Data Communications and Networking Fourth Edition

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.

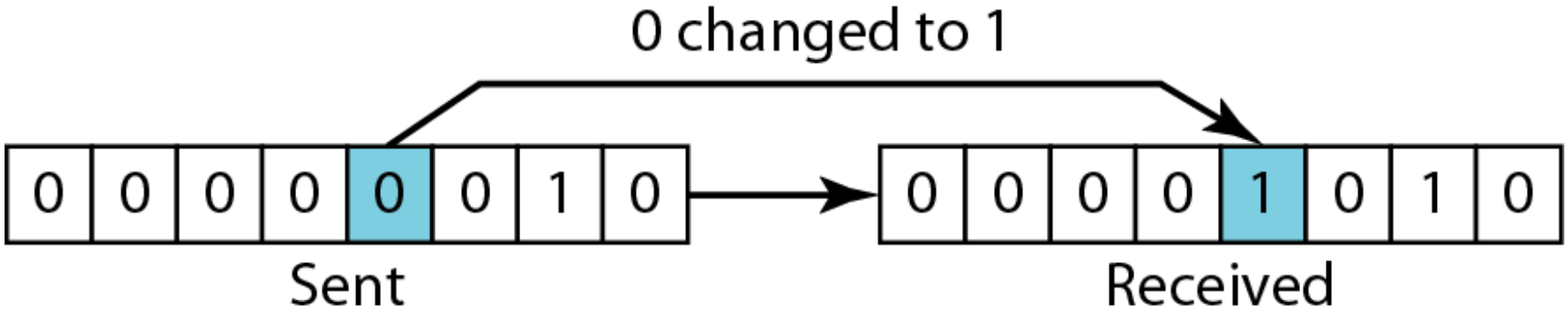
Topics discussed in this section:

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.

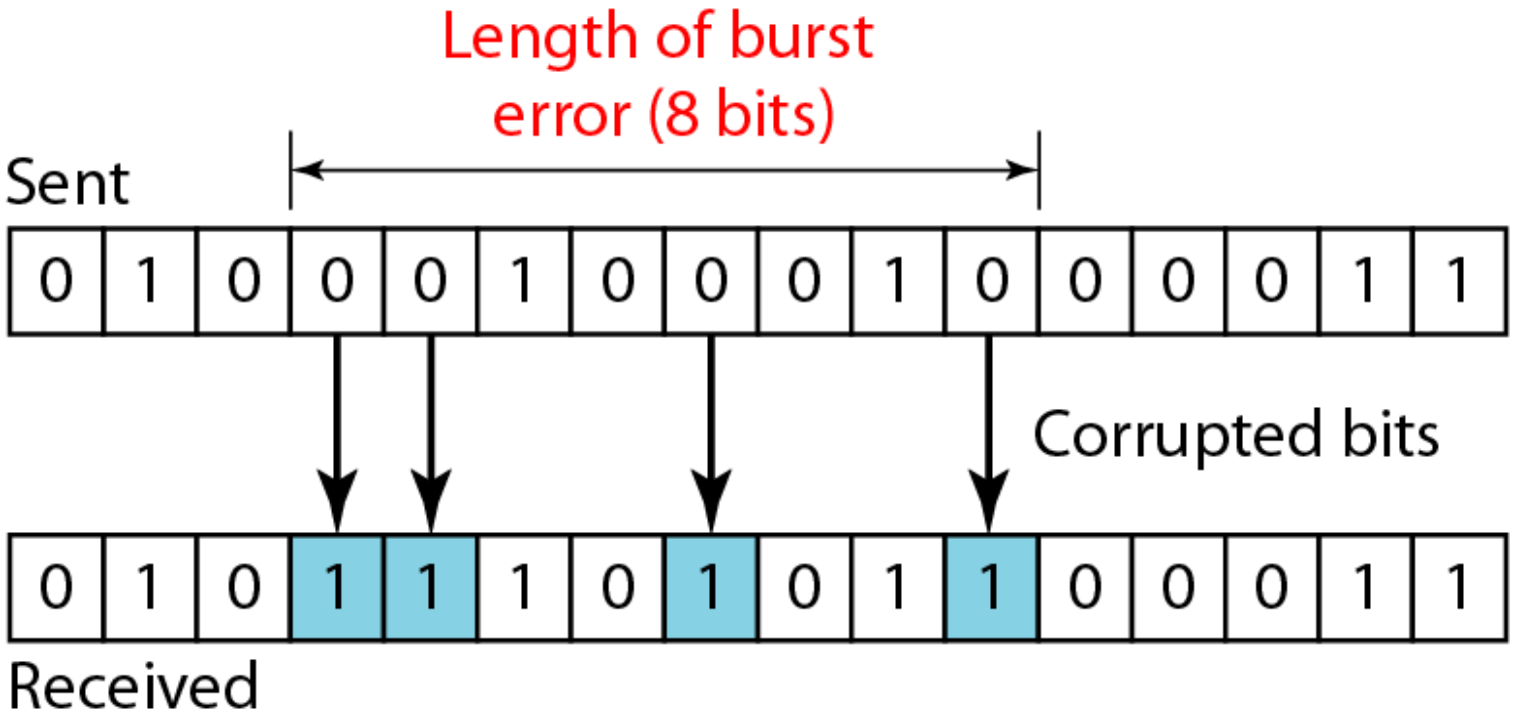
Figure 10.1 *Single-bit error*



Note

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

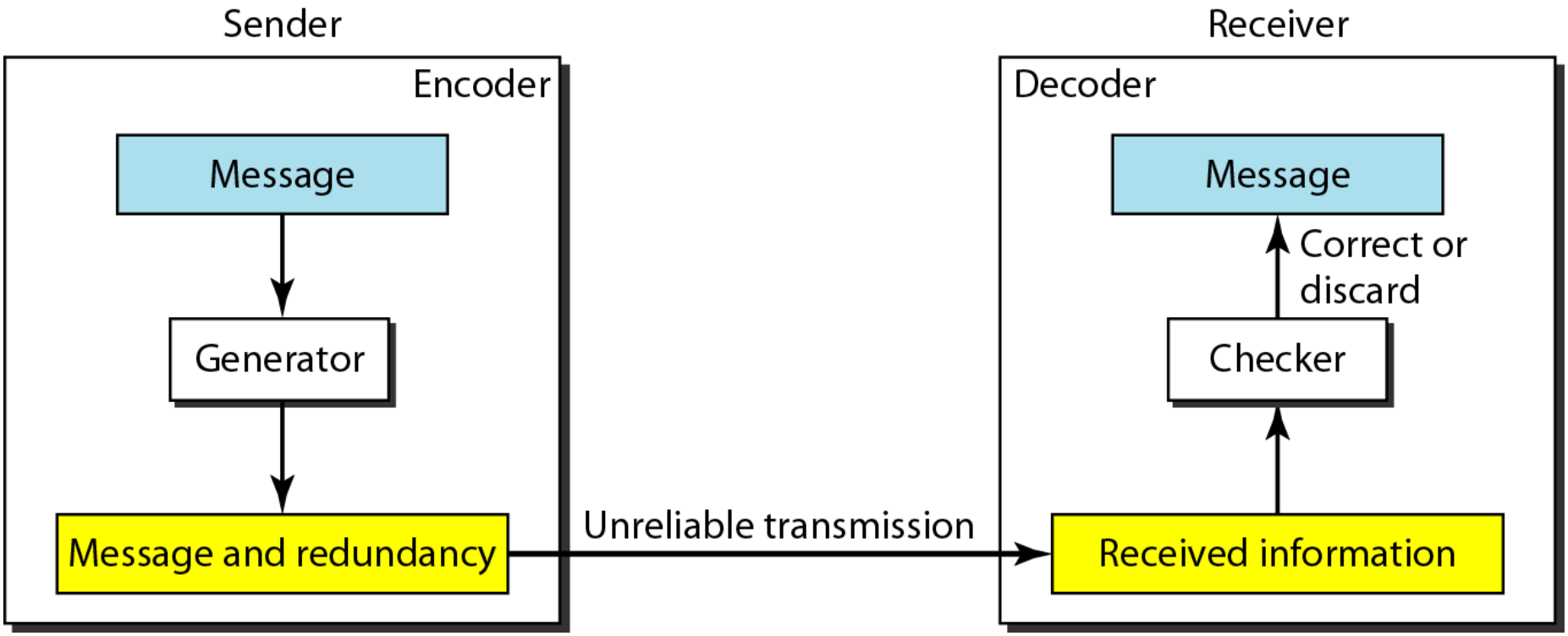
Figure 10.2 *Burst error of length 8*



Note

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

Figure 10.3 *The structure of encoder and decoder*



Note

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

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

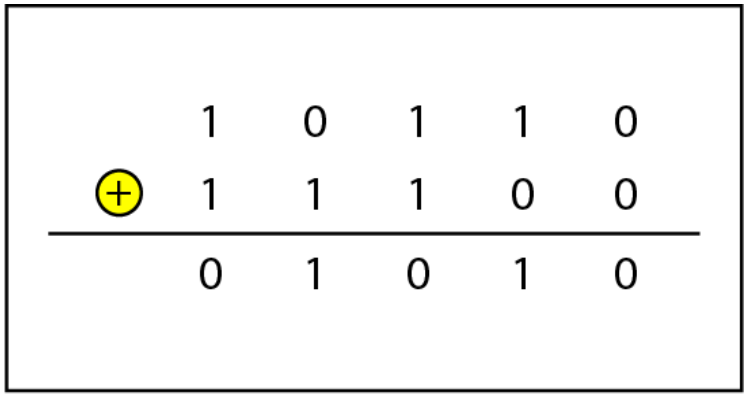
- ✓ Addition and subtraction in modulo arithmetic are simple.
- ✓ There is no carry when you add two digits in a column.
- ✓ There is no carry when you subtract one digit from another in a column.

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

Figure 10.4 *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

XORing has the same effect as Addition and subtraction in modulo-2 arithmetic

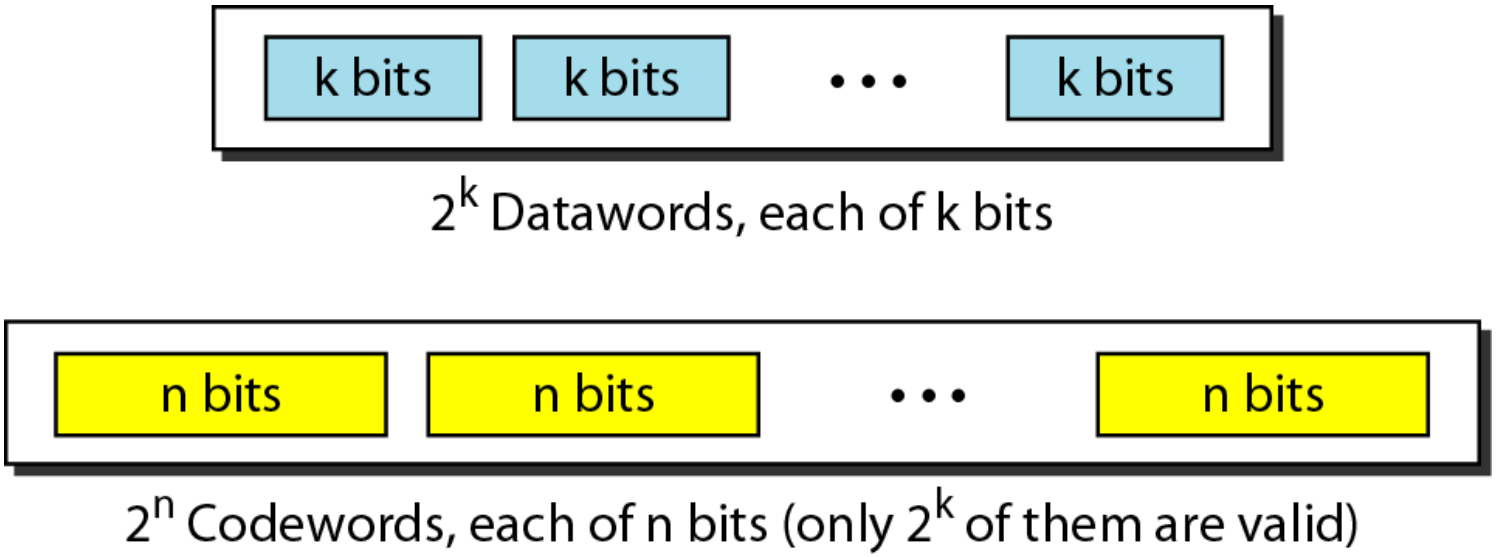
10-2 BLOCK CODING

- ✓ 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**.

