Error Detection and Correction Codes

Classification:

- Single error detection:
    - Single parity bit (odd/even)
    - 2-out-of-5 code (5-bits)
    - 63210 code (5-bits)
    - 51111 code (5-bits)
    - Shift-counter code (5-bits)
    - Biquinary code(50 43210) (7-bits)
    - Ring-counter code (10-bits)

- Single error correction:
    - Hamming code (7-bits, 15-bits) (depending upon data bits)- can be extended to any number of data bits
    - Block parity

- Double error detection:
    - Check sum
    - Block parity

Parity bit method:

- Single error detection: parity bit (odd/even)
- P1000
- 1000P
- P=0 or 1
- No. of 1’s is odd
- P=0(odd), P=1 (even)

Table 1.6 Error-detecting codes

Decimal	Even-parity BCD					2-out-of-5				
digit	8	4	2	1	*p*	0	1	2	4	7
0	0	0	0	0	0	0	0	0	1	1
1	0	0	0	1	1	1	1	0	0	0
2	0	0	1	0	1	1	0	1	0	0
3	0	0	1	1	0	0	1	1	0	0
4	0	1	0	0	1	1	0	0	1	0
5	0	1	0	1	0	0	1	0	1	0
6	0	1	1	0	0	0	0	1	1	0
7	0	1	1	1	1	1	0	0	0	1
8	1	0	0	0	1	0	1	0	0	1
9	1	0	0	1	0	0	0	1	0	1

The *minimum distance* of a code is the smallest number of bits in which any two code words differ. Thus, the minimum distance of the BCD or the Excess-3 codes is one, while that of the codes in Table 1.6 is two. Clearly, *a code is an error-detecting code if and only if its minimum distance is two or more.*

Error-Detection Codes

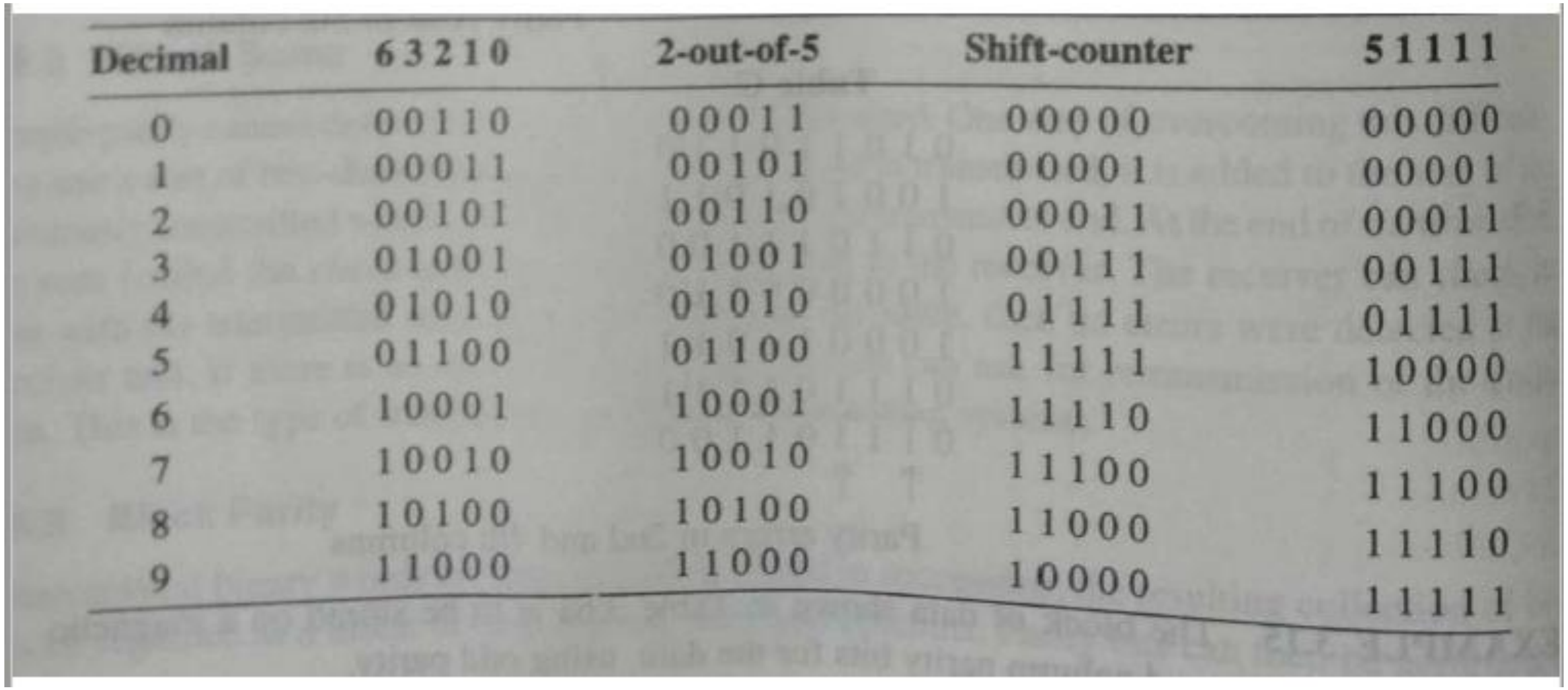
The *2-out-of-5 code* is sometimes used in communications work. It utilizes five bits to represent the ten decimal digits, so it is a form of BCD code. Each code word has exactly two 1s, a convention that facilitates decoding and provides for better error detection than the single-parity-bit method. If more or less than two 1s appear, an error is indicated.

