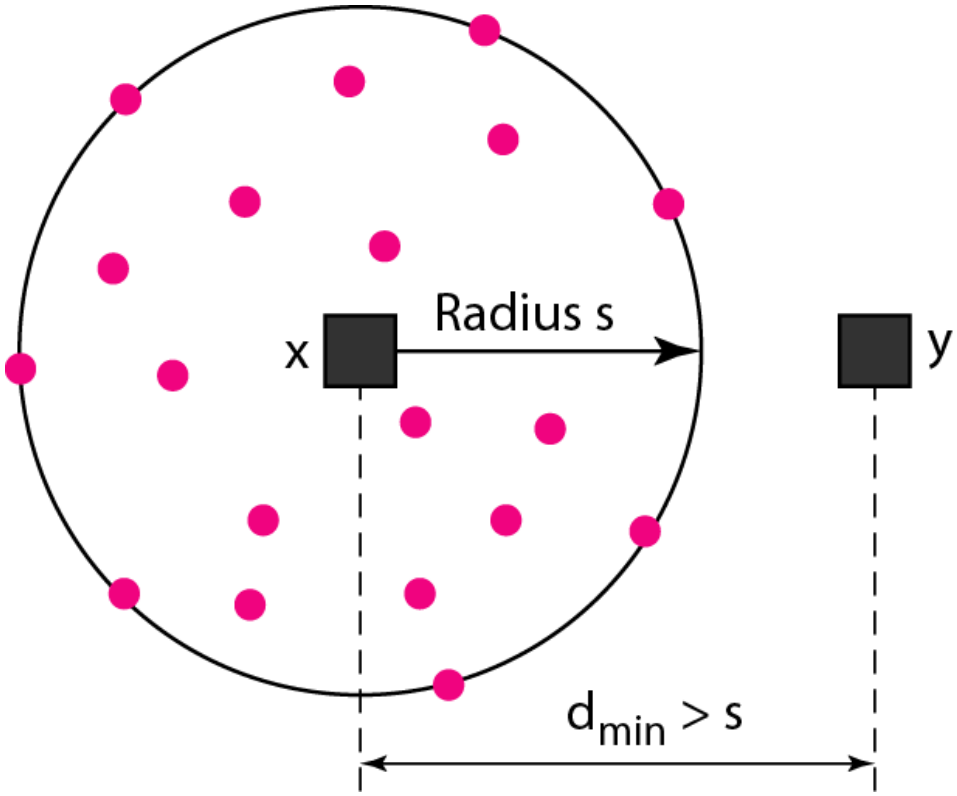
$d(00000, 01011) = 3$ $d(00000, 10101) = 3$ $d(00000, 11110) = 4$
$d(01011, 10101) = 4$ $d(01011, 11110) = 3$ $d(10101, 11110) = 3$

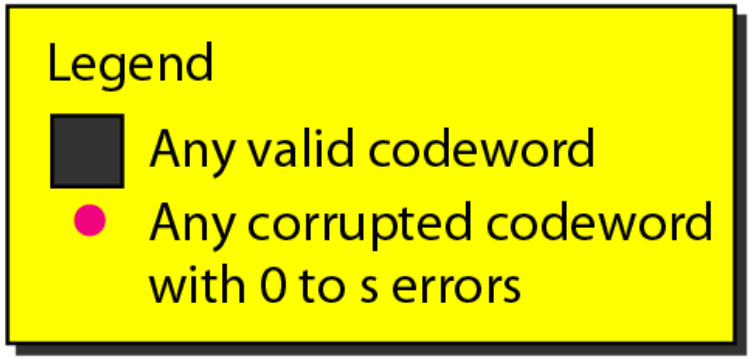
***The $d_{min}$ in this case is 3***

# Hamming distance in a block code

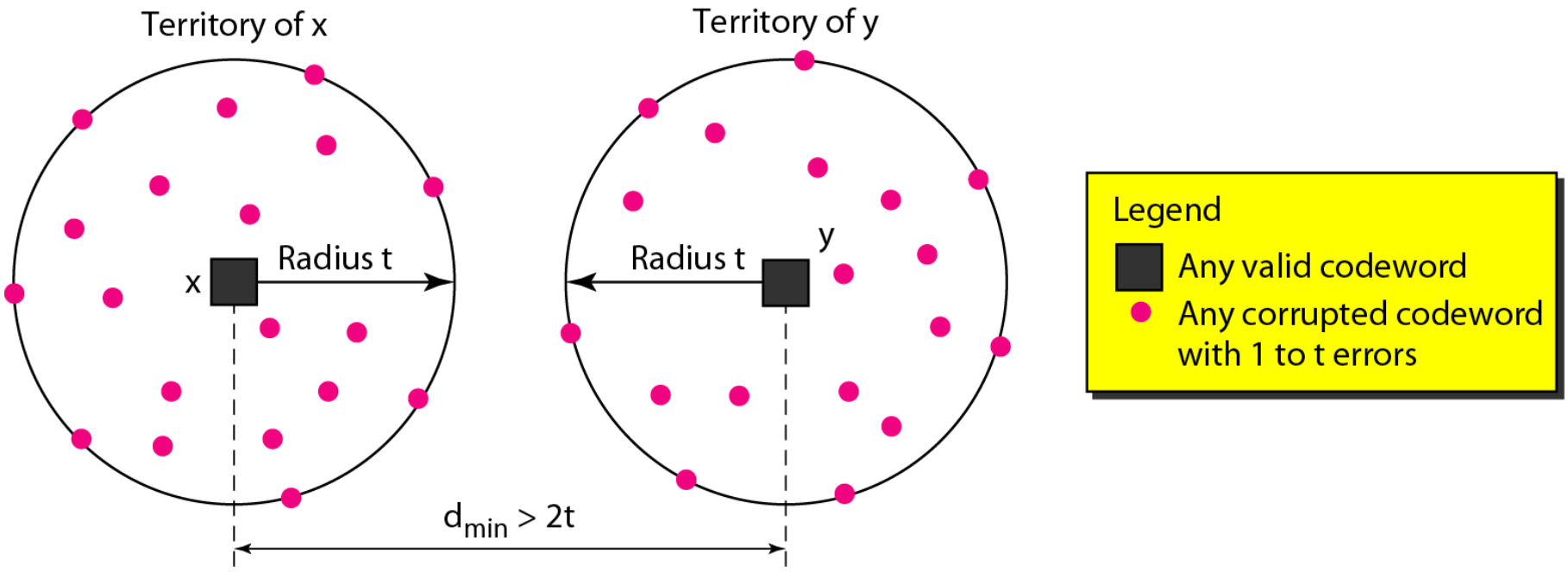
- To guarantee the detection of up to s errors in all cases, the minimum Hamming distance in a block code must be dmin = s + 1.
- The minimum Hamming distance for our first code scheme is 2. This code guarantees detection of only a single error. For example, if the third codeword (101) is sent and one error occurs, the received codeword does not match any valid codeword. If two errors occur, however, the received codeword may match a valid codeword and the errors are not detected.
- Our second block code scheme has dmin = 3. This code can detect up to two errors. Again, we see that when any of the valid codewords is sent, two errors create a codeword which is not in the table of valid codewords. The receiver cannot be fooled.
- However, some combinations of three errors change a valid codeword to another valid codeword. The receiver accepts the received codeword and the errors are undetected.

