

Data communication and networking  
fourth Edition by  
Behrouz A. Forouzan

## **Chapter 10**

# **Error Detection and Correction**



*Note*

**Data can be corrupted  
during transmission.**

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

# 10-1 INTRODUCTION

*Let us first discuss some issues related, directly or indirectly, to error detection and correction.*

## Topics discussed in this section:

Types of Errors

Redundancy

Detection Versus Correction

Forward Error Correction Versus Retransmission

Coding

Modular Arithmetic



---

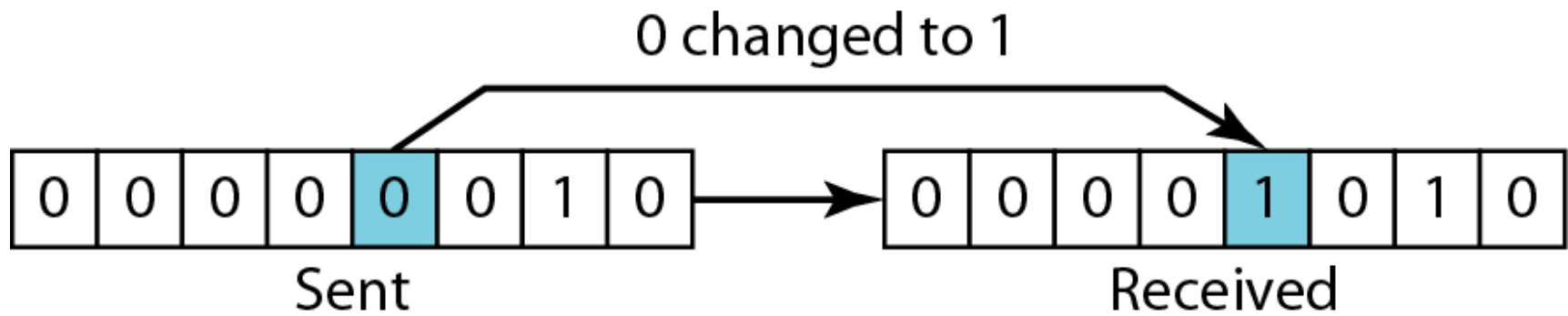
*Note*

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

---

**Figure 10.1** *Single-bit error*

---

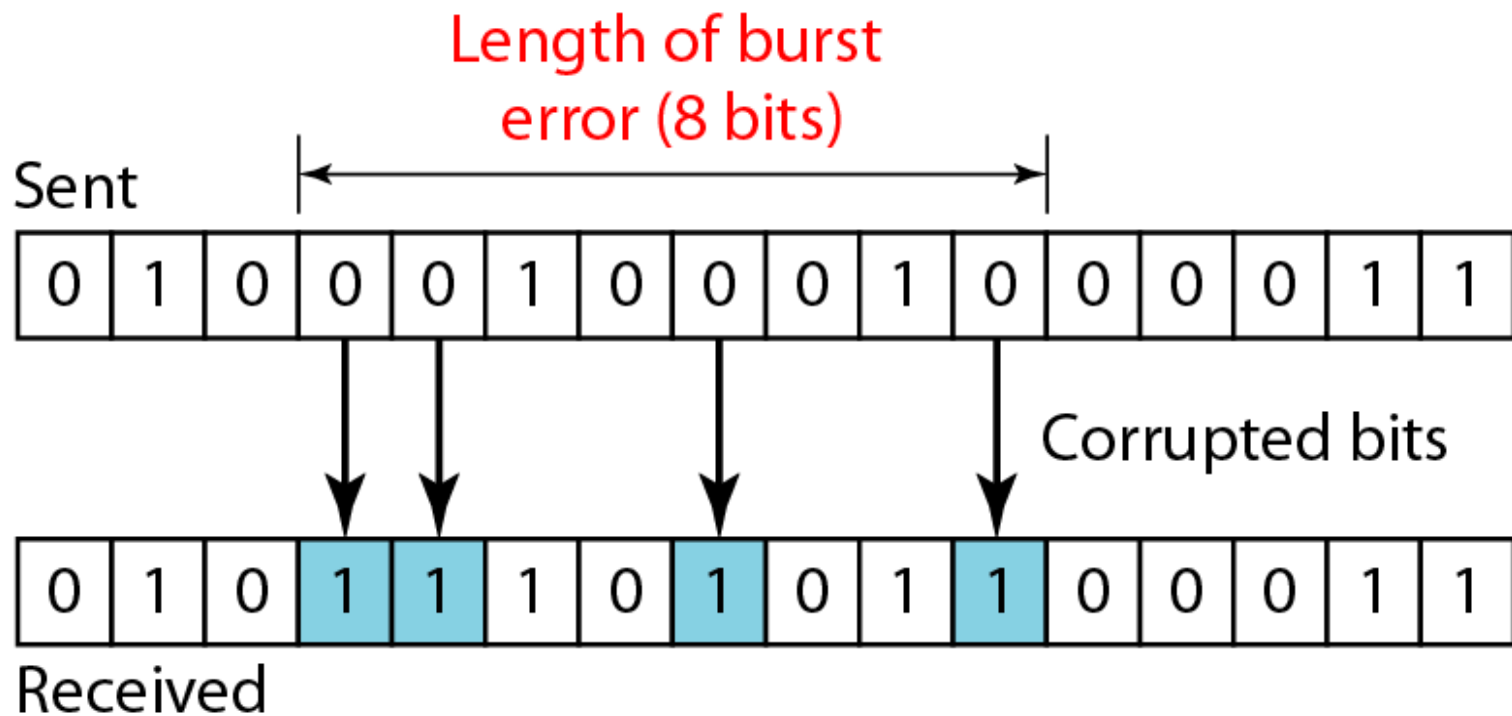




*Note*

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

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





**Redundancy** : is the central concept in detecting & correcting errors.

We need to send some extra bits with our data. These redundant bits are added by the sender and removed by the receiver .

