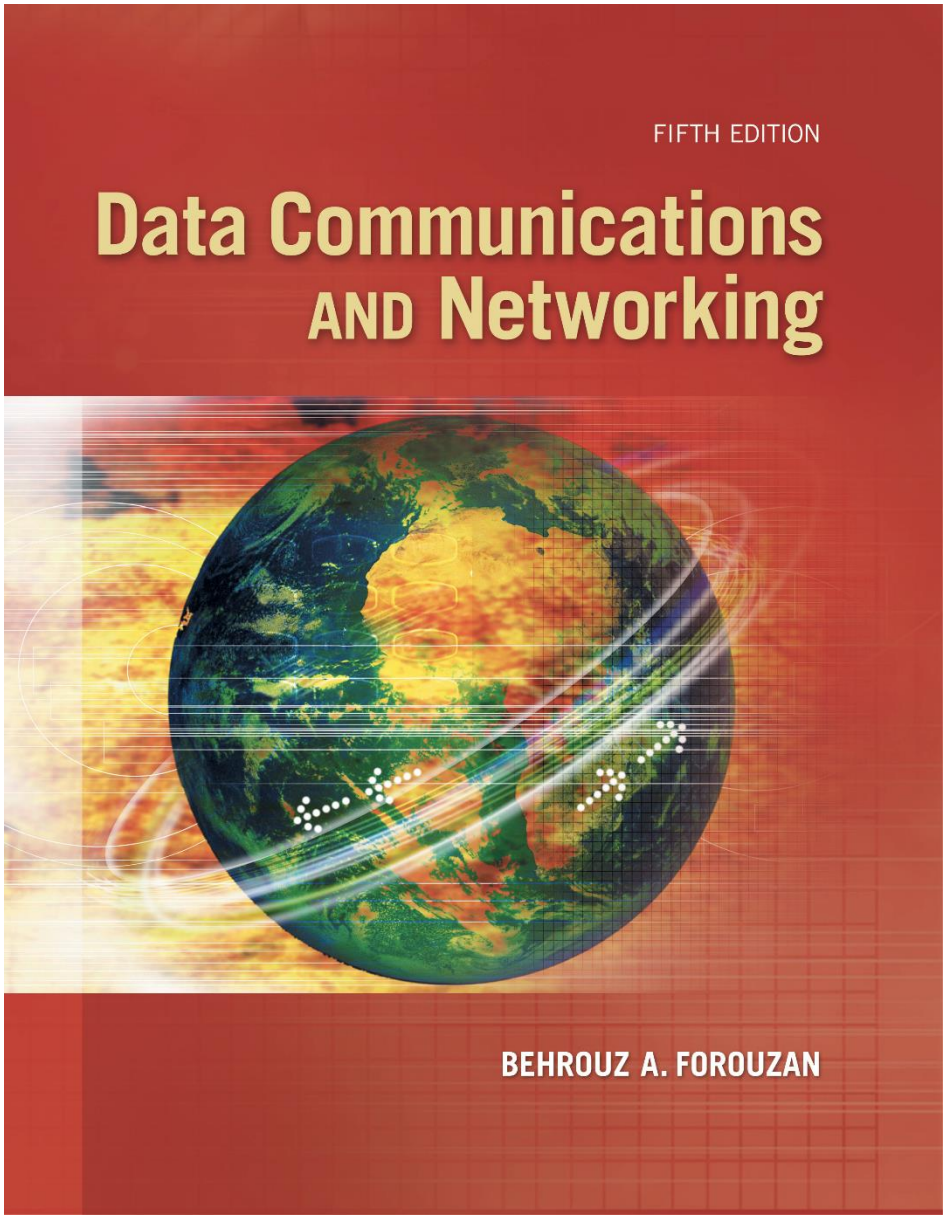
Chapter 10

Error Detection And Correction



Chapter 10: Outline

10.1 INTRODUCTION

10.2 BLOCK CODING

10.3 CYCLIC CODES

10.4 CHECKSUM

10.5 FORWARD ERROR CORRECTION

Chapter 10: Objective

- *The first section introduces types of errors, the concept of redundancy, and distinguishes between error detection and correction.*

- *The second section discusses block coding. It shows how error can be detected using block coding and also introduces the concept of Hamming distance.*

- *The third section discusses cyclic codes. It discusses a subset of cyclic code, CRC, that is very common in the data-link layer. The section shows how CRC can be easily implemented in hardware and represented by polynomials.*

Chapter 10: Objective (continued)

- *The fourth section discusses checksums. It shows how a checksum is calculated for a set of data words. It also gives some other approaches to traditional checksum.*

- *The fifth section discusses forward error correction. It shows how Hamming distance can also be used for this purpose. The section also describes cheaper methods to achieve the same goal, such as XORing of packets, interleaving chunks, or compounding high and low resolutions packets.*

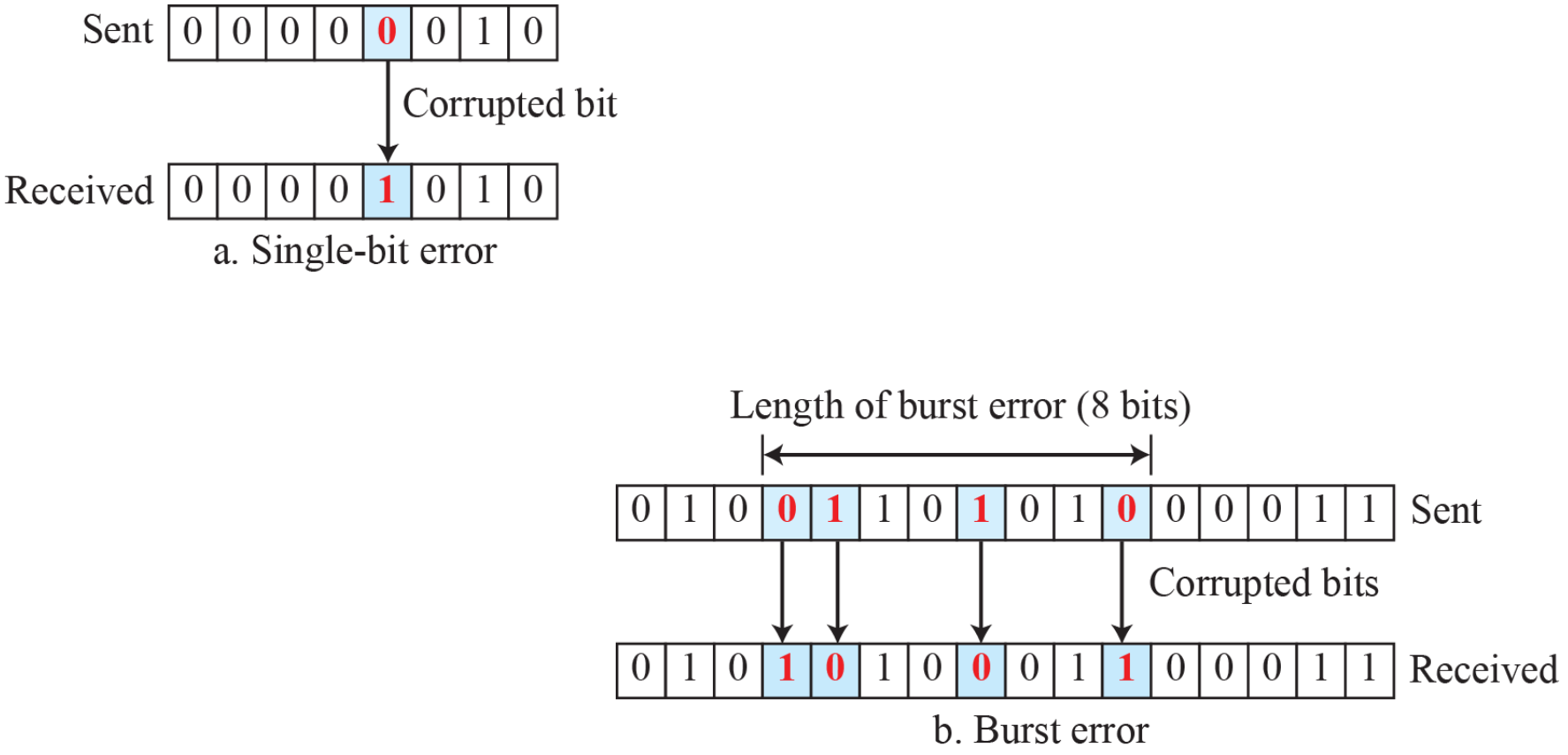
10-1 INTRODUCTION

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

10.10.1 Types of Errors

Whenever bits flow from one point to another, they are subject to unpredictable changes because of interference. This interference can change the shape of the signal. The term single-bit error means that only 1 bit of a given data unit (such as a byte, character, or packet) is changed from 1 to 0 or from 0 to 10. The term burst error means that 2 or more bits in the data unit have changed from 1 to 0 or from 0 to 10. Figure 10.1 shows the effect of a single-bit and a burst error on a data unit.

Figure 10.1: Single-bit and burst error



10.10.2 Redundancy

The central concept in detecting or correcting errors is redundancy. To be able to detect or correct errors, we need to send some extra bits with our data. These redundant bits are added by the sender and removed by the receiver. Their presence allows the receiver to detect or correct corrupted bits.

10.10.3 Detection versus Correction

The correction of errors is more difficult than the detection. In error detection, we are only looking to see if any error has occurred. The answer is a simple yes or no. We are not even interested in the number of corrupted bits. 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.

10.10.4 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 errors. The ratio of redundant bits to data bits and the robustness of the process are important factors in any coding scheme.

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. How the extra r bits are chosen or calculated is something we will discuss later.