# Geometric concept for finding $d_{min}$ in error detection



# Geometric concept for finding $d_{min}$ in error correction



# How to guarantee correction of up to *t* errors

- To guarantee correction of up to *t* errors in all cases, the minimum Hamming distance in a block code must be

$$d_{min} = 2t + 1$$

- A code scheme has a Hamming distance $d_{min} = 4$. What is the error detection and correction capability of this scheme?
- This code guarantees the detection of up to three errors (s = 3), but it can correct up to one error. In other words, if this code is used for error correction, part of its capability is wasted. Error correction codes need to have an odd minimum distance (3, 5, 7, . . . ).

# Linear Block Codes

- Almost all block codes used today belong to a subset called linear block codes.
- A linear block code is a code in which the exclusive OR (addition modulo-2) of two valid codewords creates another valid codeword.

# Table 1 and Table 2

| *Datawords* | *Codewords* |
|---|---|
| 00 | 000 |
| 01 | 011 |
| 10 | 101 |
| 11 | 110 |

| *Dataword* | *Codeword* |
|---|---|
| 00 | 00000 |
| 01 | 01011 |
| 10 | 10101 |
| 11 | 11110 |

Let us see if the two codes we defined in Table 1 and Table 2 belong to the class of linear block codes.

1. The scheme in Table 1 is a linear block code because the result of XORing any codeword with any other codeword is a valid codeword. For example, the XORing of the second and third codewords creates the fourth one. (***minimum Hamming distance is $d_{min} = 2$***)

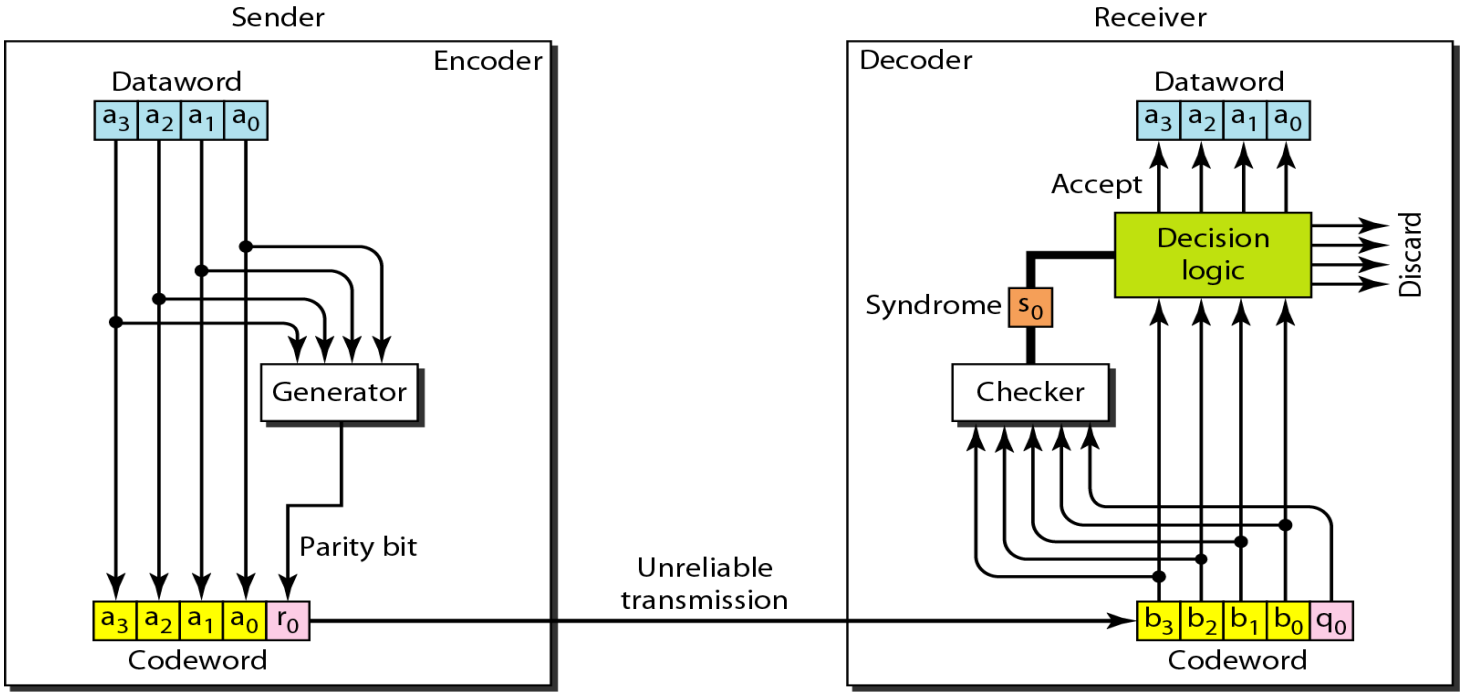
2. The scheme in Table 2 is also a linear block code. We can create all four codewords by XORing two other codewords. (***minimum Hamming distance is $d_{min} = 3$***)

# Simple parity-check

- A simple parity-check code is a single-bit error-detecting code in which $n = k + 1$ with $d_{min} = 2$.
- Even parity (ensures that a codeword has an even number of 1's) and odd parity (ensures that there are an odd number of 1's in the codeword)
- *Simple parity-check code C(5, 4) (i.e., 4 bit dataword+1 redundant bit= 5 bit codeword)*

| *Datawords* | *Codewords* | *Datawords* | *Codewords* |
|---|---|---|---|
| 0000 | 00000 | 1000 | 10001 |
| 0001 | 00011 | 1001 | 10010 |
| 0010 | 00101 | 1010 | 10100 |
| 0011 | 00110 | 1011 | 10111 |
| 0100 | 01001 | 1100 | 11000 |
| 0101 | 01010 | 1101 | 11011 |
| 0110 | 01100 | 1110 | 11101 |
| 0111 | 01111 | 1111 | 11110 |

Encoder and decoder for simple parity-check code




- The syndrome is the result of a parity check on the received codeword to determine whether it is a valid member of a codeword set.
- If it is a member the syndrome is 0 and if it is not the syndrome is 1 [for even parity]
- The syndrome is used to make the decision whether to discard the code or not

