CHAPTER 1

Computer Number Systems, Codes, and Digital Devices

Before starting our discussion of microprocessors and microcomputers, we need to make sure that some key concepts of the number systems, codes, and digital devices used in microcomputers are fresh in your mind. If the short summaries of these concepts in this chapter are not enough to refresh your memory, then you may want to consult some of the chapters in *Digital Circuits and Systems*, McGraw-Hill, 1989, before going on in this book.

OBJECTIVES

At the conclusion of this chapter you should be able to:

1. Convert numbers between the following codes: binary, hexadecimal, and BCD.
2. Define the terms *bit*, *nibble*, *byte*, *word*, *most significant bit*, and *least significant bit*.
3. Use a table to find the ASCII or EBCDIC code for a given alphanumeric character.
4. Perform addition and subtraction of binary, hexadecimal, and BCD numbers.
5. Describe the operation of gates, flip-flops, latches, registers, ROMs, PALs, dynamic RAMs, static RAMs, and buses.
6. Describe how an arithmetic logic unit can be instructed to perform arithmetic or logical operations on binary words.

COMPUTER NUMBER SYSTEMS AND CODES

Review of Decimal System

To understand the structure of the binary number system, the first step is to review the familiar decimal or base-10 number system. Here is a decimal number with the value of each place holder or digit expressed as a power of 10.

5	3	4	6	.	7	2
10^3	10^2	10^1	10^0		10^{-1}	10^{-2}

The digits in the decimal number 5346.72 thus tell you that you have 5 thousands, 3 hundreds, 4 tens, 6 ones, 7 tenths, and 2 hundredths. The number of symbols needed in any number system is equal to the base number. In the decimal number system, then, there are 10 symbols, 0 through 9. When the count in any digit position passes that of the highest-value symbol, the digit rolls back to 0 and the next higher digit is incremented by 1. A car odometer is a good example of this.

A number system can be built using powers of any number as place holders or digits, but some bases are more useful than others. It is difficult to build electronic circuits which can store and manipulate 10 different voltage levels but relatively easy to build circuits which can handle two levels. Therefore, a *binary*, or *base-2*, number system is used to represent numbers in digital systems.

The Binary Number System

Figure 1-1*a*, p. 2, shows the value of each digit in a binary number. Each binary digit represents a power of 2. A binary digit is often called a *bit*. Note that digits to the right of the *binary point* represent fractions used for numbers less than 1. The binary system uses only two symbols, zero (0) and one (1), so in binary you count as follows: 0, 1, 10, 11, 100, 101, 110, 111, 1000, etc. For reference, Figure 1-1*b* shows the powers of 2 from 2^1 to 2^{32}.

Binary numbers are often called *binary words* or just *words*. Binary words with certain numbers of bits have also acquired special names. A 4-bit binary word is called a *nibble*, and an 8-bit binary word is called a *byte*. A 16-bit binary word is often referred to just as a *word*, and a 32-bit binary word is referred to as a *doubleword*. The rightmost or *least significant bit* of a binary word is usually referred to as the LSB. The leftmost or *most significant bit* of a binary word is usually called the MSB.

To convert a binary number to its equivalent decimal number, multiply each digit times the decimal value of the digit and just add these up. The binary number 101, for example, represents: $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$,

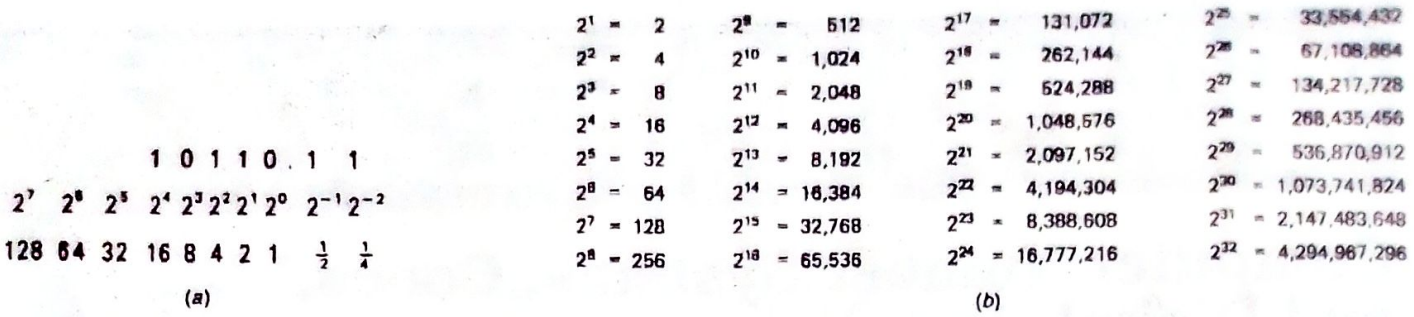


FIGURE 1-1 (*a*) Digit values in binary. (*b*) Powers of 2.

or 4 + 0 + 1 = decimal 5. For the binary number 10110.11, you have:

$$(1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2}) =$$
$$16 + 0 + 4 + 2 + 0 + 0.5 + 0.25 = \text{decimal } 22.75$$

To convert a decimal number to binary, there are two common methods. The first (Figure 1-2*a*) is simply a reverse of the binary-to-decimal method. For example, to convert the decimal number 21 (sometimes written as 21_{10}) to binary, first subtract the largest power of 2 that will fit in the number. For 21_{10} the largest power of 2 that will fit is 16 or 2^4. Subtracting 16 from 21 gives a remainder of 5. Put a 1 in the 2^4 digit position and see if the next lower power of 2 will fit in the remainder. Since 2^3 is 8 and 8 will not fit in the remainder of 5, put a 0 in the 2^3 digit position. Then try the next lower power of 2. In this case the next is 2^2 or 4, which will fit in the remainder of 5. A 1 is therefore put in the 2^2 digit position. When 2^2 or 4 is subtracted from the old remainder of 5, a new remainder of 1 is left. Since 2^1 or 2 will not fit into this remainder, a 0 is put in that position. A 1 is put in the 2^0 position because 2^0 is equal to 1 and this fits exactly into the remainder of 1. The result shows that 21_{10} is equal to 10101 in binary. This conversion process is somewhat messy to describe but easy to do. Try converting 46_{10} to binary. You should get 101110.

Another method of converting a decimal number to binary is shown in Figure 1-2*b*. Divide the decimal number by 2 and write the quotient and remainder as shown. Divide this quotient and following quotients by 2 until the quotient reaches 0. The column of remainders will be the binary equivalent of the given decimal number. Note that the MSD is on the bottom of the column and the LSD is on the top of the column if you perform the divisions in order from the top to the bottom of the page. You can demonstrate that the binary number is correct by reconverting from binary to decimal, as shown in the right-hand side of Figure 1-2*b*.

You can convert decimal numbers less than 1 to binary by successive multiplication by 2, recording carries until the quantity to the right of the decimal point becomes zero, as shown in Figure 1-2*c*. The carries represent the binary equivalent of the decimal number, with the *most significant bit* at the top of the column. Decimal 0.625 equals 0.101 in binary. For decimal values that do not convert exactly the way this one did (the quantity to the right of the decimal never becomes zero), you can continue the conversion process until you get the number of binary digits desired.

At this point it is interesting to compare the number of digits required to express numbers in decimal with the number required to express them in binary. In

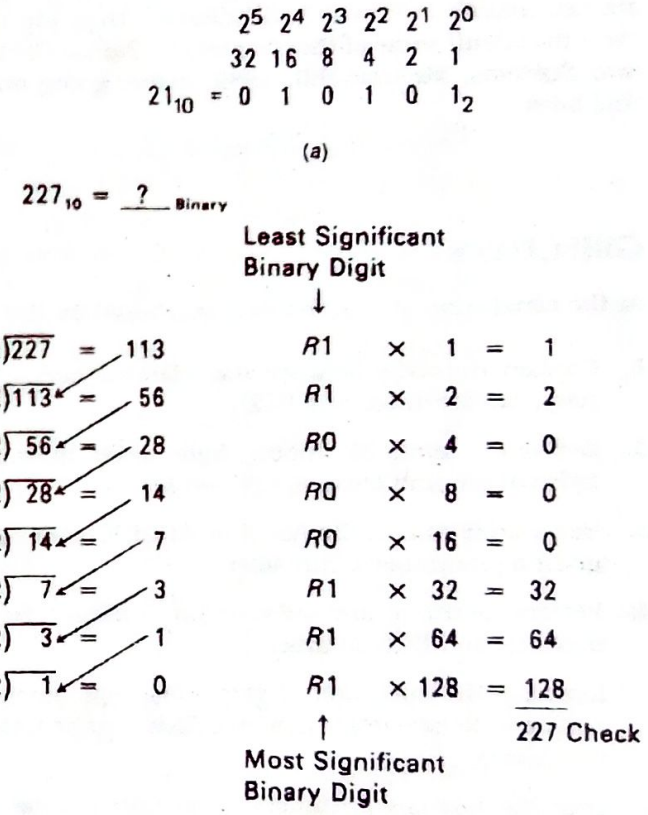


FIGURE 1-2 Converting decimal to binary. (*a*) Digit value method. (*b*) Divide by 2 method. (*c*) Decimal fraction conversion.

decimal, one digit can represent 10^1 numbers, 0 through 9; two digits can represent 10^2 or 100 numbers, 0 through 99; and three digits can represent 10^3 or 1000 numbers, 0 through 999. In binary, a similar pattern exists. One binary digit can represent 2 numbers, 0 and 1; two binary digits can represent 2^2 or 4 numbers, 0 through 11; and three binary digits can represent 2^3 or 8 numbers, 0 through 111. The pattern, then, is that N decimal digits can represent 10^N numbers and N binary digits can represent 2^N numbers. Eight binary digits can represent 2^8 or 256 numbers, 0 through 255 in decimal.