Topics discussed in this section:

Error Detection
Error Correction
Hamming Distance
Minimum Hamming Distance

Figure 10.5 *Datawords and codewords in block coding*



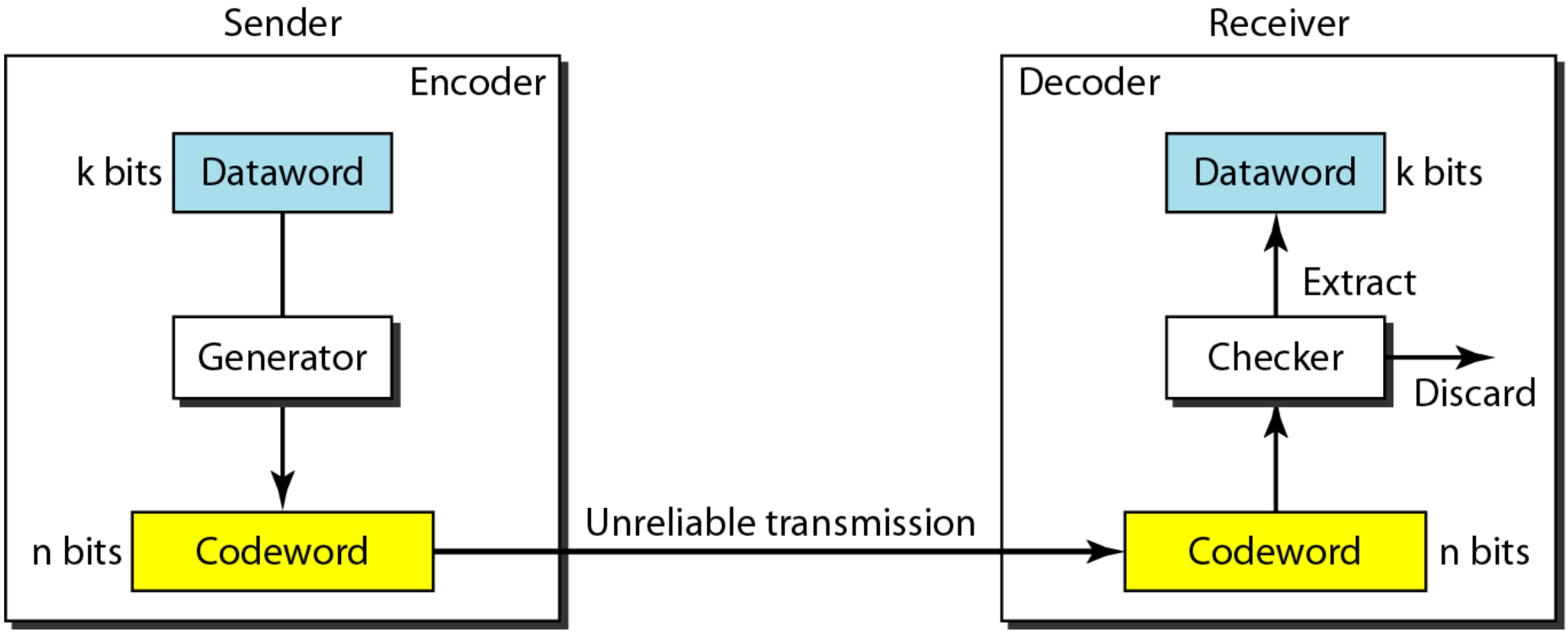
Example 10.1

The 4B/5B block coding discussed in Chapter 4 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.

Table 2.2.: 4B/5B encoding.

4-bit Data Symbol	5-bit Code
0000	11110
0001	01001
0010	10100
0011	10101
0100	01010
0101	01011
0110	01110
0111	01111
1000	10010
1001	10011
1010	10110
1011	10111
1100	11010
1101	11011
1110	11100
1111	11101

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.

Datawords	Codewords
00	000
01	011
10	101
11	110

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

Example 10.2 (continued)

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.

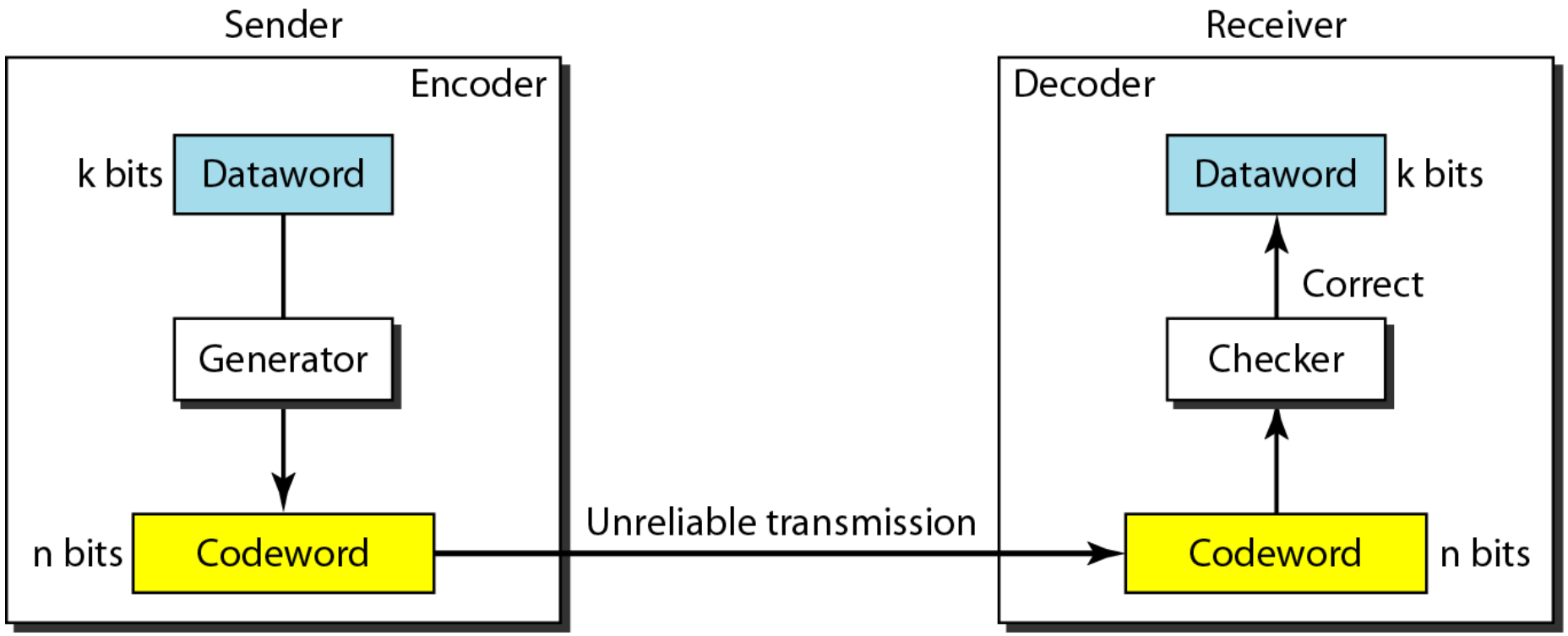
2. The codeword is corrupted during transmission, and **111** is **received**. This is not a valid codeword and is discarded.

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.

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*



Cyclic Redundancy Check (CRC)

- CRC is an error-detection technique commonly used in digital networks and storage devices to detect accidental changes to raw data.
- CRC is a type of **checksum**, and it works by taking a block of data, treating it as a large binary number, and dividing it by a predetermined "polynomial" using binary division.
- The remainder of this division is called the CRC value or checksum, which is appended to the data.
- At the receiver, if the recalculated CRC matches the transmitted CRC, it is assumed that the data has not been corrupted.

Table 10.7 *Standard Generator 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

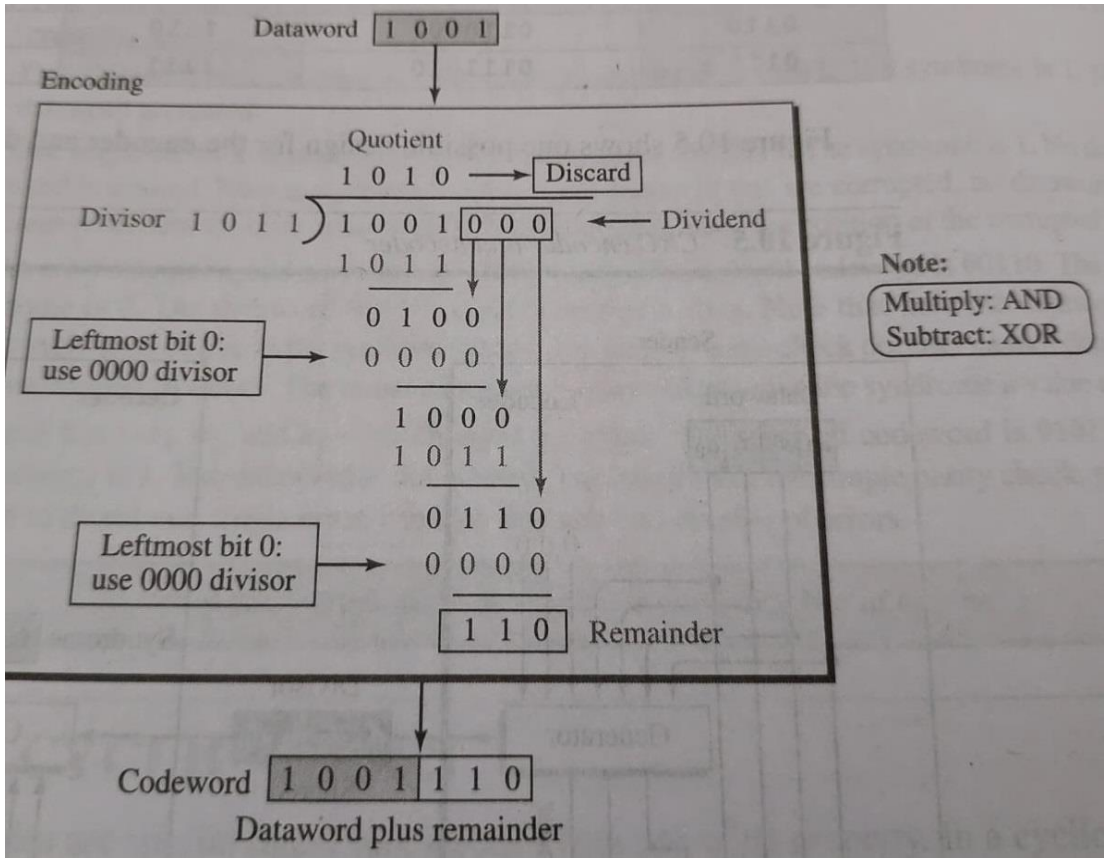
How CRC works?

1. **Data Representation**: The data to be transmitted is as a binary number.
2. **CRC Polynomial**: A generator polynomial (e.g., **1101** for a 3-bit CRC) is predefined, and it is used for the division.
3. **Binary Division**: The data is divided by the CRC polynomial using binary division (XOR operation).
4. **Remainder**: The remainder left after the division is the CRC value.
5. **Appended CRC**: This CRC value is appended to the end of the data before transmission.
6. **Verification**: On the receiving end, the data (including the CRC) is divided by the same polynomial. If the remainder is zero, the data is considered valid; otherwise, it's considered corrupted.

CRC: Implementation at the Sender

Suppose we have the **data 1001**, and we are using a generator polynomial **1011**.

1) **Append Zeros**: Append **n-1** zeros to the data (where ***n*** is the degree of the polynomial), making it **1001000.**
2) **Divide**: Perform binary division by the polynomial.
3) **Resulting Remainder**: The remainder is your CRC value, which is **appended to the original data**.

