# COMP 3721
# Introduction to Data Communications

## 07a - Week 7 - Part 1

# Learning Outcomes

- By the end of this lecture, you will be able to
  - Explain what are the types of errors.
  - Distinguish between error detection and error correction.
  - Explain how block coding and hamming distance work.
  - Explain how cyclic codes work.
  - Explain how to calculate a checksum.
  - Explain FEC (Forwarding Error Correction) techniques.

# Introduction

- Networks must be able to transfer data from one device to another with **acceptable accuracy**.
  - The data received **must be identical** to the data transmitted.
- Data may become **corrupted** in passage (one or more bits may be modified).
- A small level of error may be tolerated by some applications.

# Introduction

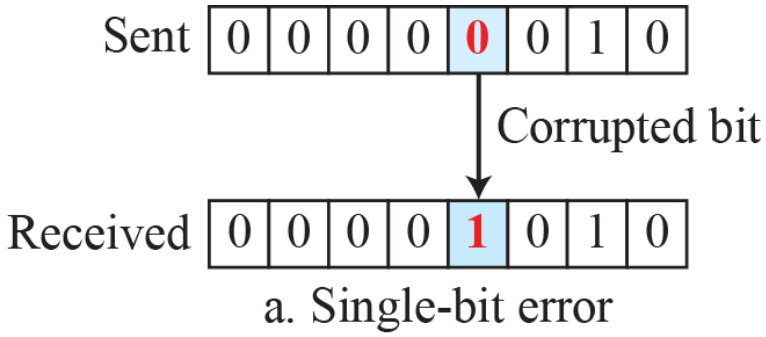
- Networks must be able to transfer data from one device to another with **acceptable accuracy**.
  - The data received **must be identical** to the data transmitted.
- Data may become **corrupted** in passage (one or more bits may be modified).
- A small level of error may be tolerated by some applications.
- The approach of most link-layer protocols for error control → simply **discard the frame** and let the **upper-layer protocols** handle the **retransmission** of the frame.

# Types of Errors

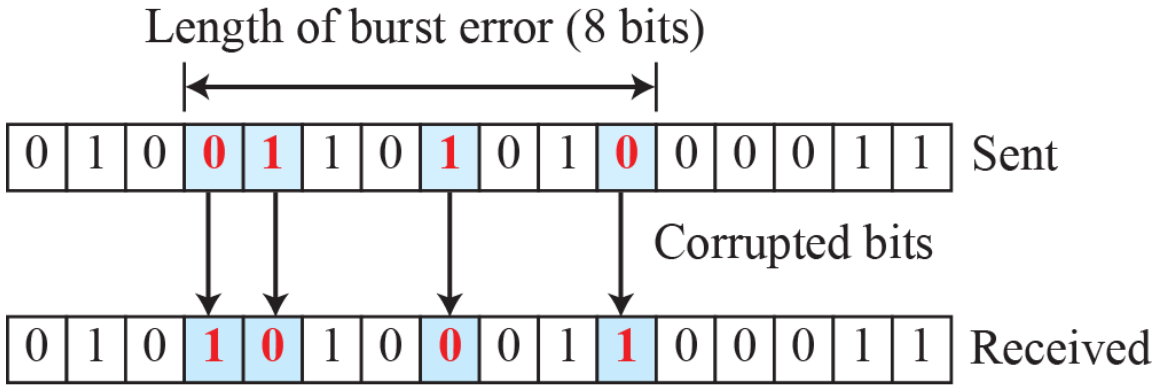
- **Interference** can change the shape of the signal.
  - Bits are subject to unpredictable changes.

# Types of Errors

- **Interference** can change the shape of the signal.
  - Bits are subject to unpredictable changes.
- **Single-bit error** and **Burst error**
  - 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 1 → **single-bit error**
  - 2 or more bits in the data unit have changed from 1 to 0 or from 0 to 1 → **Burst Error**



a. Single-bit error



b. Burst error

**More likely to happen. Why?**

# Number of Bits affected by the Noise

Number of bits affected by the noise depends on the **data rate** and the **duration of noise**.

# Example

- Assume we are sending data and there is a noise of 0.001 second.
- How many bits are affected by this noise if
  a) The data rate is 1 kbps.
  b) The data rate is 1 Mbps.

# Example

- Assume we are sending data and there is a noise of 0.001 second.
- How many bits are affected by this noise if
  a) The data rate is 1 kbps.
  b) The data rate is 1 Mbps.

- **Answer:**
  a) $1000 \times 0.001 =$ **1 bit**
  b) $1000000 \times 0.001 =$ **1000 bits**

# Error Detection vs. Error Correction

- **Error detection**
  - Only looking to see if any error has occurred.
  - The number of corrupted bits (type of error) is not important.
- **Error correction**
  - The **exact number of corrupted bits** and **their location** in the message need to be identified.
  - The number of errors and the size of the message are important factors.

- The **correction** of errors is **more difficult** than the **detection**.

# Redundancy for Error Control

- Sending some **extra bits** with our data.
- These redundant bits are **added by the sender** and **removed by the receiver**.
- Redundant bits allow the **receiver** to **detect** or **correct** corrupted bits.
- Redundancy is achieved through **coding schemes**.
- The sender adds redundant bits which **have a relationship** with the actual data.

# Block Coding

- **Dataword**
  - The message is divided into blocks, each of *k* bits.

# Block Coding

- **Dataword**
  - The message is divided into blocks, each of $k$ bits.
- **Codeword**
  - $r$ redundant bits are added to each block to make the length $n = k + r$

# Block Coding

- **Dataword**
  - The message is divided into blocks, each of $k$ bits.
- **Codeword**
  - $r$ redundant bits are added to each block to make the length $n = k + r$
- $2^k$ datawords < $2^n$ codewords

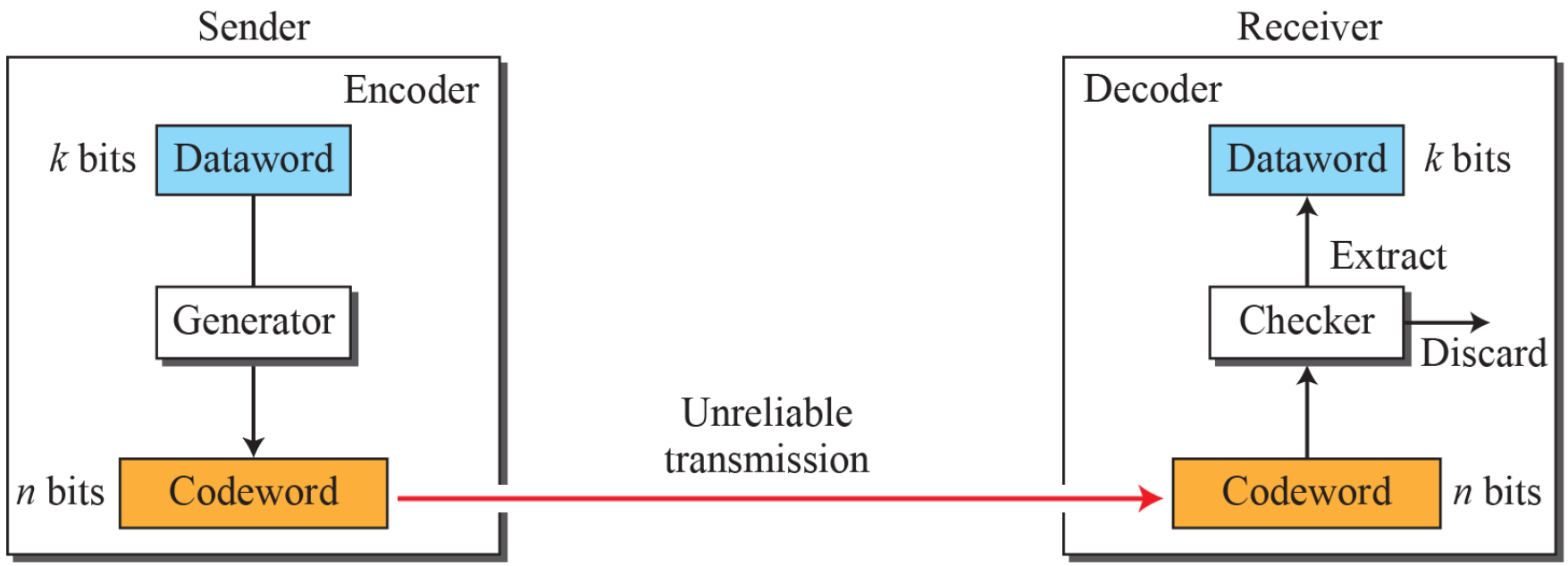
# Block Coding

- **Dataword**
  - The message is divided into blocks, each of $k$ bits.
- **Codeword**
  - $r$ redundant bits are added to each block to make the length $n = k + r$
- $2^k$ datawords < $2^n$ codewords
- The **block coding** process is **one-to-one**.
  - The **same dataword** is always encoded as the **same codeword**.
  - $(2^n - 2^k)$ codewords are not used (invalid codes) → the **trick** in error detection