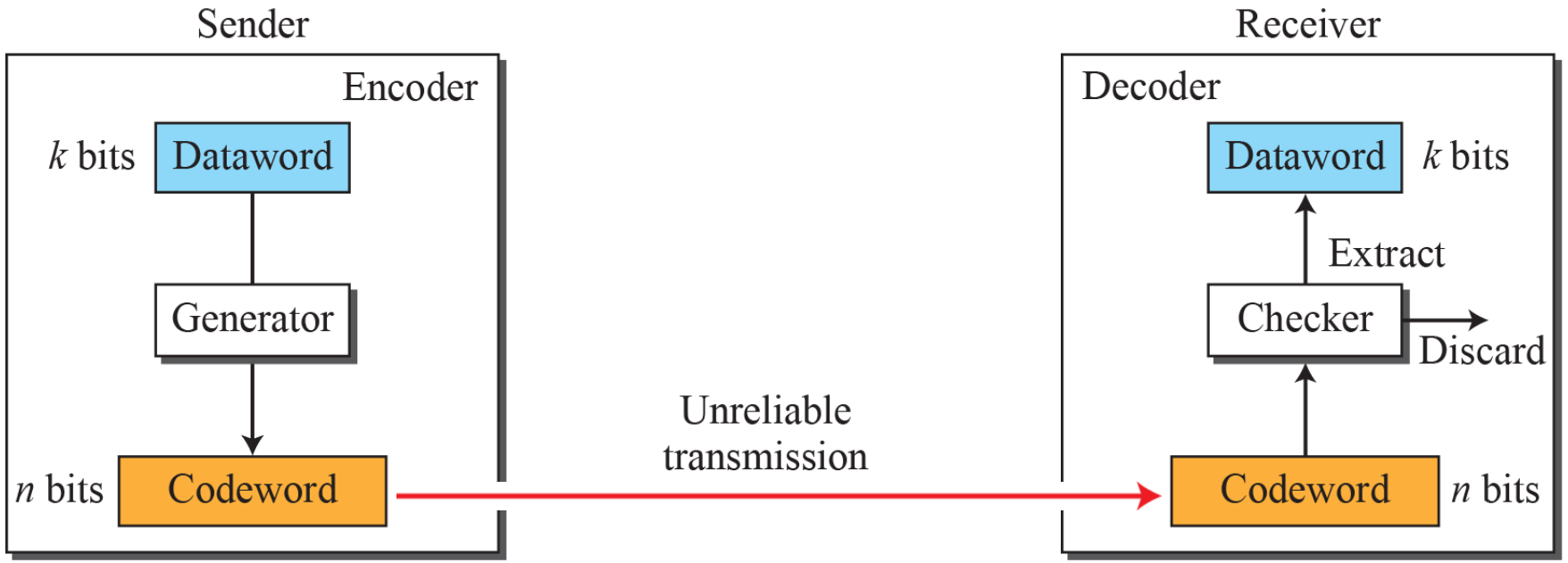
10.2.1 Error Detection

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.

10. The receiver has (or can find) a list of valid codewords.

2. The original codeword has changed to an invalid one.

Figure 10.2: Process of error detection in block coding



Example 10.1

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.

Table 10.1: A code for error detection in Example 10.1

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

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

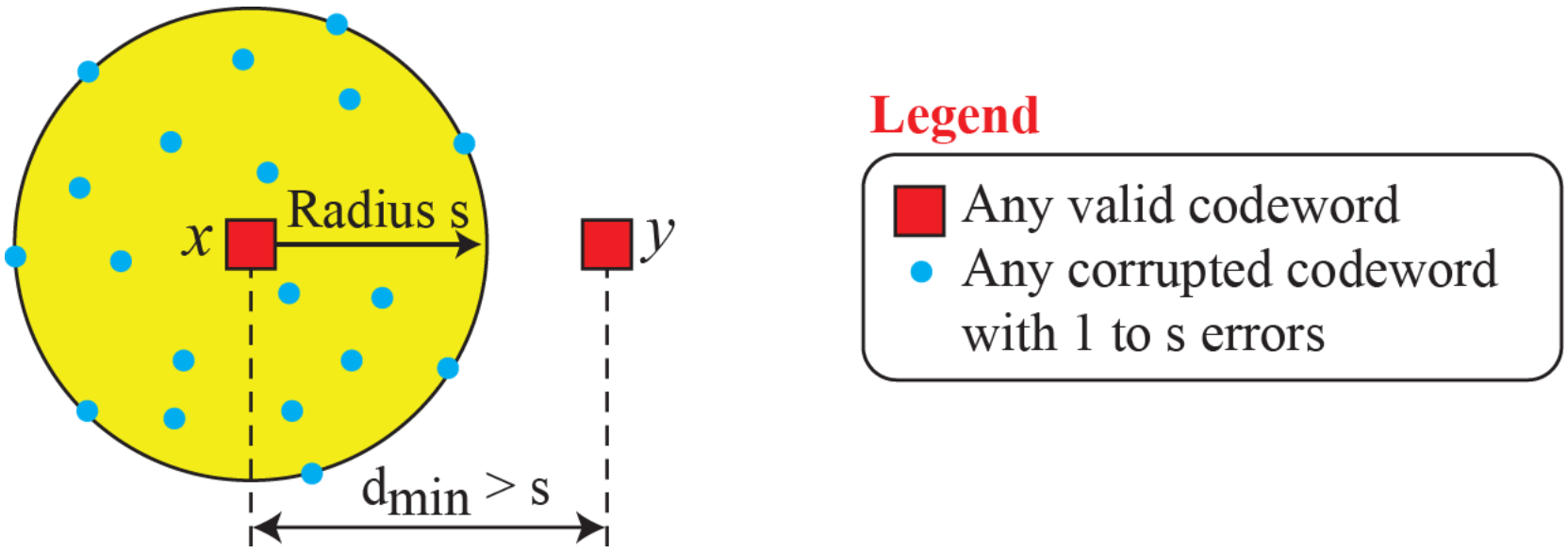
1. The receiver receives 011. It is a valid codeword. The receiver extracts the dataword 01 from it.
2. The codeword is corrupted during transmission, and 111 is received (the leftmost bit is corrupted). This is not a valid codeword and is discarded.
3. The codeword is corrupted during transmission,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

Example 10.2

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).

Figure 10.3: Geometric concept explaining d_{min} in error detection



Example 10.3

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 10.4

A code scheme has a Hamming distance $d_{min} = 4$. This code guarantees the detection of up to three errors ($d = s + 1$ or $s = 3$).

Example 10.5

The code 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.

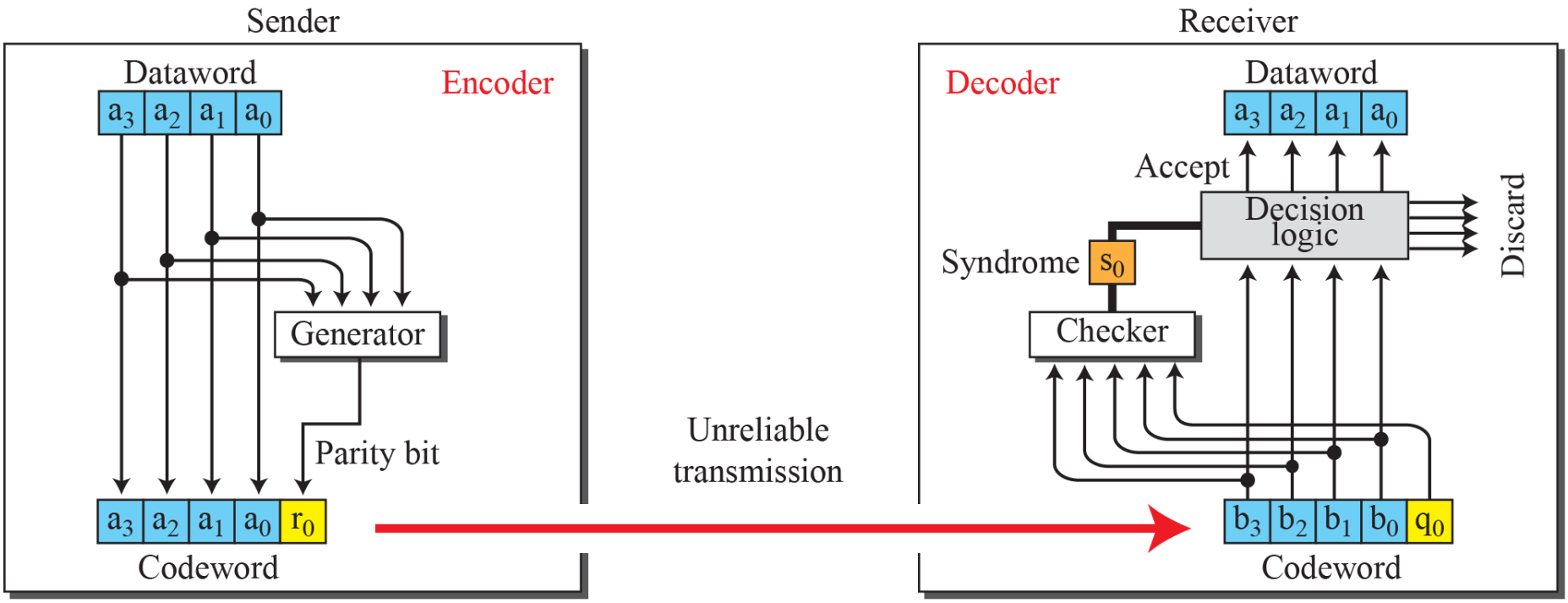
Example 10.6

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 $\boldsymbol{d}_{\text{min}} = 2$.

Table 10.2: 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 10.4: Encoder and decoder for simple parity-check code



Example 10.7

Let us look at some transmission scenarios. Assume the sender sends the dataword 10110. 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. Note that although none of the dataword bits are corrupted, no dataword is created because the code is not sophisticated enough to show the position of the corrupted bit.
4. An error changes r_0 and a second error changes a_3. The received codeword is **0**011**0**. 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. The simple parity-check decoder cannot detect an even number of errors. The errors cancel each other out and give the syndrome a value of 0.
5. Three bits—a_3, a_2, and a_1—are changed by errors. The received codeword is **010**11. 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.

10-3 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. For example, if 1011000 is a codeword and we cyclically left-shift, then 0110001 is also a codeword.

10.3.1 Cyclic Redundancy Check

We can create cyclic codes to correct errors. However, the theoretical background required is beyond the scope of this book. In this section, we simply discuss a subset of. cyclic codes called the cyclic redundancy check (CRC), which is used in networks such as LANs and WANs.

