Digital Logic & Design (Theory)

Lecture No.3

LECTURE OUTLINE

2–10 Binary Coded Decimal (BCD)

Binary coded decimal (BCD) is a way to express each of the decimal digits with a binary code. There are only ten code groups in the BCD system, so it is very easy to convert between decimal and BCD. Because we like to read and write in decimal, the BCD code provides an excellent interface to binary systems. Examples of such interfaces are keypad inputs and digital readouts.

The 8421 BCD Code

The 8421 code is a type of **BCD** (binary coded decimal) code. Binary coded decimal means that each decimal digit, 0 through 9, is represented by a binary code of four bits. The desig-nation 8421 indicates the binary weights of the four bits (2^3, 2^2, 2^1, 2^0). The ease of conver-sion between 8421 code numbers and the familiar decimal numbers is the main advantage of this code. All you have to remember are the ten binary combinations that represent the ten decimal digits as shown in Table 2–5. The 8421 code is the predominant BCD code, and when we refer to BCD, we always mean the 8421 code unless otherwise stated.

TABLE 2–5
Decimal/BCD conversion.

Decimal Digit	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Invalid Codes

You should realize that, with four bits, sixteen numbers (0000 through 1111) can be repre-sented but that, in the 8421 code, only ten of these are used. The six code combinations that are not used—1010, 1011, 1100, 1101, 1110, and 1111—are invalid in the 8421 BCD code.

To express any decimal number in BCD, simply replace each decimal digit with the appropriate 4-bit code, as shown by Example 2–33.

EXAMPLE 2–33

Convert each of the following decimal numbers to BCD:

(a) 35 **(b)** 98 **(c)** 170 **(d)** 2469

Solution

(a) 3 5
↓ ↓
00110101

(b) 9 8
↓ ↓
10011000

(c) 1 7 0
↓ ↓ ↓
000101110000

(d) 2 4 6 9
↓ ↓ ↓ ↓
0010010001101001

It is equally easy to determine a decimal number from a BCD number. Start at the right-most bit and break the code into groups of four bits. Then write the decimal digit represented by each 4-bit group.

EXAMPLE 2–34

Convert each of the following BCD codes to decimal:

(a) 10000110 **(b)** 001101010001 **(c)** 1001010001110000

Solution

(a) 10000110
↓ ↓
8 6

(b) 001101010001
↓ ↓ ↓
3 5 1

(c) 1001010001110000
↓ ↓ ↓ ↓
9 4 7 0

Applications

Digital clocks, digital thermometers, digital meters, and other devices with seven-segment displays typically use BCD code to simplify the displaying of decimal numbers. BCD is not as efficient as straight binary for calculations, but it is particularly useful if only limited processing is required, such as in a digital thermometer.

BCD Addition

BCD is a numerical code and can be used in arithmetic operations. Addition is the most important operation because the other three operations (subtraction, multiplication, and division) can be accomplished by the use of addition. Here is how to add two BCD numbers:

Step 1: Add the two BCD numbers, using the rules for binary addition in Section 2–4.

Step 2: If a 4-bit sum is equal to or less than 9, it is a valid BCD number.

Step 3: If a 4-bit sum is greater than 9, or if a carry out of the 4-bit group is generated, it is an invalid result. Add 6 (0110) to the 4-bit sum in order to skip the six invalid states and return the code to 8421. If a carry results when 6 is added, simply add the carry to the next 4-bit group.

Example 2–35 illustrates BCD additions in which the sum in each 4-bit column is equal to or less than 9, and the 4-bit sums are therefore valid BCD numbers. Example 2–36 illustrates the procedure in the case of invalid sums (greater than 9 or a carry).

An alternative method to add BCD numbers is to convert them to decimal, perform the addition, and then convert the answer back to BCD.

EXAMPLE 2–35

Add the following BCD numbers:

(a) 0011 + 0100 **(b)** 00100011 + 00010101

(c) 10000110 + 00010011 **(d)** 010001010000 + 010000010111

Solution

The decimal number additions are shown for comparison.

