

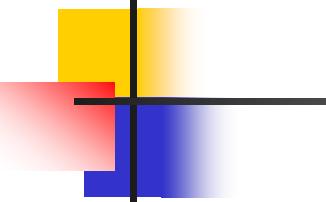
# **Module 2: Data Link and MAC Layer**

## **Error Control**

**Dr. Prasanna Shete**  
Dept. of Computer Engineering  
K. J. Somaiya School of Engineering

Slide Source: B. A. Forauzan, Data Communications and Networking, McGraw-Hill  
Online Learning Centre

[http://highered.mheducation.com/sites/0072967757/information\\_center\\_view0/index.html](http://highered.mheducation.com/sites/0072967757/information_center_view0/index.html)



## **Note**

---

**Data can be corrupted  
during transmission.**

**Some applications require that  
errors be detected and corrected.**

---

# 10-1 INTRODUCTION

**Topics discussed in this section:**

Types of Errors

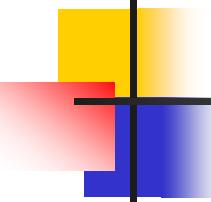
Redundancy

Detection Versus Correction

Forward Error Correction Versus Retransmission

Coding

Modular Arithmetic



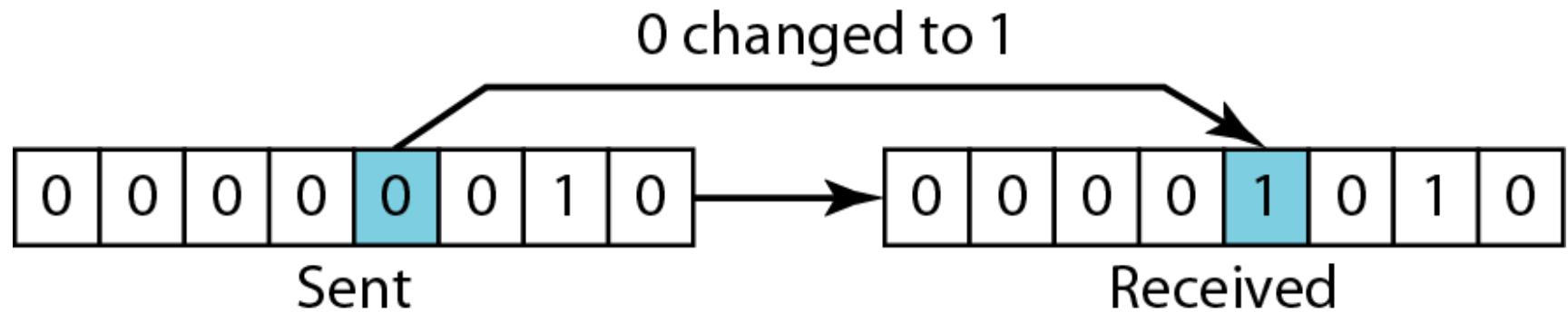
## **Note**

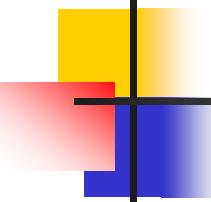
---

**Single-bit error:**  
**In a single-bit error, only 1 bit in the data unit is changed.**

---

**Figure 10.1** Single-bit error

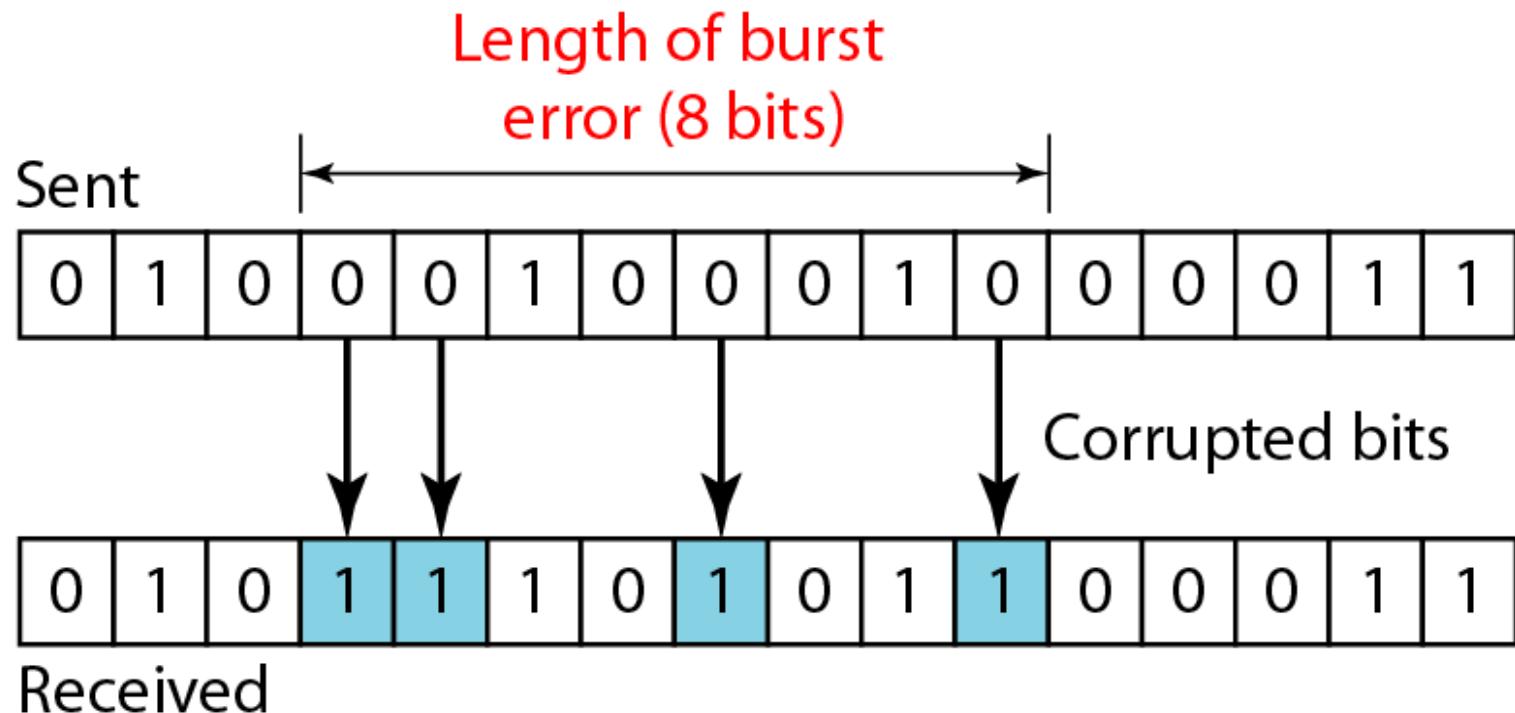


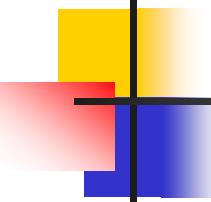


## *Note*

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

**Figure 10.2** *Burst error of length 8*



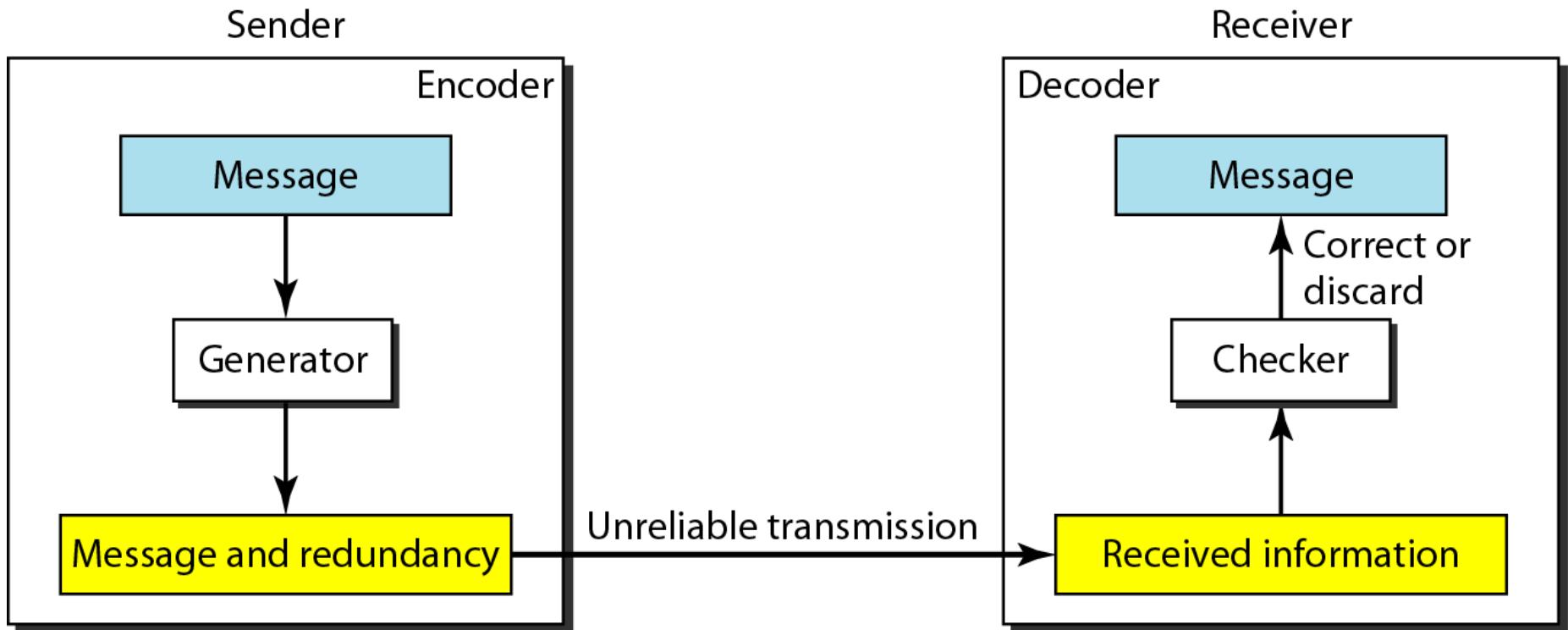


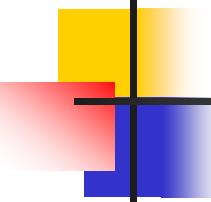
## *Note*

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

Sender/generator creates Relationship between the redundant bits and the actual bits

**Figure 10.3** *The structure of encoder and decoder*

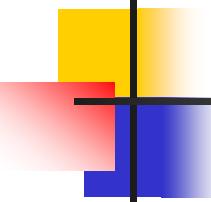




## *Note*

**Codes → Block codes and Convolution Codes**

**In this course, we concentrate on block codes; convolution codes not covered comprehensively**



## *Note*

### Modular Arithmetic

In modulo- $N$  arithmetic, we use only the integers in the range 0 to  $N - 1$ , inclusive.