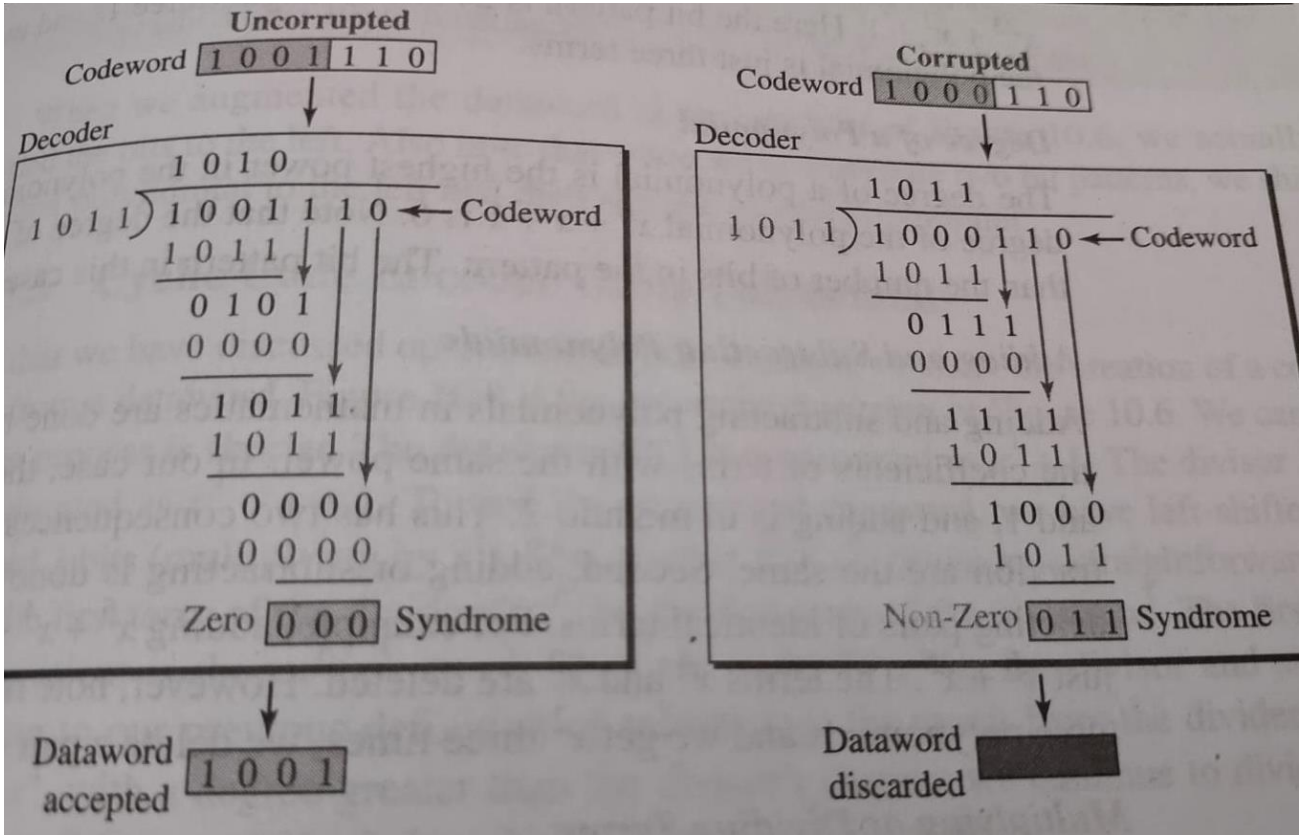


A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

CRC: Implementation at the Receiver

Suppose we received the **data 1001110**, and we are using a generator polynomial **1011**.

1) **Divide:** data using the same agreed upon generator polynomial.
2) **Resulting Remainder:** If the remainder is Zero, indicates successful data transmission. Otherwise, corrupted data received.



Test - 3

Date:13/09/24

Total marks: 15

Given the following frame data 1101011111, and a **generator polynomial represented by $x^4 + x + 1$** (which corresponds to the binary sequence 10011):

1. Compute the CRC checksum using the CRC method. Show all the steps involved in dividing the frame data by the generator polynomial.
2. Compute the codeword that the sender would encode before transmitting the data.

Note

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

Hamming code

- Hamming code is used for **error-detection** and **error-correction**
- It works by adding **redundant parity bits** to the data to enable error detection and correction
- A **frame** consists of ***m* data** (i.e., **message**) bits and ***r* redundant** bits, thus, a **codeword, n = m+r**
- **The number of positions in which two codewords differ is called the Hamming Distance (Hamming, 1950)**

XOR

10001001
10110001
00111000

3 bits differ

Hamming code

To **detect *d* errors**, you need a distance ***d* + 1** code because with such a code there is no way that *d* single-bit errors can change a valid codeword into another valid codeword **error detection and correction.**
To **correct *d* errors**, you **need a distance 2*d* + 1 code** because that way the legal codewords are so far apart that even with ***d*** changes the original codeword is still closer than any other codeword.

Four valid codewords (with hamming distance ***d*=5**):
0000000000, 0000011111, 1111100000, and 1111111111

If the codeword 0000000111 arrives and we expect only single/double-bit errors, the receiver will know that the original must have been 0000011111.

If, however, a triple error changes 0000000000 into 0000000111, the error will not be corrected properly.

Hamming code

- If we want to design a code with ***m*** message bits and ***r*** check bits that will allow all **single errors** to be corrected.
- Each of the **2^m legal messages** has ***n*** illegal codewords at a **distance of 1** from it.
- These are formed by systematically inverting each of the ***n*** bits in the ***n*-bit codeword** formed from it.
- Thus, each of the **2^m legal** messages requires ***n* + 1** bit patterns **(n-illegal** + **1-legal** codewords).
- Since the total number of bit patterns is **2^n**, we must have **$(n + 1)2^m \leq 2^n$**. Using ***n = m + r***, this requirement becomes

$$(m + r + 1) \leq 2^r$$

Hamming code

- In **Hamming codes** the bits of the **codeword are numbered consecutively**, starting with bit 1 at the left end, bit 2 to its immediate right, and so on.
- The **parity/check bits** are **powers of 2** (1, 2, 4, 8, 16, etc.).
- **The data bits are (3, 5, 6, 7, 9, etc.)** the remaining positions.

Codeword:

1	2	3	4	5	6	7	8	9	10	11

Parity bit:

P1	P2		P3				P4			
1	2	3	4	5	6	7	8	9	10	11

Data bit:

P1	P2	M1	P3	M2	M3	M4	P4	M5	M6	M7
1	2	3	4	5	6	7	8	9	10	11

Hamming code

- Example: data or message = 1000001, 7 bit message.

Codeword:

P1	P2	1	P3	0	0	0	P4	0	0	1
1	2	3	4	5	6	7	8	9	10	11

Computing Parity bit: 1's in specific positions

	P4	P3	P2	P1
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

- **To determine parity value for P1 use positions (1,3,5,7,9 and 11):**
  P1 + 1 + 0 + 0 + 0 + 1. Now since there are even number of 1s, so P1 is 0.

- **To determine parity value for P2 use positions (2,3,6,7,10 and 11):**
  P2 + 1 + 0 + 0 + 0 + 1. There are even number of 1s, so P2 is also 0.

- **To determine parity value for P3 use positions (4,5,6, and 7):**
  P3 + 0 + 0 + 0. There are no1s, so P2 is also 0 for even parity.

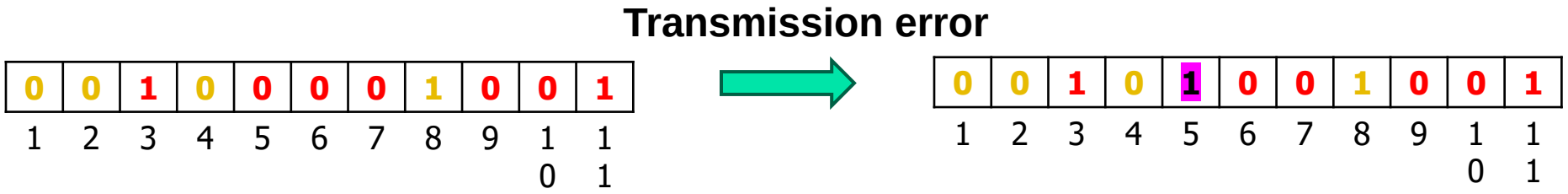
- **To determine parity value for P4 use positions (8,9,10, and 11):**
  P4 + 0 + 0 + 1. So P4 is 1 for even parity.

Thus, Codeword:

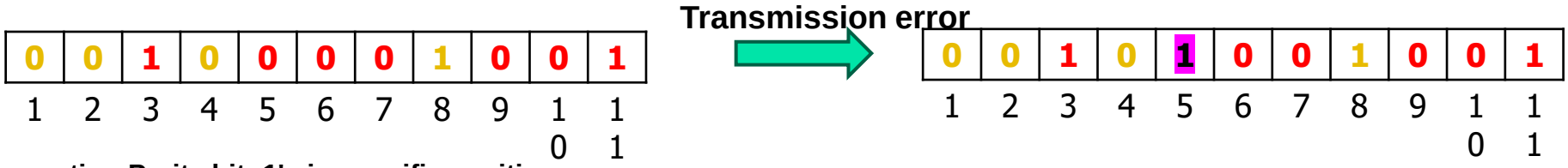
0	0	1	0	0	0	0	1	0	0	1
1	2	3	4	5	6	7	8	9	10	11

Hamming code

- When a codeword arrives, **the receiver redoes the check bit computations** including the values of the received check bits.
- We call these the check results.
- If the **check** bits **are correct** then, **for even parity sums**, each **check result should be zero**. In this case the **codeword is accepted as valid**
- **Error-detection Process:**
  - **Recalculate the parity bits** based on the received data and compare them to the received parity bits.
  - **Use the binary combination of recalculated parity bits** to locate the position of the error.



Hamming code



Computing Parity bit: 1's in specific positions

	P4	P3	P2	P1
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

- **To determine parity value for P1 use positions (1,3,5,7,9 and 11):**
  0 + 1 + 1 + 0 + 0 + 1. Error: since even parity, so P1 is 1.
- **To determine parity value for P2 use positions (2,3,6,7,10 and 11):**
  0 + 1 + 0 + 0 + 0 + 1. There are even number of 1s, so P2 is also 0.
- **To determine parity value for P3 use positions (4,5,6, and 7):**
  0 + 1 + 0 + 0. Error: only one 1s, so P3 should be 1.
- **To determine parity value for P4 use positions (8,9,10, and 11):**
  1 + 0 + 0 + 1. So P4 is 0 for even parity.

Thus, the Error syndrome: (p4, p3, p2, p1) is (0101) or (4 + 1 = **5) position**.
Therefore, we flip the fifth position to correct the error.

10.35

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)***

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

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

$000 \oplus 011$ is 011 (two 1s)

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

$10101 \oplus 11110$ is 01011 (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.

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

The d_{min} in this case is 2.

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.

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. 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.