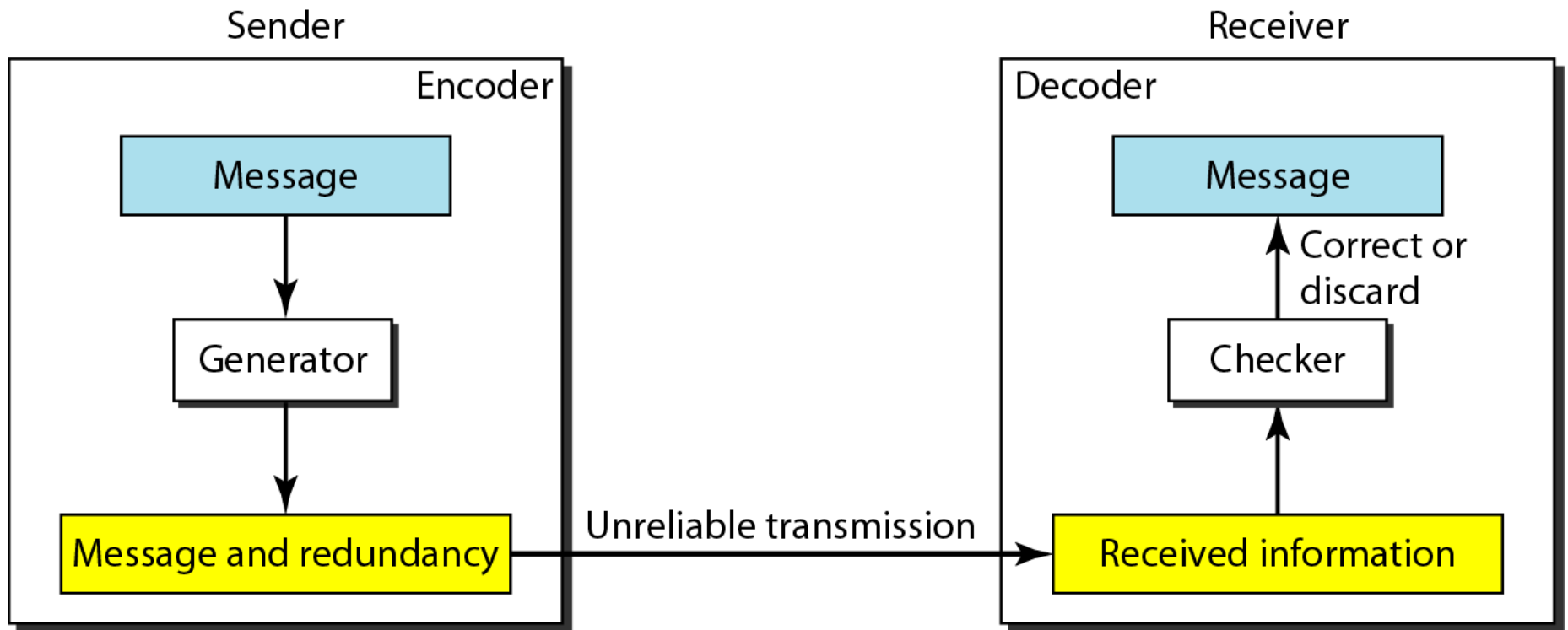
---

*Note*

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



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



# Detection Versus Correction

- In error **detection** , we are looking **only** to see **if** any error has occurred. A single-bit error is the same for us as a burst error.
- In error **correction** , we need to **know** the exact number of bits that are corrupted and more importantly, their location in the message. **So** the **number of errors** and the **size of the message** are important factors.
- **Note:** correction of errors is more difficult than the detection

# Forward Error Correction Versus Retransmission

Two main methods of error correction

- I. **Forward error correction** FEC: is the **process** in which the receiver tries to guess the message by using redundant bits.
- II. **Retransmission** : is a **technique** in which the receiver detects the occurrence of an error and asks the sender to resend the message.

**Note:** use FEC if the number of errors is small.



# 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

## **coding schemes is divided into two categories :**

1- block coding .

2- convolution coding. convolution coding is more complex than block coding.

### ***Note***

- In this section , we concentrate on block codes; we leave convolution codes to advanced texts.

# Modular Arithmetic

- In modular arithmetic, we use only a limited range of integers. We define an upper limit, called a **modulus  $N$** . We then use only the integers  *$0$  to  $N - 1$* .
- *For example*, if the modulus is 12, we use only the integers 0 to 11.
- In a **modulo- $N$  system**, if a number is greater than  $N$ , it is divided by  $N$  and the remainder is the result.

Addition and subtraction in modulo arithmetic are simple. There is no carry when you add two digits in a column. There is no carry when you subtract one digit from another in a column

## ***Modulo-2 Arithmetic***

Of particular interest is modulo-2 arithmetic. In this arithmetic, the modulus  $N$  is 2. We can use only 0 and 1. Operations in this arithmetic are very simple. The following shows how we can add or subtract 2 bits.

Adding:  $0+0=0$   $0+1=1$   $1+0=1$   $1+1=0$

Subtracting:  $0-0=0$   $0-1=1$   $1-0=1$   $1-1=0$

•use the XOR (exclusive OR) operation for both addition and subtraction.

**Note**

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

## 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.

	1	0	1	1	0
$\oplus$	1	1	1	0	0
<hr/>					
	0	1	0	1	0

c. Result of XORing two patterns

**Note** : If the modulus is not 2, addition and subtraction are distinct.



## 10-2 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**.

With **k** bits, we can create a combination of  **$2^k$**  datawords; with **n** bits, we can create a combination of  **$2^n$**  codewords.

The block coding process is one-to-one; the same dataword is always encoded as the same codeword. This means that we have  **$2^n - 2^k$  codewords that are *not used***.

## *Topics discussed in this section:*

- Error Detection
- Error Correction
- Hamming Distance
- Minimum Hamming Distance