The *63210 BCD code* is also characterized by having exactly two 1s in each of the 5-bit groups. Like the 2-out-of-5 code, it provides reliable error detection and is used in some applications.

The *biquinary* (two-five) *code* is used in certain counters and is composed of a 2-bit group and a 5-bit group, each with a single 1. Its weights are 50 43210. The 2-bit group, having weights of five and zero, indicates whether the number represented is less than, equal to, or greater than 5. The 5-bit group indicates the count above or below 5.

The *ring counter code* has ten bits, one for each decimal digit, and a single 1 makes error detection possible. It is easy to decode but wastes bits and requires more circuitry to implement than the 4-bit or 5-bit codes. The name is derived from the fact that the code is generated by a certain type of shift register, a ring counter. Its weights are 9876543210.

Each of these codes is listed in Table 2–11. You should realize that this is not an exhaustive coverage of all codes but simply an introduction to some of them.



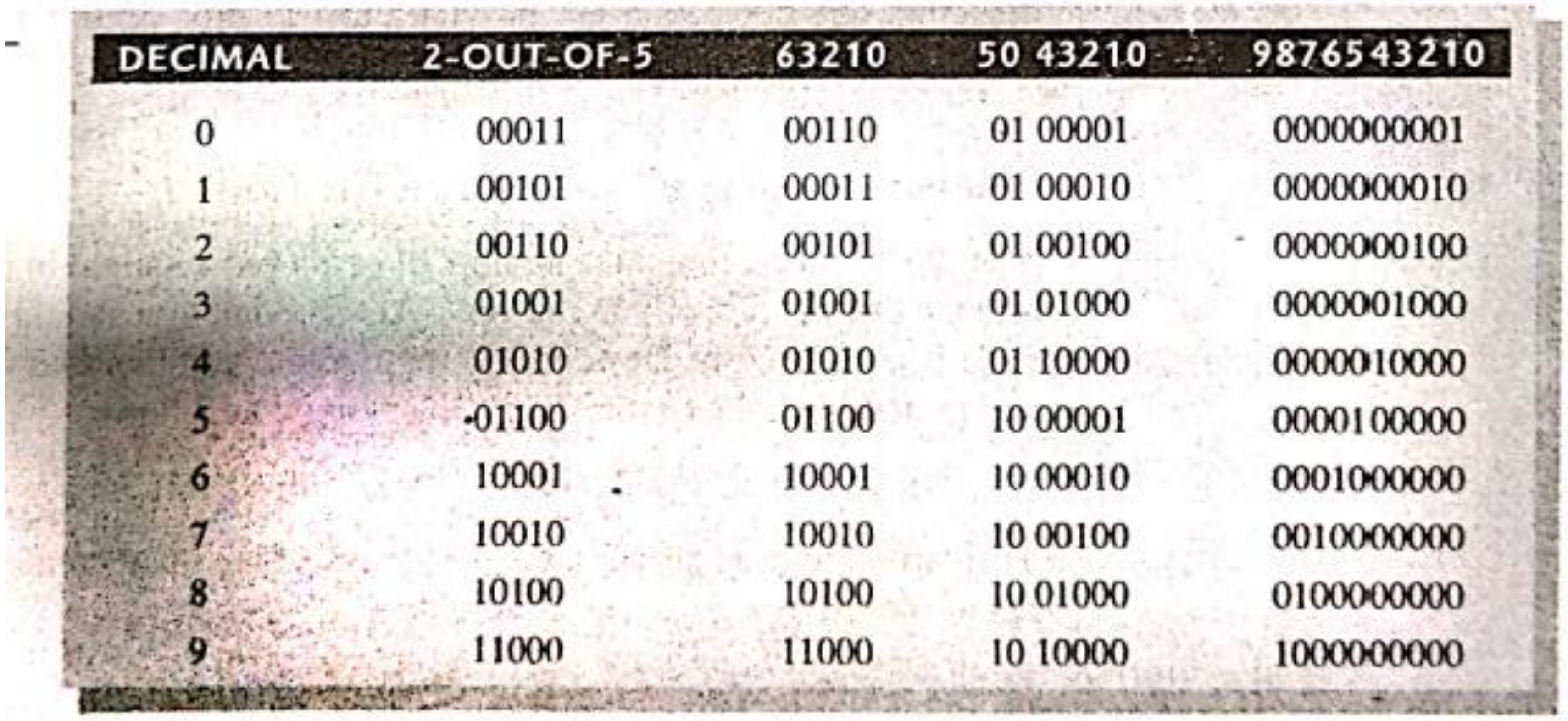
Decimal	6 3 2 1 0	2-out-of-5	Shift-counter	5 1 1 1 1
0	00110	00011	00000	00000
1	00011	00101	00001	00001
2	00101	00110	00011	00011
3	01001	01001	00111	00111
4	01010	01010	01111	01111
5	01100	01100	11111	10000
6	10001	10001	11110	11000
7	10010	10010	11100	11100
8	10100	10100	11000	11110
9	11000	11000	10000	11111

