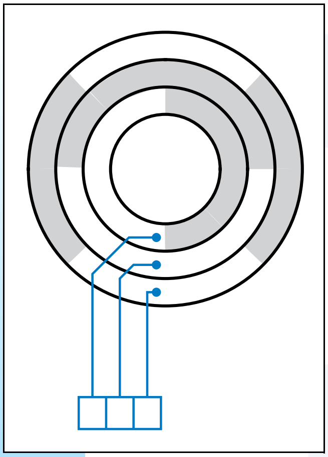


chapter 2

Number Systems and Codes

Digital systems are built from circuits that process binary digits—0s and 1s—yet very few real-life problems are based on binary numbers or any numbers at all. Therefore, a digital system designer must establish some correspondence between the binary digits processed by digital circuits and real-life numbers, events, and conditions. The purpose of this chapter is to show you how familiar numeric quantities can be represented and manipulated in a digital system, and how nonnumeric data, events, and conditions also can be represented.

The first nine sections describe binary number systems and show how addition, subtraction, multiplication, and division are performed in these systems. Sections 2.10–2.13 show how other things, such as decimal numbers, text characters, mechanical positions, and arbitrary conditions, can be encoded using strings of binary digits.

Section 2.14 introduces "*n*-cubes," which provide a way to visualize the relationship between different bit strings. The *n*-cubes are especially useful in the study of error-detecting codes in Section 2.15. We conclude the chapter with an introduction to codes for transmitting and storing data one bit at a time.

2.1 Positional Number Systems

positional number system
weight

The traditional number system that we learned in school and use every day in business is called a *positional number system*. In such a system, a number is represented by a string of digits where each digit position has an associated *weight*. The value of a number is a weighted sum of the digits, for example:

$$1734 = 1 \cdot 1000 + 7 \cdot 100 + 3 \cdot 10 + 4 \cdot 1$$

Each weight is a power of 10 corresponding to the digit's position. A decimal point allows negative as well as positive powers of 10 to be used:

$$5185.68 = 5 \cdot 1000 + 1 \cdot 100 + 8 \cdot 10 + 5 \cdot 1 + 6 \cdot 0.1 + 8 \cdot 0.01$$

In general, a number D of the form $d_1 d_0 . d_{-1} d_{-2}$ has the value

$$D = d_1 \cdot 10^1 + d_0 \cdot 10^0 + d_{-1} \cdot 10^{-1} + d_{-2} \cdot 10^{-2}$$

base
radix

Here, 10 is called the *base* or *radix* of the number system. In a general positional number system, the radix may be any integer $r \geq 2$, and a digit in position i has weight r^i. The general form of a number in such a system is

$$d_{p-1} d_{p-2} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-n}$$

radix point

where there are p digits to the left of the point and n digits to the right of the point, called the *radix point*. If the radix point is missing, it is assumed to be to the right of the rightmost digit. The value of the number is the sum of each digit multiplied by the corresponding power of the radix:

$$D = \sum_{i=-n}^{p-1} d_i \cdot r^i$$

high-order digit
most significant digit
low-order digit
least significant digit

Except for possible leading and trailing zeroes, the representation of a number in a positional number system is unique. (Obviously, 0185.6300 equals 185.63, and so on.) The leftmost digit in such a number is called the *high-order* or *most significant digit*; the rightmost is the *low-order* or *least significant digit*.

binary digit
bit
binary radix

As we'll learn in Chapter 3, digital circuits have signals that are normally in one of only two conditions—low or high, charged or discharged, off or on. The signals in these circuits are interpreted to represent *binary digits* (or *bits*) that have one of two values, 0 and 1. Thus, the *binary radix* is normally used to represent numbers in a digital system. The general form of a binary number is

$$b_{p-1} b_{p-2} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n}$$

and its value is

$$B = \sum_{i=-n}^{p-1} b_i \cdot 2^i$$

In a binary number, the radix point is called the *binary point*. When dealing with binary and other nondecimal numbers, we use a subscript to indicate the radix of each number, unless the radix is clear from the context. Examples of binary numbers and their decimal equivalents are given below.

binary point

$$10011_2 = 1\cdot16 + 0\cdot8 + 0\cdot4 + 1\cdot2 + 1\cdot1 = 19_{10}$$
$$100010_2 = 1\cdot32 + 0\cdot16 + 0\cdot8 + 0\cdot4 + 1\cdot2 + 0\cdot1 = 34_{10}$$
$$101.001_2 = 1\cdot4 + 0\cdot2 + 1\cdot1 + 0\cdot0.5 + 0\cdot0.25 + 1\cdot0.125 = 5.125_{10}$$

The leftmost bit of a binary number is called the *high-order* or *most significant bit (MSB)*; the rightmost is the *low-order* or *least significant bit (LSB)*.

MSB
LSB

2.2 Octal and Hexadecimal Numbers

Radix 10 is important because we use it in everyday business, and radix 2 is important because binary numbers can be processed directly by digital circuits. Numbers in other radices are not often processed directly, but may be important for documentation or other purposes. In particular, the radices 8 and 16 provide convenient shorthand representations for multibit numbers in a digital system.

The *octal number system* uses radix 8, while the *hexadecimal number system* uses radix 16. Table 2-1 shows the binary integers from 0 to 1111 and their octal, decimal, and hexadecimal equivalents. The octal system needs 8 digits, so it uses digits 0–7 of the decimal system. The hexadecimal system needs 16 digits, so it supplements decimal digits 0–9 with the letters *A–F*.

octal number system
hexadecimal number system
hexadecimal digits A–F

The octal and hexadecimal number systems are useful for representing multibit numbers because their radices are powers of 2. Since a string of three bits can take on eight different combinations, it follows that each 3-bit string can be uniquely represented by one octal digit, according to the third and fourth columns of Table 2-1. Likewise, a 4-bit string can be represented by one hexadecimal digit according to the fifth and sixth columns of the table.

Thus, it is very easy to convert a binary number to octal. Starting at the binary point and working left, we simply separate the bits into groups of three and replace each group with the corresponding octal digit:

binary to octal conversion

$$100011001110_2 = 100\ 011\ 001\ 110_2 = 4316_8$$
$$11101101110101001_2 = 011\ 101\ 101\ 110\ 101\ 001_2 = 355651_8$$

The procedure for binary to hexadecimal conversion is similar, except we use groups of four bits:

binary to hexadecimal conversion

$$100011001110_2 = 1000\ 1100\ 1110_2 = 8CE_{16}$$
$$11101101110101001_2 = 00011101\ 1011\ 1010\ 1001_2 = 1DBA9_{16}$$

In these examples we have freely added zeroes on the left to make the total number of bits a multiple of 3 or 4 as required.

Table 2-1
Binary, decimal, octal, and hexadecimal numbers.

Binary	Decimal	Octal	3-Bit String	Hexadecimal	4-Bit String
0	0	0	000	0	0000
1	1	1	001	1	0001
10	2	2	010	2	0010
11	3	3	011	3	0011
100	4	4	100	4	0100
101	5	5	101	5	0101
110	6	6	110	6	0110
111	7	7	111	7	0111
1000	8	10	—	8	1000
1001	9	11	—	9	1001
1010	10	12	—	A	1010
1011	11	13	—	B	1011
1100	12	14	—	C	1100
1101	13	15	—	D	1101
1110	14	16	—	E	1110
1111	15	17	—	F	1111

If a binary number contains digits to the right of the binary point, we can convert them to octal or hexadecimal by starting at the binary point and working right. Both the left-hand and right-hand sides can be padded with zeroes to get multiples of three or four bits, as shown in the example below:

$$\begin{aligned} 10.1011001011_2 &= 010\,.\,101\ 100\ 101\ 100_2 = 2.5454_8 \\ &= 0010\,.\,1011\ 0010\ 1100_2 = 2.\mathrm{B2C}_{16} \end{aligned}$$

octal or hexadecimal to binary conversion

Converting in the reverse direction, from octal or hexadecimal to binary, is very easy. We simply replace each octal or hexadecimal digit with the corresponding 3- or 4-bit string, as shown below:

$$\begin{aligned} 1357_8 &= 001\ 011\ 101\ 111_2 \\ 2046.17_8 &= 010\ 000\ 100\ 110\,.\,001\ 111_2 \\ \mathrm{BEAD}_{16} &= 1011\ 1110\ 1010\ 1101_2 \\ \mathrm{9F.46C}_{16} &= 1001\ 1111\,.\,0100\ 0110\ 1100_2 \end{aligned}$$

byte

The octal number system was quite popular 25 years ago because of certain minicomputers that had their front-panel lights and switches arranged in groups of three. However, the octal number system is not used much today, because of the preponderance of machines that process 8-bit *bytes*. It is difficult to extract individual byte values in multibyte quantities in the octal representation; for

WHEN I'M 64

As you grow older, you'll find that the hexadecimal number system is useful for more than just computers. When I turned 40, I told friends that I had just turned 28_{16}. The "$_{16}$" was whispered under my breath, of course. At age 50, I'll be only 32_{16}.

People get all excited about decennial birthdays like 20, 30, 40, 50, …, but you should be able to convince your friends that the decimal system is of no fundamental significance. More significant life changes occur around birthdays 2, 4, 8, 16, 32, and 64, when you add a most significant bit to your age. Why do you think the Beatles sang "When I'm sixty-*four*"?

example, what are the octal values of the four 8-bit bytes in the 32-bit number with octal representation 12345670123_8?

In the hexadecimal system, two digits represent an 8-bit byte, and 2*n* digits represent an *n*-byte word; each pair of digits constitutes exactly one byte. For example, the 32-bit hexadecimal number $5678ABCD_{16}$ consists of four bytes with values 56_{16}, 78_{16}, AB_{16}, and CD_{16}. In this context, a 4-bit hexadecimal digit is sometimes called a *nibble*; a 32-bit (4-byte) number has eight nibbles. Hexadecimal numbers are often used to describe a computer's memory address space. For example, a computer with 16-bit addresses might be described as having read/write memory installed at addresses 0–$EFFF_{16}$, and read-only memory at addresses F000–$FFFF_{16}$. Many computer programming languages use the prefix "0x" to denote a hexadecimal number, for example, `0xBFC0000`.

nibble

`0x` *prefix*

2.3 General Positional Number System Conversions

In general, conversion between two radices cannot be done by simple substitutions; arithmetic operations are required. In this section, we show how to convert a number in any radix to radix 10 and vice versa, using radix-10 arithmetic.

In Section 2.1, we indicated that the value of a number in any radix is given by the formula

radix-r to decimal conversion

$$D = \sum_{i=-n}^{p-1} d_i \cdot r^i$$

where *r* is the radix of the number and there are *p* digits to the left of the radix point and *n* to the right. Thus, the value of the number can be found by converting each digit of the number to its radix-10 equivalent and expanding the formula using radix-10 arithmetic. Some examples are given below:

$$\begin{aligned}
1CE8_{16} &= 1 \cdot 16^3 + 12 \cdot 16^2 + 14 \cdot 16^1 + 8 \cdot 16^0 = 7400_{10} \\
F1A3_{16} &= 15 \cdot 16^3 + 1 \cdot 16^2 + 10 \cdot 16^1 + 3 \cdot 16^0 = 61859_{10} \\
436.5_8 &= 4 \cdot 8^2 + 3 \cdot 8^1 + 6 \cdot 8^0 + 5 \cdot 8^{-1} = 286.625_{10} \\
132.3_4 &= 1 \cdot 4^2 + 3 \cdot 4^1 + 2 \cdot 4^0 + 3 \cdot 4^{-1} = 30.75_{10}
\end{aligned}$$

A shortcut for converting whole numbers to radix 10 is obtained by rewriting the expansion formula as follows:

$$D = ((\cdots((d_{p-1})\cdot r + d_{p-2})\cdot r + \cdots) \cdot r + d_1)\cdot r + d_0$$

That is, we start with a sum of 0; beginning with the leftmost digit, we multiply the sum by r and add the next digit to the sum, repeating until all digits have been processed. For example, we can write

$$\mathrm{F1AC}_{16} = (((15)\cdot 16 + 1\cdot 16 + 10)\cdot 16 + 12$$

decimal to radix-r conversion

Although this formula is not too exciting in itself, it forms the basis for a very convenient method of converting a decimal number D to a radix r. Consider what happens if we divide the formula by r. Since the parenthesized part of the formula is evenly divisible by r, the quotient will be

$$Q = (\cdots((d_{p-1})\cdot r + d_{p-2})\cdot r + \cdots)\cdot r + d_1$$

and the remainder will be d_0. Thus, d_0 can be computed as the remainder of the long division of D by r. Furthermore, the quotient Q has the same form as the original formula. Therefore, successive divisions by r will yield successive digits of D from right to left, until all the digits of D have been derived. Examples are given below:

$$\begin{array}{l}
179 \div 2 = 89 \text{ remainder } 1 \quad \text{(LSB)} \\
\qquad\quad \div 2 = 44 \text{ remainder } 1 \\
\qquad\qquad \div 2 = 22 \text{ remainder } 0 \\
\qquad\qquad\quad \div 2 = 11 \text{ remainder } 0 \\
\qquad\qquad\qquad \div 2 = 5 \text{ remainder } 1 \\
\qquad\qquad\qquad\quad \div 2 = 2 \text{ remainder } 1 \\
\qquad\qquad\qquad\qquad \div 2 = 1 \text{ remainder } 0 \\
\qquad\qquad\qquad\qquad\quad \div 2 = 0 \text{ remainder } 1 \quad \text{(MSB)} \\
179_{10} = 10110011_2
\end{array}$$

$$\begin{array}{l}
467 \div 8 = 58 \text{ remainder } 3 \quad \text{(least significant digit)} \\
\qquad\quad \div 8 = 7 \text{ remainder } 2 \\
\qquad\qquad \div 8 = 0 \text{ remainder } 7 \quad \text{(most significant digit)} \\
467_{10} = 723_8
\end{array}$$

$$\begin{array}{l}
3417 \div 16 = 213 \text{ remainder } 9 \quad \text{(least significant digit)} \\
\qquad\quad \div 16 = 13 \text{ remainder } 5 \\
\qquad\qquad \div 16 = 0 \text{ remainder } 13 \quad \text{(most significant digit)} \\
3417_{10} = \mathrm{D59}_{16}
\end{array}$$

Table 2-2 summarizes methods for converting among the most common radices.

Table 2-2 Conversion methods for common radices.

Conversion	Method	Example
Binary to		
Octal	Substitution	$10111011001_2 = 10\ 111\ 011\ 001_2 = 2731_8$
Hexadecimal	Substitution	$10111011001_2 = 101\ 1101\ 1001_2 = 5D9_{16}$
Decimal	Summation	$10111011001_2 = 1 \cdot 1024 + 0 \cdot 512 + 1 \cdot 256 + 1 \cdot 128 + 1 \cdot 64$ $+ 0 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 1497_{10}$
Octal to		
Binary	Substitution	$1234_8 = 001\ 010\ 011\ 100_2$
Hexadecimal	Substitution	$1234_8 = 001\ 010\ 011\ 100_2 = 0010\ 1001\ 1100_2 = 29C_{16}$
Decimal	Summation	$1234_8 = 1 \cdot 512 + 2 \cdot 64 + 3 \cdot 8 + 4 \cdot 1 = 668_{10}$
Hexadecimal to		
Binary	Substitution	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2$
Octal	Substitution	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2 = 1\ 100\ 000\ 011\ 011\ 110_2 = 140336_8$
Decimal	Summation	$C0DE_{16} = 12 \cdot 4096 + 0 \cdot 256 + 13 \cdot 16 + 14 \cdot 1 = 49374_{10}$
Decimal to		
Binary	Division	$108_{10} \div 2 = 54$ remainder 0 (LSB) $\div 2 = 27$ remainder 0 $\div 2 = 13$ remainder 1 $\div 2 = 6$ remainder 1 $\div 2 = 3$ remainder 0 $\div 2 = 1$ remainder 1 $\div 2 = 0$ remainder 1 (MSB) $108_{10} = 1101100_2$
Octal	Division	$108_{10} \div 8 = 13$ remainder 4 (least significant digit) $\div 8 = 1$ remainder 5 $\div 8 = 0$ remainder 1 (most significant digit) $108_{10} = 154_8$
Hexadecimal	Division	$108_{10} \div 16 = 6$ remainder 12 (least significant digit) $\div 16 = 0$ remainder 6 (most significant digit) $108_{10} = 6C_{16}$

Table 2-3
Binary addition and subtraction table.

c_{in} or b_{in}	x	y	c_{out}	s	b_{out}	d
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	1	0	1
0	1	1	1	0	0	0
1	0	0	0	1	1	1
1	0	1	1	0	1	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

2.4 Addition and Subtraction of Nondecimal Numbers

Addition and subtraction of nondecimal numbers by hand uses the same technique that we learned in grammar school for decimal numbers; the only catch is that the addition and subtraction tables are different.

binary addition

Table 2-3 is the addition and subtraction table for binary digits. To add two binary numbers X and Y, we add together the least significant bits with an initial carry (c_{in}) of 0, producing carry (c_{out}) and sum (s) bits according to the table. We continue processing bits from right to left, adding the carry out of each column into the next column's sum.