(a)

0011	3
+ 0100	+ 4
0111	7

(b)

0010	0011	23
+ 0001	0101	+ 15
0011	**1000**	38

(c)

1000	0110	86
+ 0001	0011	+ 13
1001	**1001**	**99**

(d)

0100	0101	0000	450
+ 0100	0001	0111	+ 417
1000	**0110**	**0111**	**867**

Note that in each case the sum in any 4-bit column does not exceed 9, and the results are valid BCD numbers.

EXAMPLE 2–36

Add the following BCD numbers:

(a) 1001 + 0100 **(b)** 1001 + 1001

(c) 00010110 + 00010101 **(d)** 01100111 + 01010011

Solution

The decimal number additions are shown for comparison.

(a)

	1001		9
	+ 0100		+4
	1101	Invalid BCD number (>9)	13
	+ 0110	Add 6	
0001	**0011**	Valid BCD number	
↓	↓		
1	3		

(b)

	1001		9
	+ 1001		+ 9
1	0010	Invalid because of carry	18
	+ 0110	Add 6	
0001	**1000**	Valid BCD number	
↓	↓		
1	8		

(c)

0001	0110		16
+ 0001	0101		+ 15
0010	1011	Right group is invalid (>9), left group is valid.	31
	+ 0110	Add 6 to invalid code. Add carry, 0001, to next group.	
0011	**0001**	Valid BCD number	
↓	↓		
3	1		

(d)

	0110	0111		67
	+ 0101	0011		+ 53
	1011	1010	Both groups are invalid (>9)	120
	+ 0110	+ 0110	Add 6 to both groups	
0001	**0010**	**0000**	Valid BCD number	
↓	↓	↓		
1	2	0		

2–11 Digital Codes

Many specialized codes are used in digital systems. You have just learned about the BCD code; now let's look at a few others. Some codes are strictly numeric, like BCD, and others are alphanumeric; that is, they are used to represent numbers, letters, symbols, and instructions. The codes introduced in this section are the Gray code, the ASCII code, and the Unicode.

The Gray Code

The **Gray code** is unweighted and is not an arithmetic code; that is, there are no specific weights assigned to the bit positions. The important feature of the Gray code is that *it exhibits only a single bit change from one code word to the next in sequence.* This property is important in many applications, such as shaft position encoders, where error susceptibility increases with the number of bit changes between adjacent numbers in a sequence.

Table 2–6 is a listing of the 4-bit Gray code for decimal numbers 0 through 15. Binary numbers are shown in the table for reference. Like binary numbers, *the Gray code can have any number of bits.* Notice the single-bit change between successive Gray code words. For instance, in going from decimal 3 to decimal 4, the Gray code changes from 0010 to 0110, while the binary code changes from 0011 to 0100, a change of three bits. The only bit change in the Gray code is in the third bit from the right: the other bits remain the same.

TABLE 2–6

Four-bit Gray code.

Decimal	Binary	Gray Code	Decimal	Binary	Gray Code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Binary-to-Gray Code Conversion

Conversion between binary code and Gray code is sometimes useful. The following rules explain how to convert from a binary number to a Gray code word:

1. The most significant bit (left-most) in the Gray code is the same as the corresponding MSB in the binary number.
2. Going from left to right, add each adjacent pair of binary code bits to get the next Gray code bit. Discard carries.

For example, the conversion of the binary number 10110 to Gray code is as follows:

```
1 − + → 0 − + → 1 − + → 1 − + → 0     Binary
↓       ↓       ↓       ↓       ↓
1       1       1       0       1     Gray
```

The Gray code is 11101.

Gray-to-Binary Code Conversion

To convert from Gray code to binary, use a similar method; however, there are some differences. The following rules apply:

1. The most significant bit (left-most) in the binary code is the same as the corresponding bit in the Gray code.
2. Add each binary code bit generated to the Gray code bit in the next adjacent position. Discard carries.