The biquinary code shown in Table 3.8 is a weighted 7-bit BCD code. It is a parity data code. Note that each code group can be regarded as consisting of a 2-bit subgroup and a 5-bit subgroup, and each of these subgroups contains a single 1. Thus, it has the error-checking feature, for each code group has exactly two 1s and each subgroup has exactly one 1. The weights of the bit positions are 50 43210. Since there are two positions with weight 0, it is possible to encode decimal 0 with a group containing 1s, unlike other weighted codes. The biquinary code is used in the Abacus.

Table 3.8 The biquinary code

Decimal digit	Biquinary code						
	5	**0**	**4**	**3**	**2**	**1**	**0**
0	0	1	0	0	0	0	1
1	0	1	0	0	0	1	0
2	0	1	0	0	1	0	0
3	0	1	0	1	0	0	0
4	0	1	1	0	0	0	0
5	1	0	0	0	0	0	1
6	1	0	0	0	0	1	0
7	1	0	0	0	1	0	0
8	1	0	0	1	0	0	0
9	1	0	1	0	0	0	0

Decimal digit	Ring-counter code									
	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	1	0	0
3	0	0	0	0	0	0	1	0	0	0
4	0	0	0	0	0	1	0	0	0	0
5	0	0	0	0	1	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0	0
7	0	0	1	0	0	0	0	0	0	0
8	0	1	0	0	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0	0



DECIMAL	2-OUT-OF-5	63210	50 43210	9876543210
0	00011	00110	01 00001	0000000001
1	00101	00011	01 00010	0000000010
2	00110	00101	01 00100	0000000100
3	01001	01001	01 01000	0000001000
4	01010	01010	01 10000	0000010000
5	01100	01100	10 00001	0000100000
6	10001	10001	10 00010	0001000000
7	10010	10010	10 00100	0010000000
8	10100	10100	10 01000	0100000000
9	11000	11000	10 10000	1000000000

Weighted code? Code value is according to weights assigned as per position Positional codes
8421 code 63210 1=1+0 9=6+3 Except for 0 decimal
Biquinary code 50 as upper weight sub-band
43210 as lower weight sub-band
0 = 0+0
1=0+1
(5) Decimal = 5+0= 10 00001

A code is said to be *error-correcting* if the correct code word can always be deduced from the erroneous word.
Basic principles in constructing a Hamming error-correcting code:

- To each group of ***m information*** **or** ***message digits***, ***k parity-checking digits***, denoted $\mathbf{p_1, p_2, \ldots\ldots p_k}$, are added to form an ($m$ + k)-digit code.
- The location of each of the ***m + k*** digits within a code word is assigned a decimal value; one starts by assigning **a 1 to the most significant digit** and ***m + k*** **to the least significant digit**.
- Then ***k*** **parity checks are considered as** $\mathbf{c_1, c_2, \ldots\ldots c_k}$, whose value is equal to the decimal value assigned to the location of the erroneous digit when an error occurs and is equal **to zero if no error occurs**.
- This number is called the ***position*** **(or** ***location*****)** ***number***.