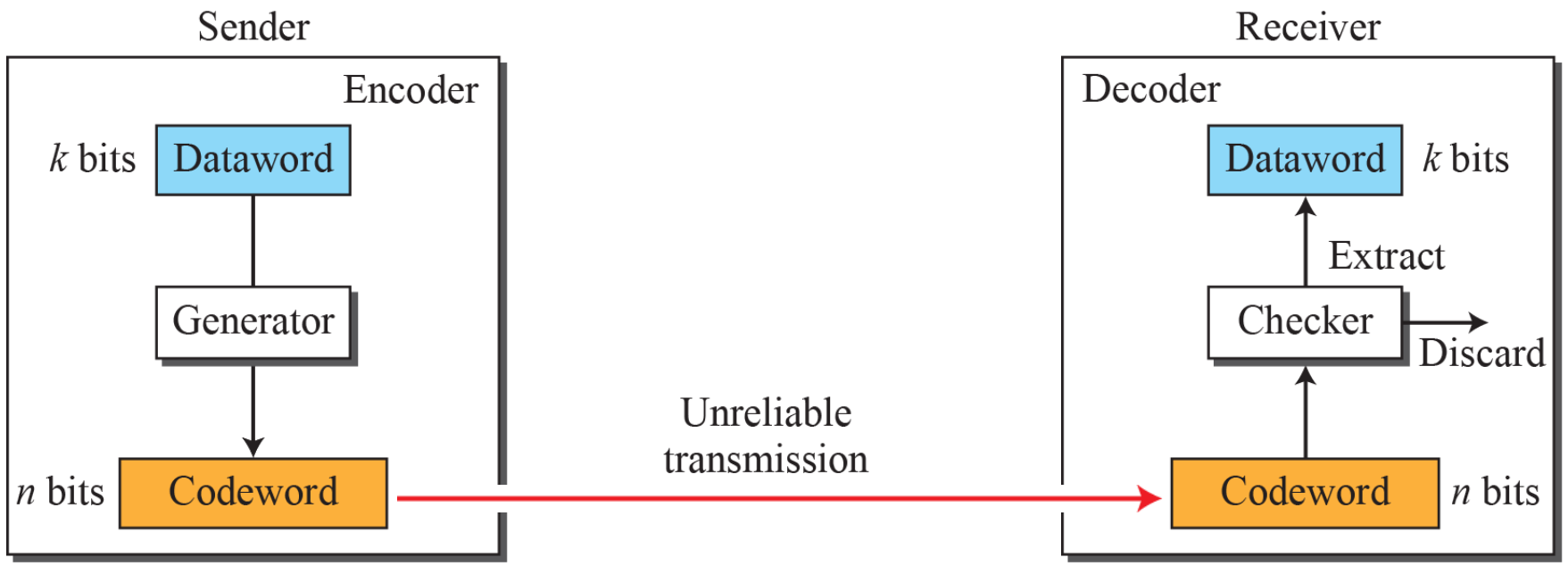
# Block Coding

- **Dataword**
  - The message is divided into blocks, each of $k$ bits.
- **Codeword**
  - $r$ redundant bits are added to each block to make the length $n = k + r$
- $2^k$ datawords < $2^n$ codewords
- The **block coding** process is **one-to-one**.
  - The **same dataword** is always encoded as the **same codeword**.
  - $(2^n - 2^k)$ codewords are not used (invalid codes) → the **trick** in error detection
- **Error detection** by **receiver** (3 conditions)
  1. The receiver has (or can find) a list of valid codewords.
  2. The original codeword has changed to an invalid one and it'll be discarded.
  3. The original codeword changes to a valid one! → makes the error **undetectable**

# Error Detection in Block Coding



# Error Detection in Block Coding




If the codeword is corrupted during transmission but the received word still matches a valid codeword, the **error remains undetected**.

# Hamming Distance

- **Definition**: The **Hamming distance** between two words (of the same size) is the number of differences between corresponding bits.
  - $d(x, y)$: Hamming distance between two words $x$ and $y$ as $d(x, y)$.
  - e.g., the Hamming distance between 0**1**1**0**11 and 0**0**1**1**11 is **two**.

# Hamming Distance

- **Definition**: The **Hamming distance** between two words (of the same size) is the number of differences between corresponding bits.
  - $d(x, y)$: Hamming distance between two words $x$ and $y$ as $d(x, y)$.
  - e.g., the Hamming distance between 0**1**1**0**11 and 0**0**1**1**11 is **two**.

- How does the Hamming distance help in error detection?
  - The Hamming distance between the **received codeword** and the **sent codeword** is the **number of bits** that are **corrupted** during transmission.
  - $d(x, y) = 0$ → no error
  - $d(x, y) > 0$ → error

# Hamming Distance (Cont.)

- **Calculating Hamming distance**
  - XOR operation ($\oplus$) on the two words and **count the number of 1s** in the result.
  - Example: $d(10101, 11110) = 3 \leftarrow 10101 \oplus 11110 = 01011$

# Hamming Distance (Cont.)

To **guarantee** the **detection of up to *s* errors** in all cases in a block coding scheme, the **minimum Hamming distance** between all pairs of valid codewords must be $d_{min} = s + 1$.

# Linear Block Codes

- Almost all block codes used today belong to a subset of block codes called **linear block codes**.
- The use of nonlinear block codes for error detection and correction is **not as widespread** because their structure makes theoretical analysis and implementation difficult.
- The formal definition of linear block codes requires the knowledge of **abstract algebra** (particularly **Galois fields**). → beyond our scope
- **Informal Definition for our purpose**: a **linear block code** is a code in which the **XOR** (addition modulo-2) of two valid codewords creates another valid codeword.

# Linear Block Codes – Parity-Check Code

- **Parity-check code** is a **linear block code**.
- A *k*-bit dataword is changed to an *n*-bit codeword where ***n* = *k* + 1**.
- **Parity bit**
  - The extra bit and it is selected such that the total number of 1s in the codeword becomes **even** (We only consider even parity here, while some implementations specify an odd number of 1s).

# Linear Block Codes – Parity-Check Code

- **Parity-check code** is a **linear block code**.
- A *k*-bit dataword is changed to an *n*-bit codeword where $n = k + 1$.
- **Parity bit**
  - The extra bit and it is selected such that the total number of 1s in the codeword becomes **even** (We only consider even parity here, while some implementations specify an odd number of 1s).
- The minimum Hamming distance for this category is $d_{min} = 2$, which means that the code is **a single-bit error-detecting code**.

# Linear Block Codes – Parity-Check Code

- The simple parity check, guaranteed to **detect one single error**, and can also **find any odd number of errors**.
  - E.g., the dataword is **10001** and the codeword **100010** is generated using **even parity bit** of **0** (added to the dataword as the rightmost bit). This codeword is transferred to another computer. In transit, the data is corrupted, and the computer receives the incorrect data **011010**. This codeword has **odd parity** (the number of 1s = 3), therefore, the codeword is **corrupted**.