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

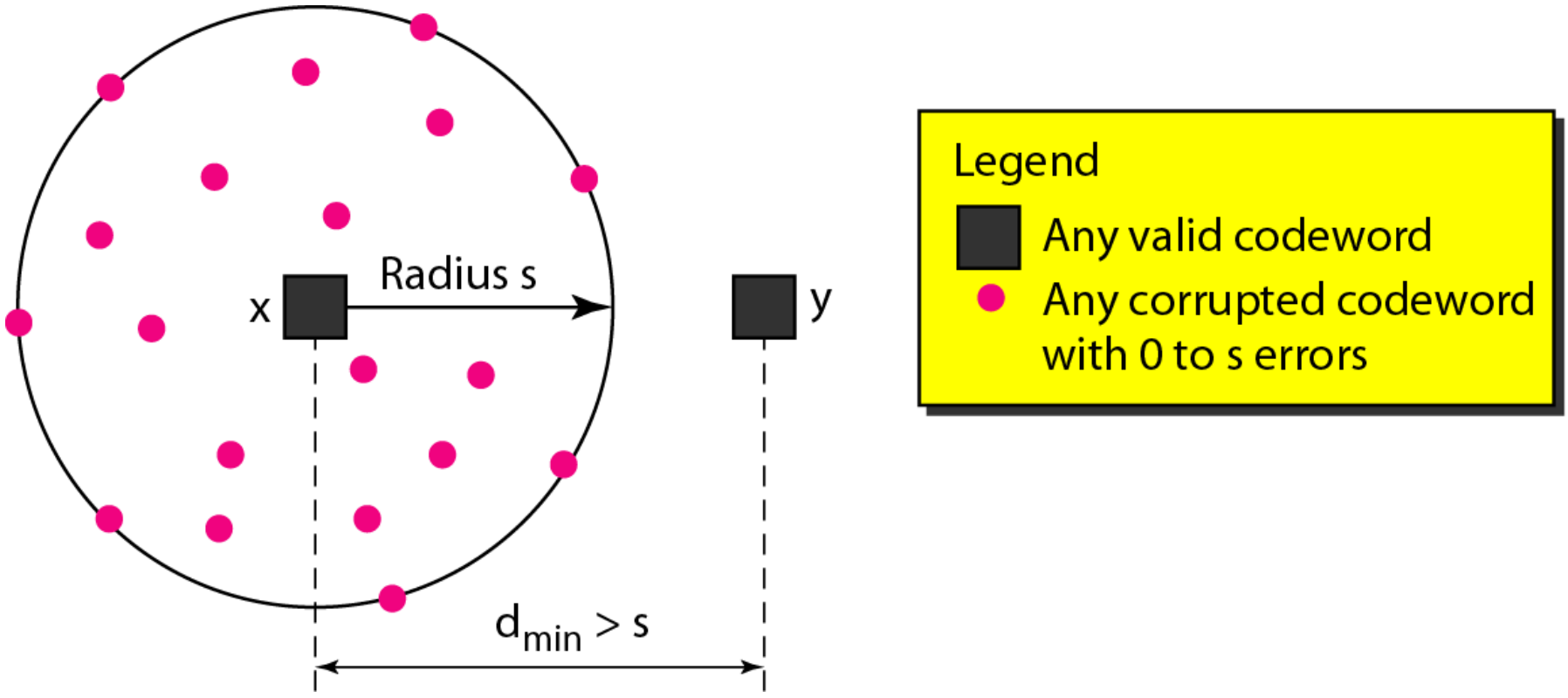
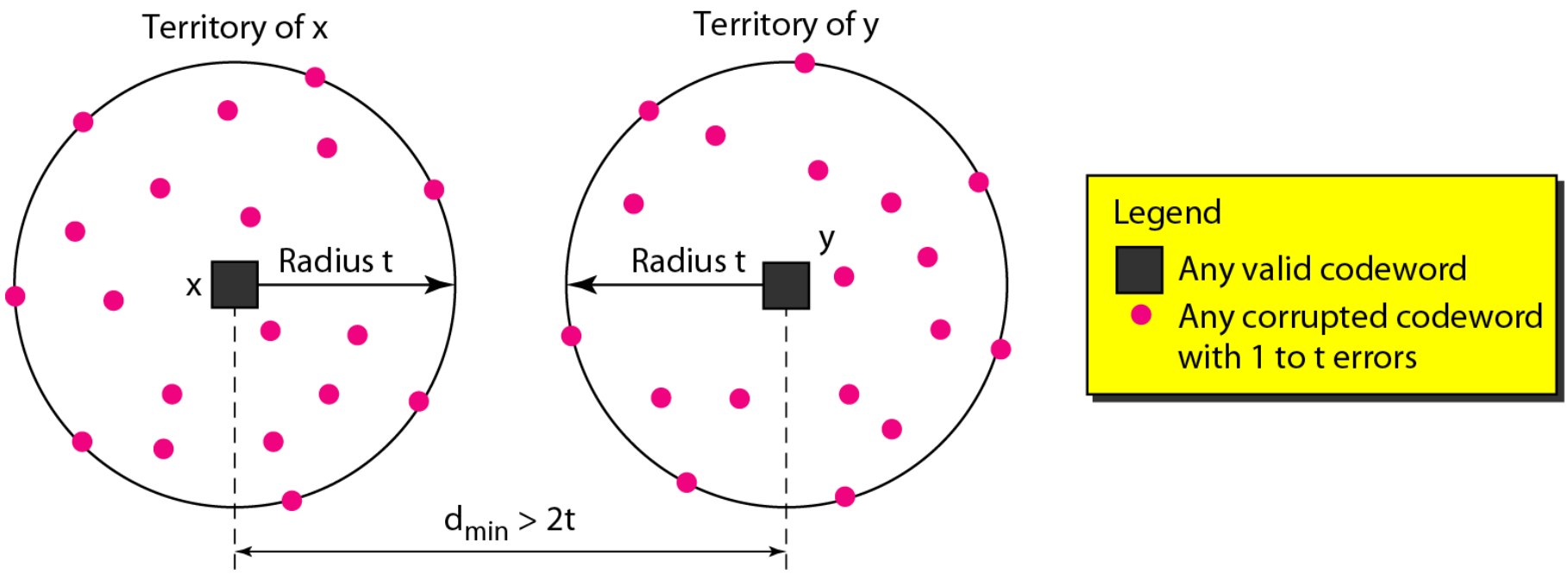


Figure 10.9 *Geometric concept for finding d_{min} in error correction*



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 (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 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 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 $d_{min} = 3$.

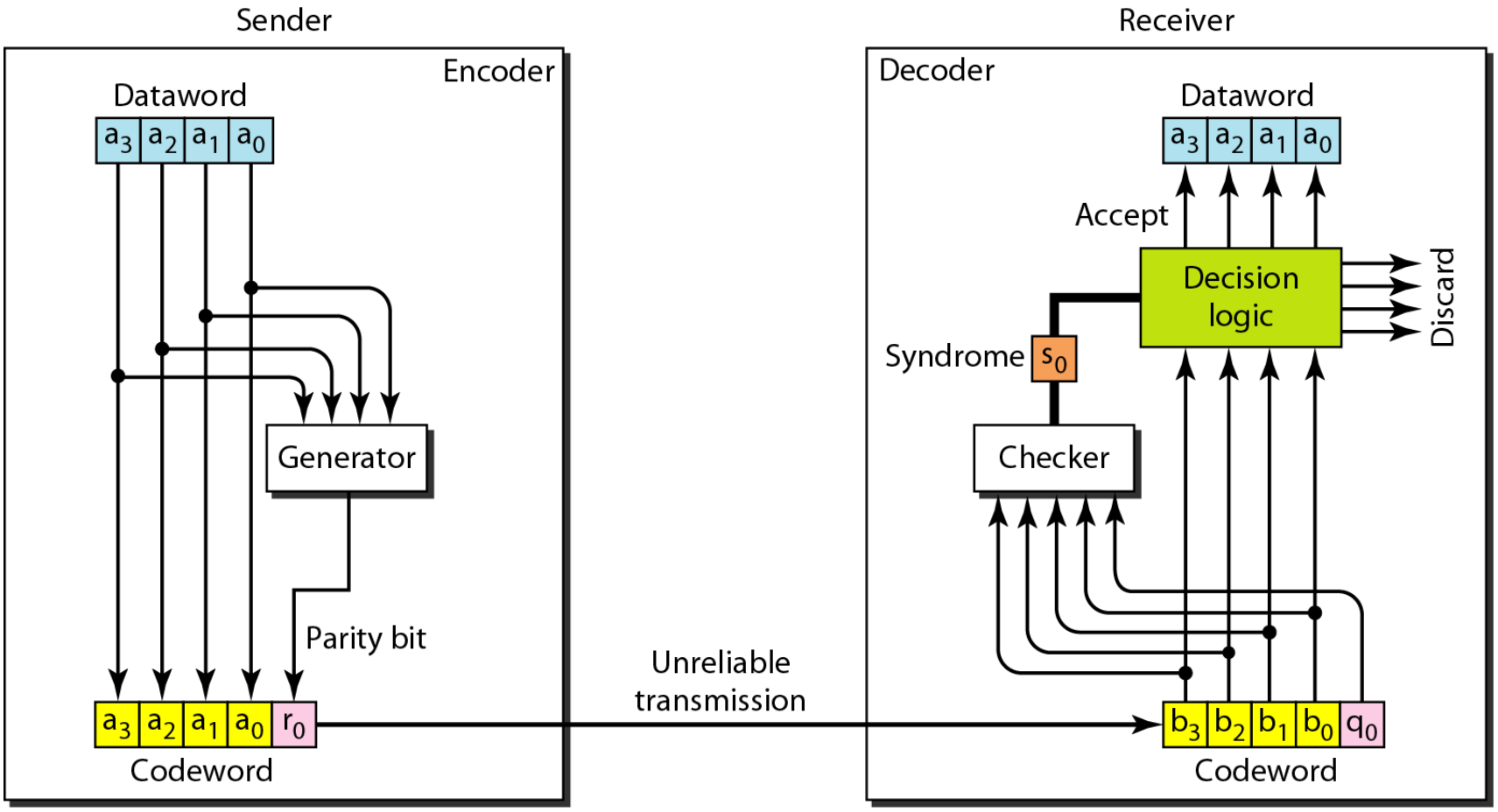
Note

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)***

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

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 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.*

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

Note

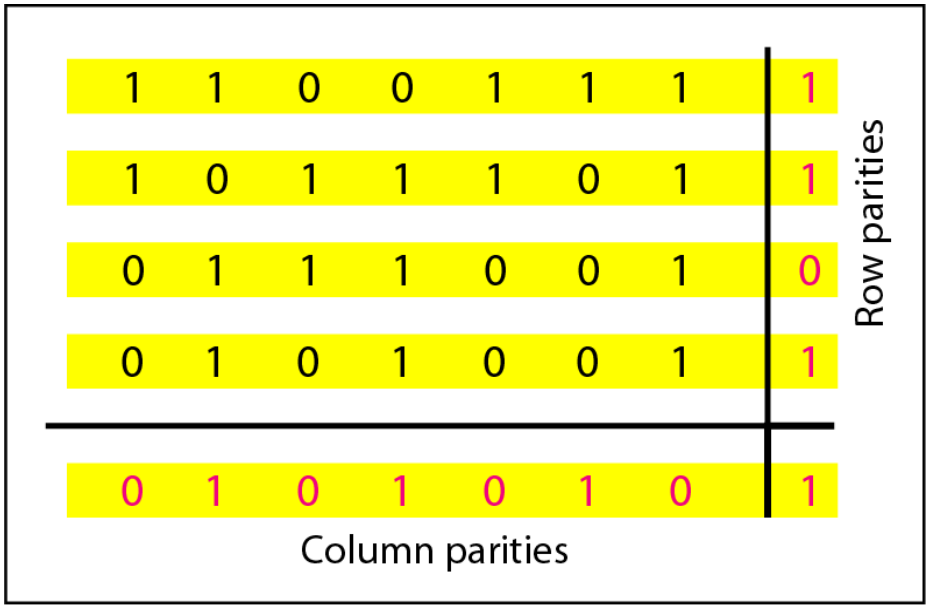
A simple parity-check code can detect an odd number of errors.

Note

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

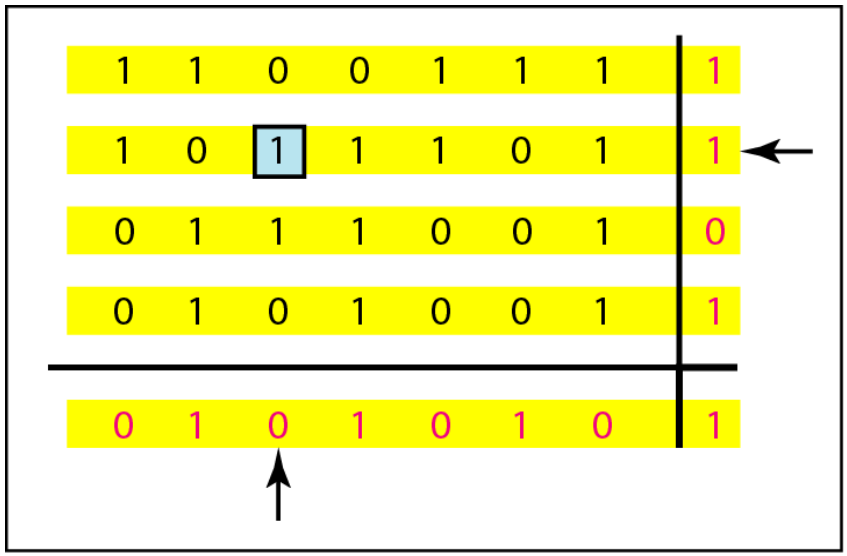
The relationship between *m* and *n* in these codes is $n = 2m - 1$.