Hexadecimal

Binary is not a very compact code. This means that it requires many more digits to express a number than does, for example, decimal. Twelve binary digits can only describe a number up to 4095_{10}. Computers require binary data, but people working with computers have trouble remembering long binary words. One solution to the problem is to use the *hexadecimal* or base-16 number system.

Figure 1-3*a* shows the digit values for hexadecimal, which is often just called *hex*. Since hex is base 16, you have to have 16 possible symbols, one for each digit. The table of Figure 1-3*b* shows the symbols for hex code.

$$16^3 \quad 16^2 \quad 16^1 \quad 16^0 . \quad 16^{-1} \quad 16^{-2} \quad 16^{-3}$$
$$4096 \quad 256 \quad 16 \quad 1 \quad \frac{1}{16} \quad \frac{1}{256} \quad \frac{1}{4096}$$

(*a*)

Dec	Hex	Dec	Hex
0	0	8	8
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F

(*b*)

$227_D = \underline{\quad ? \quad}_{Hex}$ LSD

$16\overline{)227} = 14$ *R*3 × 1 = 3

$16\overline{)14} = 0$ *R*E × 16 = 224

MSD 227

$227_{10} = E3_{16}$

(*c*)

FIGURE 1-3 Hexadecimal numbers. (*a*) Value of place holders. (*b*) Symbols. (*c*) Decimal-to-hexadecimal conversion.

After the decimal symbols 0 through 9 are used up, you use the letters A through F for values 10 through 15.

As mentioned above, each hex digit is equal to four binary digits. To convert the binary number 11010110 to hex, mark off the binary bits in groups of 4, moving to the left from the binary point. Then write the hex symbol for the value of each group of 4.

Binary	1101	0110
Hex	D	6

The 0110 group is equal to 6 and the 1101 group is equal to 13. Since 13 is D in hex, 11010110 binary is equal to D6 in hex. "H" is usually used after a number to indicate that it is a hexadecimal number. For example, D6 hex is usually written D6H. As you can see, 8 bits can be represented with only 2 hex digits.

If you want to convert a number from decimal to hexadecimal, Figure 1-3*c* shows a familiar trick for doing this. The result shows that 227_{10} is equal to E3H. As you can see, hex is an even more compact code than decimal. Two hexadecimal digits can represent a decimal number up to 255. Four hex digits can represent a decimal number up to 65,535.

To illustrate how hexadecimal numbers are used in digital logic, a service manual tells you that the 8-bit-wide data bus of an 8088A microprocessor should contain 3FH during a certain operation. Converting 3FH to binary gives the pattern of 1's and 0's (0011 1111) you would expect to find with your oscilloscope or logic analyzer on the parallel lines. The 3FH is simply a shorthand which is easier to remember and less prone to errors than the binary equivalent.

BCD Codes

STANDARD BCD

In applications such as frequency counters, digital voltmeters, or calculators, where the output is a decimal display, a *binary-coded decimal* or *BCD* code is often used. BCD uses a 4-bit binary code to individually represent each decimal digit in a number. As you can see in Table 1-1, p. 4, the simplest BCD code uses the first 10 numbers of standard binary code for the BCD numbers 0 through 9. The hex codes A through F are invalid BCD codes. To convert a decimal number to its BCD equivalent, just represent each decimal digit by its 4-bit binary equivalent, as shown here.

Decimal	5	2	9
BCD	0101	0010	1001

To convert a BCD number to its decimal equivalent, reverse the process.

GRAY CODE

Gray code is another important binary code; it is often used for encoding shaft position data from machines such as computer-controlled lathes. This code has the same possible combinations as standard binary, but as you can see in the 4-bit example in Table 1-1, they are

TABLE 1-1
COMMON NUMBER CODES

Decimal	Binary	Octal	Hex	Binary-Coded Decimal		Reflected Gray Code	7-Segment Display (1 = on)	
				8421 BCD	EXCESS-3		a b c d e f g	Display
0	0000	0	0	0000	0011 0011	0000	1 1 1 1 1 1 0	0
1	0001	1	1	0001	0011 0100	0001	0 1 1 0 0 0 0	1
2	0010	2	2	0010	0011 0101	0011	1 1 0 1 1 0 1	2
3	0011	3	3	0011	0011 0110	0010	1 1 1 1 0 0 1	3
4	0100	4	4	0100	0011 0111	0110	0 1 1 0 0 1 1	4
5	0101	5	5	0101	0011 1000	0111	1 0 1 1 0 1 1	5
6	0110	6	6	0110	0011 1001	0101	1 0 1 1 1 1 1	6
7	0111	7	7	0111	0011 1010	0100	1 1 1 0 0 0 0	7
8	1000	10	8	1000	0011 1011	1100	1 1 1 1 1 1 1	8
9	1001	11	9	1001	0011 1100	1101	1 1 1 0 0 1 1	9
10	1010	12	A	0001 0000	0100 0011	1111	1 1 1 1 1 0 1	A
11	1011	13	B	0001 0001	0100 0100	1110	0 0 1 1 1 1 1	B
12	1100	14	C	0001 0010	0100 0101	1010	0 0 0 1 1 0 1	C
13	1101	15	D	0001 0011	0100 0110	1011	0 1 1 1 1 0 1	D
14	1110	16	E	0001 0100	0100 0111	1001	1 1 0 1 1 1 1	E
15	1111	17	F	0001 0101	0100 1000	1000	1 0 0 0 1 1 1	F

arranged in a different order. Notice that only one binary digit changes at a time as you count up in this code.

If you need to construct a Gray-code table larger than that in Table 1-1, a handy way to do so is to observe the pattern of 1's and 0's and just extend it. The least significant digit column starts with one 0 and then has alternating groups of two 1's and two 0's as you go down the column. The second most significant digit column starts with two 0's and then has alternating groups of four 1's and four 0's. The third column starts with four 0's, then has alternating groups of eight 1's and eight 0's. By now you should see the pattern. Try to figure out the Gray code for the decimal number 16. You should get 11000.

7-Segment Display Code

Figure 1-4*a* shows the segment identifiers for a 7-segment display such as those commonly used in digital instruments. Table 1-1 shows the logic levels required to display 0 to 9 and A to F on a common-cathode LED display such as that shown in Figure 1-4*b*. For a common-anode LED display such as that in Figure 1-4*c*, simply invert the segment codes shown in Table 1-1.

Alphanumeric Codes

When communicating with or between computers, you need a binary-based code which can represent letters of the alphabet as well as numbers. Common codes used for this have 7 or 8 bits per word and are referred to as *alphanumeric codes*. To detect possible errors in these codes, an additional bit, called a *parity bit*, is often added as the most significant bit.

Parity is a term used to identify whether a data word has an odd or even number of 1's. If a data word contains an odd number of 1's, the word is said to have *odd parity*. The binary word 0110111 with five 1's has odd parity. The binary word 0110000 has an even number of 1's (two), so it has *even parity*.

In practice the parity bit is used as follows. The system that is sending a data word checks the parity of the word. If the parity of the data word is odd, the system will set the parity bit to a 1. This makes the parity of the data word plus parity bit even. If the parity of the data word is even, the sending system will reset the parity bit to a 0. This again makes the parity of the data word plus parity even. The receiving system checks the

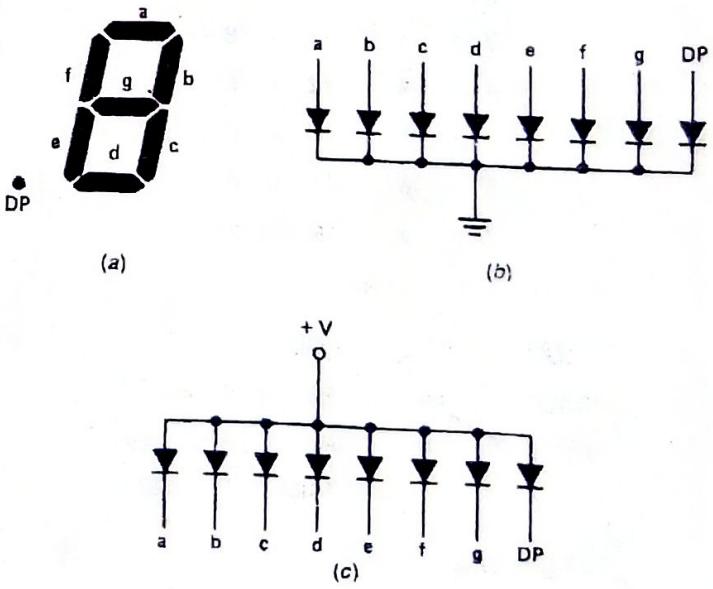


FIGURE 1-4 7-segment LED display. (*a*) Segment labels. (*b*) Schematic of common-cathode type. (*c*) Schematic of common-anode type.

parity of the data word plus parity bit that it receives. If the receiving system detects odd parity in the received data word plus parity, it assumes an error has occurred and tells the sending system to send the data again. The system is then said to be using even parity. The system could have been set up to use (maintain) odd parity in a similar manner.

