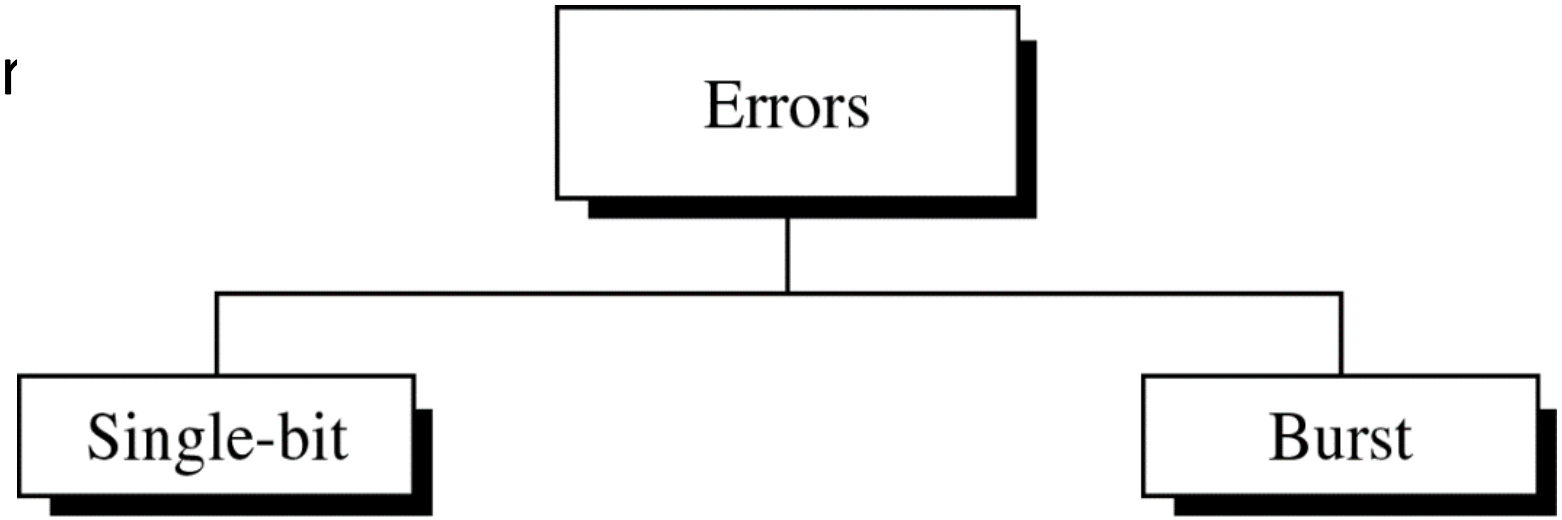
Data Link Layer

Data can be corrupted
during transmission.

Some applications require that
errors be detected and corrected.

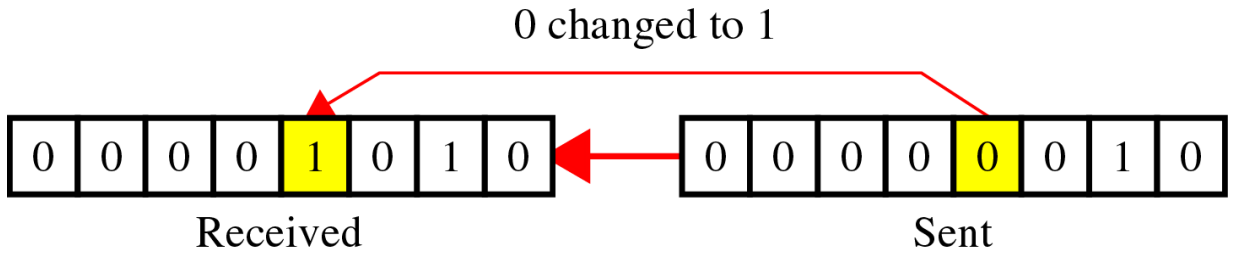
Error Detection and Correction

- Network must be able to transfer data from one devices to another with acceptable accuracy.
- The chances of data being corrupt cannot be ignored.
- So there must be a mechanism for detecting such errors and correct them.
- Types of Error



Single bit error:

- Only 1 bit of a given data unit (byte/packet) is changed from 1 to 0 or 0 to 1.

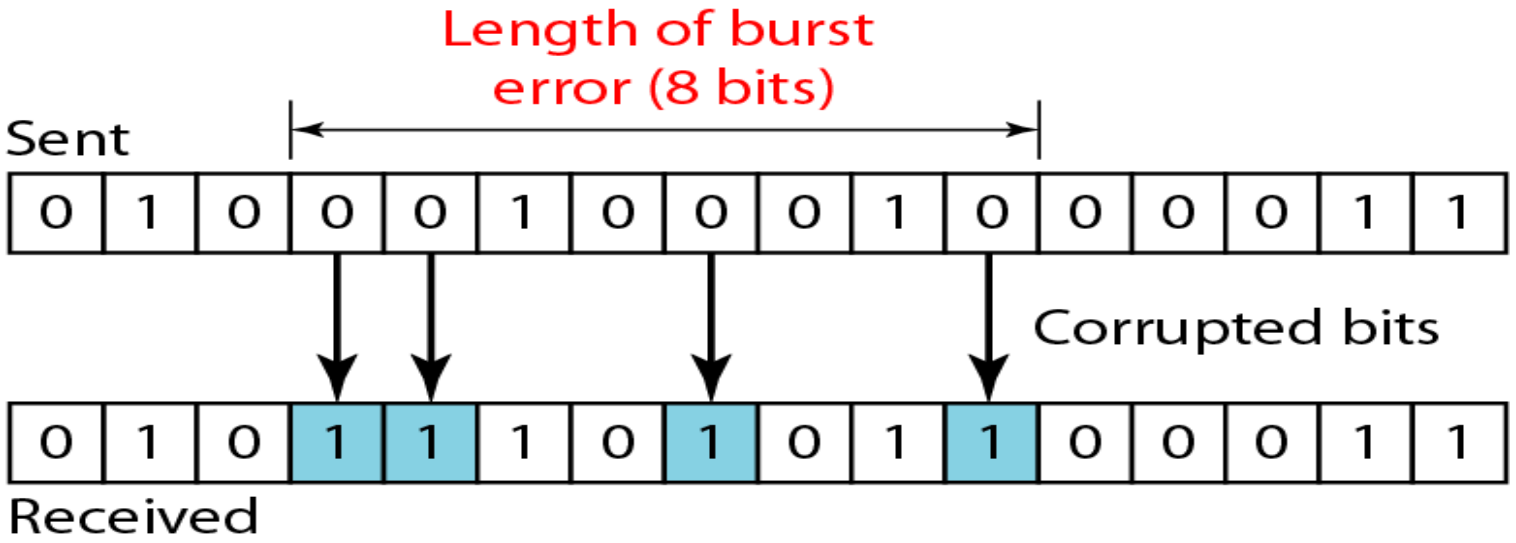


This type of error is very rare to occur.