# Simple Parity-Check Code – C(5, 4)

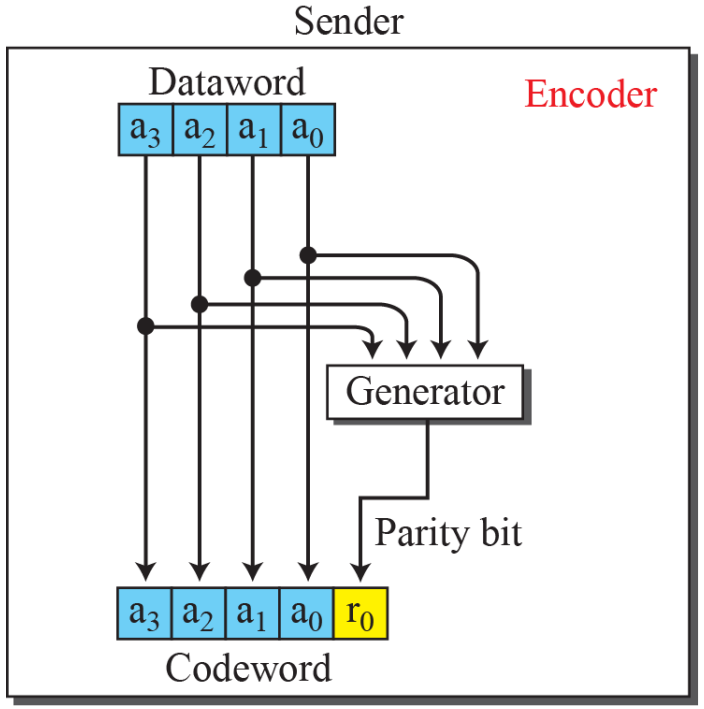
- $n$=5, $k$=4 (even parity)

| *Dataword* | *Codeword* | *Dataword* | *Codeword* |
|---|---|---|---|
| 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** |

# Encoder and Decoder for Simple Parity-Check Code

- The calculation is done in **modular arithmetic**.

$r_0 = a_3 + a_2 + a_1 + a_0$ **(modulo-2)**

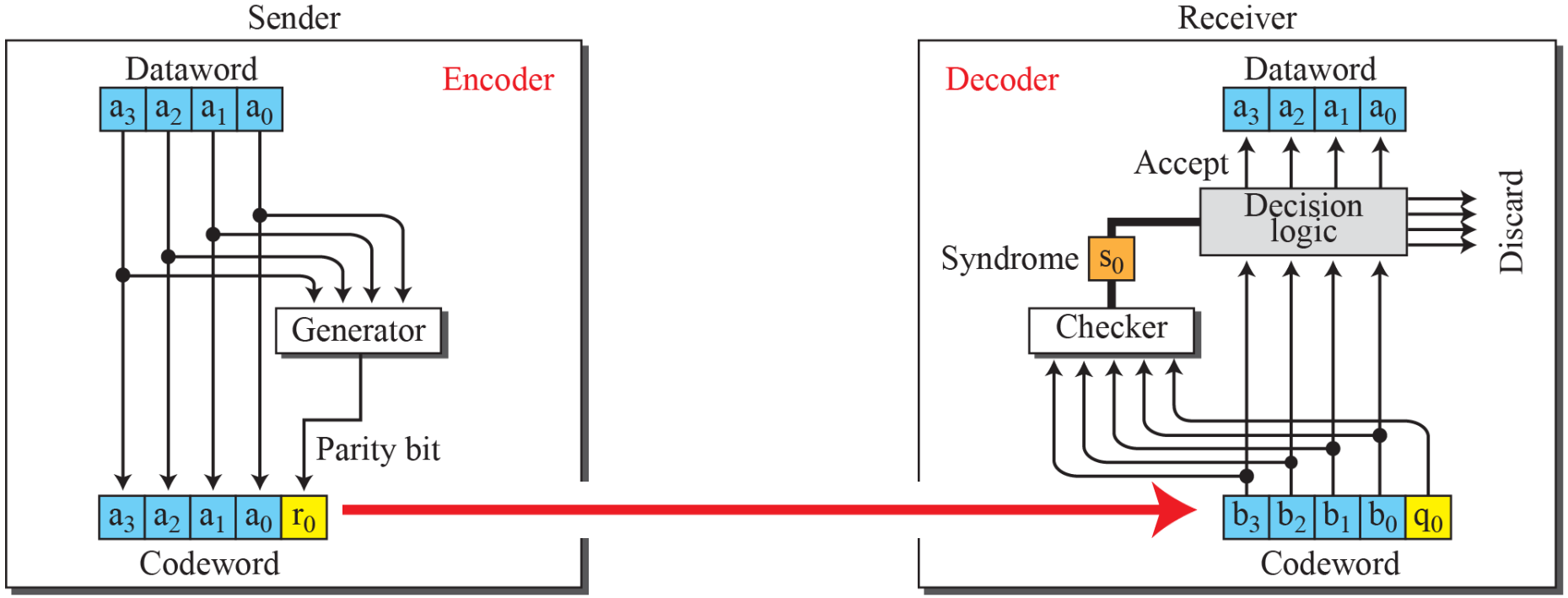


# Encoder and Decoder for Simple Parity-Check Code

- The calculation is done in **modular arithmetic**.

$r_0 = a_3 + a_2 + a_1 + a_0$ (modulo-2)

$s_0 = b_3 + b_2 + b_1 + b_0 + q_0$ (modulo-2)

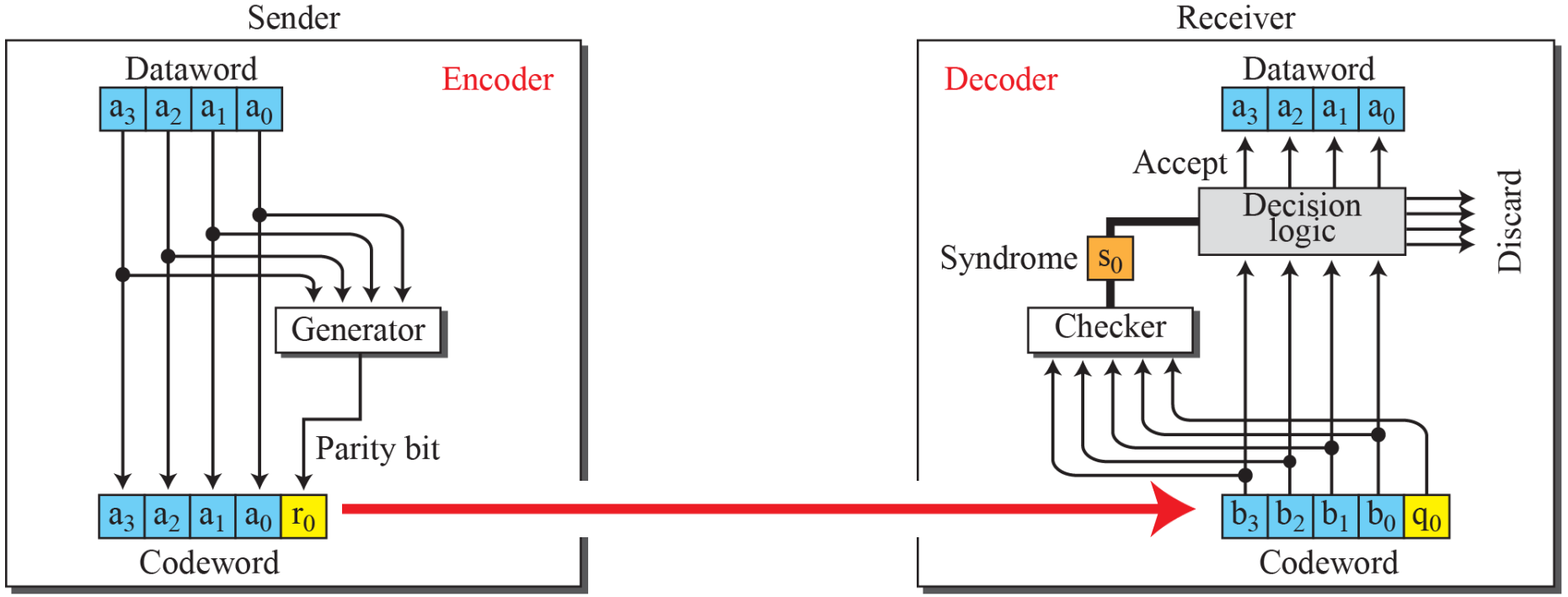


# Encoder and Decoder for Simple Parity-Check Code

- The calculation is done in **modular arithmetic**.

$r_0 = a_3 + a_2 + a_1 + a_0$ (modulo-2)

$s_0 = b_3 + b_2 + b_1 + b_0 + q_0$ (modulo-2)




**If the syndrome is 0, there is no detectable error in the received codeword.**

# Linear Block Codes – Cyclic Codes

- **Cyclic codes**
  - Special linear block codes with **one extra property**: if a codeword is cyclically shifted (**rotated**), the result is another codeword
  - E.g., If **1011000** is a codeword → cyclic left-shift → **0110001** is also a codeword

# Linear Block Codes – Cyclic Codes

- **Cyclic codes**
  - Special linear block codes with **one extra property**: if a codeword is cyclically shifted (**rotated**), the result is another codeword
  - E.g., If **1011000** is a codeword → cyclic left-shift → **0110001** is also a codeword
- **Advantages** of cyclic codes
  - Very good performance in detecting single-bit errors, double errors, and burst errors.
  - Can easily be implemented in hardware and software.
  - Especially fast when implemented in hardware.