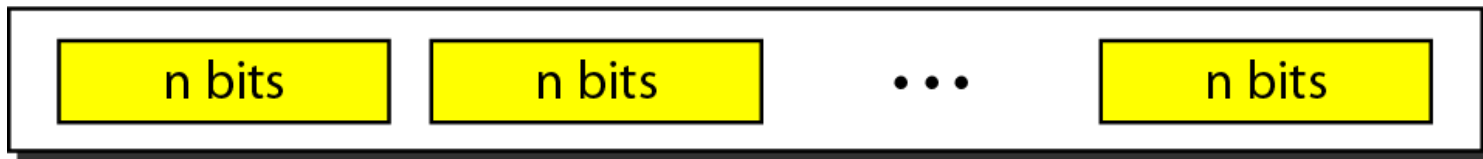
---

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

---



$2^k$  Datawords, each of  $k$  bits



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



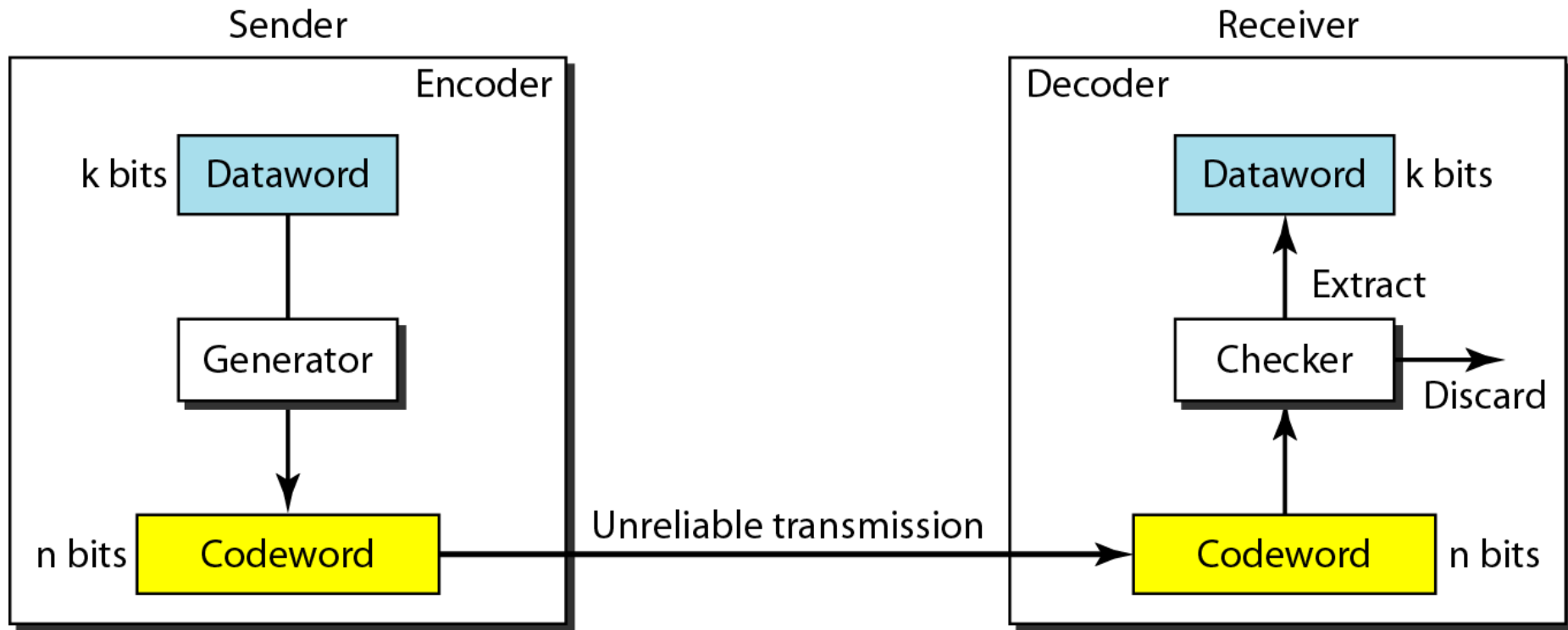
## *Example 10.1*

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

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

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



# Error Detection

How can errors be detected by using block coding?

If the following two conditions are met, the receiver can detect a change in the original codeword.

- 1. The receiver has (or can **find**) a list of valid codewords.
- 2. The original codeword has **changed** to an invalid one.

## Example 10.2

Let us assume that  $k = 2$  and  $n = 3$ . **Table 10.1** shows the list of datawords and codewords. Later, we will see how to **derive** a codeword from a dataword.

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.**

## *Example 10.2 (continued)*

- 2. The codeword is corrupted during transmission, and **111** is received. This is **not a valid** codeword and is **discarded** (don't exist in table).*
- 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)*

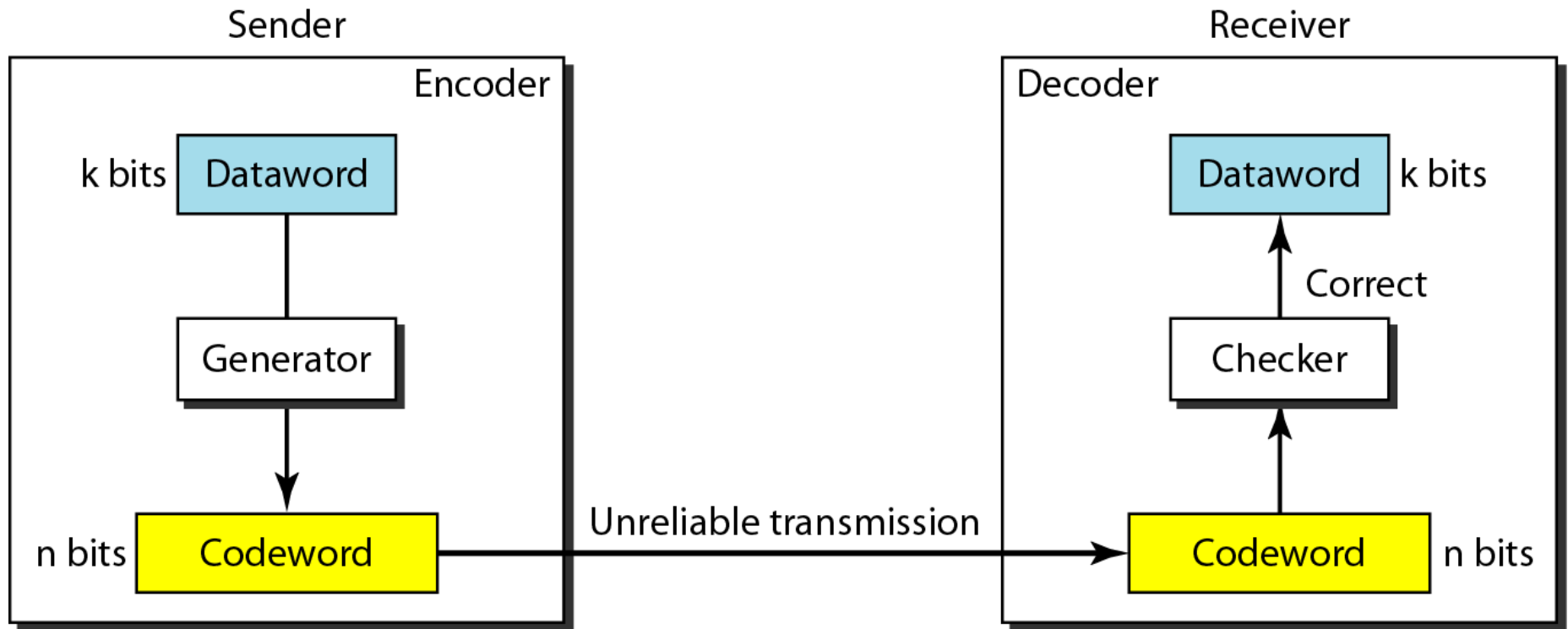
<i>Datawords</i>	<i>Codewords</i>
00	000
01	011
10	101
11	110



*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*



# Error Correction

As we said before, **error correction** is much more difficult than **error detection**. In error detection, the receiver needs to know only that the received codeword is **invalid**; in error correction the receiver needs to **find** (or – guess) the original codeword sent.

**Figure 10.7** shows the role of block coding in error correction. We can see that the idea is the same as error detection but the **checker functions** are much more complex.

## 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.

**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.**

## *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

One of the **central concepts** in coding for error control is the idea of the Hamming distance.

The Hamming distance can easily be found if we apply the XOR operation ( $\ominus$ ) on the two words and count the number of **1's** in the result. **Note** that the Hamming distance is a value **greater than zero**.

**Note**

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



## Example 10.4

*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*

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

## Minimum Hamming Distance

the measurement that is used for **designing** a code is the minimum Hamming distance.

We use  **$d_{\min}$**  to define the minimum Hamming distance in a coding scheme.

To find this value, we find the Hamming distances between **all** words and **select** the **smallest** one.

*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**.

## Example 10.6

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

### 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.

## ■ Three Parameters

Before we continue with our discussion, we need to mention that any coding scheme needs to have at least three parameters: the **codeword size  $n$** , the **dataword size  $k$** , and the **minimum Hamming distance  $d_{\min}$** .