ASCII

Table 1-2 shows several alphanumeric codes. The first of these is *ASCII*, or American Standard Code for Information Interchange. This is shown in the table as a 7-bit code. With 7 bits you can code up to 128 characters, which is enough for the full upper- and lowercase

TABLE 1-2
COMMON ALPHANUMERIC CODES

ASCII Symbol	*HEX Code for 7-Bit ASCII*	*EBCDIC Symbol*	*HEX Code for EBCDIC*	*ASCII Symbol*	*HEX Code for 7-Bit ASCII*	*EBCDIC Symbol*	*HEX Code for EBCDIC*	*ASCII Symbol*	*HEX Code for 7-Bit ASCII*	*EBCDIC Symbol*	*HEX Code for EBCDIC*
NUL	00	NUL	00	*	2A	*	5C	T	54	T	E3
SOH	01	SOH	01	+	2B	+	4E	U	55	U	E4
STX	02	STX	02	,	2C	,	6B	V	56	V	E5
ETX	03	ETX	03	-	2D	-	60	W	57	W	E6
EOT	04	EOT	37	.	2E	.	4B	X	58	X	E7
ENQ	05	ENQ	2D	/	2F	/	61	Y	59	Y	E8
ACK	06	ACK	2E	0	30	0	F0	Z	5A	Z	E9
BEL	07	BEL	2F	1	31	1	F1	[	5B	[	AD
BS	08	BS	16	2	32	2	F2	\	5C	NL	15
HT	09	HT	05	3	33	3	F3	]	5D	]	DD
LF	0A	LF	25	4	34	4	F4	^	5E	¬	5F
VT	0B	VT	0B	5	35	5	F5	_	5F	—	6D
FF	0C	FF	0C	6	36	6	F6	`	60	RES	14
CR	0D	CR	0D	7	37	7	F7	a	61	a	81
SO	0E	SO	0E	8	38	8	F8	b	62	b	82
SI	0F	SI	0F	9	39	9	F9	c	63	c	83
DLE	10	DLE	10	:	3A	:	7A	d	64	d	84
DC1	11	DC1	11	;	3B	;	5E	e	65	e	85
DC2	12	DC2	12	<	3C	<	4C	f	66	f	86
DC3	13	DC3	13	=	3D	=	7E	g	67	g	87
DC4	14	DC4	35	>	3E	>	6E	h	68	h	88
NAK	15	NAK	3D	?	3F	?	6F	i	69	i	89
SYN	16	SYN	32	@	40	@	7C	j	6A	j	91
ETB	17	EOB	26	A	41	A	C1	k	6B	k	92
CAN	18	CAN	18	B	42	B	C2	l	6C	l	93
EM	19	EM	19	C	43	C	C3	m	6D	m	94
SUB	1A	SUB	3F	D	44	D	C4	n	6E	n	95
ESC	1B	BYP	24	E	45	E	C5	o	6F	o	96
FS	1C	FLS	1C	F	46	F	C6	p	70	p	97
GS	1D	GS	1D	G	47	G	C7	q	71	q	98
RS	1E	RDS	1E	H	48	H	C8	r	72	r	99
US	1F	US	1F	I	49	I	C9	s	73	s	A2
SP	20	SP	40	J	4A	J	D1	t	74	t	A3
!	21	!	5A	K	4B	K	D2	u	75	u	A4
"	22	"	7F	L	4C	L	D3	v	76	v	A5
#	23	#	7B	M	4D	M	D4	w	77	w	A6
$	24	$	5B	N	4E	N	D5	x	78	x	A7
%	25	%	6C	O	4F	O	D6	y	79	y	A8
&	26	&	50	P	50	P	D7	z	7A	z	A9
'	27	'	7D	Q	51	Q	D8	{	7B	{	8B
(	28	(	4D	R	52	R	D9	|	7C	|	4F
)	29	)	5D	S	53	S	E2	}	7D	}	9B
								~	7E	¢	4A
								DEL	7F	DEL	07

TABLE 1-3
DEFINITIONS OF CONTROL CHARACTERS

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

alphabet, numbers, punctuation marks, and control characters. The code is arranged so that if only uppercase letters, numbers, and a few control characters are needed, the lower 6 bits are all that are required. If a parity check is wanted, a parity bit is added to the basic 7-bit code in the MSB position. The binary word 1100 0100, for example, is the ASCII code for uppercase D with odd parity. Table 1-3 gives the meanings of the control character symbols used in the ASCII code table.

EBCDIC

Another alphanumeric code commonly encountered in IBM equipment is the Extended Binary-Coded Decimal Interchange Code or *EBCDIC*. This is an 8-bit code without parity. A ninth bit can be added for parity. To save space in Table 1-2, the eight binary digits of EBCDIC are represented by their 2-digit hex equivalent.

ARITHMETIC OPERATIONS ON BINARY, HEX, AND BCD NUMBERS

Binary Arithmetic

ADDITION

Figure 1-5*a* shows the truth table for addition of two binary digits and a carry in (C_{IN}) from addition of previous digits. Figure 1-5*b* shows the result of adding two 8-bit binary numbers together using these rules. Assuming that C_{IN} = 1, 1 + 0 + C_{IN} = a sum of 0 and a carry into the next digit, and 1 + 1 + C_{IN} = a sum of 1 and a carry into the next digit because the result in any digit position can only be a 1 or a 0.

2'S-COMPLEMENT SIGNS-AND-MAGNITUDE BINARY

When you handwrite a number that represents some physical quantity such as temperature, you can simply put a + sign in front of the number to indicate that the number is positive, or you can write a − sign to indicate that the number is negative. However, if you want to store values such as temperatures, which can be positive or negative, in a computer memory, there is a problem. Since the computer memory can store only 1's and 0's, some way must be established to represent the sign of the number with a 1 or a 0.

A common way to represent signed numbers is to reserve the most significant bit of the data word as a *sign bit* and to use the rest of the bits of the data word to represent the size (magnitude) of the quantity. A computer that works with 8-bit words will use the MSB (bit 7) as the sign bit and the lower 7 bits to represent the magnitude of the numbers. The usual convention is to represent a positive number with a 0 sign bit and a negative number with a 1 sign bit.

To make computations with signed numbers easier, the magnitude of negative numbers is represented in a special form called *2's complement*. The 2's complement of a binary number is formed by inverting each bit of the data word and adding 1 to the result. Some examples should help clarify all of this.

The number $+7_{10}$ is represented in 8-bit sign-and-magnitude form as 00000111. The sign bit is 0, which indicates a positive number. The magnitude of positive numbers is represented in straight binary, so 00000111 in the least significant bits represents 7_{10}.

To represent -7_{10} in 8-bit 2's-complement sign-and-magnitude form, start with the 8-bit code for +7, 0000 0111. Invert each bit, including the MSB, to get 1111 1000. Then add 1 to get 11111001. This result is the correct representation of -7_{10}. Figure 1-6 shows some more examples of positive and negative numbers expressed in 8-bit sign-and-magnitude form. For practice, try generating each of these yourself to see if you get the same result.

To reverse this procedure and find the magnitude of a number expressed in sign-and-magnitude form, proceed as follows. If the number is positive, as indicated

INPUTS			OUTPUTS	
A	B	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(*a*)

$S = A \oplus B \oplus C_{IN}$

$C_{OUT} = A \cdot B + C_{IN}(A \oplus B)$

```
  10011010
+ 11011100
-----------
 [1] 01110110
  ↑
  └ Carry
```

(*b*)

FIGURE 1-5 Binary addition. (*a*) Truth table for 2 bits plus carry. (*b*) Addition of two 8-bit words.

	Sign bit ↓	
+ 7	0	0000111
+ 46	0	0101110
+105	0	1101001
− 12	1	1110100
− 54	1	1001010
−117	1	0001011
− 46	1	1010010

Sign and two's complement of magnitude

FIGURE 1-6 Positive and negative numbers represented with a sign bit and 2's complement.

by the sign bit being a 0, then the least significant 7 bits represent the magnitude directly in binary. If the number is negative, as indicated by the sign bit being a 1, then the magnitude is expressed in 2's complement. To get the magnitude of this negative number expressed in standard binary, invert each bit of the data word, including the sign bit, and add 1 to the result. For example, given the word 11101011, invert each bit to get 00010100. Then add 1 to get 00010101. This equals 21_{10}, so you know that the original numbers represent -21_{10}. Again, try reconverting a few of the numbers in Figure 1-6 for practice.

Figure 1-7 shows some examples of addition of signed binary numbers of this type. Sign bits are added together just as the other bits are. Figure 1-7*a* shows the results of adding two positive numbers. The sign bit of the result is zero, so the result is positive. The second example, in Figure 1-7*b*, adds a −9 to a +13 or, in effect, subtracts 9 from 13. As indicated by the zero sign bit, the result of 4 is positive and in true binary form.

Figure 1-7*c* shows the result of adding a −13 to a smaller positive number, +9. The sign bit of the result is a 1. This indicates that the result is negative and the magnitude is in 2's-complement form. To reconvert a 2's complement result to a signed number in true binary form:

1. Invert each bit to produce the 1's complement.
2. Add 1.
3. Put a minus sign in front to indicate that the result is negative.

The final example, in Figure 1-7*d*, shows the result of adding two negative numbers. The sign bit of the result is a 1, so the result is negative and in 2's-complement form. Again, inverting each bit, adding 1, and prefixing a minus sign will put the result in a more recognizable form.

Now let's consider the range of numbers that can be represented with 8 bits in sign-and-magnitude form. Eight bits can represent a maximum of 2^8 or 256 numbers. Since we are representing both positive and negative numbers, half of this range will be positive and half negative. Therefore, the range is −128 to +127. Here are the sign-and-magnitude binary representations for these values:

01111111	+127
⋮	
00000001	+1
00000000	zero
11111111	−1
⋮	
10000001	−127
10000000	−128

If you like number patterns, you might notice that this scheme shifts the normal codes for 128 to 255 downward to represent −128 to −1.

If a computer is storing signed numbers as 16-bit words, then a much larger range of numbers can be represented. Since 16 bits gives 2^{16} or 65,536 possible values, the range for 16-bit sign-and-magnitude numbers is −32,768 to +32,767. Operations with 16-bit sign-and-magnitude numbers are done the same way as operations with 8-bit sign-and-magnitude numbers.

```
+13     00001101
+ 9     00001001
+22     00010110
          └─Sign bit is 0
            so result is positive
```
(a)

```
+13     00001101
− 9     11110111  2's complement for −9 with sign bit
+ 4   1 00000100
        └─Sign bit is 0
          so result is positive
      └─Ignore carry
```
(b)

```
+ 9     00001001
−13     11110011  2's complement for −13 with sign bit
− 4     11111100  Sign bit is 1
        00000011  So invert each bit
      +         1 Add 1
equals ─00000100  Prefix with minus sign
```
(c)

```
− 9     11110111 } 2's complement,
−13     11110011 } sign-and-magnitude form
−22     11101010  Sign bit is 1
        00010101  So invert each bit
      +         1 Add 1
equals ─00010110  Prefix with minus sign
```
(d)

FIGURE 1-7 Addition of signed binary numbers. (*a*) +9 and +13. (*b*) −9 and +13. (*c*) +9 and −13. (*d*) −9 and −13.