# Linear Block Codes – Cyclic Codes

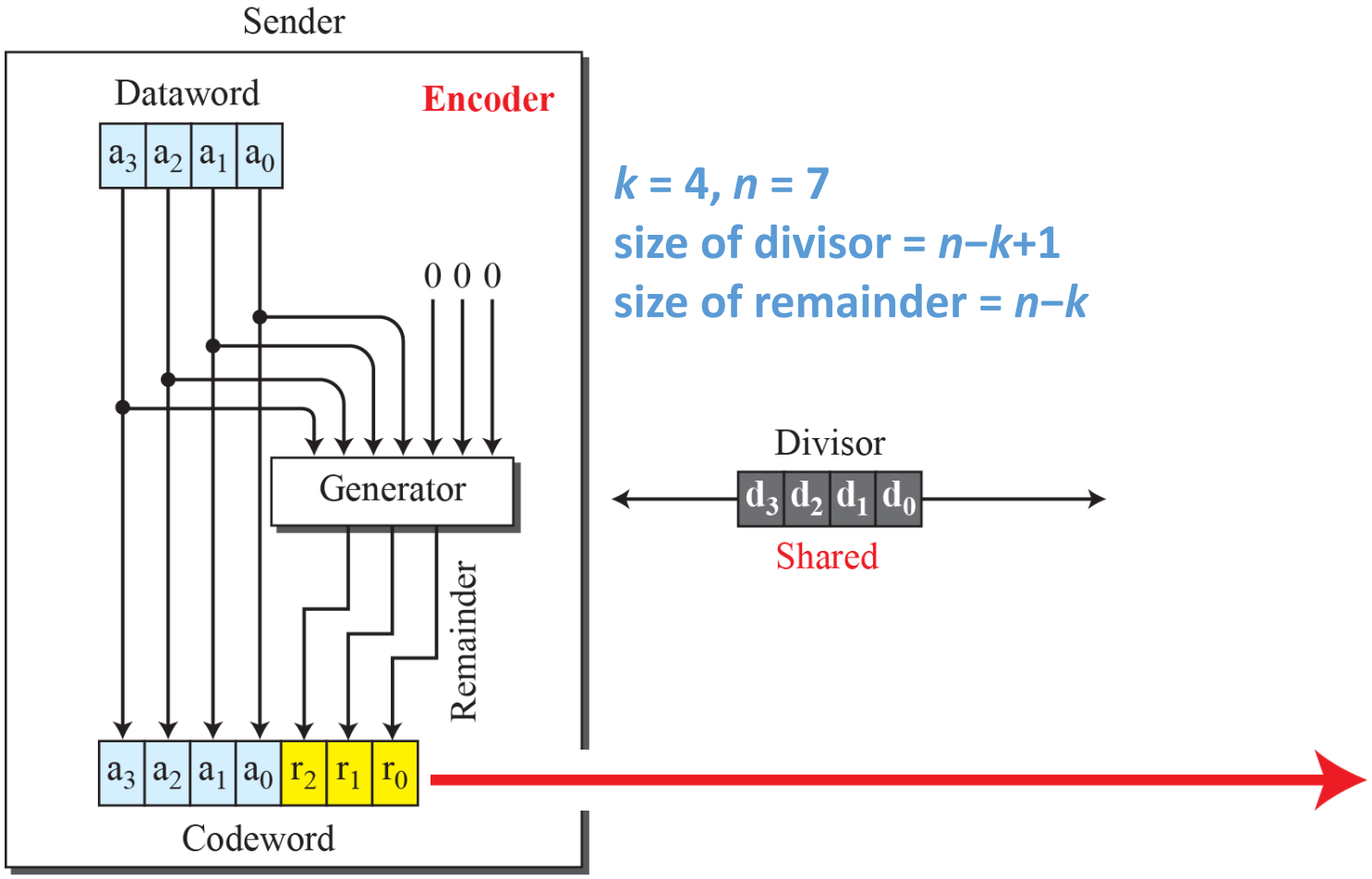
- **Cyclic codes**
  - Special linear block codes with **one extra property**: if a codeword is cyclically shifted (**rotated**), the result is another codeword
  - E.g., If **1011000** is a codeword → cyclic left-shift → **0110001** is also a codeword
- **Advantages** of cyclic codes
  - Very good performance in detecting single-bit errors, double errors, and burst errors.
  - Can easily be implemented in hardware and software.
  - Especially fast when implemented in hardware.
- **Our focus**
  - A subset of cyclic codes called **Cyclic Redundancy Check** (**CRC**), which is used in LANs and WANs