- For eg. If a data is sent at 1 mbps, then each one bit last only for 1/1,000,000 second (1 (u)micro second).

- So for single bit error to occur, the noise must have (duration of only 1 micro second which is very rare.)

Burst Error



- **A burst error means that 2 or more bits in the data unit have changed.**
- **The length of the burst is measured from the 1st corrupted bit to the last corrupted bit.**
  - **Some bits in between may not have been corrupted.**

Redundancy

- The central concept in detecting or correcting errors is Redundancy.
- Instead of repeating the entire data stream, a shorter group of bits may be appended to the end of each unit.
- → This technique is called **Redundancy**
- These extra bits are discarded as soon as the accuracy of the transmission has been determined.

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

Detection Vs Correction

- **In detection**
  - we only checked if any errors have occurred.
  - The answer is **yes** or **no** and we are not concerned on the number of errors.
- **In correction**,
  - we need to know the exact number of bits that are corrupted, and their location in the message.
  - The number of errors and the size of the message are important factors.
    - 8 bit data, correcting one (single) error bit = 8 possibility
    - 8 bit data, correcting 2 error bits = 28 possibilities
    - 1000 bit data, correcting 10 error bits = ……. Possibilities.

Error Correction Method

- There are two main methods of error correction.
  - **Forward Error Correction**
    - Process in which the receiver tries to **guess** the message by Redundant bit.
  - **Retransmission:**
    - Receiver asks for **re-transmission** of message if it detects error. (usually not all errors are detected)

Coding

- The sender adds redundant bits

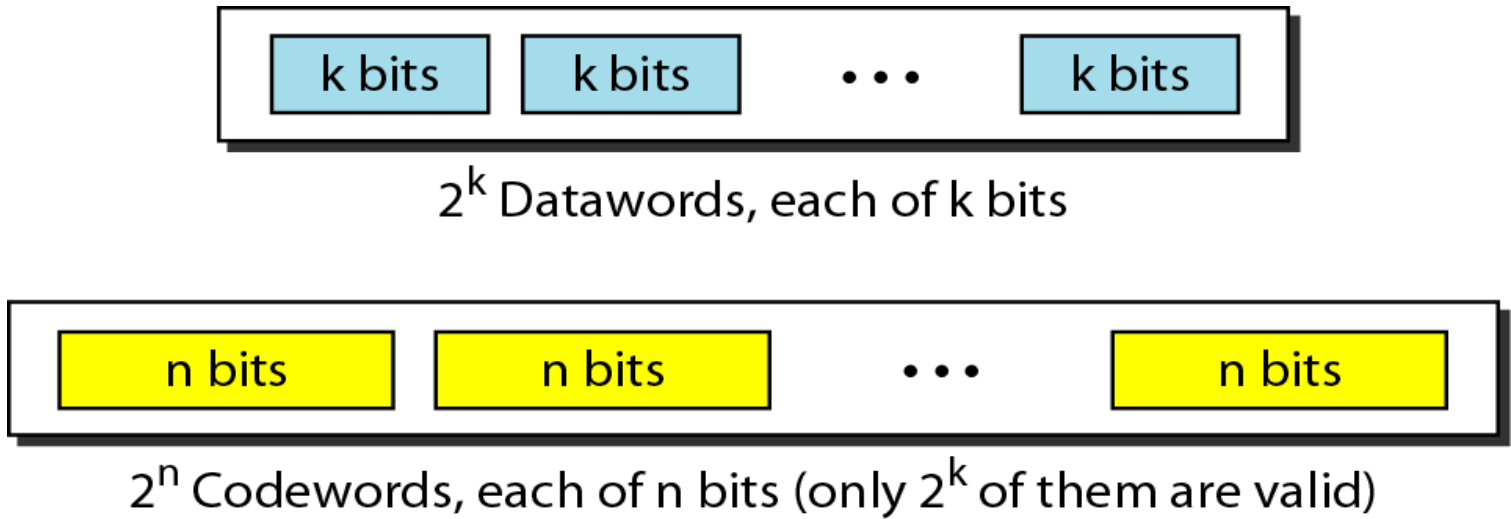
- The receiver checks the relationships between the two sets of bits to detect or correct the errors.

- Coding schemes are divided into two broad categories:
  - **Block coding**
  - **Convolution coding**

Dataword	Codeword
00	000
01	011
10	101
11	110

Block Coding

- Message is divided 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**.



Datawords and codewords in block coding

Block Coding

- With **k** bits, we can create a combination of $\mathbf{2^k}$ datawords.
- Since **n** bits, we can create a combination of $\mathbf{2^n}$ codewords.
- Since n > k. The possibility codeword is greater than possible data words.
- This means we have $\mathbf{2^n - 2^k}$ codewords extra.

- Eg if **k=4 and n=5**,
- we have $\mathbf{2^4}$ =16 dataword and $\mathbf{2^5}$ =32 codewords.
- Hence 16 out of 32 codewords are used for message transfer and rest are unused.

- This means that we have $2^n - 2^k$ codewords that are not used.
- We call these codewords invalid or illegal.

- If the receiver receives an invalid codeword, this indicates that the data was corrupted during transmission.

Error Detection in Block coding

- Two conditions satisfy the error detection.
  - The receiver has a list of valid codewords.
  - The original codewords has changed to an invalid one.