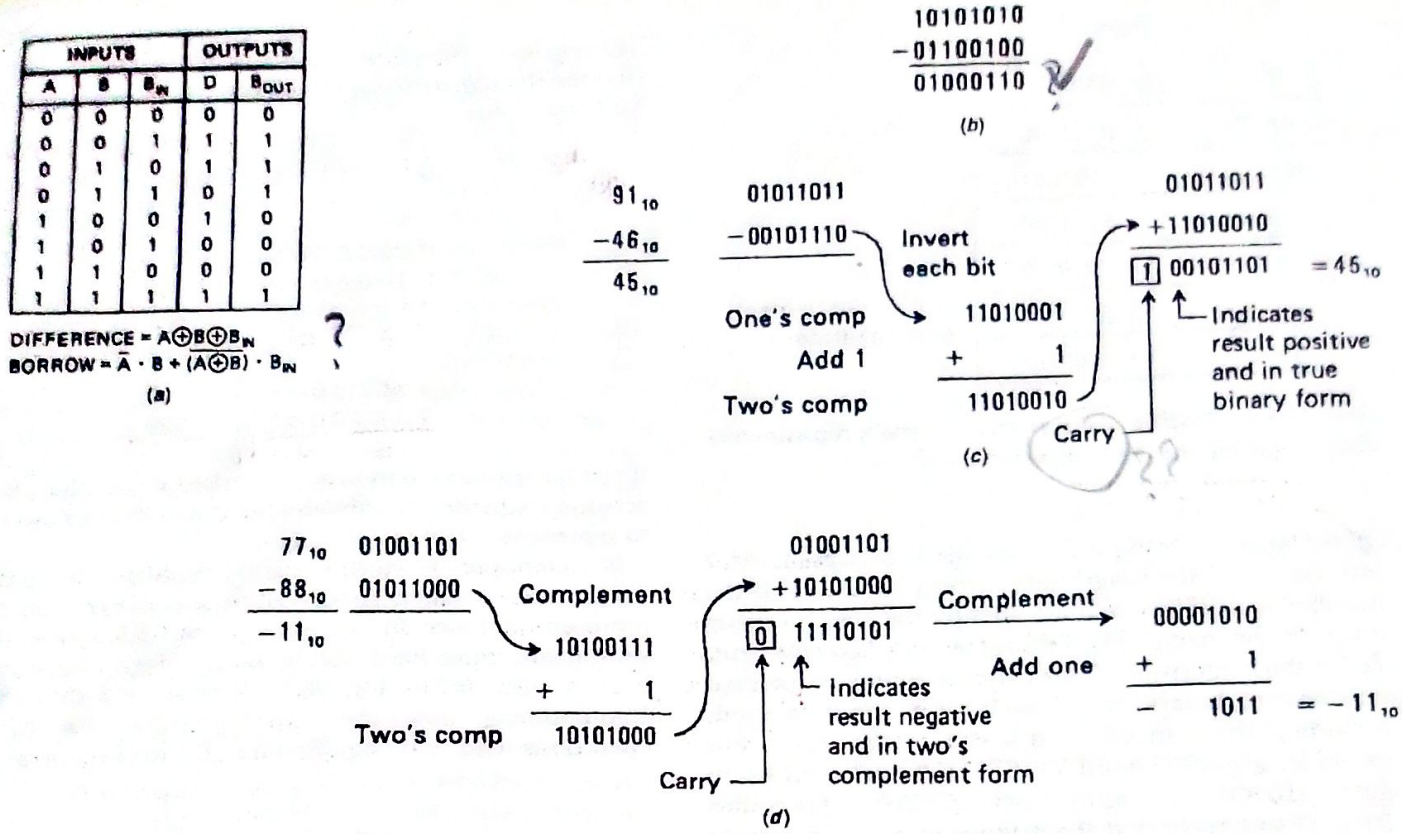


FIGURE 1-8 Binary subtraction. (a) Truth table for 2 bits and borrow. (b) Pencil method. (c) 2's-complement positive result. (d) 2's-complement negative result.

SUBTRACTION

There are two common methods for doing binary subtraction. These are the pencil method and the 2's-complement add method. Figure 1-8*a* shows the truth table for binary subtraction of two binary digits A and B. Also included in the truth table is the effect of a borrow-in, B_{IN}, from subtracting previous digits. Figure 1-8*b* shows an example of the "pencil" method of subtracting two 8-bit numbers. Using the truth table, this method is done the same way that you do decimal subtraction.

A second method of performing binary subtraction is by adding the 2's-complement representation of the bottom number (subtrahend) to the top number (minuend). Figure 1-8*c* shows how this is done. First represent the top number in sign-and-magnitude form. Then form the 2's-complement sign-and-magnitude representation for the negative of the bottom number. Finally, add the two parts formed. For the example in Figure 1-8*c*, the sign of the result is a 0, which indicates that the result is positive and in true form. The final carry produced by the addition can be ignored. Figure 1-8*d* shows another example of this method of subtraction. In this case the bottom number is larger than the top number. Again, represent the top number in sign-and-magnitude form, produce the 2's-complement sign-and-magnitude form for the negative of the bottom number, and add the two together. The sign bit of the result is a 1 for this example. This indicates that the result is negative and its magnitude is represented in 2's-complement form. To get the result into a form that is more recognizable to you, invert each bit of the result, add 1 to it, and put a minus sign in front of it as shown in Figure 1-8*d*.

Problems that may occur when doing signed addition or subtraction are *overflow* and *underflow*. If the magnitude of the number produced by adding two signed numbers is larger than the number of bits available to represent the magnitude, the result will "overflow" into the sign bit position and give an incorrect result. For example, if the signed positive number 01001001 is added to the signed positive number 01101101, the result is 10110110. The 1 in the MSB of this result indicates that it is negative, which is obviously incorrect for the sum of two positive numbers. In a similar manner, doing an 8-bit signed subtraction that produces a magnitude greater than −128 will cause an "underflow" into the sign bit and produce an incorrect result.

For simplicity the examples shown use 8 bits, but the method works for any number of bits. This method may seem awkward, but it is easy to do in a computer or microprocessor because it requires only the simple operations of inverting and adding.

MULTIPLICATION

There are several methods of doing binary multiplication. Figure 1-9 shows what is called the *pencil method* because it is the same way you learned to multiply decimal numbers. The top number, or multiplicand, is multiplied by the least significant digit of the bottom number, or multiplier. The partial product is written

```
  11        1011    MULTIPLICAND
×  9    ×   1001    MULTIPLIER
          ------
            1011 ⎫
           0000  ⎬  PARTIAL PRODUCTS
          0000   ⎪
         1011    ⎭
         -------
         1100011    PRODUCT
```

FIGURE 1-9 Binary multiplication.

down. The top number is then multiplied by the next digit of the multiplier. The resultant partial product is written down under the last, but shifted one place to the left. Adding all the partial products gives the total product. This method works well when doing multiplication by hand, but it is not practical for a computer because the type of shifts required makes it awkward to implement.

One of the multiplication methods used by computers is repeated addition. To multiply 7 × 55, for example, the computer can just add up seven 55's. For large numbers, however, this method is slow. To multiply 786 × 253, for example, requires 252 add operations. Most computers use an add-and-shift-right method. This method takes advantage of the fact that for binary multiplication, the partial product can only be either the top number exactly if the multiplier digit is a 1 or a 0 if the multiplier digit is a 0. The method does the same thing as the pencil method, except that the partial products are added as they are produced and the sum of the partial products is shifted right rather than each partial product being shifted left.

A point to note about multiplying numbers is the number of bits the product requires. For example, multiplying two 4-bit numbers can give a product with as many as 8 bits, and two 8-bit numbers can give a 16-bit product.

DIVISION

Binary division can also be performed in several ways. Figure 1-10 shows two examples of the pencil method. This is the same process as decimal long division. However, it is much simpler than decimal long division because the digits of the result (quotient) can only be 0 or 1. A division is attempted on part of the dividend. If this is not possible because the divisor is larger than that part of the dividend, a 0 is entered in the quotient. Another attempt is then made to divide using one more digit of the dividend. When a division is possible, a 1 is entered in the quotient. The divisor is then subtracted from the portion of the dividend used. As with standard long division, the process is continued until all the dividend is used. As shown in Figure 1-10*b*, 0's can be added to the right of the binary point and division continued to convert a remainder to a binary equivalent.

Another method of division that is easier for computers and microprocessors to perform uses successive subtractions. The divisor is subtracted from the dividend and from each successive remainder until a borrow is produced. The desired quotient is 1 less than the number of subtractions needed to produce a borrow. This method is simple, but for large numbers it is slow.

For faster division of large numbers, computers use a subtract-and-shift-left method that is essentially the same process you go through with a pencil long division.

```
               01100   QUOTIENT
DIVISOR  110 ) 1001000 DIVIDEND        12
              -110                  6 ) 72
               ----
                 110
                -110
                 ----
                   0
                  (a)

          110.01                      6.25
   100 ) 11001.00                 4 ) 25
        -100
         ---
          100
         -100
          ----
           0100
                  (b)
```

FIGURE 1-10 Binary division.

Hexadecimal Addition and Subtraction

People working with computers or microprocessors often use hexadecimal as a shorthand way of representing long binary numbers such as memory addresses. It is therefore useful to be able to add and subtract hexadecimal numbers.

ADDITION

As shown in Figure 1-11*a*, one way to add two hexadecimal numbers is to convert each hexadecimal number to its binary equivalent, add the two binary numbers, and convert the binary result back to its hex equivalent. For converting to binary, remember that each hex digit represents 4 binary digits.