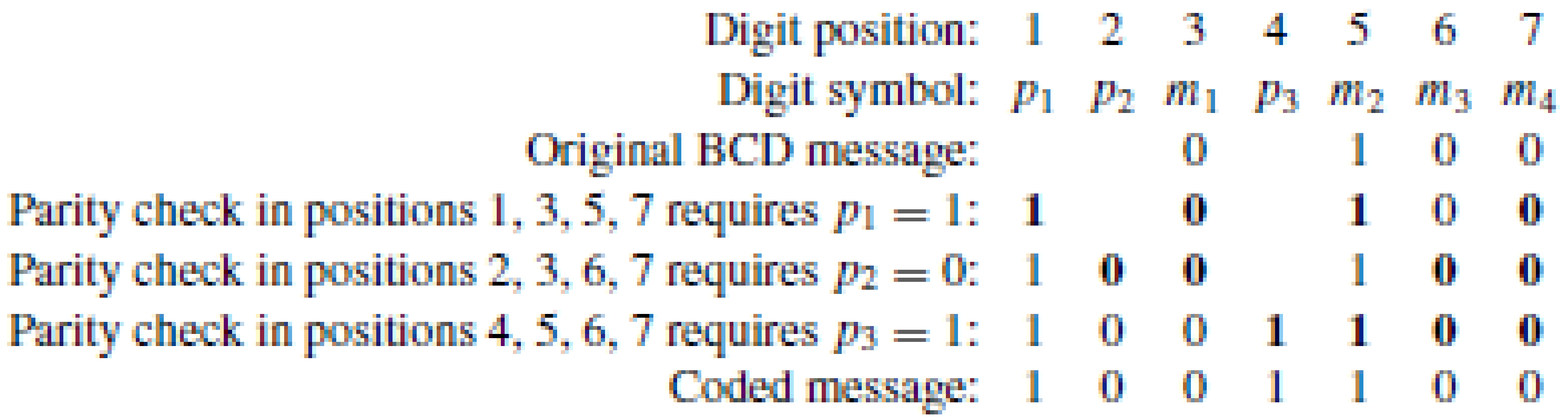
Inequality for Hamming Code: 2^k>= m+k+1.

M=4,
K=1, 2 >= 6
K=2, 4 >= 7
K=3, 8>= 8

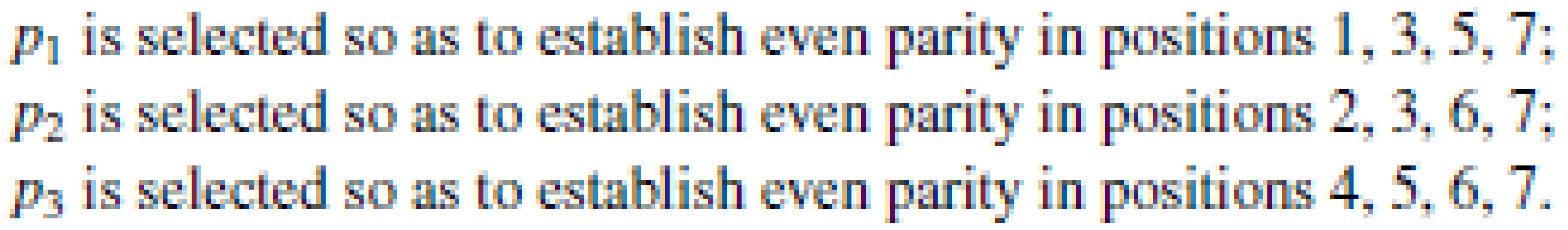
Example,

If original message is in **BCD** where ***m* = 4 (data bits)** then ***k* = 3 (parity bits),**
Length of code is 7.
3 Parity Check bits are used to check error in the code.
If position number of parity check is equal to 101, it means that an error has occurred in position 5.
If, the position number is equal to 000, the message is correct.