# A CRC Code with C(7, 4)

- $n$=7, $k$=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 |

# CRC Encoder and Decoder

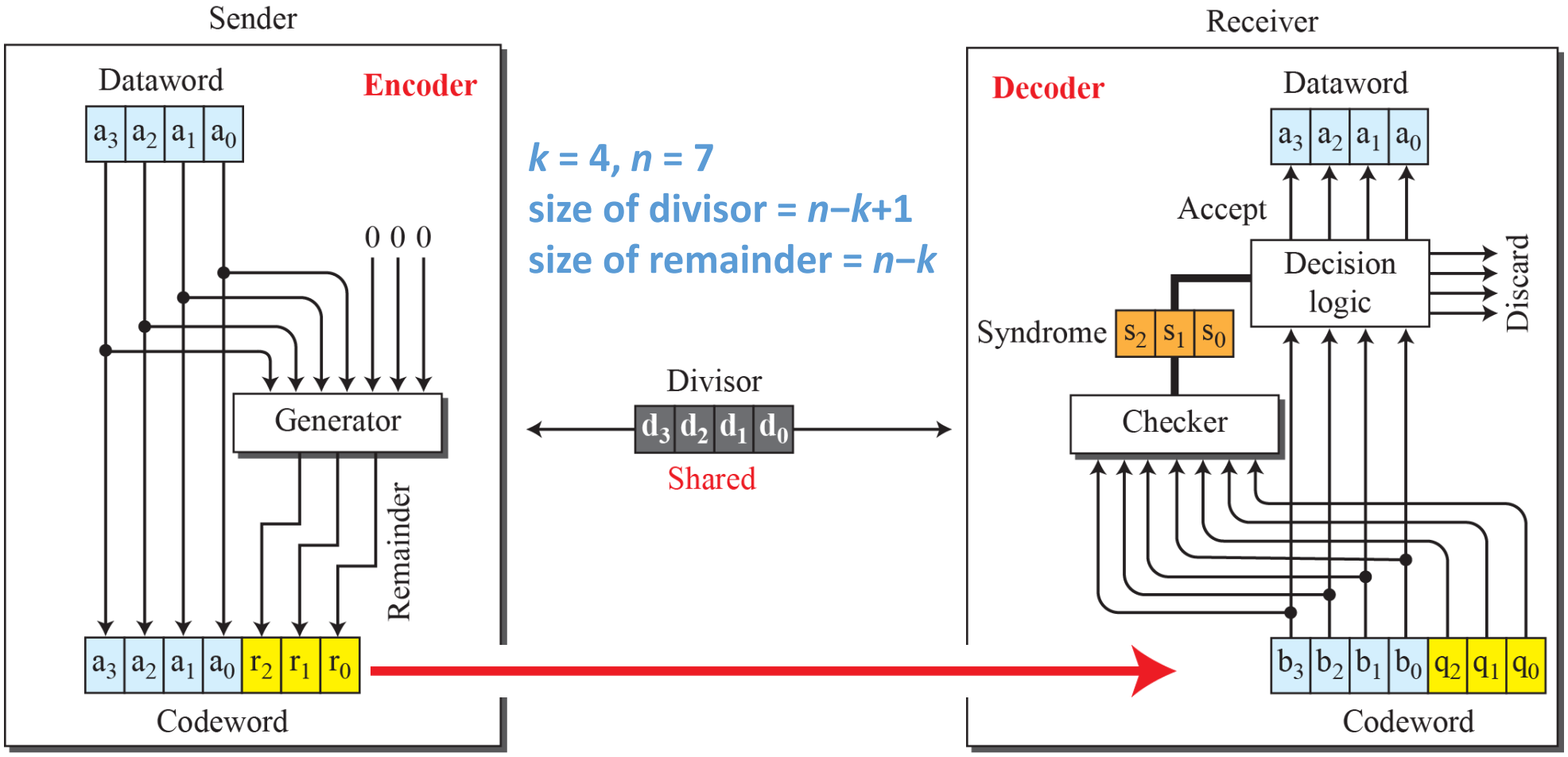



*k* = 4, *n* = 7

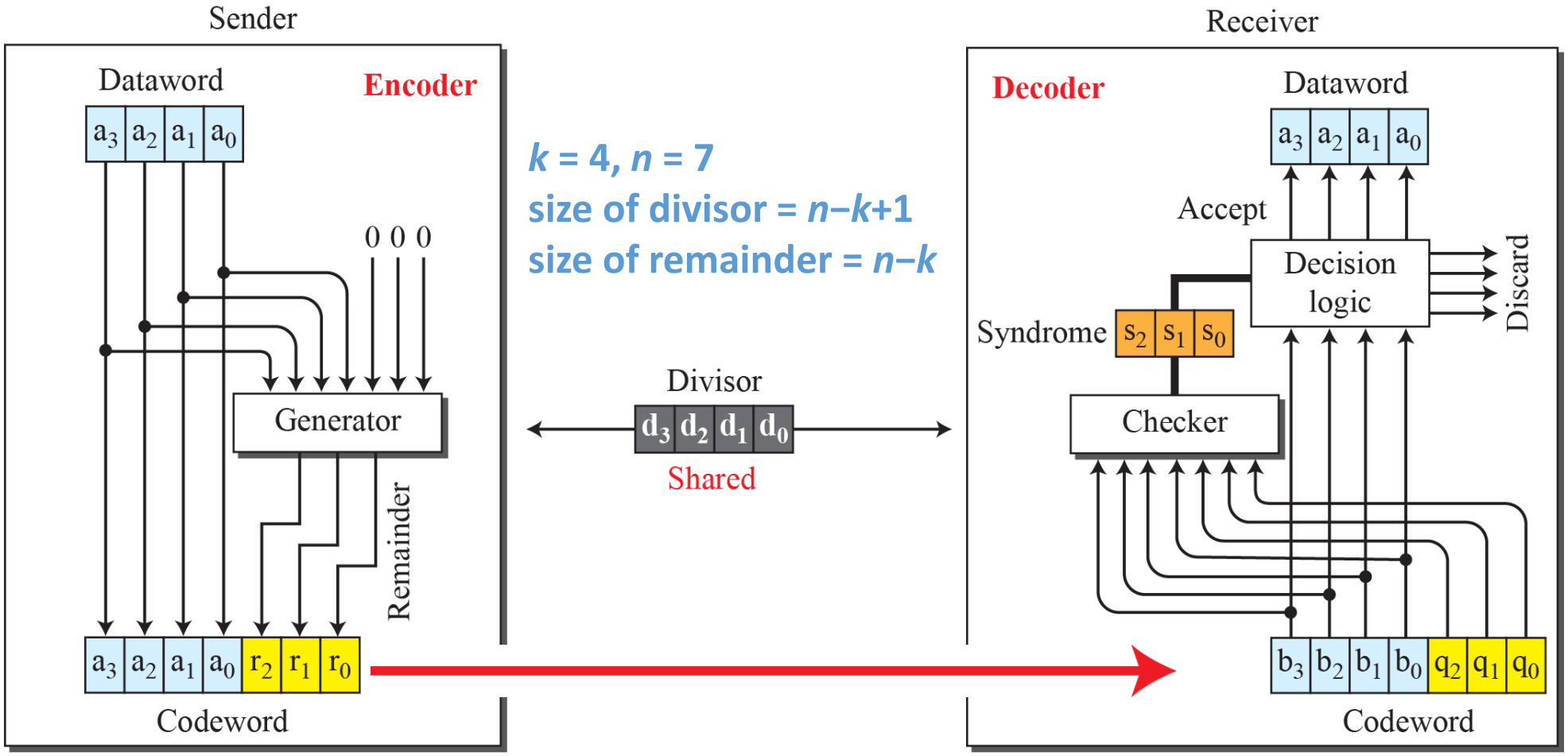
size of divisor = *n*–*k*+1

size of remainder = *n*–*k*

# CRC Encoder and Decoder

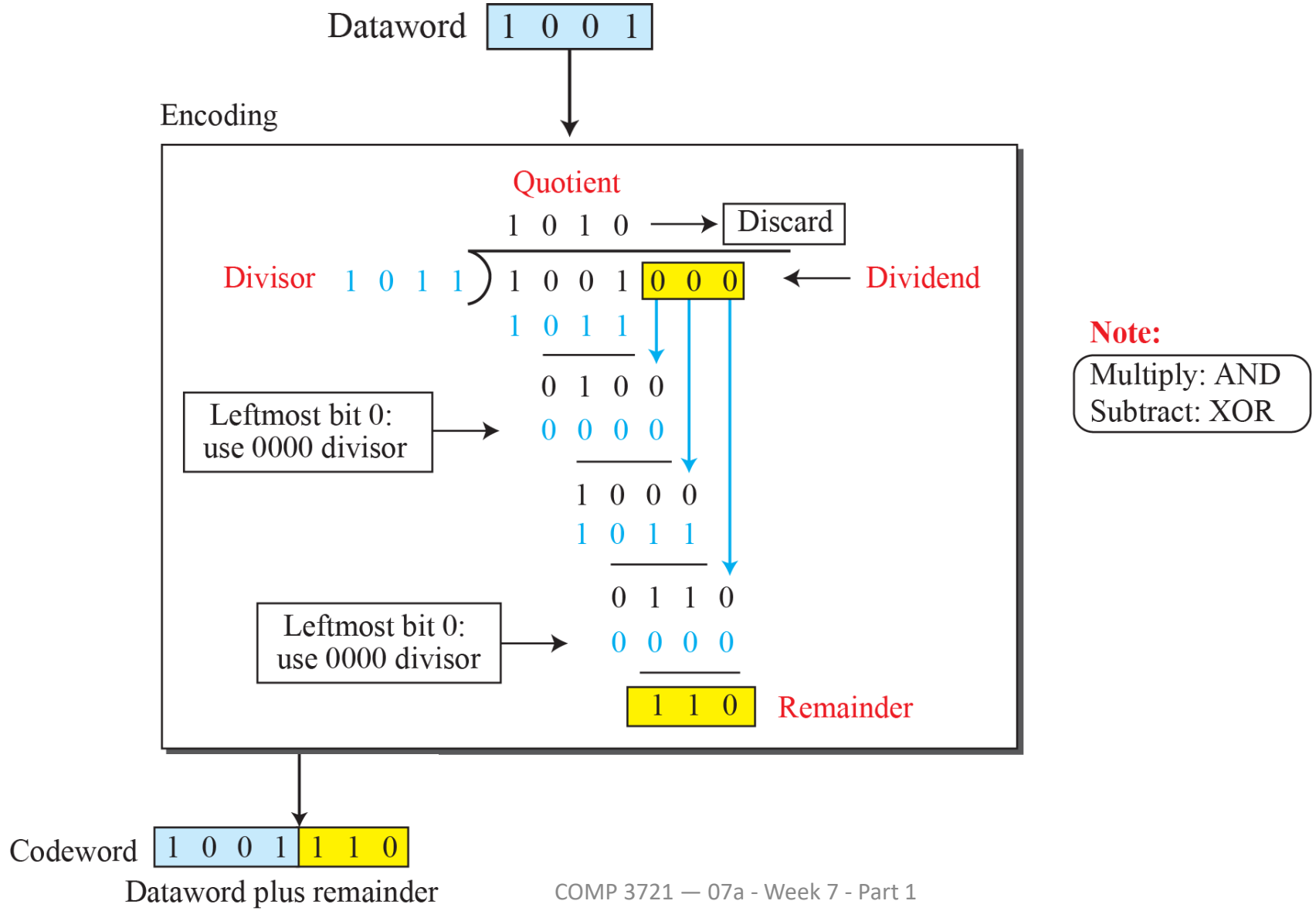


# CRC Encoder and Decoder




If syndrome is 0, either no bit is corrupted, or the decoder failed to detect any errors.