Table 10.3: 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 10.5: CRC encoder and decoder

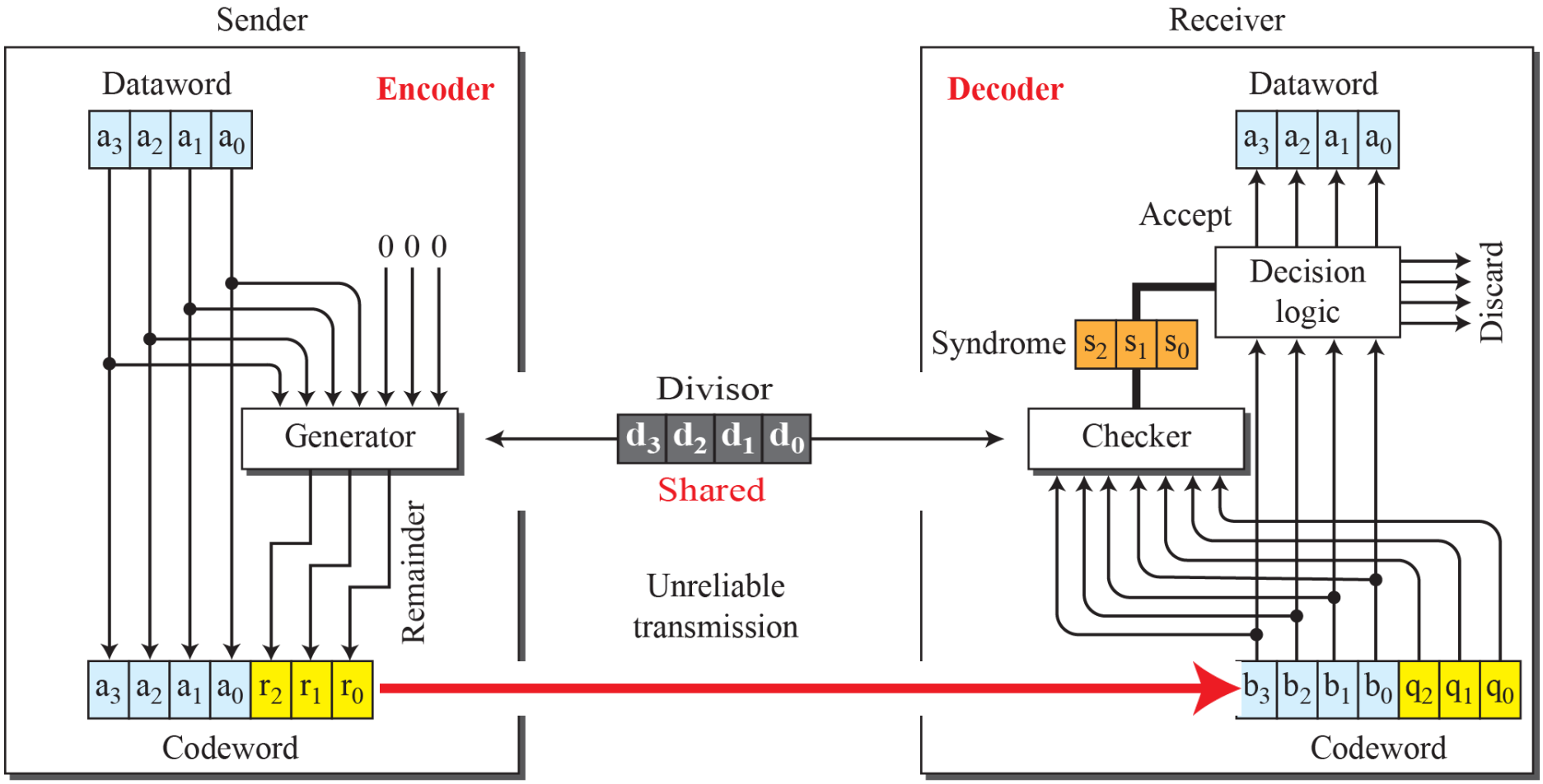


Figure 10.6: Division in CRC encoder

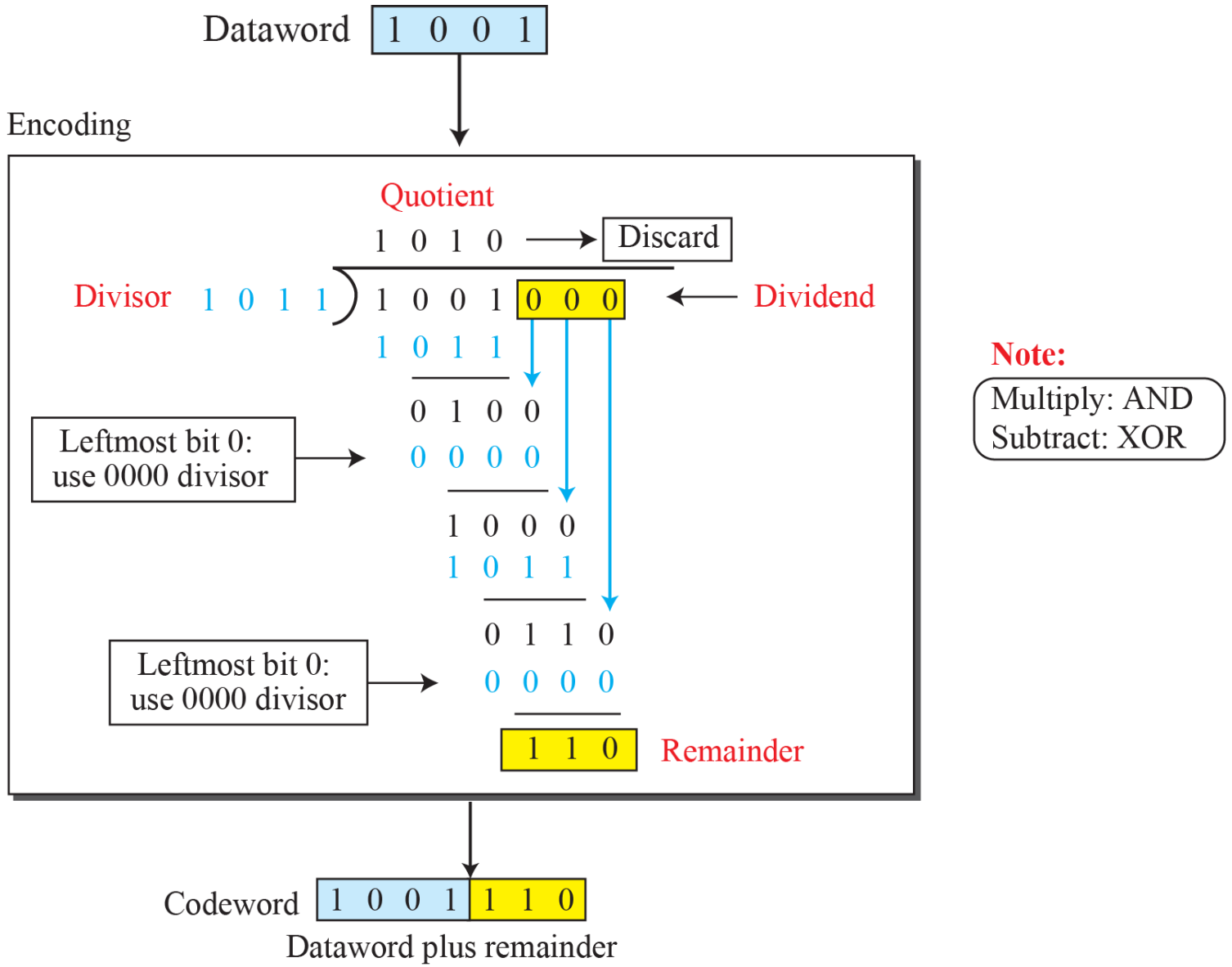
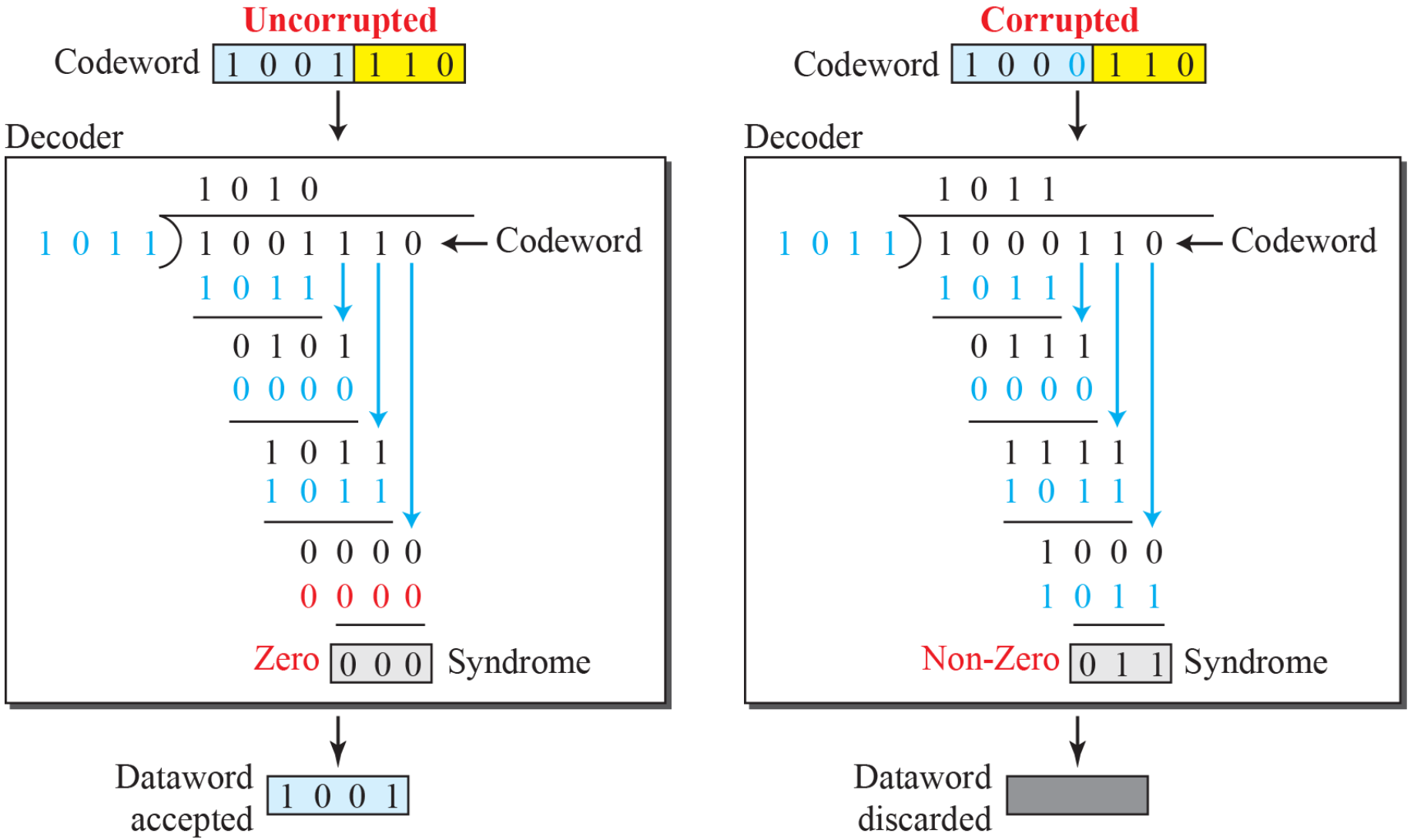