Positions
1-001
2-010
3-011
4-100
5-101
6-110
7-111



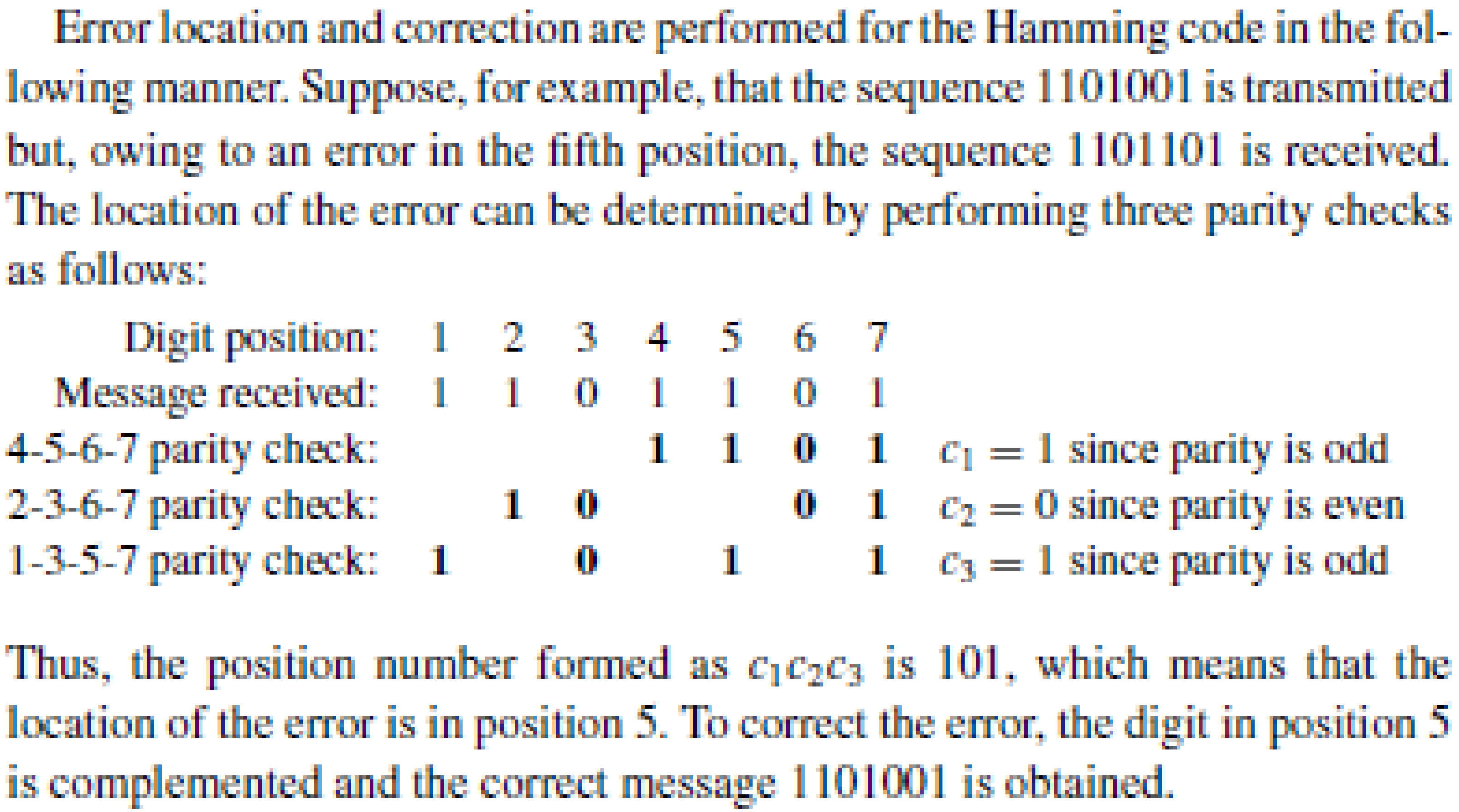
Digit position:	1	2	3	4	5	6	7
Digit symbol:	p_1	p_2	m_1	p_3	m_2	m_3	m_4
Original BCD message:			0		1	0	0
Parity check in positions 1, 3, 5, 7 requires $p_1 = 1$:	**1**		**0**		**1**	0	**0**
Parity check in positions 2, 3, 6, 7 requires $p_2 = 0$:	1	**0**	**0**		1	**0**	**0**
Parity check in positions 4, 5, 6, 7 requires $p_3 = 1$:	1	0	0	**1**	**1**	**0**	**0**
Coded message:	1	0	0	1	1	0	0



p_1 is selected so as to establish even parity in positions 1, 3, 5, 7;
p_2 is selected so as to establish even parity in positions 2, 3, 6, 7;
p_3 is selected so as to establish even parity in positions 4, 5, 6, 7.