For example, the conversion of the Gray code word 11011 to binary is as follows:

1	1	0	1	1	Gray
↓	+ ↓	+ ↓	+ ↓	+ ↓	
1	0	0	1	0	Binary

The binary number is 10010.

EXAMPLE 2–37

(a) Convert the binary number 11000110 to Gray code.

(b) Convert the Gray code 10101111 to binary.

Solution

(a) Binary to Gray code:

1 − + → 1 − + → 0 − + → 0 − + → 0 − + → 1 − + → 1 − + → 0

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

1 0 1 0 0 1 0 1

(b) Gray code to binary:

1 0 1 0 1 1 1 1

↓ +↓ +↓ +↓ +↓ +↓ +↓ +↓

1 1 0 0 1 0 1 0

An Application

The concept of a 3-bit shaft position encoder is shown in Figure 2–7. Basically, there are three concentric rings that are segmented into eight sectors. The more sectors there are, the more accurately the position can be represented, but we are using only eight to illustrate. Each sector of each ring is either reflective or nonreflective. As the rings rotate with the shaft, they come under an IR emitter that produces three separate IR beams. A 1 is indicated where there is a reflected beam, and a 0 is indicated where there is no reflected beam. The IR detector senses the presence or absence of reflected

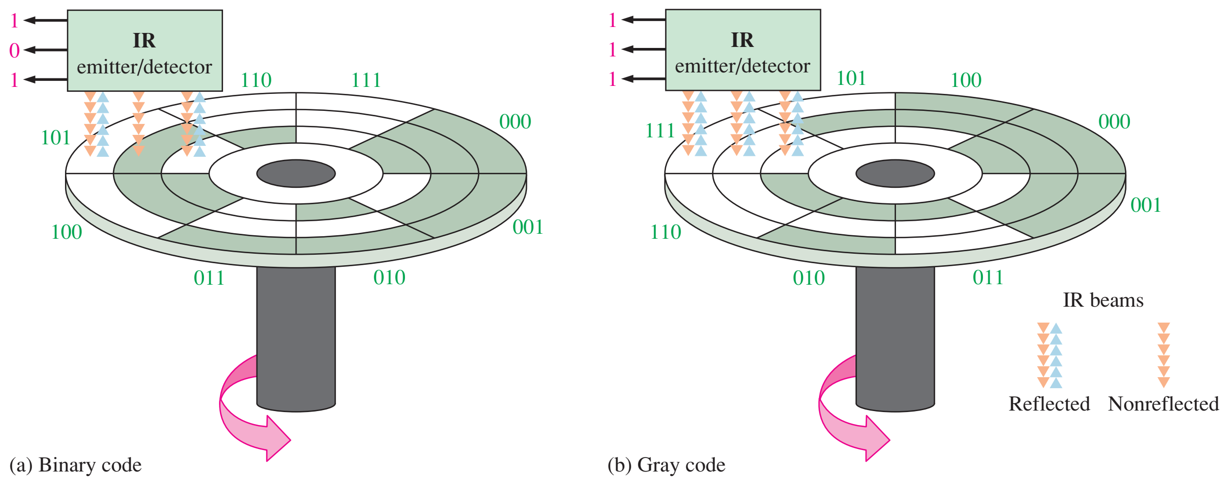

FIGURE 2–7 A simplified illustration of how the Gray code solves the error problem in shaft position encoders. Three bits are shown to illustrate the concept, although most shaft encoders use more than 10 bits to achieve a higher resolution.

beams and produces a corresponding 3-bit code. The IR emitter/detector is in a fixed position. As the shaft rotates counterclockwise through 360°, the eight sectors move under the three beams. Each beam is either reflected or absorbed by the sector surface to represent a binary or Gray code number that indicates the shaft position.