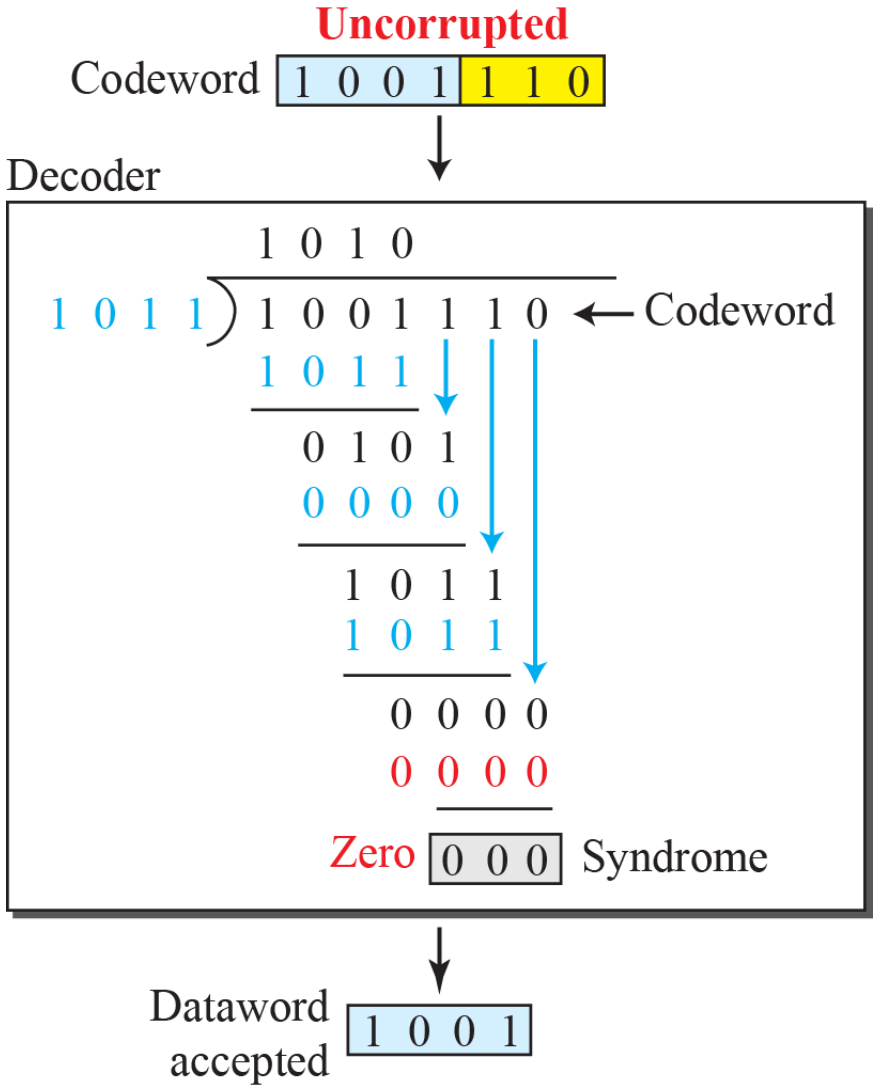
# Division in CRC Encoder



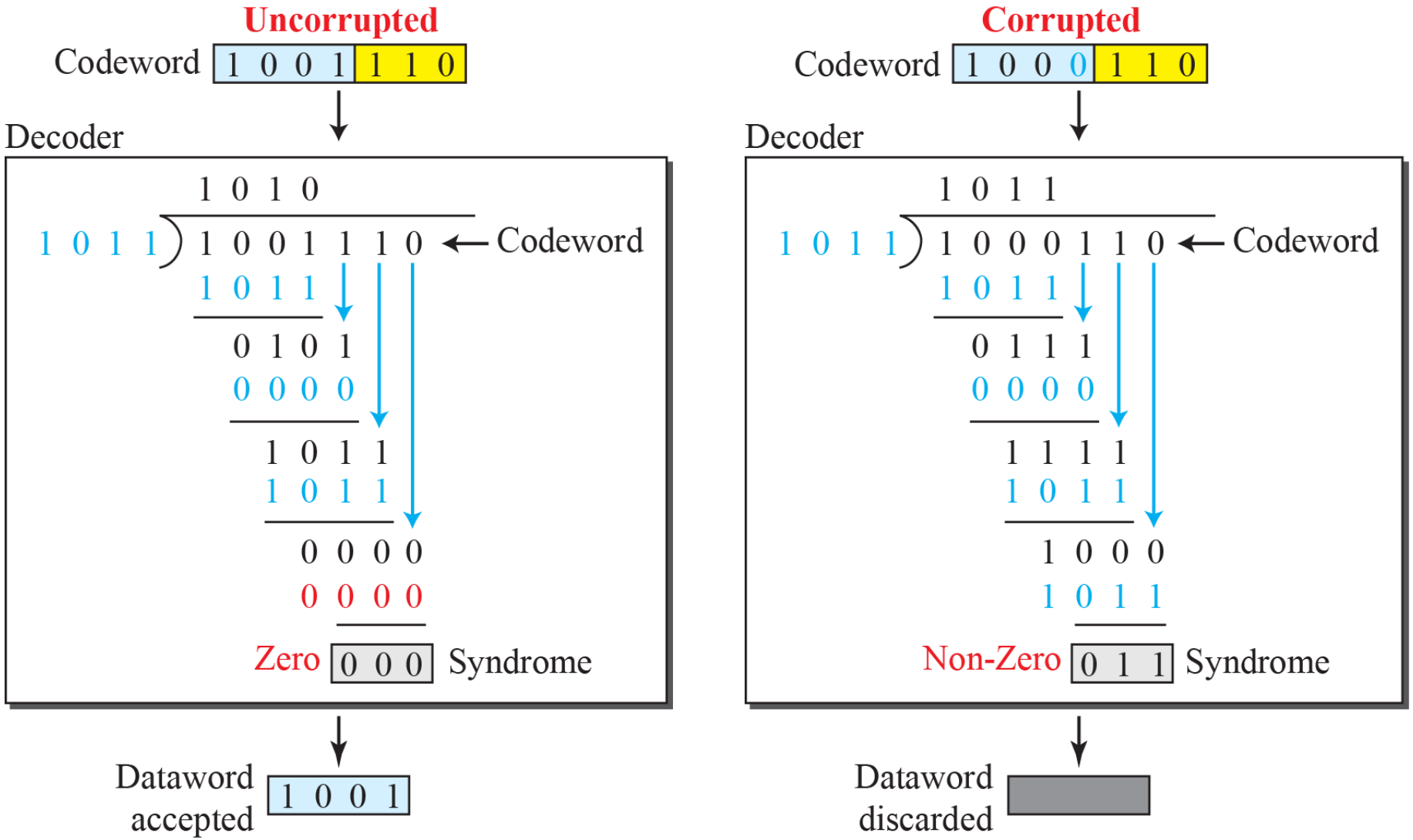

**Note:**

Multiply: AND
Subtract: XOR

# Division in CRC Decoder for Two Cases



# Division in CRC Decoder for Two Cases

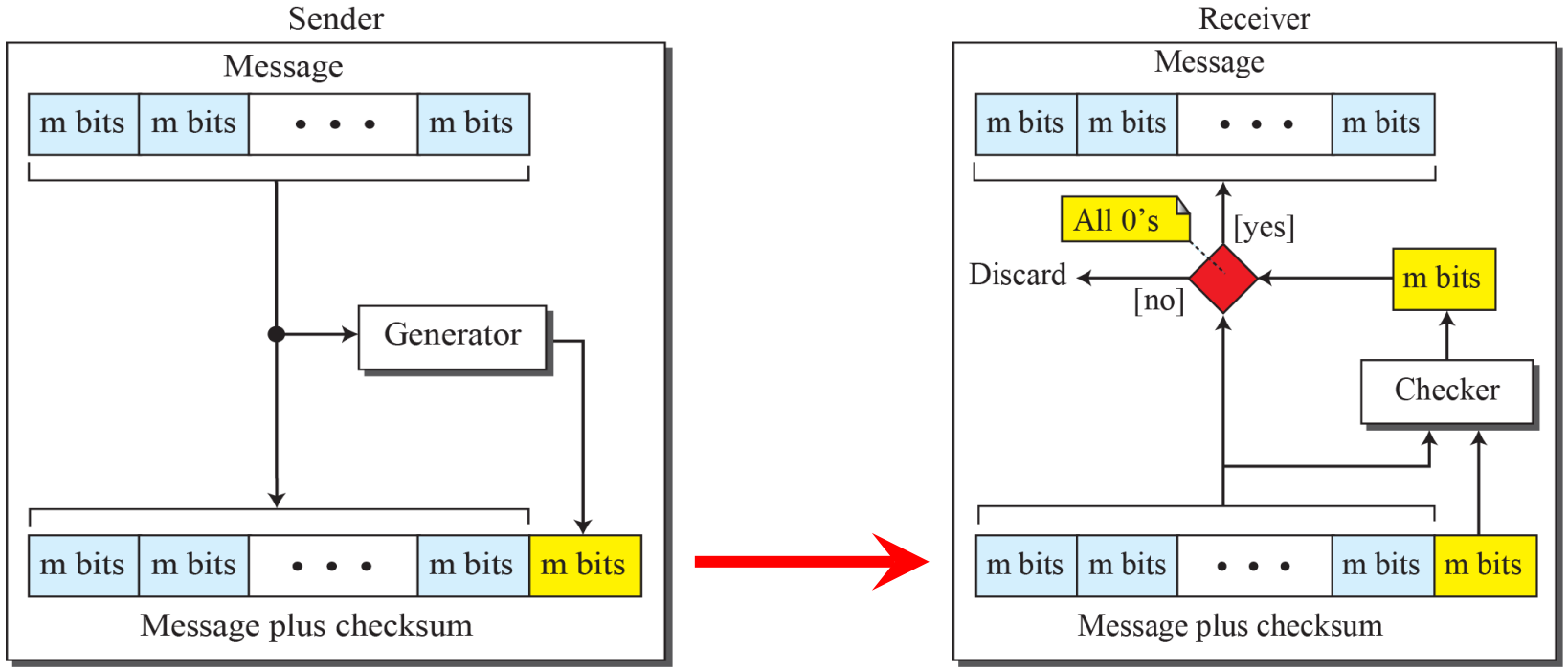


# Checksum

- An **error-detecting technique** that can be applied to a message of **any length**.
- Mostly used at the **network** and **transport layers** rather than the data-link layer.

# Checksum

- An **error-detecting technique** that can be applied to a message of **any length**.
- Mostly used at the **network** and **transport layers** rather than the data-link layer.



# Checksum – Example

- Suppose the message is a list of **five 4-bit numbers** that we want to send to a destination (**7**, **11**, **12**, **0**, **6**):
  - In addition to sending these numbers, we send the **sum** of the numbers.
  - E.g., 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 is not accepted.

# Checksum – Example

- The previous example has one issue:
  - Each number can be written as a 4-bit word (each is less than 15) **except for the sum**.

- **Solution**: Use **one's complement** arithmetic.