A coding scheme  $C$  is written as  $C(n, k)$  with a separate expression for  **$d_{\min}$** .

**For example**, we can call our first coding scheme  $C(3, 2)$  with  $d_{\min} = 2$  and our second coding scheme  $C(5, 2)$  with  $d_{\min} = 3$ .

## Hamming Distance and Error

let us discuss the relationship between the Hamming distance and errors occurring during transmission.

When a codeword is corrupted during transmission, the Hamming distance between the sent and received codewords is the number of bits **affected** by the error.

**In other words**, the Hamming distance between the **received codeword** and the **sent codeword** is the **number of bits** that are corrupted during transmission.

**For example**, if the codeword 00000 is sent and 01101 is received, 3 bits are in error and the Hamming distance between the two is  $d(00000, 01101) = 3$ .



*Note*

To guarantee the detection of up to  $s$  errors in all cases, the minimum Hamming distance in a blockcode 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. 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.*

## 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 (**addition modulo-2**) 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 code words **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.*

# Minimum Distance for Linear Block Codes

It is simple to find the minimum Hamming distance for a linear block code. The minimum Hamming distance is the number of **1s** in the nonzero valid codeword with the smallest number of **1s**.

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

# Types of linear Block Codes

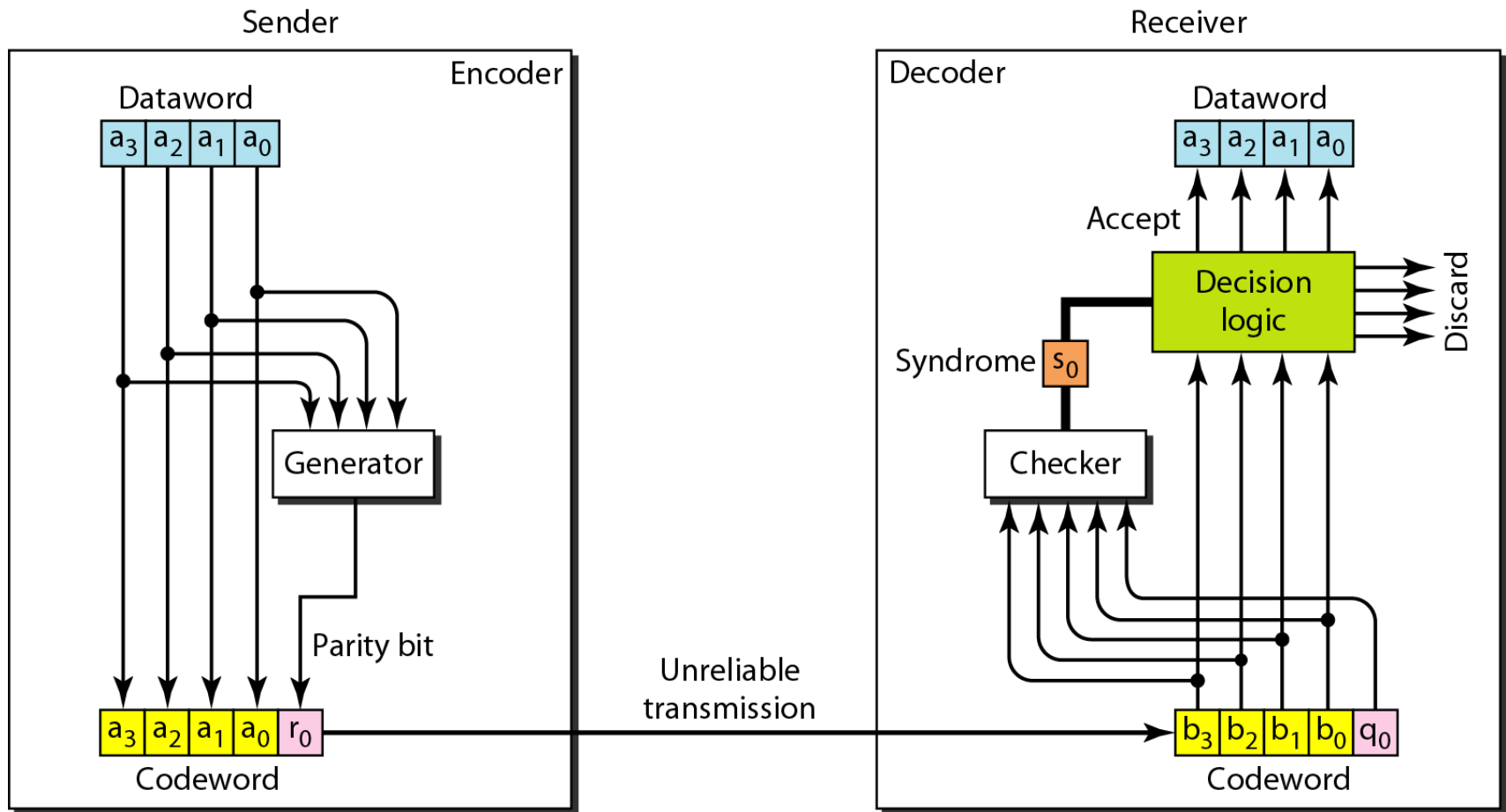
**1-Simple Parity-Check Code:** the most familiar error-detecting code is the simple parity-check code. In this code, a **k-bit** dataword is **changed** to an **n-bit** codeword where  $n = k + 1$ . The **extra bit**, called the **parity bit**, is selected to make the total number of 1s in the codeword **even**.

**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  $a_1$ . The received codeword is 10**0**11. The syndrome is 1. **No** dataword is created.*
- 3. One **single-bit error** changes  $r_0$ . The received codeword is 1011**0**. The syndrome is 1. **No** dataword is created.*

## Example 10.12 (continued)

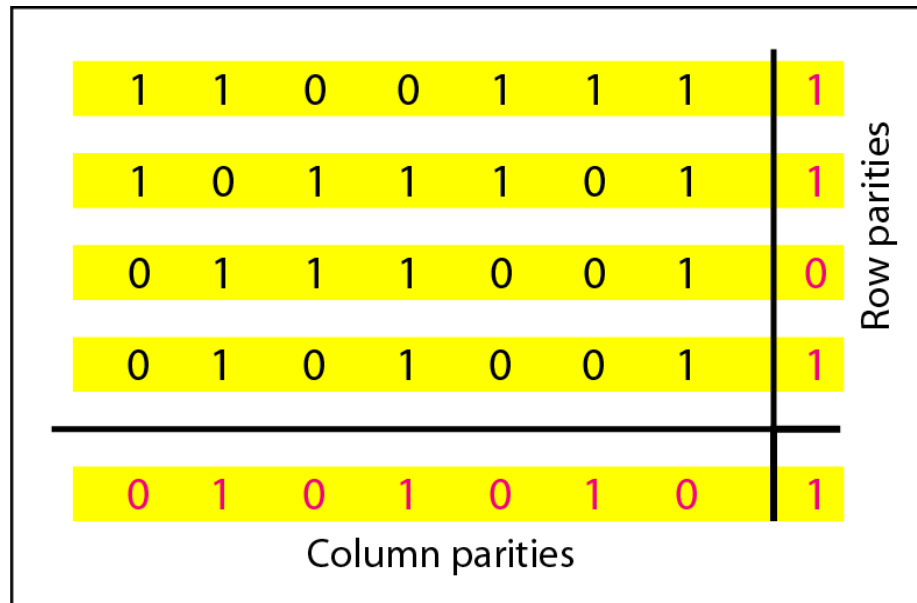
4. An error changes  $r_0$  and a second error changes  $a_3$ .  
The received codeword is **00110**. The syndrome is 0.  
The dataword **0011** is created **at the receiver**. **Note** that here the dataword is wrongly created due to the syndrome value.
  5. Three bits— $a_3$ ,  $a_2$ , and  $a_1$ —are changed by errors.  
The received codeword is (**01011**). The syndrome is 1.  
The dataword is **not created**.
- Note** : This shows that the simple parity check, guaranteed to detect **one single error**, can also find **any odd number of errors**.