Two examples of decimal additions and the corresponding binary additions are shown in Figure 2-1, using a colored arrow to indicate a carry of 1. The same examples are repeated below along with two more, with the carries shown as a bit string C:

C		101111000	C		001011000
X	190	10111110	X	173	10101101
Y	+141	+ 10001101	Y	+ 44	+ 00101100
$X+Y$	331	101001011	$X+Y$	217	11011001

C		011111110	C		000000000
X	127	01111111	X	170	10101010
Y	+ 63	+ 00111111	Y	+ 85	+ 01010101
$X+Y$	190	10111110	$X+Y$	255	11111111

binary subtraction

minuend

subtrahend

Binary subtraction is performed similarly, using borrows (b_{in} and b_{out}) instead of carries between steps, and producing a difference bit d. Two examples of decimal subtractions and the corresponding binary subtractions are shown in Figure 2-2. As in decimal subtraction, the binary minuend values in the columns are modified when borrows occur, as shown by the colored arrows and bits. The

					1	1	1	1			
X	190			1	0	1	1	1	1	1	0
Y	+ 141	+		1	0	0	0	1	1	0	1
$X + Y$	331		1	0	1	0	0	1	0	1	1

				1		1	1			
X	173		1	0	1	0	1	1	0	1
Y	+ 44	+	0	0	1	0	1	1	0	0
$X + Y$	217		1	1	0	1	1	0	0	1

Figure 2-1 Examples of decimal and corresponding binary additions.

examples from the figure are repeated below along with two more, this time showing the borrows as a bit string B:

B		001111100	B		011011010
X	229	11100101	X	210	11010010
Y	− 46	− 00101110	Y	−109	− 01101101
$X - Y$	183	10110111	$X - Y$	101	01100101
B		010101010	B		000000000
X	170	10101010	X	221	11011101
Y	− 85	− 01010101	Y	− 76	− 01001100
$X - Y$	85	01010101	$X - Y$	145	10010001

comparing numbers

A very common use of subtraction in computers is to compare two numbers. For example, if the operation $X - Y$ produces a borrow out of the most significant bit position, then X is less than Y; otherwise, X is greater than or equal to Y. The relationship between carries and borrow in adders and subtractors will be explored in Section 5.10.

Addition and subtraction tables can be developed for octal and hexadecimal digits, or any other desired radix. However, few computer engineers bother to memorize these tables. If you rarely need to manipulate nondecimal numbers,

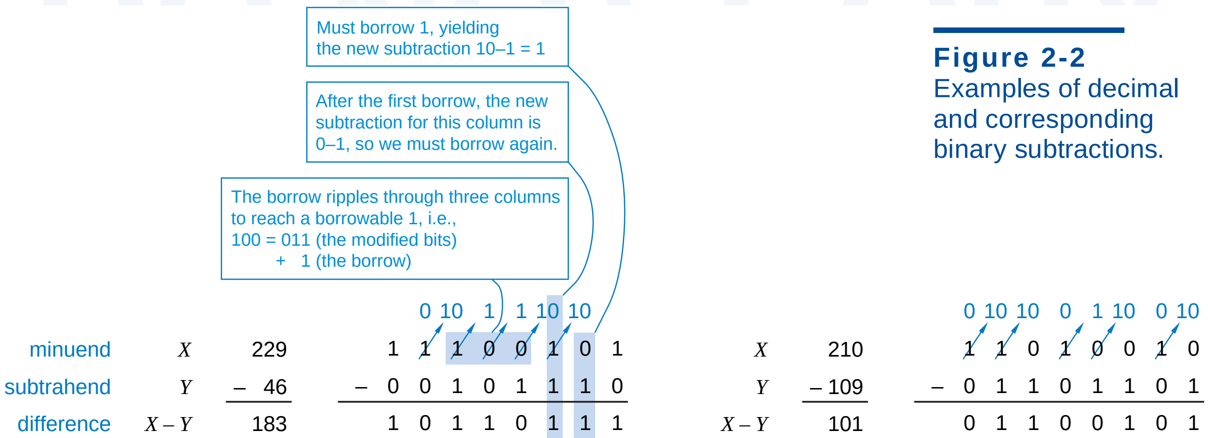


Figure 2-2 Examples of decimal and corresponding binary subtractions.

then it's easy enough on those occasions to convert them to decimal, calculate results, and convert back. On the other hand, if you must perform calculations in binary, octal, or hexadecimal frequently, then you should ask Santa for a programmer's "hex calculator" from Texas Instruments or Casio.

If the calculator's battery wears out, some mental shortcuts can be used to facilitate nondecimal arithmetic. In general, each column addition (or subtraction) can be done by converting the column digits to decimal, adding in decimal, and converting the result to corresponding sum and carry digits in the nondecimal radix. (A carry is produced whenever the column sum equals or exceeds the radix.) Since the addition is done in decimal, we rely on our knowledge of the decimal addition table; the only new thing that we need to learn is the conversion from decimal to nondecimal digits and vice versa. The sequence of steps for mentally adding two hexadecimal numbers is shown below:

hexadecimal addition

C		1	1	0	0		1	1	0	0
X		1	9	B	9	16	1	9	11	9
Y	+	C	7	E	6	16	+12	7	14	6
X + *Y*		E	1	9	F	16	14	17	25	15
							14	16+1	16+9	15
							E	1	9	F

2.5 Representation of Negative Numbers

So far, we have dealt only with positive numbers, but there are many ways to represent negative numbers. In everyday business, we use the signed-magnitude system, discussed next. However, most computers use one of the complement number systems that we introduce later.

2.5.1 Signed-Magnitude Representation

signed-magnitude system

In the *signed-magnitude system*, a number consists of a magnitude and a symbol indicating whether the magnitude is positive or negative. Thus, we interpret decimal numbers +98, –57, +123.5, and –13 in the usual way, and we also assume that the sign is "+" if no sign symbol is written. There are two possible representations of zero, "+0" and "–0", but both have the same value.

sign bit

The signed-magnitude system is applied to binary numbers by using an extra bit position to represent the sign (the *sign bit*). Traditionally, the most significant bit (MSB) of a bit string is used as the sign bit (0 = plus, 1 = minus), and the lower-order bits contain the magnitude. Thus, we can write several 8-bit signed-magnitude integers and their decimal equivalents:

$$01010101_2 = +85_{10} \qquad 11010101_2 = -85_{10}$$
$$01111111_2 = +127_{10} \qquad 11111111_2 = -127_{10}$$
$$00000000_2 = +0_{10} \qquad 10000000_2 = -0_{10}$$

The signed-magnitude system has an equal number of positive and negative integers. An n-bit signed-magnitude integer lies within the range $-(2^{n-1}-1)$ through $+(2^{n-1}-1)$, and there are two possible representations of zero.

Now suppose that we wanted to build a digital logic circuit that adds signed-magnitude numbers. The circuit must examine the signs of the addends to determine what to do with the magnitudes. If the signs are the same, it must add the magnitudes and give the result the same sign. If the signs are different, it must compare the magnitudes, subtract the smaller from the larger, and give the result the sign of the larger. All of these "ifs," "adds," "subtracts," and "compares" translate into a lot of logic-circuit complexity. Adders for complement number systems are much simpler, as we'll show next. Perhaps the one redeeming feature of a signed-magnitude system is that, once we know how to build a signed-magnitude adder, a signed-magnitude subtractor is almost trivial to build—it need only change the sign of the subtrahend and pass it along with the minuend to an adder.

signed-magnitude adder

signed-magnitude subtractor

2.5.2 Complement Number Systems

While the signed-magnitude system negates a number by changing its sign, a *complement number system* negates a number by taking its complement as defined by the system. Taking the complement is more difficult than changing the sign, but two numbers in a complement number system can be added or subtracted directly without the sign and magnitude checks required by the signed-magnitude system. We shall describe two complement number systems, called the "radix complement" and the "diminished radix-complement."

complement number system

In any complement number system, we normally deal with a fixed number of digits, say n. (However, we can increase the number of digits by "sign extension" as shown in Exercise 2.23, and decrease the number by truncating high-order digits as shown in Exercise 2.24.) We further assume that the radix is r, and that numbers have the form

$$D = d_{n-1}d_{n-2}\cdots d_1 d_0 .$$

The radix point is on the right and so the number is an integer. If an operation produces a result that requires more than n digits, we throw away the extra high-order digit(s). If a number D is complemented twice, the result is D.

2.5.3 Radix-Complement Representation

In a *radix-complement system*, the complement of an n-digit number is obtained by subtracting it from r^n. In the decimal number system, the radix complement is called the *10's complement*. Some examples using 4-digit decimal numbers (and subtraction from 10,000) are shown in Table 2-4.

radix-complement system

10's complement

By definition, the radix complement of an n-digit number D is obtained by subtracting it from r^n. If D is between 1 and $r^n - 1$, this subtraction produces

Table 2-4
Examples of 10's and 9s' complements.

Number	10's complement	9s' complement
1849	8151	8150
2067	7933	7932
100	9900	9899
7	9993	9992
8151	1849	1848
0	10000 (= 0)	9999

another number between 1 and $r^n - 1$. If D is 0, the result of the subtraction is r^n, which has the form $100 \cdots 00$, where there are a total of $n + 1$ digits. We throw away the extra high-order digit and get the result 0. Thus, there is only one representation of zero in a radix-complement system.

computing the radix complement

It seems from the definition that a subtraction operation is needed to compute the radix complement of D. However, this subtraction can be avoided by rewriting r^n as $(r^n - 1) + 1$ and $r^n - D$ as $((r^n - 1) - D) + 1$. The number $r^n - 1$ has the form $mm \cdots mm$, where $m = r - 1$ and there are n m's. For example, 10,000 equals 9,999 + 1. If we define the complement of a digit d to be $r - 1 - d$, then $(r^n - 1) - D$ is obtained by complementing the digits of D. Therefore, the radix complement of a number D is obtained by complementing the individual

Table 2-5
Digit complements.

	Complement			
Digit	Binary	Octal	Decimal	Hexadecimal
0	1	7	9	F
1	0	6	8	E
2	–	5	7	D
3	–	4	6	C
4	–	3	5	B
5	–	2	4	A
6	–	1	3	9
7	–	0	2	8
8	–	–	1	7
9	–	–	0	6
A	–	–	–	5
B	–	–	–	4
C	–	–	–	3
D	–	–	–	2
E	–	–	–	1
F	–	–	–	0

digits of D and adding 1. For example, the 10's complement of 1849 is 8150 + 1, or 8151. You should confirm that this trick also works for the other 10's-complement examples above. Table 2-5 lists the digit complements for binary, octal, decimal, and hexadecimal numbers.

2.5.4 Two's-Complement Representation

two's complement

For binary numbers, the radix complement is called the *two's complement*. The MSB of a number in this system serves as the sign bit; a number is negative if and only if its MSB is 1. The decimal equivalent for a two's-complement binary number is computed the same way as for an unsigned number, except that the weight of the MSB is -2^{n-1} instead of $+2^{n-1}$. The range of representable numbers is $-(2^{n-1})$ through $+(2^{n-1}-1)$. Some 8-bit examples are shown below:

weight of MSB

$$
\begin{array}{rcrl}
17_{10} & = & 00010001_2 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 11101110 & \\
 & & +1 & \\
\hline
 & & 11101111_2 & = -17_{10}
\end{array}
\qquad
\begin{array}{rcrl}
-99_{10} & = & 10011101_2 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 01100010 & \\
 & & +1 & \\
\hline
 & & 01100011_2 & = 99_{10}
\end{array}
$$

$$
\begin{array}{rcrl}
119_{10} & = & 01110111 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 10001000 & \\
 & & +1 & \\
\hline
 & & 10001001_2 & = -119_{10}
\end{array}
\qquad
\begin{array}{rcrl}
-127_{10} & = & 10000001 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 01111110 & \\
 & & +1 & \\
\hline
 & & 01111111_2 & = 127_{10}
\end{array}
$$

$$
\begin{array}{rcrl}
0_{10} & = & 00000000_2 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 11111111 & \\
 & & +1 & \\
\hline
 & & 1\ 00000000_2 & = 0_{10}
\end{array}
\qquad
\begin{array}{rcrl}
-128_{10} & = & 10000000_2 & \\
 & & \Downarrow & \text{complement bits} \\
 & & 01111111 & \\
 & & +1 & \\
\hline
 & & 10000000_2 & = -128_{10}
\end{array}
$$

A carry out of the MSB position occurs in one case, as shown in color above. As in all two's-complement operations, this bit is ignored and only the low-order n bits of the result are used.

In the two's-complement number system, zero is considered positive because its sign bit is 0. Since two's complement has only one representation of zero, we end up with one extra negative number, $-(2^{n-1})$, that doesn't have a positive counterpart.

extra negative number

We can convert an n-bit two's-complement number X into an m-bit one, but some care is needed. If $m > n$, we must append $m - n$ copies of X's sign bit to the left of X (see Exercise 2.23). That is, we pad a positive number with 0s and a negative one with 1s; this is called *sign extension*. If $m < n$, we discard X's $n - m$

sign extension

leftmost bits; however, the result is valid only if all of the discarded bits are the same as the sign bit of the result (see Exercise 2.24).

Most computers and other digital systems use the two's-complement system to represent negative numbers. However, for completeness, we'll also describe the diminished radix-complement and ones'-complement systems.

*2.5.5 Diminished Radix-Complement Representation

diminished radix-complement system

9s' complement

In a *diminished radix-complement system*, the complement of an n-digit number D is obtained by subtracting it from $r^n - 1$. This can be accomplished by complementing the individual digits of D, *without* adding 1 as in the radix-complement system. In decimal, this is called the *9s' complement*; some examples are given in the last column of Table 2-4 on page 32.

*2.5.6 Ones'-Complement Representation

ones' complement

The diminished radix-complement system for binary numbers is called the *ones' complement*. As in two's complement, the most significant bit is the sign, 0 if positive and 1 if negative. Thus there are two representations of zero, positive zero $(00\cdots00)$ and negative zero $(11\cdots11)$. Positive number representations are the same for both ones' and two's complements. However, negative number representations differ by 1. A weight of $-(2^{n-1} - 1)$, rather than -2^{n-1}, is given to the most significant bit when computing the decimal equivalent of a ones'-complement number. The range of representable numbers is $-(2^{n-1} - 1)$ through $+(2^{n-1} - 1)$. Some 8-bit numbers and their ones' complements are shown below:

$$17_{10} = 00010001_2 \quad\Downarrow\quad 11101110_2 = -17_{10}$$

$$-99_{10} = 10011100_2 \quad\Downarrow\quad 01100011_2 = 99_{10}$$

$$119_{10} = 01110111_2 \quad\Downarrow\quad 10001000_2 = -119_{10}$$

$$-127_{10} = 10000000_2 \quad\Downarrow\quad 01111111_2 = 127_{10}$$

$$0_{10} = 00000000_2 \text{ (positive zero)} \quad\Downarrow\quad 11111111_2 = 0_{10} \text{ (negative zero)}$$

The main advantages of the ones'-complement system are its symmetry and the ease of complementation. However, the adder design for ones'-complement numbers is somewhat trickier than a two's-complement adder (see Exercise 7.67). Also, zero-detecting circuits in a ones'-complement system

* Throughout this book, *optional sections* are marked with an asterisk.

either must check for both representations of zero, or must always convert $11 \cdots 11$ to $00 \cdots 00$.

*2.5.7 Excess Representations

excess-B representation

bias

excess-2^{m-1} system

Yes, the number of different systems for representing negative numbers *is* excessive, but there's just one more for us to cover. In *excess-B representation*, an m-bit string whose unsigned integer value is M ($0 \le M < 2^m$) represents the signed integer $M - B$, where B is called the *bias* of the number system.

For example, an *excess–2^{m-1} system* represents any number X in the range -2^{m-1} through $+2^{m-1} - 1$ by the m-bit binary representation of $X + 2^{m-1}$ (which is always nonnegative and less than 2^m). The range of this representation is exactly the same as that of m-bit two's-complement numbers. In fact, the representations of any number in the two systems are identical except for the sign bits, which are always opposite. (Note that this is true only when the bias is 2^{m-1}.)

The most common use of excess representations is in floating-point number systems (see References).

