Error Detection And Correction

Chapter 10: Outline

10.1 INTRODUCTION

10.2 BLOCK CODING

10.3 CYCLIC CODES

10.4 CHECKSUM

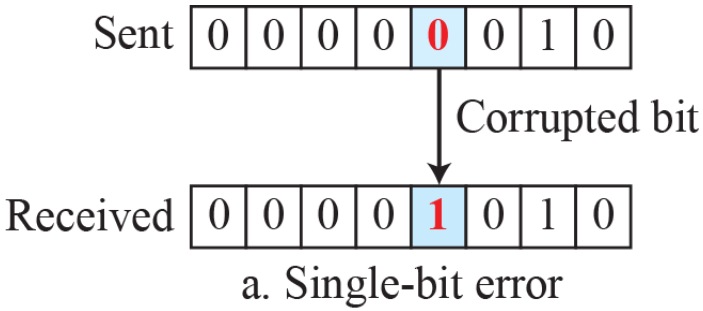
10.5 FORWARD ERROR CORRECTION

10.10.1 Types of Errors

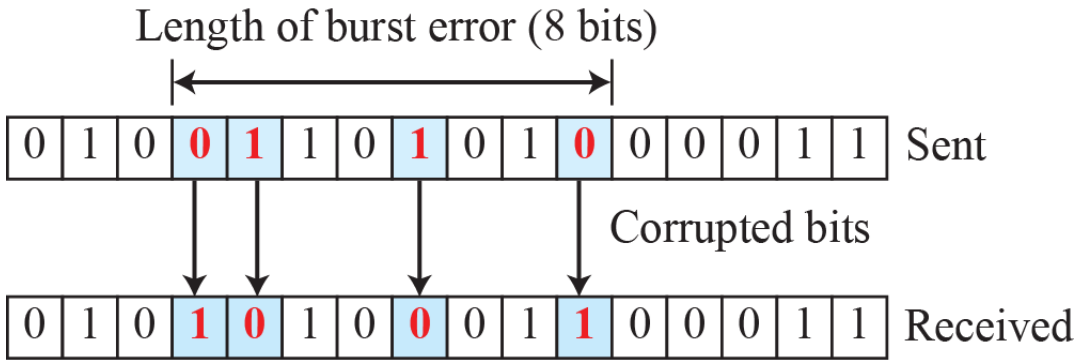
Whenever bits flow from one point to another, they are subject to unpredictable changes because of interference. This interference can change the shape of the signal.

- *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



a. Single-bit error



b. 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.