Eg. Let k =2 and n =3

Dataword	Codeword
00	000
01	011
10	101
11	110

Sender

Receiver

- Let the sender encodes dataword 01 as 011 and send it receiver.
- Following cases may arise.
  - Receiver receives 011, it is valid codeword. Dataword 01 is extracted by receiver. **No error.**
  - Receiver receives **1**11. Code word is corrupted. This is not a valid codeword and it is discarded.
  - Receiver receives 0**0**0. Codeword is corrupted but this is valid codeword. Receiver incorrectly extracts dataword 00. **Here two corrupted bits have made the error undetectable.**

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

Dataword	Codeword
00	000
01	011
10	101
11	110

Hamming Distance

- It is one of the central concept in coding for error control.
- The Hamming distance between two words (of the same size) as the number of difference between the corresponding bits.
- Why do you think Hamming distance is important for error detection?
- The reason is that the Hamming distance between the received codeword and the sent codeword is the number of bits that are corrupted during transmission.

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)

Hamming Distance

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

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

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

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

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

BLOCK CODING - Minimum Hamming Distance

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

d_{min} used to define the minimum Hamming distance in a coding scheme.

Solution

We first find all Hamming distances.

The d_{min} in this case is 2.

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

BLOCK CODING - Minimum Hamming Distance

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

Solution

We first find all the Hamming distances.

The d_{min} in this case is 3.

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

- **Three Parameters:**
  - For any coding scheme we need three parameters.
    - Code word size **n**
    - Dataword size **k**
    - Minimum Hamming distance d_{min}
  - A **coding scheme *C*** is written is *C(n, k)* with a separate expression for d_{min}
  - For example *C(3,2)* with d_{min}=2 and *C(5,2)* with d_{min}=3.

Relationship in between Hamming distance and errors occurring

- Hamming distance between the sent and received codewords is the **number of bits affected by the error**.

- For example,
  - send codeword 00000
  - received codeword 01101, 3 bits are in error and the Hamming distance is
  - d(00000,01101) =3

Minimum Hamming Distance for Error Detection

- If s errors occur during transmission,
  - the Hamming distance between the sent codeword and received codeword is s.

- If it is necessary to detect upto s errors,
  - the minimum hamming distance between the valid codes must be s+1, so that the received codeword does not match a valid codeword.

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

- The minimum Hamming distance for our first code scheme from the table is

  $d_{min} = s+1$

  or $2=s+1$

  $S=1$

  So, this code guarantees detection of

Dataword	Codeword
00	000
01	011
10	101
11	110

BLOCK CODING - Minimum Distance for Error Correction

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

BLOCK CODING - Minimum Distance for Error Correction

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.**

- **Error correction codes need to have an odd minimum distance (3, 5, 7, . . .).**

Linear Block Codes

- This is the widespread used coding scheme. A linear block code is a code in which the exclusive OR of two valid codeword creates another valid codeword.
- Eg.

  000 ⊕ 011 = 011

  011 ⊕ 101 = 110

  101 ⊕ 110 = 011

Simple parity-Check Code

- In Simple parity-check 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**.
- Although some implementations specify an odd number of 1s, we discuss the even case.
- This code is a single-bit error-detecting code; it cannot correct any error.

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

Sender

Receiver

- At Generator, $r_o = a_3+a_2+a_1+a_0$ (modulo 2)

  Where if number of 1s is even, the result is zero

  if number of 1s is odd, the result is one

Minimum Distance for Linear Block Codes

Simple parity-check code C(5, 4)

At Generator, $r_o = a_3+a_2+a_1+a_0$ (modulo 2)

Where if number of 1s is even, the result is zero

if number of 1s is odd, the result is one

Sender

Receiver

- The checker at the receiver does the same thing as the generator in the sender with one exception: The addition is done over all 5 bits.
- The result, which is called the **syndrome**, is just 1 bit.
- The syndrome is **0** when the number of **1s** in the received codeword is even; otherwise, it is **1**.

- The syndrome is passed to the decision logic analyzer.
- If the syndrome is 0, **there is no error in the received codeword;** codeword is accepted as the dataword;
- if the syndrome is 1, the data portion of the received codeword is discarded.
- **The dataword is not created**.

Sender

Receiver

Simple parity-check (Cont..)

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.

Simple parity-check (Cont…)

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.
The simply parity-check decoder cannot detect an even numbers of errors.

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.

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

Hamming codes

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

A codeword consists of n bits of which k are data bits and r are check bits.

Let m = r, then we have:

$n=2^m-1$ and

$k=n-m$

- For eg. If m=3 then n=7 and k=4,
  - Hence it is a Hamming code C(7,4) with $d_{min}=3$

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

A copy of a 4-bit dataword is fed into the generator that creates three parity checks r_0, r_1, r_2, as shown below:

$$r_0 = a_2 + a_1 + a_0 \quad \text{(modulo -2)}$$
$$r_1 = a_3 + a_2 + a_1 \quad \text{(modulo -2)}$$
$$r_2 = a_1 + a_0 + a_3 \quad \text{(modulo -2)}$$

Each of parity check bits handles **3 out of 4 bits** of **dataword**. The total number 1s in each 4-bit combination (3 dataword bits & 1 parity bit) must be even.

Sender

Receiver

$r_0 = a_2 + a_1 + a_0$ (modulo -2)

$r_1 = a_3 + a_2 + a_1$ (modulo -2)

$r_2 = a_1 + a_0 + a_3$ (modulo -2)

Hamming code C(7, 4)

- At Checker

$$S_0 = b_2 + b_1 + b_0 + q_0 \pmod 2$$
$$S_1 = b_3 + b_2 + b_1 + q_1 \pmod 2$$
$$S_2 = b_1 + b_0 + b_3 + q_2 \pmod 2$$

- The 3 bit syndrome creates 8 different bit patterns (000 to 111) that can represent 8 different conditions.

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

Fig: logical decision made by the correction logic Analyzer

- In Syndrome 000, 001, 010 & 100 is not concerned with generator since there is error or an error in the parity bit.

- In Syndrome 011, 101, 110, 111 one of the bits must be flipped (from 0 to 1 or from 1 to 0) to find correct dataword.

Sender Receiver

$S_0=b_2+b_1+b_0+q_0 \pmod 2$
$S_1=b_3+b_2+b_1+q_1 \pmod 2$
$S_2=b_1+b_0+b_3+q_2 \pmod 2$

Cases:

1. Dataword **0100** becomes 0100011 codeword **0100011** is received, syndrome 000. no error

2. Dataword **0111** becomes 0111001. Codeword 0011001 is received. Syndrome is 011. Hence the error is b_2. After flipping b_2 (changing 0 to 1), the final dataword 0111 is obtained.