Table 1.8 Hamming code for BCD

Decimal digit	Digit position and symbol						
	1 p_1	2 p_2	3 m_1	4 p_3	5 m_2	6 m_3	7 m_4
0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1
2	0	1	0	1	0	1	0
3	1	0	0	0	0	1	1
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	0	1	1	1	1
8	1	1	1	0	0	0	0
9	0	0	1	1	0	0	1



Error location and correction are performed for the Hamming code in the following manner. Suppose, for example, that the sequence 1101001 is transmitted but, owing to an error in the fifth position, the sequence 1101101 is received. The location of the error can be determined by performing three parity checks as follows:

Digit position:	1	2	3	4	5	6	7	
Message received:	1	1	0	1	1	0	1	
4-5-6-7 parity check:				**1**	**1**	**0**	**1**	$c_1 = 1$ since parity is odd
2-3-6-7 parity check:		**1**	**0**			**0**	**1**	$c_2 = 0$ since parity is even
1-3-5-7 parity check:	**1**		**0**		**1**		**1**	$c_3 = 1$ since parity is odd

Thus, the position number formed as $c_1c_2c_3$ is 101, which means that the location of the error is in position 5. To correct the error, the digit in position 5 is complemented and the correct message 1101001 is obtained.

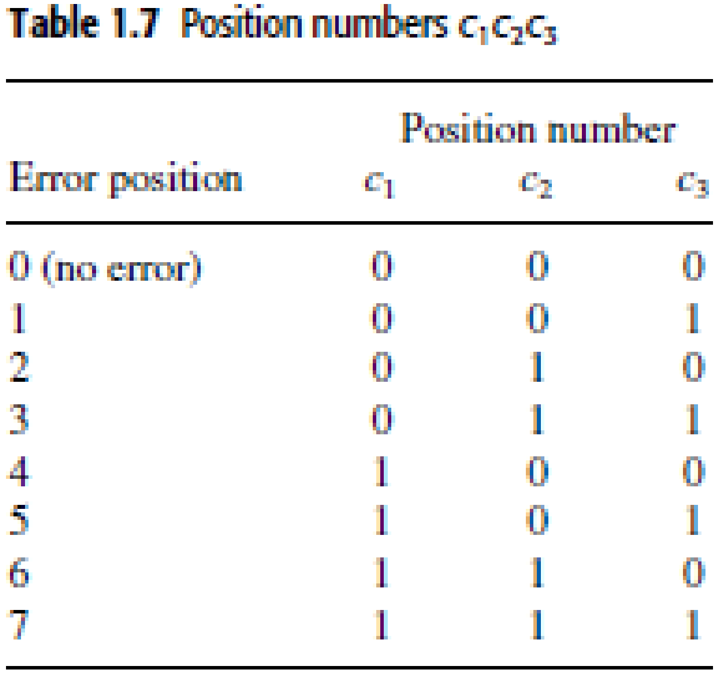


Table 1.7 Position numbers $c_1c_2c_3$

Error position	Position number c_1	c_2	c_3
0 (no error)	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Transmitted code: 1101**0**01

Received code: 1101**1**01

Double Error Detection Codes

- Checksum
- Block parity

Check sum - Transmitter end

- At sender side,
- If m bit checksum is used, the data unit to be transmitted is divided into segments of m bits.
- All the m bit segments are added.
- The result of the sum is then complemented using 1’s complement arithmetic.
- The value so obtained is called as **checksum**.
- The data along with the checksum value is transmitted to the receiver.

Receiver end:

- At receiver side,
- If m bit checksum is being used, the received data unit is divided into segments of m bits.
- All the m bit segments are added along with the checksum value.
- The value so obtained is complemented and the result is checked.
- If the result is zero,
- Receiver assumes that no error occurred in the data during the transmission.
- Receiver accepts the data.

If the result is non-zero,

- Receiver assumes that error occurred in the data during the transmission.
- Receiver discards the data and asks the sender for retransmission.

Example:

- Consider the data unit to be transmitted is-
    - 10011001111000100010010010000100
- Consider 8 bit checksum is used.
- Now, all the segments are added and the result is obtained as-
- 10011001 + 11100010 + 00100100 + 10000100 = 1000100011
- Since the result consists of 10 bits, so extra 2 bits are wrapped around.
- 00100011 + 10 = 00100101 (8 bits)
- Now, 1’s complement is taken which is 11011010.
- Thus, checksum value = 11011010
- The data along with the checksum value is transmitted to the receiver.

No error

- At receiver side,
- The received data unit is divided into segments of 8 bits.
- All the segments along with the checksum value are added.
- 10011001 + 11100010 + 00100100 + 10000100 = 1000100011
- Since the result consists of 10 bits, so extra 2 bits are wrapped around.
- 00100011 + 10 = 00100101 (8 bits)
- Sum of all segments + Checksum value = 00100101 + 11011010 = 11111111
- Complemented value = 00000000
- Since the result is 0, receiver assumes no error occurred in the data and therefore accepts it.