## Modulo-2 Arithmetic

Modulus N=2; 0 and 1

Addition :  $0+0=0$ ,  $0+1=1$ ,  $1+0=1$ ,  $1+1=0$

Subtraction:  $0-0=0$ ,  $0-1=1$ ,  $1-0=1$ ,  $1-1=0$

**Figure 10.4** *XORing of two single bits or two words*

$$0 \oplus 0 = 0$$

$$1 \oplus 1 = 0$$

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

$$0 \oplus 1 = 1$$

$$1 \oplus 0 = 1$$

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

$$\begin{array}{r} 1 & 0 & 1 & 1 & 0 \\ \oplus & 1 & 1 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 & 0 \end{array}$$

c. Result of XORing two patterns

## 10-2 BLOCK CODING

*In block coding, message is divided into blocks, each of **k** bits, called **datawords**.*

*- **r** redundant bits are added to each block to make the length **n = k + r***

*The resulting n-bit blocks are called **codewords**.*

**Topics discussed in this section:**

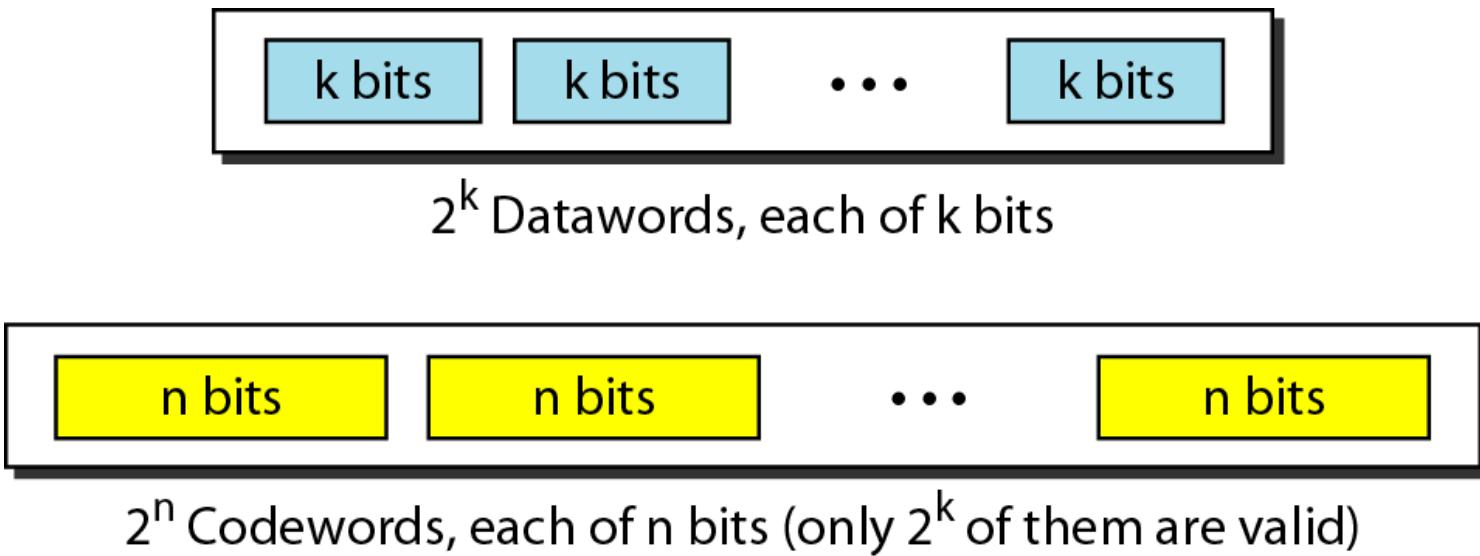
Error Detection

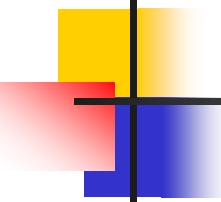
Error Correction

Hamming Distance

Minimum Hamming Distance

**Figure 10.5** *Datawords and codewords in block coding*





## *Example 10.1*

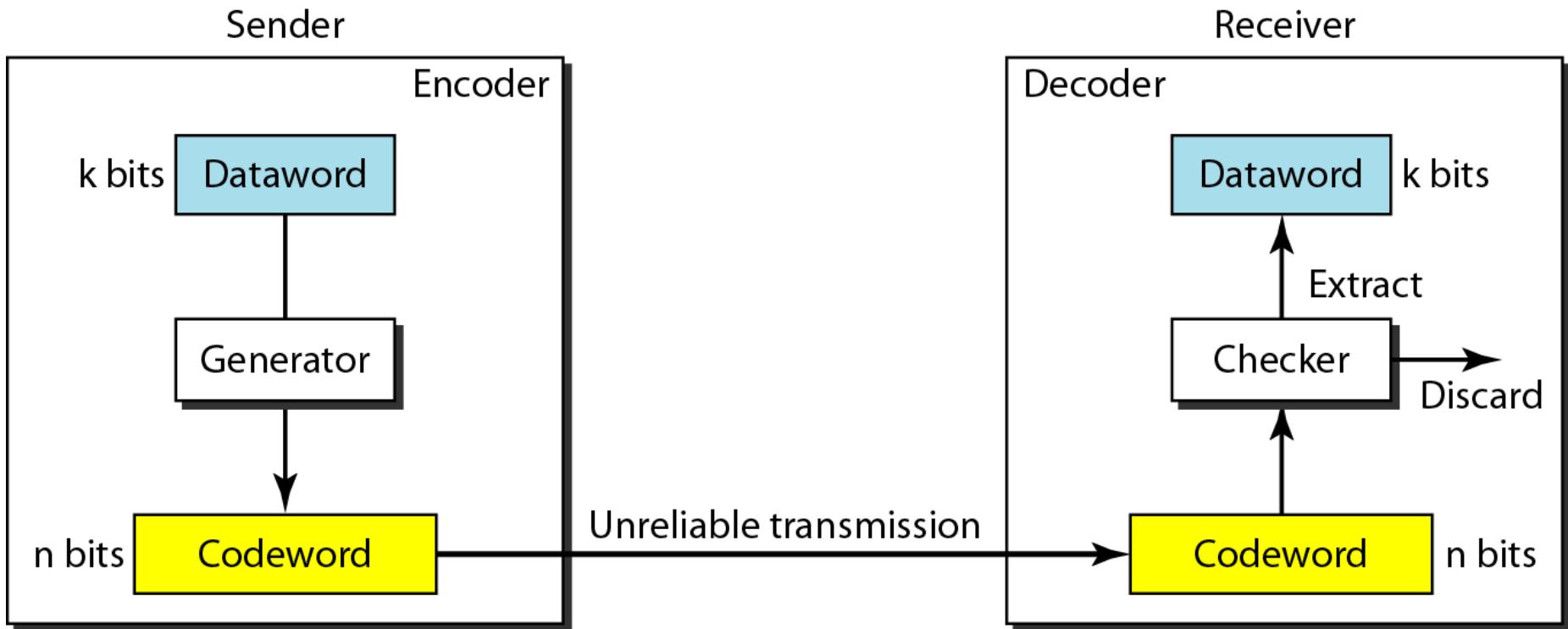
*The 4B/5B block coding is a good example of this type of coding.*

*In this coding scheme,  $k = 4$  and  $n = 5$*

- we have  $2^k = 16$  datawords
- and  $2^n = 32$  codewords

*16 out of 32 codewords are used for message transfer and the rest are either unused or used for other purposes*

**Figure 10.6** *Process of error detection in block coding*



## *Example 10.2*

*Let us assume that  $k = 2$  and  $n = 3$ . Table 10.1 shows the list of datawords and codewords.*

<i>Datawords</i>	<i>Codewords</i>
00	000
01	011
10	101
11	110

## *Example 10.2*

*Assume the sender encodes the dataword 01 as 011 and sends it to the receiver. Consider the following cases:*

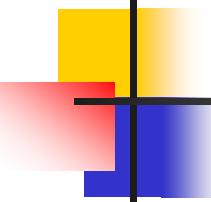
1. *The receiver receives 011. It is a valid codeword. The receiver extracts the dataword 01 from it.*
2. *The codeword is corrupted during transmission, and 111 is received. This is not a valid codeword and is discarded.*
3. *The codeword is corrupted during transmission, and 000 is received. This is a valid codeword.*

*The receiver incorrectly extracts the dataword 00.*

**→ Two corrupted bits have made the error undetectable.**

**Table 10.1** *A code for error detection (Example 10.2)*

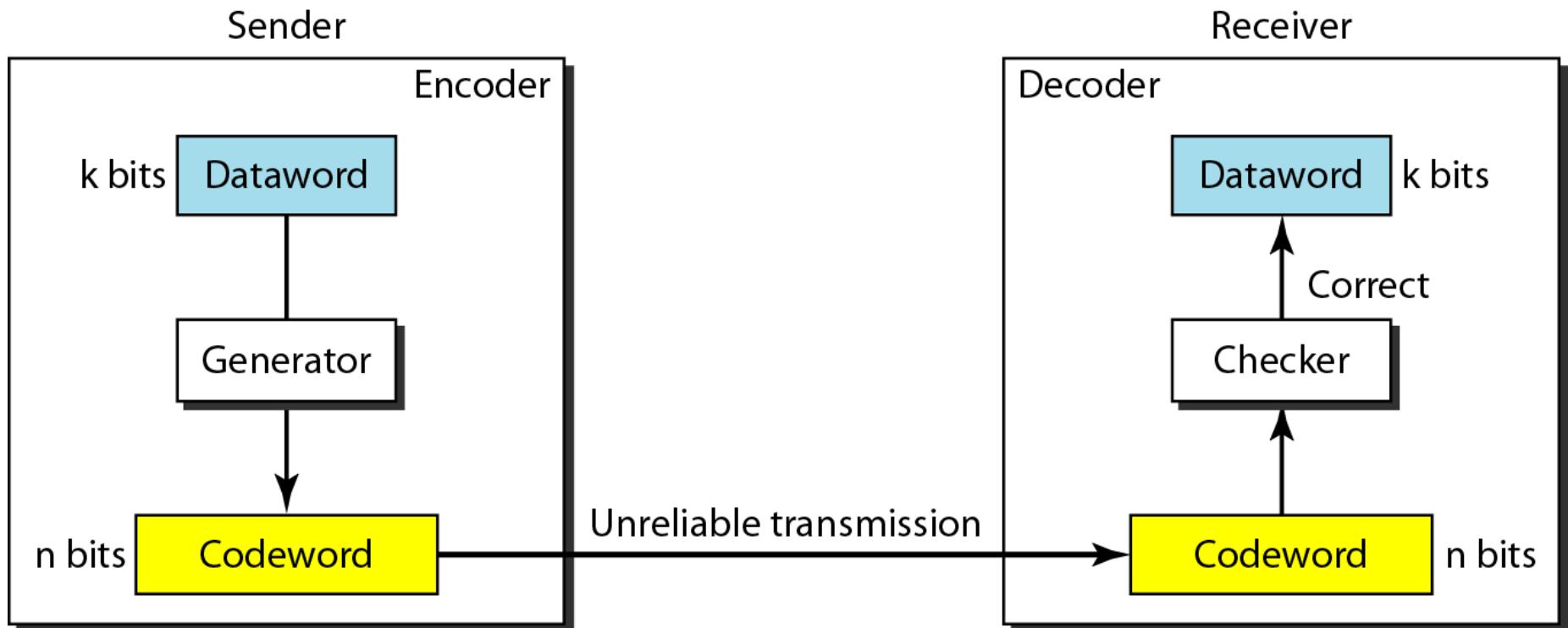
<i>Datawords</i>	<i>Codewords</i>
00	000
01	011
10	101
11	110



## *Note*

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

**Figure 10.7** Structure of encoder and decoder in *error correction*



## *Example 10.3*

*Let us add more redundant bits to Example 10.2 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.*

*Table 10.2 shows the datawords and codewords.*

<i>Dataword</i>	<i>Codeword</i>
00	00000
01	01011
10	10101
11	11110

- 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 → means an error has occurred.
- The receiver, assuming that there is only 1 bit corrupted, uses the following strategy to guess the correct dataword:

## *Example 10.3 (continued)*

- 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.*

**Table 10.2** *A code for error correction (Example 10.3)*

<i>Dataword</i>	<i>Codeword</i>
00	00000
01	01011
10	10101
11	11110

## *Hamming Distance*

**The Hamming distance between two words is the number of differences between corresponding bits.**

*The Hamming Distance between the received codeword and the sent codeword is the number of bits that are corrupted during transmission*

## *Example 10.4: finding hamming distance*

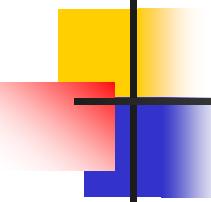
*Let us find the Hamming distance between two pairs of words.*

**1.** *The Hamming distance  $d(000, 011)$  is 2 because*

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

**2.** *The Hamming distance  $d(10101, 11110)$  is 3 because*

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



## *Note*

The minimum Hamming distance is the smallest Hamming distance between all possible pairs in a set of words.

## *Example 10.5*

*Find the minimum Hamming distance of the coding scheme in Table 10.1.*

### *Solution*

*We first find all Hamming distances.*

$$\begin{array}{llll} d(000, 011) = 2 & d(000, 101) = 2 & d(000, 110) = 2 & d(011, 101) = 2 \\ d(011, 110) = 2 & d(101, 110) = 2 & & \end{array}$$

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

<i>Datawords</i>	<i>Codewords</i>
00	000
01	011
10	101
11	110

## *Example 10.6*

*Find the minimum Hamming distance of the coding scheme in Table 10.2.*

<i>Dataword</i>	<i>Codeword</i>
00	00000
01	01011
10	10101
11	11110

*Solution*

*We first find all the Hamming distances.*

$$d(00000, 01011) = 3$$

$$d(00000, 10101) = 3$$

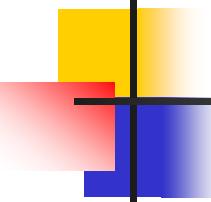
$$d(00000, 11110) = 4$$

$$d(01011, 10101) = 4$$

$$d(01011, 11110) = 3$$

$$d(10101, 11110) = 3$$

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

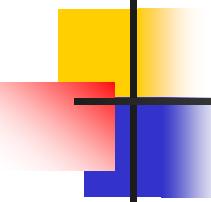


## **Note**

---

To guarantee the detection of up to  $s$  errors in all cases, the minimum Hamming distance in a block code must be  $d_{min} = s + 1$ .