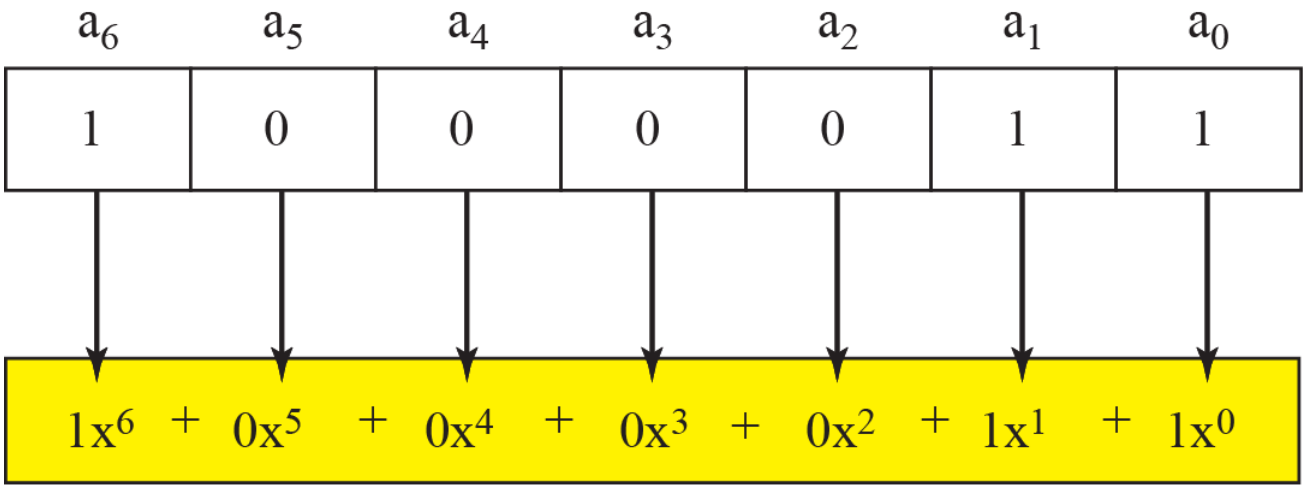
Figure 10.7: Division in the CRC decoder for two cases



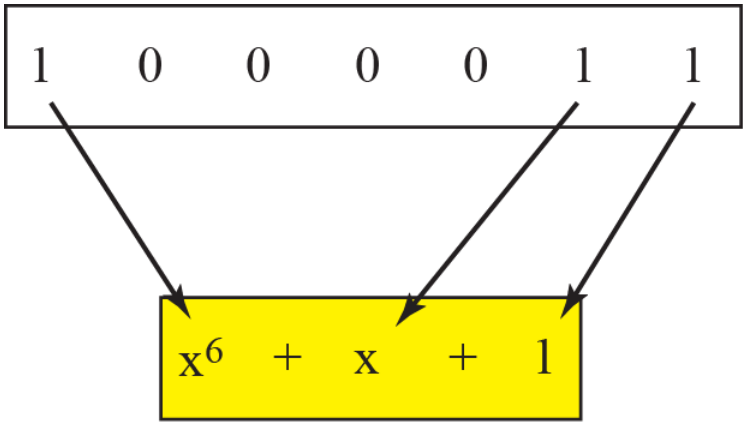
10.3.2 Polynomials

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 10. The power of each term shows the position of the bit; the coefficient shows the value of the bit. Figure 10.8 shows a binary pattern and its polynomial representation.

Figure 10.8: A polynomial to represent a binary word



a. Binary pattern and polynomial

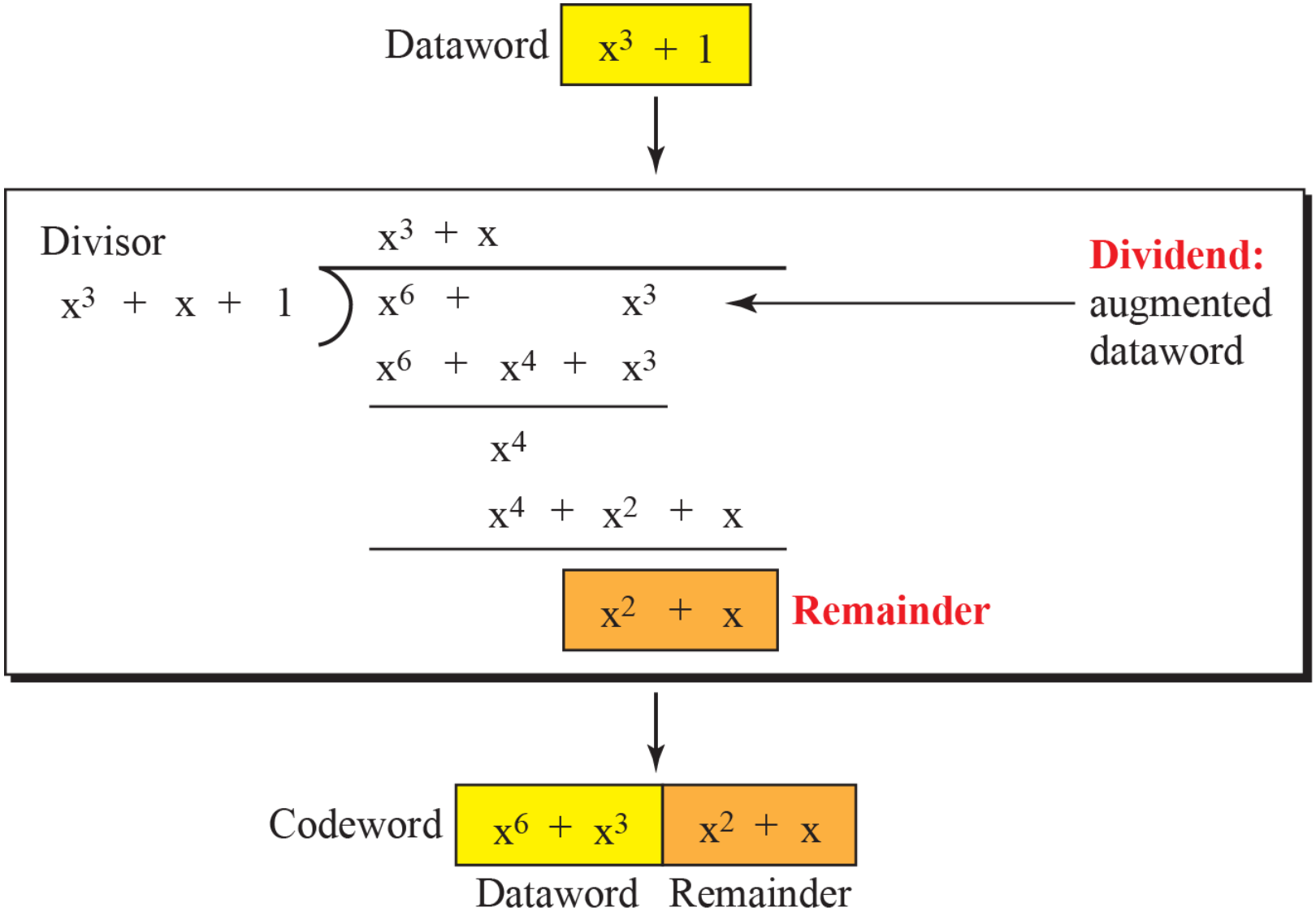


b. Short form

10.3.3 Encoder Using Polynomials

Now that we have discussed operations on polynomials, we show the creation of a codeword from a dataword. Figure 10.9 is the polynomial version of Figure 10.6. We can see that the process is shorter.

Figure 10.9: CRC division using polynomials



10.3.4 Cyclic Code Analysis

We can analyze a cyclic code to find its capabilities by using polynomials. We define the following, where f(x) is a polynomial with binary coefficients.

Dataword: $d(x)$ **Codeword:** $c(x)$

Generator: $g(x)$ **Syndrome:** $s(x)$ **Error:** $e(x)$

Example 10.8

Which of the following g(x) values guarantees that a single-bit error is caught? $x + 1$, x^3 and 1

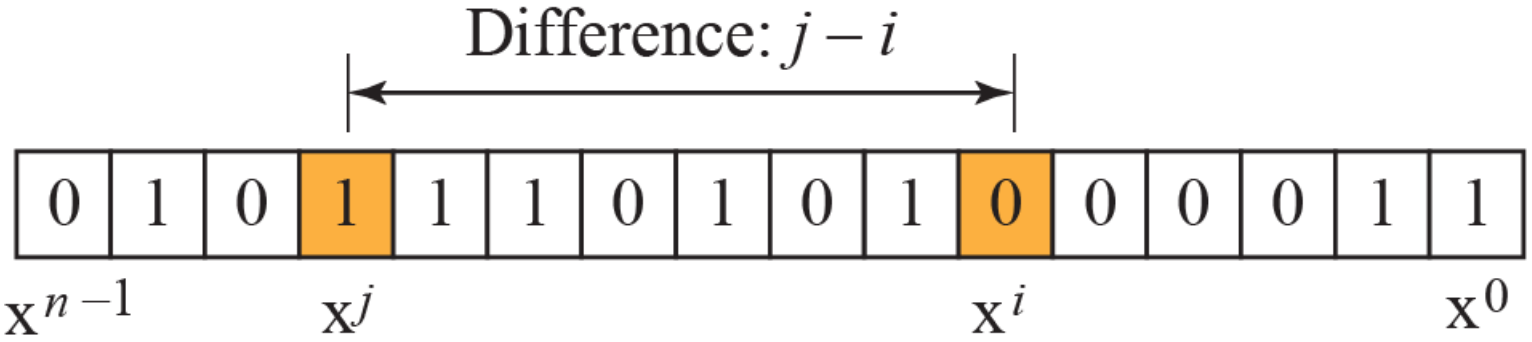
Solution

a. No x^i can be divisible by $x + 1$. In other words, $x^i/(x + 1)$ always has a remainder. So the syndrome is nonzero. Any single-bit error can be caught.

b. If i is equal to or greater than 3, x^i is divisible by $g(x)$. The remainder of x^i/x^3 is zero, and the receiver is fooled into believing that there is no error, although there might be one. Note that in this case, the corrupted bit must be in position 4 or above. 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. In addition, this $g(x)$ is useless because it means the codeword is just the dataword augmented with $n - k$ zeros.