# Checksum – One's Complement Arithmetic

- Unsigned numbers between 0 and $2^m-1$ are represented using only *m* bits.

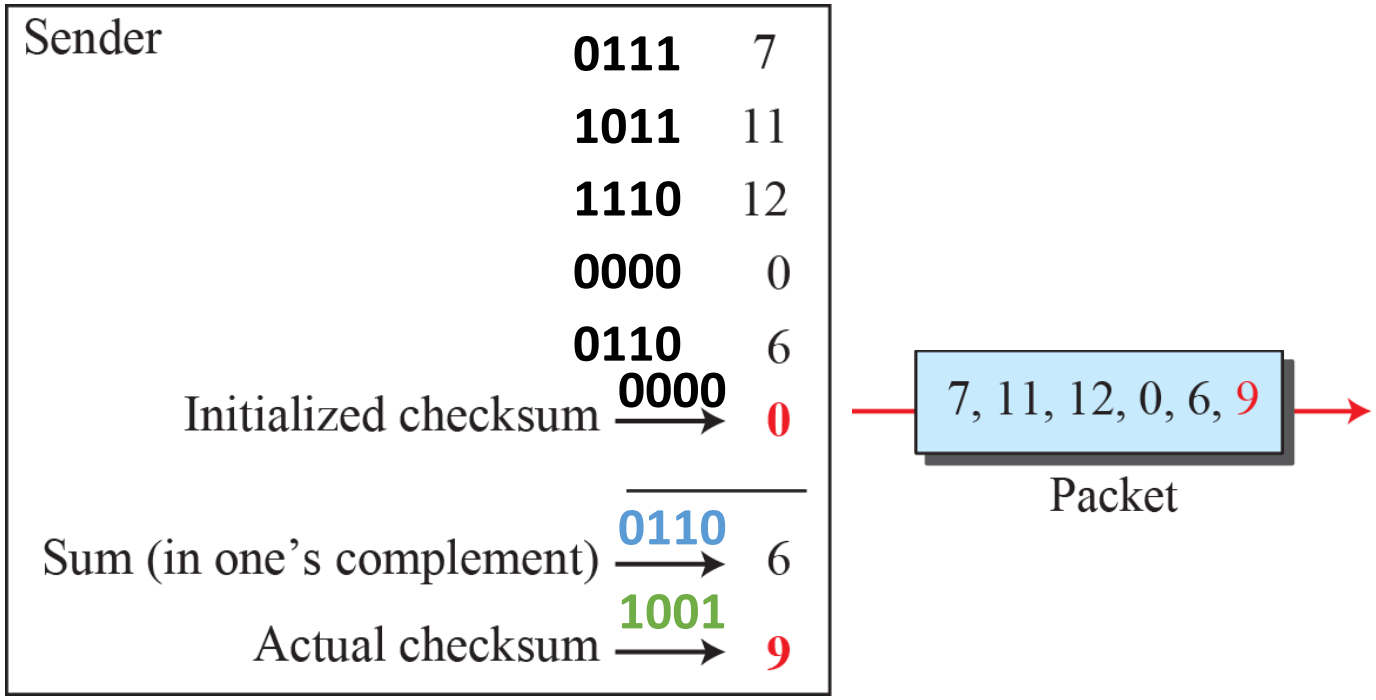
# Checksum – One's Complement Arithmetic

- Unsigned numbers between 0 and $2^m-1$ are represented 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**).
  - E.g., decimal number 36 in binary → **100100**, to change it to a 4-bit number: $(\mathbf{10})_2 + (\mathbf{0100})_2 = (\mathbf{0110})_2 \rightarrow (\mathbf{6})_{10}$

# Checksum – One's Complement Arithmetic

- Unsigned numbers between 0 and $2^m-1$ are represented 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**).
  - E.g., decimal number 36 in binary → **100100**, to change it to a 4-bit number: $(\mathbf{10})_2 + (\mathbf{0100})_2 = (\mathbf{0110})_2 \rightarrow (\mathbf{6})_{10}$
- The **complement** of a number is found by changing all 1s to 0s and all 0s to 1s.
  - The same as subtracting the number from $2^m-1$.
- In one's complement arithmetic, we have **two 0s**: one positive and one negative, which are complements of each other.
- The positive zero has all *m* bits set to 0 (**0000**); the negative zero has all bits set to 1 (it is $2^m-1$) (**1111**).