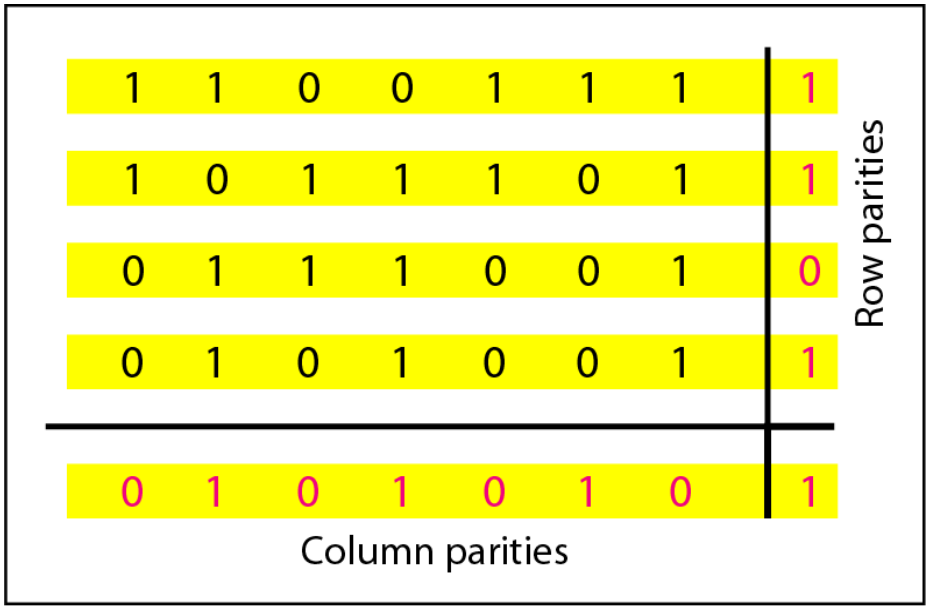
# Let us look at some transmission scenarios

Assume the sender sends the dataword 1011. The codeword created from this dataword is 10111, which is sent to the receiver. We examine five cases:

1. No error occurs; the received codeword is 10111. The syndrome is 0. The dataword 1011 is created.
2. One single-bit error changes $a_1$. The received codeword is 10011. The syndrome is 1. No dataword is created.
3. One single-bit error changes $r_0$ . The received codeword is 10110. The syndrome is 1. No dataword is created.
4. An error changes r0 and a second error changes a3 . The received codeword is 00110. The syndrome is 0. The dataword 0011 is created at the receiver. Note that here the dataword is wrongly created due to the syndrome value.
5. Three bits—a3, a2, and a1—are changed by errors. The received codeword is 01011. The syndrome is 1. The dataword is not created. This shows that the simple parity check, guaranteed to detect one single error, can also find any odd number of errors.

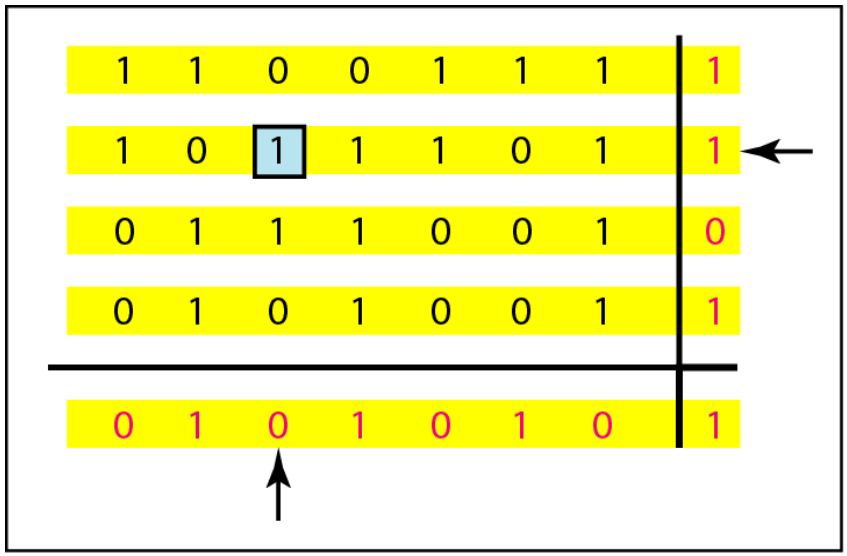
- A simple parity-check code can detect an odd number of errors.
- All Hamming codes discussed have $d_{min} = 3$ (2 bit error detection and single bit error correction).
- Hamming codes are a particular type of block codes
- A codeword consists of n bits of which k are data bits and r are check bits i.e., $n = 2^r - 1$ and $k = n-r$

# Two-dimensional parity-check code

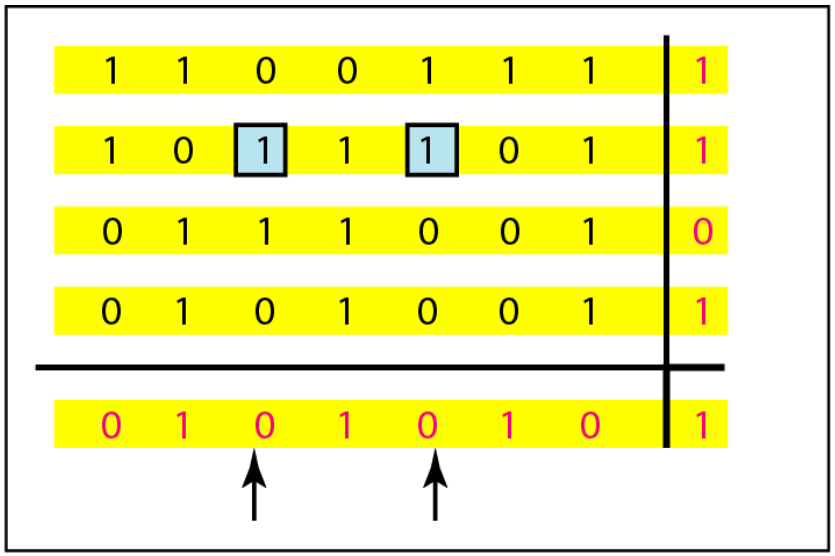



a. Design of row and column parities

# Two-dimensional parity-check code




b. One error affects two parities




c. Two errors affect two parities

# Hamming code C(7, 4) - n=7, k = 4

| *Datawords* | *Codewords* | *Datawords* | *Codewords* |
|---|---|---|---|
| 0000 | 0000000 | 1000 | 1000110 |
| 0001 | 0001101 | 1001 | 1001011 |
| 0010 | 0010111 | 1010 | 1010001 |
| 0011 | 0011010 | 1011 | 1011100 |
| 0100 | 0100011 | 1100 | 1100101 |
| 0101 | 0101110 | 1101 | 1101000 |
| 0110 | 0110100 | 1110 | 1110010 |
| 0111 | 0111001 | 1111 | 1111111 |

Calculating the parity bits at the transmitter :

Modulo 2 arithmetic:

$r_0 = a_2 + a_1 + a_0$

$r_1 = a_3 + a_2 + a_1$

$r_2 = a_1 + a_0 + a_3$

Calculating the syndrome at the receiver:

$s_0 = b_2 + b_1 + b_0 + q_0$

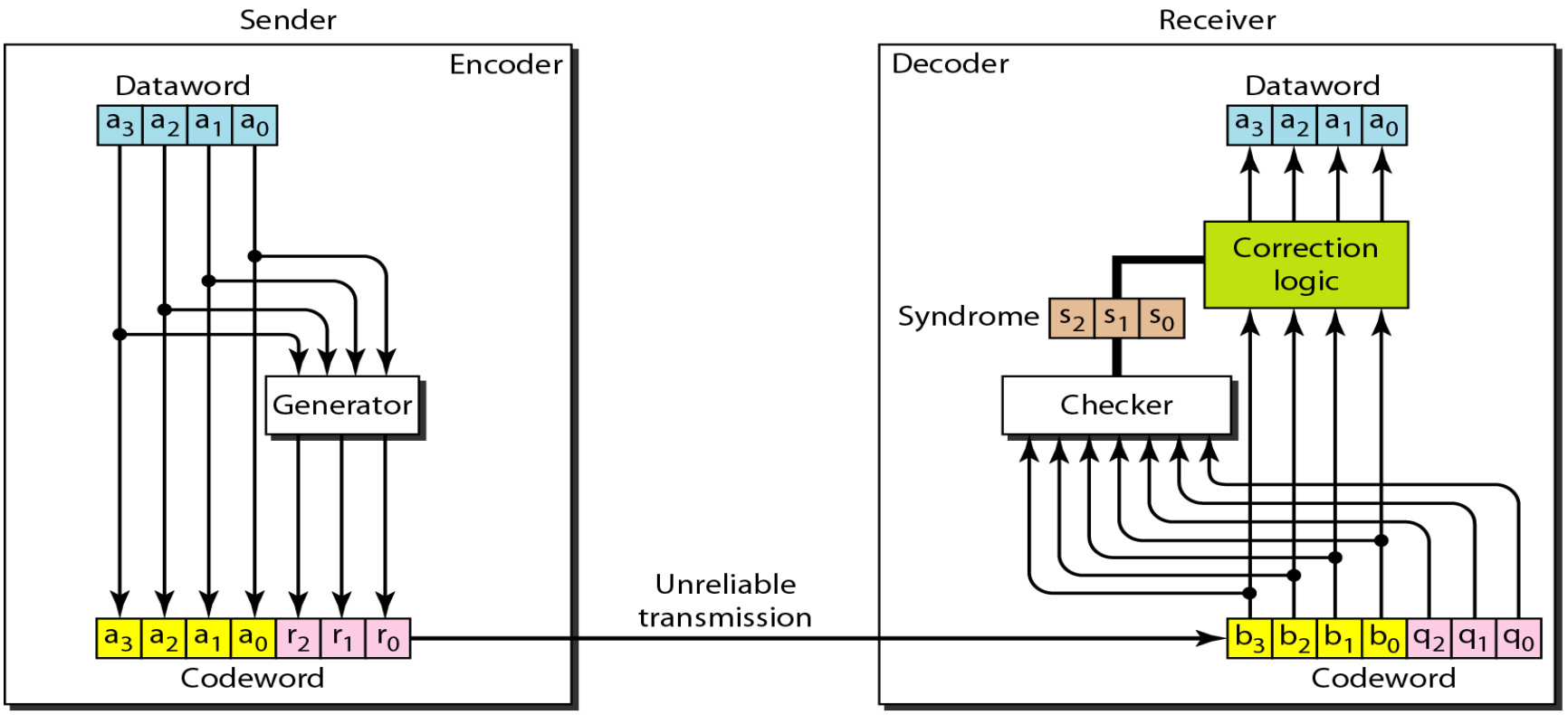
$s_1 = b_3 + b_2 + b_1 + q_1$

$s_2 = b_1 + b_0 + b_3 + q_2$

***Logical decision made by the correction logic analyzer***

| *Syndrome* | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| *Error* | None | $q_0$ | $q_1$ | $b_2$ | $q_2$ | $b_0$ | $b_3$ | $b_1$ |

# The structure of the encoder and decoder for a Hamming code



Let us trace the path of three datawords from the sender to the destination:

1. The dataword 0100 becomes the codeword 0100011. The codeword 0100011 is received. The syndrome is 000, the final dataword is 0100.
2. The dataword 0111 becomes the codeword 0111001. The received codeword is: 0011001. The syndrome is 011. After flipping $b_2$ (changing the 1 to 0), the final dataword is 0111.
3. The dataword 1101 becomes the codeword 1101000. The codeword 0001000 is received (2 errors). The syndrome is 101. After flipping $b_0$, we get 0000, the wrong dataword. This shows that our code cannot correct two errors.