Figure 10.10: Representation of isolated single-bit errors



Example 10.9

Find the suitability of the following generators in relation to burst errors of different lengths: $x^6 + 1$, $x^{18} + x^7 + x + 1$, and $x^{32} + x^{23} + x^7 + 10$.

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.

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.

Example 10.10

Find the status of the following generators related to two isolated, single-bit errors: $x + 1$, $x^4 + 1$, $x^7 + x^6 + 1$, and $x^{15} + x^{14} + 1$

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. The two errors can be anywhere, but if their distance is 4, they remain undetected.

c. This is a good choice for this purpose.

d. This polynomial cannot divide any error of type $x^t + 1$ if t is less than 32,768. This means that a codeword with two isolated errors that are next to each other or up to 32,768 bits apart can be detected by this generator.

Table 10.4: Standard polynomials

Name	*Polynomial*	*Used in*
CRC-8	$x^8 + x^2 + x + 1$ **100000111**	ATM header
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^2 + 1$ **11000110101**	ATM AAL
CRC-16	$x^{16} + x^{12} + x^5 + 1$ **1000100000100001**	HDLC
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ **10000010011000001000111011011011**	LANs

10.3.5 Advantages of Cyclic Codes

We have seen that cyclic codes have a very good performance in detecting single-bit errors, double errors, an odd number of errors, and burst errors. They can easily be implemented in hardware and software. They are especially fast when implemented in hardware. This has made cyclic codes a good candidate for many networks.

10.3.6 Other Cyclic Codes

The cyclic codes we have discussed in this section are very simple. The check bits and syndromes can be calculated by simple algebra. There are, however, more powerful polynomials that are based on abstract algebra involving Galois fields. These are beyond the scope of this book. One of the most interesting of these codes is the Reed-Solomon code used today for both detection and correction.

10.3.7 Hardware Implementation

One of the advantages of a cyclic code is that the encoder and decoder can easily and cheaply be implemented in hardware by using a handful of electronic devices. Also, a hardware implementation increases the rate of check bit and syndrome bit calculation. In this section, we try to show, step by step, the process. The section, however, is optional and does not affect the understanding of the rest of the chapter.

Figure 10.11: Hand-wired design of the divisor in CRC

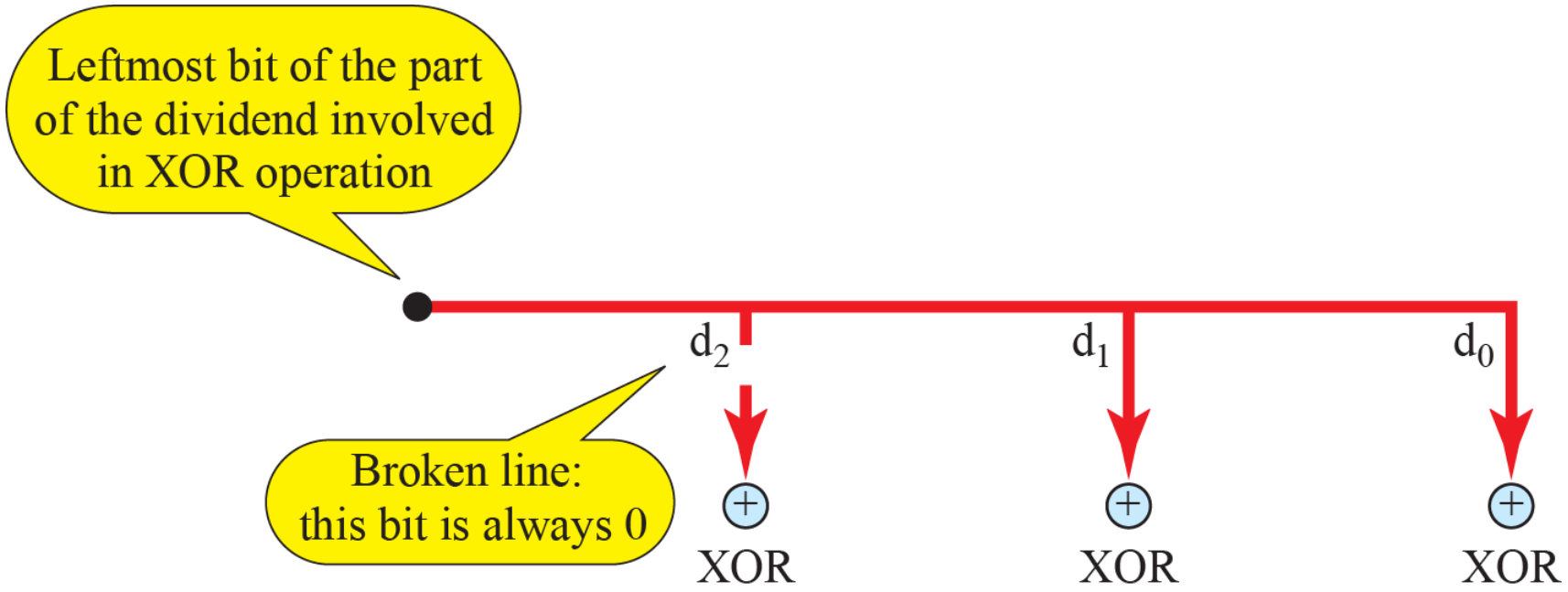


Figure 10.12: Simulation of division in CRC encoder

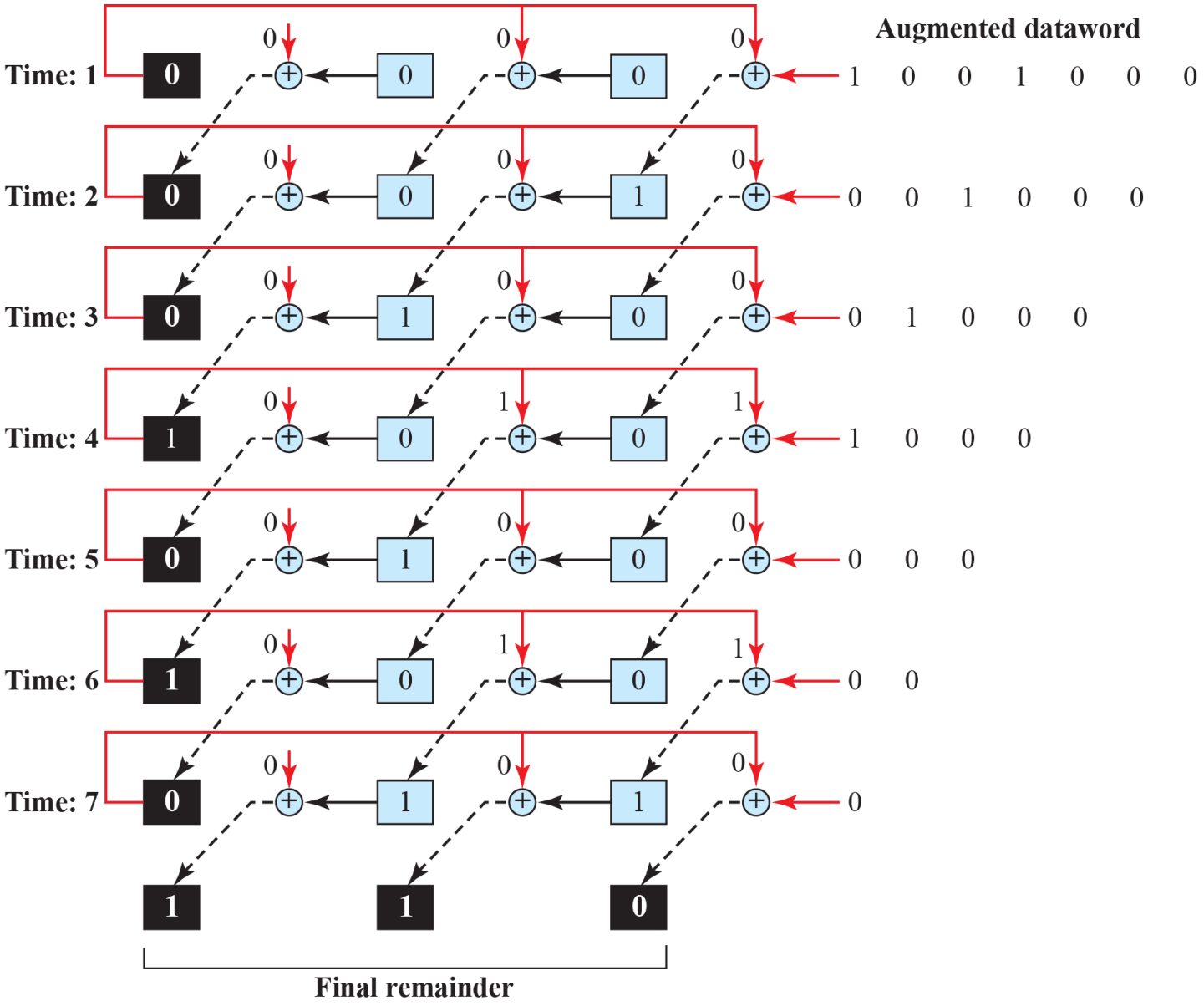


Figure 10.13: CRC encoding design using shift register

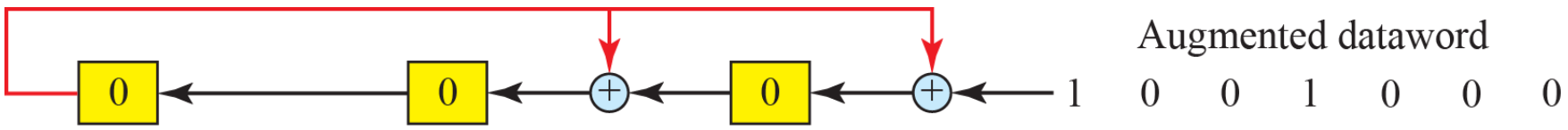
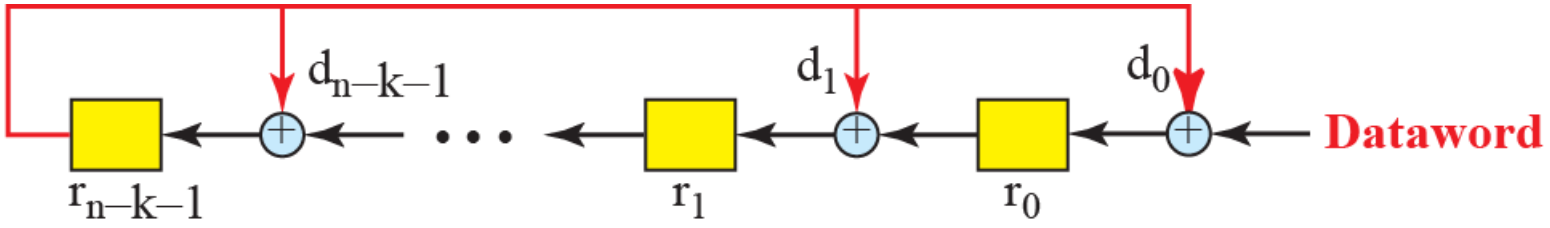