In Figure 2–7(a), the sectors are arranged in a straight binary pattern, so that the detector output goes from 000 to 001 to 010 to 011 and so on. When a beam is aligned over a reflective sector, the output is 1; when a beam is aligned over a nonreflective sector, the output is 0. If one beam is slightly ahead of the others during the transition from one sector to the next, an erroneous output can occur. Consider what happens when the beams are on the 111 sector and about to enter the 000 sector. If the MSB beam is slightly ahead, the position would be incorrectly indicated by a transitional 011 instead of a 111 or a 000. In this type of application, it is virtually impossible to maintain precise mechanical alignment of the IR emitter/detector beams; therefore, some error will usually occur at many of the transitions between sectors.

The Gray code is used to eliminate the error problem which is inherent in the binary code. As shown in Figure 2–7(b), the Gray code assures that only one bit will change between adjacent sectors. This means that even though the beams may not be in precise alignment, there will never be a transitional error. For example, let's again consider what happens when the beams are on the 111 sector and about to move into the next sector, 101. The only two possible outputs during the transition are 111 and 101, no matter how the beams are aligned. A similar situation occurs at the transitions between each of the other sectors.

Alphanumeric Codes

In order to communicate, you need not only numbers, but also letters and other symbols. In the strictest sense, **alphanumeric** codes are codes that represent numbers and alphabetic characters (letters). Most such codes, however, also represent other characters such as sym-bols and various instructions necessary for conveying information.

At a minimum, an alphanumeric code must represent 10 decimal digits and 26 letters of the alphabet, for a total of 36 items. This number requires six bits in each code combination because five bits are insufficient ($2^5 = 32$). There are 64 total combinations of six bits, so there are 28 unused code combinations. Obviously, in many applications, symbols other than just numbers and letters are necessary to communicate completely. You need spaces, periods, colons, semicolons, question marks, etc. You also need instructions to tell the receiving system what to do with the information. With codes that are six bits long, you can handle decimal numbers, the alphabet, and 28 other symbols. This should give you an idea of the requirements for a basic alphanumeric code. The ASCII is a common alphanumeric code and is covered next.

ASCII

ASCII is the abbreviation for American Standard Code for Information Interchange. Pronounced "askee," ASCII is a universally accepted alphanumeric code used in most computers and other electronic equipment. Most computer keyboards are standardized with the ASCII. When you enter a letter, a number, or control command, the corresponding ASCII code goes into the computer.

ASCII has 128 characters and symbols represented by a 7-bit binary code. Actually, ASCII can be considered an 8-bit code with the MSB always 0. This 8-bit code is 00 through 7F in hexadecimal. The first thirty-two ASCII characters are nongraphic commands that are never printed or displayed and are used only for control purposes. Examples of the control characters are "null," "line feed," "start of text," and "escape." The other characters are graphic symbols that can be printed or displayed and include the letters of the alphabet (lowercase and uppercase), the ten decimal digits, punctuation signs, and other commonly used symbols.

Table 2–7 is a listing of the ASCII code showing the decimal, hexadecimal, and binary representations for each character and symbol. The left section of the table lists the names of the 32 control characters (00 through 1F hexadecimal). The graphic symbols are listed in the rest of the table (20 through 7F hexadecimal).

EXAMPLE 2–38

Use Table 2–7 to determine the binary ASCII codes that are entered from the computer's keyboard when the following C language program statement is typed in. Also express each code in hexadecimal.

if (x > 5)

Solution

The ASCII code for each symbol is found in Table 2–7.

Symbol	Binary	Hexadecimal
i	1101001	69_{16}
f	1100110	66_{16}
Space	0100000	20_{16}
(	0101000	28_{16}
x	1111000	78_{16}
>	0111110	$3E_{16}$
5	0110101	35_{16}
)	0101001	29_{16}

The ASCII Control Characters

The first thirty-two codes in the ASCII table (Table 2–7) represent the control characters. These are used to allow devices such as a computer and printer to communicate with each other when passing information and data. The control key function allows a control character to be entered directly from an ASCII keyboard by pressing the control key (CTRL) and the corresponding symbol.