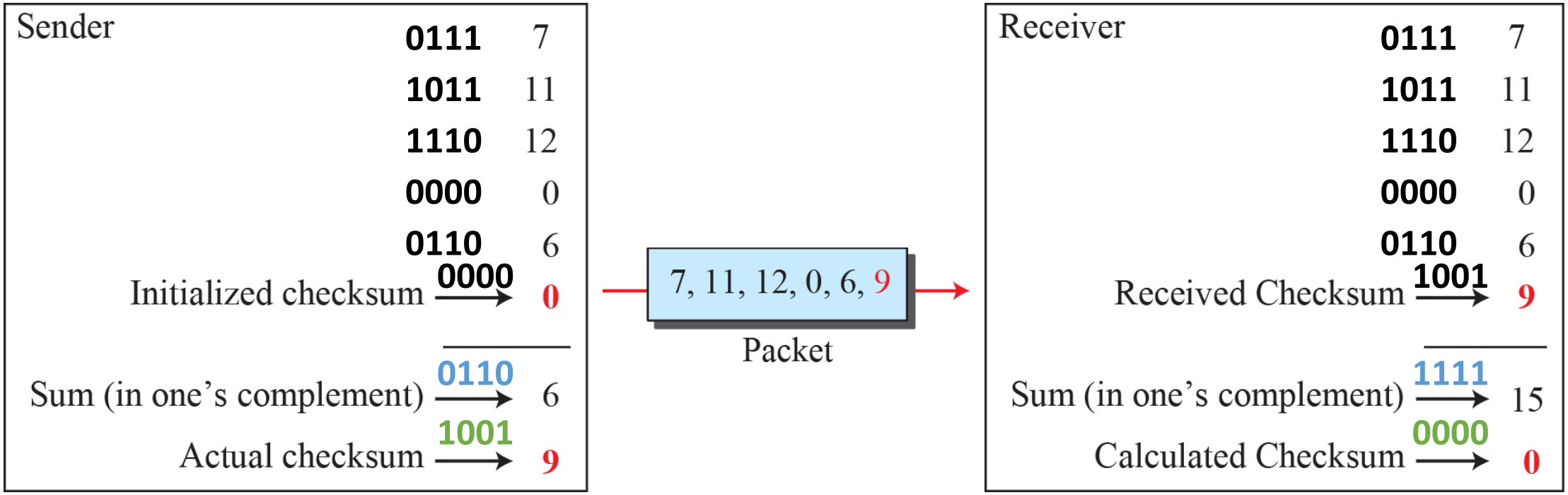
# Checksum – Example (4-bit binary numbers)




**$36 = (100100)_2 = (10)_2 + (0100)_2$**
**$= (0110)_2 = 6$**

**One's Complement of 6 is 9 = $(1001)_2$**

# Checksum – Example (4-bit binary numbers)




**$36 = (100100)_2 = (10)_2 + (0100)_2$**
**$= (0110)_2 = 6$**

**One's Complement of 6 is 9 = $(1001)_2$**

**$36 + 9 = 45 = (101101)_2 = (10)_2 + (1101)_2$**
**$= (1111)_2 = 15$**

**One's Complement of 15 is 0 = $(0000)_2$**

# Procedure to Calculate the Traditional Checksum

- Traditionally, the Internet has used a 16-bit 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. |

# Forward Error Correction (FEC)

- Retransmission of corrupted and lost packets is not useful for real-time **multimedia transmission**.
  - Due to **unacceptable delay** in reproducing.
- In such applications, we need to correct the error or reproduce the packet immediately.

# FEC Techniques

- Hamming distance
- Using XOR
- Chunk interleaving
- …

# FEC Techniques – Hamming Distance

- To **detect *s* errors**, $d_{min} = s + 1$. For error correction, we need more distance.
- To **correct *t* errors**, we need to have $\boldsymbol{d_{min} = 2t + 1}$.
  - E.g., if we want to correct 10 bits in a packet, we need to make the minimum hamming distance 21 bits → lot of redundant bits need to be sent with the data

# FEC Techniques – Using XOR

- Using the property of XOR operation:

$$\mathbf{R} = \mathbf{P}_1 \oplus \mathbf{P}_2 \oplus \ldots \oplus \mathbf{P}_i \oplus \ldots \oplus \mathbf{P}_N$$

- This means:

$$\mathbf{P}_i = \mathbf{P}_1 \oplus \mathbf{P}_2 \oplus \ldots \oplus \mathbf{R} \oplus \ldots \oplus \mathbf{P}_N$$

# FEC Techniques – Using XOR

- Using the property of XOR operation:

$$\mathbf{R} = \mathbf{P}_1 \oplus \mathbf{P}_2 \oplus \ldots \oplus \mathbf{P}_i \oplus \ldots \oplus \mathbf{P}_N$$

- This means:

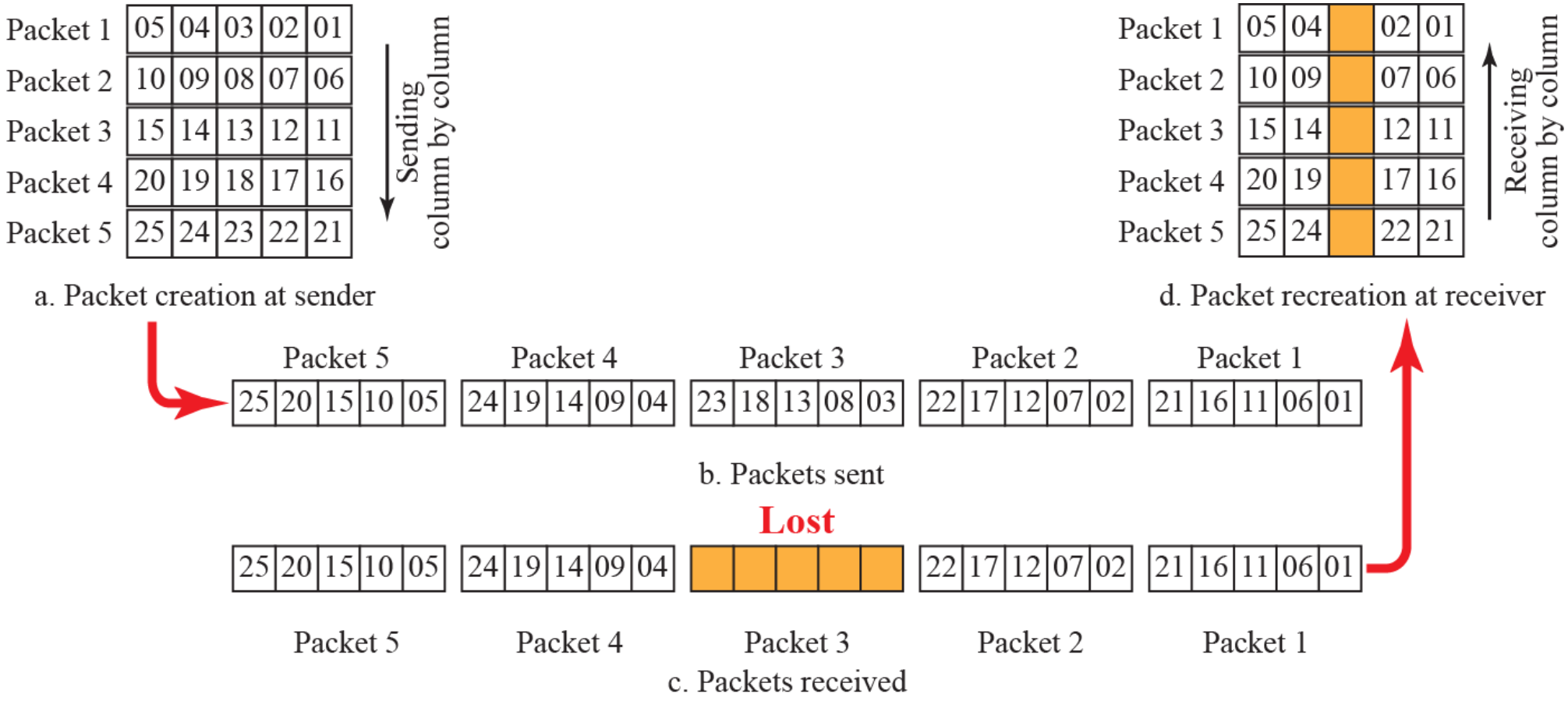
$$\mathbf{P}_i = \mathbf{P}_1 \oplus \mathbf{P}_2 \oplus \ldots \oplus \mathbf{R} \oplus \ldots \oplus \mathbf{P}_N$$

- We can divide a packet into *N* chunks, create the XOR of all the chunks and send *N*+1 Chunks. If any chunk is lost or corrupted, it can be created at the receiver site.
- (If *N* = 4, it means that we need to send 25 percent extra data and be able to correct the data if only one out of four chunks is lost.)

# FEC Techniques – Chunk Interleaving

- Allowing some small chunks to be missing at the receiver.
  - Not all the chunks belonging to the same packet can be missing.
  - We can afford to let one chunk be missing in each packet.

# Chunk Interleaving – Example



# Summary

- Possibility of data corruption during transmission
- Sending redundant bits with data for error detection/correction
- Block coding
- Hamming distance
- Cyclic codes as special linear block codes
- Internet checksum to detect errors in messages of any size

# References

[1] Behrouz A.Forouzan, Data Communications & Networking with TCP/IP Protocol Suite, 6th Ed, 2022, McGraw-Hill companies.

# Reading

- Chapter 3 of the textbook, section 3.2.2.
- Chapter 3 of the textbook, section 3.6 (Practice Test)