# C(15,11)

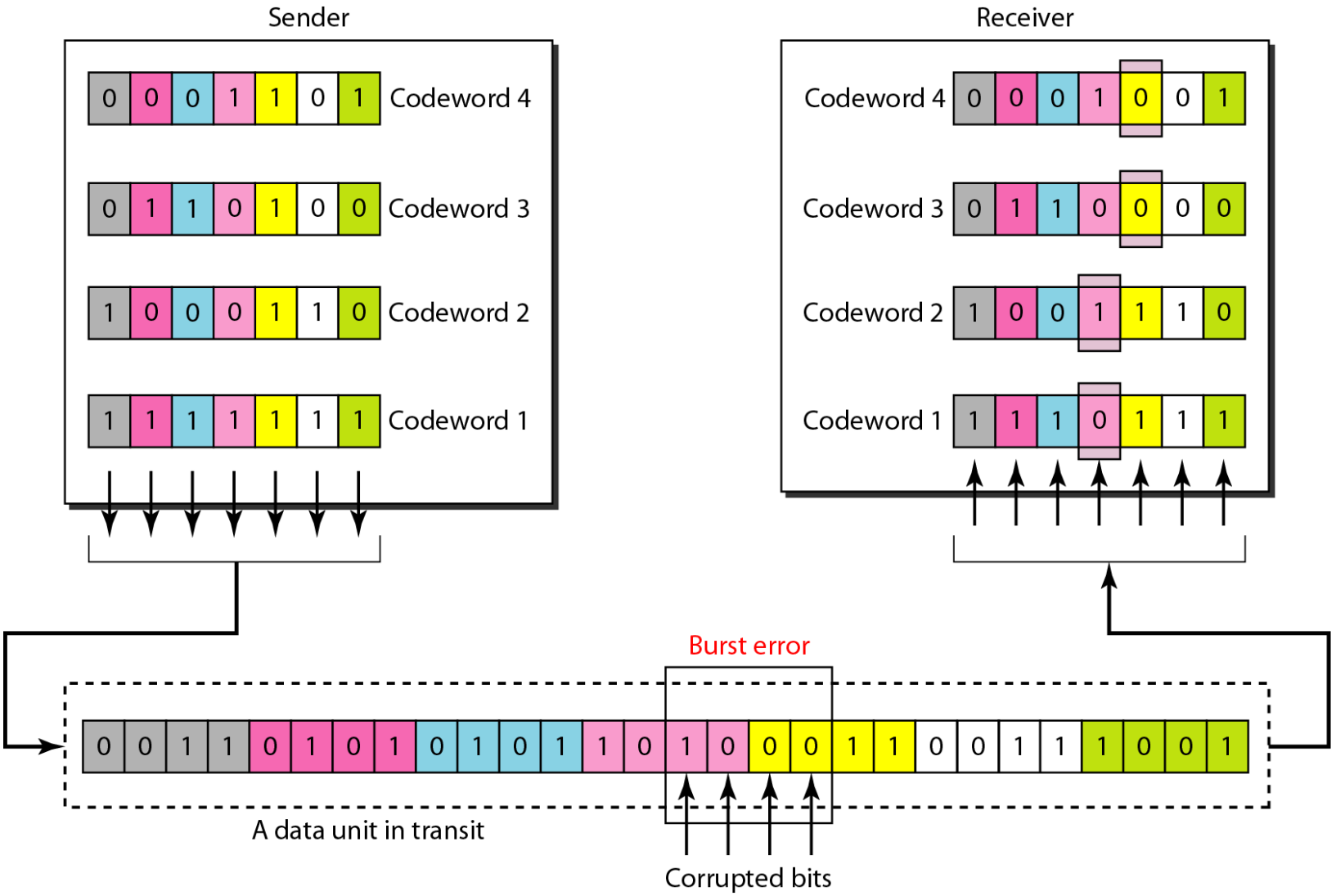
- We need a dataword of at least 7 bits. Calculate values of k and n that satisfy this requirement.
- We need to make $k = n - m$ greater than or equal to 7, or $2^m - 1 - m \geq 7$.
- If we set m = 3, the result is $n = 2^3 - 1 = 7$ and $k = 7 - 3$, or 4, which is < 7.
- If we set m = 4, then $n = 2^4 - 1 = 15$ and $k = 15 - 4 = 11$, which satisfies the condition k>7. So the code is

***C(15, 11)***

# Burst Errors

- Burst errors are very common, in particular in wireless environments where a fade will affect a group of bits in transit. The length of the burst is dependent on the duration of the fade.
- One way to counter burst errors, is to break up a transmission into shorter words and create a block (one word per row), then have a parity check per word.
- The words are then sent column by column. When a burst error occurs, it will affect 1 bit in several words as the transmission is read back into the block format and each word is checked individually.