Extended ASCII Characters

In addition to the 128 standard ASCII characters, there are an additional 128 characters that were adopted by IBM for use in their PCs (personal computers). Because of the popularity of the PC, these particular extended ASCII characters are also used in applications other than PCs and have become essentially an unofficial standard.

The extended ASCII characters are represented by an 8-bit code series from hexadecimal 80 to hexadecimal FF and can be grouped into the following general categories: foreign (non-English) alphabetic characters, foreign currency symbols, Greek letters, mathematical symbols, drawing characters, bar graphing characters, and shading characters.

Unicode

Unicode provides the ability to encode all of the characters used for the written languages of the world by assigning each character a unique numeric value and name utilizing the universal character set (UCS). It is applicable in computer applications dealing with multilingual text, mathematical symbols, or other technical characters.

Unicode has a wide array of characters, and their various encoding forms are used in many environments. While ASCII basically uses 7-bit codes, Unicode uses relatively abstract "code points"—non-negative integer numbers—that map sequences of one or more bytes, using different encoding forms and schemes. To permit compatibility, Unicode assigns the first 128 code points to the same characters as ASCII. One can, therefore, think of ASCII as a 7-bit encoding scheme for a very small subset of Unicode and of the UCS.

Unicode consists of about 100,000 characters, a set of code charts for visual reference, an encoding methodology and set of standard character encodings, and an enumeration of character properties such as uppercase and lowercase. It also consists of a number of related items, such as character properties, rules for text normalization, decomposition, collation, rendering, and bidirectional display order (for the correct display of text containing both right-to-left scripts, such as Arabic or Hebrew, and left-to-right scripts).

TABLE 2–7

American Standard Code for Information Interchange (ASCII).

Control Characters				Graphic Symbols											
Name	**Dec**	**Binary**	**Hex**	**Symbol**	**Dec**	**Binary**	**Hex**	**Symbol**	**Dec**	**Binary**	**Hex**	**Symbol**	**Dec**	**Binary**	**Hex**
NUL	0	0000000	00	space	32	0100000	20	@	64	1000000	40	′	96	1100000	60
SOH	1	0000001	01	!	33	0100001	21	A	65	1000001	41	a	97	1100001	61
STX	2	0000010	02	”	34	0100010	22	B	66	1000010	42	b	98	1100010	62
ETX	3	0000011	03	#	35	0100011	23	C	67	1000011	43	c	99	1100011	63
EOT	4	0000100	04	$	36	0100100	24	D	68	1000100	44	d	100	1100100	64
ENQ	5	0000101	05	%	37	0100101	25	E	69	1000101	45	e	101	1100101	65
ACK	6	0000110	06	&	38	0100110	26	F	70	1000110	46	f	102	1100110	66
BEL	7	0000111	07	’	39	0100111	27	G	71	1000111	47	g	103	1100111	67
BS	8	0001000	08	(	40	0101000	28	H	72	1001000	48	h	104	1101000	68
HT	9	0001001	09	)	41	0101001	29	I	73	1001001	49	i	105	1101001	69
LF	10	0001010	0A	*	42	0101010	2A	J	74	1001010	4A	j	106	1101010	6A
VT	11	0001011	0B	+	43	0101011	2B	K	75	1001011	4B	k	107	1101011	6B
FF	12	0001100	0C	,	44	0101100	2C	L	76	1001100	4C	l	108	1101100	6C
CR	13	0001101	0D	−	45	0101101	2D	M	77	1001101	4D	m	109	1101101	6D
SO	14	0001110	0E	.	46	0101110	2E	N	78	1001110	4E	n	110	1101110	6E
SI	15	0001111	0F	/	47	0101111	2F	O	79	1001111	4F	o	111	1101111	6F
DLE	16	0010000	10	0	48	0110000	30	P	80	1010000	50	p	112	1110000	70
DC1	17	0010001	11	1	49	0110001	31	Q	81	1010001	51	q	113	1110001	71
DC2	18	0010010	12	2	50	0110010	32	R	82	1010010	52	r	114	1110010	72
DC3	19	0010011	13	3	51	0110011	33	S	83	1010011	53	s	115	1110011	73
DC4	20	0010100	14	4	52	0110100	34	T	84	1010100	54	t	116	1110100	74
NAK	21	0010101	15	5	53	0110101	35	U	85	1010101	55	u	117	1110101	75
SYN	22	0010110	16	6	54	0110110	36	V	86	1010110	56	v	118	1110110	76
ETB	23	0010111	17	7	55	0110111	37	W	87	1010111	57	w	119	1110111	77
CAN	24	0011000	18	8	56	0111000	38	X	88	1011000	58	x	120	1111000	78
EM	25	0011001	19	9	57	0111001	39	Y	89	1011001	59	y	121	1111001	79
SUB	26	0011010	1A	:	58	0111010	3A	Z	90	1011010	5A	z	122	1111010	7A
ESC	27	0011011	1B	;	59	0111011	3B	[	91	1011011	5B	{	123	1111011	7B
FS	28	0011100	1C	<	60	0111100	3C	\	92	1011100	5C	|	124	1111100	7C
GS	29	0011101	1D	=	61	0111101	3D	]	93	1011101	5D	}	125	1111101	7D
RS	30	0011110	1E	>	62	0111110	3E	^	94	1011110	5E	~	126	1111110	7E
US	31	0011111	1F	?	63	0111111	3F	_	95	1011111	5F	Del	127	1111111	7F