---

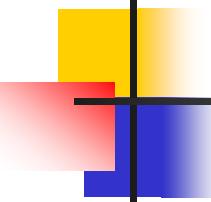


## *Example 10.7*

*The minimum Hamming distance for our first code scheme (Table 10.1) 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.*



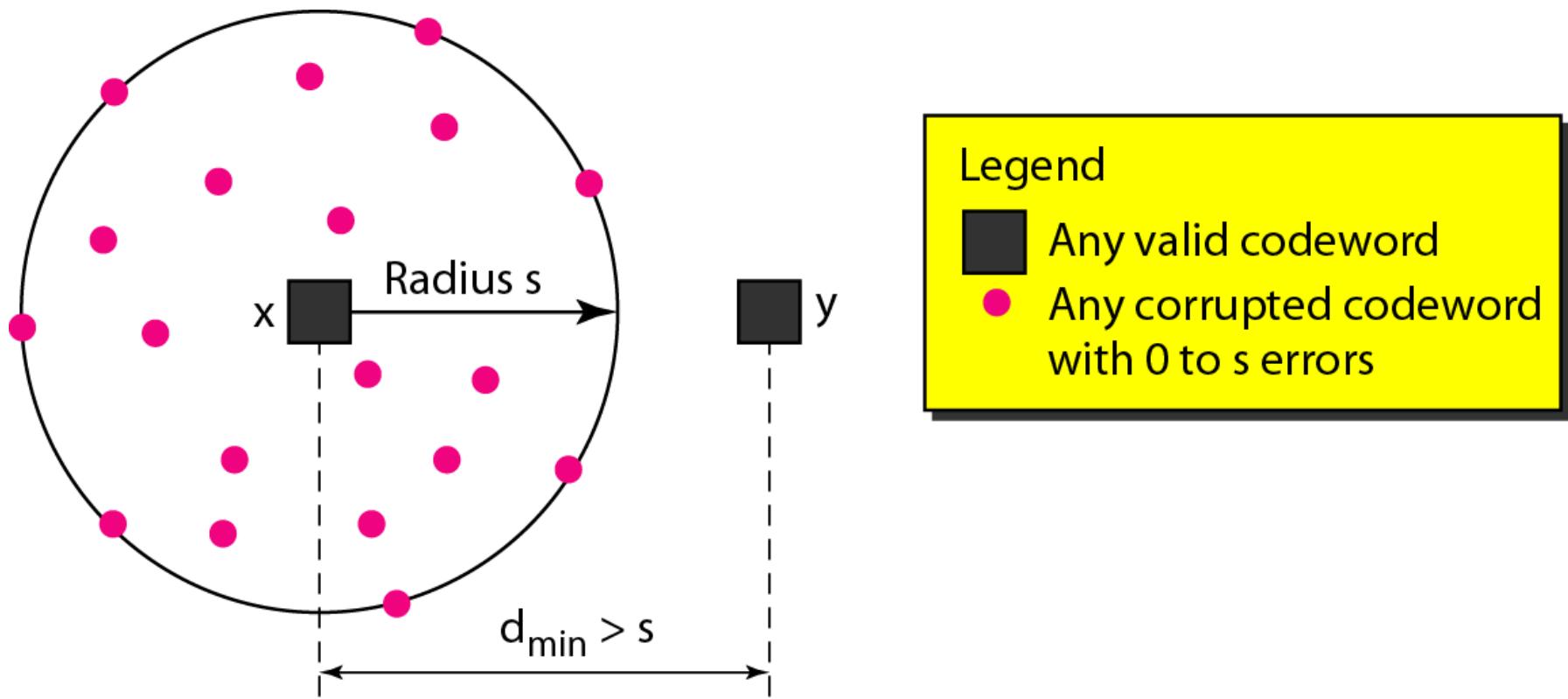
## *Example 10.8*

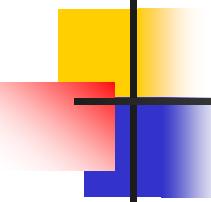
*Our second block code scheme (Table 10.2) has  $d_{min} = 3$ . This code can detect up to two errors.*

*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.*

**Figure 10.8** Geometric concept for finding  $d_{min}$  in error detection

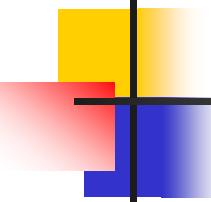




## **Note**

---

To guarantee correction of up to  $t$  errors in all cases, the minimum Hamming distance in a block code must be  $d_{min} = 2t + 1$ .



## *Example 10.9*

*A code scheme has a Hamming distance  $d_{min} = 4$ . What is the error detection and correction capability of this scheme?*

### **Solution**

*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, . . .).*

## 10-3 LINEAR BLOCK CODES

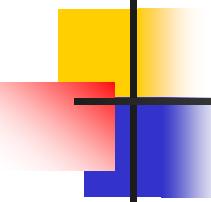
*Almost all block codes used today belong to a subset called **linear block codes***

*A linear block code is a code in which the exclusive OR (modulo-2 addition) of two valid codewords creates another valid codeword.*

**Topics discussed in this section:**

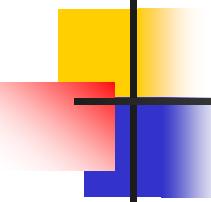
Minimum Distance for Linear Block Codes

Some Linear Block Codes



## *Note*

**In a linear block code, the exclusive OR (XOR) of any two valid codewords creates another valid codeword.**



## *Example 10.10*

*Let us see if the two codes we defined in Table 10.1 and Table 10.2 belong to the class of linear block codes.*

- 1. The scheme in Table 10.1 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.*
- 2. The scheme in Table 10.2 is also a linear block code. We can create all four codewords by XORing two other codewords.*

## Example 10.11

In our first code (Table 10.1), the numbers of 1s in the nonzero codewords are 2, 2, and 2. So the minimum Hamming distance is  $d_{min} = 2$ .

In the second code (Table 10.2), the numbers of 1s in the nonzero codewords are 3, 3, and 4. So we have  $d_{min} = 3$ .

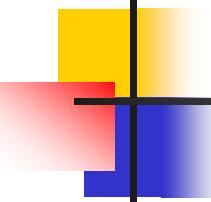
Datawords	Codewords
00	000
01	011
10	101
11	110

Dataword	Codeword
00	00000
01	01011
10	10101
11	11110

## Simple Parity Check Code

---

- Here, the *k*-bit dataword is changed to an *n*-bit codeword where  $n = k+1$ 
  - Extra bit is called the parity bit; selected to make the total number of 1s in the codeword even (or odd)
- Min. hamming distance  $d_{min} = 2 \rightarrow$  it's a single bit-error detecting code
  - Cannot correct error



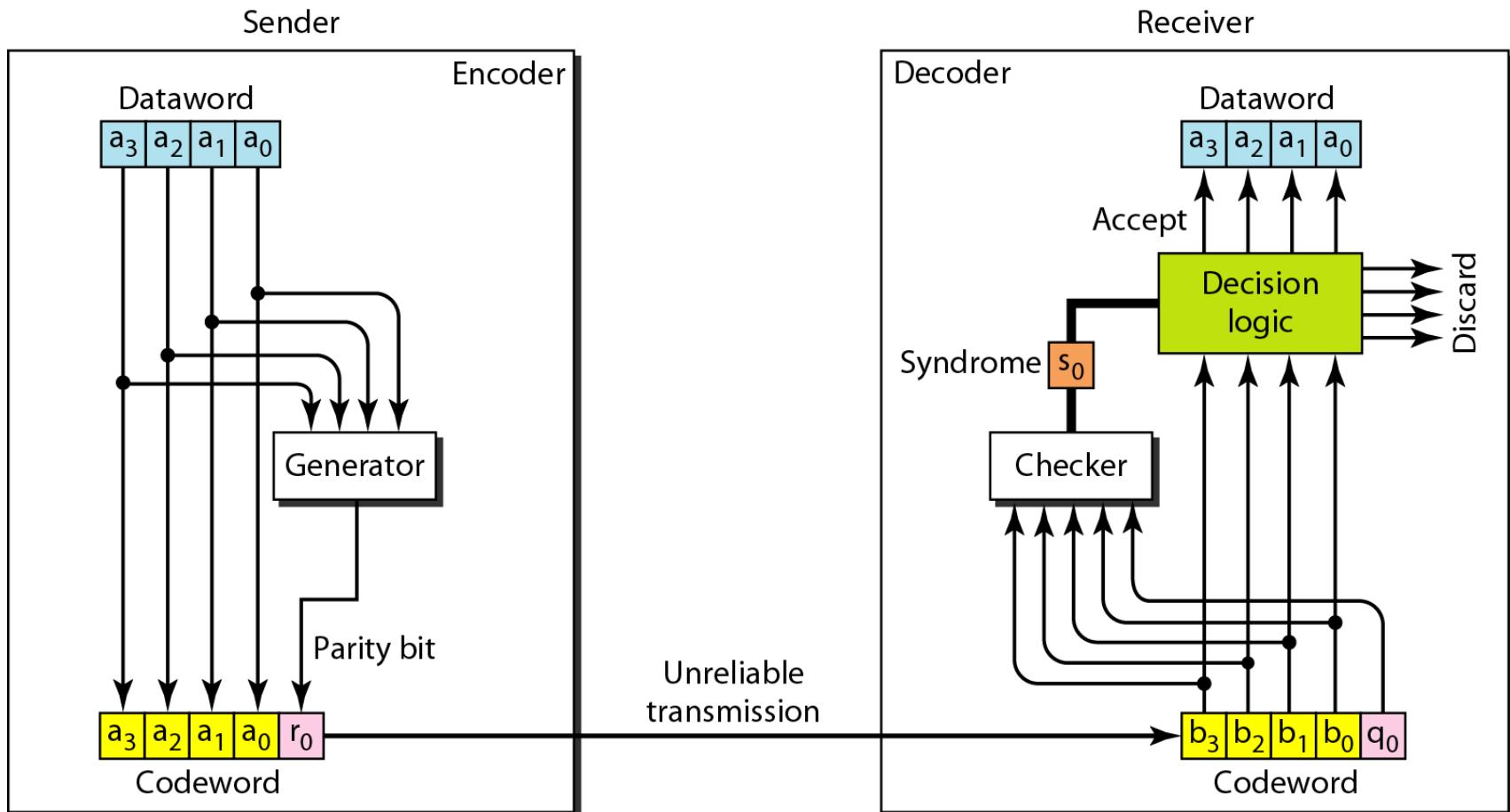
## *Note*

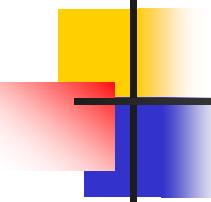
**A simple parity-check code is a single-bit error-detecting code in which  $n = k + 1$  with  $d_{\min} = 2$ .**

**Table 10.3** *Simple parity-check code C(5, 4)*

<i>Datawords</i>	<i>Codewords</i>	<i>Datawords</i>	<i>Codewords</i>
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

**Figure 10.10 Encoder and decoder for simple parity-check code**





## *Example 10.12*

*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<sub>1</sub>**; The received codeword is **10011**. The syndrome is 1 → No dataword is created.*
- 3. One single-bit error changes **r<sub>0</sub>**. The received codeword is **10110**. The syndrome is 1 → No dataword is created*

## *Example 10.12 (continued)*

**4.** An error changes  $r_0$  and a second error changes  $a_3$ .

The received codeword is **00110**. The syndrome is 0.

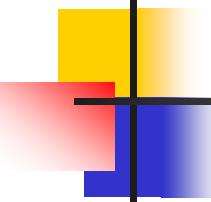
The dataword **0011** is created at the receiver;  
here the dataword is wrongly created due to the  
syndrome value

**5.** Three bits— $a_3$ ,  $a_2$ , and  $a_1$ —are changed by errors.

The received codeword is **01011** The syndrome is 1

→No dataword is created

→This shows that the simple parity check, guaranteed to detect one single error, can also find any odd number of errors.