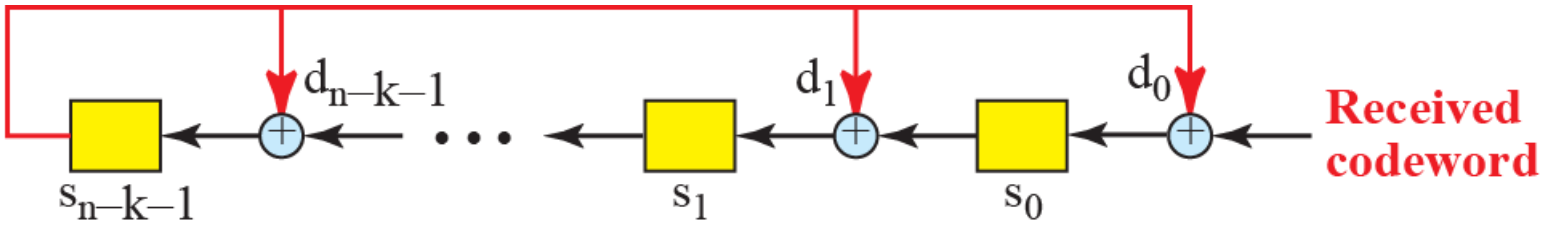
Figure 10.14: General design of encoder and decoder of CRC

Note:

> The divisor line and XOR are missing if the corresponding bit in the divisor is 0.



a. Encoder

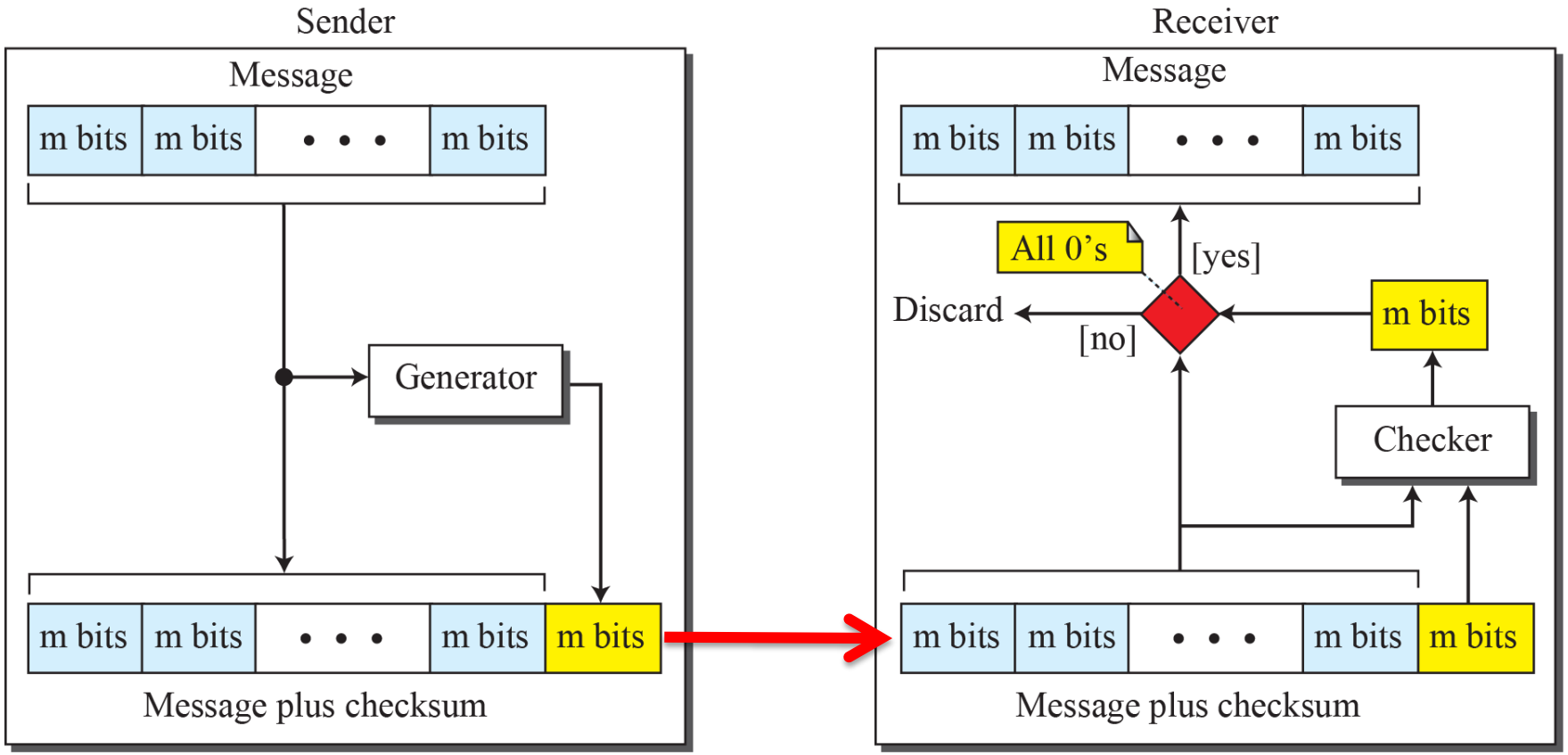


b. Decoder

10-4 CHECKSUM

Checksum is an error-detecting technique that can be applied to a message of any length. In the Internet, the checksum technique is mostly used at the network and transport layer rather than the data-link layer. However, to make our discussion of error detecting techniques complete, we discuss the checksum in this chapter.

Figure 10.15: Checksum



10.4.1 Concept

The idea of the traditional checksum is simple. We show this using a simple example.

Example 10.11

Suppose the message 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 message not accepted.

Example 10.12

In the previous example, the decimal number 36 in binary is (100100)2. To change it to a 4-bit number we add the extra leftmost bit to the right four bits as shown below.

$$(10)_2 + (0100)_2 = (0110)_2 \rightarrow (6)_{10}$$

Instead of sending 36 as the sum, we can send 6 as the sum (7, 11, 12, 0, 6, 6). The receiver can add the first five numbers in one's complement arithmetic. If the result is 6, the numbers are accepted; otherwise, they are rejected.

Example 10.13

Let us use the idea of the checksum in Example 10.12. The sender adds all five numbers in one's complement to get the sum = 6. The sender then complements the result to get the checksum = **9**, which is 15 − 6. Note that 6 = (0110)2 and **9** = (1001)2; they are complements of each other. The sender sends the five data numbers and the checksum (7, 11, 12, 0, 6, **9**). If there is no corruption in transmission, the receiver receives (7, 11, 12, 0, 6, **9**) and adds them in one's complement to get 15 (See Figure 10.16).