2–12 Error Codes

In this section, three methods for adding bits to codes to detect a single-bit error are discussed. The parity method of error detection is introduced, and the cyclic redundancy check is discussed. Also, the Hamming code for error detection and correction is presented.

Parity Method for Error Detection

Many systems use a parity bit as a means for bit **error detection**. Any group of bits contain either an even or an odd number of 1s. A parity bit is attached to a group of bits to make the total number of 1s in a group always even or always odd. An even parity bit makes the total number of 1s even, and an odd parity bit makes the total odd.

A given system operates with even or odd **parity**, but not both. For instance, if a system operates with even parity, a check is made on each group of bits received to make sure the total number of 1s in that group is even. If there is an odd number of 1s, an error has occurred.

As an illustration of how parity bits are attached to a code, Table 2–8 lists the parity bits for each BCD number for both even and odd parity. The parity bit for each BCD number is in the *P* column.

TABLE 2–8

The BCD code with parity bits.

Even Parity		Odd Parity	
P	**BCD**	***P***	**BCD**
0	0000	1	0000
1	0001	0	0001
1	0010	0	0010
0	0011	1	0011
1	0100	0	0100
0	0101	1	0101
0	0110	1	0110
1	0111	0	0111
1	1000	0	1000
0	1001	1	1001

The parity bit can be attached to the code at either the beginning or the end, depending on system design. Notice that the total number of 1s, including the parity bit, is always even for even parity and always odd for odd parity.

Detecting an Error

A parity bit provides for the detection of a single bit error (or any odd number of errors, which is very unlikely) but cannot check for two errors in one group. For instance, let's assume that we wish to transmit the BCD code 0101. (Parity can be used with any number of bits; we are using four for illustration.) The total code transmitted, including the even parity bit, is

Even parity bit

00101

BCD code

Now let's assume that an error occurs in the third bit from the left (the 1 becomes a 0).

Even parity bit

00001

Bit error

When this code is received, the parity check circuitry determines that there is only a single 1 (odd number), when there should be an even number of 1s. Because an even number of 1s does not appear in the code when it is received, an error is indicated.

An odd parity bit also provides in a similar manner for the detection of a single error in a given group of bits.

EXAMPLE 2–39

Assign the proper even parity bit to the following code groups:

(a) 1010 **(b)** 111000 **(c)** 101101

(d) 1000111001001 **(e)** 101101011111

Solution

Make the parity bit either 1 or 0 as necessary to make the total number of 1s even. The parity bit will be the left-most bit (color).