*Note*

---

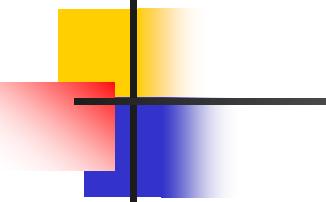
**A simple parity-check code can detect  
an odd number of errors.**

---

## Hamming Code

---

- Designed with  $d_{min}=3$ 
  - Can detect 2 bit errors and correct single bit error
- Relationship between  $n$  and  $k$ :
  - Choose integer  $r \geq 3$
  - the value of  $n$  and  $k$  are calculated as:  
$$n = 2^r - 1 \text{ and } k = n - r$$
  - $2^r \geq r + k + 1$  ( $r$ -parity bits; for understanding you may take  $p$  instead of  $r$ )
  - → No. of redundant bits reqd. depends on no. of data bits and can be calculated as:  $r \geq \log_2(r + k + 1)$



## **Note**

---

**All Hamming codes discussed in this book have  $d_{min} = 3$ .**

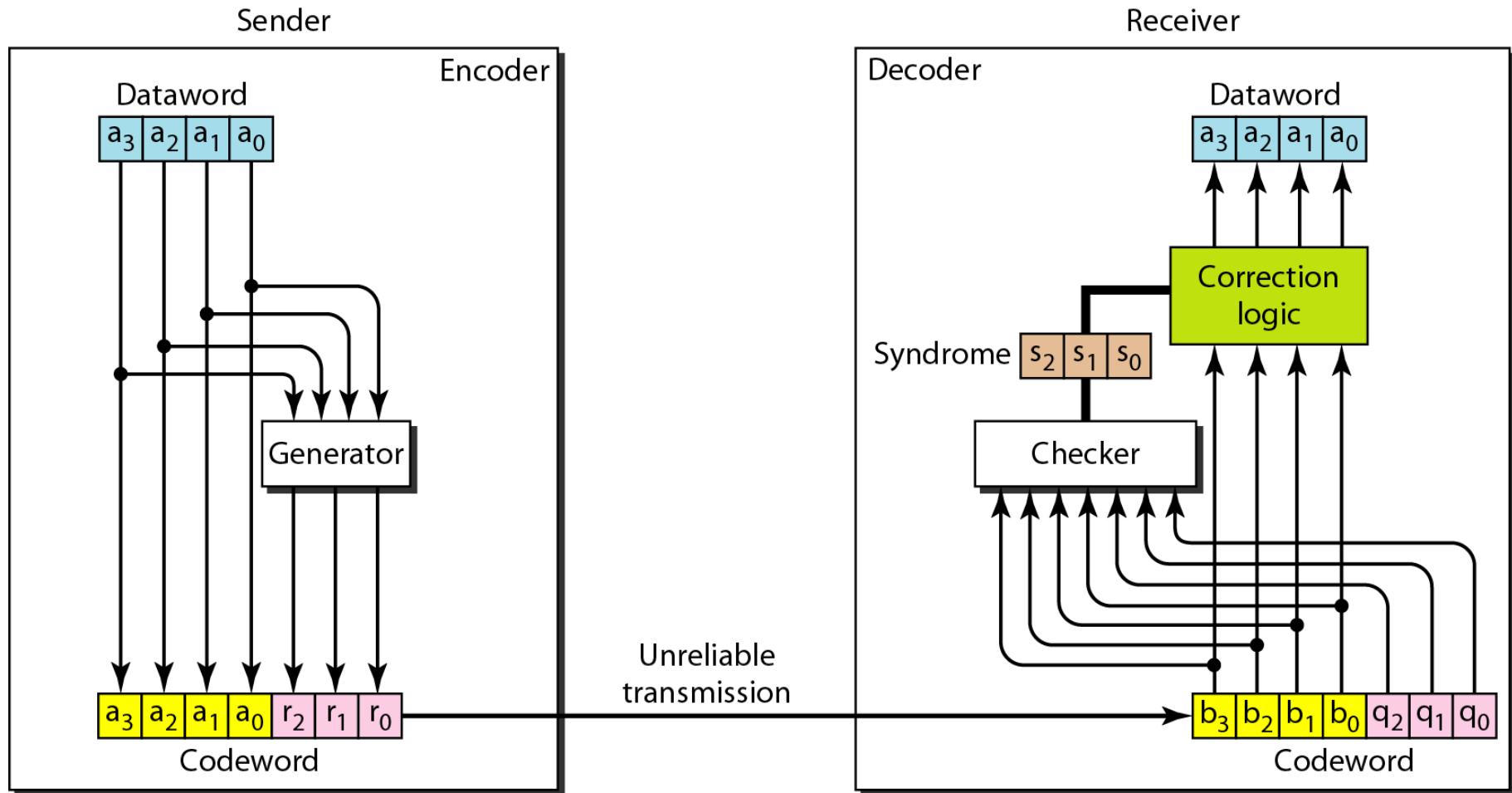
**The relationship between  $r$  and  $n$  in these codes is  $n = 2^r - 1$**

---

**Table 10.4** *Hamming code C(7, 4)*

<i>Datawords</i>	<i>Codewords</i>	<i>Datawords</i>	<i>Codewords</i>
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

**Figure 10.12** The structure of the encoder and decoder for a Hamming code



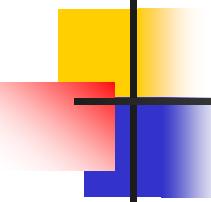
## Hamming Code: Generator

---

- Parity check bits ( $r_0, r_1, r_2$ ) calculation at Encoder
  - $r_0 = a_2 + a_1 + a_0$  (modulo 2)
  - $r_1 = a_3 + a_2 + a_1$
  - $r_2 = a_1 + a_0 + a_3$
- Syndrome bits calculation at Decoder
  - $s_0 = b_2 + b_1 + b_0 + q_0$  (modulo 2)
  - $s_1 = b_3 + b_2 + b_1 + q_1$
  - $s_2 = b_1 + b_0 + b_3 + q_2$

**Table 10.5** *Logical decision made by the correction logic analyzer*

<i>Syndrome</i>	000	001	010	011	100	101	110	111
<i>Error</i>	None	$q_0$	$q_1$	$b_2$	$q_2$	$b_0$	$b_3$	$b_1$



## *Example 10.13*

*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 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 syndrome is 101. After flipping  $b_0$ , we get 0000, the wrong dataword. This shows that our code cannot correct two errors.*

## *Example 10.14*

We need to send a dataword of atleast 7 bits. Calculate values of  $r$  and  $n$ .

### *Solution*

We need to make  $k = n - r$  greater than or equal to 7, or  $2^m - 1 - m \geq 7$ . (since  $2^r \geq r+k+1$ )

- 1.** If we set  $r = 3$ , the result is  $n = 2^3 - 1$  and  $k = 7 - 3$ , or 4, which is not acceptable.
- 2.** If we set  $r = 4$ , then  $n = 2^4 - 1 = 15$  and  $k = 15 - 4 = 11$ , which satisfies the condition. So the code is

$$C(15, 11)$$

## Structure of Hamming Code

- In Hamming code, position of parity check bits and data bits is predefined. The code structure is as follows:

...	7	6	5	4	3	2	1
	D7	D6	D5	P4	D3	P2	P1
...	111	110	101	100	011	010	001

- Parity check bits are placed at the bit position (index) that is power of 2 (have a single 1 bit in the binary form of their position)
- Data bits placed at other positions (two or more 1 bits in the binary form of their position)

# Algorithm

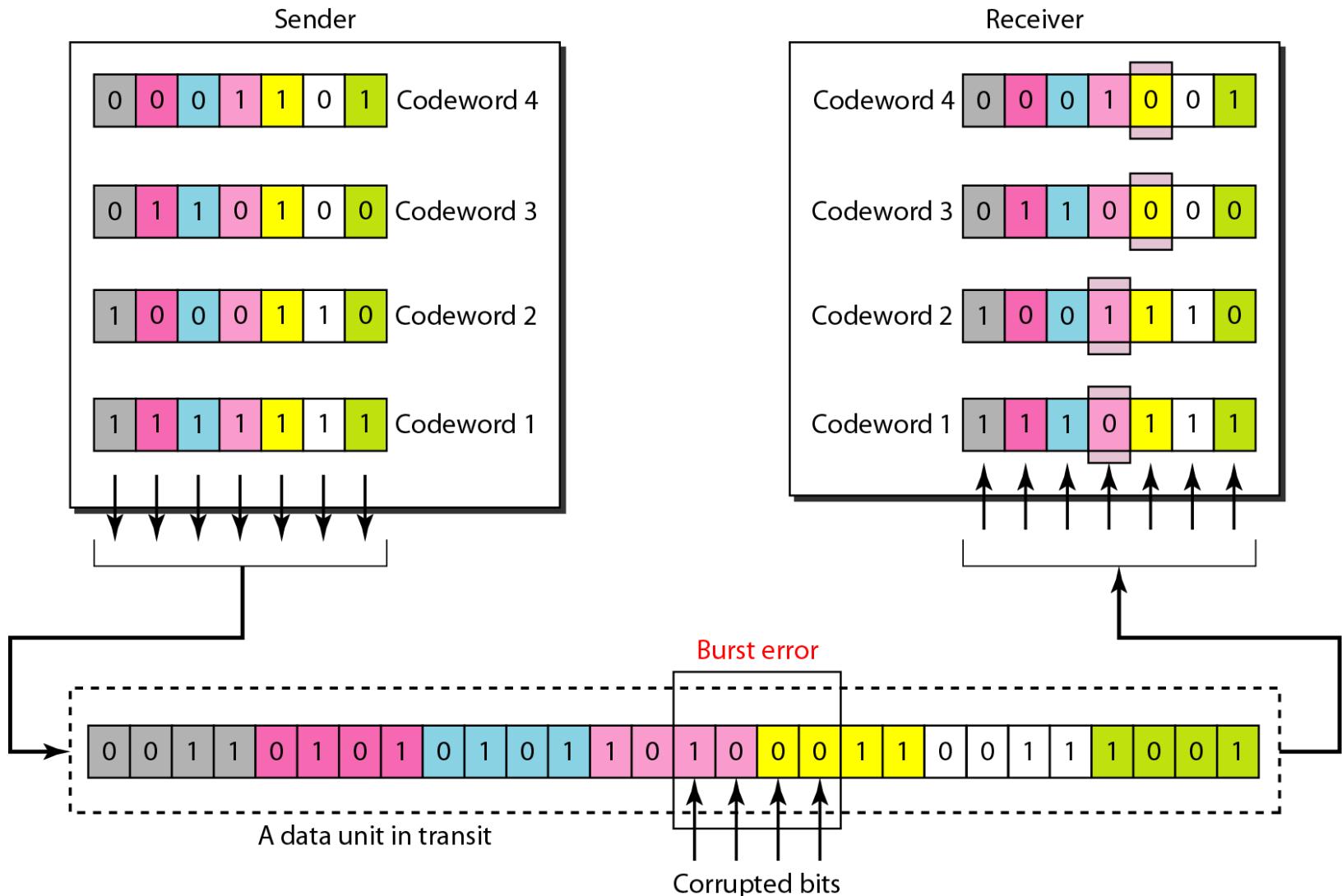
- Steps to implement the algorithm at encoder:
  1. Number the bits as 1, 2, 3, 4, 5, 6, 7,...etc.
  2. Write the bit numbers in binary: 1, 10, 11, 100, 101, 110, 111, etc.
  3. All bit positions that are powers of 2 (two) are **parity bits**: 1, 2, 4, 8, etc. (1, 10, 100, 1000)
  4. All other bit positions, are **data bits**.

1. Parity Bit: Each data bit is included in a unique set of 2 or more parity bits, as determined by the binary form of its bit position.
  1. Parity bit P1 covers all bit positions which have the **least** significant bit set: bit 1 (the parity bit itself), 3, 5, 7, 9, etc.
  2. Parity bit P2 covers all bit positions which have the **second** least significant bit set: bit 2 (the parity bit itself), 3, 6, 7, 10, 11, etc.
  3. Parity bit P4 covers all bit positions which have the **third** least significant bit set: bits 4–7, 12–15, 20–23, etc.
  4. Parity bit P8 covers all bit positions which have the **fourth** least significant bit set: bits 8–15, 24–31, 40–47, etc.

In general each parity bit covers all bits where the bitwise AND of the parity position and the bit position is non-zero.