2.6 Two's-Complement Addition and Subtraction

2.6.1 Addition Rules

two's-complement addition

A table of decimal numbers and their equivalents in different number systems, Table 2-6, reveals why the two's complement is preferred for arithmetic operations. If we start with 1000_2 (-8_{10}) and count up, we see that each successive two's-complement number all the way to 0111_2 ($+7_{10}$) can be obtained by adding 1 to the previous one, ignoring any carries beyond the fourth bit position. The same cannot be said of signed-magnitude and ones'-complement numbers. Because ordinary addition is just an extension of counting, two's-complement numbers can thus be added by ordinary binary addition, ignoring any carries beyond the MSB. The result will always be the correct sum as long as the range of the number system is not exceeded. Some examples of decimal addition and the corresponding 4-bit two's-complement additions confirm this:

$$
\begin{array}{rr}
 & +3 \\
+ & +4 \\
\hline
 & +7
\end{array}
\quad
\begin{array}{rr}
 & 0011 \\
+ & 0100 \\
\hline
 & 0111
\end{array}
\qquad\qquad
\begin{array}{rr}
 & -2 \\
+ & -6 \\
\hline
 & -8
\end{array}
\quad
\begin{array}{rr}
 & 1110 \\
+ & 1010 \\
\hline
 & 11000
\end{array}
$$

$$
\begin{array}{rr}
 & +6 \\
+ & -3 \\
\hline
 & +3
\end{array}
\quad
\begin{array}{rr}
 & 0110 \\
+ & 1101 \\
\hline
 & 10011
\end{array}
\qquad\qquad
\begin{array}{rr}
 & +4 \\
+ & -7 \\
\hline
 & -3
\end{array}
\quad
\begin{array}{rr}
 & 0100 \\
+ & 1001 \\
\hline
 & 1101
\end{array}
$$

Table 2-6 Decimal and 4-bit numbers.

Decimal	*Two's Complement*	*Ones' Complement*	*Signed Magnitude*	*Excess 2^{m-1}*
–8	1000	—	—	0000
–7	1001	1000	1111	0001
–6	1010	1001	1110	0010
–5	1011	1010	1101	0011
–4	1100	1011	1100	0100
–3	1101	1100	1011	0101
–2	1110	1101	1010	0110
–1	1111	1110	1001	0111
0	0000	1111 or 0000	1000 or 0000	1000
1	0001	0001	0001	1001
2	0010	0010	0010	1010
3	0011	0011	0011	1011
4	0100	0100	0100	1100
5	0101	0101	0101	1101
6	0110	0110	0110	1110
7	0111	0111	0111	1111

2.6.2 A Graphical View

Another way to view the two's-complement system uses the 4-bit "counter" shown in Figure 2-3. Here we have shown the numbers in a circular or "modular" representation. The operation of this counter very closely mimics that of a real up/down counter circuit, which we'll study in Section 8.4. Starting

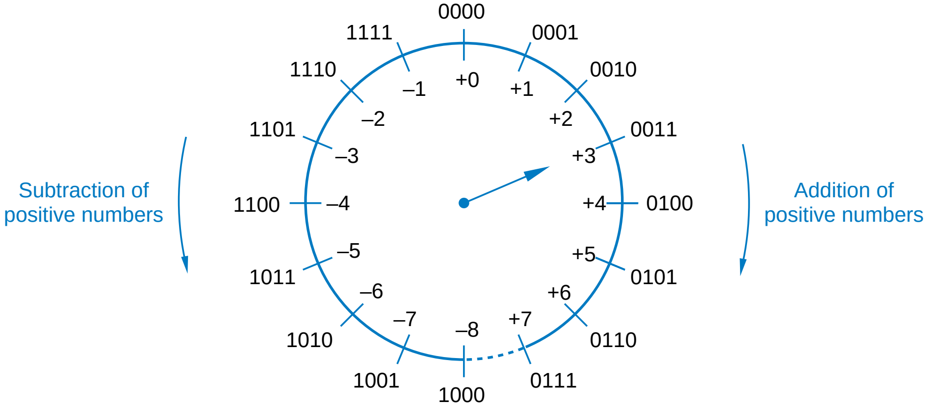


Figure 2-3
A modular counting representation of 4-bit two's-complement numbers.

with the arrow pointing to any number, we can add $+n$ to that number by counting up n times, that is, by moving the arrow n positions clockwise. It is also evident that we can subtract n from a number by counting down n times, that is, by moving the arrow n positions counterclockwise. Of course, these operations give correct results only if n is small enough that we don't cross the discontinuity between −8 and +7.

What is most interesting is that we can also subtract n (or add $-n$) by moving the arrow $16 - n$ positions clockwise. Notice that the quantity $16 - n$ is what we defined to be the 4-bit two's complement of n, that is, the two's-complement representation of $-n$. This graphically supports our earlier claim that a negative number in two's-complement representation may be added to another number simply by adding the 4-bit representations using ordinary binary addition. Adding a number in Figure 2-3 is equivalent to moving the arrow a corresponding number of positions clockwise.

2.6.3 Overflow

overflow

If an addition operation produces a result that exceeds the range of the number system, *overflow* is said to occur. In the modular counting representation of Figure 2-3, overflow occurs during addition of positive numbers when we count past +7. Addition of two numbers with different signs can never produce overflow, but addition of two numbers of like sign can, as shown by the following examples:

$$
\begin{array}{rr}
-3 & 1101 \\
+\ -6 & +\ 1010 \\
\hline
-9 & 10111 = +7
\end{array}
\qquad
\begin{array}{rr}
+5 & 0101 \\
+\ +6 & +\ 0110 \\
\hline
+11 & 1011 = -5
\end{array}
$$

$$
\begin{array}{rr}
-8 & 1000 \\
+\ -8 & +\ 1000 \\
\hline
-16 & 10000 = +0
\end{array}
\qquad
\begin{array}{rr}
+7 & 0111 \\
+\ +7 & +\ 0111 \\
\hline
+14 & 1110 = -2
\end{array}
$$

overflow rules

Fortunately, there is a simple rule for detecting overflow in addition: An addition overflows if the signs of the addends are the same and the sign of the sum is different from the addends' sign. The overflow rule is sometimes stated in terms of carries generated during the addition operation: An addition overflows if the carry bits c_{in} into and c_{out} out of the sign position are different. Close examination of Table 2-3 on page 28 shows that the two rules are equivalent—there are only two cases where $c_{\text{in}} \neq c_{\text{out}}$, and these are the only two cases where $x = y$ and the sum bit is different.

2.6.4 Subtraction Rules

two's-complement subtraction

Two's-complement numbers may be subtracted as if they were ordinary unsigned binary numbers, and appropriate rules for detecting overflow may be formulated. However, most subtraction circuits for two's-complement numbers do not perform subtraction directly. Rather, they negate the subtrahend by taking its two’s complement, and then add it to the minuend using the normal rules for addition.

Negating the subtrahend and adding the minuend can be accomplished with only one addition operation as follows: Perform a bit-by-bit complement of the subtrahend and add the complemented subtrahend to the minuend with an initial carry (c_{in}) of 1 instead of 0. Examples are given below:

$$
\begin{array}{rrrl}
 & & 1 & \text{—}\ c_{in} \\
+4 & 0100 & 0100 & \\
-\ \ +3 & -\ \ 0011 & +\ \ 1100 & \\
\hline
+3 & & 10001 &
\end{array}
\qquad
\begin{array}{rrrl}
 & & 1 & \text{—}\ c_{in} \\
+3 & 0011 & 0011 & \\
-\ \ +4 & -\ \ 0100 & +\ \ 1011 & \\
\hline
-1 & & 1111 &
\end{array}
$$

$$
\begin{array}{rrrl}
 & & 1 & \text{—}\ c_{in} \\
+3 & 0011 & 0011 & \\
-\ \ -4 & -\ \ 1100 & +\ \ 0011 & \\
\hline
+7 & & 0111 &
\end{array}
\qquad
\begin{array}{rrrl}
 & & 1 & \text{—}\ c_{in} \\
-3 & 1101 & 1101 & \\
-\ \ -4 & -\ \ 1100 & +\ \ 0011 & \\
\hline
+1 & & 10001 &
\end{array}
$$

Overflow in subtraction can be detected by examining the signs of the minuend and the *complemented* subtrahend, using the same rule as in addition. Or, using the technique in the preceding examples, the carries into and out of the sign position can be observed and overflow detected irrespective of the signs of inputs and output, again using the same rule as in addition.

An attempt to negate the “extra” negative number results in overflow according to the rules above, when we add 1 in the complementation process:

$$
\begin{array}{rrl}
-(-8) = -1000 = & 0111 & \\
 & +\ \ 0001 & \\
\hline
 & 1000 & = -8
\end{array}
$$

However, this number can still be used in additions and subtractions as long as the final result does not exceed the number range:

$$
\begin{array}{rr}
+4 & 0100 \\
+\ \ -8 & +\ \ 1000 \\
\hline
-4 & 1100
\end{array}
\qquad
\begin{array}{rrrl}
 & & 1 & \text{—}\ c_{in} \\
-3 & 1101 & 1101 & \\
-\ \ -8 & -\ \ 1000 & +\ \ 0111 & \\
\hline
+5 & & 10101 &
\end{array}
$$

2.6.5 Two’s-Complement and Unsigned Binary Numbers

Since two’s-complement numbers are added and subtracted by the same basic binary addition and subtraction algorithms as unsigned numbers of the same length, a computer or other digital system can use the same adder circuit to handle numbers of both types. However, the results must be interpreted differently depending on whether the system is dealing with signed numbers (e.g., −8 through +7) or unsigned numbers (e.g., 0 through 15).

signed vs. unsigned numbers

We introduced a graphical representation of the 4-bit two's-complement system in Figure 2-3. We can relabel this figure as shown in Figure 2-4 to obtain a representation of the 4-bit unsigned numbers. The binary combinations occupy the same positions on the wheel, and a number is still added by moving the arrow a corresponding number of positions clockwise, and subtracted by moving the arrow counterclockwise.

An addition operation can be seen to exceed the range of the 4-bit unsigned number system in Figure 2-4 if the arrow moves clockwise through the discontinuity between 0 and 15. In this case a *carry* out of the most significant bit position is said to occur.

carry

Likewise a subtraction operation exceeds the range of the number system if the arrow moves counterclockwise through the discontinuity. In this case a *borrow* out of the most significant bit position is said to occur.

borrow

From Figure 2-4 it is also evident that we may subtract an unsigned number n by counting *clockwise* $16 - n$ positions. This is equivalent to *adding* the 4-bit two's-complement of n. The subtraction produces a borrow if the corresponding addition of the two's complement *does not* produce a carry.

In summary, in unsigned addition the carry or borrow in the most significant bit position indicates an out-of-range result. In signed, two's-complement addition the overflow condition defined earlier indicates an out-of-range result. The carry from the most significant bit position is irrelevant in signed addition in the sense that overflow may or may not occur independently of whether or not a carry occurs.

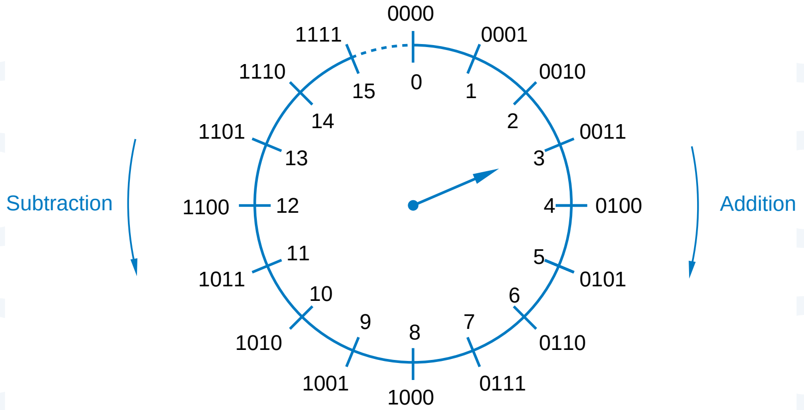


Figure 2-4
A modular counting representation of 4-bit unsigned numbers.

*2.7 Ones'-Complement Addition and Subtraction

Another look at Table 2-6 helps to explain the rule for adding ones'-complement numbers. If we start at 1000_2 (-7_{10}) and count up, we obtain each successive ones'-complement number by adding 1 to the previous one, *except* at the transition from 1111_2 (negative 0) to 0001_2 ($+1_{10}$). To maintain the proper count, we must add 2 instead of 1 whenever we count past 1111_2. This suggests a technique for adding ones'-complement numbers: Perform a standard binary addition, but add an extra 1 whenever we count past 1111_2.

ones'-complement addition

Counting past 1111_2 during an addition can be detected by observing the carry out of the sign position. Thus, the rule for adding ones'-complement numbers can be stated quite simply:

- Perform a standard binary addition; if there is a carry out of the sign position, add 1 to the result.

end-around carry

This rule is often called *end-around carry*. Examples of ones'-complement addition are given below; the last three include an end-around carry:

$$
\begin{array}{rr}
+3 & 0011 \\
+\ +4 & +\ 0100 \\
\hline
+7 & 0111
\end{array}
\qquad
\begin{array}{rr}
+4 & 0100 \\
+\ -7 & +\ 1000 \\
\hline
-3 & 1100
\end{array}
\qquad
\begin{array}{rr}
+5 & 0101 \\
+\ -5 & +\ 1010 \\
\hline
-0 & 1111
\end{array}
$$

$$
\begin{array}{rr}
-2 & 1101 \\
+\ -5 & +\ 1010 \\
\hline
-7 & 10111 \\
 & +\quad 1 \\
\cline{2-2}
 & 1000
\end{array}
\qquad
\begin{array}{rr}
+6 & 0110 \\
+\ -3 & +\ 1100 \\
\hline
+3 & 10010 \\
 & +\quad 1 \\
\cline{2-2}
 & 0011
\end{array}
\qquad
\begin{array}{rr}
-0 & 1111 \\
+\ -0 & +\ 1111 \\
\hline
-0 & 11110 \\
 & +\quad 1 \\
\cline{2-2}
 & 1111
\end{array}
$$

Following the two-step addition rule above, the addition of a number and its ones' complement produces negative 0. In fact, an addition operation using this rule can never produce positive 0 unless both addends are positive 0.

ones'-complement subtraction

As with two's complement, the easiest way to do ones'-complement subtraction is to complement the subtrahend and add. Overflow rules for ones'-complement addition and subtraction are the same as for two's complement.

Table 2-7 summarizes the rules that we presented in this and previous sections for negation, addition, and subtraction in binary number systems.

Table 2-7 Summary of addition and subtraction rules for binary numbers.

Number System	***Addition Rules***	***Negation Rules***	***Subtraction Rules***
Unsigned	Add the numbers. Result is out of range if a carry out of the MSB occurs.	Not applicable	Subtract the subtrahend from the minuend. Result is out of range if a borrow out of the MSB occurs.
Signed magnitude	(same sign) Add the magnitudes; overflow occurs if a carry out of MSB occurs; result has the same sign. (opposite sign) Subtract the smaller magnitude from the larger; overflow is impossible; result has the sign of the larger.	Change the number's sign bit.	Change the sign bit of the subtrahend and proceed as in addition.
Two's complement	Add, ignoring any carry out of the MSB. Overflow occurs if the carries into and out of MSB are different.	Complement all bits of the number; add 1 to the result.	Complement all bits of the subtrahend and add to the minuend with an initial carry of 1.
Ones' complement	Add; if there is a carry out of the MSB, add 1 to the result. Overflow if carries into and out of MSB are different.	Complement all bits of the number.	Complement all bits of the subtrahend and proceed as in addition.

*2.8 Binary Multiplication

shift-and-add multiplication

unsigned binary multiplication

In grammar school we learned to multiply by adding a list of shifted multiplicands computed according to the digits of the multiplier. The same method can be used to obtain the product of two unsigned binary numbers. Forming the shifted multiplicands is trivial in binary multiplication, since the only possible values of the multiplier digits are 0 and 1. An example is shown below:

$$
\begin{array}{r}
11 \\
\times\ 13 \\
\hline
33 \\
11 \\
\hline
143
\end{array}
\qquad\qquad
\begin{array}{rl}
1011 & \text{multiplicand} \\
\times\quad 1101 & \text{multiplier} \\
\hline
1011 & \\
0000 & \\
1011 & \\
1011 & \\
\hline
10001111 & \text{product}
\end{array}
\left.\begin{array}{l}\\ \\ \\ \\ \end{array}\right\}\text{shifted multiplicands}
$$

partial product

Instead of listing all the shifted multiplicands and then adding, in a digital system it is more convenient to add each shifted multiplicand as it is created to a *partial product*. Applying this technique to the previous example, four additions and partial products are used to multiply 4-bit numbers:

11	1011	multiplicand
× 13	× 1101	multiplier
	0000	partial product
	1011	shifted multiplicand
	01011	partial product
	0000↓	shifted multiplicand
	001011	partial product
	1011↓↓	shifted multiplicand
	0110111	partial product
	1011↓↓↓	shifted multiplicand
	10001111	product

In general, when we multiply an n-bit number by an m-bit number, the resulting product requires at most $n + m$ bits to express. The shift-and-add algorithm requires m partial products and additions to obtain the result, but the first addition is trivial, since the first partial product is zero. Although the first partial product has only n significant bits, after each addition step the partial product gains one more significant bit, since each addition may produce a carry. At the same time, each step yields one more partial product bit, starting with the rightmost and working toward the left, that does not change. The shift-and-add algorithm can be performed by a digital circuit that includes a shift register, an adder, and control logic, as shown in Section 8.7.2.

signed multiplication

Multiplication of signed numbers can be accomplished using unsigned multiplication and the usual grammar school rules: Perform an unsigned multiplication of the magnitudes and make the product positive if the operands had the same sign, negative if they had different signs. This is very convenient in signed-magnitude systems, since the sign and magnitude are separate.

two's-complement multiplication

In the two's-complement system, obtaining the magnitude of a negative number and negating the unsigned product are nontrivial operations. This leads us to seek a more efficient way of performing two's-complement multiplication, described next.