# Burst error correction using Hamming code

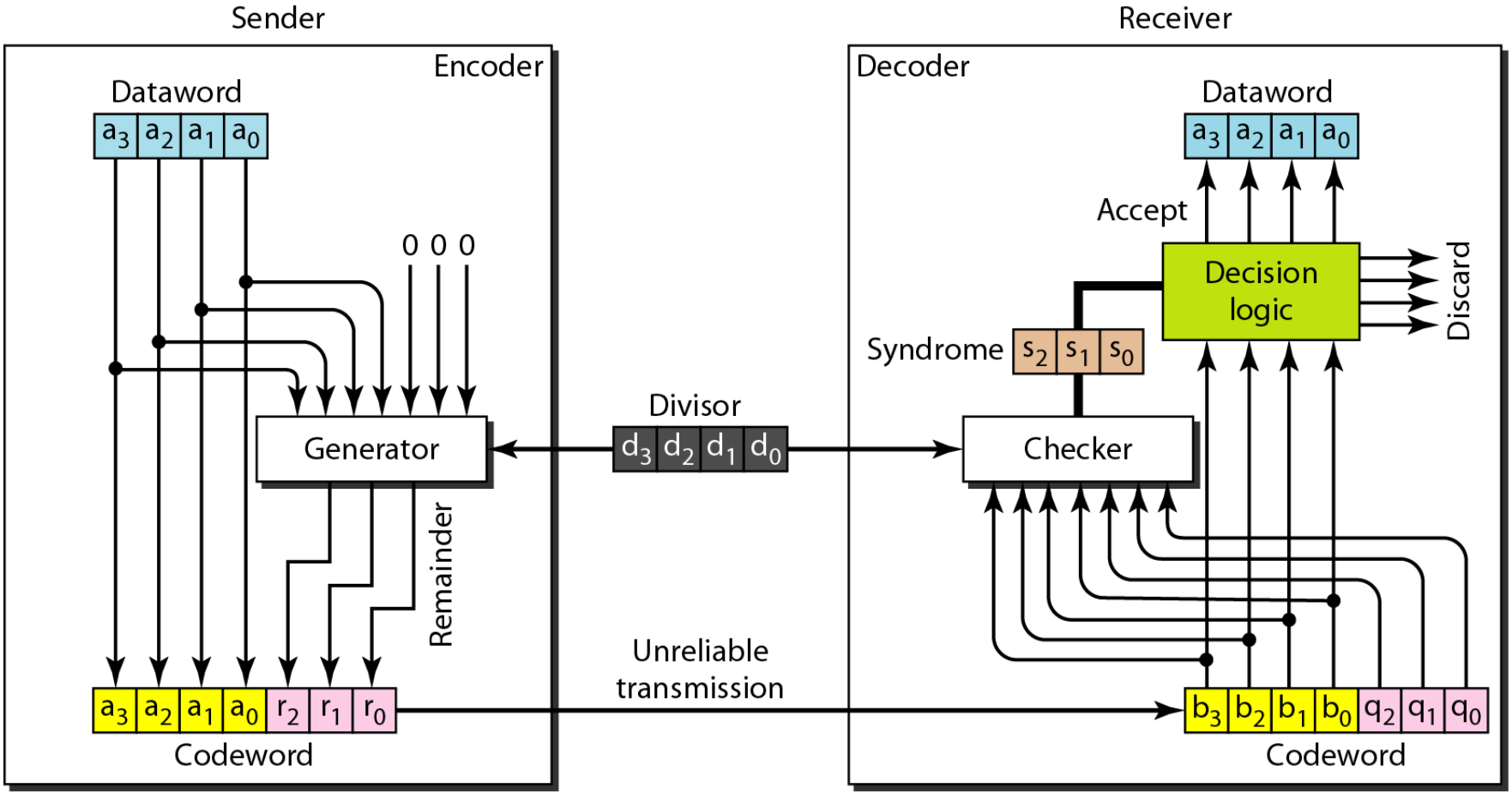


# Cyclic codes

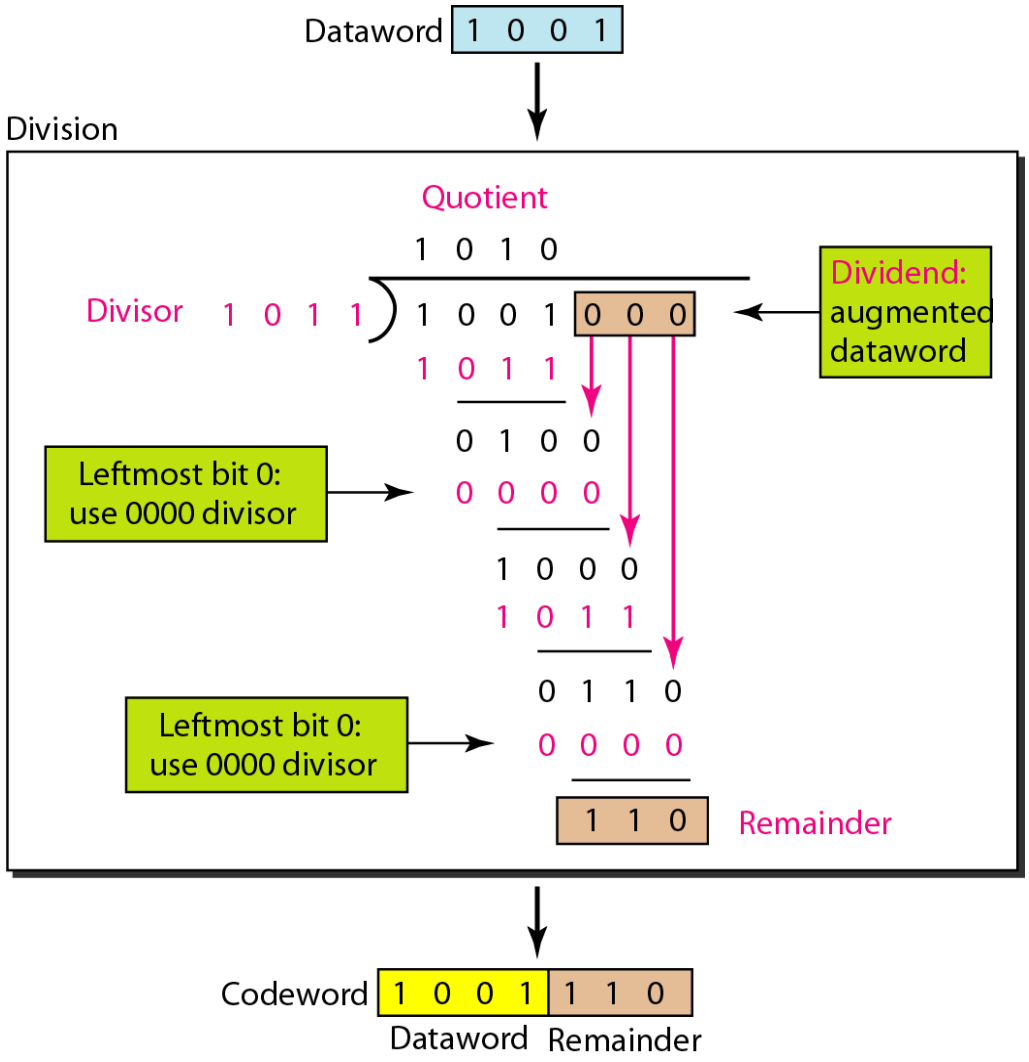
- Cyclic codes are special linear block codes with one extra property. In a cyclic code, if a codeword is cyclically shifted (rotated), the result is another codeword.
- Cyclic Redundancy Check

| *Dataword* | *Codeword* | *Dataword* | *Codeword* |
|---|---|---|---|
| 0000 | 0000000 | 1000 | 1000101 |
| 0001 | 0001011 | 1001 | 1001110 |
| 0010 | 0010110 | 1010 | 1010011 |
| 0011 | 0011101 | 1011 | 1011000 |
| 0100 | 0100111 | 1100 | 1100010 |
| 0101 | 0101100 | 1101 | 1101001 |
| 0110 | 0110001 | 1110 | 1110100 |
| 0111 | 0111010 | 1111 | 1111111 |