- **Error detection algorithm at decoder:**
- The receiver finds the syndrome by Ex-ORing data bits along with respective parity bits
- If the syndrome bit is all zeroes → No error
- Else there is an error; the syndrome value shows which bit position is in an error
- [E.g in (7, 4) Hamming code, if  $s = 110 \rightarrow$  Sixth bit (D6) is in error]

**Figure 10.13** Burst error correction using Hamming code



## 10-4 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.*

### *Topics discussed in this section:*

Cyclic Redundancy Check

Hardware Implementation

Polynomials

Cyclic Code Analysis

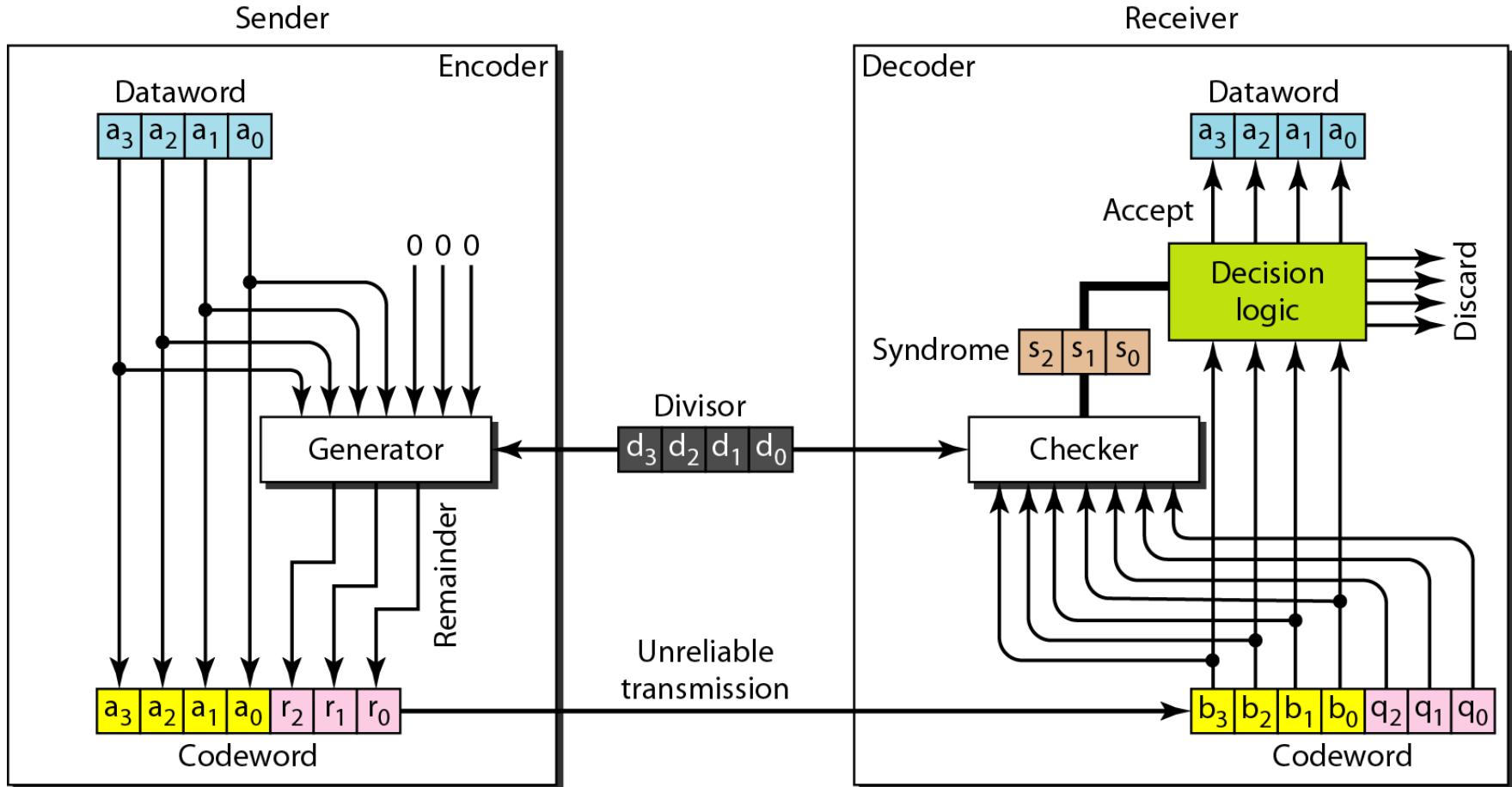
Advantages of Cyclic Codes

Other Cyclic Codes

**Table 10.6** A CRC code with  $C(7, 4)$

<i>Dataword</i>	<i>Codeword</i>	<i>Dataword</i>	<i>Codeword</i>
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

**Figure 10.14** CRC encoder and decoder



# Algorithm

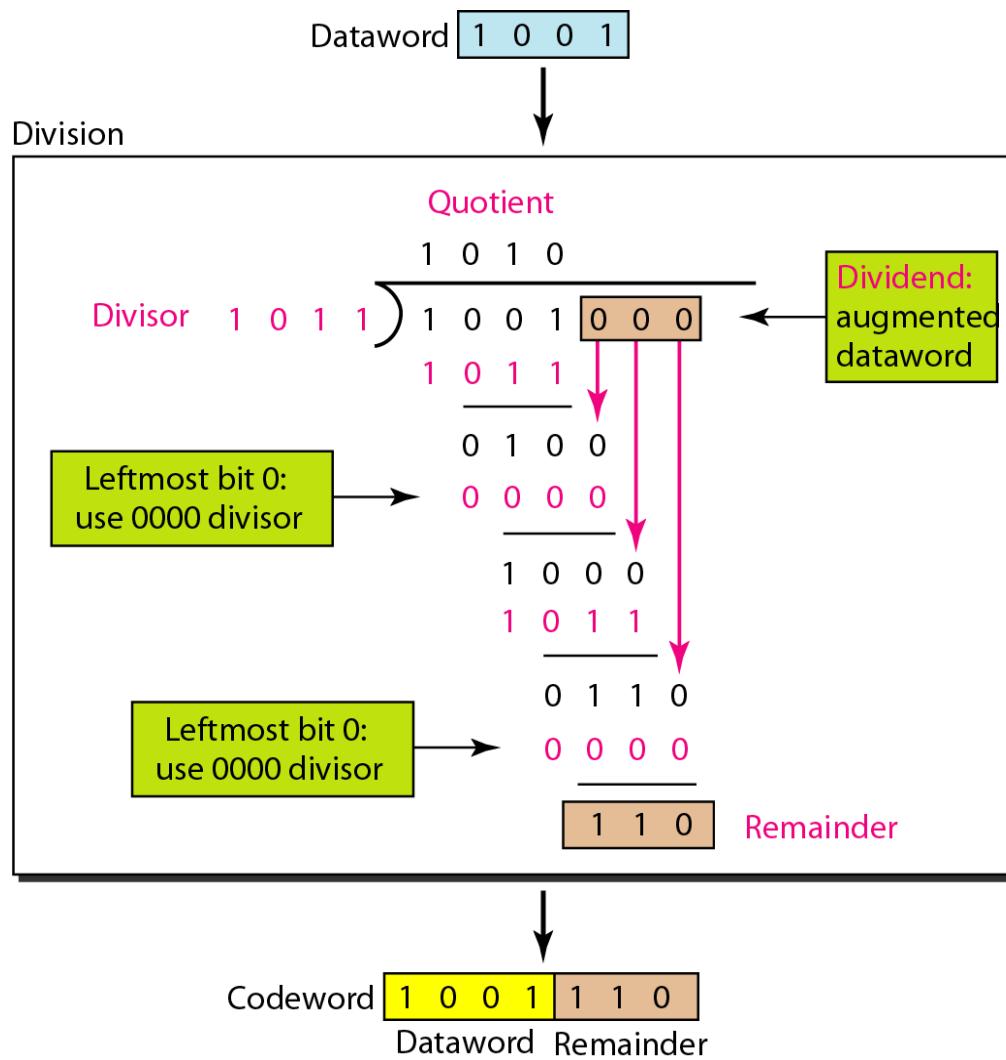
- Steps to implement the algorithm at encoder:

Dataword ( $k$ ) = 4 bits; Codeword ( $n$ ) = 7 bits

1. Augment Dataword  $k$  with  $n-k$  (= 3), 0's at the RHS
2. Feed n-bit result into the generator
3. Generator uses predefined and agreed upon divisor of size  $n-k+1$  (= 4) bits
4. Generator divides Augmented Dataword by the divisor → modulo 2 division
5. Quotient is discarded and the remainder is appended to the Dataword to create the Codeword

Ex. Dataword: 1001, Divisor: 1011

**Figure 10.15** Division in CRC encoder



# Algorithm

Decoder performs the same division process

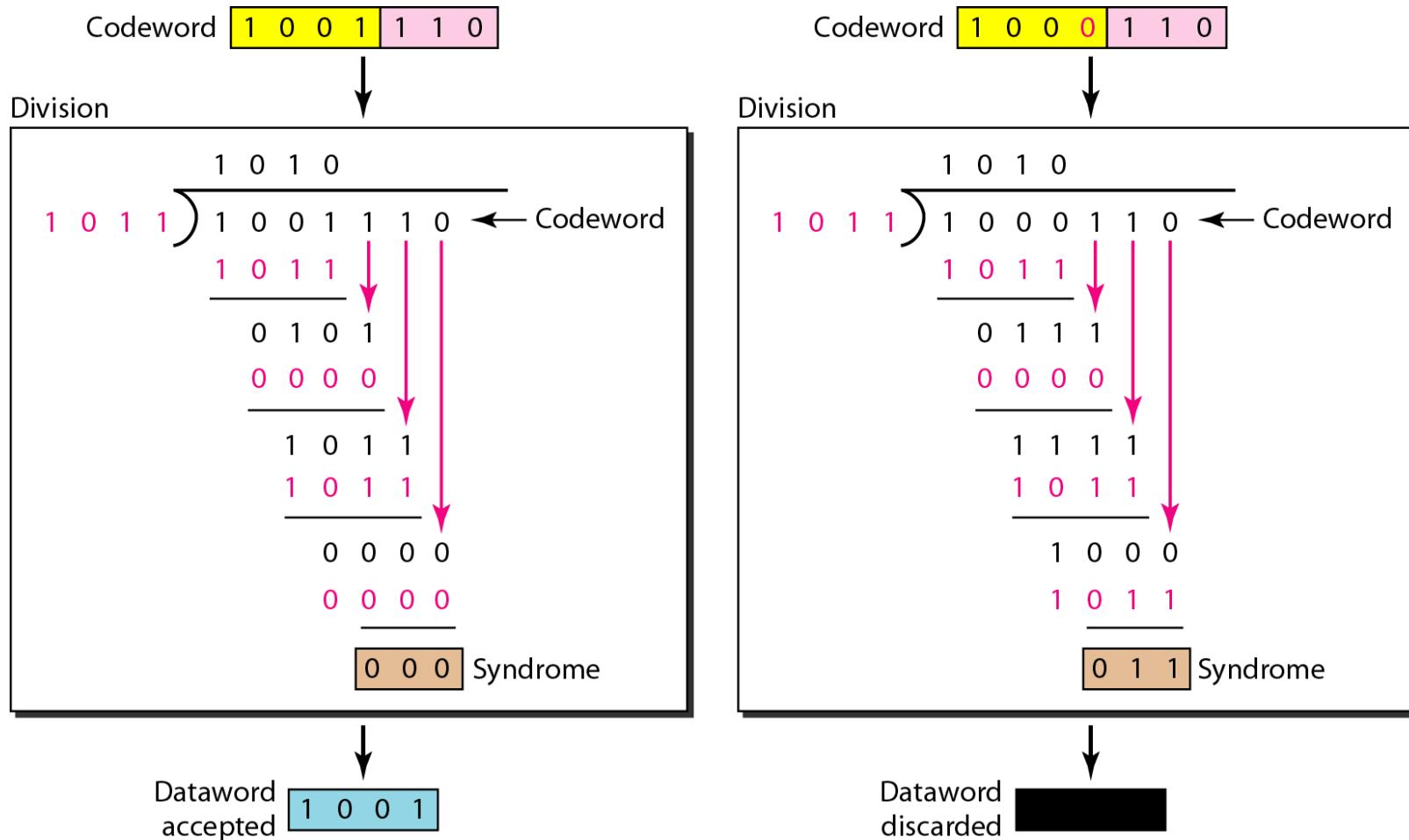
1. Divides received Codeword by the divisor → modulo 2 division
2. Quotient is discarded and the remainder is taken as Syndrome
3. If Syndrome is all 0's → No Error;
4. the dataword is separated from the received codeword and accepted; discarded otherwise