Conceptually, unsigned multiplication is accomplished by a sequence of unsigned additions of the shifted multiplicands; at each step, the shift of the multiplicand corresponds to the weight of the multiplier bit. The bits in a two's-complement number have the same weights as in an unsigned number, except for the MSB, which has a negative weight (see Section 2.5.4). Thus, we can perform two's-complement multiplication by a sequence of two's-complement additions of shifted multiplicands, except for the last step, in which the shifted multiplicand corresponding to the MSB of the multiplier must be negated before it is added to the partial product. Our previous example is repeated below, this time interpreting the multiplier and multiplicand as two's-complement numbers:

−5	1011	multiplicand
× −3	× 1101	multiplier
	00000	partial product
	11011	shifted multiplicand
	111011	partial product
	00000↓	shifted multiplicand
	1111011	partial product
	11011↓↓	shifted multiplicand
	11100111	partial product
	00101↓↓↓	shifted and negated multiplicand
	00001111	product

Handling the MSBs is a little tricky because we gain one significant bit at each step and we are working with signed numbers. Therefore, before adding each shifted multiplicand and k-bit partial product, we change them to $k + 1$ significant bits by sign extension, as shown in color above. Each resulting sum has $k + 1$ bits; any carry out of the MSB of the $k + 1$-bit sum is ignored.

*2.9 Binary Division

The simplest binary division algorithm is based on the shift-and-subtract method that we learned in grammar school. Table 2-8 gives examples of this method for unsigned decimal and binary numbers. In both cases, we mentally compare the

shift-and-subtract division

unsigned division

Table 2-8 Example of long division.

19		10011	quotient
11)217	1011	)11011001	dividend
11		1011	shifted divisor
107		0101	reduced dividend
99		0000	shifted divisor
8		1010	reduced dividend
		0000	shifted divisor
		10100	reduced dividend
		1011	shifted divisor
		10011	reduced dividend
		1011	shifted divisor
		1000	remainder

reduced dividend with multiples of the divisor to determine which multiple of the shifted divisor to subtract. In the decimal case, we first pick 11 as the greatest multiple of 11 less than 21, and then pick 99 as the greatest multiple less than 107. In the binary case, the choice is somewhat simpler, since the only two choices are zero and the divisor itself.

Division methods for binary numbers are somewhat complementary to binary multiplication methods. A typical division algorithm accepts an n+m-bit dividend and an n-bit divisor, and produces an m-bit quotient and an n-bit remainder. A division *overflows* if the divisor is zero or the quotient would take more than m bits to express. In most computer division circuits, $n = m$.

division overflow

signed division

Division of signed numbers can be accomplished using unsigned division and the usual grammar school rules: Perform an unsigned division of the magnitudes and make the quotient positive if the operands had the same sign, negative if they had different signs. The remainder should be given the same sign as the dividend. As in multiplication, there are special techniques for performing division directly on two's-complement numbers; these techniques are often implemented in computer division circuits (see References).

2.10 Binary Codes for Decimal Numbers

Even though binary numbers are the most appropriate for the internal computations of a digital system, most people still prefer to deal with decimal numbers. As a result, the external interfaces of a digital system may read or display decimal numbers, and some digital devices actually process decimal numbers directly.

The human need to represent decimal numbers doesn't change the basic nature of digital electronic circuits—they still process signals that take on one of only two states that we call 0 and 1. Therefore, a decimal number is represented in a digital system by a string of bits, where different combinations of bit values in the string represent different decimal numbers. For example, if we use a 4-bit string to represent a decimal number, we might assign bit combination 0000 to decimal digit 0, 0001 to 1, 0010 to 2, and so on.

code
code word

A set of n-bit strings in which different bit strings represent different numbers or other things is called a *code*. A particular combination of n bit-values is called a *code word*. As we'll see in the examples of decimal codes in this section, there may or may not be an arithmetic relationship between the bit values in a code word and the thing that it represents. Furthermore, a code that uses n-bit strings need not contain 2^n valid code words.

At least four bits are needed to represent the ten decimal digits. There are billions and billions of different ways to choose ten 4-bit code words, but some of the more common decimal codes are listed in Table 2-9.

binary-coded decimal (BCD)

Perhaps the most "natural" decimal code is *binary-coded decimal (BCD)*, which encodes the digits 0 through 9 by their 4-bit unsigned binary representa-

Table 2-9 Decimal codes.

Decimal digit	*BCD (8421)*	*2421*	*Excess-3*	*Biquinary*	*1-out-of-10*
0	0000	0000	0011	0100001	1000000000
1	0001	0001	0100	0100010	0100000000
2	0010	0010	0101	0100100	0010000000
3	0011	0011	0110	0101000	0001000000
4	0100	0100	0111	0110000	0000100000
5	0101	1011	1000	1000001	0000010000
6	0110	1100	1001	1000010	0000001000
7	0111	1101	1010	1000100	0000000100
8	1000	1110	1011	1001000	0000000010
9	1001	1111	1100	1010000	0000000001
		Unused code words			
	1010	0101	0000	0000000	0000000000
	1011	0110	0001	0000001	0000000011
	1100	0111	0010	0000010	0000000101
	1101	1000	1101	0000011	0000000110
	1110	1001	1110	0000101	0000000111
	1111	1010	1111	. . .	. . .

tions, 0000 through 1001. The code words 1010 through 1111 are not used. Conversions between BCD and decimal representations are trivial, a direct substitution of four bits for each decimal digit. Some computer programs place two BCD digits in one 8-bit byte in *packed-BCD representation*; thus, one byte may represent the values from 0 to 99 as opposed to 0 to 255 for a normal unsigned 8-bit binary number. BCD numbers with any desired number of digits may be obtained by using one byte for each two digits.

packed-BCD representation

As with binary numbers, there are many possible representations of negative BCD numbers. Signed BCD numbers have one extra digit position for the

BINOMIAL COEFFICIENTS

The number of different ways to choose m items from a set of n items is given by a *binomial coefficient*, denoted $\binom{n}{m}$, whose value is $\frac{n!}{m! \cdot (n-m!)}$. For a 4-bit decimal code, there are $\binom{16}{10}$ different ways to choose 10 out of 16 4-bit code words, and 10! ways to assign each different choice to the 10 digits. So there are $\frac{16!}{10! \cdot 6!} \cdot 10!$ or 29,059,430,400 different 4-bit decimal codes.

sign. Both the signed-magnitude and 10's-complement representations are popular. In signed-magnitude BCD, the encoding of the sign bit string is arbitrary; in 10's-complement, 0000 indicates plus and 1001 indicates minus.

BCD addition

Addition of BCD digits is similar to adding 4-bit unsigned binary numbers, except that a correction must be made if a result exceeds 1001. The result is corrected by adding 6; examples are shown below:

```
    5      0101                     4      0100
 +  9   + 1001                   +  5   + 0101
 ----   ------                   ----   ------
   14     1110                      9     1001
        + 0110 — correction
        ------
 10+4   1 0100

    8      1000                     9      1001
 +  8   + 1000                    + 9   + 1001
 ----   ------                   ----   ------
  –16   1 0000                     18   1 0010
        + 0110 — correction             + 0110 — correction
        ------                          ------
 10+6   1 0110                   10+8   1 1000
```

Notice that the addition of two BCD digits produces a carry into the next digit position if either the initial binary addition or the correction factor addition produces a carry. Many computers perform packed-BCD arithmetic using special instructions that handle the carry correction automatically.

weighted code

8421 code

2421 code

self-complementing code

Binary-coded decimal is a *weighted code* because each decimal digit can be obtained from its code word by assigning a fixed weight to each code-word bit. The weights for the BCD bits are 8, 4, 2, and 1, and for this reason the code is sometimes called the *8421 code*. Another set of weights results in the *2421 code* shown in Table 2-9. This code has the advantage that it is *self-complementing*, that is, the code word for the 9s' complement of any digit may be obtained by complementing the individual bits of the digit's code word.

excess-3 code

Another self-complementing code shown in Table 2-9 is the *excess-3 code*. Although this code is not weighted, it has an arithmetic relationship with the BCD code—the code word for each decimal digit is the corresponding BCD code word plus 0011_2. Because the code words follow a standard binary counting sequence, standard binary counters can easily be made to count in excess-3 code, as we'll show in Figure 8-37 on page 600.

biquinary code

Decimal codes can have more than four bits; for example, the *biquinary code* in Table 2-9 uses seven. The first two bits in a code word indicate whether the number is in the range 0–4 or 5–9, and the last five bits indicate which of the five numbers in the selected range is represented.

One potential advantage of using more than the minimum number of bits in a code is an error-detecting property. In the biquinary code, if any one bit in a code word is accidentally changed to the opposite value, the resulting code word does not represent a decimal digit and can therefore be flagged as an error. Out of 128 possible 7-bit code words, only 10 are valid and recognized as decimal digits; the rest can be flagged as errors if they appear.

A *1-out-of-10 code* such as the one shown in the last column of Table 2-9 is the sparsest encoding for decimal digits, using 10 out of 1024 possible 10-bit code words.

1-out-of-10 code

2.11 Gray Code

In electromechanical applications of digital systems—such as machine tools, automotive braking systems, and copiers—it is sometimes necessary for an input sensor to produce a digital value that indicates a mechanical position. For example, Figure 2-5 is a conceptual sketch of an encoding disk and a set of contacts that produce one of eight 3-bit binary-coded values depending on the rotational position of the disk. The dark areas of the disk are connected to a signal source corresponding to logic 1, and the light areas are unconnected, which the contacts interpret as logic 0.

The encoder in Figure 2-5 has a problem when the disk is positioned at certain boundaries between the regions. For example, consider the boundary between the 001 and 010 regions of the disk; two of the encoded bits change here. What value will the encoder produce if the disk is positioned right on the theoretical boundary? Since we're on the border, both 001 and 010 are acceptable. However, because the mechanical assembly is not perfect, the two right-hand contacts may both touch a "1" region, giving an incorrect reading of 011. Likewise, a reading of 000 is possible. In general, this sort of problem can occur at any boundary where more than one bit changes. The worst problems occur when all three bits are changing, as at the 000–111 and 011–100 boundaries.

The encoding-disk problem can be solved by devising a digital code in which only one bit changes between each pair of successive code words. Such a code is called a *Gray code*; a 3-bit Gray code is listed in Table 2-10. We've rede-

Gray code

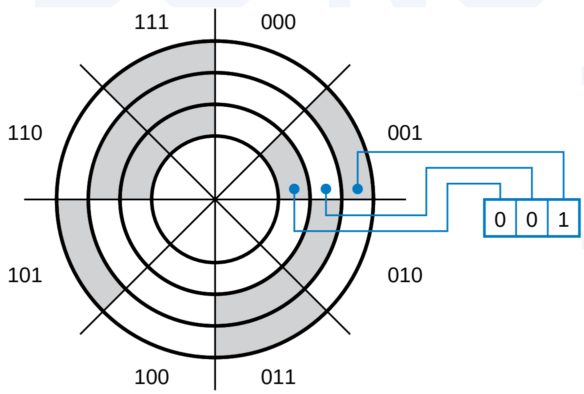


Figure 2-5
A mechanical encoding disk using a 3-bit binary code.

Table 2-10
A comparison of 3-bit binary code and Gray code.

Decimal number	*Binary code*	*Gray code*
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

signed the encoding disk using this code as shown in Figure 2-6. Only one bit of the new disk changes at each border, so borderline readings give us a value on one side or the other of the border.

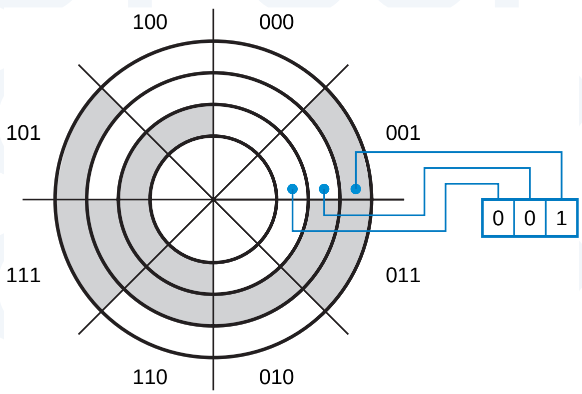
There are two convenient ways to construct a Gray code with any desired number of bits. The first method is based on the fact that Gray code is a *reflected code*; it can be defined (and constructed) recursively using the following rules:

reflected code

1. A 1-bit Gray code has two code words, 0 and 1.
2. The first 2^n code words of an n+1-bit Gray code equal the code words of an n-bit Gray code, written in order with a leading 0 appended.
3. The last 2^n code words of an n+1-bit Gray code equal the code words of an n-bit Gray code, but written in reverse order with a leading 1 appended.

If we draw a line between rows 3 and 4 of Table 2-10, we can see that rules 2 and 3 are true for the 3-bit Gray code. Of course, to construct an n-bit Gray code for an arbitrary value of n with this method, we must also construct a Gray code of each length smaller than n.

Figure 2-6
A mechanical encoding disk using a 3-bit Gray code.



The second method allows us to derive an n-bit Gray-code code word directly from the corresponding n-bit binary code word:

1. The bits of an n-bit binary or Gray-code code word are numbered from right to left, from 0 to $n-1$.
2. Bit i of a Gray-code code word is 0 if bits i and $i+1$ of the corresponding binary code word are the same, else bit i is 1. (When $i+1=n$, bit n of the binary code word is considered to be 0.)

Again, inspection of Table 2-10 shows that this is true for the 3-bit Gray code.

*2.12 Character Codes

As we showed in the preceding section, a string of bits need not represent a number, and in fact most of the information processed by computers is nonnumeric. The most common type of nonnumeric data is *text*, strings of characters from some character set. Each character is represented in the computer by a bit string according to an established convention.

text

The most commonly used character code is *ASCII* (pronounced *ASS key*), the American Standard Code for Information Interchange. ASCII represents each character with a 7-bit string, yielding a total of 128 different characters shown in Table 2-11. The code contains the uppercase and lowercase alphabet, numerals, punctuation, and various nonprinting control characters. Thus, the text string "Yeccch!" is represented by a rather innocuous-looking list of seven 7-bit numbers:

ASCII

1011001 1100101 1100011 1100011 1100011 1101000 0100001

2.13 Codes for Actions, Conditions, and States

The codes that we've described so far are generally used to represent things that we would probably consider to be "data"—things like numbers, positions, and characters. Programmers know that dozens of different data types can be used in a single computer program.

In digital system design, we often encounter nondata applications where a string of bits must be used to control an action, to flag a condition, or to represent the current state of the hardware. Probably the most commonly used type of code for such an application is a simple binary code. If there are n different actions, conditions, or states, we can represent them with a b-bit binary code with $b = \lceil \log_2 n \rceil$ bits. (The brackets $\lceil\ \rceil$ denote the *ceiling function*—the smallest integer greater than or equal to the bracketed quantity. Thus, b is the smallest integer such that $2^b \geq n$.)

$\lceil\ \rceil$

ceiling function

For example, consider a simple traffic-light controller. The signals at the intersection of a north-south (N-S) and an east-west (E-W) street might be in any

Table 2-11 American Standard Code for Information Interchange (ASCII), Standard No. X3.4-1968 of the American National Standards Institute.

		$b_6b_5b_4$ (column)							
$b_3b_2b_1b_0$	Row (hex)	000 0	001 1	010 2	011 3	100 4	101 5	110 6	111 7
0000	0	NUL	DLE	SP	0	@	P	‘	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	’	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[	k	{
1100	C	FF	FS	,	<	L	\	l	|
1101	D	CR	GS	–	=	M	]	m	}
1110	E	SO	RS	.	>	N	^	n	~
1111	F	SI	US	/	?	O	_	o	DEL

Control codes

NUL	Null	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronize
BEL	Bell	ETB	End transmitted block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete or rubout

Table 2-12 States in a traffic-light controller.

State	Lights						Code word
	N-S green	N-S yellow	N-S red	E-W green	E-W yellow	E-W red	
N-S go	ON	off	off	off	off	ON	000
N-S wait	off	ON	off	off	off	ON	001
N-S delay	off	off	ON	off	off	ON	010
E-W go	off	off	ON	ON	off	off	100
E-W wait	off	off	ON	off	ON	off	101
E-W delay	off	off	ON	off	off	ON	110

of the six states listed in Table 2-12. These states can be encoded in three bits, as shown in the last column of the table. Only six of the eight possible 3-bit code words are used, and the assignment of the six chosen code words to states is arbitrary, so many other encodings are possible. An experienced digital designer chooses a particular encoding to minimize circuit cost or to optimize some other parameter (like design time—there's no need to try billions and billions of possible encodings).