Dataword	Codeword
0000	0000000
0001	0001101
0010	0010111
0011	0011010
0100	0100011
0101	0101110
0110	0110100
0111	**0111001**
1000	1000110

Sender

Receiver

Syndrome	000	001	010	011	100	101	110
Error	None	q_0	q_1	b_2	q_2	b_0	b_3

3. The dataword 1101 is sent as codeword 1101000. Receiver receives 0001000 (2 errors). The syndrome is 101 i.e b_0 is in error.

After flipping b_0 we get 0000 (wrong dataword).

Hence two error cannot be correctec but can be detected. Single error is detected and corrected.

Dataword	Codeword
1101	1101000
1110	1110010
1111	1111111

$$S_0=b_2+b_1+b_0+q_0 \pmod 2$$
$$S_1=b_3+b_2+b_1+q_1 \pmod 2$$
$$S_2=b_1+b_0+b_3+q_2 \pmod 2$$

Sender

Receiver

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

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

  1 0 1 1 0 0 0 is a codeword and we cyclically left-shift, then

  0 1 1 0 0 0 1 is also a codeword.

- In this case, if we call the bits in the first word a_0 to a_6, and the bits in the second word b_0 to b_6, we can shift the bits by using the following:

$$b_1=a_0\,,\ b_2=a_1\,,\quad b_3=a_2\quad ,b_4=a_3,\ b_5=a_4,\quad b_6=a_5\ ,\ \ b_0=a_6$$

- A category of cyclic code is Cyclic Redundancy Check (CRC)

- Used in LANs & WAN's

Cyclic Redundancy Check

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

The generator uses n-k+1 bit divisor which is predefined and agreed

The remainder (3 bit) is appended to the dataword to make the final codeword.

At Encoder

- In encoder, the dataword has k bits (4 here)
- The codeword has **n** bits (7 here)
- The size of the dataword is augmented by adding n – k (3 here)
- The generator uses a **divisor of size** n-k + 1 (4 here)
- The quotient of the division is discarded.
- The remainder ($r_2r_1r_0$) is **appended** to the dataword to create the codeword.

- Eg. Dataword =1001 (k=4), & if codeword (n) = 7 bits.
- Augment dataword by (n-k) zero bits i.e 7-4=3 zero bits.
  - i.e 1001000
- Let the divisor is 1101 (Predefined and agreed) (n-k+1) digits
- The remainder ($r_2r_1r_0$) is appended to the dataword to create the codeword

At decoder

- Decoder does the same division as encoder.
- Eg. Dataword =1001 (k=4), & if codeword (n) = 7 bits.
- Augment D.W by (n-k) zero bits i.e 7-4=3 zero bits.
  - i.e 1001000
- Let the divisor is 1101 (Predefined and agreed) (n-k+1)
- If remainder is all **0's there is no error**, **else there is error**.