*Note*

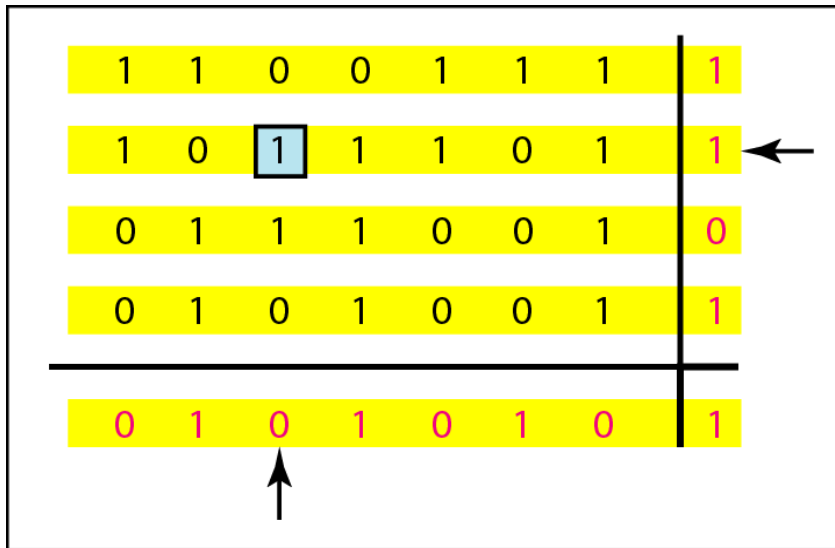
**A simple parity-check code can detect  
an odd number of errors.**

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

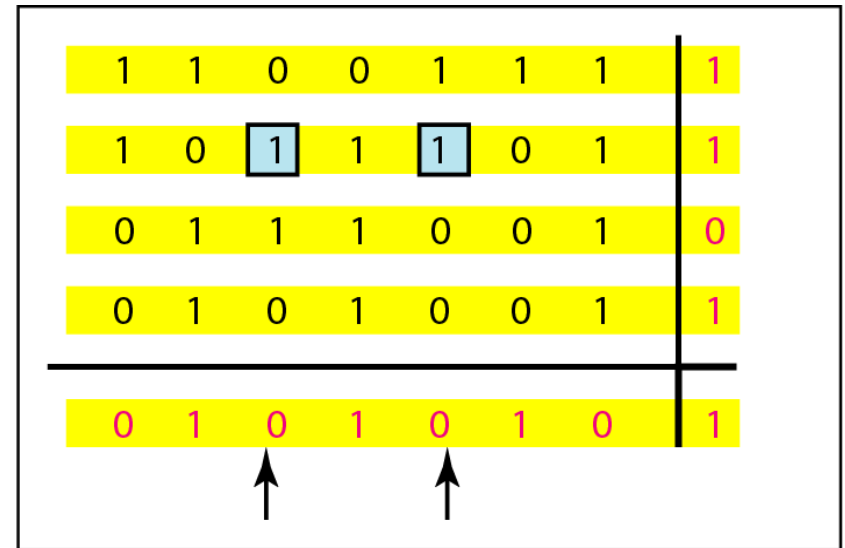


a. Design of row and column parities

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

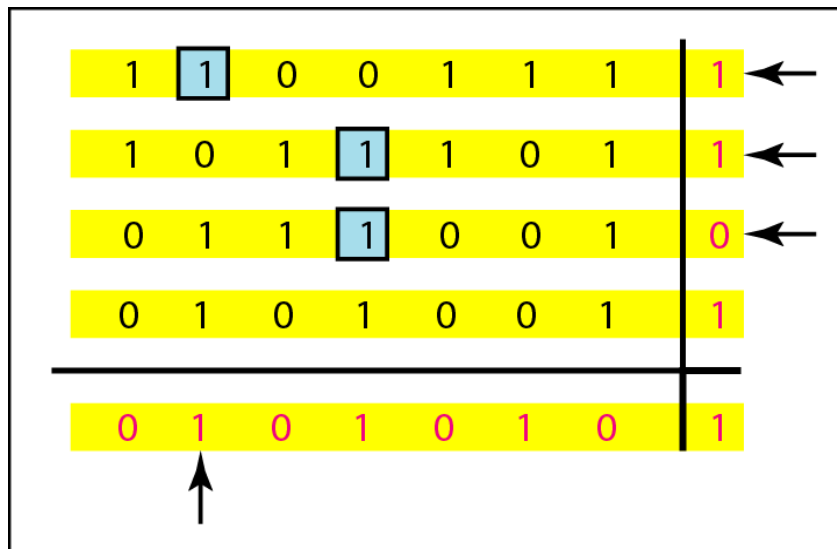


b. One error affects two parities

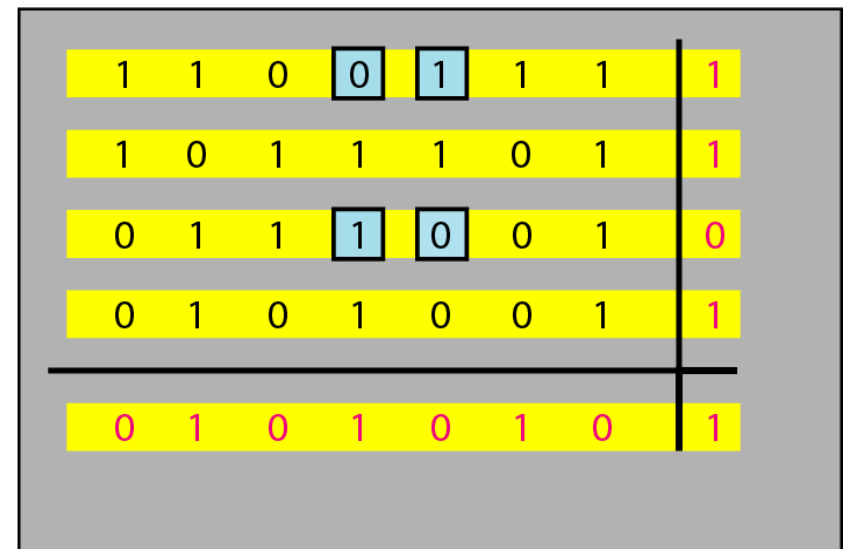


c. Two errors affect two parities

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



d. Three errors affect four parities



e. Four errors cannot be detected

**2- Hamming Codes:** are error-correcting codes.

These codes were originally designed with  $d_{\min} = 3$ , which means that they can detect up to two errors or correct one single error.