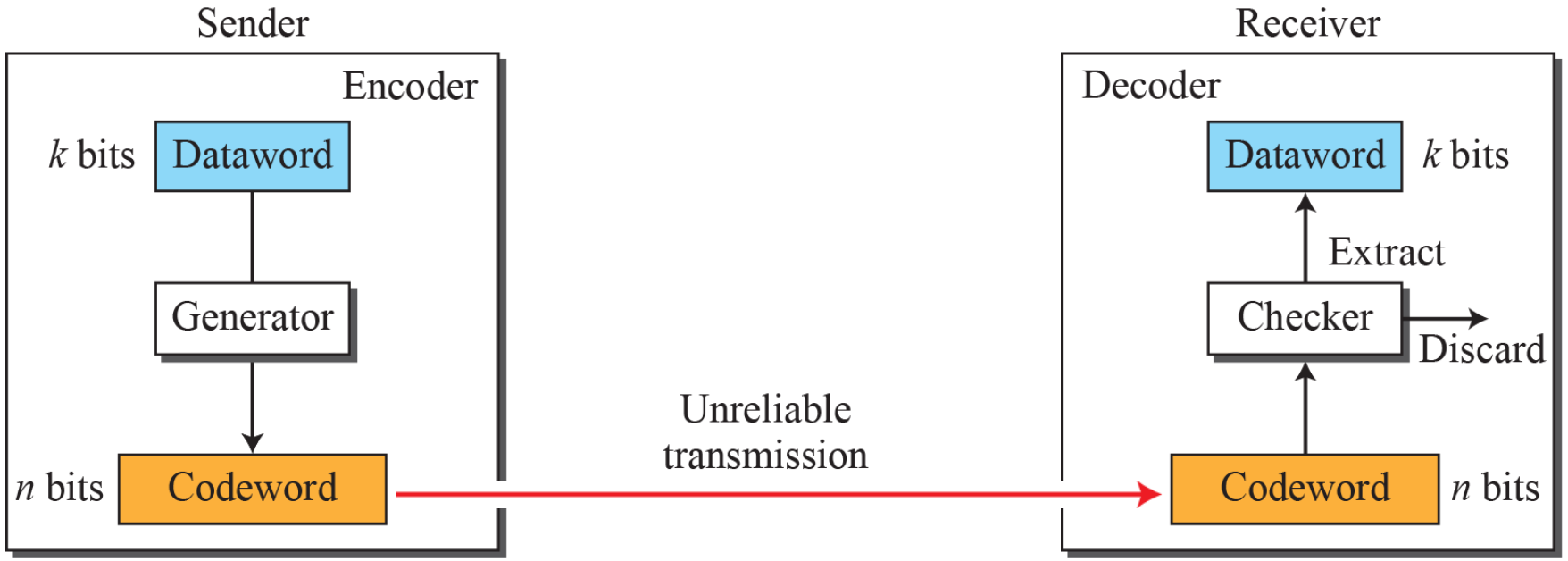
10.2.1 Error Detection

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



Error Detection methods

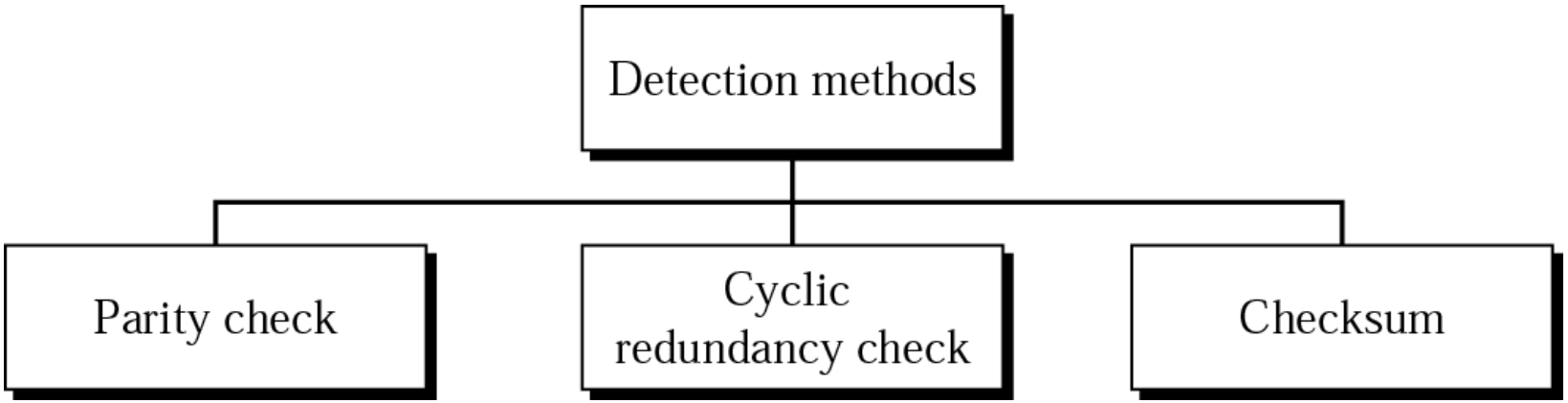
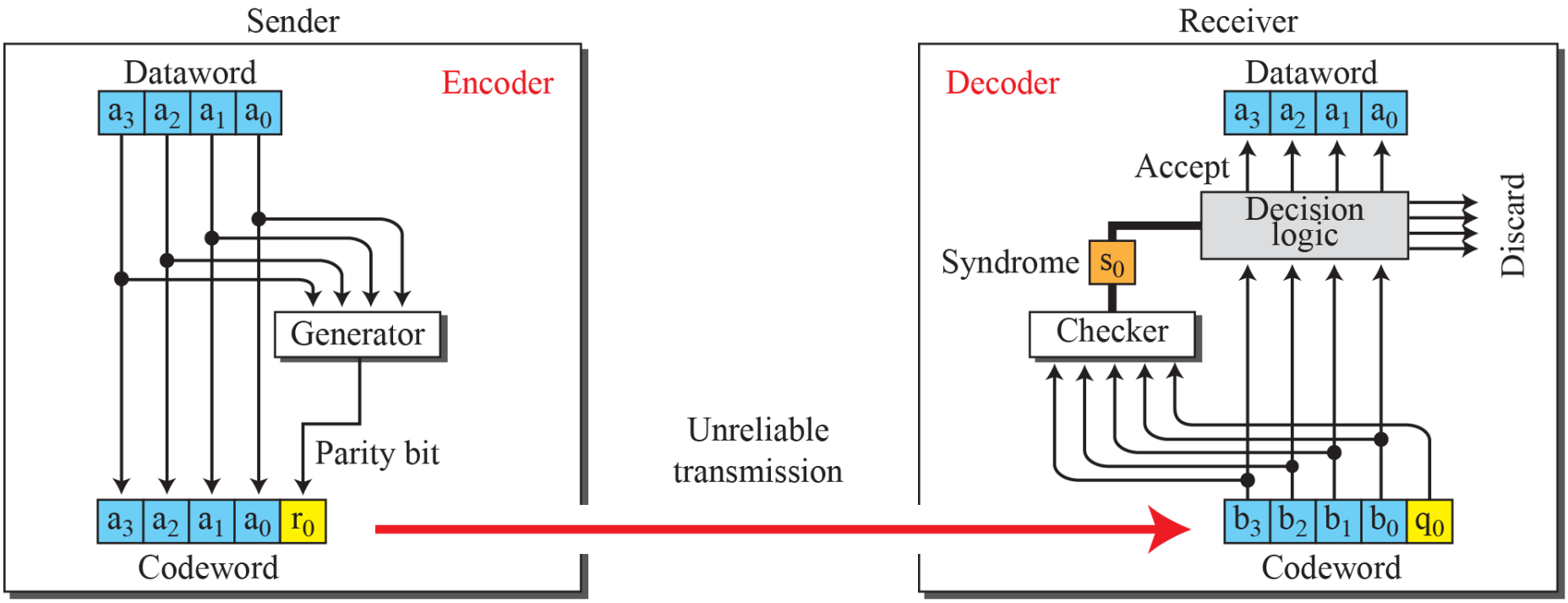


Table 10.2: Simple parity-check code C(5, 4)

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



Error Detection methods

Suppose the sender wants to send the word world. In ASCII the five characters are coded as

W	o	r	l	d
1110111	1101111	1110010	1101100	1100100

The following shows the actual bits sent

11101110 11011110 11100100 11011000 11001001

Error Detection methods

Now suppose the word world in Example 1 is received by the receiver without being corrupted in transmission.