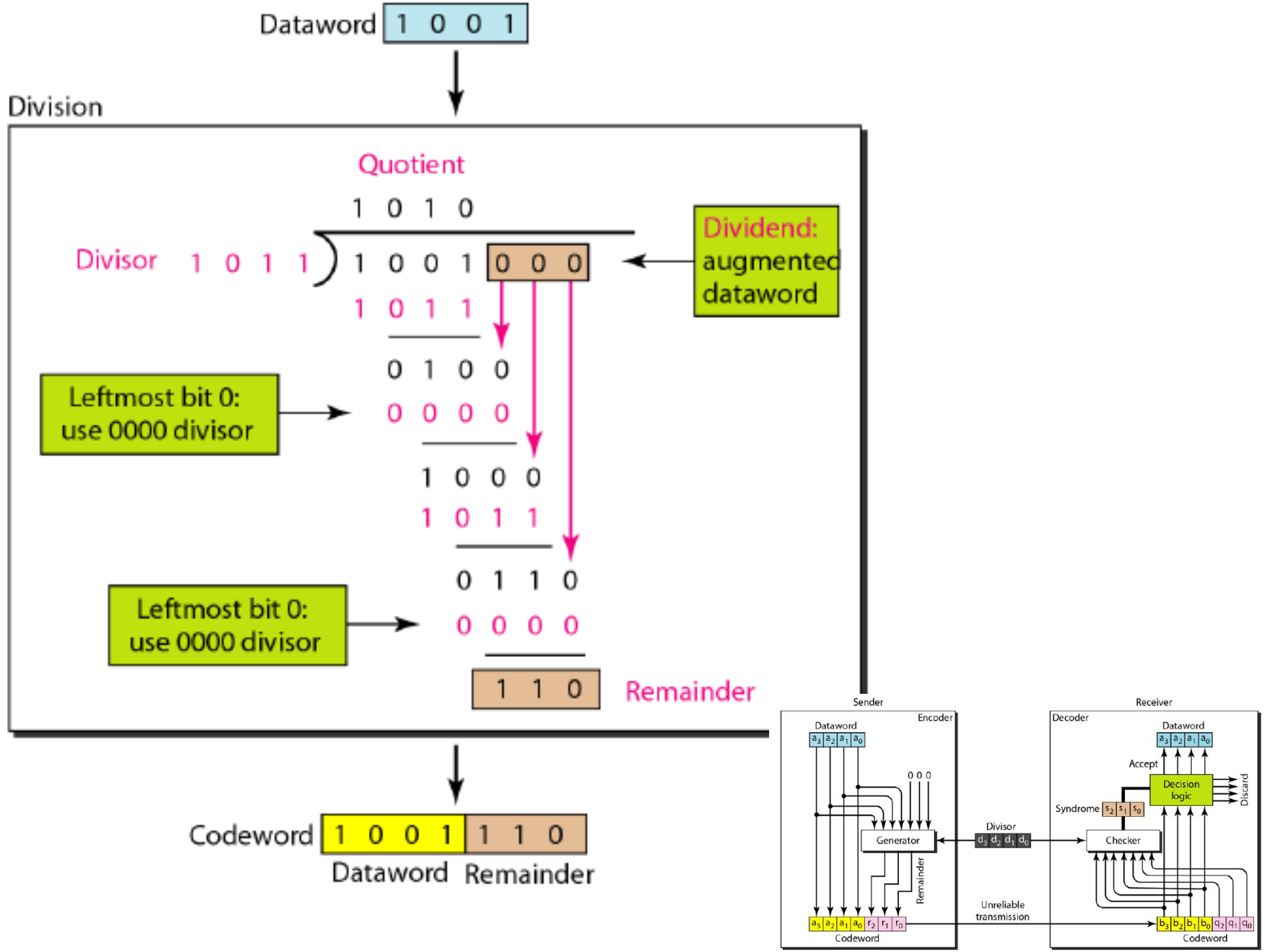
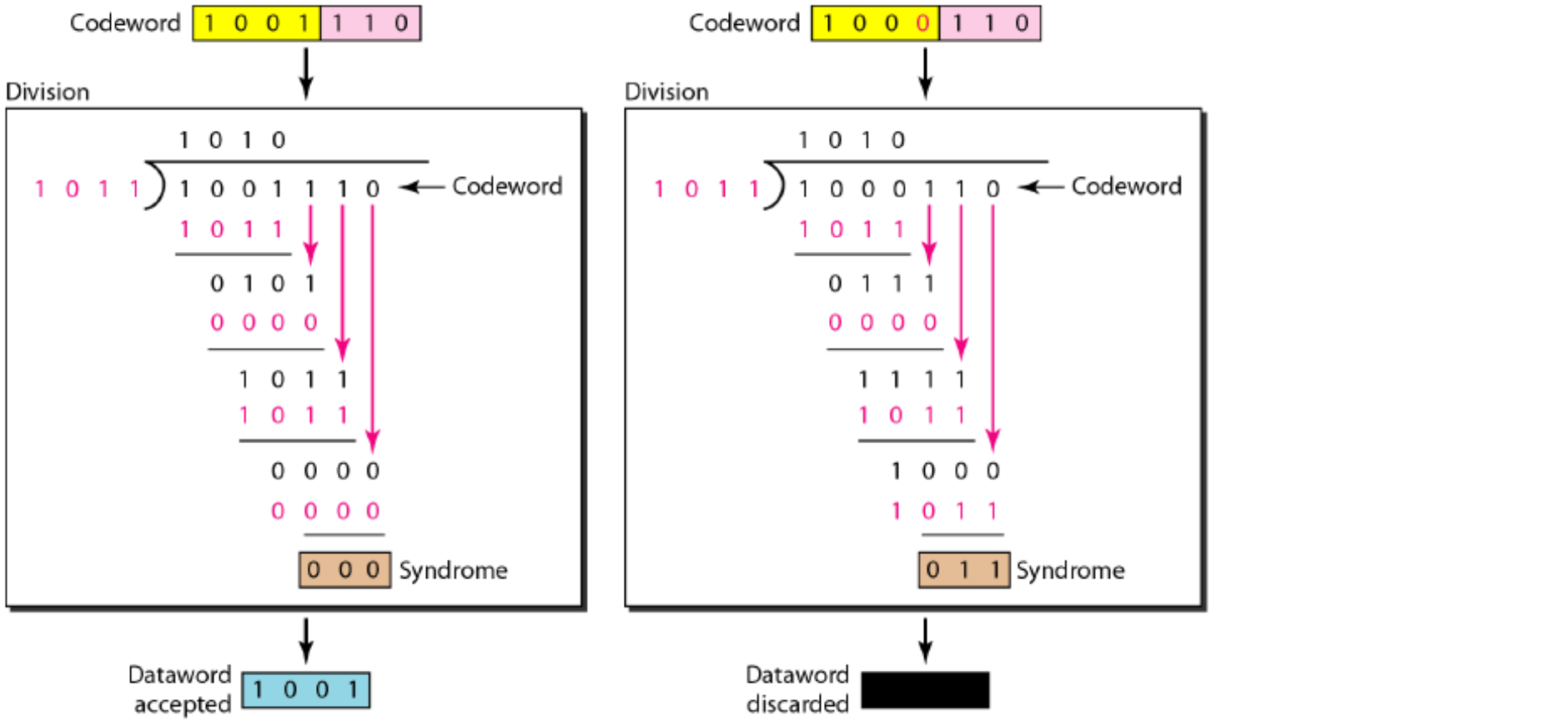
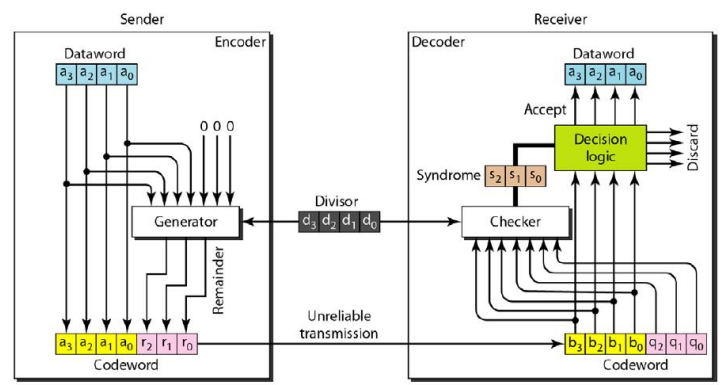

Dataword 1 0 0 1
Division
Quotient
1 0 1 0
Divisor 1 0 1 1
1 0 0 1 0 0 0
Dividend: augmented dataword
1 0 1 1
0 1 0 0
Leftmost bit 0: use 0000 divisor
0 0 0 0
1 0 0 0
1 0 1 1
0 1 1 0
Leftmost bit 0: use 0000 divisor
0 0 0 0
1 1 0
Remainder
Codeword 1 0 0 1 1 1 0
Dataword Remainder
Sender
Encoder
Dataword
Generator
Remainder
Codeword
Divisor
Unreliable transmission
Receiver
Decoder
Dataword
Accept
Decision logic
Discard
Syndrome
Checker
Codeword

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



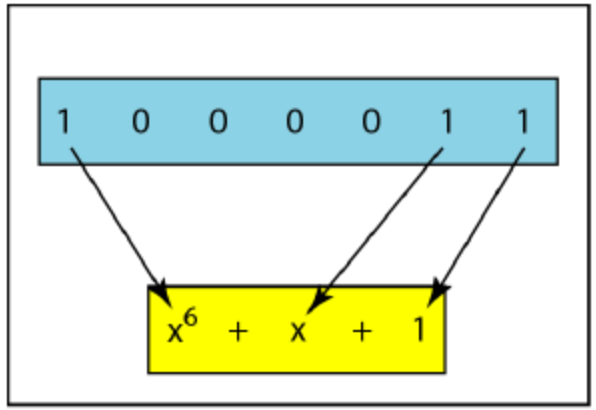


Polynomials

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 0s and 1.