**Note:** some Hamming codes that can correct more than one error, our discussion focuses on the single-bit error-correcting code.

- First let us find the relationship between  $n$  and  $k$  in a Hamming code. We need to
- choose an integer  $m \geq 3$ . The values of  $n$  and  $k$  are then calculated from  $m$  as  $n = 2^m - 1$
- and  $k = n - m$ . The number of check bits  $r = m$ .

*Note*

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

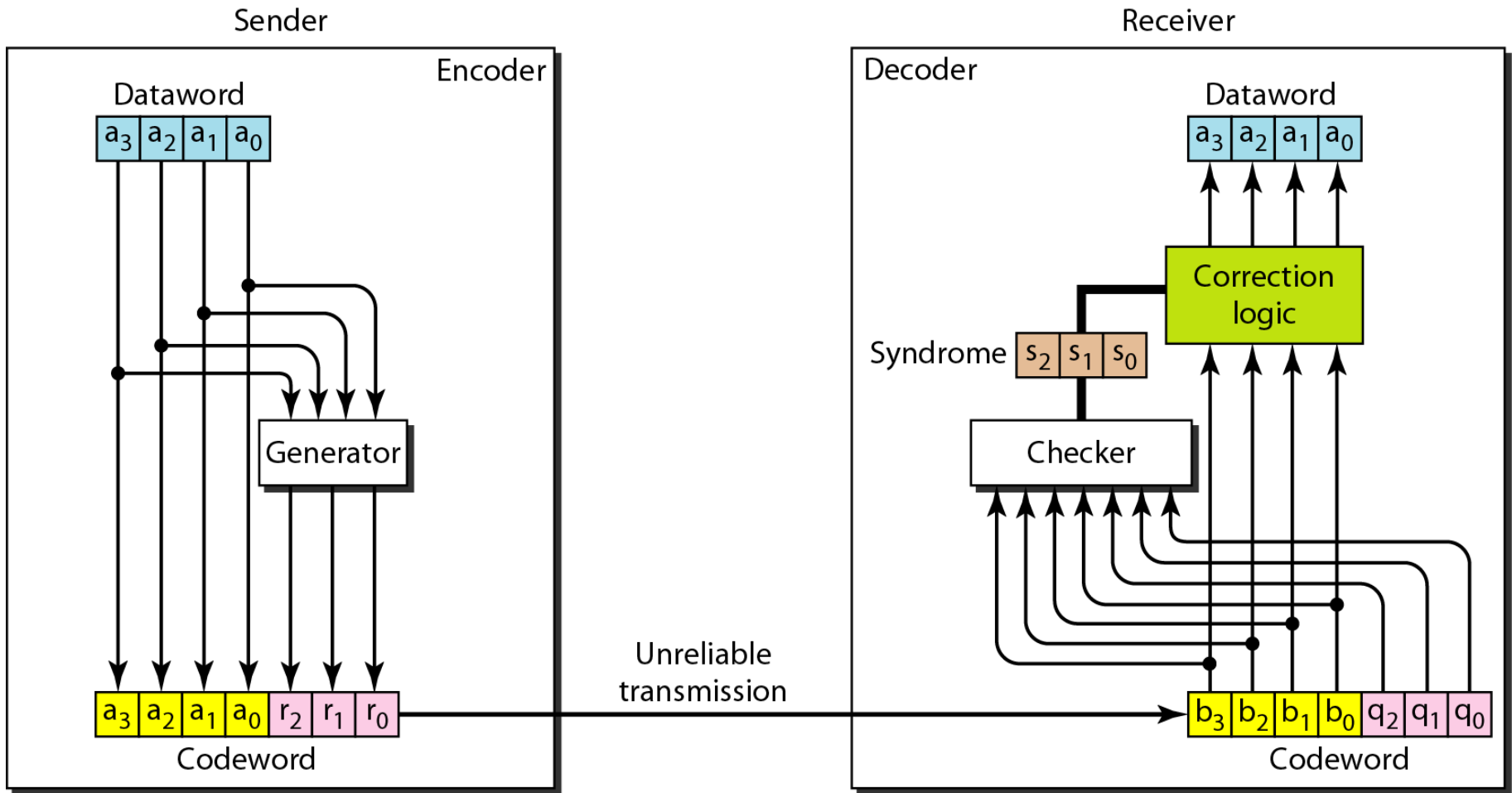
The relationship between  $m$  and  $n$  in these codes is  $n=2^m-1$ .



**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*



**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 a dataword of at least **7 bits**. Calculate values of  $k$  and  $n$  that satisfy this requirement.

### Solution

We need to make  $k = n - m$  greater than or equal to 7.