11101110 11011110 11100100 11011000 11001001

The receiver counts the 1s in each character and comes up with even numbers:
(6, 6, 4, 4, 4). The data are accepted.

Error Detection methods

Now suppose the word world in Example 1 is corrupted during transmission.

11111110 11011110 11101100 11011000 11001001

The receiver counts the 1s in each character and comes up with even and odd numbers:
(7, 6, 5, 4, 4).

The receiver knows that the data are corrupted, discards them, and asks for retransmission.

Performance of Simple Parity Check

Example:

We have an even-parity data unit where the total number of 1s, including the parity bit, is 6: 1000111011.

- ***If any 3 bits changed, the resulting parity will be odd and the error will be detected:***
  - ***1111111011: " 9 ones"***
  - ***0110111011: "7 ones"***
  - ***1100010011: " 5 ones"***

- ***The same holds true for any odd number of errors***

Performance of Simple Parity Check

Example:

- Suppose 2 bits are changed:
  - 1110111011: ”8 ones”
  - 1100011011: ”6 ones”
- The number of 1s in the data unit is still even .The same holds true for any even number of errors
- The parity checker cannot detect errors when number of bits changed is even. The change cancel each other and the data unit will pass a parity check even though the data unit is damaged.

Note

Simple parity check can detect all single--bit error. It can detect burst errors only if the total number of errors in each data unit is odd.

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.

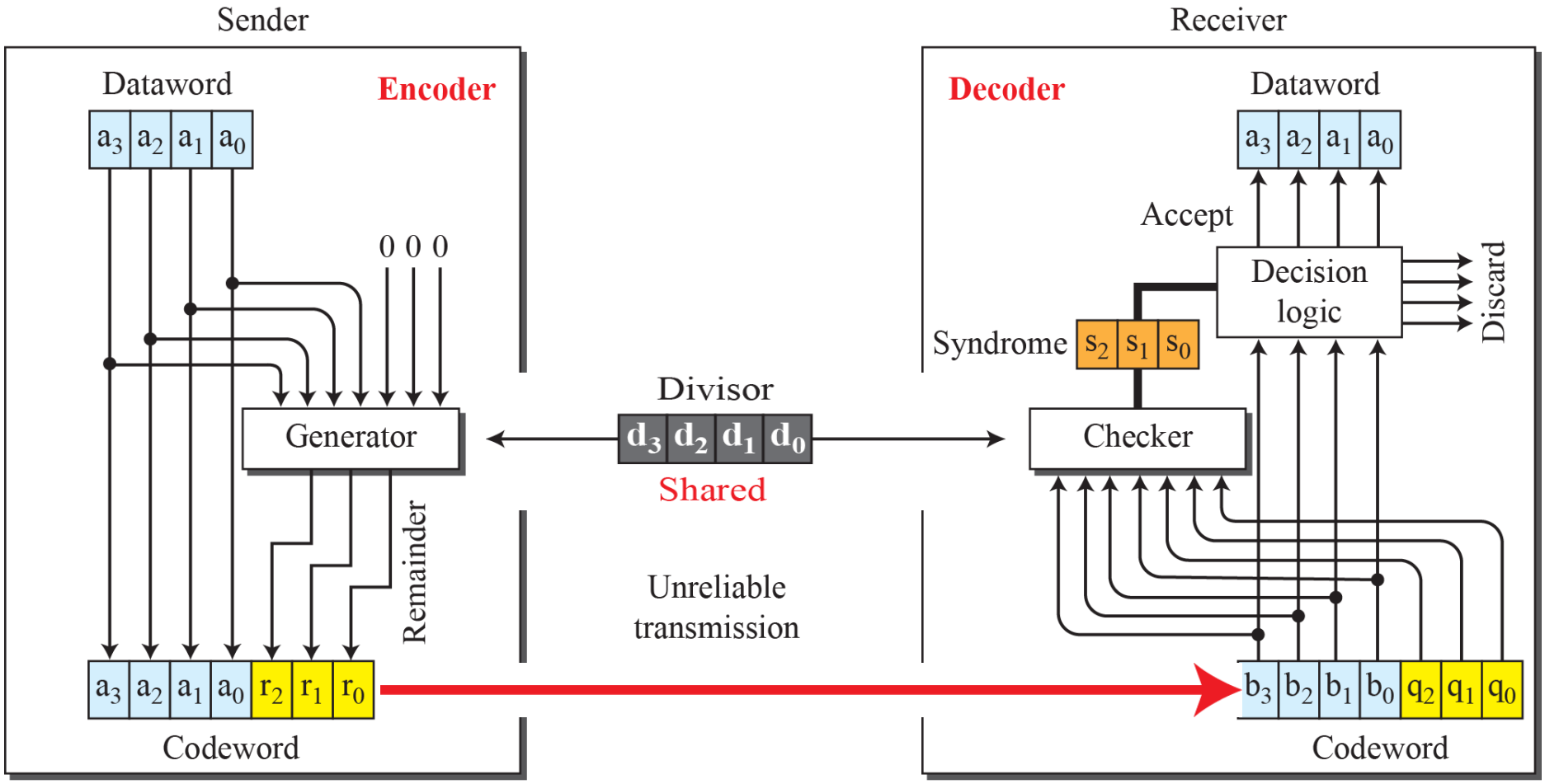
10.3.1 Cyclic Redundancy Check

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



10.3.1 Cyclic Redundancy Check

In CRC generator (At sender)

- *A string of n 0s is appended to the data unit*
- *The number n is 1 less than the number of bits in the divisor*
- *Divide the data word plus appended zeros by the divisor*

Use module-2 binary division:

- *There is no carry when you add or subtract two digits in a column*
- *Addition and subtraction gives the same results*
- *This means: you can use XOR operation for both Addition and subtraction*

- *The remainder resulting from the division is the CRC*
- *The CRC of n bits replaces the appended 0s at the end of the data unit.*
- *Appending CRC to the end of the data must make resulting bit sequence divisible by the divisor*

10.3.1 Cyclic Redundancy Check

In CRC generator (At receiver)

- *After receiving the data appended with the CRC, it does the same module -2 division*
- *If the remainder is all 0s the CRC dropped and the data are accepted (the data is correct)*
- *If the remainder is not equal zero, the received stream of bits is discarded and data must be resent (the data is corrupted)*

Figure 10.6: Division in CRC encoder

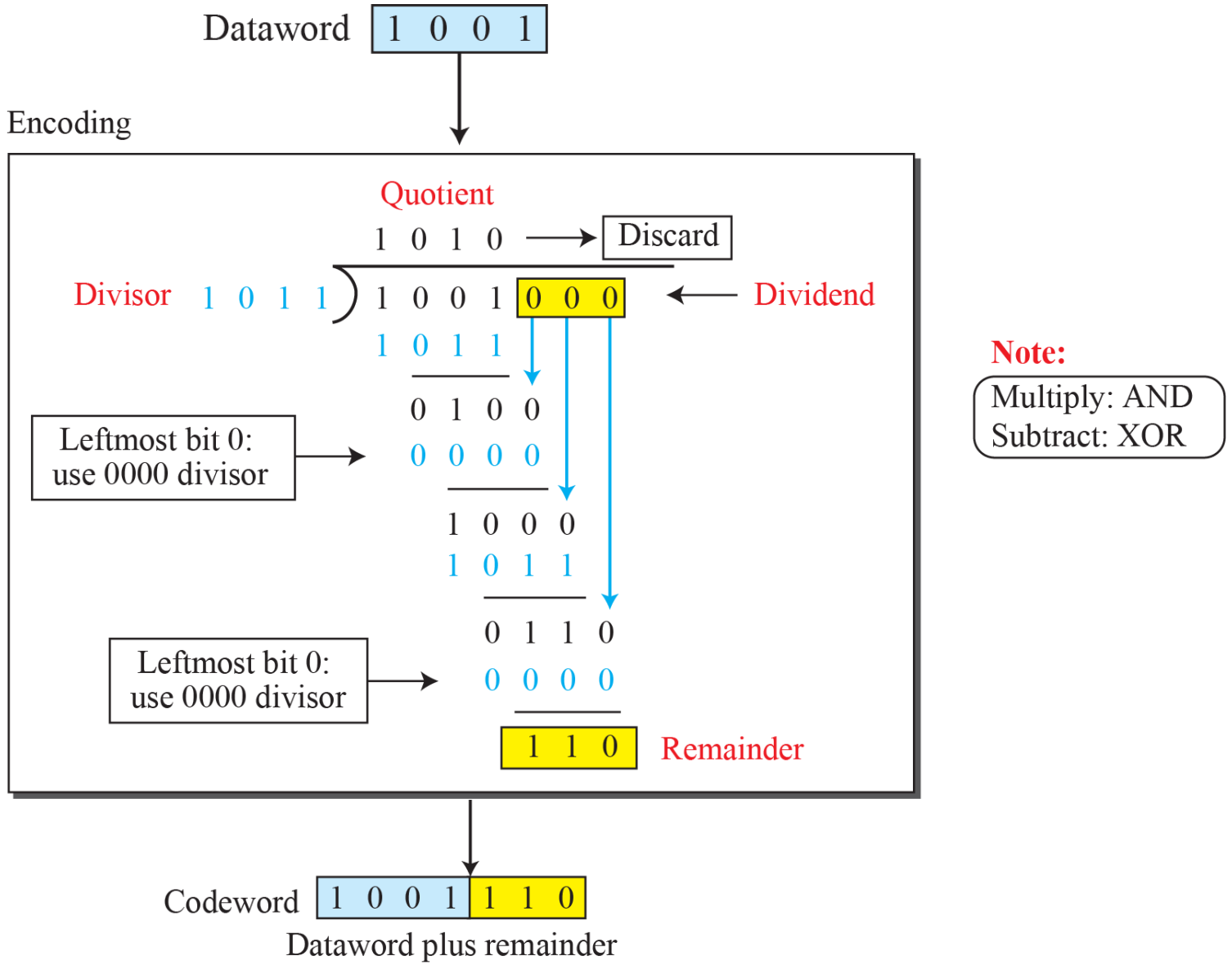
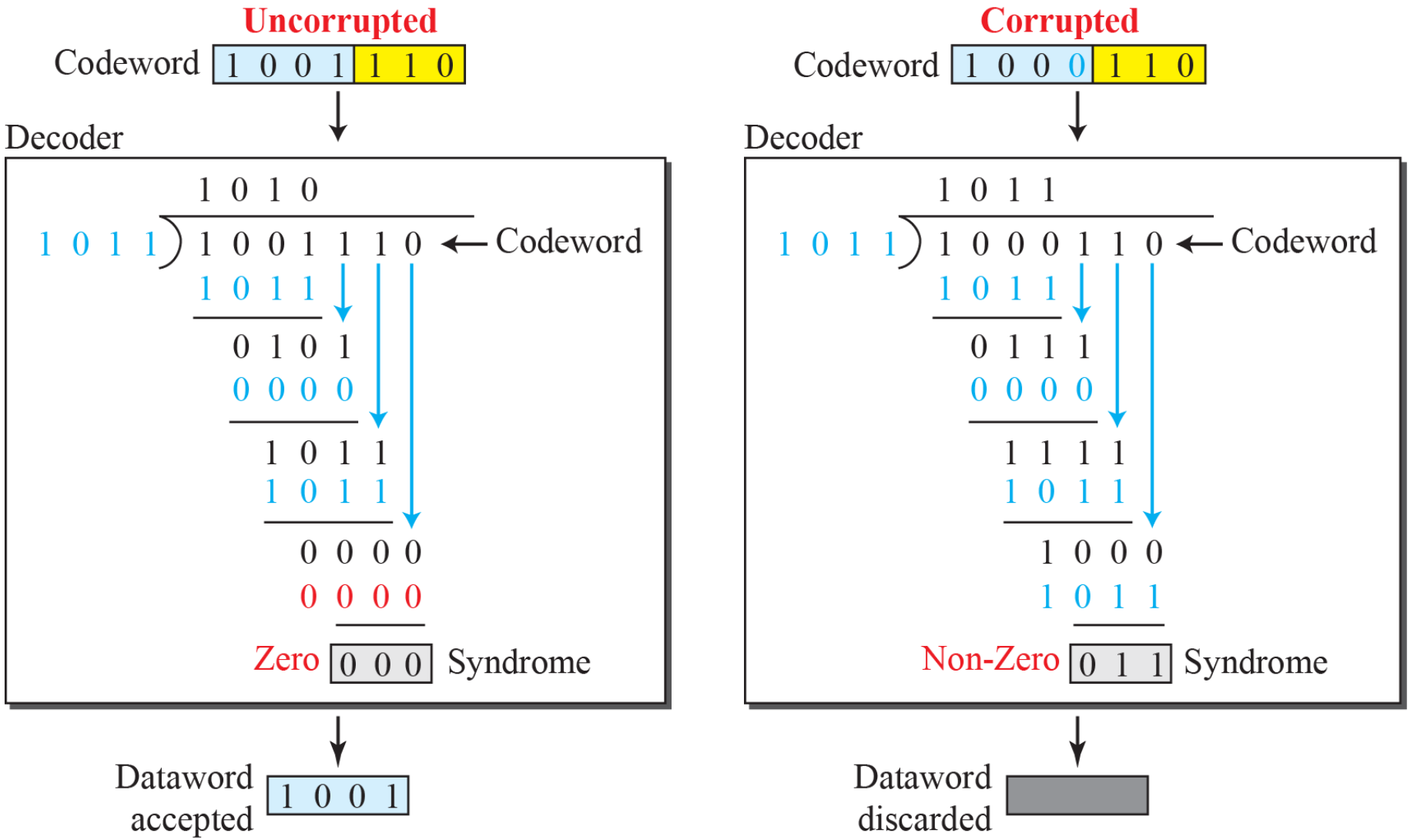