Figure 10.21 *A polynomial to represent a binary word*

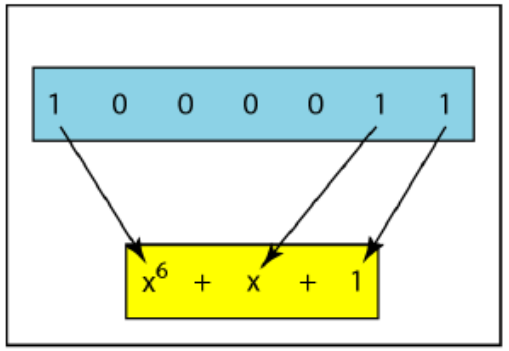


a. Binary pattern and polynomial

b. Short form

- **Degree of Polynomial** : It is the highest power
  - Eg $x^6+x+1 \Rightarrow 6$
- Note: The degree of a polynomial is less than that the number of bits in the pattern. The bit pattern in this case has 7 bits.
  - So general D.P => n-1 bit if n is number of bits.

Figure 10.21 *A polynomial to represent a binary word*



a. Binary pattern and polynomial

b. Short form

Addition & Subtraction :

Addition or subtraction is done by combining terms and deleting pairs of identical terms.

Eg. $(x^5+x^4+x^2) + (x^6+x^4+x^2) => x^6 + x^5$

X^4 and x^2 are deleted

- **Multiplying or dividing terms**:
  - Just add the powers of multiplication.
    - Eg. $X^3 \times x^4 = x^7$
  - Subtract the powers for division.
    - Eg. $X^5/x^4 = x$

Multiplying two polynomials

- It is conceptually same as we did it for the encoder/decoder, pairs equal terms are deleted.
  - $(x^5+x^3+x^2+x)\ (x^2+x+1)$

(let the student do rest of the problem)

- $x^7+x^6+x^3+x^2+x$

Shifting

- A binary pattern is often shifted a number of bits to the right or left.
- Shifting to the left means adding 0's extra bit at right side.
- Shifting to the Right means deleting some right most bits.

Sifting left 3 bits:

10011 => 10011000

$X^4+x+1 => x^7+x^4+x^3$

Sifting right 3 bits :

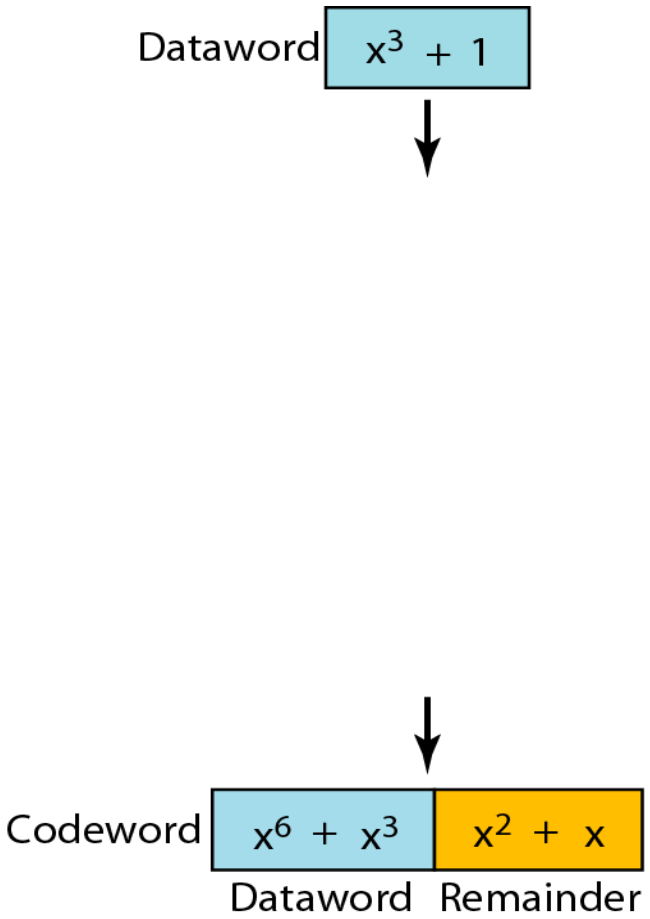
- 10011 => 10
- $X^4+x+1 => x$

Cyclic Code Encoder Using Polynomials

- Cyclic Code Encoder Using Polynomial
  - Dataword 1001 => x^3+1
  - Divisor 1011 => x^3+x+1
  - To find the augmented dataword, we have left-shifted the dataword 3 bits (multiplying by x^3)
  - Finally we augmented data is => x^6+x^3

Cyclic Redundancy Check

Figure 10.22 *CRC division using polynomials*



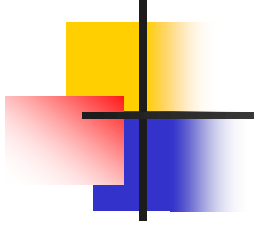
Equivalence of Polynomial and Binary

- In a polynomial representation, the divisor is normally referred to as the generator polynomial *g(x).*

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

Cyclic Code Analysis

- The divisor is normally called the generator polynomial or generator
- Analysis:
  - Let *f(x)* is a polynomial with binary coefficient
  - *d(x)*: Dataword
  - *c(x)*: Codeword
  - *g(x)*: Generator
  - *s(x)*: Syndrome
  - *e(x)*: Error



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.

- To find the **criteria** that must be imposed on the generator **g(x)** to detect the type of error that need to be detected.

- Relationship among the **sent codeword**, **error**, **received codeword**, and the generator.

$$Received\ Codeword = c(x) + e(x)$$

- The receiver divides the received codeword by g(x) to get the syndrome. We can write this

$$\frac{Received\ Codeword}{g(x)} = \frac{c(x)}{g(x)} + \frac{e(x)}{g(x)}$$

$$\frac{\text{Received Codeword}}{g(x)} = \frac{c(x)}{g(x)} + \frac{e(x)}{g(x)}$$

The first term at the right hand side of the equality has a remainder of zero (according to the definition of codeword).