A second method, shown in Figure 1-11*b*, works directly with the hex numbers. When adding hex digits, a carry is produced whenever the sum is 16 decimal or greater. Another way of saying this is that the value of a carry in hex is 16 decimal. For the least significant digits in Figure 1-11*b*, an A in hex is 10 in decimal and an F is 15 in decimal. These add to give 25 decimal. This is greater than 16, so mentally subtract 16 from the 25 to give a carry and a remainder of 9. The 9 is written down and the carry is added to the next digit column. In this column 7 plus 3 plus a carry gives a decimal 11, or B in hex.

```
                                    Carry
                                      ↓
 7A      0111  1010            7    1  A16
+3F     +0011  1111         +  3       F16
----    -----------         ---------------
 B9      1011  1001            11_10   25_10
            B     9            B16     9_16
            (a)                        (b)
```

FIGURE 1-11 Hexadecimal addition.

$$\begin{array}{rcl} 77_{16} & = & 119_{10} \\ -3B_{16} & = & -59_{10} \\ \hline 3C_{16} & & 60_{10} \end{array}$$

FIGURE 1-12 Hexadecimal subtraction.

You may use whichever method seems easier to you and gives you consistently right answers. If you are doing a great deal of hexadecimal arithmetic, you might buy an electronic calculator specifically designed to do decimal, binary, and hexadecimal arithmetic.

SUBTRACTION

Hexadecimal subtraction is similar to decimal subtraction except that when a borrow is needed, 16 is borrowed from the next most significant digit. Figure 1-12 shows an example of this. It may help you to follow the example if you do partial conversions to decimal in your head. For example, 7 plus a borrowed 16 is 23. Subtracting B or 11 leaves 12 or C in hexadecimal. Then 3 from the 6 left after a borrow leaves 3, so the result is 3CH.

BCD Addition and Subtraction

In systems where the final result of a calculation is to be displayed, such as a calculator, it may be easier to work with numbers in a BCD format. These codes, as shown in Table 1-1, represent each decimal digit, 0 through 9, by its 4-bit binary equivalent.

ADDITION

BCD can have no digit-word with a value greater than 9. Therefore, a carry must be generated if the result of a BCD addition is greater than 1001 or 9. Figure 1-13

```
                BCD
 35        0011 0101
+23       +0010 0011
----      ----------
 58        0101 1000
```

(a)

```
                 BCD
  7               0111
+ 5             + 0101
----            -------
 12              1100    INCORRECT BCD
                 + 0110  ADD 6
                -------
                0001 0010  CORRECT BCD 12
```

(b)

```
                 BCD
  9              1001
+ 8            + 1000
----           -------
 17             0001 0001  INCORRECT BCD
                0000 0110  ADD 6
                -------
                0001 0111  CORRECT BCD 17
```

(c)

FIGURE 1-13 BCD addition. (a) No correction needed. (b) Correction needed because of illegal BCD result. (c) Correction needed because of carry-out of BCD digit.

```
 17        0001 0111
 - 9       0000 1001
----       ---------
  8        0000 1110  ILLEGAL BCD
           -0110      SUBTRACT 6
           ---------
           0000 1000  CORRECT BCD
```

FIGURE 1-14 BCD subtraction.

shows three examples of BCD addition. The first, in Figure 1-13*a*, is very straightforward because the sum for each BCD digit is less than 9. The result is the same as it would be for adding standard binary.

For the second example, in Figure 1-13*b*, adding BCD 7 to BCD 5 produces 1100. This is a correct binary result of 12, but it is an illegal BCD code. To convert the result to BCD format, a correction factor of 6 is added. The result of adding 6 is 0001 0010, which is the legal BCD code for 12.

Figure 1-13*c* shows another case where a correction factor must be added. The initial addition of 9 and 8 produces 0001 0001. Even though the lower four digits are less than 9, this is an incorrect BCD result because a carry out of bit 3 of the BCD digit-word was produced. This carry out of bit 3 is often called an *auxiliary carry*. Adding the correction factor of 6 gives the correct BCD result of 0001 0111 or 17.

To summarize, a correction factor of 6 must be added if the result in the lower 4 bits is greater than 9 or if the initial addition produces a carry out of bit 3 of any BCD digit-word. This correction is sometimes called a *decimal adjust operation*.

The reason for the correction factor of 6 is that in BCD we want a carry into the next digit after 1001 or 9, but in binary a carry out of the lower 4 bits does not occur until after 1111 or 15. The difference between the two carry points is 6, so you have to add 6 to produce the desired carry if the result of an addition in any BCD digit is more than 1001.

SUBTRACTION

Figure 1-14 shows a subtraction, BCD 17 (0001 0111) minus BCD 9 (0000 1001). The initial result, 0000 1110, is not a legal BCD number. Whenever this occurs in BCD subtraction, 6 must be subtracted from the initial result to produce the correct BCD result. For the example shown in Figure 1-14, subtracting 6 gives a correct BCD result of 0000 1000 or 8.

The correction factor of 6 must be subtracted from any BCD digit-word if that digit-word is greater than 1001, or if a borrow from the next higher digit was required to do the subtraction.

BASIC DIGITAL DEVICES

Microcomputers such as those we discuss throughout this book often contain basic logic gates as "glue" between LSI (large-scale integration) devices. For troubleshooting these systems, it is important to be able to predict logic levels at any point directly from the schematic rather than having to work your way through a

truth table for each gate. This section should help refresh your memory of basic logic functions and help you remember how to quickly analyze logic gate circuits.

Inverting and Noninverting Buffers

Figure 1-15 shows the schematic symbols and truth tables for simple buffers and logic gates. The first thing to remember about these symbols is that the shape of the symbol indicates the logic function performed by the device. The second thing to remember about these symbols is that a bubble or no bubble indicates the *assertion* level for an input or output signal. Let's review how modern logic designers use these symbols.

The first symbol for a *buffer* in Figure 1-15*a* has no bubbles on the input or output. Therefore, the input is active high and the output is active high. We read this symbol as follows: If the input A is asserted high, then the output Y will be asserted high. The rest of the truth table is covered by the assumption that if the A input is not asserted high, then the Y output will not be asserted high.

The next two symbols for a buffer each contain a bubble. The bubble on the output of the first of these indicates that the output is active low. The input has no bubble, so it is active high. You can read the function of the device directly from the schematic symbol as follows. If the A input is asserted high, then the Y output will be asserted low. This device simply changes the assertion level of a signal. The output Y will always have a logic state which is the complement or inverse of that on the input, so the device is usually referred to as an *inverter*.

The second schematic symbol for an inverter in Figure 1-15*a* has the bubble on the input. We draw the symbol this way when we want to indicate that we are using the device to change an asserted-low signal to an asserted-high signal. For example, if we pass the signal $\overline{CS}$ through this device, it becomes CS. The symbol tells you directly that if the input is asserted low, then the output will be asserted high. Now let's review how you express the functions of logic gates using this approach.

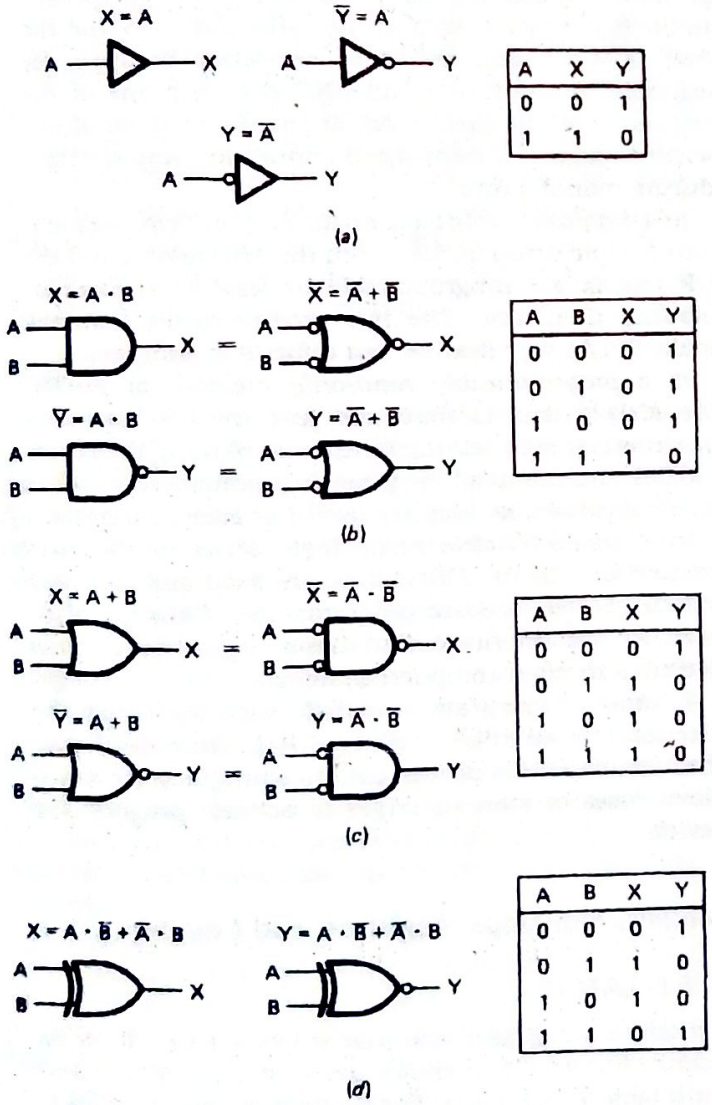


FIGURE 1-15 Buffers and logic gates. (*a*) Buffers. (*b*) AND-NAND. (*c*) OR-NOR. (*d*) Exclusive OR.