Ex. Transmitted codeword: 1001110

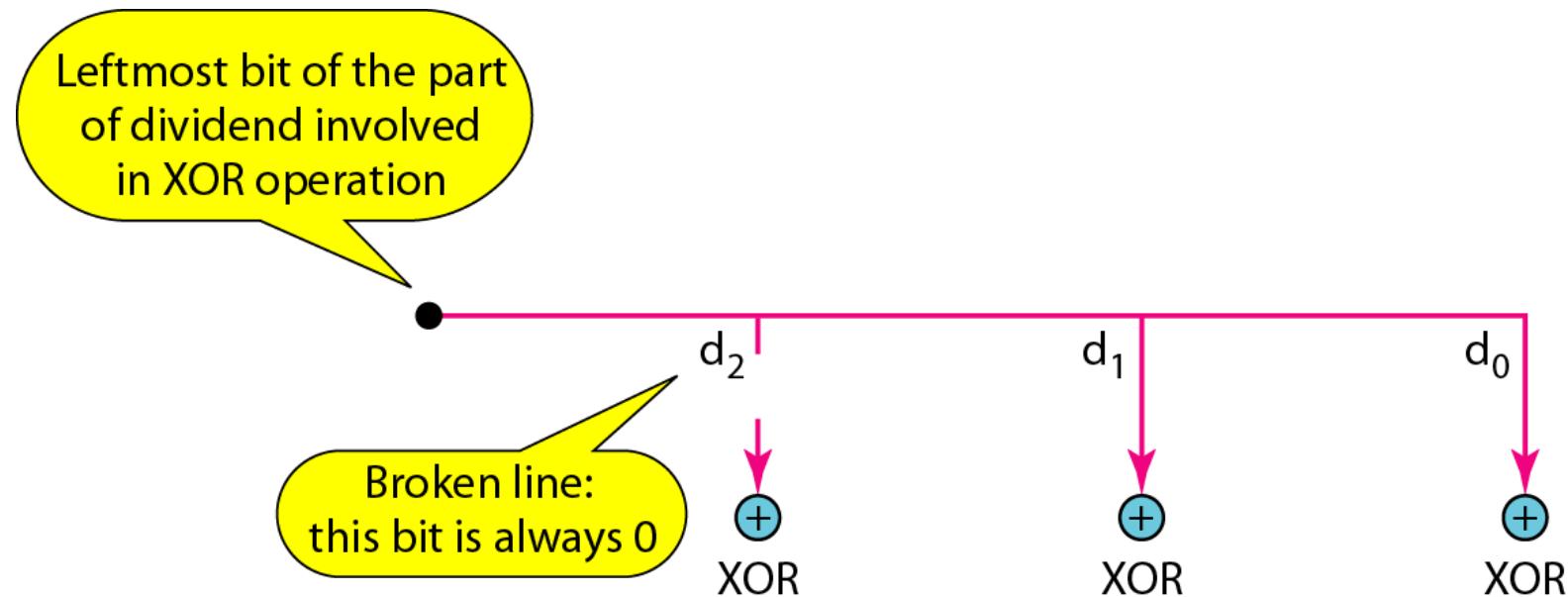
Received codeword/s: 1001110, 1000110

Calculate the Syndrome values

**Figure 10.16** Division in the CRC decoder for two cases



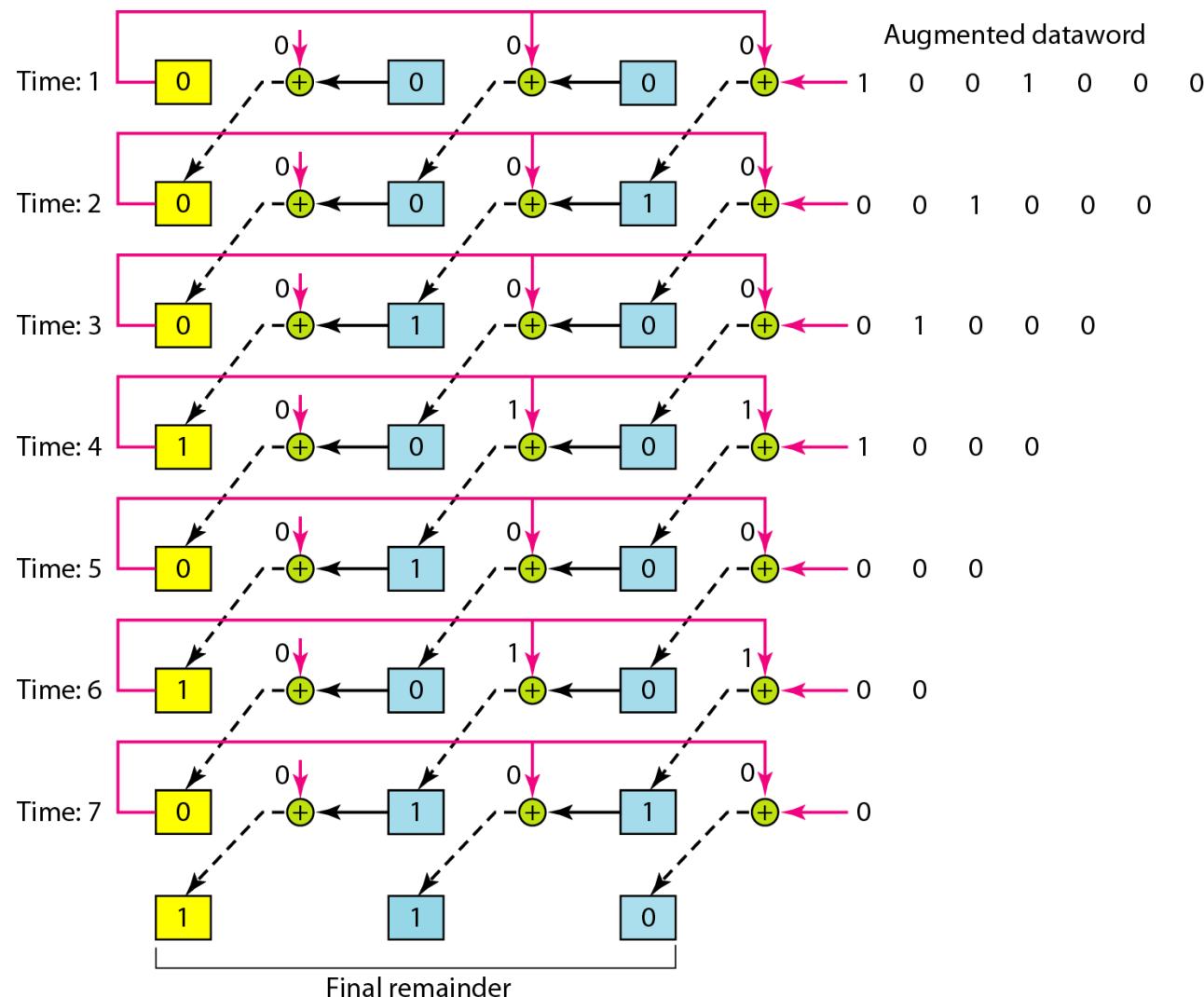
**Figure 10.17 Hardwired design of the divisor in CRC**



# Step-by-step for Division in H/W-S/W

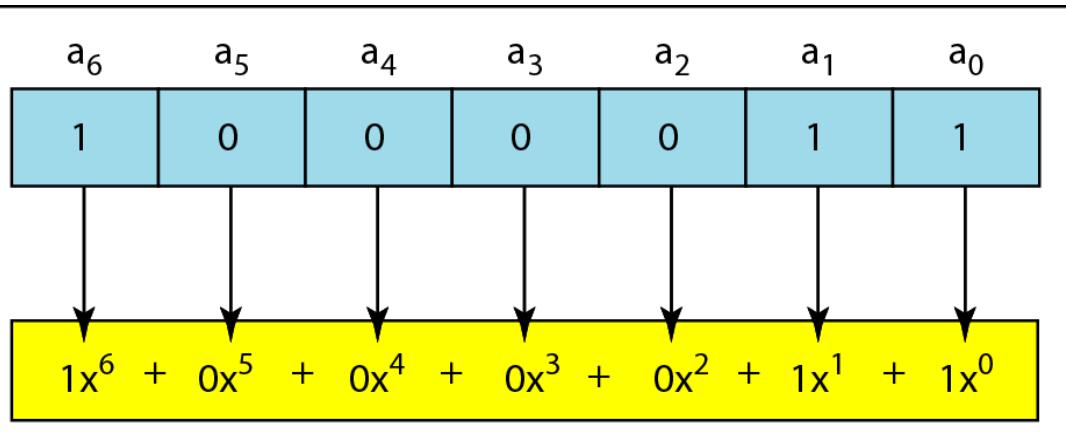
1. Assume the remainder is originally all 0s (000)
2. At each tick of the clock (arrival of 1 bit from augmented dataword), following two actions are repeated
  - a. The leftmost bit is used to make a decision about the divisor (011 or 000)
  - b. The other 2 bits of the remainder and the next bit from the augmented dataword (total of 3 bits) are XORed with the 3-bit divisor to create the next remainder

**Figure 10.18** Simulation of division in CRC encoder

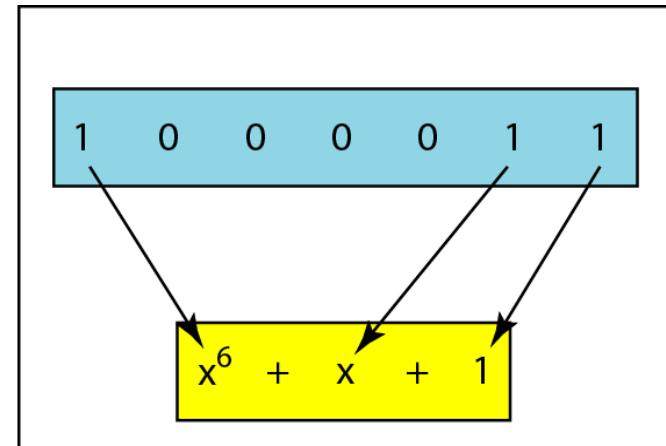


## Figure 10.21 A polynomial to represent a binary word

- 0s and 1s can be represented as a polynomial with coefficients 0 and 1
- Power of each term shows the position of the bit; coefficient shows the value of the bit
- Degree of a polynomial is the highest power in it