Another application of a binary code is illustrated in Figure 2-7(a). Here, we have a system with n devices, each of which can perform a certain action. The characteristics of the devices are such that they may be enabled to operate only one at a time. The control unit produces a binary-coded "device select" word with $\lceil \log_2 n \rceil$ bits to indicate which device is enabled at any time. The "device select" code word is applied to each device, which compares it with its own "device ID" to determine whether it is enabled.Although its code words have the minimum number of bits, a binary code isn't always the best choice for encoding actions, conditions, or states. Figure 2-7(b) shows how to control n devices with a *1-out-of-n code*, an n-bit code in which valid code words have one bit equal to 1 and the rest of the bits equal to 0. Each bit of the 1-out-of-n code word is connected directly to the enable input of a corresponding device. This simplifies the design of the devices, since they no longer have device IDs; they need only a single "enable" input bit.

1-out-of-n code

The code words of a 1-out-of-10 code were listed in Table 2-9. Sometimes an all-0s word may also be included in a 1-out-of-n code, to indicate that no device is selected. Another common code is an *inverted 1-out-of-n code*, in which valid code words have one 0~bit and the rest of the bits equal to 1.

inverted 1-out-of-n code

In complex systems, a combination of coding techniques may be used. For example, consider a system similar to Figure 2-7(b), in which each of the n devices contains up to s subdevices. The control unit could produce a device

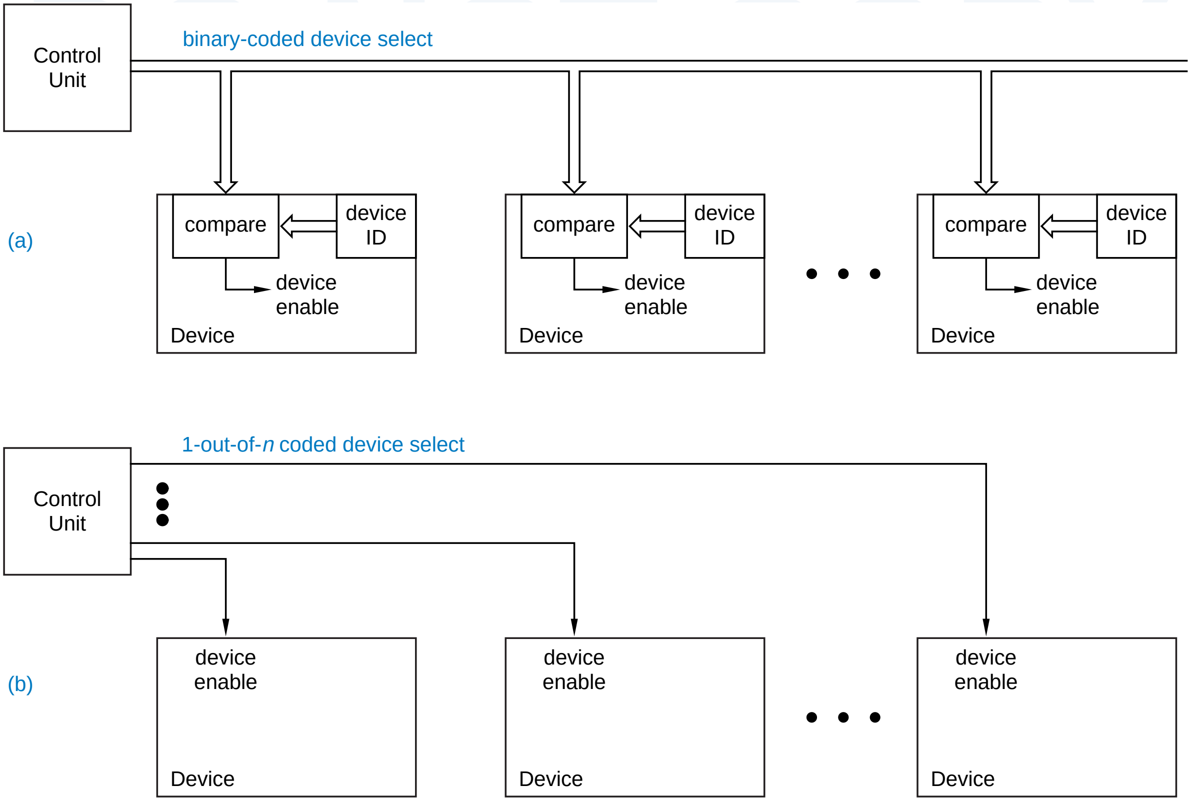


Figure 2-7 Control structure for a digital system with n devices: (a) using a binary code; (b) using a 1-out-of-n code.

select code word with a 1-out-of-n coded field to select a device, and a $\lceil \log_2 s \rceil$-bit binary-coded field to select one of the s subdevices of the selected device.

m-out-of-n code

An *m-out-of-n code* is a generalization of the 1-out-of-n code in which valid code words have m bits equal to 1 and the rest of the bits equal to 0. A valid m-out-of-n code word can be detected with an m-input AND gate, which produces a 1 output if all of its inputs are 1. This is fairly simple and inexpensive to do, yet for most values of m, an m-out-of-n code typically has far more valid code words than a 1-out-of-n code. The total number of code words is given by the binomial coefficient $\binom{n}{m}$, which has the value $\frac{n!}{m! \cdot (n-m)!}$. Thus, a 2-out-of-4 code has 6 valid code words, and a 3-out-of-10 code has 120.

8B10B code

An important variation of an m-out-of-n code is the *8B10B code* used in the 802.3z Gigabit Ethernet standard. This code uses 10 bits to represent 256 valid code words, or 8 bits worth of data. Most code words use a 5-out-of-10 coding. However, since $\binom{10}{5}$ is only 252, some 4- and 6-out-of-10 words are also used to complete the code in a very interesting way; more on this in Section 2.16.2.

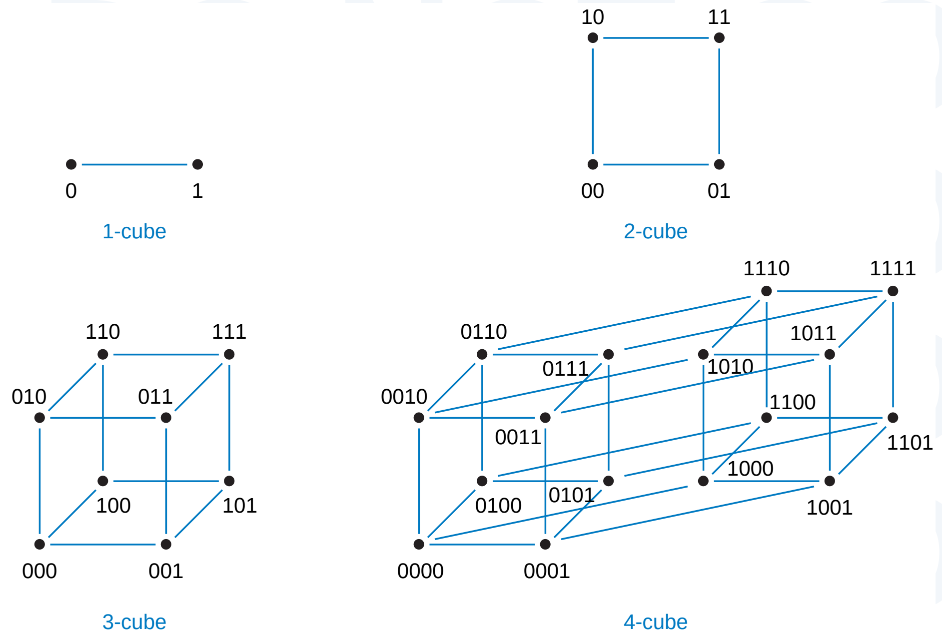


Figure 2-8
n-cubes for $n = 1, 2, 3$, and 4.

*2.14 *n*-Cubes and Distance

n-cube

An n-bit string can be visualized geometrically, as a vertex of an object called an *n-cube*. Figure 2-8 shows n-cubes for $n = 1, 2, 3, 4$. An n-cube has 2^n vertices, each of which is labeled with an n-bit string. Edges are drawn so that each vertex is adjacent to n other vertices whose labels differ from the given vertex in only one bit. Beyond $n = 4$, n-cubes are really tough to draw.

For reasonable values of n, n-cubes make it easy to visualize certain coding and logic minimization problems. For example, the problem of designing an n-bit Gray code is equivalent to finding a path along the edges of an n-cube, a path that visits each vertex exactly once. The paths for 3- and 4-bit Gray codes are shown in Figure 2-9.

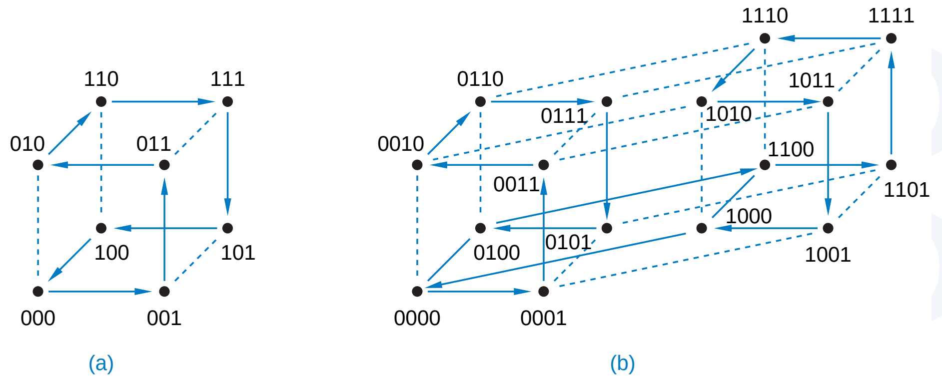


Figure 2-9
Traversing n-cubes in Gray-code order:
(a) 3-cube;
(b) 4-cube.

distance
Hamming distance

Cubes also provide a geometrical interpretation for the concept of *distance*, also called *Hamming distance*. The distance between two n-bit strings is the number of bit positions in which they differ. In terms of an n-cube, the distance is the minimum length of a path between the two corresponding vertices. Two adjacent vertices have distance 1; vertices 001 and 100 in the 3-cube have distance 2. The concept of distance is crucial in the design and understanding of error-detecting codes, discussed in the next section.

m-subcube

An *m-subcube* of an n-cube is a set of 2^m vertices in which $n - m$ of the bits have the same value at each vertex, and the remaining m bits take on all 2^m combinations. For example, the vertices (000, 010, 100, 110) form a 2-subcube of the 3-cube. This subcube can also be denoted by a single string, xx0, where "x" denotes that a particular bit is a *don't-care*; any vertex whose bits match in the non-x positions belongs to this subcube. The concept of subcubes is particularly useful in visualizing algorithms that minimize the cost of combinational logic functions, as we'll show in Section 4.4.

don't-care

*2.15 Codes for Detecting and Correcting Errors

error
failure
temporary failure
permanent failure

An *error* in a digital system is the corruption of data from its correct value to some other value. An error is caused by a physical *failure*. Failures can be either temporary or permanent. For example, a cosmic ray or alpha particle can cause a temporary failure of a memory circuit, changing the value of a bit stored in it. Letting a circuit get too hot or zapping it with static electricity can cause a permanent failure, so that it never works correctly again.

error model
independent error model
single error
multiple error

The effects of failures on data are predicted by *error models*. The simplest error model, which we consider here, is called the *independent error model*. In this model, a single physical failure is assumed to affect only a single bit of data; the corrupted data is said to contain a *single error*. Multiple failures may cause *multiple errors*—two or more bits in error—but multiple errors are normally assumed to be less likely than single errors.

2.15.1 Error-Detecting Codes

error-detecting code
noncode word

Recall from our definitions in Section 2.10 that a code that uses n-bit strings need not contain 2^n valid code words; this is certainly the case for the codes that we now consider. An *error-detecting code* has the property that corrupting or garbling a code word will likely produce a bit string that is not a code word (a *noncode word*).

A system that uses an error-detecting code generates, transmits, and stores only code words. Thus, errors in a bit string can be detected by a simple rule—if the bit string is a code word, it is assumed to be correct; if it is a noncode word, it contains an error.

An n-bit code and its error-detecting properties under the independent error model are easily explained in terms of an n-cube. A code is simply a subset

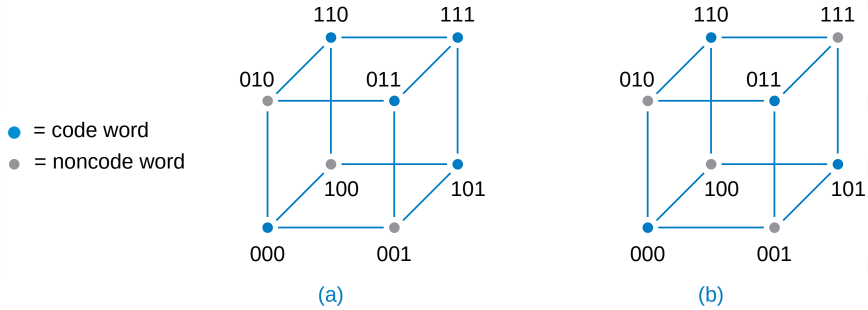


Figure 2-10 Code words in two different 3-bit codes: (a) minimum distance = 1, does not detect all single errors; (b) minimum distance = 2, detects all single errors.

of the vertices of the *n*-cube. In order for the code to detect all single errors, no code-word vertex can be immediately adjacent to another code-word vertex.

For example, Figure 2-10(a) shows a 3-bit code with five code words. Code word 111 is immediately adjacent to code words 110, 011 and 101. Since a single failure could change 111 to 110, 011 or 101 this code does not detect all single errors. If we make 111 a noncode word, we obtain a code that does have the single-error-detecting property, as shown in (b). No single error can change one code word into another.

The ability of a code to detect single errors can be stated in terms of the concept of distance introduced in the preceding section:

- A code detects all single errors if the *minimum distance* between all possible pairs of code words is 2.

minimum distance

In general, we need *n* + 1 bits to construct a single-error-detecting code with 2*n* code words. The first *n* bits of a code word, called *information bits*, may be any of the 2*n* *n*-bit strings. To obtain a minimum-distance-2 code, we add one more bit, called a *parity bit*, that is set to 0 if there are an even number of 1s among the information bits, and to 1 otherwise. This is illustrated in the first two columns of Table 2-13 for a code with three information bits. A valid *n*+1-bit code word has an even number of 1s, and this code is called an *even-parity code*.

information bit

parity bit

even-parity code

Table 2-13 Distance-2 codes with three information bits.

Information Bits	Even-parity Code	Odd-parity Code
000	000 0	000 1
001	001 1	001 0
010	010 1	010 0
011	011 0	011 1
100	100 1	100 0
101	101 0	101 1
110	110 0	110 1
111	111 1	111 0

odd-parity code
1-bit parity code

We can also construct a code in which the total number of 1s in a valid n+1-bit code word is odd; this is called an *odd-parity code* and is shown in the third column of the table. These codes are also sometimes called *1-bit parity codes*, since they each use a single parity bit.

The 1-bit parity codes do not detect 2-bit errors, since changing two bits does not affect the parity. However, the codes can detect errors in any *odd* number of bits. For example, if three bits in a code word are changed, then the resulting word has the wrong parity and is a noncode word. This doesn't help us much, though. Under the independent error model, 3-bit errors are much less likely than 2-bit errors, which are not detectable. Thus, practically speaking, the 1-bit parity codes' error detection capability stops after 1-bit errors. Other codes, with minimum distance greater than 2, can be used to detect multiple errors.

2.15.2 Error-Correcting and Multiple-Error-Detecting Codes

check bits

By using more than one parity bit, or *check bits*, according to some well-chosen rules, we can create a code whose minimum distance is greater than 2. Before showing how this can be done, let's look at how such a code can be used to correct single errors or detect multiple errors.