(a) **0**1010 **(b)** **1**111000 **(c)** **0**101101

(d) **0**1000111001001 **(e)** **1**101101011111

EXAMPLE 2–40

An odd parity system receives the following code groups: 10110, 11010, 110011, 110101110100, and 1100010101010. Determine which groups, if any, are in error.

Solution

Since odd parity is required, any group with an even number of 1s is incorrect. The following groups are in error: **110011** and **1100010101010**.

Cyclic Redundancy Check

The **cyclic redundancy check (CRC)** is a widely used code used for detecting one- and two-bit transmission errors when digital data are transferred on a communication link. The communication link can be between two computers that are connected to a network or between a digital storage device (such as a CD, DVD, or a hard drive) and a PC. If it is properly designed, the CRC can also detect multiple errors for a number of bits in sequence (burst errors). In CRC, a certain number of check bits, sometimes called a *checksum*, are appended to the data bits (added to end) that are being transmitted. The transmitted data are tested by the receiver for errors using the CRC. Not every possible error can be identified, but the CRC is much more efficient than just a simple parity check.

CRC is often described mathematically as the division of two polynomials to generate a remainder. A polynomial is a mathematical expression that is a sum of terms with positive exponents. When the coefficients are limited to 1s and 0s, it is called a *univariate polynomial*. An example of a univariate polynomial is $1x^3 + 0x^2 + 1x^1 + 1x^0$ or simply $x^3 + x^1 + x^0$, which can be fully described by the 4-bit binary number 1011. Most cyclic redundancy checks use a 16-bit or larger polynomial, but for simplicity the process is illustrated here with four bits.

Modulo-2 Operations

Simply put, CRC is based on the division of two binary numbers; and, as you know, division is just a series of subtractions and shifts. To do subtraction, a method called *modulo-2* addition can be used. Modulo-2 addition (or subtraction) is the same as binary addition with the carries discarded, as shown in the truth table in Table 2–9. **Truth tables** are widely used to describe the operation of logic circuits, as you will learn in Chapter 3. With two bits, there is a total of four possible combinations, as shown in the table. This particular table describes the modulo-2 operation also known as *exclusive-OR* and can be implemented with a logic gate. A simple rule for modulo-2 is that the output is 1 if the inputs are different; otherwise, it is 0.

TABLE 2–9

Modulo-2 operation.

Input Bits	Output Bit
0 0	0
0 1	1
1 0	1
1 1	0

Figure 2–8 illustrates the CRC process.