Figure 10.11 *Two-dimensional parity-check code*

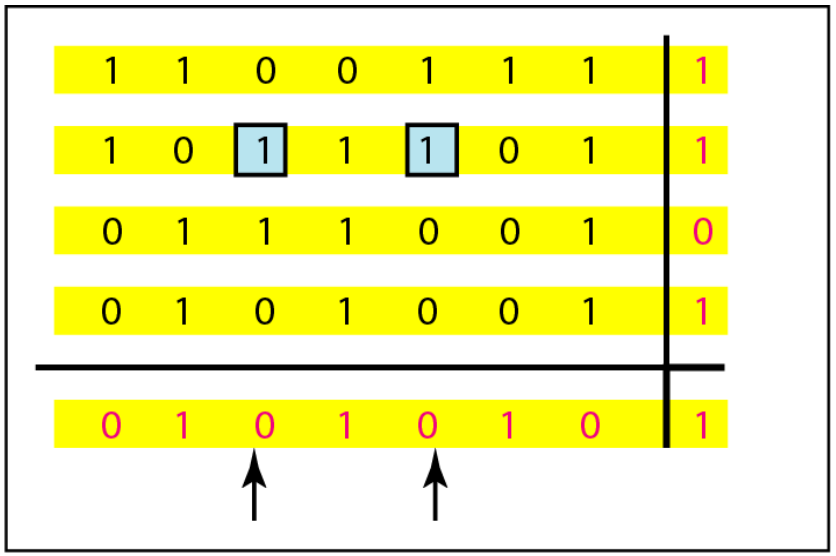


a. Design of row and column parities

Figure 10.11 *Two-dimensional parity-check code*

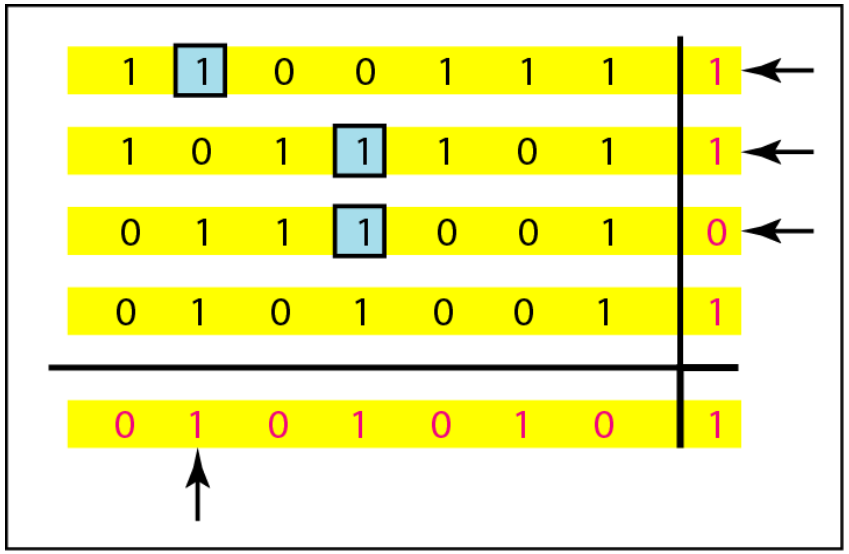


b. One error affects two parities

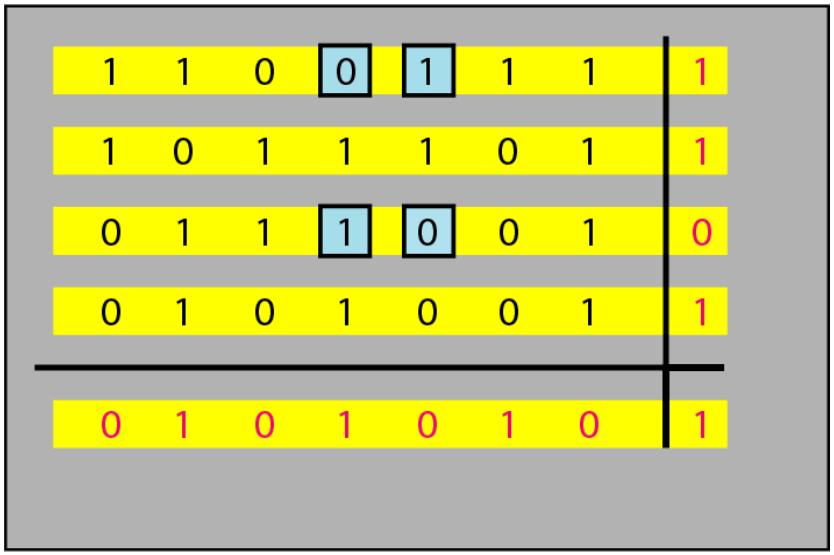


c. Two errors affect two parities

Figure 10.11 *Two-dimensional parity-check code*



d. Three errors affect four parities



e. Four errors cannot be detected

Table 10.4 *Hamming code C(7, 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

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

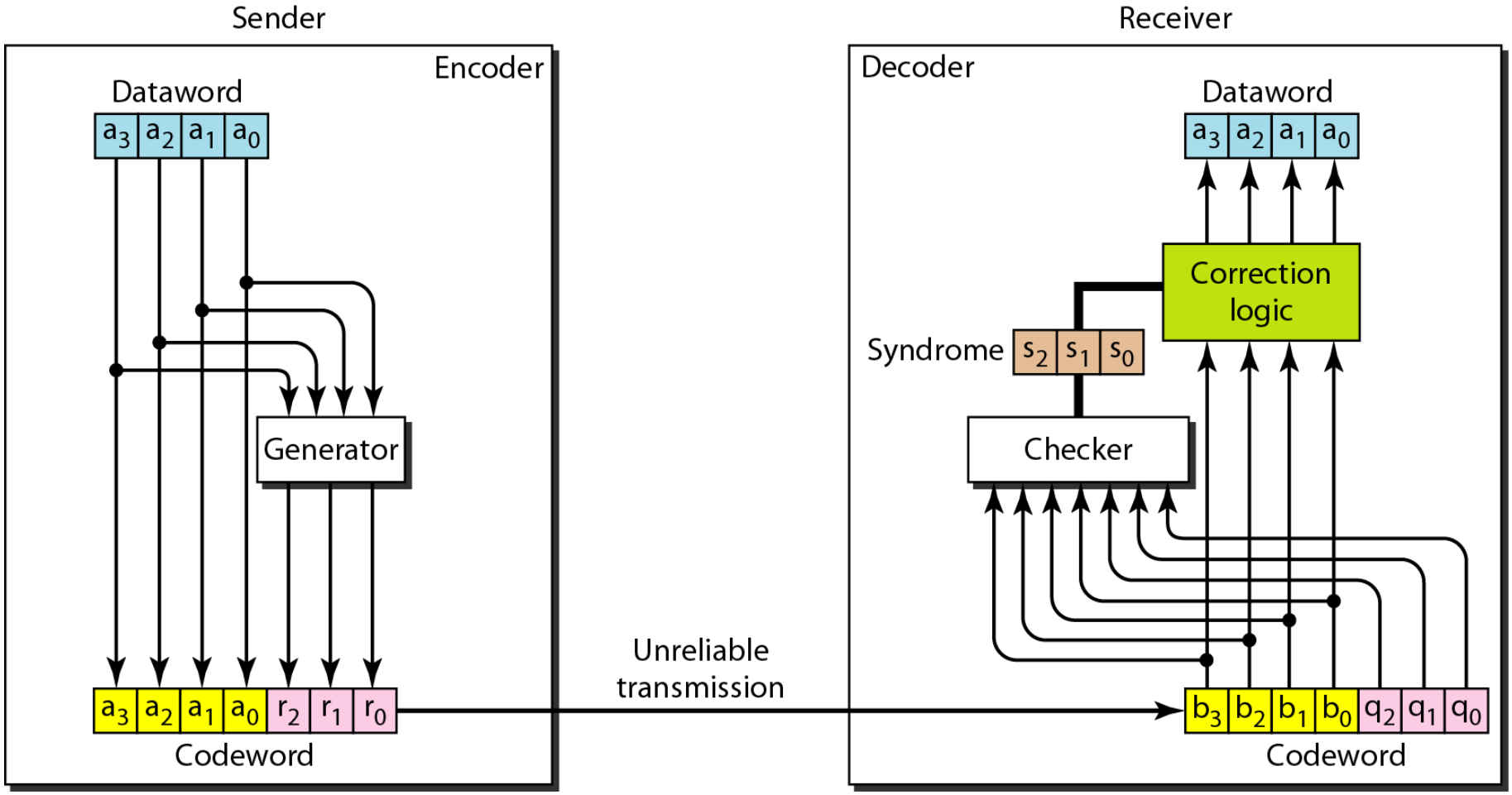


Table 10.5 ***Logical decision made by the correction logic analyzer***

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

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.

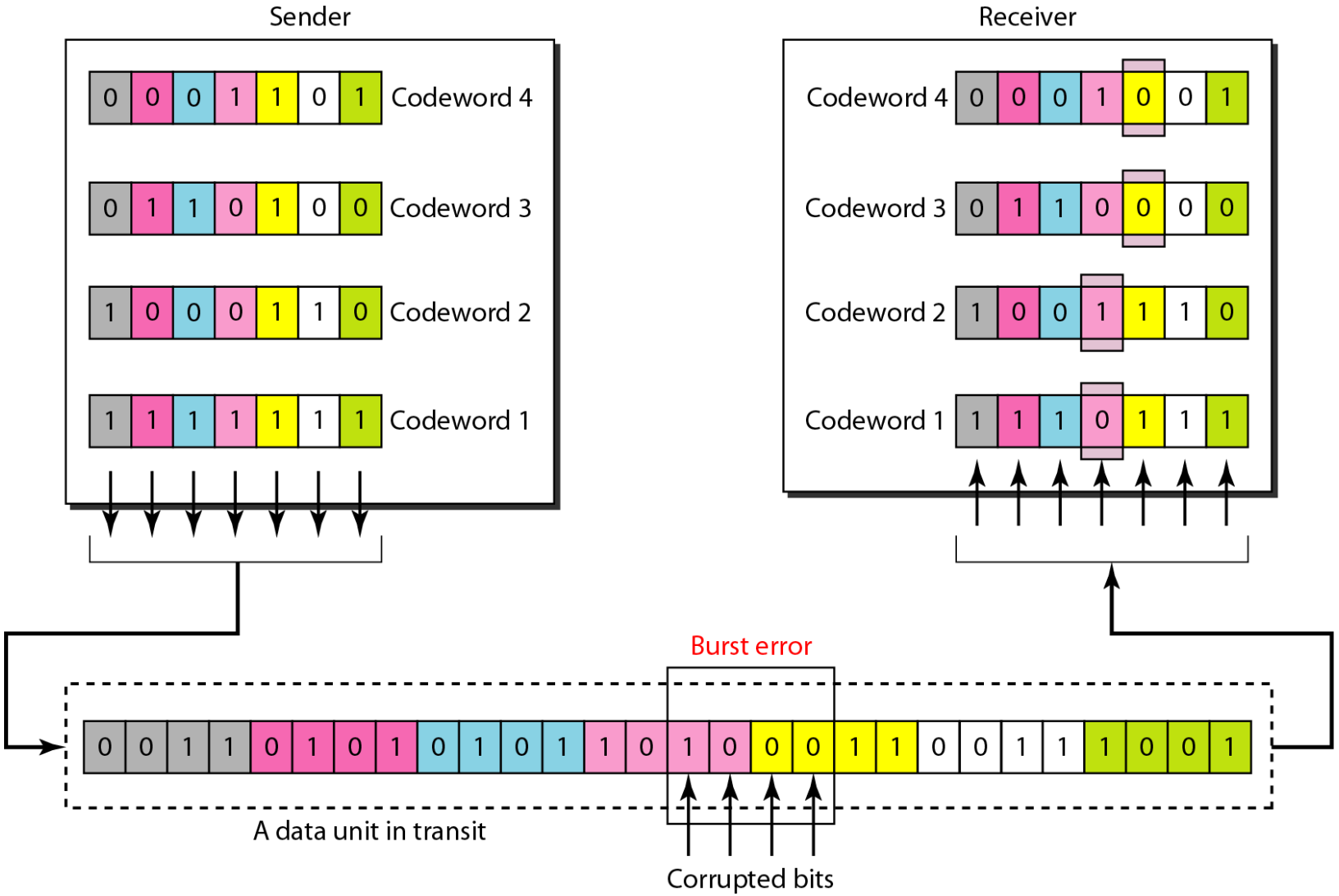
Solution

We need to make $k = n - m$ greater than or equal to 7, or $2m - 1 - m \geq 7$.

1. *If we set $m = 3$, the result is $n = 23 - 1$ and $k = 7 - 3$, or 4, which is not acceptable.*
2. *If we set $m = 4$, then $n = 24 - 1 = 15$ and $k = 15 - 4 = 11$, which satisfies the condition. So the code is*

C(15, 11)