Logic Gates

Figure 1-15*b* shows the symbols and truth tables for simple logic gates. A symbol with a flat back and a round front indicates that the device performs the logical *AND* function. This means that the output will be asserted if the A input is asserted *and* the B input is asserted. Again, bubbles or no bubbles are used to indicate the assertion level of each input and output. The first AND symbol in Figure 1-15*b* has no bubbles, so the inputs and the output are active high. The output then will be asserted high if the A input is asserted high *and* the B input is asserted high. The bubble on the output of the second AND symbol in Figure 1-15*b* indicates that this device, commonly called a *NAND* gate, has an active low output. If the A input is asserted high *and* the B input is asserted high, then the Y output will be asserted low. Look at the truth table in Figure 1-15*b* to see if you agree with this.

Figure 1-15*c* shows the other two possible cases for the AND symbol. The first of these has bubbles on the inputs and on the output. If you see this symbol in a schematic, you should immediately see that the output will be asserted low if the A input is asserted low *and* the B input is asserted low. The second AND symbol in Figure 1-15*c* has no bubble on the output, so the output will be asserted high if the A *and* B inputs are both asserted low.

A logic symbol with a curved back indicates that the output of the device will be asserted if the A input is asserted *or* the B input of the device is asserted. Again, bubbles or no bubbles are used to indicate the assertion level for inputs and outputs. Note in Figure 1-15*b* and *c* that each of the AND symbol forms has an equivalent OR symbol form. An AND symbol with active high inputs and an active high output, for example, represents the same device (a 74LS08 perhaps) as an OR symbol with active low inputs and an active low output. Use the truth table in Figure 1-15*b* to convince yourself of this. The bubbled-OR representation tells you that if one input is asserted low, the output will be low, regardless of the state of the other input. As we will show later in this chapter, this is often a useful way to think of the operation of an AND gate.

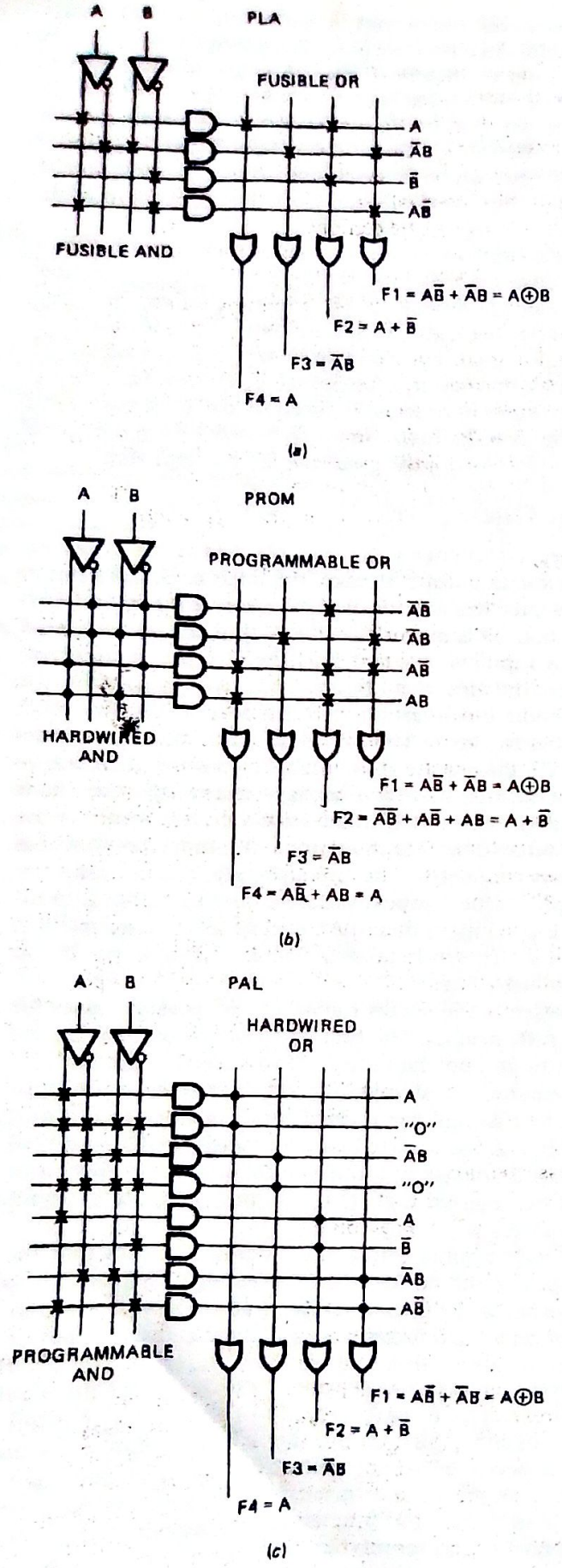


FIGURE 1-16 FPLA, PROM, and PAL programmed to implement some simple logic functions. (*a*) FPLA. (*b*) PROM. (*c*) PAL.

Figure 1-15*d* shows the symbol and truth table for an *exclusive* OR gate and for an *exclusive* NOR gate. The output of an exclusive OR gate will be high if the logic levels on the two inputs are different. The output of an exclusive NOR gate will be high if the logic levels on the two inputs are the same.

You need to be familiar with all these symbols, because most logic designers will use the symbol that best describes the function they want a device to perform in a particular circuit.

Programmable Logic Devices

Instead of using discrete gates, modern microcomputer systems usually use *programmable logic devices* such as PLAs, PROMs, or PALs to implement the "glue" logic between LSI devices. To refresh your memory, Figure 1-16 shows the internal structure of each of these devices. As you can see, they all consist of a programmable AND-OR matrix, so they can easily implement any sum-of-products logic expression. Each AND gate in these figures has up to four inputs, but to simplify the drawing only a single input line is shown. Likewise, the OR gates have several inputs, but are shown with a single input line to simplify the drawing. These devices are programmed by blowing out fuses, which are represented in the figure by Xs. An X in the figure indicates that the fuse is intact and makes a connection between, for example, the output of an AND gate and one of the inputs of an OR gate. A dot at the intersection of two wires indicates a hard-wired connection implemented during manufacture.

In a *programmable logic array* (*PLA*) or *field programmable logic array* (*FPLA*), both the AND matrix and the OR matrix are programmable by leaving in fuses or blowing them out. The two programmable matrixes make FPLAs very flexible, but difficult to program.

In a *programmable read-only memory* or *PROM*, the AND matrix is fixed and just the OR matrix is programmable by leaving in fuses or blowing them out. PROMs implement all the possible product terms for the input variables, so they are useful as code converters.

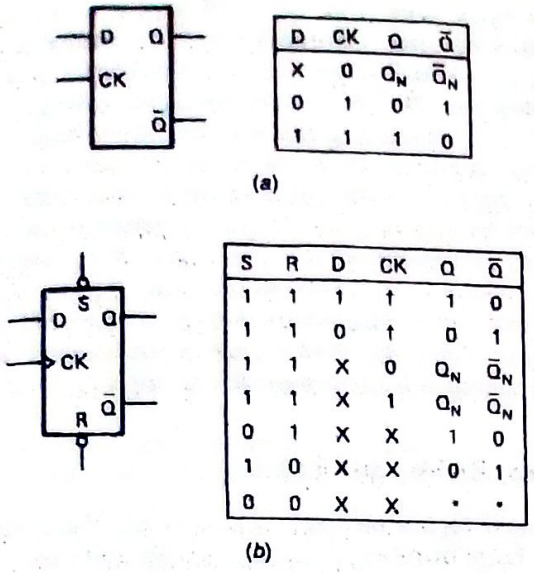
In a *programmable array logic* device or *PAL*, the connections in the OR matrix are fixed and the AND matrix connections are programmable. PALs are often used to implement combinational logic and address decoders in microcomputer systems.

A computer program is usually used to develop the fuse map for an FPLA, PROM, or PAL. Once developed, the fuse-map file is downloaded to a programmer which blows fuses or stores charges to actually program the device.

Latches, Flip-Flops, Registers, and Counters

THE D LATCH

A *latch* is a digital device that stores a 1 or a 0 on its output. Figure 1-17*a* shows the schematic symbol and truth table for a D latch. The device functions as follows. If the *enable* input CK is low, the logic level present on the D input will have no effect on the Q and $\bar{Q}$ outputs.



D	CK	Q	$\bar{Q}$
X	0	Q_N	$\bar{Q}_N$
0	1	0	1
1	1	1	0

S	R	D	CK	Q	$\bar{Q}$
1	1	1	↑	1	0
1	1	0	↑	0	1
1	1	X	0	Q_N	$\bar{Q}_N$
1	1	X	1	Q_N	$\bar{Q}_N$
0	1	X	X	1	0
1	0	X	X	0	1
0	0	X	X	*	*

FIGURE 1-17 Latches and flip-flops. (a) D latch. (*b*) D flip-flop.

This is indicated in the truth table by an X in the D column. If the enable input is high, a high or a low on the D input will be passed to the Q output. In other words, the Q output will follow the D input as long as the enable input is high. The $\bar{Q}$ output will contain the complement of the logic state on Q. When the enable input is made low again, the state on Q at that time will be latched there. Any changes on D will have no effect on Q until the enable input is made high again. When the enable input goes low, then, the state present on D just before the enable goes low will be stored on the Q output. Keep this operation in mind as you read about the D flip-flop in the next section.

THE D FLIP-FLOP

Figure 1-17*b* shows the schematic symbol and the truth table for a typical D flip-flop. The small triangle next to the CK input of this device tells you that the Q and $\bar{Q}$ outputs are updated when a rising signal edge is applied to the CK input. The up arrows in the clock column of the truth table also indicate that a 1 or 0 on the D input will be copied to the Q output when the clock input goes from low to high. In other words, the D flip-flop takes a snapshot of whatever state is on the D input when the clock goes high, and displays the "photo" on the Q output. If the clock input is low, a change on D will have no effect on the output. Likewise, if the clock input is high, a change on D will have no effect on the Q output. Contrast this operation with that of the D latch to make sure you understand the difference between the two devices.