Figure 10.16: Example 10.13

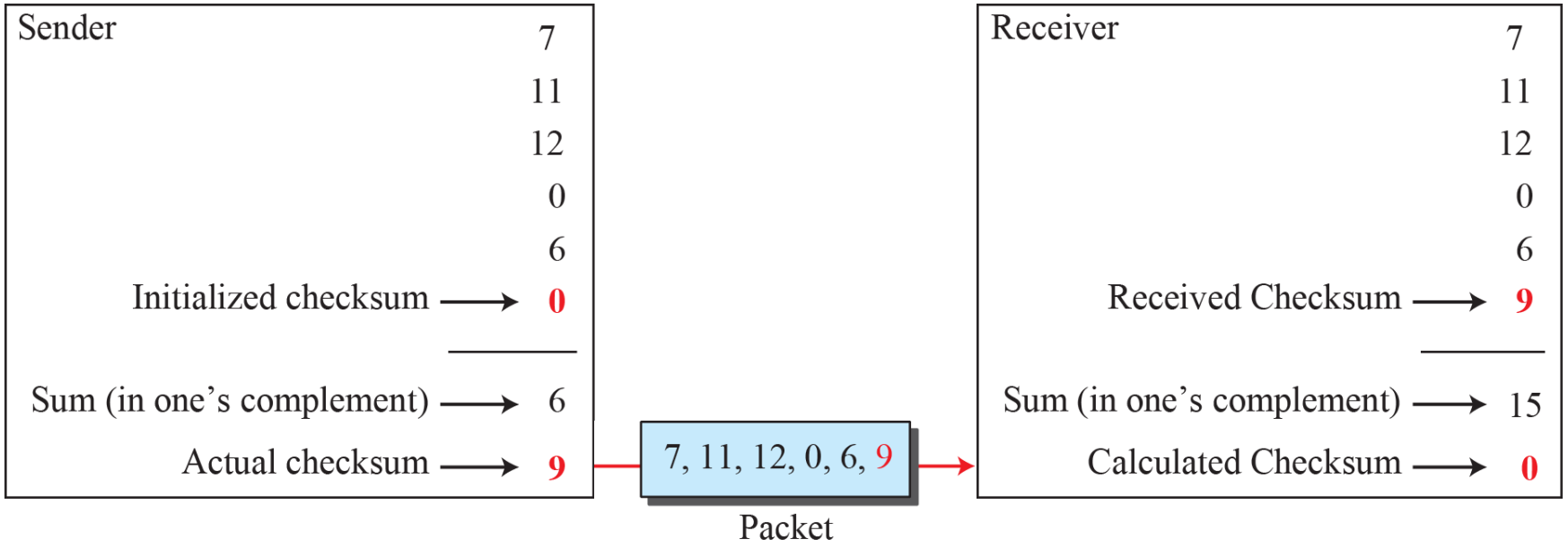


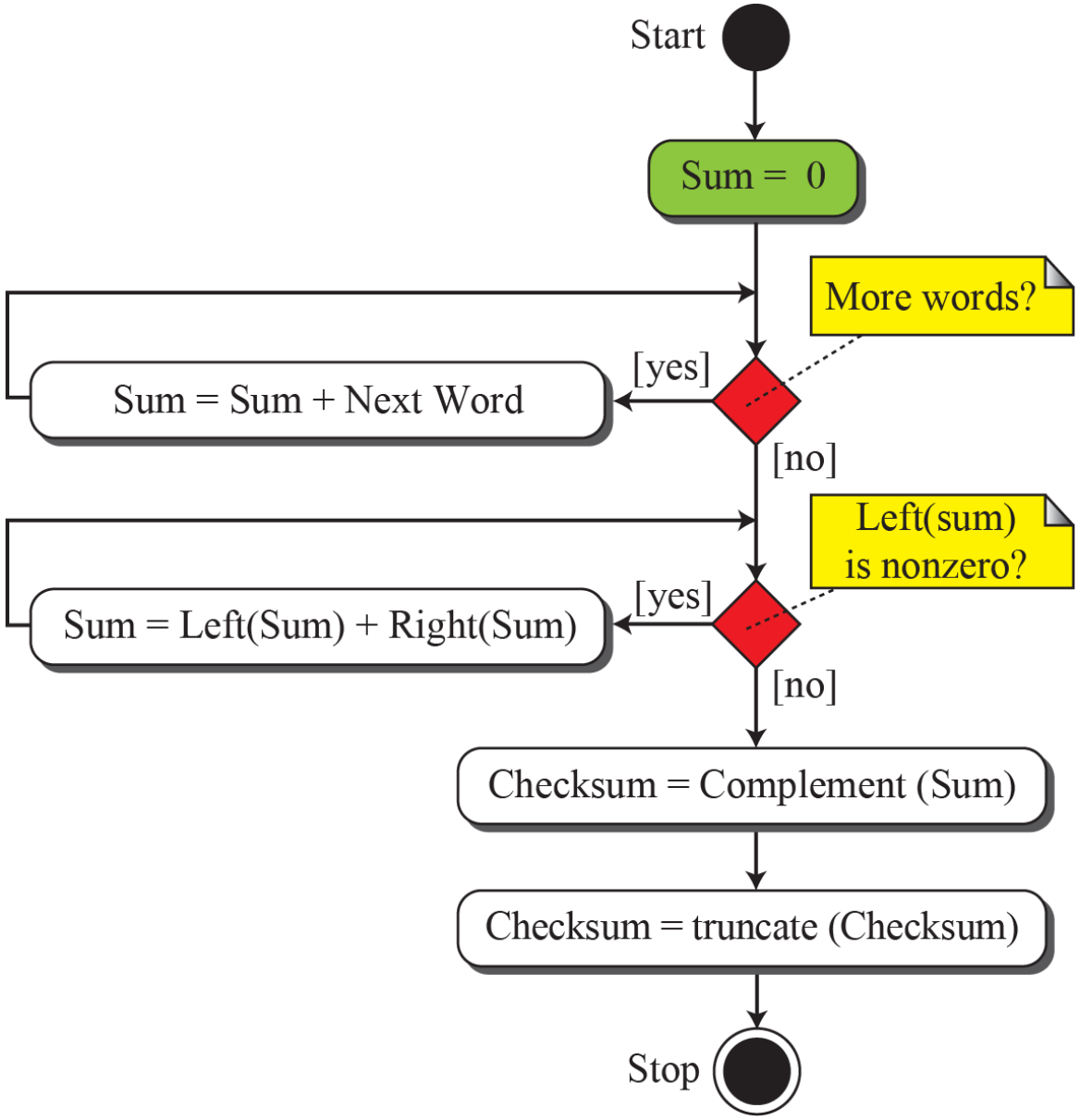
Table 10.5: Procedure to calculate the traditional checksum

Sender	*Receiver*
1. The message is divided into 16-bit words.	1. The message and the checksum is received.
2. The value of the checksum word is initially set to zero.	2. The message is divided into 16-bit words.
3. All words including the checksum are added using one's complement addition.	3. All words are added using one's complement addition.
4. The sum is complemented and becomes the checksum.	4. The sum is complemented and becomes the new checksum.
5. The checksum is sent with the data.	5. If the value of the checksum is 0, the message is accepted; otherwise, it is rejected.

Figure 10.17: Algorithm to calculate a traditional checksum

Notes:

a. Word and Checksum are each 16 bits, but Sum is 32 bits.
b. Left(Sum) can be found by shifting Sum 16 bits to the right.
c. Right(Sum) can be found by ANDing Sum with $(0000FFFF)_{16}$.
d. After Checksum is found, truncate it to 16 bits.



10.4.2 Other Approaches

As mentioned before, there is one major problem with the traditional checksum calculation. If two 16-bit items are transposed in transmission, the checksum cannot catch this error. The reason is that the traditional checksum is not weighted: it treats each data item equally. In other words, the order of data items is immaterial to the calculation. Several approaches have been used to prevent this problem. We mention two of them here: Fletcher and Adler.

Figure 10.18: Algorithm to calculate an 8-bit Fletcher checksum

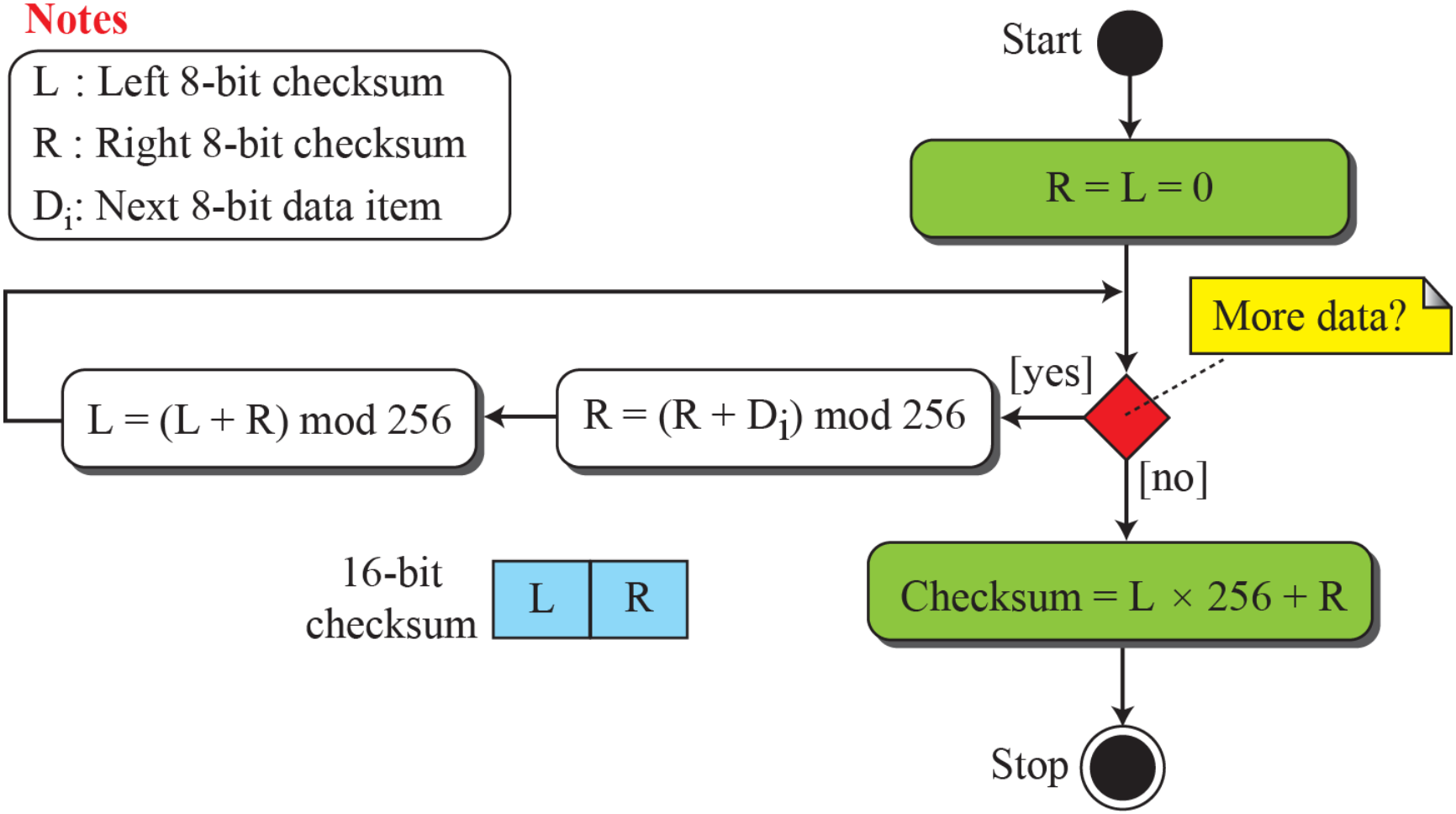
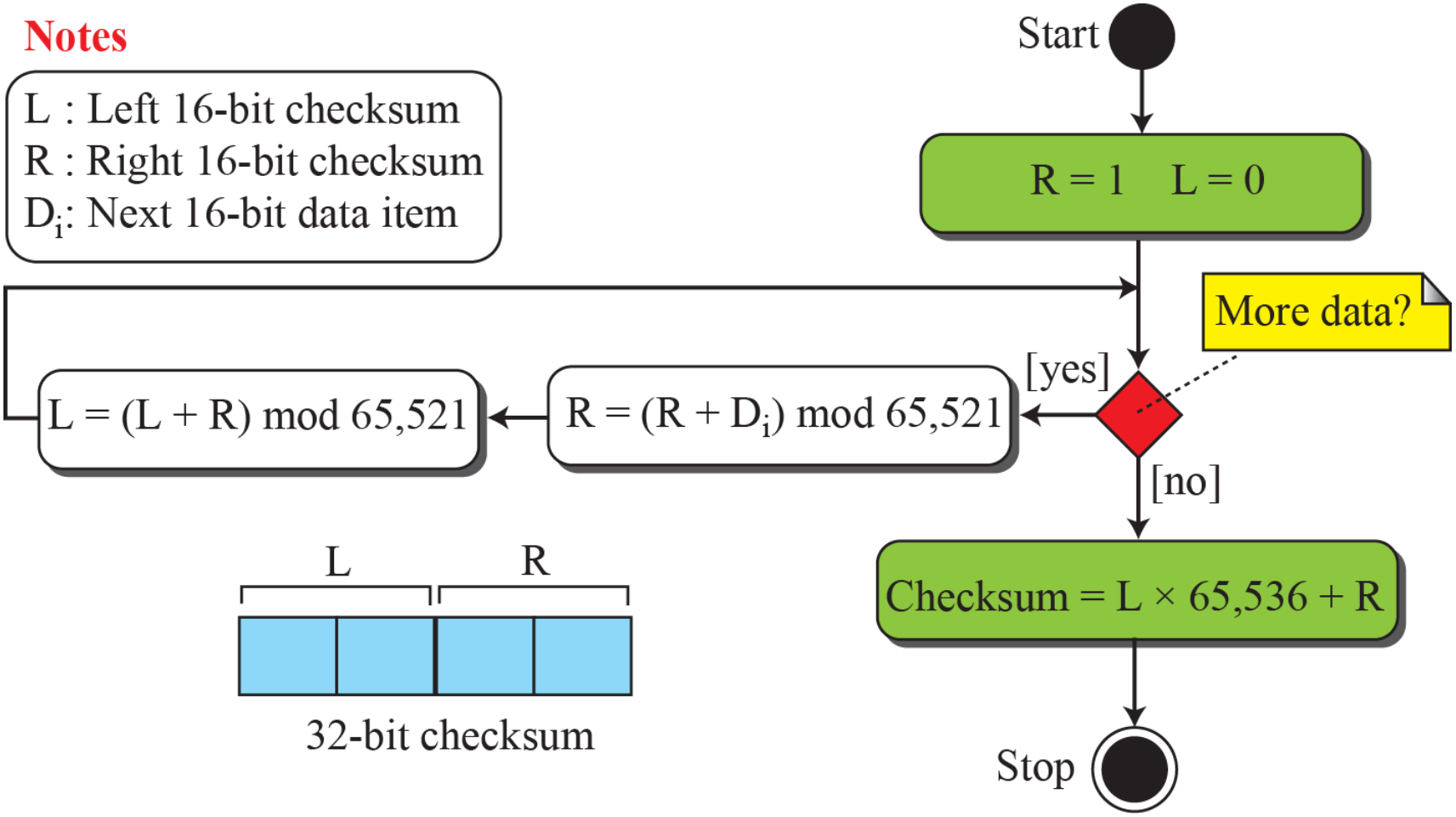