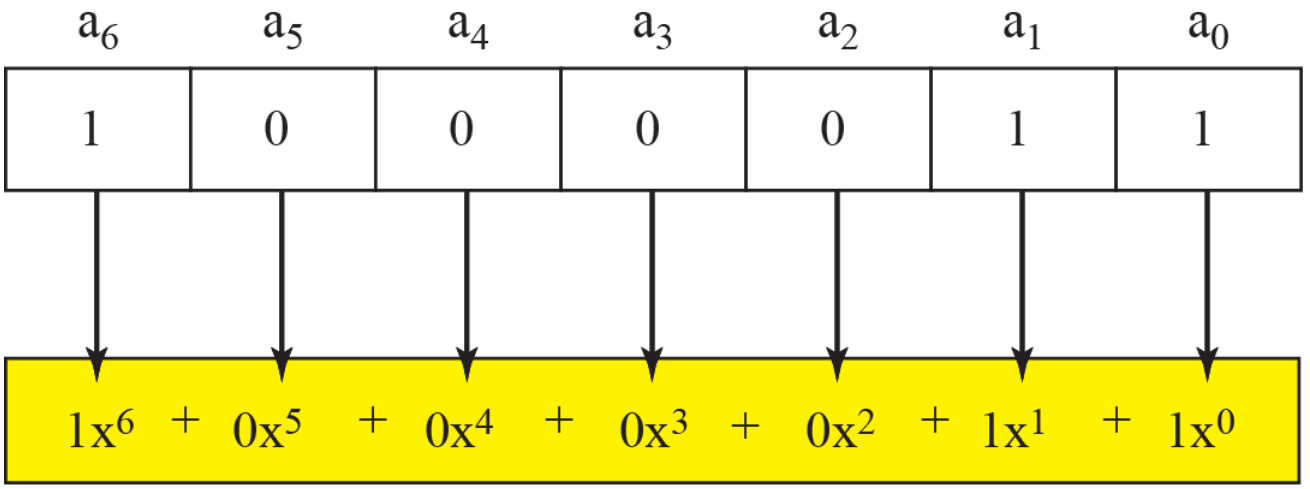
Figure 10.7: Division in the CRC decoder for two cases