The D flip-flop in Figure 1-17*b* also has direct *set* (S) and *reset* (R) inputs. A flip-flop is considered *set* if its Q output is a 1. It is *reset* if its Q output is a 0. The bubbles on the set and reset inputs tell you that these inputs are active low. The truth table for the D flip-flop in Figure 1-17*b* indicates that the set and reset inputs are *asynchronous*. This means that if the set input is asserted low, the output will be set, regardless of the states on the D and the clock inputs. Likewise, if the reset input is asserted low, the Q output will be reset, regardless of the state of the D and clock inputs. The Xs in the D and CK columns of the truth table remind you that these inputs are "don't cares" if set or reset is asserted. The condition indicated by the asterisks (*) is a nonstable condition; that is, it will not persist when reset or clear inputs return to their inactive (high) level.

REGISTERS

Flip-flops can be used individually or in groups to store binary data. A *register* is a group of D flip-flops connected in parallel, as shown in Figure 1-18*a*. A binary word applied to the data inputs of this register will be transferred to the Q outputs when the clock input is made high. The binary word will remain stored on the Q outputs until a new binary word is applied to the D inputs and a low-to-high signal is applied to the clock input. Other circuitry can read the stored binary word from the Q outputs at any time without changing its value.

If the Q output of each flip-flop in the register is connected to the D input of the next as shown in Figure 1-18*b*, then the register will function as a *shift register*. A 1 applied to the first D input will be shifted to the first Q output by a clock pulse. The next clock pulse will shift this 1 to the output of the second flip-flop. Each additional clock pulse will shift the 1 to the next flip-flop in the register. Some shift registers allow you to load a binary word into the register and shift the loaded word left or right when the register is clocked. As we will show later, the ability to shift binary numbers is very useful.

COUNTERS

Flip-flops can also be connected to make devices whose outputs step through a binary or other count sequence

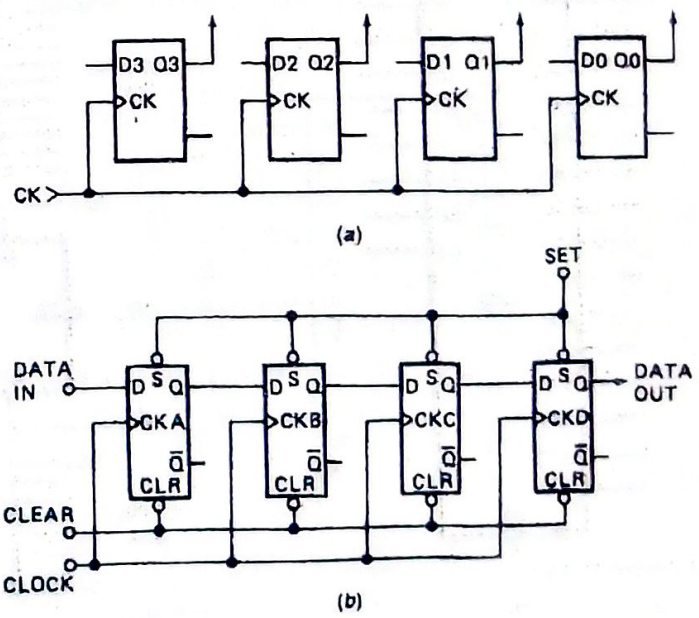
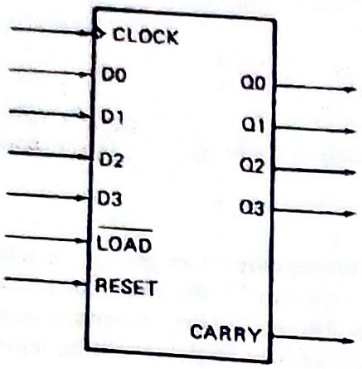


FIGURE 1-18 Registers. (a) Simple data storage. (*b*) Shift register.



(a)

Q3	Q2	Q1	Q0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

(b)

FIGURE 1-19 Four-bit, presettable binary counter. (a) Schematic symbol. (b) Count sequence.

when they are clocked. Figure 1-19*a* shows a schematic symbol and count sequence for a presettable 4-bit binary counter. The main point we want to review here is how a presettable counter functions, so there is no need to go into the internal circuitry of the device. If the reset input is asserted, the Q outputs will all be made 0's. After the reset signal is unasserted, each clock pulse will cause the binary count on the outputs to be incremented by 1. As shown in Figure 1-19*b*, the count sequence will go from 0000 to 1111. If the outputs are at 1111, then the next clock pulse will cause the outputs to "roll over" to 0000 and a carry pulse to be sent out the carry output.

This carry pulse can be used as the clock input for another counter. Counters can be cascaded to produce as large a count sequence as is needed for a particular application. The maximum count for a binary counter is $2^N - 1$, where N is the number of flip-flops.

Now, suppose that we want the counter to start counting from some number other than 0000. We can do this by applying the desired number to the four data inputs and asserting the load input. For example, if we apply a binary 6, 0110, to the data inputs and assert the load input, this value will be transferred to the Q outputs. After the load signal is unasserted, the next clock signal will increment the Q outputs to 0111 or 7.

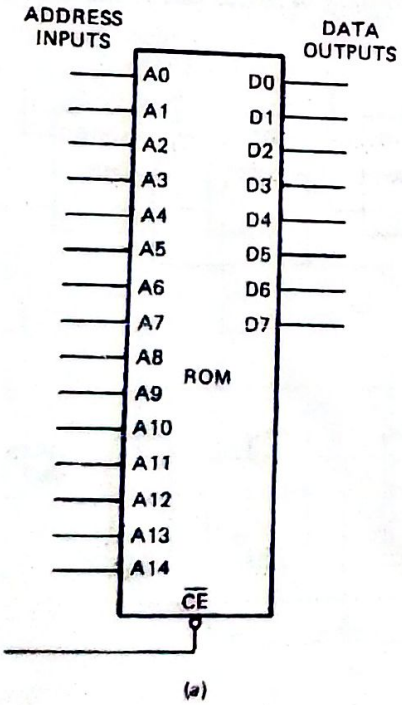
ROMs, RAMs, and Buses

The next topics we need to review are the devices that store large numbers of binary words and how several of these devices can be connected on common data lines.

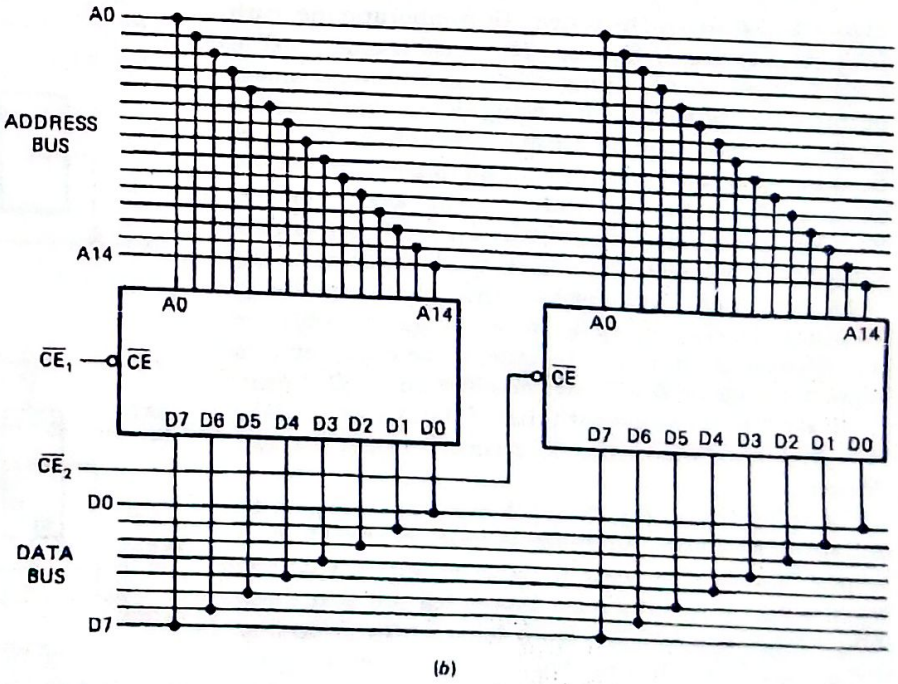
ROMS

The term *ROM* stands for *read-only memory*. There are several types of ROM that can be written to, read, erased, and written to with new data, but the main feature of ROMs is that they are *nonvolatile*. This means that the information stored in them is not lost when the power is removed from them.

Figure 1-20*a* shows the schematic symbol of a common ROM. As indicated by the eight *data* outputs, D0 to D7, this ROM stores 8-bit data words. The data outputs are *three-state* outputs. This means that each output can be at a logic low state, a logic high state, or a high-impedance floating state. In the high-impedance state an output is essentially disconnected from anything connected to it. If the $\overline{CE}$ input of the ROM is not asserted, then all the outputs will be in the high-



(a)



(b)

FIGURE 1-20 ROMs. (a) Schematic symbol. (b) Connection in parallel.

impedance state. Most ROMs also switch to a lower-power-consumption standby mode if $\overline{CE}$ is not asserted. If the $\overline{CE}$ input is asserted, the device will be powered up, and the output buffers will be enabled. Therefore, the outputs will be at a normal logic low or logic high state. If you don't happen to remember, you will soon see why this is important.

You can think of the binary words stored in the ROM as being in a long, numbered list. The number that identifies the location of each stored word in the list is called its *address*. You can tell the number of binary words stored in the ROM by the number of address inputs. The number of words is equal to 2^N, where N is the number of address lines. The device in Figure 1-20*a* has 15 address lines, A0 to A14, so the number of words is 2^{15} or 32,768. In a data sheet this device would be referred to as a 32K × 8 ROM. This means it has 32K addresses with 8 bits per address.

In order to get a particular word onto the outputs of the ROM, you have to do two things. You have to apply the address of that word to the address inputs, A0 to A14, and you have to assert the $\overline{CE}$ input to power up the device and to enable the three-state outputs.