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)*

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

Figure 10.14 *CRC encoder and decoder*

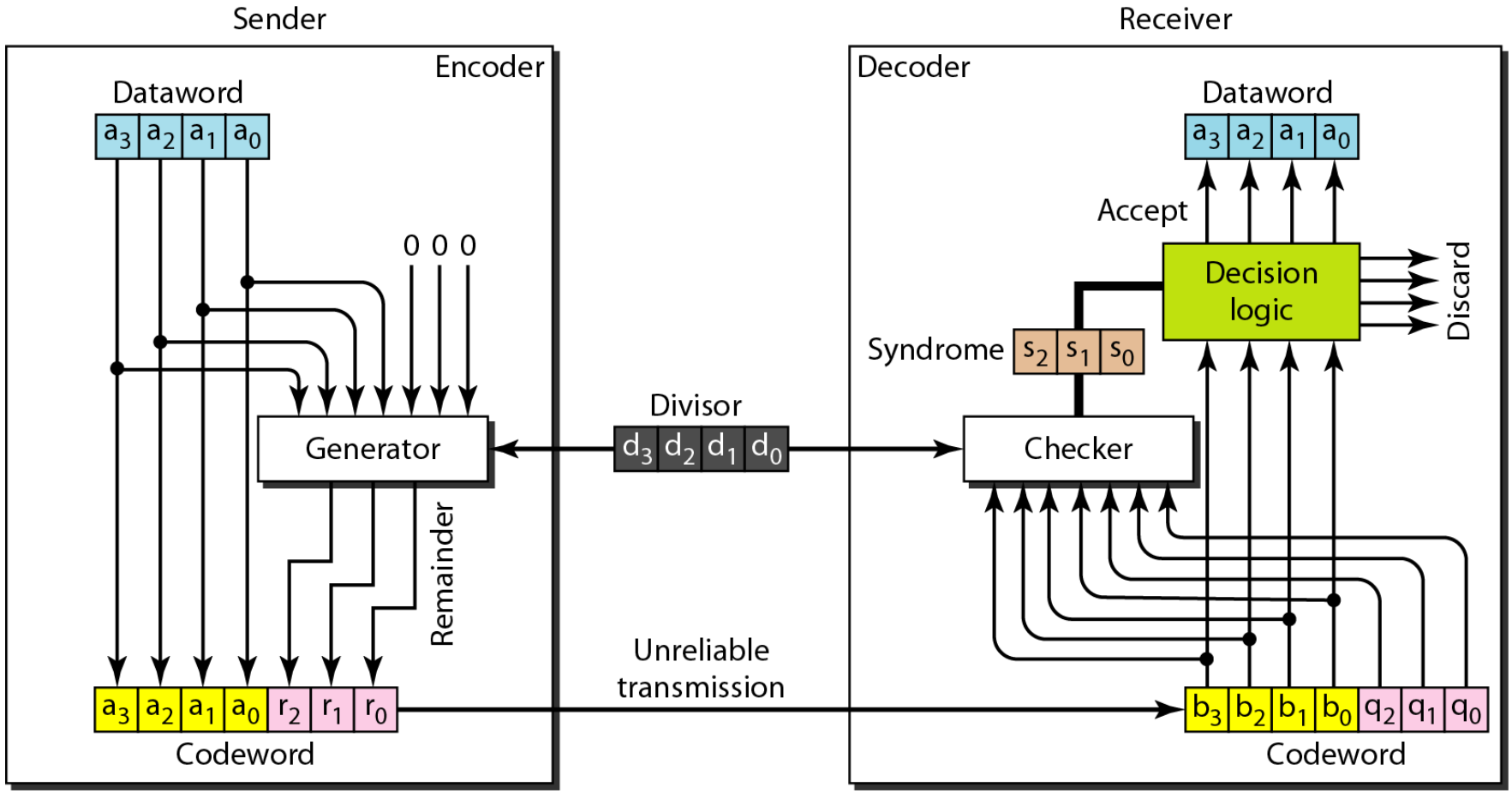


Figure 10.15 *Division in CRC encoder*

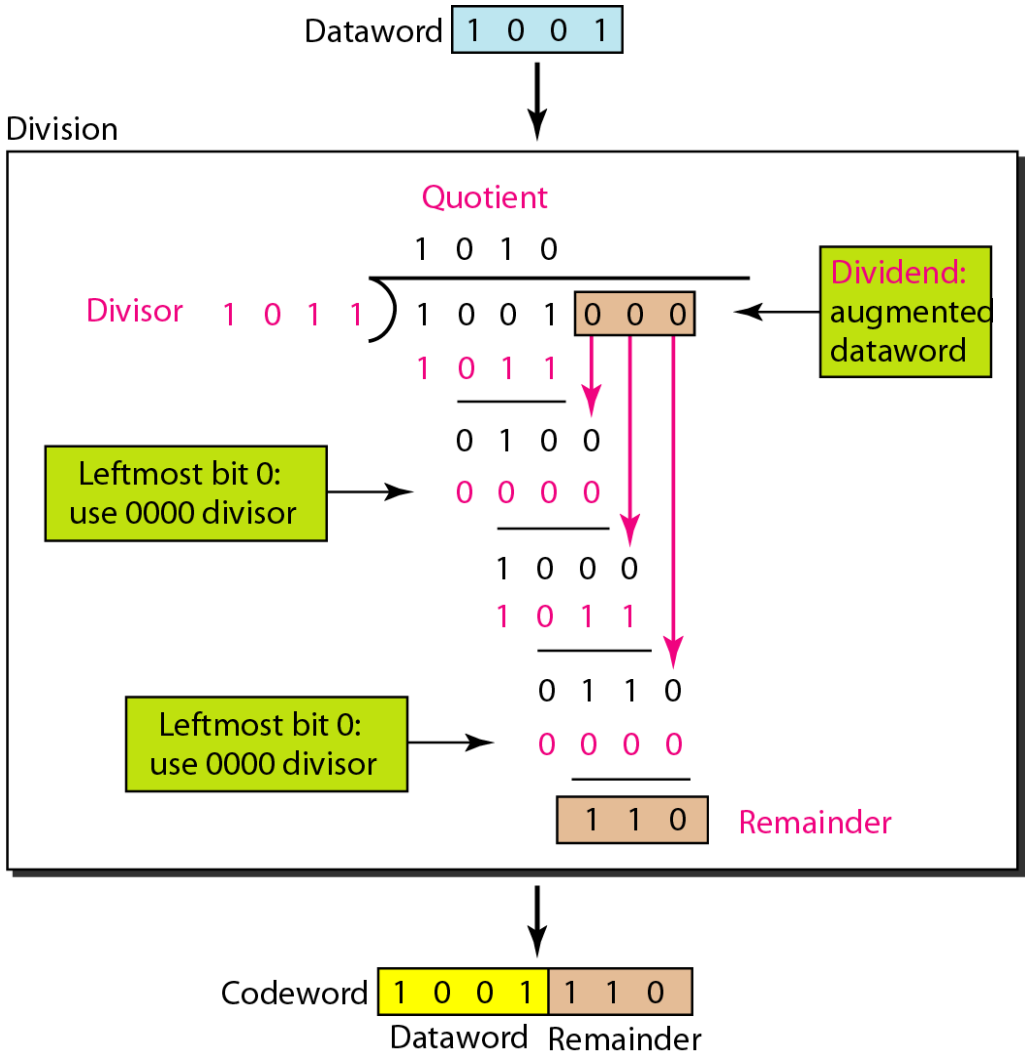


Figure 10.16 *Division in the CRC decoder for two cases*

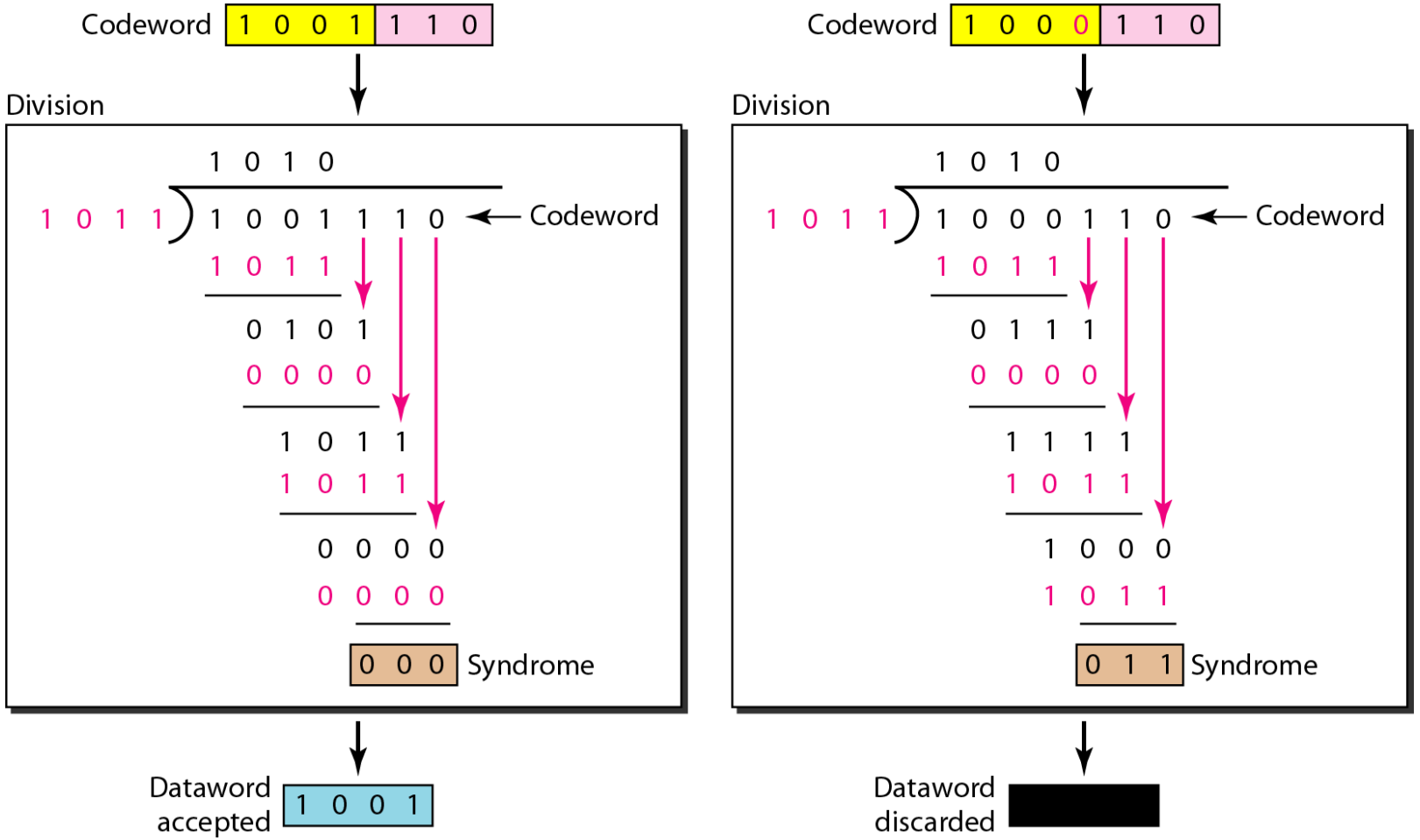


Figure 10.17 *Hardwired design of the divisor in CRC*