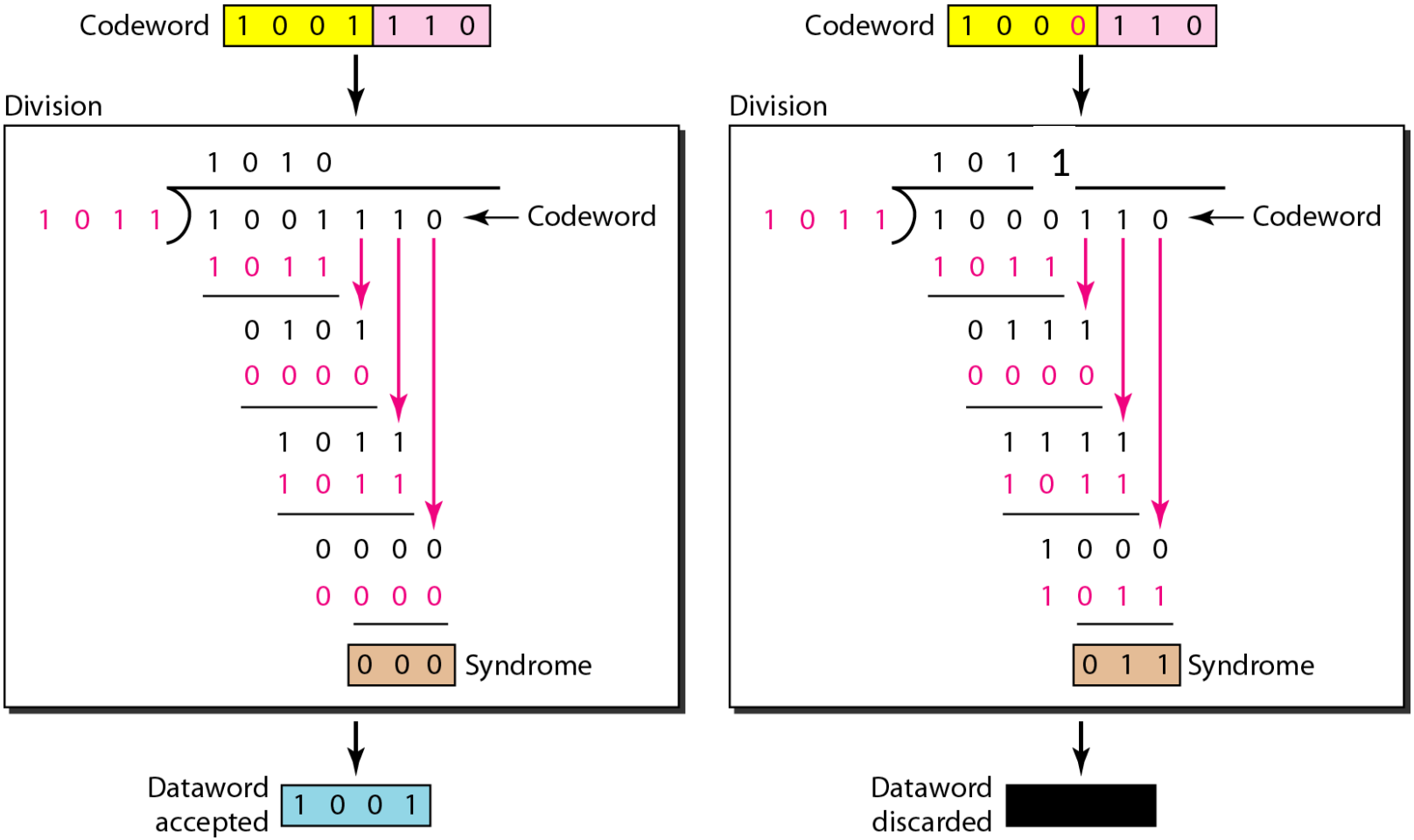
# CRC encoder and decoder



# Division in CRC encoder



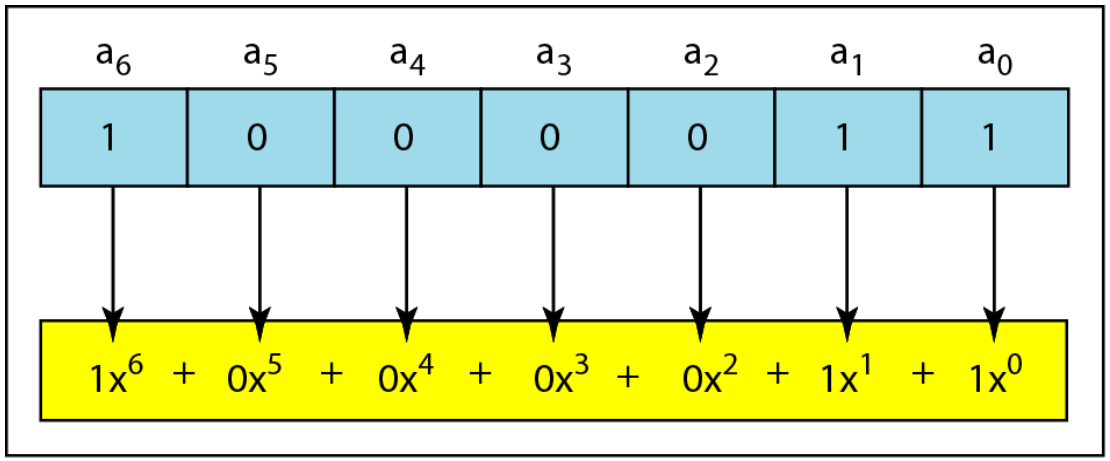
# Division in the CRC decoder for two cases



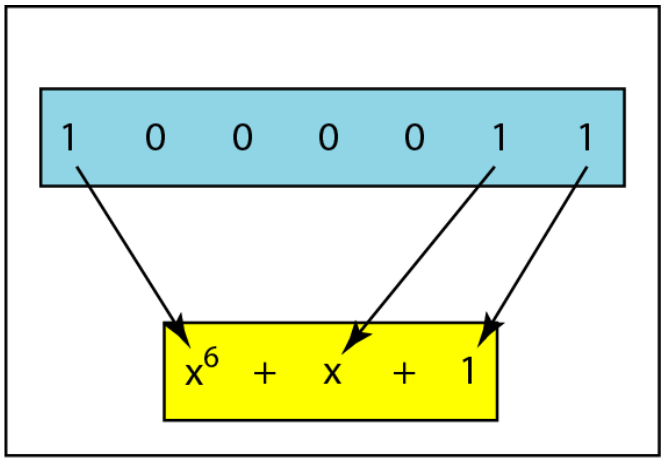
# Using Polynomials

- We can use a polynomial to represent a binary word.
- Each bit from right to left is mapped onto a power term.
- The rightmost bit represents the “0” power term. The bit next to it the “1” power term, etc.
- If the bit is of value zero, the power term is deleted from the expression.