1 error

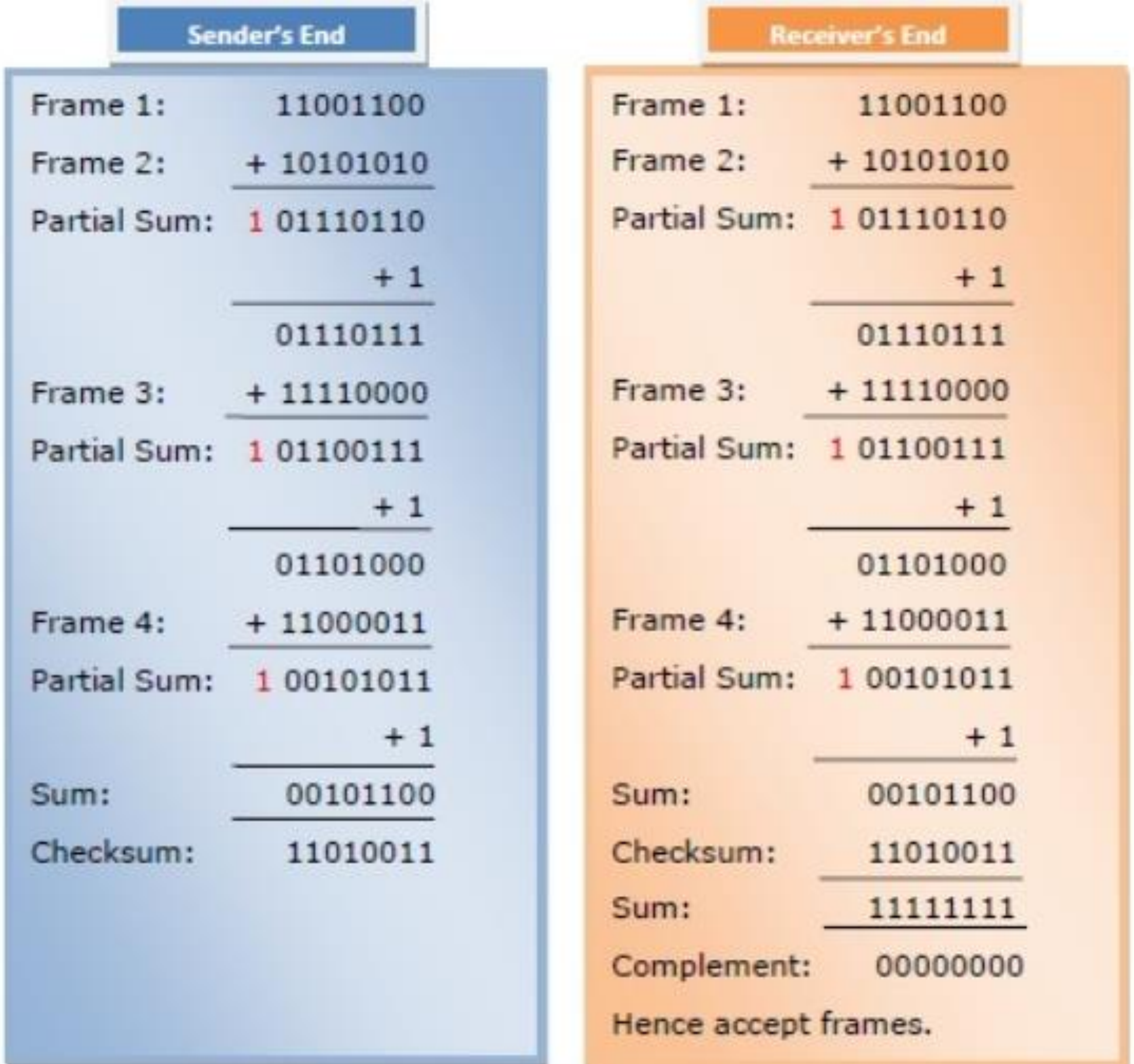
- At receiver side,
- The received data unit is divided into segments of 8 bits.
- All the segments along with the checksum value are added.
- 1001100**0** + 11100010 + 00100100 + 10000100 = 100010001**0**
- Since the result consists of 10 bits, so extra 2 bits are wrapped around.
- 0010001**0** + 10 = 0010010**0** (8 bits)
- Sum of all segments + Checksum value = 0010010**0** + 11011010 = 1111111**0**
- Complemented value = 0000000**1**
- Since the result is 1, receiver detects 1 error.