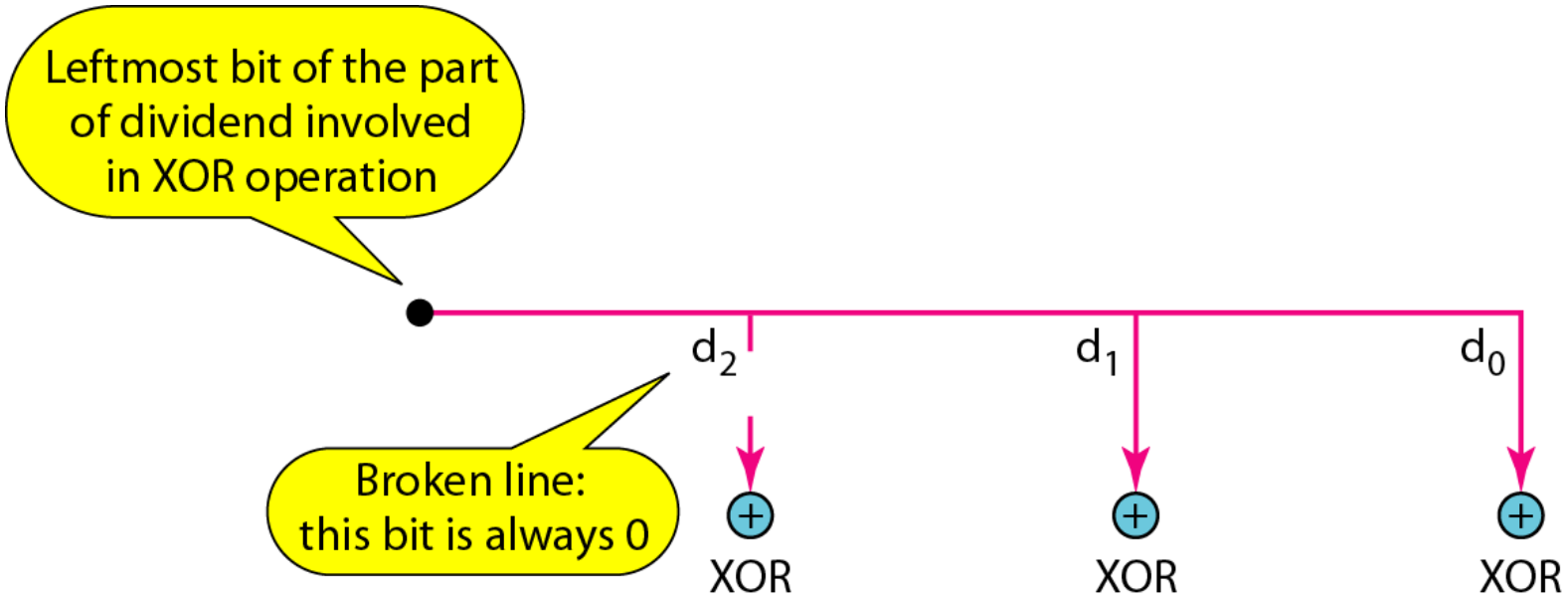


Figure 10.18 *Simulation of division in CRC encoder*

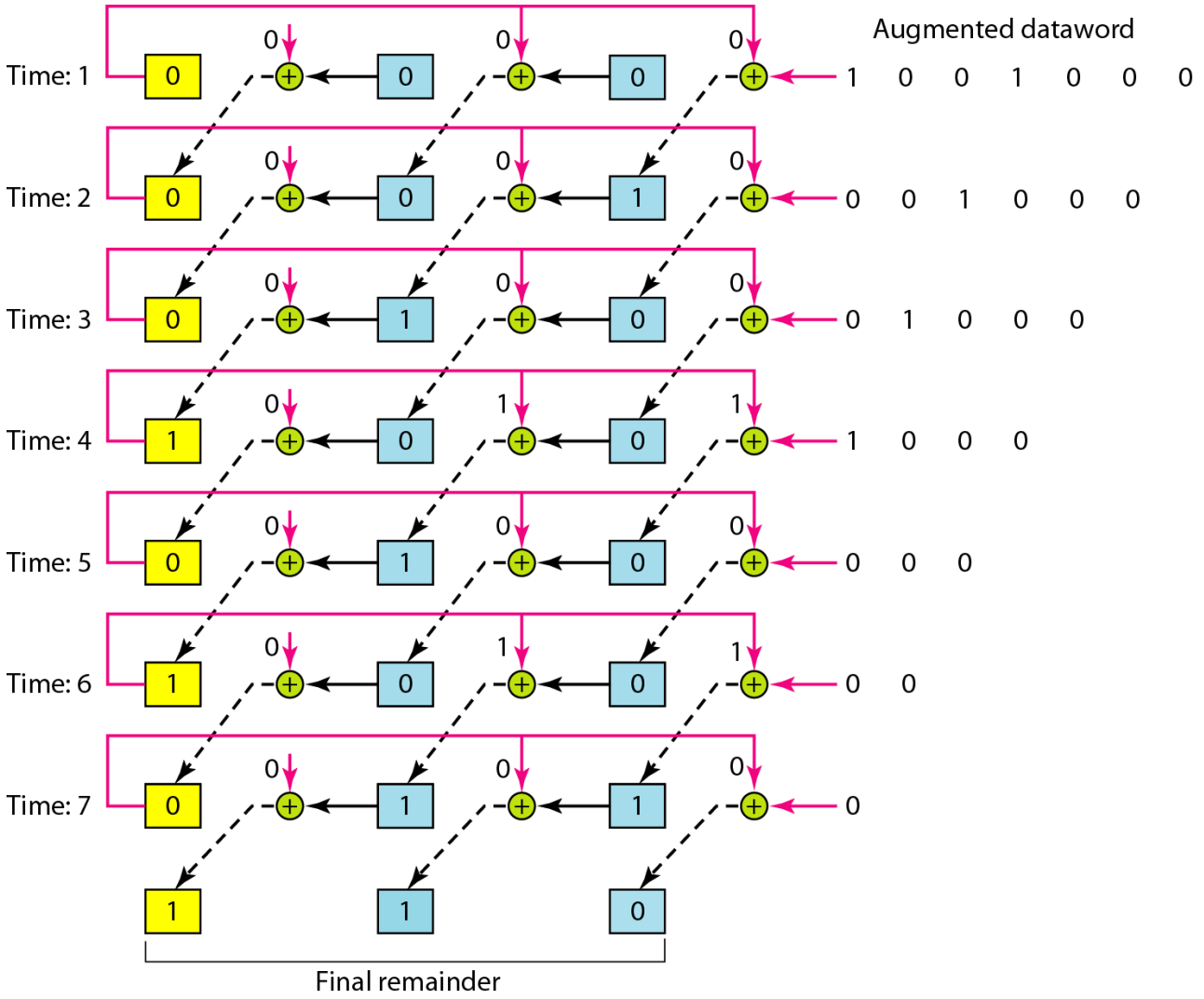


Figure 10.19 *The CRC encoder design using shift registers*

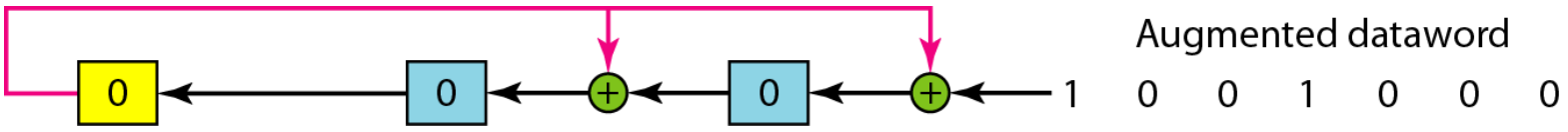


Figure 10.20 *General design of encoder and decoder of a CRC code*

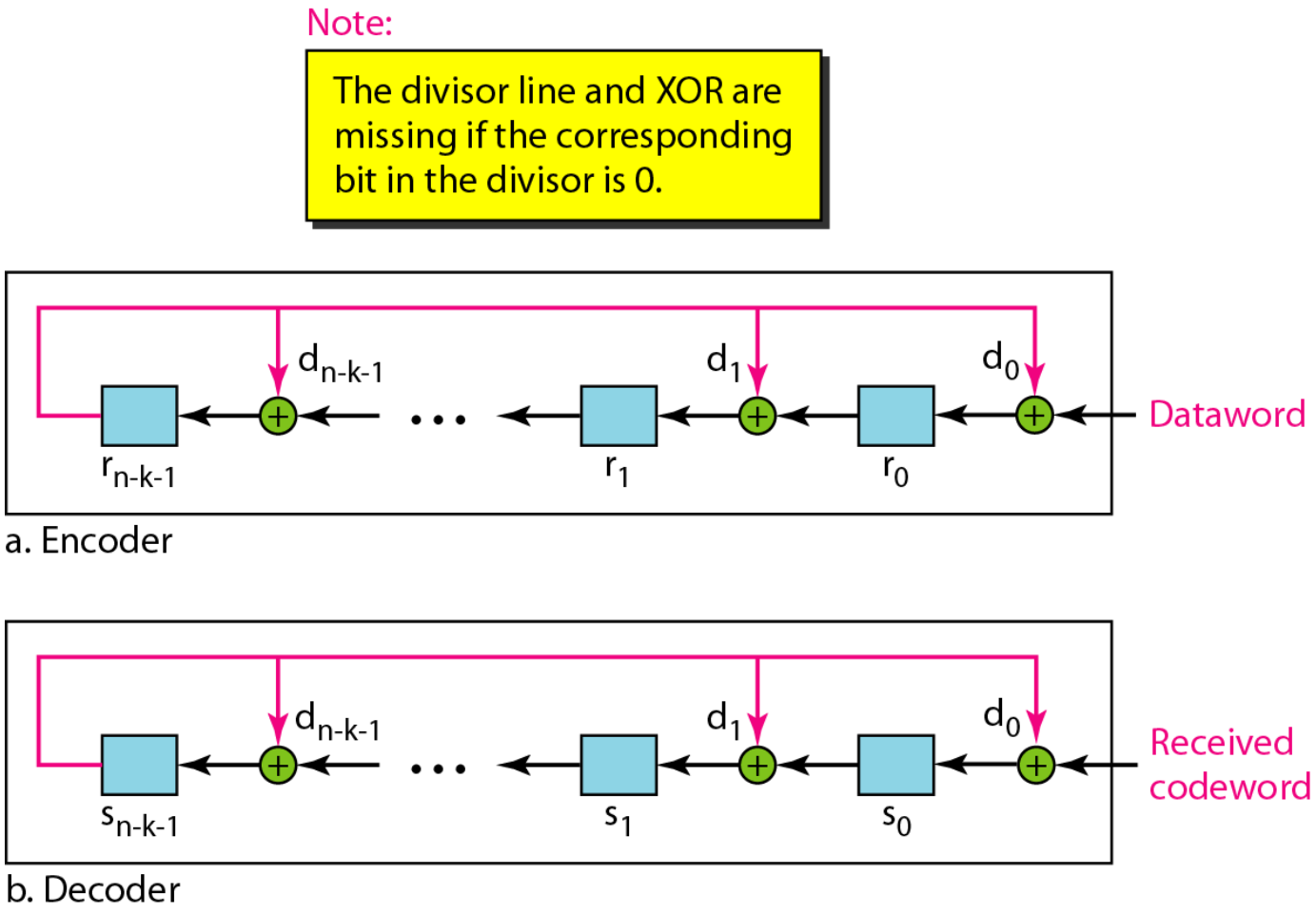
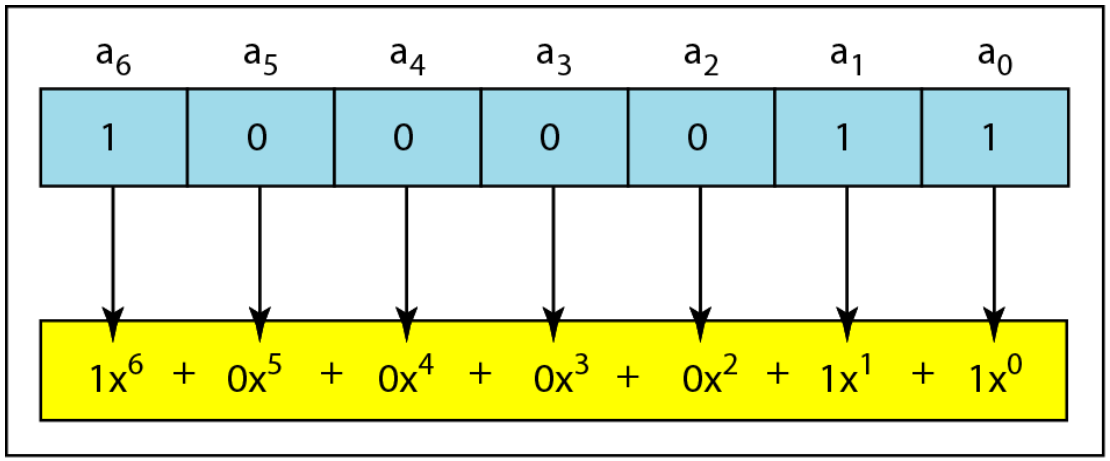
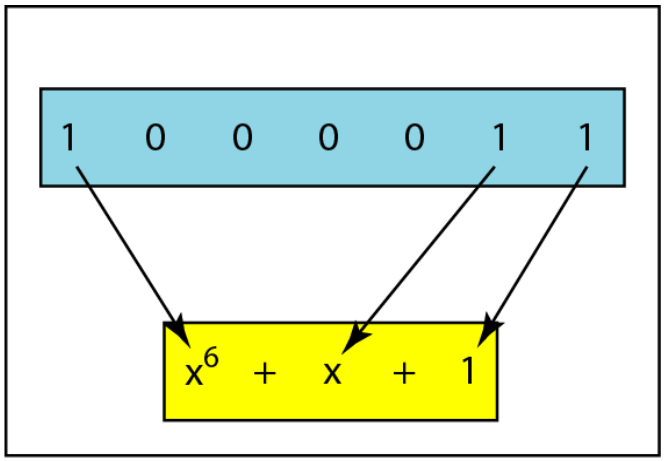