1. If we set  $m = 3$ , the result is  $n = 2^3 - 1 = 7$  and  $k = 7 - 3$ , or 4, which is not acceptable.

2. If we set  $m = 4$ , then  $n = 2^4 - 1 = 15$  and  $k = 15 - 4 = 11$ , which satisfies the condition. So the code is

$C(15, 11)$

## 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

# Cyclic Redundancy Check

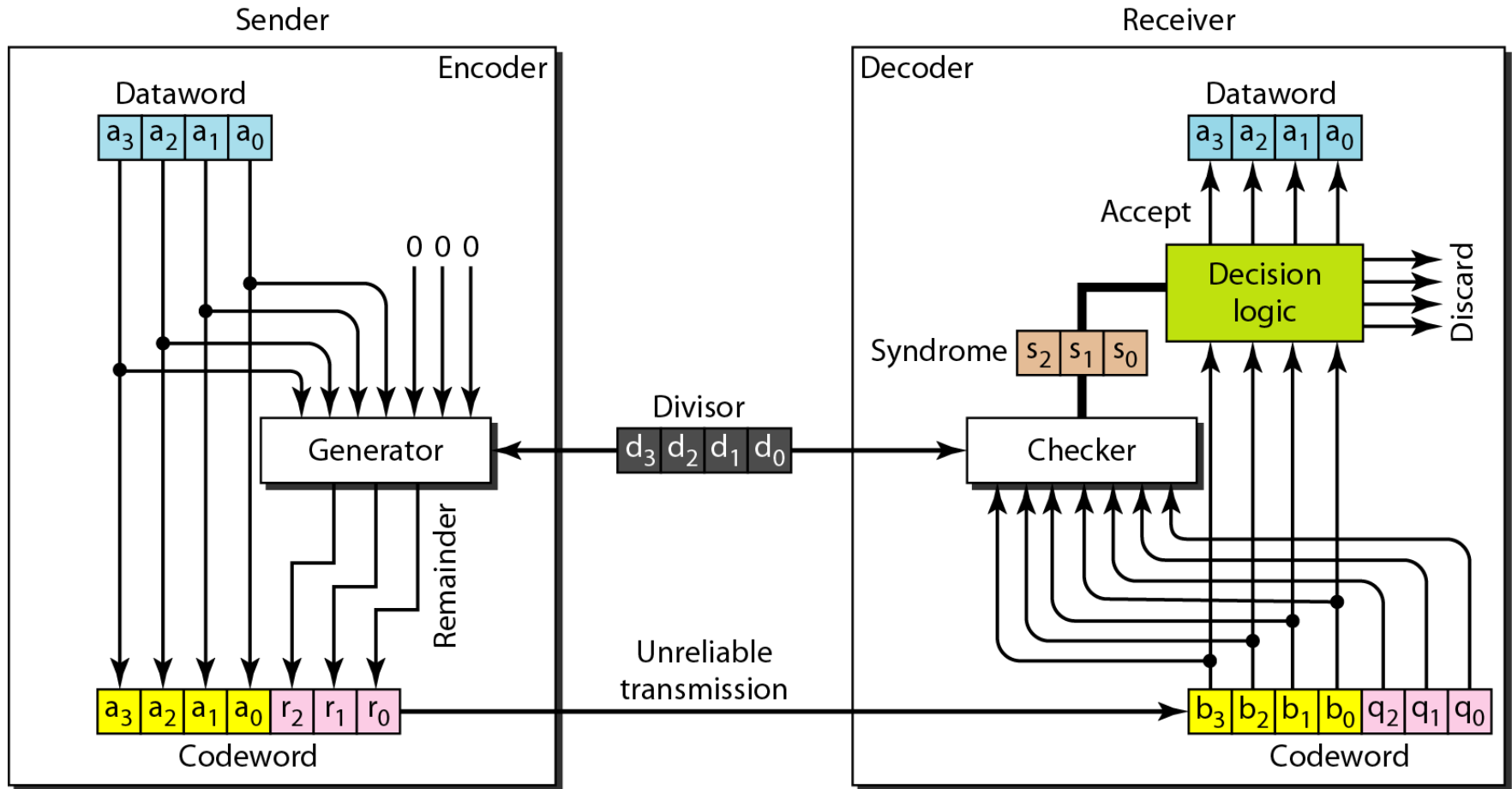
We can create cyclic codes to correct errors. In this section, we simply discuss a category of cyclic codes called the cyclic redundancy check (CRC) that is used in networks such as LANs and WANs.

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

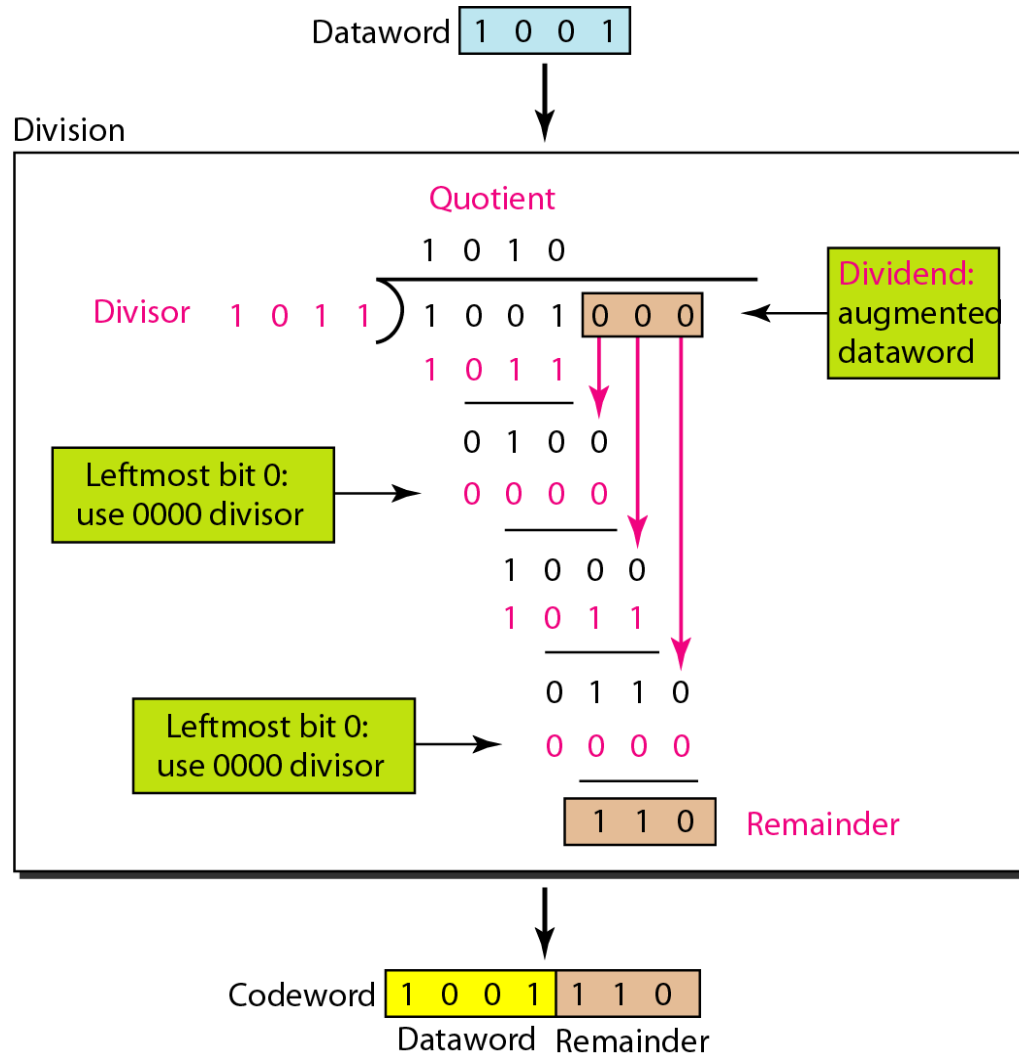
<i>Dataword</i>	<i>Codeword</i>	<i>Dataword</i>	<i>Codeword</i>
0000	0000000	1000	1000101
0001	0001011	1001	1001110
0010	0010110	1010	1010011
0011	0011101	1011	1011000
0100	0100111	1100	1100010
0101	0101100	1101	1101001
0110	0110001	1110	1110100
0111	0111010	1111	1111111



**Figure 10.14** *CRC encoder and decoder*



**Figure 10.15** *Division in CRC encoder*



# Polynomials

A better way to understand cyclic codes and how they can be analyzed is to represent them as polynomials.