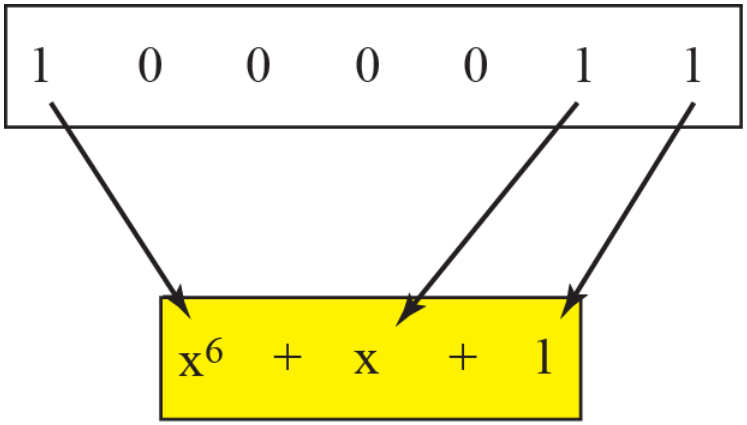
10.3.2 Polynomials

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

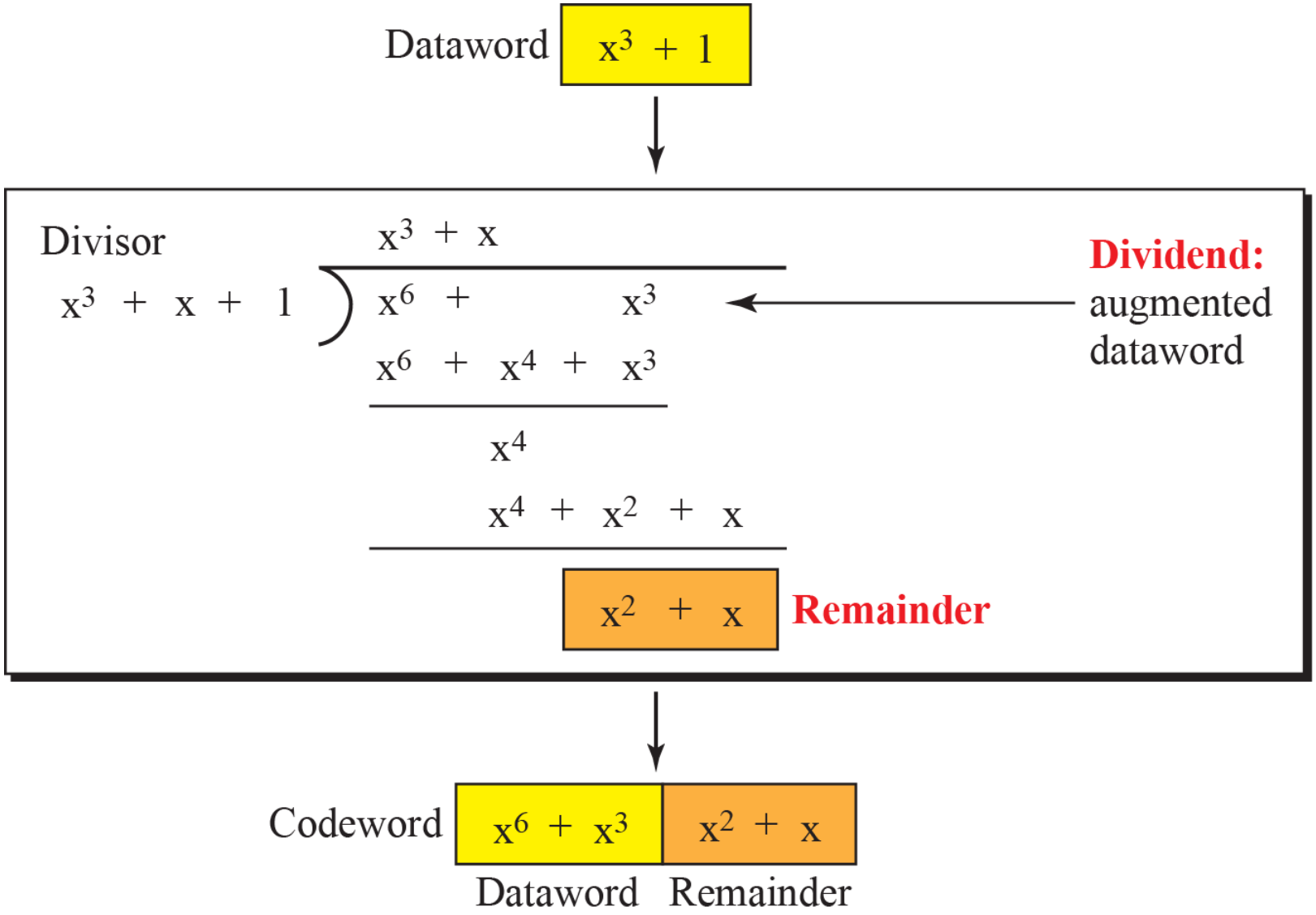


b. Short form