Figure 10.21 *A polynomial to represent a binary word*

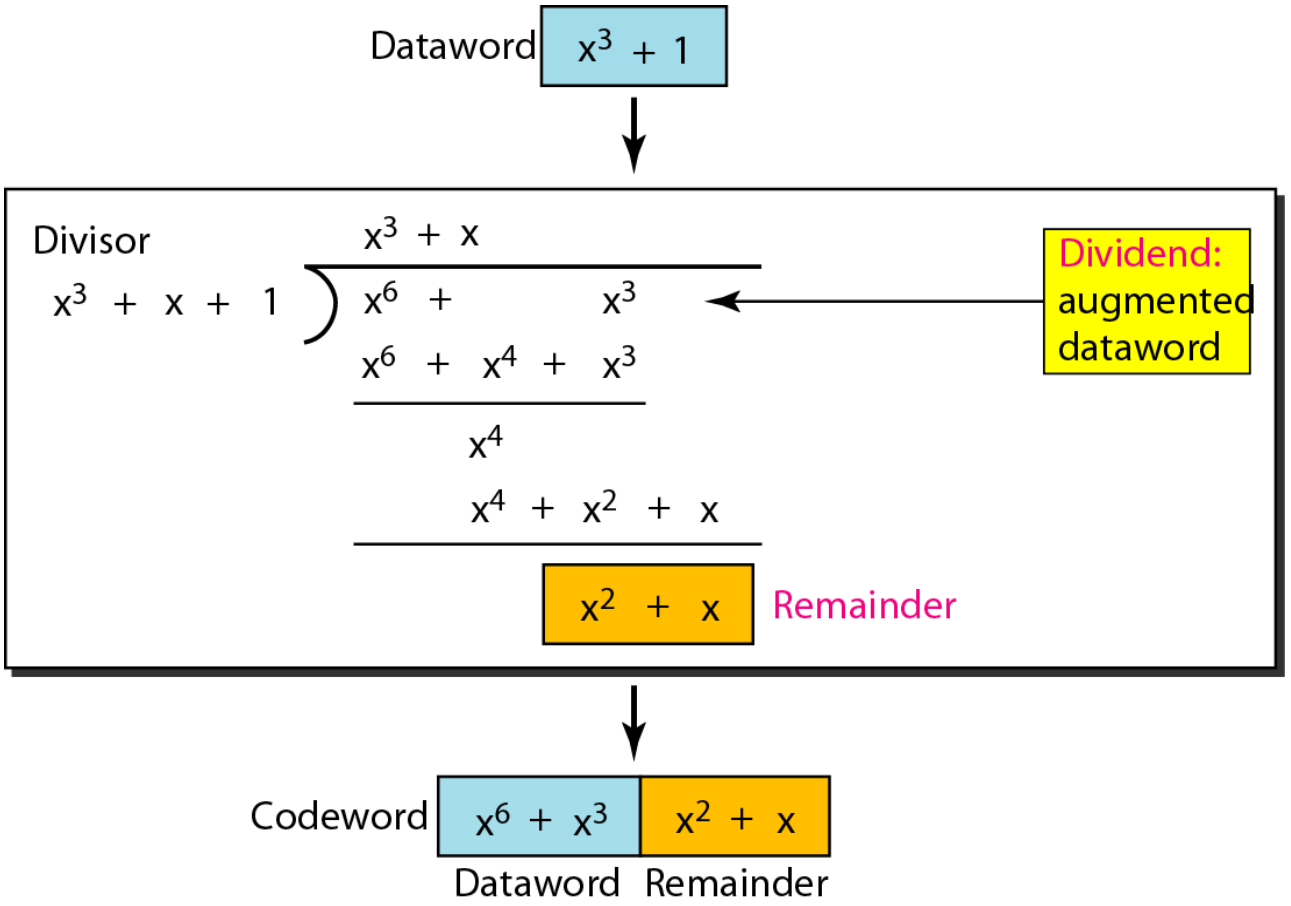


a. Binary pattern and polynomial



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) ≠ 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 *e*(x) errors 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 errors can be caught.

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

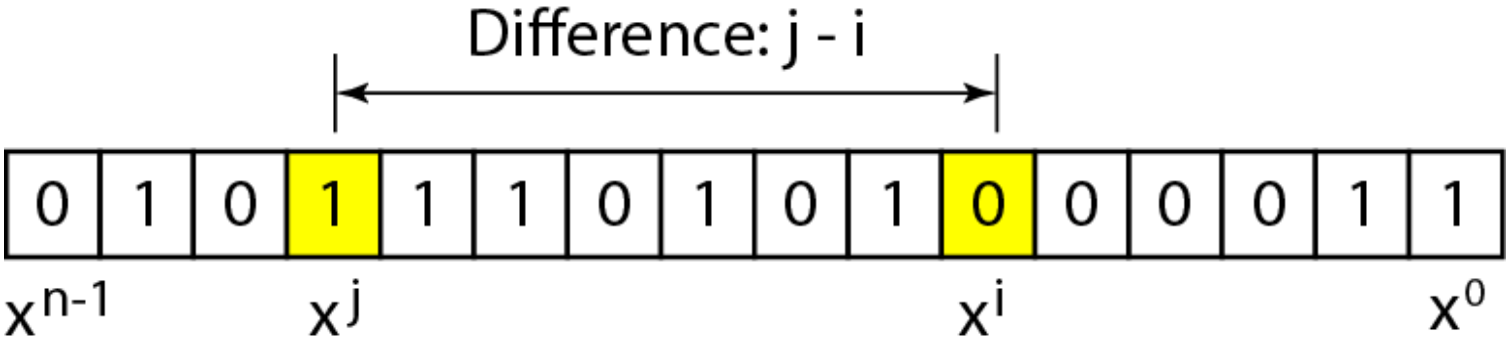
Solution

a. No x^i can be divisible by $x + 1$. Any single-bit error can be caught.

b. If i is equal to or greater than 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 (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 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).

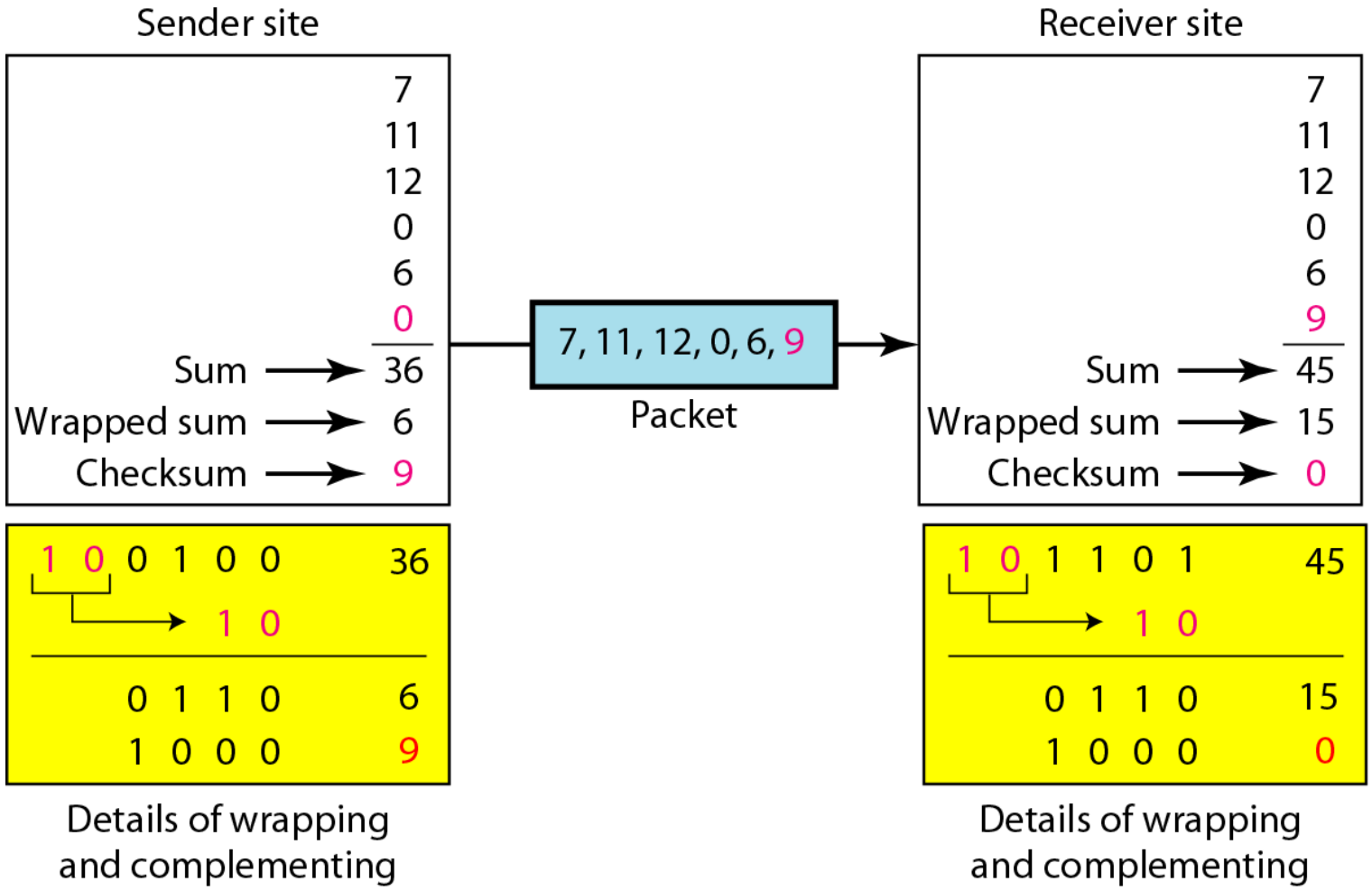
Example 10.22

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

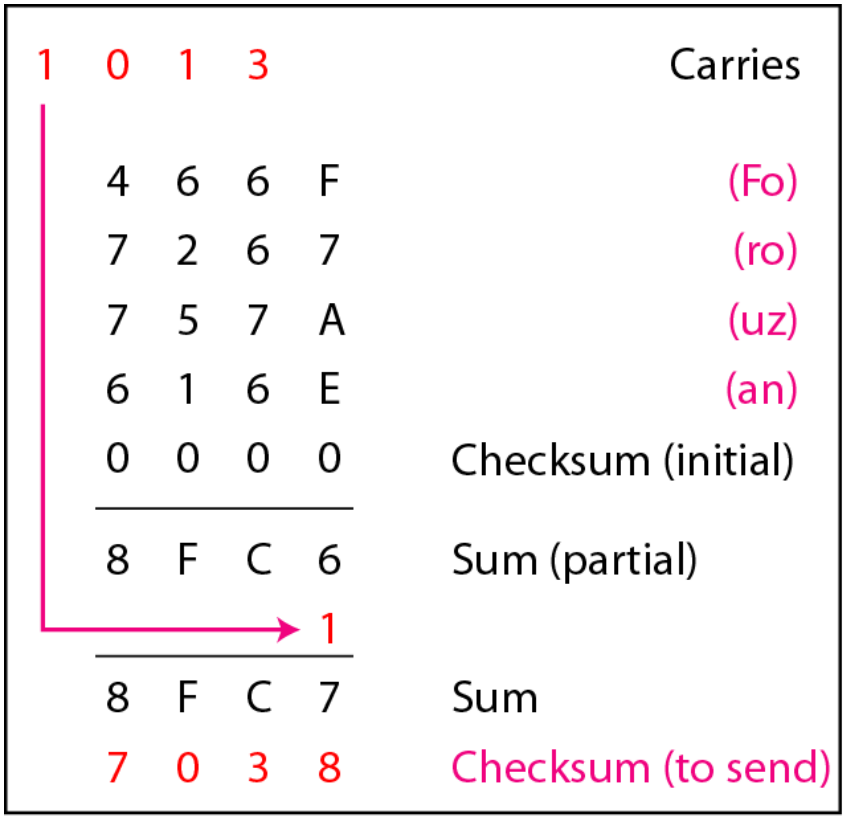
Receiver site:

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.**

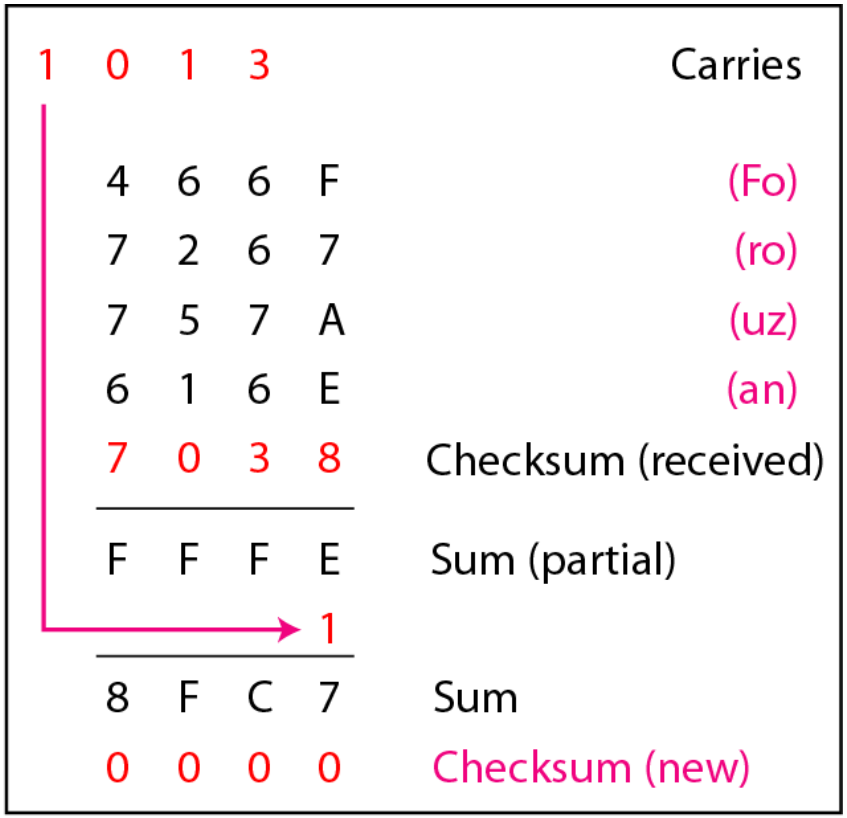
Example 10.23

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*



a. Checksum at the sender site



a. Checksum at the receiver site