2 error (shows no error if 2 changes in separate data block)

- At receiver side,
- The received data unit is divided into segments of 8 bits.
- All the segments along with the checksum value are added.
- 1001100**<u>0</u>** + 1110001**<u>1</u>** + 00100100 + 10000100 = 100010001**<u>1</u>**
- Since the result consists of 10 bits, so extra 2 bits are wrapped around.
- 0010001**<u>1</u>** + 10 = 00100101 (8 bits) **<u>(remains same as for no error)</u>**
- Sum of all segments + Checksum value = 00100101 + 11011010 = 11111111
- Complemented value = 00000000
- Since the result is 0, receiver detects no error.

2 error (within same data block)

- At receiver side,
- The received data unit is divided into segments of 8 bits.
- All the segments along with the checksum value are added.
- 100110<u>**10**</u> + 11100010 + 00100100 + 10000100 = 10001001<u>**00**</u>
- Since the result consists of 10 bits, so extra 2 bits are wrapped around.
- 001001<u>**00**</u> + 10 = 00100<u>**110**</u> (8 bits)
- Sum of all segments + Checksum value = 00100<u>**110**</u> + 11011010
  = 100000000 (9-bits) = 00000000+1=00000001
- Complemented value = 11111110
- Since the result is not 0, receiver detects 2 errors.



Sender's End

```
Frame 1:         11001100
Frame 2:       + 10101010
               ----------
Partial Sum:   1 01110110
                      + 1
               ----------
                 01110111
Frame 3:       + 11110000
               ----------
Partial Sum:   1 01100111
                      + 1
               ----------
                 01101000
Frame 4:       + 11000011
               ----------
Partial Sum:   1 00101011
                      + 1
               ----------
Sum:             00101100
               ----------
Checksum:        11010011
```

Receiver's End

```
Frame 1:         11001100
Frame 2:       + 10101010
               ----------
Partial Sum:   1 01110110
                      + 1
               ----------
                 01110111
Frame 3:       + 11110000
               ----------
Partial Sum:   1 01100111
                      + 1
               ----------
                 01101000
Frame 4:       + 11000011
               ----------
Partial Sum:   1 00101011
                      + 1
               ----------
Sum:             00101100
Checksum:        11010011
               ----------
Sum:             11111111
               ----------
Complement:      00000000
```

Hence accept frames.

Block parity:

- Parity check bits are calculated for each row, which is equivalent to a simple parity check bit.
- Parity check bits are also calculated for all columns, then both are sent along with the data.
- At the receiving end these are compared with the parity bits calculated on the received data.

Original Data

10011001	11100010	00100100	10000100

Row parities

	1 0 0 1 1 0 0 1	0
	1 1 1 0 0 0 1 0	0
	0 0 1 0 0 1 0 0	0
	1 0 0 0 0 1 0 0	0
Column parities ➡	1 1 0 1 1 0 1 1	0

100110010	111000100	001001000	100001000	110110110

Data to be sent

For example, six 8-bit words in succession can be formed into a 6 × 8 block for transmission. Parity bits are added so that odd parity is maintained both row-wise and column-wise and the block is transmitted as a 7 × 9 block as shown in Table A. At the receiving end, parity is checked both row-wise and column-wise and suppose errors are detected as shown in Table B. These single-bit errors detected can be corrected by complementing the error bit. In Table B, parity errors in the 3rd row and 5th column mean that the 5th bit in the 3rd row is in error. It can be corrected by complementing it.

Two errors as shown in Table C can only be detected but not corrected. In Table C, parity errors are observed in both columns 2 and 4. It indicates that in one row there are two errors.

Table A

	0	1	0	1	1	0	1	1	0
	1	0	0	1	0	1	0	1	1
	0	1	1	0	1	1	1	0	0
	1	1	0	1	0	0	1	1	0
	1	0	0	0	1	1	0	1	1
	0	1	1	1	0	1	1	1	1
Parity row →	0	1	1	1	0	1	1	0	0
									↑ Parity column

Table B

```
0 1 0 1 1 0 1 1 | 0
1 0 0 1 0 1 0 1 | 1
0 1 1 0 0 1 1 0 | 0   ← Parity error in 3rd row
1 1 0 1 0 0 1 1 | 0
1 0 0 0 1 1 0 1 | 1
0 1 1 1 0 1 1 1 | 1
- - - - - - - - -
0 1 1 1 0 1 1 0 0
        ↑
                        Parity error in 5th column
```

Table C

```
0 1 0 1 1 0 1 1 | 0
1 0 0 1 0 1 0 1 | 1
0 1 1 0 1 1 1 0 | 0
1 0 0 0 0 0 1 1 | 0
1 0 0 0 1 1 0 1 | 1
0 1 1 1 0 1 1 1 | 1
- - - - - - - - -
0 1 1 1 0 1 1 0 0
  ↑   ↑
  Parity errors in 2nd and 4th columns
```