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



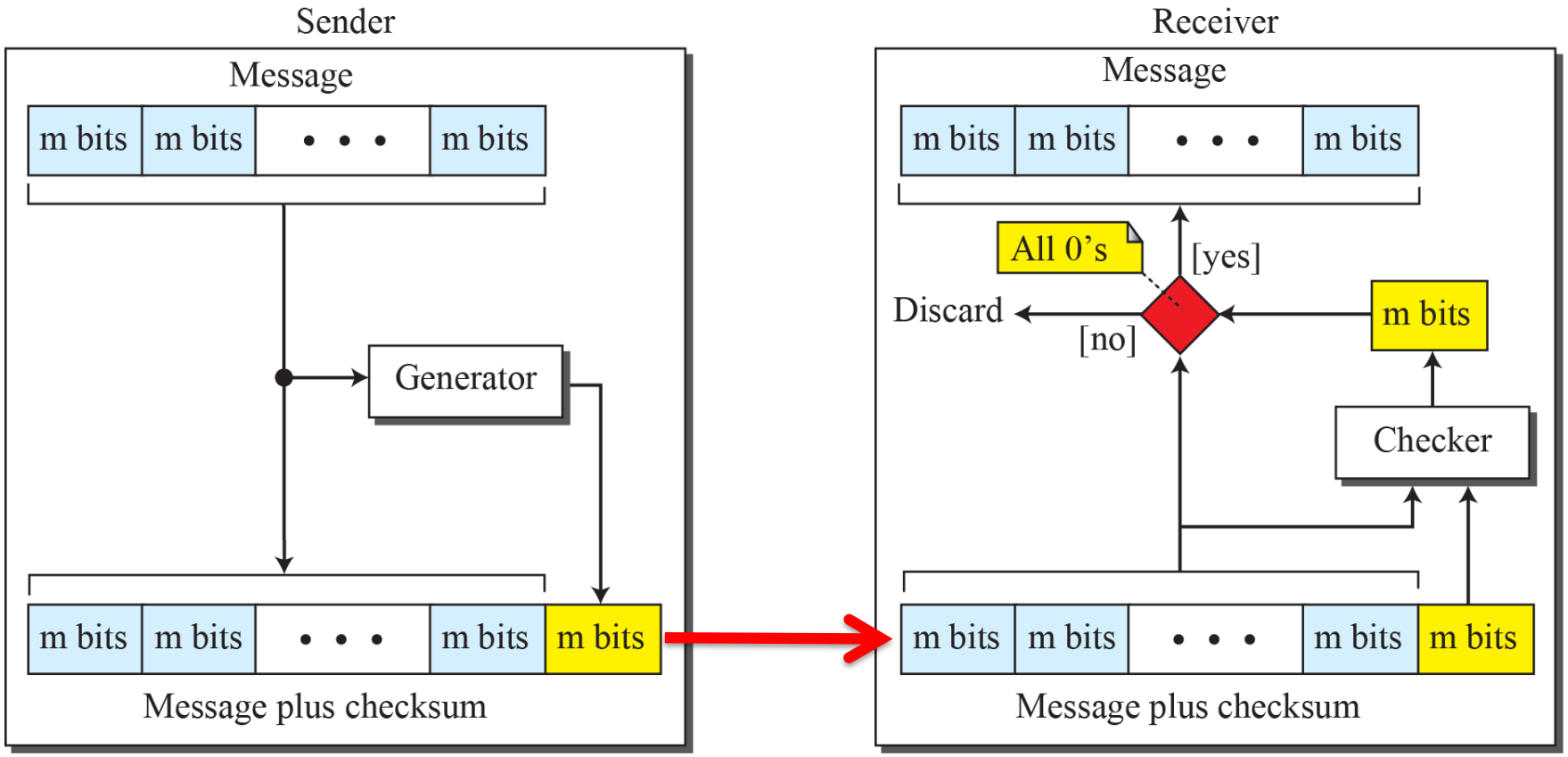
10.3.5 Advantages of Cyclic Codes

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.