Suppose that a code has a minimum distance of 3. Figure 2-11 shows a fragment of the n-cube for such a code. As shown, there are at least two noncode words between each pair of code words. Now suppose we transmit code words

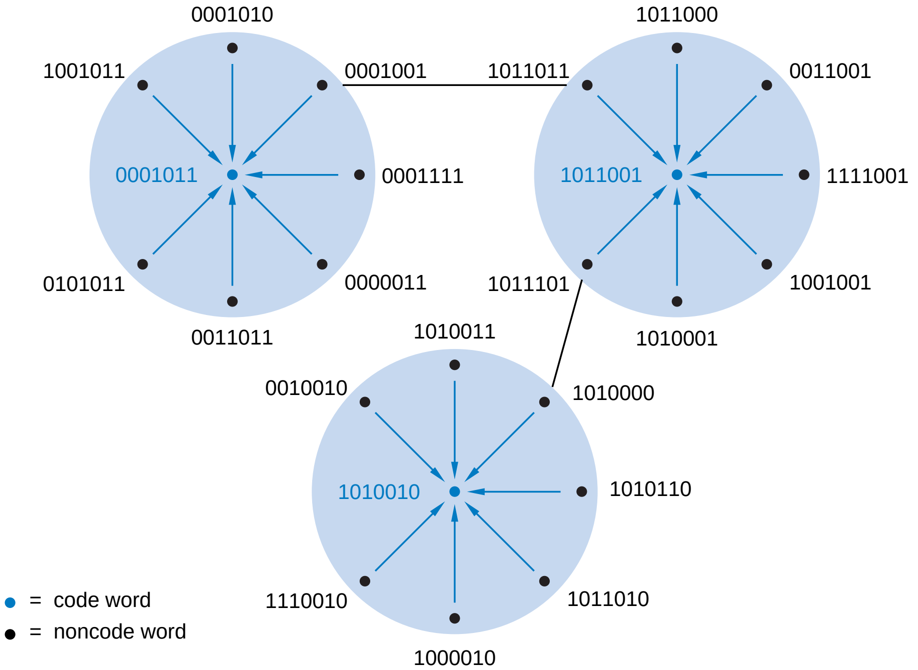


Figure 2-11
Some code words and noncode words in a 7-bit, distance-3 code.

and assume that failures affect at most one bit of each received code word. Then a received noncode word with a 1-bit error will be closer to the originally transmitted code word than to any other code word. Therefore, when we receive a noncode word, we can *correct* the error by changing the received noncode word to the nearest code word, as indicated by the arrows in the figure. Deciding which code word was originally transmitted to produce a received word is called *decoding*, and the hardware that does this is an error-correcting *decoder*.

error correction

decoding
decoder

A code that is used to correct errors is called an *error-correcting code*. In general, if a code has minimum distance $2c + 1$, it can be used to correct errors that affect up to c bits ($c = 1$ in the preceding example). If a code's minimum distance is $2c + d + 1$, it can be used to correct errors in up to c bits and to detect errors in up to d additional bits.

error-correcting code

For example, Figure 2-12(a) shows a fragment of the n-cube for a code with minimum distance 4 ($c = 1$, $d = 1$). Single-bit errors that produce noncode words 00101010 and 11010011 can be corrected. However, an error that produces 10100011 cannot be corrected, because no single-bit error can produce this noncode word, and either of two 2-bit errors could have produced it. So the code can detect a 2-bit error, but it cannot correct it.

When a noncode word is received, we don't know which code word was originally transmitted; we only know which code word is closest to what we've received. Thus, as shown in Figure 2-12(b), a 3-bit error may be "corrected" to the wrong value. The possibility of making this kind of mistake may be acceptable if 3-bit errors are very unlikely to occur. On the other hand, if we are concerned about 3-bit errors, we can change the decoding policy for the code. Instead of trying to correct errors, we just flag all noncode words as uncorrectable errors. Thus, as shown in (c), we can use the same distance-4 code to detect up to 3-bit errors but correct no errors ($c = 0$, $d = 3$).

2.15.3 Hamming Codes

In 1950, R. W. Hamming described a general method for constructing codes with a minimum distance of 3, now called *Hamming codes*. For any value of i, his method yields a 2^i–1-bit code with i check bits and $2^i - 1 - i$ information bits. Distance-3 codes with a smaller number of information bits are obtained by deleting information bits from a Hamming code with a larger number of bits.

Hamming code

The bit positions in a Hamming code word can be numbered from 1 through 2^i–1. In this case, any position whose number is a power of 2 is a check bit, and the remaining positions are information bits. Each check bit is grouped with a subset of the information bits, as specified by a *parity-check matrix*. As

parity-check matrix

DECISIONS, DECISIONS The names *decoding* and *decoder* make sense, since they are just distance-1 perturbations of *deciding* and *decider*.

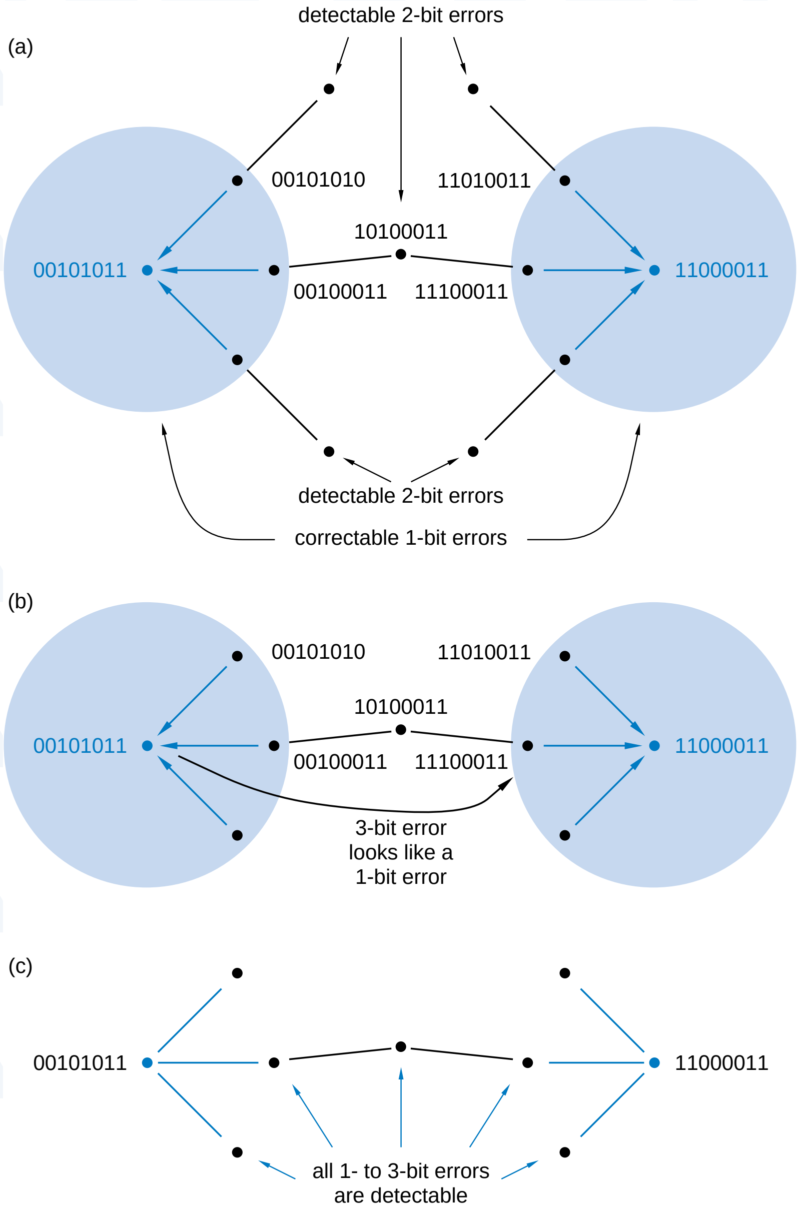


Figure 2-12
Some code words and noncode words in an 8-bit, distance-4 code:
(a) correcting 1-bit and detecting 2-bit errors;
(b) incorrectly "correcting" a 3-bit error;
(c) correcting no errors but detecting up to 3-bit errors.

shown in Figure 2-13(a), each check bit is grouped with the information positions whose numbers have a 1 in the same bit when expressed in binary. For example, check bit 2 (010) is grouped with information bits 3 (011), 6 (110), and 7 (111). For a given combination of information-bit values, each check bit is chosen to produce even parity, that is, so the total number of 1s in its group is even.

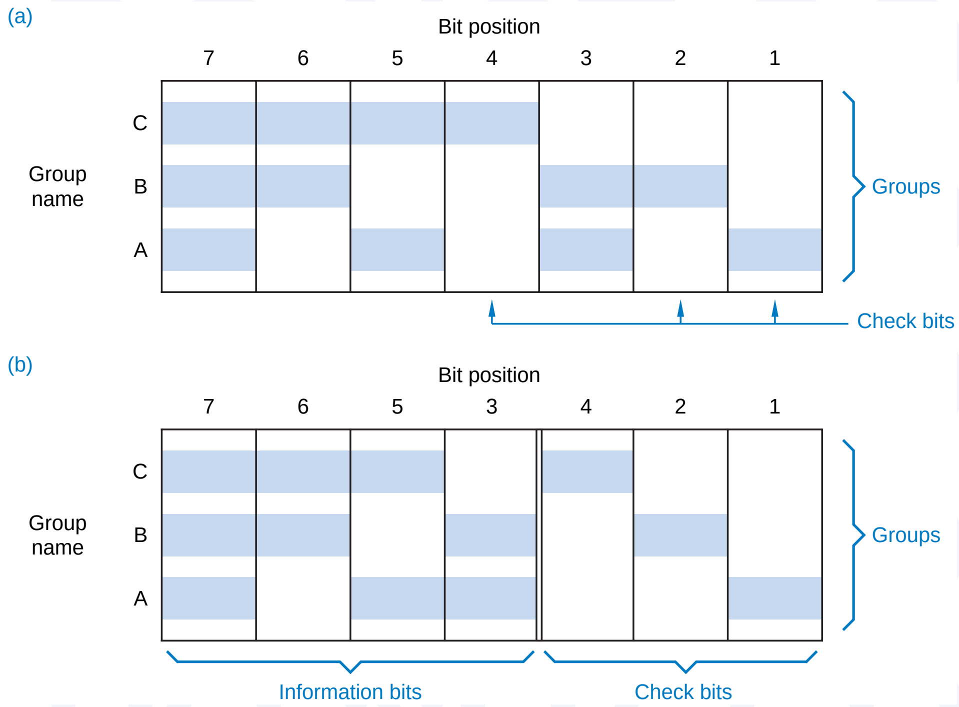


Figure 2-13 Parity-check matrices for 7-bit Hamming codes: (a) with bit positions in numerical order; (b) with check bits and information bits separated.

Traditionally, the bit positions of a parity-check matrix and the resulting code words are rearranged so that all of the check bits are on the right, as in Figure 2-13(b). The first two columns of Table 2-14 list the resulting code words.

We can prove that the minimum distance of a Hamming code is 3 by proving that at least a 3-bit change must be made to a code word to obtain another code word. That is, we'll prove that a 1-bit or 2-bit change in a code word yields a noncode word.

If we change one bit of a code word, in position *j*, then we change the parity of every group that contains position *j*. Since every information bit is contained in at least one group, at least one group has incorrect parity, and the result is a noncode word.

What happens if we change two bits, in positions *j* and *k*? Parity groups that contain both positions *j* and *k* will still have correct parity, since parity is unaffected when an even number of bits are changed. However, since *j* and *k* are different, their binary representations differ in at least one bit, corresponding to one of the parity groups. This group has only one bit changed, resulting in incorrect parity and a noncode word.

If you understand this proof, you should also see how the position numbering rules for constructing a Hamming code are a simple consequence of the proof. For the first part of the proof (1-bit errors), we required that the position numbers be nonzero. And for the second part (2-bit errors), we required that no

Table 2-14 Code words in distance-3 and distance-4 Hamming codes with four information bits.

Minimum-distance-3 code		*Minimum-distance-4 code*	
Information Bits	***Parity Bits***	***Information Bits***	***Parity Bits***
0000	000	0000	0000
0001	011	0001	0111
0010	101	0010	1011
0011	110	0011	1100
0100	110	0100	1101
0101	101	0101	1010
0110	011	0110	0110
0111	000	0111	0001
1000	111	1000	1110
1001	100	1001	1001
1010	010	1010	0101
1011	001	1011	0010
1100	001	1100	0011
1101	010	1101	0100
1110	100	1110	1000
1111	111	1111	1111

two positions have the same number. Thus, with an i-bit position number, you can construct a Hamming code with up to $2^i - 1$ bit positions.

error-correcting decoder

The proof also suggests how we can design an *error-correcting decoder* for a received Hamming code word. First, we check all of the parity groups; if all have even parity, then the received word is assumed to be correct. If one or more groups have odd parity, then a single error is assumed to have occurred. The pattern of groups that have odd parity (called the *syndrome*) must match one of the columns in the parity-check matrix; the corresponding bit position is assumed to contain the wrong value and is complemented. For example, using the code defined by Figure 2-13(b), suppose we receive the word 0101011. Groups B and C have odd parity, corresponding to position 6 of the parity-check matrix (the

syndrome

syndrome is 110, or 6). By complementing the bit in position 6 of the received word, we determine that the correct word is 0001011.

A distance-3 Hamming code can easily be modified to increase its minimum distance to 4. We simply add one more check bit, chosen so that the parity of all the bits, including the new one, is even. As in the 1-bit even-parity code, this bit ensures that all errors affecting an odd number of bits are detectable. In particular, any 3-bit error is detectable. We already showed that 1- and 2-bit errors are detected by the other parity bits, so the minimum distance of the modified code must be 4.

Distance-3 and distance-4 Hamming codes are commonly used to detect and correct errors in computer memory systems, especially in large mainframe computers where memory circuits account for the bulk of the system's failures. These codes are especially attractive for very wide memory words, since the required number of parity bits grows slowly with the width of the memory word, as shown in Table 2-15.

Table 2-15 Word sizes of distance-3 and distance-4 Hamming codes.

	Minimum-distance-3 Codes		*Minimum-distance-4 Codes*	
Information Bits	***Parity Bits***	***Total Bits***	***Parity Bits***	***Total Bits***
1	2	3	3	4
≤ 4	3	≤ 7	4	≤ 8
≤ 11	4	≤ 15	5	≤ 16
≤ 26	5	≤ 31	6	≤ 32
≤ 57	6	≤ 63	7	≤ 64
≤ 120	7	≤ 127	8	≤ 128

2.15.4 CRC Codes

Beyond Hamming codes, many other error-detecting and -correcting codes have been developed. The most important codes, which happen to include Hamming codes, are the *cyclic redundancy check (CRC) codes*. A rich set of knowledge has been developed for these codes, focused both on their error detecting and correcting properties and on the design of inexpensive encoders and decoders for them (see References).

cyclic redundancy check (CRC) code

Two important applications of CRC codes are in disk drives and in data networks. In a disk drive, each block of data (typically 512 bytes) is protected by a CRC code, so that errors within a block can be detected and, in some drives, corrected. In a data network, each packet of data ends with check bits in a CRC code. The CRC codes for both applications were selected because of their burst-error detecting properties. In addition to single-bit errors, they can detect multi-bit errors that are clustered together within the disk block or packet. Such errors are more likely than errors of randomly distributed bits, because of the likely physical causes of errors in the two applications—surface defects in disc drives and noise bursts in communication links.

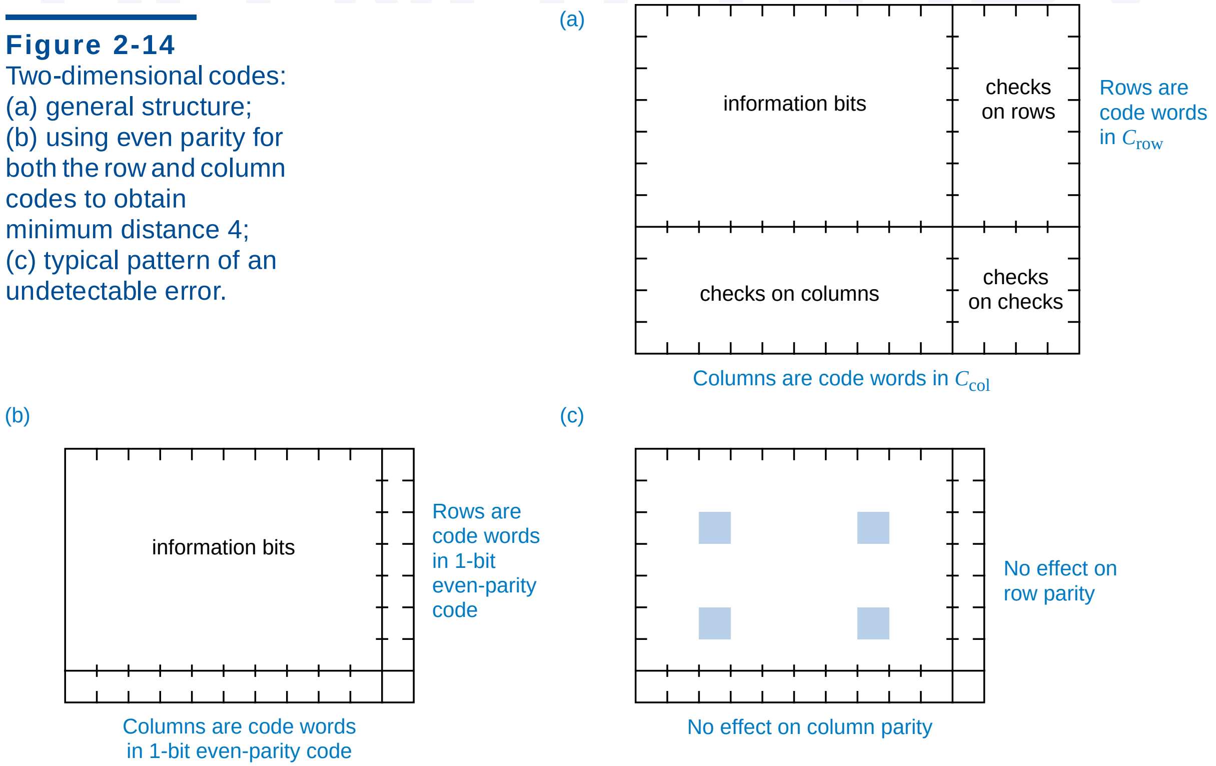
2.15.5 Two-Dimensional Codes

two-dimensional code

Another way to obtain a code with large minimum distance is to construct a *two-dimensional code*, as illustrated in Figure 2-14(a). The information bits are conceptually arranged in a two-dimensional array, and parity bits are provided to check both the rows and the columns. A code C_{row} with minimum distance d_{row} is used for the rows, and a possibly different code C_{col} with minimum distance d_{col} is used for the columns. That is, the row-parity bits are selected so that each row is a code word in C_{row} and the column-parity bits are selected so that each column is a code word in C_{col}. (The "corner" parity bits can be chosen according to either code.) The minimum distance of the two-dimensional code is the product of d_{row} and d_{col}; in fact, two-dimensional codes are sometimes called *product codes*.

product code

Figure 2-14
Two-dimensional codes:
(a) general structure;
(b) using even parity for both the row and column codes to obtain minimum distance 4;
(c) typical pattern of an undetectable error.