# *A polynomial to represent a binary word*

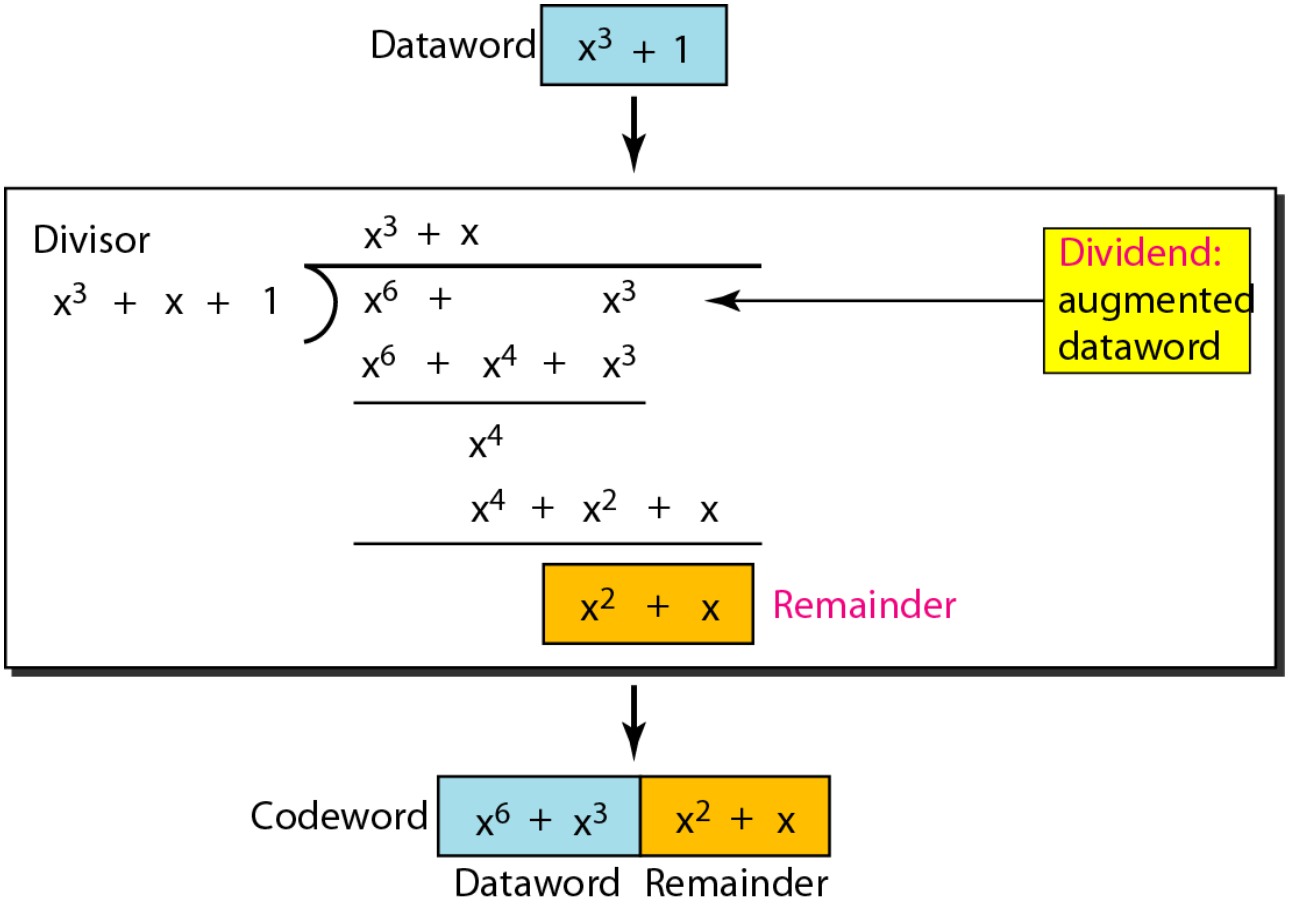



a. Binary pattern and polynomial




b. Short form

# CRC division using polynomials



- The divisor in a cyclic code is normally called the generator polynomial or simply the generator.
- In a cyclic code,
  - If $s(x) \neq 0$, one or more bits is corrupted.
  - If $s(x) = 0$, either
    a. No bit is corrupted. Or
    b. Some bits are corrupted, but the decoder failed to detect them.

- In a cyclic code, those *e*(x) errors that are divisible by *g*(x) are not caught.
- Received codeword (c(x) + e(x))/g(x) = c(x)/g(x) + e(x)/g(x)
  - The first part is by definition divisible and the second part will determine the error.
  - If “0” conclusion -> no error occurred.
    - Note: that could mean that an error went undetected.
- If the generator has more than one term and the coefficient of x0 is 1, all single errors can be caught.

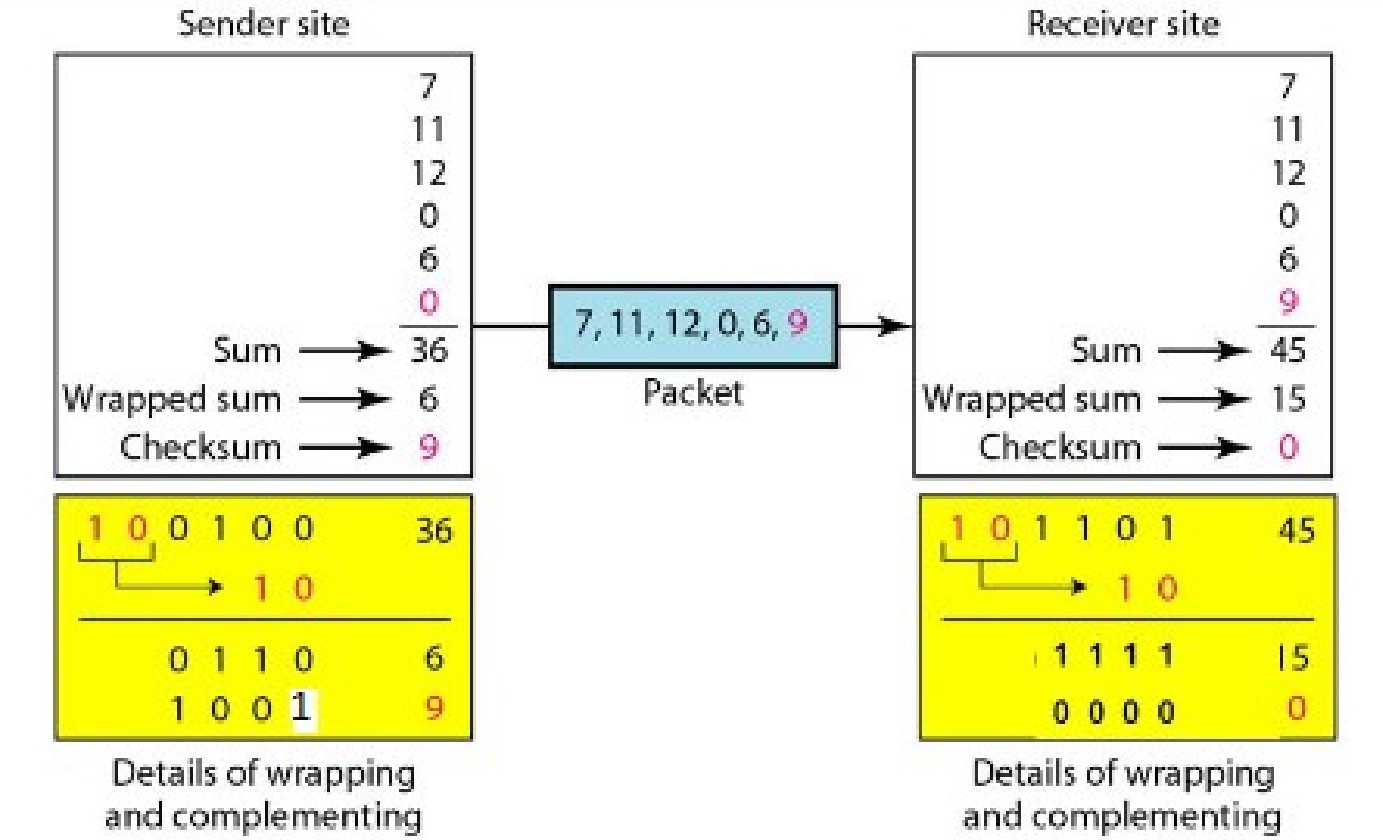
# Standard polynomials

| *Name* | *Polynomial* | *Application* |
|---|---|---|
| CRC-8 | $x^8 + x^2 + x + 1$ | ATM header |
| CRC-10 | $x^{10} + x^9 + x^5 + x^4 + x^2 + 1$ | ATM AAL |
| CRC-16 | $x^{16} + x^{12} + x^5 + 1$ | HDLC |
| CRC-32 | $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ | LANs |

# Checksum

- Checksum is an error detection method
- The checksum is used in the Internet by several protocols although not at the data link layer.
- 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.
- 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.

# Example


Sender site
7
11
12
0
6
0
Sum → 36
Wrapped sum → 6
Checksum → 9
1 0 0 1 0 0 36
1 0
0 1 1 0 6
1 0 0 1 9
Details of wrapping and complementing
7, 11, 12, 0, 6, 9
Packet
Receiver site
7
11
12
0
6
9
Sum → 45
Wrapped sum → 15
Checksum → 0
1 0 1 1 0 1 45
1 0
1 1 1 1 15
0 0 0 0 0
Details of wrapping and complementing

## 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.

## 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