10-4 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

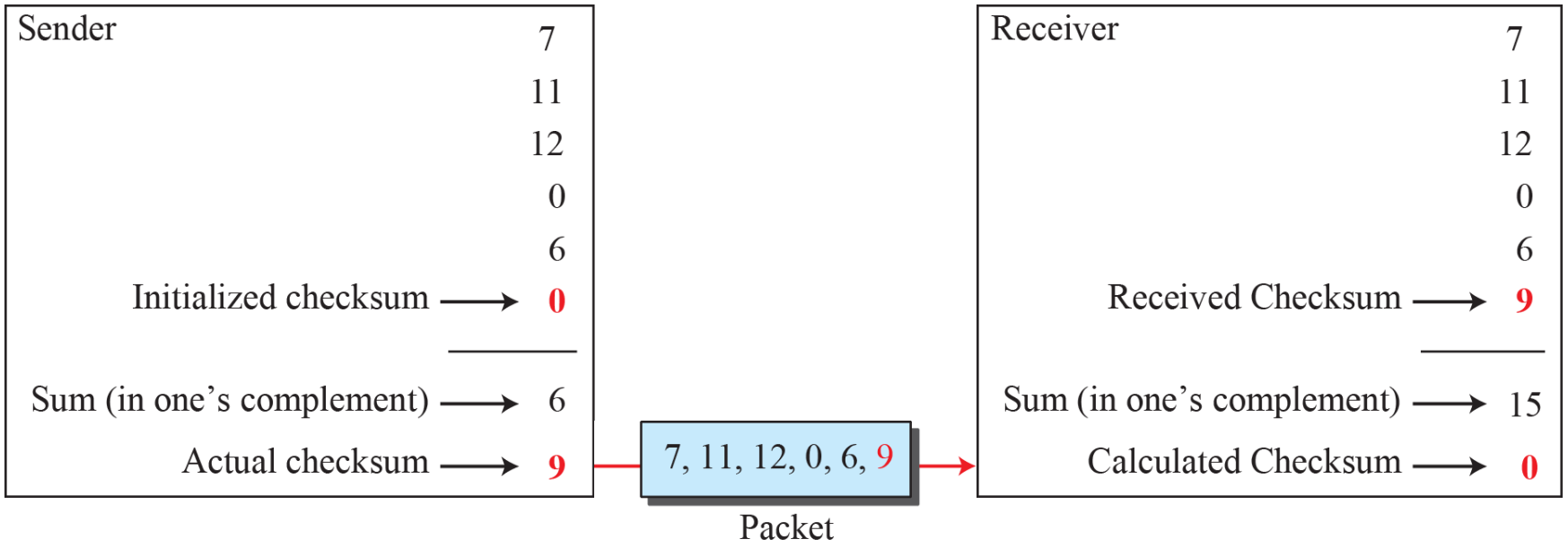


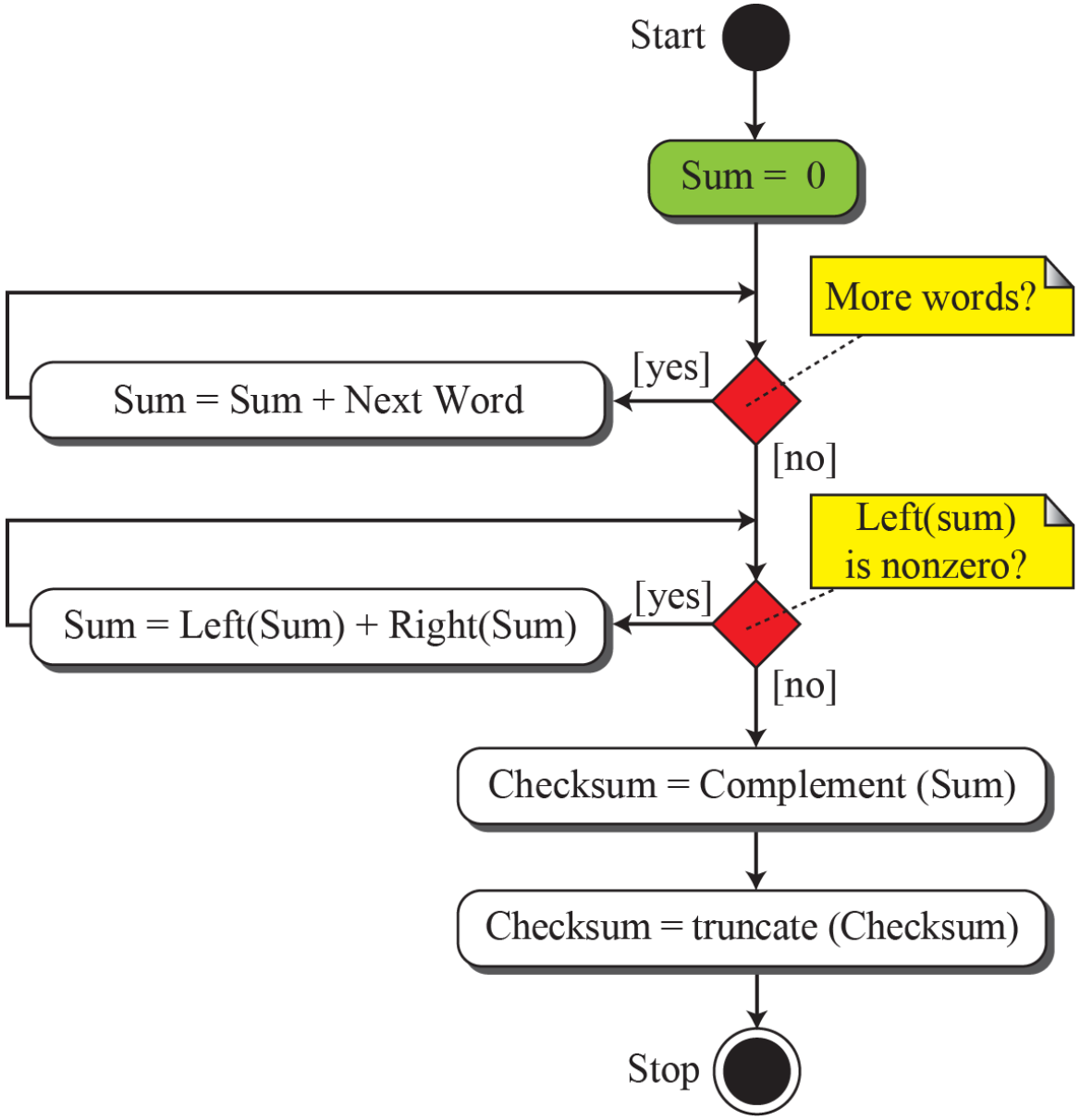
Table 10.5: Procedure to calculate the traditional checksum

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.



Example

Suppose the following block of 16 bits is to be sent using a checksum of 8 bits.
10101001 00111001

The numbers are added using one's complement

```
  1 0 1 0 1 0 0 1
  0 0 1 1 1 0 0 1
-----------------
Sum   1 1 1 0 0 0 1 0
```

Checksum 00011101

The pattern sent is 10101001 00111001 00011101

Example 7

Now suppose the receiver receives the pattern sent in Example 7 and there is no error.
10101001 00111001 00011101

When the receiver adds the three sections, it will get all 1s, which, after complementing, is all 0s and shows that there is no error.

	1 0 1 0 1 0 0 1
	0 0 1 1 1 0 0 1
	0 0 0 1 1 1 0 1
Sum	1 1 1 1 1 1 1 1

Complement 00000000 means that the pattern is OK.

Example

Now suppose there is a burst error of length 5 that affects 4 bits.

10101111 11111001 00011101

When the receiver adds the three sections, it gets

	1 0 1 0 1 1 1 1
	1 1 1 1 1 0 0 1
	0 0 0 1 1 1 0 1
Partial Sum	1 1 1 0 0 0 1 0 1
Carry	1
Sum	1 1 0 0 0 1 1 0

Performance of checksum

Almost detects all errors involving odd numbers of bits or even.

if one or more bits of a segment are damaged and the corresponding bit or bits of opposite value in a second segment are also damaged, then the sum of columns will not change. Receiver will not be able to detect the error