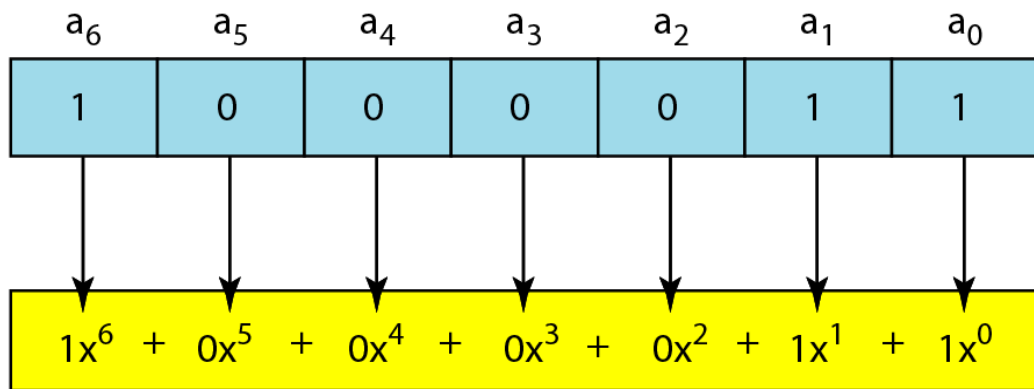
A pattern of **0s** and **1s** can be represented as a polynomial with coefficients of **0** and **1**. The **power** of each term shows the **position** of the bit; the **coefficient** shows the **value** of the bit.

Figure 10.21 shows a binary pattern and its polynomial representation.

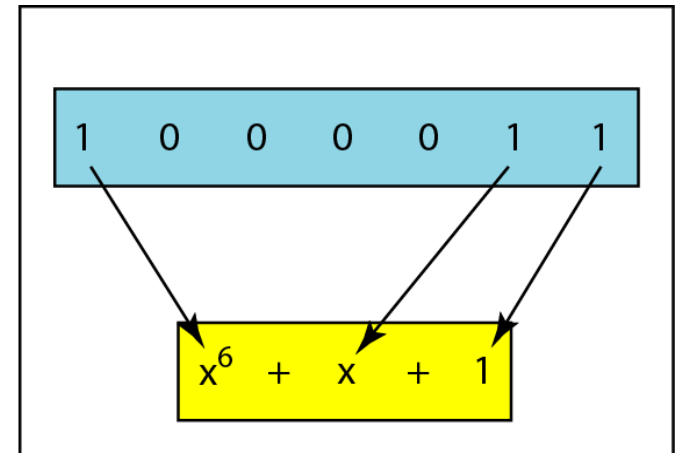
---

**Figure 10.21** *A polynomial to represent a binary word*

---



a. Binary pattern and polynomial



b. Short form

## 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 (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.*



## *Example 10.19*

*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(not Xor)** 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  $2^n - 1$  (**16** - 1 in this case).

**Why 16?**

## *Example 10.22*

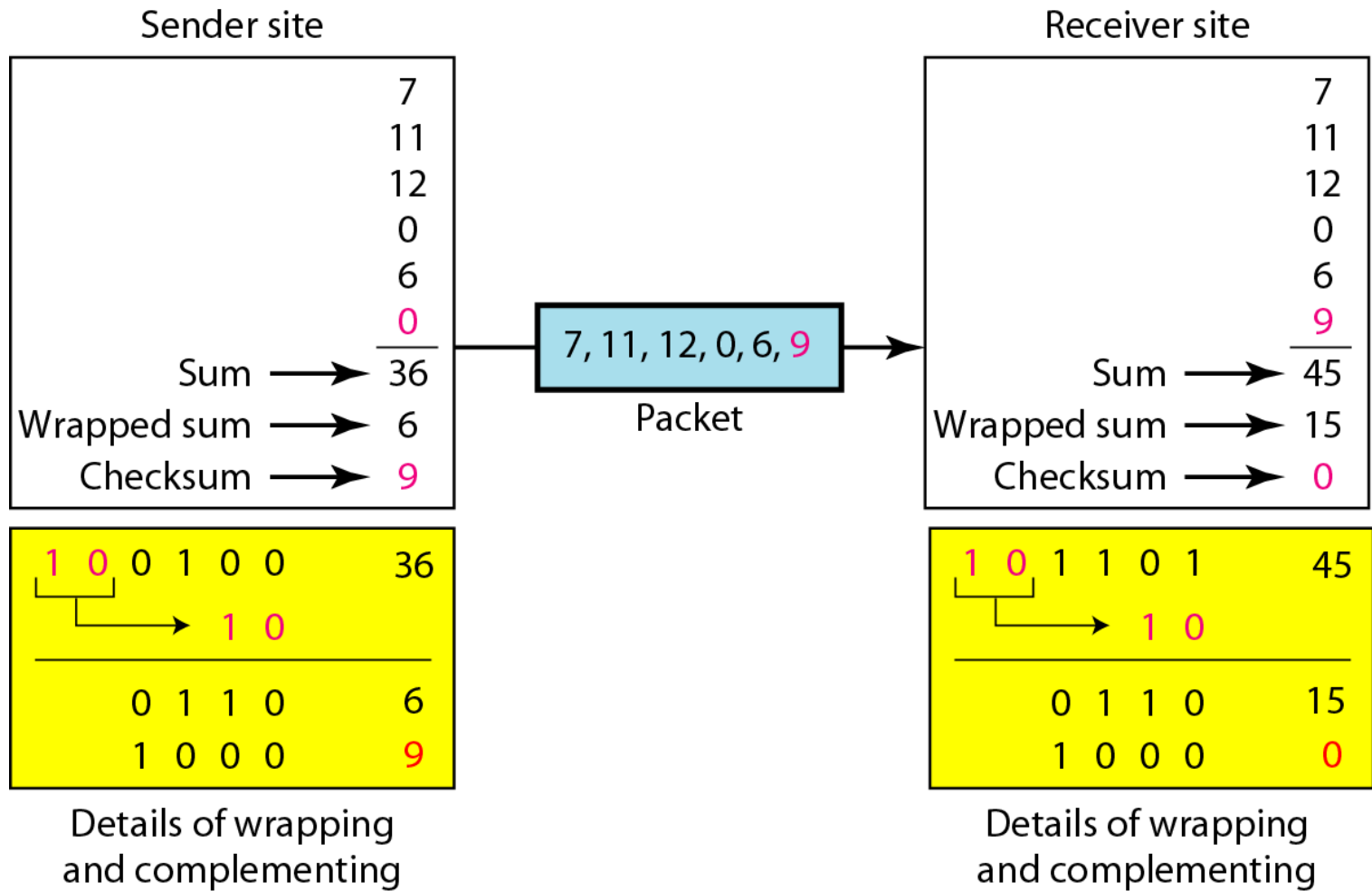
*Let us redo Exercise 10.19 using one's complement arithmetic. Figure 10.24 shows the process at the sender and at the receiver. The sender initializes the checksum to 0 and adds all data items and the checksum (the checksum is considered as one data item and is shown in color). The result is 36. However, 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. In the figure, we have shown the details in binary. 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.*

**Figure 10.25** *Example 10.23*

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)	
0	0	0	0		Checksum (initial)
8	F	C	6		Sum (partial)
			1		
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)
F	F	F	E		Sum (partial)
			1		
8	F	C	7		Sum
0	0	0	0		Checksum (new)

a. Checksum at the receiver site