As shown in Figure 2-14(b), the simplest two-dimensional code uses 1-bit even-parity codes for the rows and columns, and has a minimum distance of $2 \cdot 2$, or 4. You can easily prove that the minimum distance is 4 by convincing yourself that any pattern of one, two, or three bits in error causes incorrect parity of a row or a column or both. In order to obtain an undetectable error, at least four bits must be changed in a rectangular pattern as in (c).

The error detecting and correcting procedures for this code are straightforward. Assume we are reading information one row at a time. As we read each row, we check its row code. If an error is detected in a row, we can't tell which bit is wrong from the row check alone. However, assuming only one row is bad, we can reconstruct it by forming the bit-by-bit Exclusive OR of the columns, omitting the bad row, but including the column-check row.

To obtain an even larger minimum distance, a distance-3 or -4 Hamming code can be used for the row or column code or both. It is also possible to construct a code in three or more dimensions, with minimum distance equal to the product of the minimum distances in each dimension.

RAID

An important application of two-dimensional codes is in RAID storage systems. *RAID* stands for "redundant array of inexpensive disks." In this scheme, $n+1$ identical disk drives are used to store n disks worth of data. For example, eight 8-Gigabyte drives could be use to store 64 Gigabytes of nonredundant data, and a ninth 8-gigabyte drive would be used to store checking information.

Figure 2-15 shows the general scheme of a two-dimensional code for a RAID system; each disk drive is considered to be a row in the code. Each drive stores m blocks of data, where a block typically contains 512 bytes. For example, an 8-gigabyte drive would store about 16 million blocks. As shown in the figure, each block includes its own check bits in a CRC code, to detect errors within that block. The first n drives store the nonredundant data. Each block in drive $n+1$

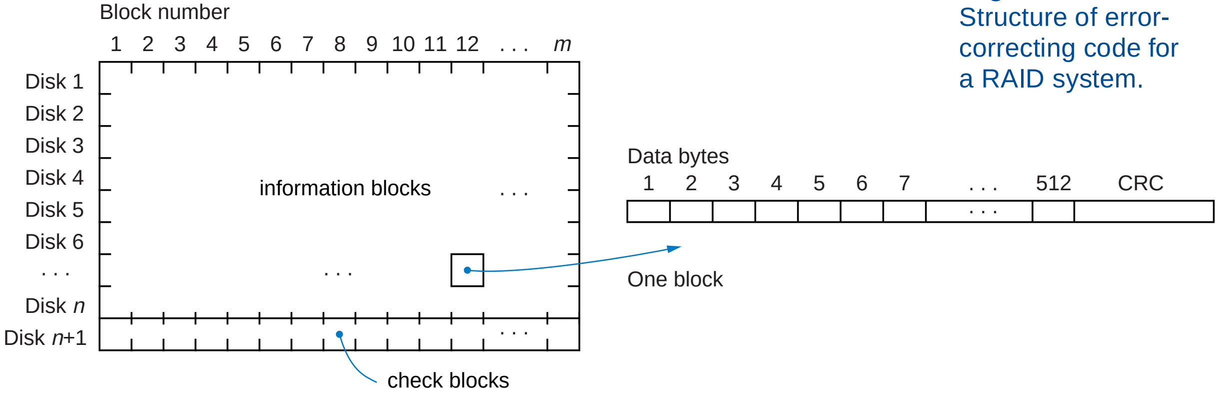


Figure 2-15
Structure of error-correcting code for a RAID system.

stores parity bits for the corresponding blocks in the first n drives. That is, each bit i in drive $n+1$ block b is chosen so that there are an even number of 1s in block b bit position i across all the drives.

In operation, errors in the information blocks are detected by the CRC code. Whenever an error is detected in a block on one of the drives, the correct contents of that block can be constructed simply by computing the parity of the corresponding blocks in all the other drives, including drive $n+1$. Although this requires n extra disk read operations, it's better than losing your data! Write operations require extra disk accesses as well, to update the corresponding check block when an information block is written (see Exercise 2.46). Since disk writes are much less frequent than reads in typical applications, this overhead usually is not a problem.

2.15.6 Checksum Codes

The parity-checking operation that we've used in the previous subsections is essentially modulo-2 addition of bits—the sum modulo 2 of a group of bits is 0 if the number of 1s in the group is even, and 1 if it is odd. The technique of modular addition can be extended to other bases besides 2 to form check digits.

For example, a computer stores information as a set of 8-bit bytes. Each byte may be considered to have a decimal value from 0 to 255. Therefore, we can use modulo-256 addition to check the bytes. We form a single check byte, called a *checksum*, that is the sum modulo 256 of all the information bytes. The resulting *checksum code* can detect any single *byte* error, since such an error will cause a recomputed sum of bytes to disagree with the checksum.

checksum
checksum code

Checksum codes can also use a different modulus of addition. In particular, checksum codes using modulo-255, ones'-complement addition are important because of their special computational and error detecting properties, and because they are used to check packet headers in the ubiquitous Internet Protocol (IP) (see References).

ones'-complement checksum code

2.15.7 *m*-out-of-*n* Codes

The 1-out-of-n and m-out-of-n codes that we introduced in Section 2.13 have a minimum distance of 2, since changing only one bit changes the total number of 1s in a code word and therefore produces a noncode word.

These codes have another useful error-detecting property—they detect unidirectional multiple errors. In a *unidirectional error*, all of the erroneous bits change in the same direction (0s change to 1s, or vice versa). This property is very useful in systems where the predominant error mechanism tends to change all bits in the same direction.

unidirectional error

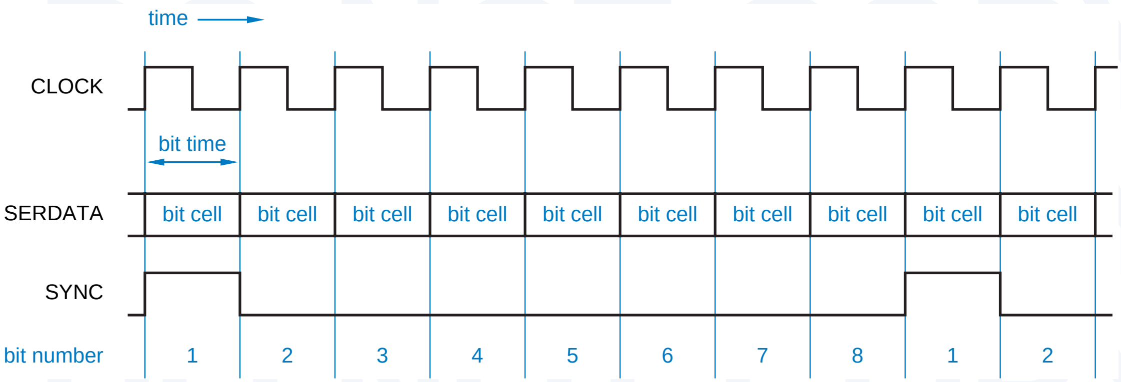


Figure 2-16 Basic concepts for serial data transmission.

2.16 Codes for Serial Data Transmission and Storage

2.16.1 Parallel and Serial Data

parallel data

Most computers and other digital systems transmit and store data in a *parallel* format. In parallel data transmission, a separate signal line is provided for each bit of a data word. In parallel data storage, all of the bits of a data word can be written or read simultaneously.

serial data

Parallel formats are not cost-effective for some applications. For example, parallel transmission of data bytes over the telephone network would require eight phone lines, and parallel storage of data bytes on a magnetic disk would require a disk drive with eight separate read/write heads. *Serial* formats allow data to be transmitted or stored one bit at a time, reducing system cost in many applications.

bit rate, bps

Figure 2-16 illustrates some of the basic ideas in serial data transmission. A repetitive clock signal, named CLOCK in the figure, defines the rate at which bits are transmitted, one bit per clock cycle. Thus, the *bit rate* in bits per second (bps) numerically equals the clock frequency in cycles per second (hertz, or Hz).

bit time

bit cell

line code

Non-Return-to-Zero (NRZ)

The reciprocal of the bit rate is called the *bit time* and numerically equals the clock period in seconds (s). This amount of time is reserved on the serial data line (named SERDATA in the figure) for each bit that is transmitted. The time occupied by each bit is sometimes called a *bit cell*. The format of the actual signal that appears on the line during each bit cell depends on the *line code*. In the simplest line code, called *Non-Return-to-Zero (NRZ)*, a 1 is transmitted by placing a 1 on the line for the entire bit cell, and a 0 is transmitted as a 0. However, more complex line codes have other rules, as discussed in the next subsection.

Regardless of the line code, a serial data transmission or storage system needs some way of identifying the significance of each bit in the serial stream. For example, suppose that 8-bit bytes are transmitted serially. How can we tell which is the first bit of each byte? A *synchronization signal*, named SYNC in Figure 2-16, provides the necessary information; it is 1 for the first bit of each byte.

synchronization signal

Evidently, we need a minimum of three signals to recover a serial data stream: a clock to define the bit cells, a synchronization signal to define the word boundaries, and the serial data itself. In some applications, like the interconnection of modules in a computer or telecommunications system, a separate wire is used for each of these signals, since reducing the number of wires per connection from *n* to three is savings enough. We'll give an example of a 3-wire serial data system in Section 8.5.4.

In many applications, the cost of having three separate signals is still too high (e.g., three phone lines, three read/write heads). Such systems typically combine all three signals into a single serial data stream and use sophisticated analog and digital circuits to recover the clock and synchronization information from the data stream.

*2.16.2 Serial Line Codes

The most commonly used line codes for serial data are illustrated in Figure 2-17. In the NRZ code, each bit value is sent on the line for the entire bit cell. This is the simplest and most reliable coding scheme for short distance transmission. However, it generally requires a clock signal to be sent along with the data to define the bit cells. Otherwise, it is not possible for the receiver to determine how many 0s or 1s are represented by a continuous 0 or 1 level. For example, without a clock to define the bit cells, the NRZ waveform in Figure 2-17 might be erroneously interpreted as 01010.

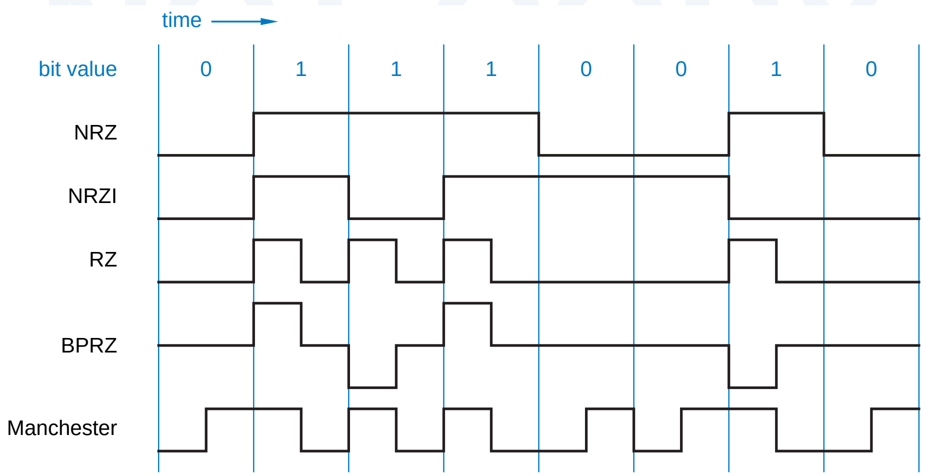


Figure 2-17 Commonly used line codes for serial data.

A *digital phase-locked loop (DPLL)* is an analog/digital circuit that can be used to recover a clock signal from a serial data stream. The DPLL works only if the serial data stream contains enough 0-to-1 and 1-to-0 transitions to give the DPLL "hints" about when the original clock transitions took place. With NRZ-coded data, the DPLL works only if the data does not contain any long, continuous streams of 1s or 0s.

digital phase-locked loop (DPLL)

Some serial transmission and storage media are *transition sensitive*; they cannot transmit or store absolute 0 or 1 levels, only transitions between two discrete levels. For example, a magnetic disk or tape stores information by changing the polarity of the medium's magnetization in regions corresponding to the stored bits. When the information is recovered, it is not feasible to determine the absolute magnetization polarity of a region, only that the polarity changes between one region and the next.

transition-sensitive media

Data stored in NRZ format on transition-sensitive media cannot be recovered unambiguously; the data in Figure 2-17 might be interpreted as 01110010 or 10001101. The *Non-Return-to-Zero Invert-on-1s (NRZI)* code overcomes this limitation by sending a 1 as the opposite of the level that was sent during the previous bit cell, and a 0 as the same level. A DPLL can recover the clock from NRZI-coded data as long as the data does not contain any long, continuous streams of 0s.

Non-Return-to-Zero Invert-on-1s (NRZI)

The Return-to-Zero (RZ) code is similar to NRZ except that, for a 1 bit, the 1 level is transmitted only for a fraction of the bit time, usually 1/2. With this code, data patterns that contain a lot of 1s create lots of transitions for a DPLL to use to recover the clock. However, as in the other line codes, a string of 0s has no transitions, and a long string of 0s makes clock recovery impossible.

Return-to-Zero (RZ)

Another requirement of some transmission media, such as high-speed fiber-optic links, is that the serial data stream be *DC balanced*. That is, it must have an equal number of 1s and 0s; any long-term DC component in the stream (created by have a lot more 1s than 0s or vice versa) creates a bias at the receiver that reduces its ability to distinguish reliably between 1s and 0s.

DC balance

Ordinarily, NRZ, NRZI or RZ data has no guarantee of DC balance; there's nothing to prevent a user data stream from having a long string of words with more than 1s than 0s or vice versa. However, DC balance can still be achieved using a few extra bits to code the user data in a *balanced code*, in which each code word has an equal number of 1s and 0s, and then sending these code words in NRZ format.

balanced code

For example, in Section 2.13 we introduced the 8B10B code, which codes 8 bits of user data into 10 bits in a mostly 5-out-of-10 code. Recall that there are only 252 5-out-of-10 code words, but there are another $\binom{4}{10} = 210$ 4-out-of-10 code words and an equal number of 6-out-of-10 code words. Of course, these code words aren't quite DC balanced. The 8B10B code solves this problem by associating with each 8-bit value to be encoded a *pair* of unbalanced code words, one 4-out-of-10 ("light") and the other 6-out-of-10 ("heavy"). The coder also

KILO-, MEGA-, GIGA-, TERA-

The prefixes K (kilo-), M (mega-), G (giga-), and T (tera-) mean 10^3, 10^6, 10^9, and 10^{12}, respectively, when referring to bps, hertz, ohms, watts, and most other engineering quantities. However, when referring to memory sizes, the prefixes mean 2^{10}, 2^{20}, 2^{30}, and 2^{40}. Historically, the prefixes were co-opted for this purpose because memory sizes are normally powers of 2, and 2^{10} (1024) is very close to 1000,

Now, when somebody offers you 50 kilobucks a year for your first engineering job, it's up to you to negotiate what the prefix means!

running disparity

keeps track of the *running disparity*, a single bit of information indicating whether the last unbalanced code word that it transmitted was heavy or light. When it comes time to transmit another unbalanced code word, the coder selects the one of the pair with the opposite weight. This simple trick makes available 252 + 210 = 462 code words for the 8B10B to encode 8 bits of user data. Some of the "extra" code words are used to conveniently encode non-data conditions on the serial line, such as IDLE, SYNC, and ERROR. Not all the unbalanced code words are used. Also, some of the balanced code words, such as 0000011111, are not used either, in favor of unbalanced pairs that contain more transitions.