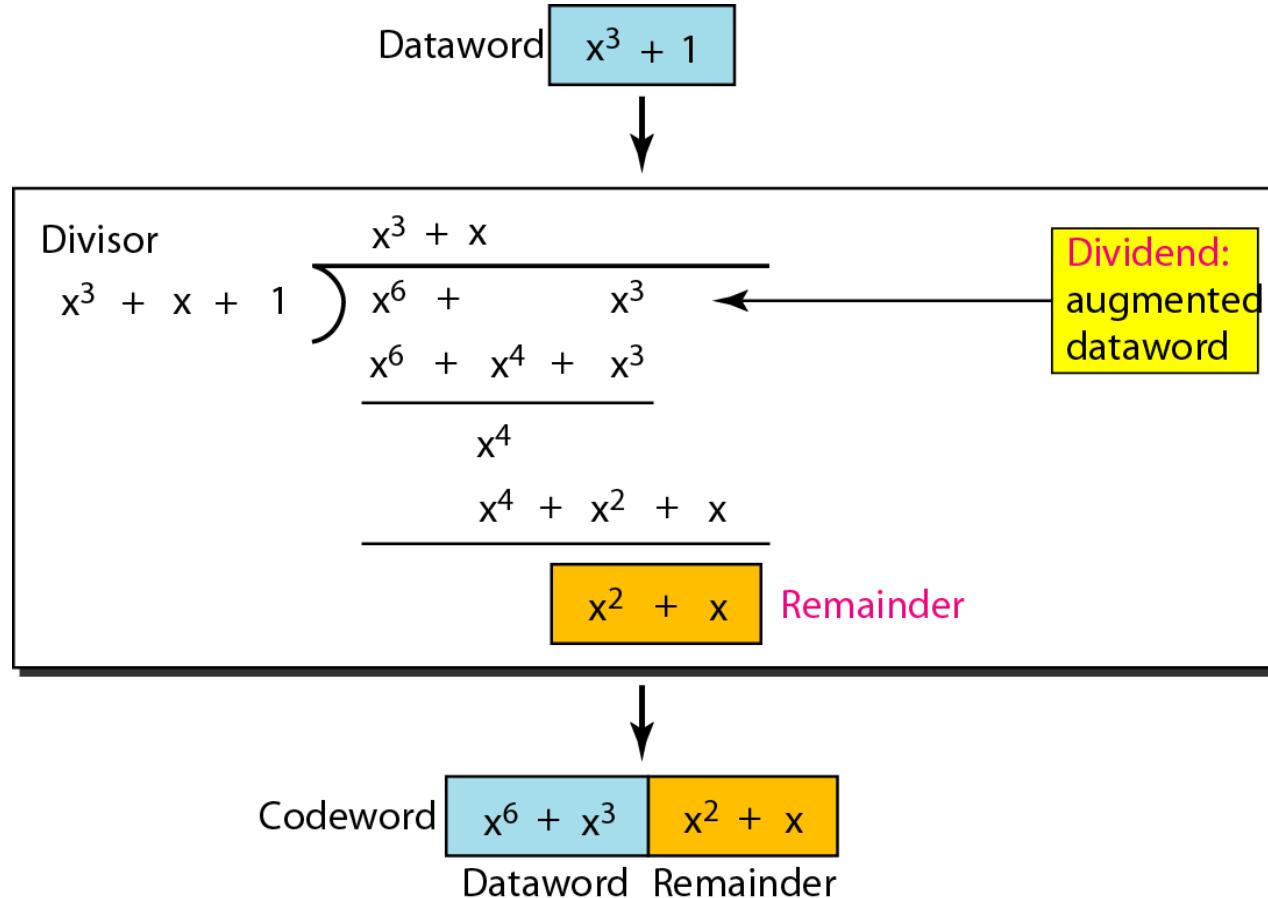


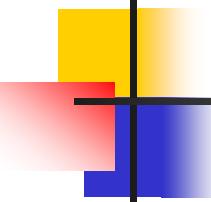
a. Binary pattern and polynomial



b. Short form

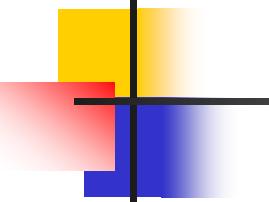
**Figure 10.22** CRC division using polynomials





## *Note*

**The divisor in a cyclic code is normally called the generator polynomial or simply the generator.**



## **Note**

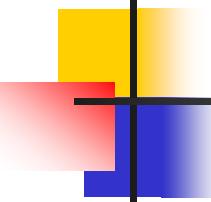
---

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.

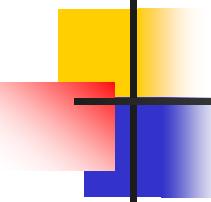


### *Note*

---

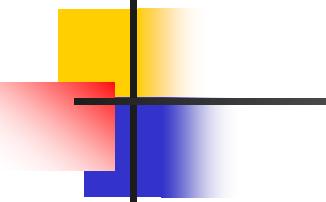
In a cyclic code, those errors [ $e(x)$ ] that are divisible by  $g(x)$  are not caught.

---



## *Note*

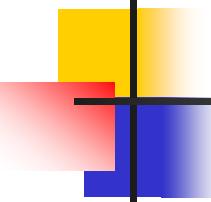
If the generator has more than one term  
and the coefficient of  $x^0$  is 1,  
all single bit errors can be caught.



## **Note**

A good polynomial generator needs to have the following characteristics:

- 1. It should have at least two terms.**
- 2. The coefficient of the term  $x^0$  should be 1.**
- 3. It should not divide  $x^t + 1$ , for  $t$  between 2 and  $n - 1$ .**
- 4. It should have the factor  $x + 1$ .**



## *Example 10.15*

*Which of the following  $g(x)$  values guarantees that a single-bit error is caught? For each case, what is the error that cannot be caught?*

- a.*  $x + 1$
- b.*  $x^3$
- c.* 1

### *Solution*

*a. No  $x^i$  is divisible by  $x + 1$*

*→ Any single-bit error can be caught*

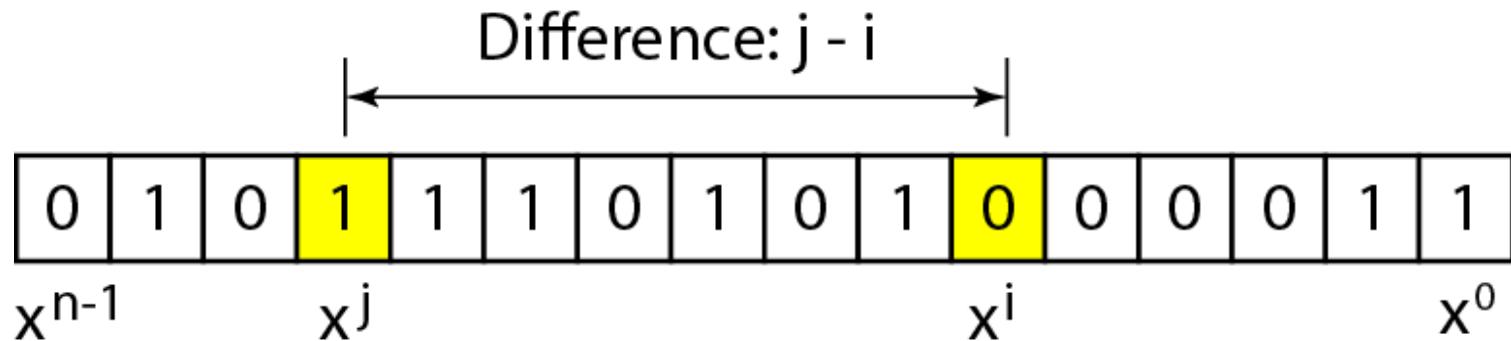
*b. If  $i \geq 3$ ,  $x^i$  is divisible by  $g(x)$*

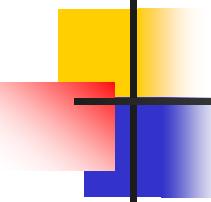
*→ All single-bit errors in positions 1 to 3 are caught*

*c. All values of  $i$  make  $x^i$  divisible by  $g(x)$*

*→ No single-bit error can be caught. This  $g(x)$  is useless*

**Figure 10.23** *Representation of two isolated single-bit errors using polynomials*





## **Note**

**If a generator cannot divide  $x^t + 1$  (t between 0 and  $n - 1$ ), then all isolated double errors can be detected.**

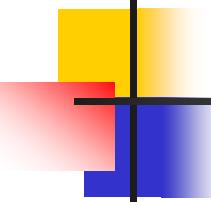
## *Example 10.16*

*Find the status of the following generators related to two isolated, single-bit errors.*

- a.**  $x + 1$     **b.**  $x^4 + 1$     **c.**  $x^7 + x^6 + 1$     **d.**  $x^{15} + x^{14} + 1$

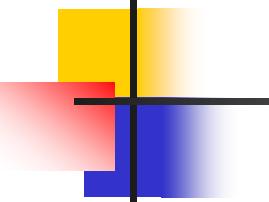
### *Solution*

- a.** *This is a very poor choice for a generator. Any two errors next to each other cannot be detected.*
- b.** *This generator cannot detect two errors that are four positions apart.*
- c.** *This is a good choice for this purpose.*
- d.** *This polynomial cannot divide  $x^t + 1$  if  $t$  is less than 32,768. A codeword with two isolated errors up to 32,768 bits apart can be detected by this generator.*



## *Note*

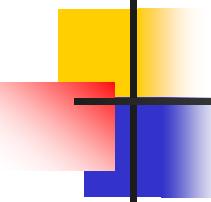
**A generator that contains a factor of  $x + 1$  can detect all odd-numbered errors.**



## **Note**

---

- All burst errors with  $L \leq r$  will be detected.
  - All burst errors with  $L = r + 1$  will be detected with probability  $1 - (1/2)^{r-1}$ .
  - All burst errors with  $L > r + 1$  will be detected with probability  $1 - (1/2)^r$ .
-



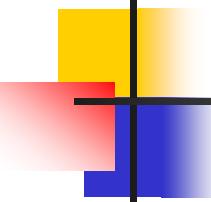
## *Example 10.17*

*Find the suitability of the following generators in relation to burst errors of different lengths.*

*a.*  $x^6 + 1$       *b.*  $x^{18} + x^7 + x + 1$       *c.*  $x^{32} + x^{23} + x^7 + 1$

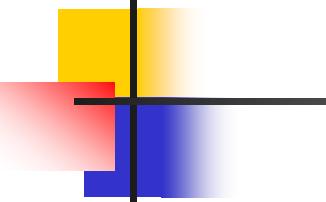
### *Solution*

*a. This generator can detect all burst errors with a length less than or equal to 6 bits;  
3 out of 100 burst errors with length 7 will slip by;  
16 out of 1000 burst errors of length 8 or more will slip by*



## *Example 10.17 (continued)*

- b. This generator can detect all burst errors with a length less than or equal to 18 bits; 8 out of 1 million burst errors with length 19 will slip by; 4 out of 1 million burst errors of length 20 or more will slip by.*
  
- c. This generator can detect all burst errors with a length less than or equal to 32 bits; 5 out of 10 billion burst errors with length 33 will slip by; 3 out of 10 billion burst errors of length 34 or more will slip by.*



## **Note**

A good polynomial generator needs to have the following characteristics:

- 1. It should have at least two terms.**
- 2. The coefficient of the term  $x^0$  should be 1.**
- 3. It should not divide  $x^t + 1$ , for  $t$  between 2 and  $n - 1$ .**
- 4. It should have the factor  $x + 1$ .**

**Table 10.7** *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

## 10-5 CHECKSUM

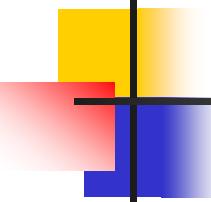
*The last error detection method we discuss here is called the checksum. The checksum is used in the Internet by several protocols although not at the data link layer. However, we briefly discuss it here to complete our discussion on error checking*

### **Topics discussed in this section:**

Idea

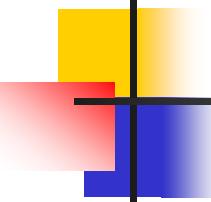
One's Complement

Internet Checksum



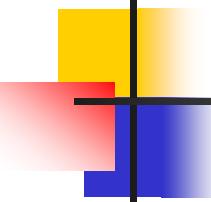
## *Example 10.18*

- 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 (1, 2, 3, 4, 5), we send (1, 2, 3, 4, 5 and 15), where 15 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.



## *Example 10.19*

- *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.*
- *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 10.20*

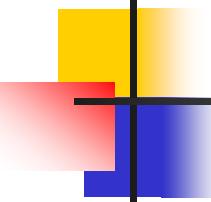
*How can we represent the number 21 in one's complement arithmetic using only four bits?*

### *Solution*

*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 10.21*

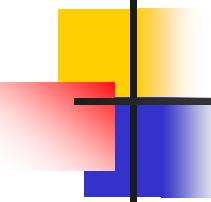
*How can we represent the number -6 in one's complement arithmetic using only four bits?*

### *Solution*

*In one's complement arithmetic, the negative or complement of a number is found by inverting all bits. Positive 6 is 0110; negative 6 is 1001.*

*If we consider only unsigned numbers, this is 9. In other words, the complement of 6 is 9.*

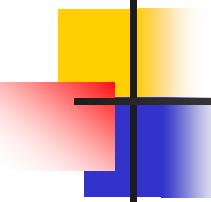
*Another way to find the complement of a number in one's complement arithmetic is to subtract the number from  $2^n - 1$  (i.e  $15 - 6$  in this case).*



## *Example 10.22*

*Let us redo Exercise 10.19 using one's complement arithmetic. Figure 10.24 shows the process at the sender and at the receiver.*