Figure 10.19: Algorithm to calculate an Adler checksum



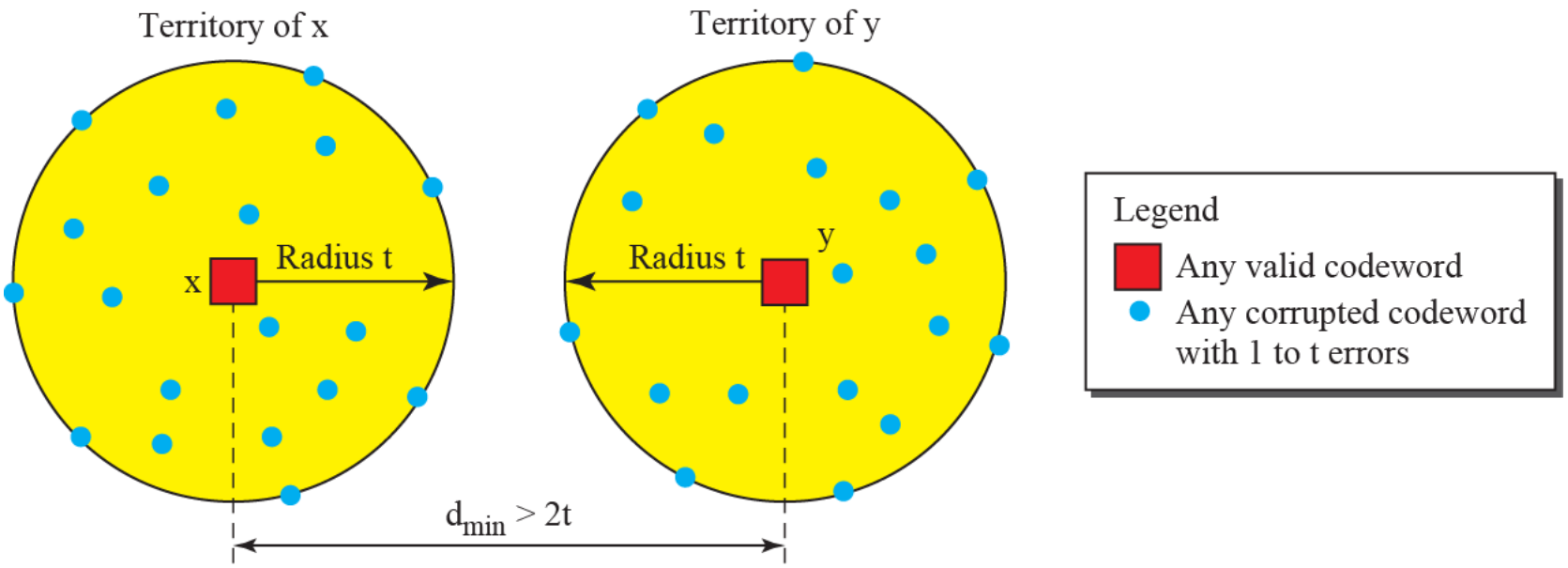
10-5 FORWARD ERROR CORRECTION

We discussed error detection and retransmission in the previous sections. However, retransmission of corrupted and lost packets is not useful for real-time multimedia transmission. We need to correct the error or reproduce the packet immediately.

10.5.1 Using Hamming Distance

We earlier discussed the Hamming distance for error detection. For error detection, we definitely need more distance. It can be shown that to detect t errors, we need to have $d_{min} = 2t + 10$. *In other words, if we want to correct 10 bits in a packet, we need to make the minimum hamming distance 21 bits, which means a lot of redundant bits need to be sent with the data. the geometrical representation of this concept.*

Figure 10.20: Hamming distance for error correction



10.5.2 Using XOR

Another recommendation is to use the property of the exclusive OR operation as shown below.

$$R = P_1 \oplus P_2 \oplus \ldots \oplus P_i \oplus \ldots \oplus P_N$$

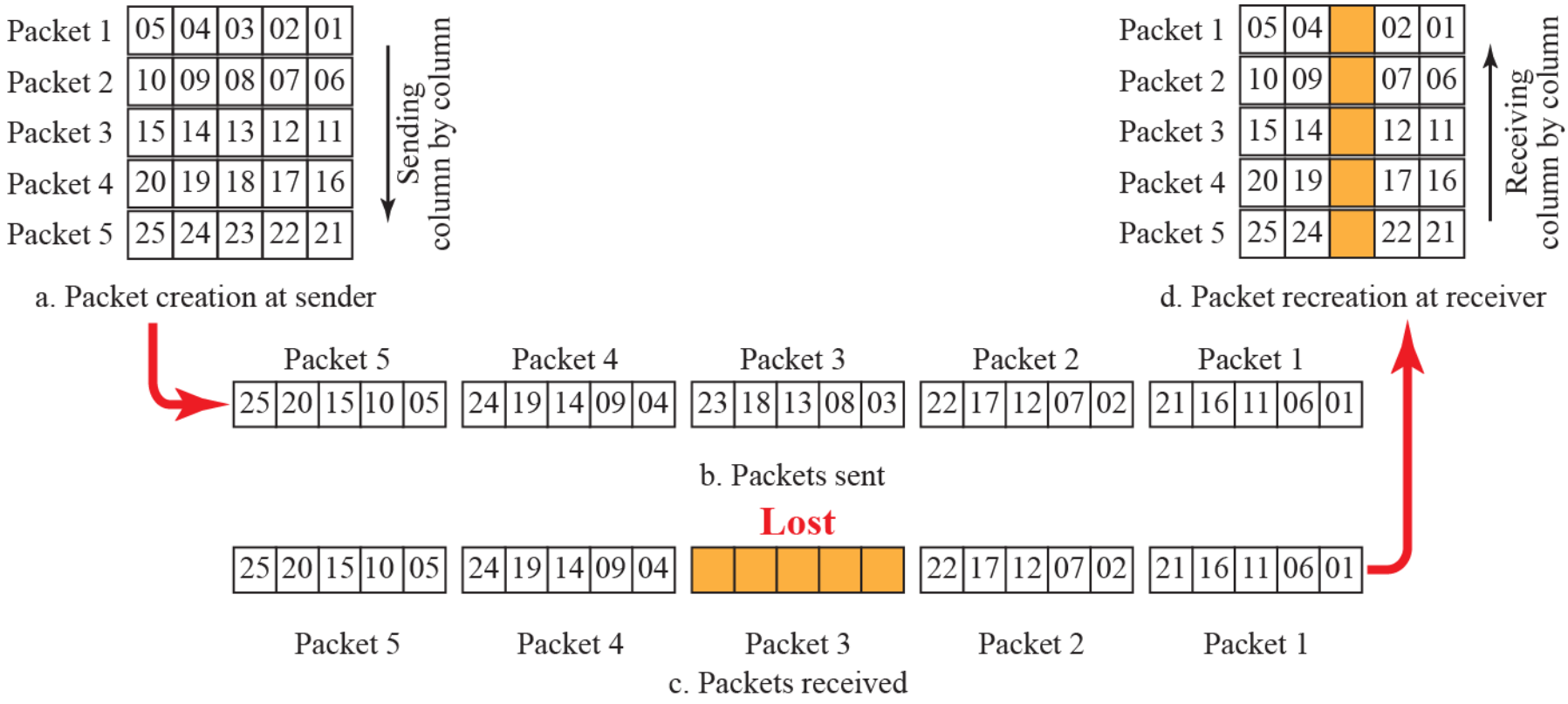
This means:

$$P_i = P_1 \oplus P_2 \oplus \ldots \oplus R \oplus \ldots \oplus P_N$$

10.5.3 Chunk Interleaving

Another way to achieve FEC in multimedia is to allow some small chunks to be missing at the receiver. We cannot afford to let all the chunks belonging to the same packet be missing; however, we can afford to let one chunk be missing in each packet.

Figure 10.21: Interleaving



10.5.4 Combining

Hamming distance and interleaving can be combined. We can first create n-bit packets that can correct t-bit errors. Then we interleave m rows and send the bits column by column. In this way, we can automatically correct burst errors up to m × t bits of errors.

10.5.5 Compounding

Still another solution is to create a duplicate of each packet with a low-resolution redundancy and combine the redundant version with the next packet. For example, we can create four low-resolution packets out of five high-resolution packets and send them as shown in Figure 10.22.

Figure 10.22: Compounding high-and-low resolution packets