So the syndrome is actually the remainder of the **second term** on the right-hand side.

If the term does not have remainder (syndrome=0), either

1. e(x) = 0 or
2. e(x) is divisible by g(x).

We don't need to worry about first case, but

The second term is important. Those errors that are not divisible by g(x) are not caught.

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

Single bit error.

- Question is :

What should be the structure of g(x) to guarantee the detection of single-bit error?

- Here, A single bit error ***e(x) =x^i*** where ***i***=position of error bit.
  - then **x^i/g(x)** => will have a remainder.
  - If g(x) have at least two terms and the coefficient of **x^o is 1** then e(x) cannot be divided by g(x) i.e there will be some remainder.

Note

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

Example 10.8

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.

Table 10.7 *Standard polynomials*

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**.

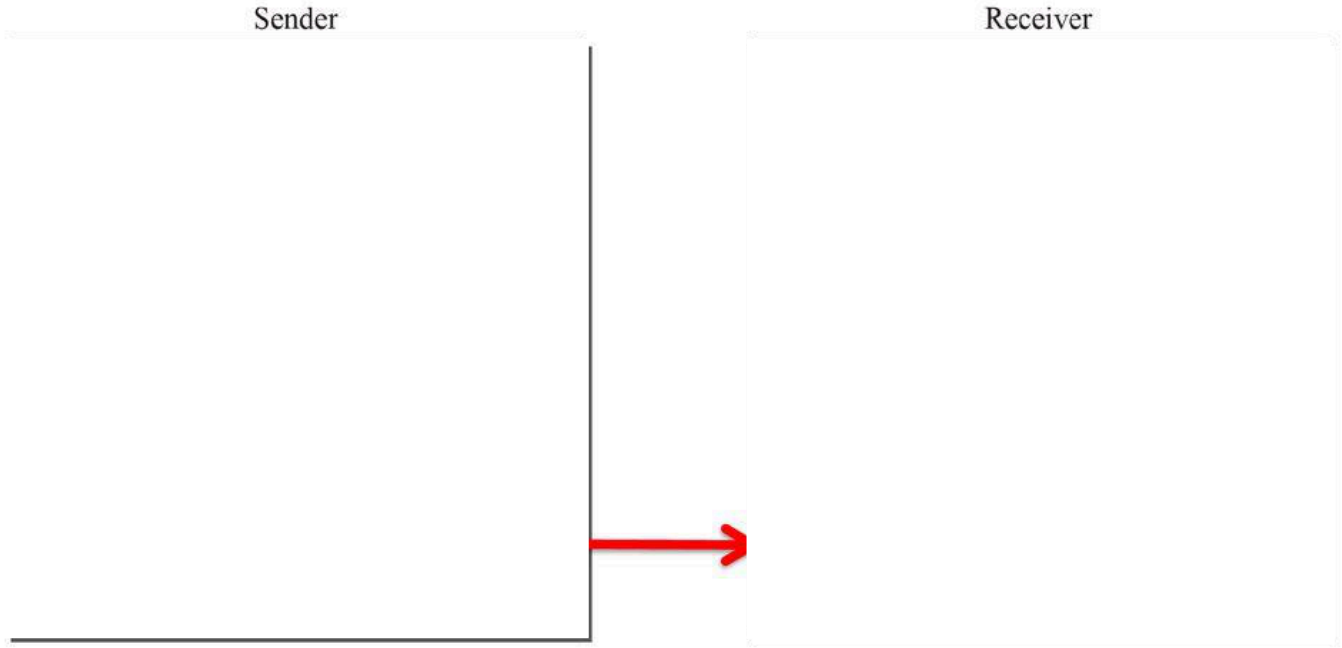
- At the source, the message is first divided into **m-bits** units.

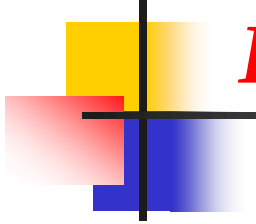
- The generator then creates an **extra m-bit** unit called the checksum, which is sent with the message.

- At the destination, the checker creates a new checksum from the combination of the message and sent checksum.

- If the checksup is all 0s, the message is accepted; other wise the message is discarded.

Figure 10.15: 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.

- **For Simplicity,**

- 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, assumes no error, otherwise, there is an error.

One’s Complement

- The previous example has one major drawback. All our data can be written as a **4-bit word (they are less than 15)** except the checksum.

- We have solution to use **one’s complement** arithmetic.

- In this one's complement arithmetic, we can represent **unsigned numbers** between **0 and 2^m-1** using only **m** bits.

- If the number has more than **m** bits, the extra leftmost bits need to be added to the **m** rightmost bits (wrapping).

- In one's complement arithmetic, we have two 0s; one positive and one negative, which are complements of each other.