Bipolar Return-to-Zero (BPRZ)

Alternate Mark Inversion (AMI)

All of the preceding codes transmit or store only two signal levels. The *Bipolar Return-to-Zero (BPRZ)* code transmits three signal levels: +1, 0, and −1. The code is like RZ except that 1s are alternately transmitted as +1 and −1; for this reason, the code is also known as *Alternate Mark Inversion (AMI)*.

The big advantage of BPRZ over RZ is that it's DC balanced. This makes it possible to send BPRZ streams over transmission media that cannot tolerate a DC component, such as transformer-coupled phone lines. In fact, the BPRZ code has been used in T1 digital telephone links for decades, where analog speech signals are carried as streams of 8000 8-bit digital samples per second that are transmitted in BPRZ format on 64 Kbps serial channels.

zero-code suppression

As with RZ, it is possible to recover a clock signal from a BPRZ stream as long as there aren't too many 0s in a row. Although TPC (The Phone Company) has no control over what you say (at least, not yet), they still have a simple way of limiting runs of 0s. If one of the 8-bit bytes that results from sampling your analog speech pattern is all 0s, they simply change second-least significant bit to 1! This is called *zero-code suppression* and I'll bet you never noticed it. And this is also why, in many data applications of T1 links, you get only 56 Kbps of usable data per 64 Kbps channel; the LSB of each byte is always set to 1 to prevent zero-code suppression from changing the other bits.

Manchester

diphase

The last code in Figure 2-17 is called *Manchester* or *diphase* code. The major strength of this code is that, regardless of the transmitted data pattern, it provides at least one transition per bit cell, making it very easy to recover the clock. As shown in the figure, a 0 is encoded as a 0-to-1 transition in the middle

ABOUT TPC Watch the 1967 James Coburn movie, *The President's Analyst*, for an amusing view of TPC. With the growing pervasiveness of digital technology and cheap wireless communications, the concept of universal, *personal* connectivity to the phone network presented in the movie's conclusion has become much less far-fetched.

of the bit cell, and a 1 is encoded as a 1-to-0 transition. The Manchester code's major strength is also its major weakness. Since it has more transitions per bit cell than other codes, it also requires more media bandwidth to transmit a given bit rate. Bandwidth is not a problem in coaxial cable, however, which was used in the original Ethernet local area networks to carry Manchester-coded serial data at the rate of 10 Mbps (megabits per second).

References

The presentation in the first nine sections of this chapter is based on Chapter 4 of *Microcomputer Architecture and Programming*, by John F. Wakerly (Wiley, 1981). Precise, thorough, and entertaining discussions of these topics can also be found in Donald E. Knuth's *Seminumerical Algorithms*, 3rd edition (Addison-Wesley, 1997). Mathematically inclined readers will find Knuth's analysis of the properties of number systems and arithmetic to be excellent, and all readers should enjoy the insights and history sprinkled throughout the text.

Descriptions of digital logic circuits for arithmetic operations, as well as an introduction to properties of various number systems, appear in *Computer Arithmetic* by Kai Hwang (Wiley, 1979). *Decimal Computation* by Hermann Schmid (Wiley, 1974) contains a thorough description of techniques for BCD arithmetic.

An introduction to algorithms for binary multiplication and division and to floating-point arithmetic appears in *Microcomputer Architecture and Programming: The 68000 Family* by John F. Wakerly (Wiley, 1989). A more thorough discussion of arithmetic techniques and floating-point number systems can be found in *Introduction to Arithmetic for Digital Systems Designers* by Shlomo Waser and Michael J. Flynn (Holt, Rinehart and Winston, 1982).

finite fields

CRC codes are based on the theory of *finite fields*, which was developed by French mathematician Évariste Galois (1811–1832) shortly before he was killed in a duel with a political opponent. The classic book on error-detecting and error-correcting codes is *Error-Correcting Codes* by W. W. Peterson and E. J. Weldon, Jr. (MIT Press, 1972, 2nd ed.); however, this book is recommended only for mathematically sophisticated readers. A more accessible introduction can be found in *Error Control Coding: Fundamentals and Applications* by S. Lin and D. J. Costello, Jr. (Prentice Hall, 1983). Another recent, communication-oriented introduction to coding theory can be found in *Error-Control*

Techniques for Digital Communication by A. M. Michelson and A. H. Levesque (Wiley-Interscience, 1985). Hardware applications of codes in computer systems are discussed in *Error-Detecting Codes, Self-Checking Circuits, and Applications* by John F. Wakerly (Elsevier/North-Holland, 1978).

As shown in the above reference by Wakerly, ones'-complement checksum codes have the ability to detect long bursts of unidirectional errors; this is useful in communication channels where errors all tend to be in the same direction. The special computational properties of these codes also make them quite amenable to efficient checksum calculation by software programs, important for their use in the Internet Protocol; see RFC-1071 and RFC-1141.

An introduction to coding techniques for serial data transmission, including mathematical analysis of the performance and bandwidth requirements of several codes, appears in *Introduction to Communications Engineering* by R. M. Gagliardi (Wiley-Interscience, 1988, 2nd ed.). A nice introduction to the serial codes used in magnetic disks and tapes is given in *Computer Storage Systems and Technology* by Richard Matick (Wiley-Interscience, 1977).

The structure of the 8B10B code and the rationale behind it is explained nicely in the original IBM patent by Peter Franaszek and Albert Widmer, U.S. patent number 4,486,739 (1984). This and almost all U.S. patents issued after 1971 can be found on the web at `www.patents.ibm.com`. When you're done reading Franaszek, for a good time do a boolean search for inventor "`wakerly`".

Drill Problems

2.1 Perform the following number system conversions:

(a) $1101011_2 = ?_{16}$
(b) $174003_8 = ?_2$
(c) $10110111_2 = ?_{16}$
(d) $67.24_8 = ?_2$
(e) $10100.1101_2 = ?_{16}$
(f) $\text{F3A5}_{16} = ?_2$
(g) $11011001_2 = ?_8$
(h) $\text{AB3D}_{16} = ?_2$
(i) $101111.0111_2 = ?_8$
(j) $\text{15C.38}_{16} = ?_2$

2.2 Convert the following octal numbers into binary and hexadecimal:

(a) $1023_8 = ?_2 = ?_{16}$
(b) $761302_8 = ?_2 = ?_{16}$
(c) $163417_8 = ?_2 = ?_{16}$
(d) $552273_8 = ?_2 = ?_{16}$
(e) $5436.15_8 = ?_2 = ?_{16}$
(f) $13705.207_8 = ?_2 = ?_{16}$

2.3 Convert the following hexadecimal numbers into binary and octal:

(a) $1023_{16} = ?_2 = ?_8$
(b) $\text{7E6A}_{16} = ?_2 = ?_8$
(c) $\text{ABCD}_{16} = ?_2 = ?_8$
(d) $\text{C350}_{16} = ?_2 = ?_8$
(e) $\text{9E36.7A}_{16} = ?_2 = ?_8$
(f) $\text{DEAD.BEEF}_{16} = ?_2 = ?_8$

2.4 What are the octal values of the four 8-bit bytes in the 32-bit number with octal representation 12345670123_8?

2.5 Convert the following numbers into decimal:

(a) $1101011_2 = ?_{10}$ (b) $174003_8 = ?_{10}$
(c) $10110111_2 = ?_{10}$ (d) $67.24_8 = ?_{10}$
(e) $10100.1101_2 = ?_{10}$ (f) $F3A5_{16} = ?_{10}$
(g) $12010_3 = ?_{10}$ (h) $AB3D_{16} = ?_{10}$
(i) $7156_8 = ?_{10}$ (j) $15C.38_{16} = ?_{10}$

2.6 Perform the following number system conversions:

(a) $125_{10} = ?_2$ (b) $3489_{10} = ?_8$
(c) $209_{10} = ?_2$ (d) $9714_{10} = ?_8$
(e) $132_{10} = ?_2$ (f) $23851_{10} = ?_{16}$
(g) $727_{10} = ?_5$ (h) $57190_{10} = ?_{16}$
(i) $1435_{10} = ?_8$ (j) $65113_{10} = ?_{16}$

2.7 Add the following pairs of binary numbers, showing all carries:

(a) $\begin{array}{r} 110101 \\ +\ \ 11001 \\ \hline \end{array}$ (b) $\begin{array}{r} 101110 \\ +\ 100101 \\ \hline \end{array}$ (c) $\begin{array}{r} 11011101 \\ +\ \ 1100011 \\ \hline \end{array}$ (d) $\begin{array}{r} 1110010 \\ +\ 1101101 \\ \hline \end{array}$

2.8 Repeat Drill 2.7 using subtraction instead of addition, and showing borrows instead of carries.

2.9 Add the following pairs of octal numbers:

(a) $\begin{array}{r} 1372 \\ +\ 4631 \\ \hline \end{array}$ (b) $\begin{array}{r} 47135 \\ +\ \ 5125 \\ \hline \end{array}$ (c) $\begin{array}{r} 175214 \\ +\ 152405 \\ \hline \end{array}$ (d) $\begin{array}{r} 110321 \\ +\ 56573 \\ \hline \end{array}$

2.10 Add the following pairs of hexadecimal numbers:

(a) $\begin{array}{r} 1372 \\ +\ 4631 \\ \hline \end{array}$ (b) $\begin{array}{r} 4F1A5 \\ +\ B8D5 \\ \hline \end{array}$ (c) $\begin{array}{r} F35B \\ +\ 27E6 \\ \hline \end{array}$ (d) $\begin{array}{r} 1B90F \\ +\ C44E \\ \hline \end{array}$

2.11 Write the 8-bit signed-magnitude, two's-complement, and ones'-complement representations for each of these decimal numbers: +18, +115, +79, −49, −3, −100.

2.12 Indicate whether or not overflow occurs when adding the following 8-bit two's-complement numbers:

(a) $\begin{array}{r} 11010100 \\ +\ 10101011 \\ \hline \end{array}$ (b) $\begin{array}{r} 10111001 \\ +\ 11010110 \\ \hline \end{array}$ (c) $\begin{array}{r} 01011101 \\ +\ 00100001 \\ \hline \end{array}$ (d) $\begin{array}{r} 00100110 \\ +\ 01011010 \\ \hline \end{array}$

2.13 How many errors can be detected by a code with minimum distance d?

2.14 What is the minimum number of parity bits required to obtain a distance-4, two-dimensional code with n information bits?

Exercises

2.15 Here's a problem to whet your appetite. What is the hexadecimal equivalent of 61453_{10}?

2.16 Each of the following arithmetic operations is correct in at least one number system. Determine possible radices of the numbers in each operation.

(a) 1234 + 5432 = 6666 (b) 41 / 3 = 13
(c) 33/3 = 11 (d) 23+44+14+32 = 223
(e) 302/20 = 12.1 (f) $\sqrt{41}$ = 5

2.17 The first expedition to Mars found only the ruins of a civilization. From the artifacts and pictures, the explorers deduced that the creatures who produced this civilization were four-legged beings with a tentacle that branched out at the end with a number of grasping "fingers." After much study, the explorers were able to translate Martian mathematics. They found the following equation:

$$5x^2 - 50x + 125 = 0$$

with the indicated solutions $x = 5$ and $x = 8$. The value $x = 5$ seemed legitimate enough, but $x = 8$ required some explanation. Then the explorers reflected on the way in which Earth's number system developed, and found evidence that the Martian system had a similar history. How many fingers would you say the Martians had? (From *The Bent of Tau Beta Pi*, February, 1956.)

2.18 Suppose a $4n$-bit number B is represented by an n-digit hexadecimal number H. Prove that the two's complement of B is represented by the 16's complement of H. Make and prove true a similar statement for octal representation.

2.19 Repeat Exercise 2.18 using the ones' complement of B and the 15s' complement of H.

2.20 Given an integer x in the range $-2n^{-1} \le x \le 2n^{-1} - 1$, we define $[x]$ to be the two's-complement representation of x, expressed as a positive number: $[x] = x$ if $x \ge 0$ and $[x] = 2n - |x|$ if $x < 0$, where $|x|$ is the absolute value of x. Let y be another integer in the same range as x. Prove that the two's-complement addition rules given in Section 2.6 are correct by proving that the following equation is always true:

$$[x + y] = ([x] + [y]) \text{ modulo } 2^n$$

(*Hints:* Consider four cases based on the signs of x and y. Without loss of generality, you may assume that $|x| \ge |y|$.)

2.21 Repeat Exercise 2.20 using appropriate expressions and rules for ones'-complement addition.

2.22 State an overflow rule for addition of two's-complement numbers in terms of counting operations in the modular representation of Figure 2-3.

2.23 Show that a two's-complement number can be converted to a representation with more bits by *sign extension*. That is, given an n-bit two's-complement number X, show that the m-bit two's-complement representation of X, where $m > n$, can be obtained by appending $m - n$ copies of X's sign bit to the left of the n-bit representation of X.

2.24 Show that a two's-complement number can be converted to a representation with fewer bits by removing higher-order bits. That is, given an n-bit two's-complement number X, show that the m-bit two's-complement number Y obtained by discarding the d leftmost bits of X represents the same number as X if and only if the discarded bits all equal the sign bit of Y.

2.25 Why is the punctuation of "two's complement" and "ones' complement" inconsistent? (See the first two citations in the References.)

2.26 A n-bit binary adder can be used to perform an n-bit unsigned subtraction operation $X - Y$, by performing the operation $X + Y + 1$, where X and Y are n-bit unsigned numbers and Y represents the bit-by-bit complement of Y. Demonstrate this fact as follows. First, prove that $(X - Y) = (X + Y + 1) - 2^n$. Second, prove that the carry out of the n-bit adder is the opposite of the borrow from the n-bit subtraction. That is, show that the operation $X - Y$ produces a borrow out of the MSB position if and only if the operation $X + Y + 1$ *does not* produce a carry out of the MSB position.

2.27 In most cases, the product of two n-bit two's-complement numbers requires fewer than $2n$ bits to represent it. In fact, there is only one case in which $2n$ bits are needed—find it.

2.28 Prove that a two's-complement number can be multiplied by 2 by shifting it one bit position to the left, with a carry of 0 into the least significant bit position and disregarding any carry out of the most significant bit position, assuming no overflow. State the rule for detecting overflow.

2.29 State and prove correct a technique similar to the one described in Exercise 2.28, for multiplying a ones'-complement number by 2.

2.30 Show how to subtract BCD numbers, by stating the rules for generating borrows and applying a correction factor. Show how your rules apply to each of the following subtractions: 9 – 3, 5 – 7, 4 – 9, 1 – 8.

2.31 How many different 3-bit binary state encodings are possible for the traffic-light controller of Table 2-12?

2.32 List all of the "bad" boundaries in the mechanical encoding disc of Figure 2-5, where an incorrect position may be sensed.

2.33 As a function of n, how many "bad" boundaries are there in a mechanical encoding disc that uses an n-bit binary code?

2.34 On-board altitude transponders on commercial and private aircraft use Gray code to encode the altitude readings that are transmitted to air traffic controllers. Why?

2.35 An incandescent light bulb is stressed every time it is turned on, so in some applications the lifetime of the bulb is limited by the number of on/off cycles rather than the total time it is illuminated. Use your knowledge of codes to suggest a way to double the lifetime of 3-way bulbs in such applications.

2.36 As a function of n, how many different distinct subcubes of an n-cube are there?

2.37 Find a way to draw a 3-cube on a sheet of paper (or other two-dimensional object) so that none of the lines cross, or prove that it's impossible.

2.38 Repeat Exercise 2.37 for a 4-cube.

2.39 Write a formula that gives the number of m-subcubes of an n-cube for a specific value of m. (Your answer should be a function of n and m.)

2.40 Define parity groups for a distance-3 Hamming code with 11 information bits.

2.41 Write the code words of a Hamming code with one information bit.

2.42 Exhibit the pattern for a 3-bit error that is not detected if the "corner" parity bits are not included in the two-dimensional codes of Figure 2-14.

2.43 The *rate of a code* is the ratio of the number of information bits to the total number of bits in a code word. High rates, approaching 1, are desirable for efficient transmission of information. Construct a graph comparing the rates of distance-2 parity codes and distance-3 and -4 Hamming codes for up to 100 information bits.

2.44 Which type of distance-4 code has a higher rate—a two-dimensional code or a Hamming code? Support your answer with a table in the style of Table 2-15, including the rate as well as the number of parity and information bits of each code for up to 100 information bits.

2.45 Show how to construct a distance-6 code with four information bits. Write a list of its code words.

2.46 Describe the operations that must be performed in a RAID system to write new data into information block b in drive d, so the data can be recovered in the event of an error in block b in any drive. Minimize the number of disk accesses required.

2.47 In the style of Figure 2-17, draw the waveforms for the bit pattern 10101110 when sent serially using the NRZ, NRZI, RZ, BPRZ, and Manchester codes, assuming that the bits are transmitted in order from left to right.