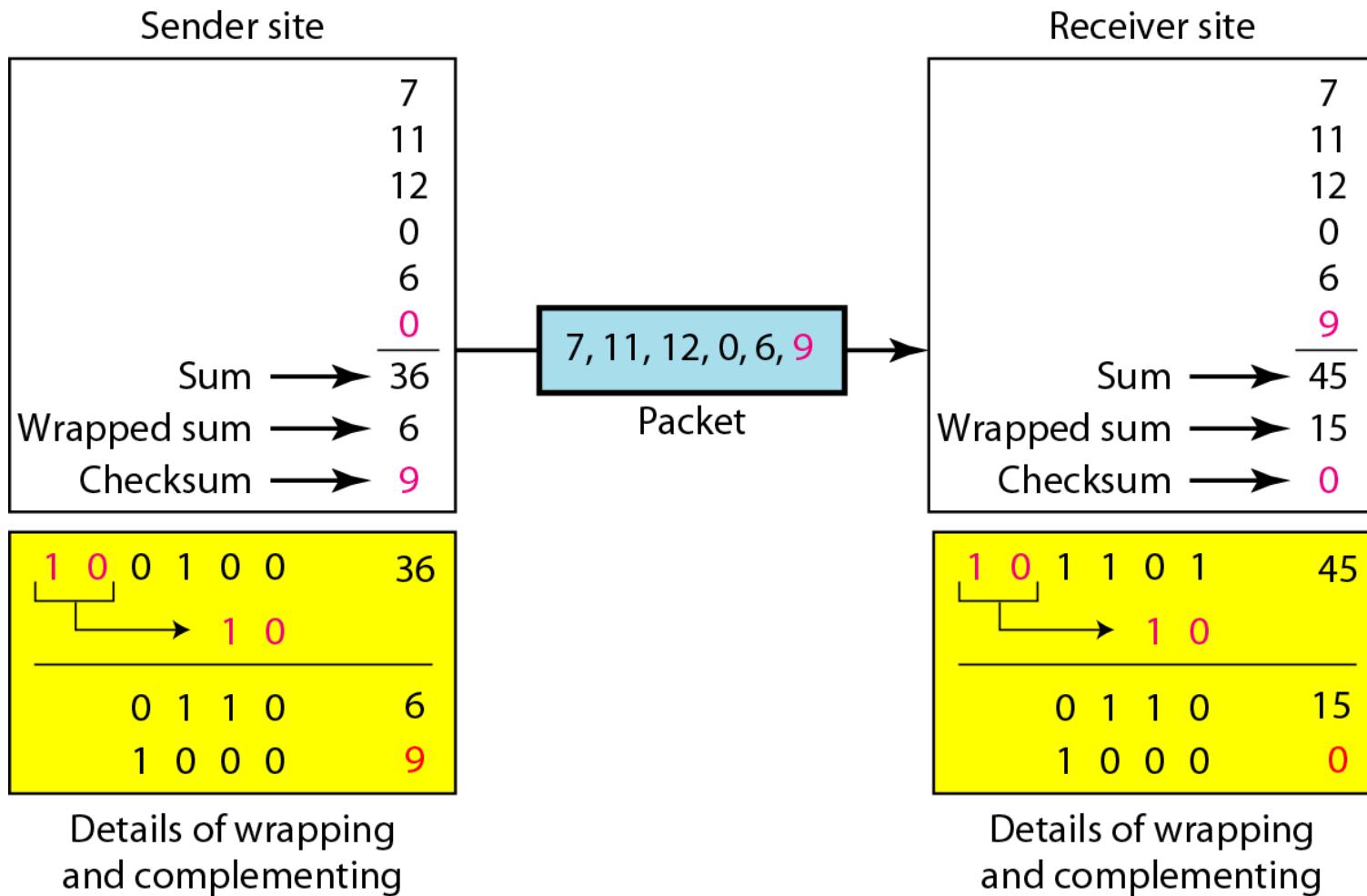
- *Sender initializes the checksum to 0 and adds all data items and the checksum (the checksum is considered as one data item and is shown in color) → The result is 36*
- *However, 36 cannot be expressed in 4 bits. The extra two bits are wrapped and added with the sum to create the wrapped sum value 6*
- *The sum is then complemented, resulting in the checksum value 9 ( $15 - 6 = 9$ )*
- *The sender now sends six data items to the receiver including the checksum 9.*



## *Example 10.22 (continued)*

- *The receiver follows the same procedure as the sender.*
- *It adds all data items (including the checksum); the result is 45. The sum is wrapped and becomes 15.*
- *The wrapped sum is complemented and becomes 0.*
- *Since the value of the checksum is 0, this means that the data is not corrupted. The receiver drops the checksum and keeps the other data items.*
- *If the checksum is not zero, the entire packet is dropped.*

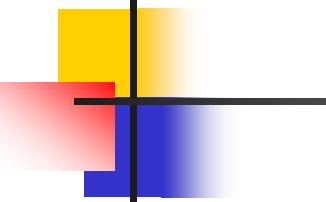
**Figure 10.24 Example 10.22**



**Note**

### 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.**



## **Note**

### **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 10.23*

*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 (see Appendix A) to change each byte to a 2-digit hexadecimal number. For example, F is represented as 0x46 and o is represented as 0x6F. Figure 10.25 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 0x36. 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. Note that if there is any corruption, the checksum recalculated by the receiver is not all 0s. We leave this an exercise.*

## Internet Checksum Example

---

- Checksum for a text of 8 characters “**Forouzan**”
  - ASCII Hex: **F- 46, o-6F, r-72, u-75, z-7A, a-61, n-6E**
  - Arrange in 16-bit word as:
    - Fo: 46 6F
    - ro: 72 6F
    - uz: 75 7A
    - an: 61 6E
  - Initialize checksum as 00 00
  - Perform the sum
  - Wrap to 16 bits
  - Complement
-

**Figure 10.25 Example 10.23**

1	0	1	3	Carries
4	6	6	F	(Fo)
7	2	6	F	(ro)
7	5	7	A	(uz)
6	1	6	E	(an)
0	0	0	0	Checksum (initial)
<hr/>				
8	F	C	6	Sum (partial)
<hr/>				
8	F	C	7	Sum
7	0	3	8	Checksum (to send)

a. Checksum at the sender site

1	0	1	3	Carries
4	6	6	F	(Fo)
7	2	6	7	(ro)
7	5	7	A	(uz)
6	1	6	E	(an)
7	0	3	8	Checksum (received)
<hr/>				
F	F	F	E	Sum (partial)
<hr/>				
F	F	F	F	Sum
0	0	0	0	Checksum (new)

a. Checksum at the receiver site

## Internet Checksum Example

---

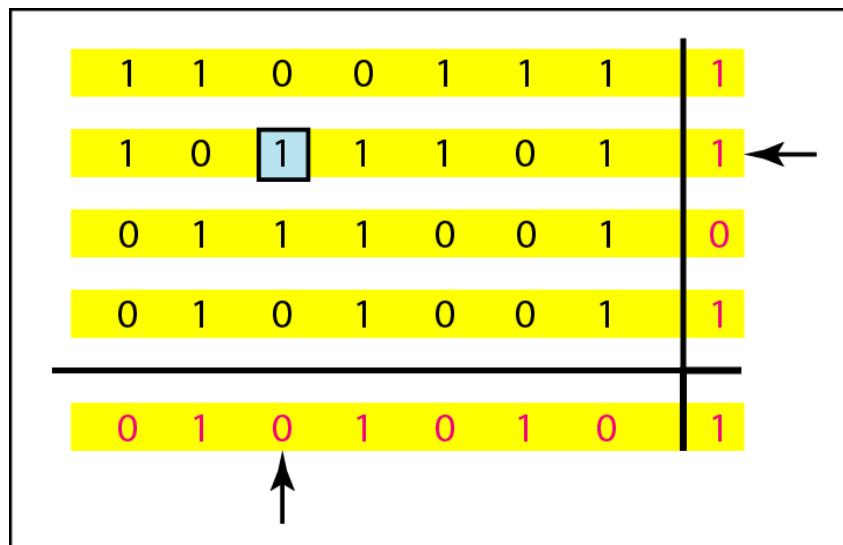
- Checksum for a text “**KJSomaiya**”
- ASCII Hex: **K- 4B, J-4A, S-53, o-6F, m-6D, a-61, i-69, y-79**
- Arrange in 16-bit word as:
- Initialize checksum as 00 00
- Perform the sum
- Wrap to 16 bits
- Complement

**Figure 10.11** Two-dimensional parity-check code

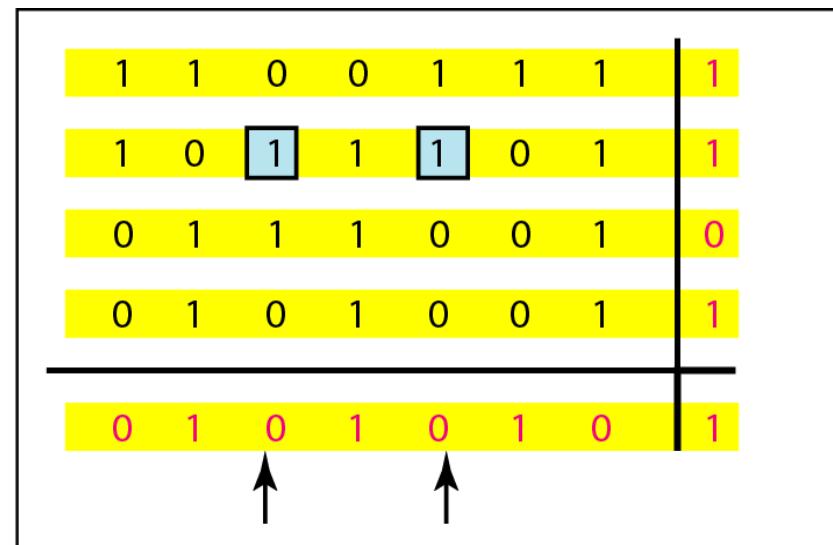
1	1	0	0	1	1	1	1	1
1	0	1	1	1	1	0	1	1
0	1	1	1	0	0	1	0	0
0	1	0	1	0	0	1	1	1
0	1	0	1	0	1	0	1	1

a. Design of row and column parities

**Figure 10.11 Two-dimensional parity-check code**

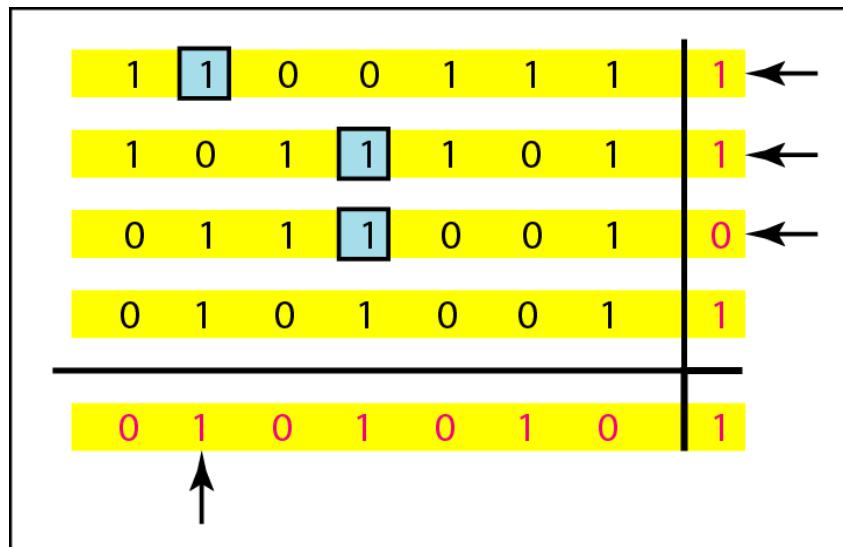


b. One error affects two parities

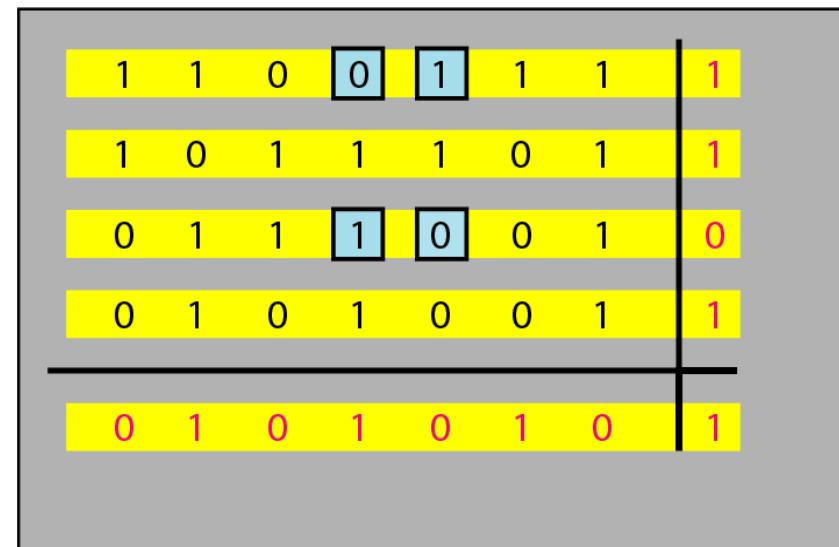


c. Two errors affect two parities

**Figure 10.11 Two-dimensional parity-check code**



d. Three errors affect four parities



e. Four errors cannot be detected