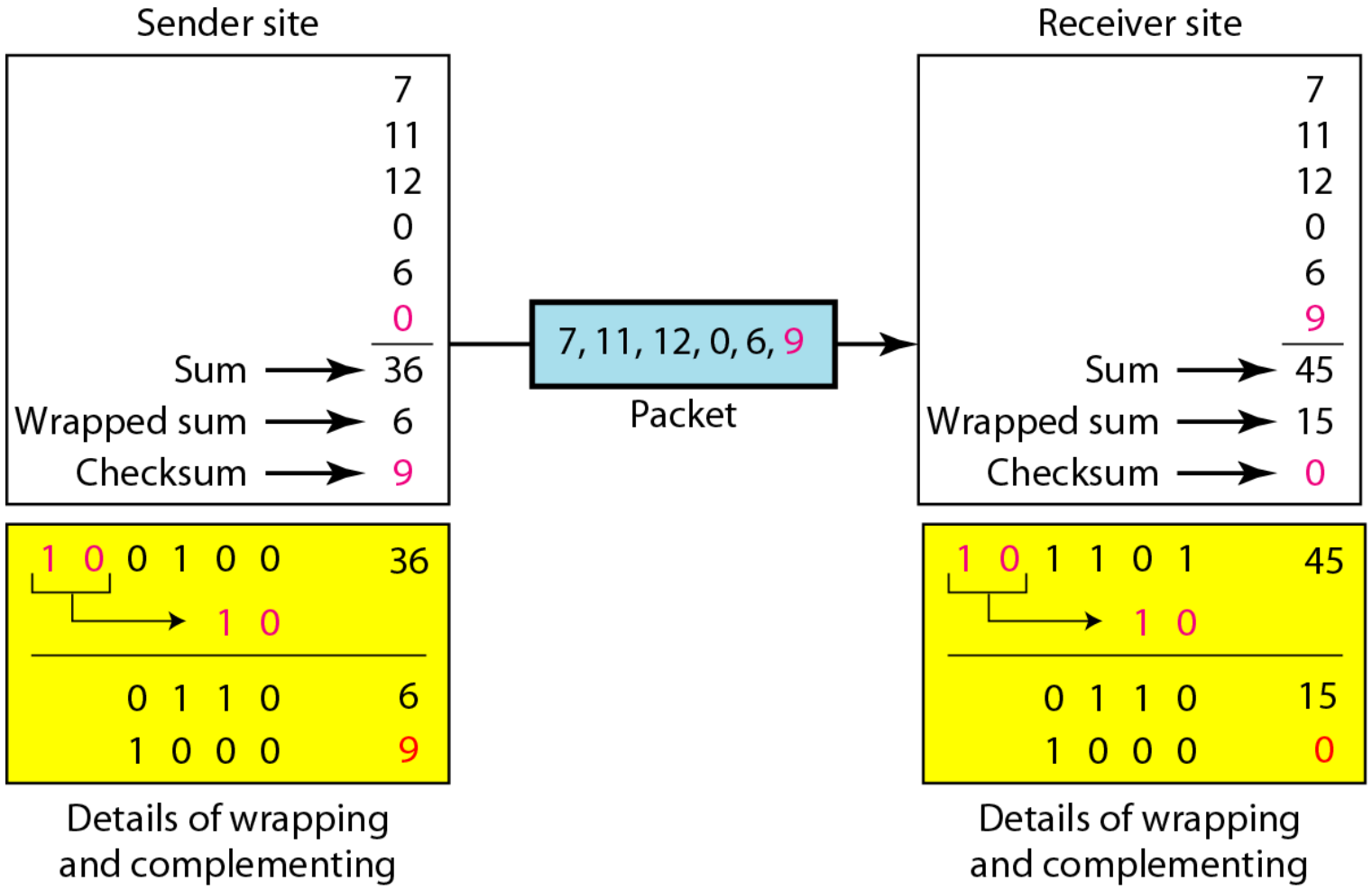
How can we represent the number 36 in one's complement arithmetic using only four bits ?

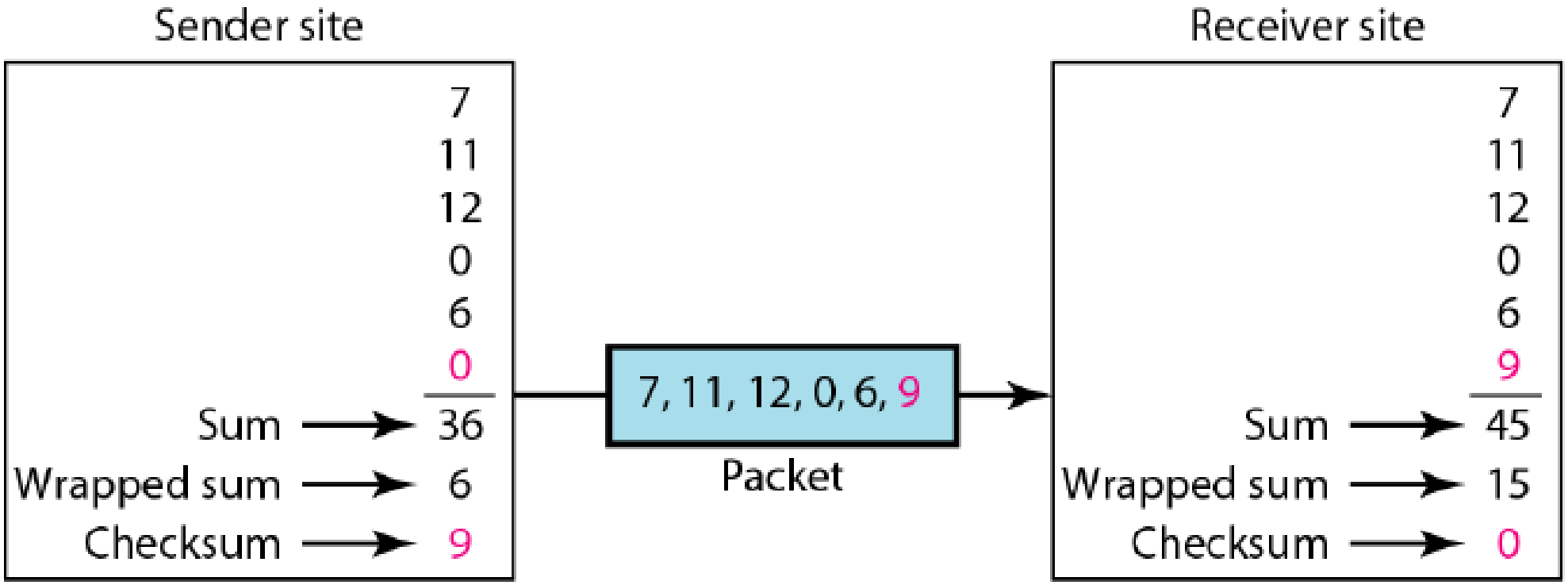
- **Solution**
  - The number 36 is binary is 100100 (it needs six bits).
  - We can wrap the leftmost bit and add it to the four rightmost bits.
  - We have $(10)_2+(0100)_2 =(0110)_2 = (6)_{10}$

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;
  - One’s complement of 6 is 1001;
  - If we consider only unsigned number, this is 9.
  - In other words, the complement of 6 is 9.

Figure 10.24 *Example 10.22*




Sender site
7
11
12
0
6
0
Sum
36
Wrapped sum
6
Checksum
9
7, 11, 12, 0, 6, 9
Packet
Receiver site
7
11
12
0
6
9
Sum
45
Wrapped sum
15
Checksum
0