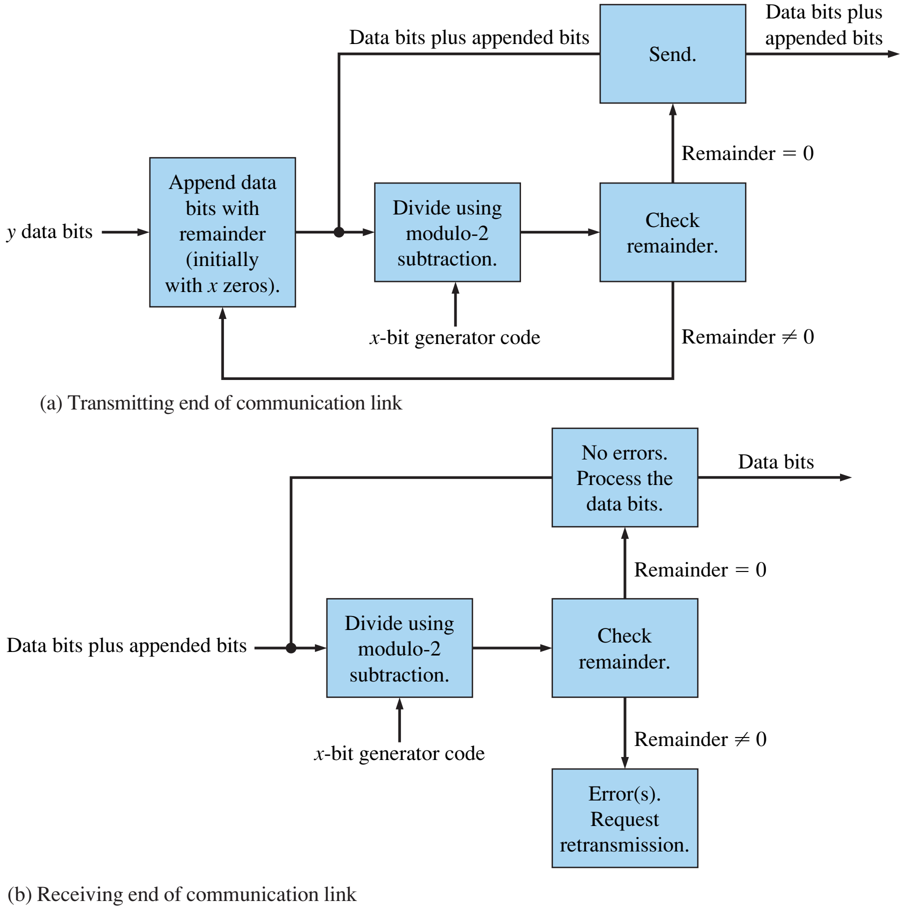

FIGURE 2–8 The CRC process.

CRC Process

The process is as follows:

1. Select a fixed generator code; it can have fewer bits than the data bits to be checked. This code is understood in advance by both the sending and receiving devices and must be the same for both.
2. Append a number of 0s equal to the number of bits in the generator code to the data bits.
3. Divide the data bits including the appended bits by the generator code bits using modulo-2.

4. If the remainder is 0, the data and appended bits are sent as is.
5. If the remainder is not 0, the appended bits are made equal to the remainder bits in order to get a 0 remainder before data are sent.
6. At the receiving end, the receiver divides the incoming appended data bit code by the same generator code as used by the sender.
7. If the remainder is 0, there is no error detected (it is possible in rare cases for multiple errors to cancel). If the remainder is not 0, an error has been detected in the transmission and a retransmission is requested by the receiver.

EXAMPLE 2–41

Determine the transmitted CRC for the following byte of data (D) and generator code (G). Verify that the remainder is 0.

D: 11010011

G: 1010

Solution

Since the generator code has four data bits, add four 0s (blue) to the data byte. The appended data (D′) is

$$D' = 11010011\mathbf{0000}$$

Divide the appended data by the generator code (red) using the modulo-2 operation until all bits have been used.

$$\frac{D'}{G} = \frac{11010011\mathbf{0000}}{1010}$$

```
110100110000
1010
 1110
 1010
  1000
  1010
    1011
    1010
       1000
       1010
        100
```

Remainder = 0100. Since the remainder is not 0, append the data with the four remainder bits (blue). Then divide by the generator code (red). The transmitted CRC is **110100110100**.

```
110100110100
1010
 1110
 1010
  1000
  1010
    1011
    1010
       1010
       1010
         00
```

Remainder = 0

EXAMPLE 2–42

During transmission, an error occurs in the second bit from the left in the appended data byte generated in Example 2–41. The received data is

$$D' = 10010011\mathbf{0100}$$

Apply the CRC process to the received data to detect the error using the same generator code (1010).

Solution

```
10010011 0100
1010
   1100
   1010
    1101
    1010
     1111
     1010
      1010
      1010
         0100
```

Remainder = 0100. Since it is not zero, an error is indicated.

Hamming Code

The **Hamming code** is used to detect and correct a single-bit error in a transmitted code. To accomplish this, four redundancy bits are introduced in a 7-bit group of data bits. These redundancy bits are interspersed at bit positions 2^n ($n = 0, 1, 2, 3$) within the original data bits. At the end of the transmission, the redundancy bits have to be removed from the data bits. A recent version of the Hamming code places all the redundancy bits at the end of the data bits, making their removal easier than that of the interspersed bits. *A coverage of the classic Hamming code is available on the website.*