Now, let's see why we want three-state outputs on this ROM. Suppose that we want to store more than 32K data words. We can do this by connecting two or more ROMs in parallel, as shown in Figure 1-20*b*. The address lines connect to each device in parallel, so we can address one of the 32,768 words in each. A set of parallel lines used to send addresses or data to several devices in this way is called a *bus*. The data outputs of the ROMs are likewise connected in parallel so that any one of the ROMs can output data on the common data bus. If these ROMs had standard two-state outputs, a serious problem would occur when both ROMs tried to output data words on the bus. The resulting argument between data outputs would probably destroy some of the outputs and give meaningless information on the data bus. Since the ROMs have three-state outputs, however, we can use external circuitry to make sure that only one ROM at a time has its outputs enabled. The very important principle here is that whenever several outputs are connected on a bus, the outputs should all be three-state, and only one set of outputs should be enabled at a time.

At the beginning of this section we mentioned that some ROMs can be erased and rewritten or reprogrammed with new data. Here's a summary of the different types of ROMs.

Mask-programmed ROM—Programmed during manufacture; cannot be altered.

PROM—User programs by blowing fuses; cannot be altered except to blow additional fuses.

EPROM—Electrically programmable by user; erased by shining ultraviolet light on quartz window in package.

EEPROM—Electrically programmable by user; erased with electrical signals, so it can be reprogrammed in circuit.

Flash EPROM—Electrically programmable by user; erased electrically, so it can be reprogrammed in circuit.

STATIC AND DYNAMIC RAMS

The name RAM stands for *random-access memory*, but since ROMs are also random access, the name probably should be *read-write memory*. RAMs are also used to store binary words. A *static RAM* is essentially a matrix of flip-flops. Therefore, we can write a new data word in a RAM location at any time by applying the word to the flip-flop data inputs and clocking the flip-flops. The stored data word will remain on the flip-flop outputs as long as the power is left on. This type of memory is *volatile* because data is lost when the power is turned off.

Figure 1-21 shows the schematic symbol for a common RAM. This RAM has 12 address lines, A0 to A11, so it stores 2^{12} (4096) binary words. The eight data lines tell you that the RAM stores 8-bit words. When we are reading a word from the RAM, these lines function as outputs. When we are writing a word to the RAM, these lines function as inputs. The *chip enable* input, $\overline{CE}$, is used to enable the device for a read or for a write. The R/$\overline{W}$ input will be asserted high if we want to read from the RAM or asserted low if we want to write a word to the RAM. Here's how all these lines work for reading from and writing to the device.

To write to the RAM, we apply the desired address to the address inputs, assert the $\overline{CE}$ input low to turn on the device, and assert the R/$\overline{W}$ input low to tell the RAM we want to write to it. We then apply the data word we want to store to the data lines of the RAM for a specified time. To read a word from the RAM, we address the desired word, assert $\overline{CE}$ low to turn on the device, and assert R/$\overline{W}$ high to tell the RAM we want to read from it. For a read operation the output buffers on the data lines will be enabled and the addressed data word will be present on the outputs.

The static RAMs we have just reviewed store binary words in a matrix of flip-flops. In *dynamic RAMs* (*DRAMs*), binary 1's and 0's are stored as an electric charge or no charge on a tiny capacitor. Since these tiny capacitors take up less space on a chip than a flip-flop

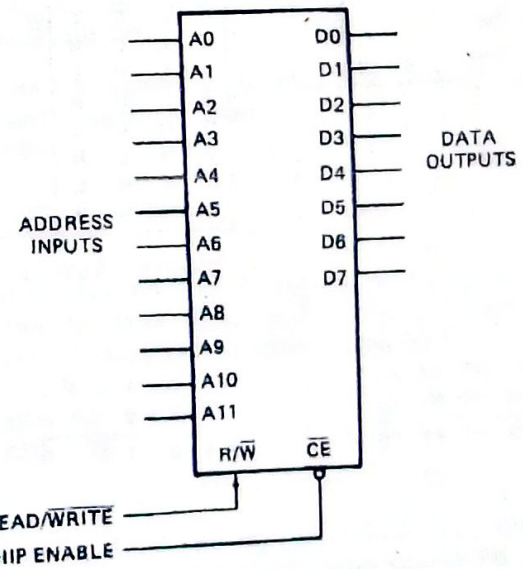


FIGURE 1-21 RAM schematic symbol.

would, a dynamic RAM chip can store many more bits than the same size static RAM chip. The disadvantage of dynamic RAMs is that the charge leaks off the tiny capacitors. The logic state stored in each capacitor must be *refreshed* every 2 milliseconds (ms) or so. A device called a *dynamic RAM refresh controller* can be used to refresh a large number of dynamic RAMs in a system. Some newer dynamic RAM devices contain built-in refresh circuitry, so they appear static to external circuitry.

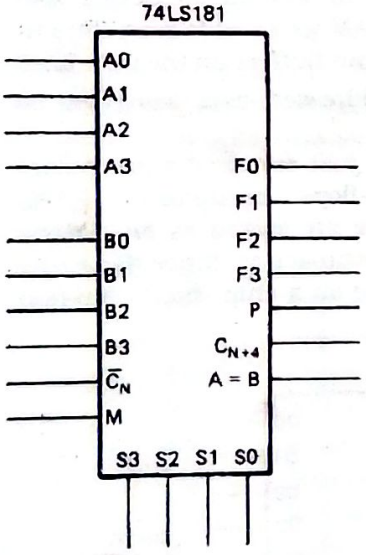
Arithmetic Logic Units

An *arithmetic logic unit*, or *ALU*, is a device that can AND, OR, add, subtract, and perform a variety of other operations on binary words. Figure 1-22*a* shows a block diagram for the 74LS181, which is a 4-bit ALU. This device can perform any one of 16 logic functions or any one of 16 arithmetic functions on two 4-bit binary words. The function performed on the two words is determined by the logic level applied to the mode input M and by the 4-bit binary code applied to the select inputs S0 to S3.

Figure 1-22*b* shows the truth table for the 74LS181. In this truth table, A represents the 4-bit binary word applied to the A0 to A3 inputs, and B represents the 4-bit binary word applied to the B0 to B3 inputs. F represents the 4-bit binary word that will be produced on the F0 to F3 outputs. If the mode input M is high, the device will perform one of 16 logic functions on the two words applied to the A and B inputs. For example, if M is high and we make S3 high, S2 low, S1 high, and S0 high, the 4-bit word on the A inputs will be ANDed with the 4-bit word on the B inputs. The result of this ANDing will appear on the F outputs. Each bit of the A word is ANDed with the corresponding bit of the B word to produce the result on F. Figure 1-22*c* shows an example of ANDing two words with this device. As you can see in this example, an output bit is high only if the corresponding bit is high in both the A word and the B word.

For another example of the operation of the 74LS181, suppose that the M input is high, S3 is high, S2 is high, S1 is high, and S0 is low. According to the truth table, the device will now OR each bit in the A word with the corresponding bit in the B word and give the result on the corresponding F output. Figure 1-22*c* shows the result that will be produced by ORing two 4-bit words. Figure 1-22*c* also shows for your reference the result that would be produced by exclusive ORing these two 4-bit words together.

If the M input of the 74LS181 is low, then the device will perform one of 16 arithmetic functions on the A and B words. Again, the result of the operation will be put on the F outputs. Several 74LS181s can be cascaded to operate on words longer than 4 bits. The ripple-carry input, $\overline{C}_N$, allows a carry from an operation on previous words to be included in the current operation. If the $\overline{C}_N$



(*a*)

SELECTION				ACTIVE–HIGH DATA		
				M = H	M = L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0	LOGIC FUNCTIONS	$\overline{C}_N$ = H (NO CARRY)	$\overline{C}_N$ = L (WITH CARRY)
L	L	L	L	$F = \overline{A}$	F = A	F = A PLUS 1
L	L	L	H	$F = \overline{A + B}$	F = A + B	F = (A + B) PLUS 1
L	L	H	L	$F = \overline{A}B$	$F = A + \overline{B}$	F = (A + $\overline{B}$) PLUS 1
L	L	H	H	F = 0	F = MINUS 1 (2's COMPL)	F = 0
L	H	L	L	$F = \overline{AB}$	F = A PLUS A$\overline{B}$	F = A PLUS A$\overline{B}$ PLUS 1
L	H	L	H	$F = \overline{B}$	F = (A + B) PLUS A$\overline{B}$	F = (A + B) PLUS A$\overline{B}$ PLUS 1
L	H	H	L	F = A⊕B	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	F = A$\overline{B}$	F = A$\overline{B}$ MINUS 1	F = A$\overline{B}$
H	L	L	L	$F = \overline{A} + B$	F = A PLUS AB	F = A PLUS AB PLUS 1
H	L	L	H	F = A⊕B	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	F = (A + $\overline{B}$) PLUS AB	F = (A + $\overline{B}$) PLUS AB PLUS 1
H	L	H	H	F = AB	F = AB MINUS 1	F = AB
H	H	L	L	F = 1	F = A PLUS A*	F = A PLUS A PLUS 1
H	H	L	H	$F = A + \overline{B}$	F = (A + B) PLUS A	F = (A + B) PLUS A PLUS 1
H	H	H	L	F = A + B	F = (A + $\overline{B}$) PLUS A	F = (A + $\overline{B}$) PLUS A PLUS 1
H	H	H	H	F = A	F = A MINUS 1	F = A

*EACH BIT IS SHIFTED TO THE NEXT MORE SIGNIFICANT BIT POSITION

(*b*)

A =	A3	A2	A1	A0
B =	B3	B2	B1	B0
F =	F3	F2	F1	F0

A =	1	0	1	0
B =	0	1	1	0
F = A + B =	1	1	1	0

A =	1	0	1	0
B =	0	1	1	0
F = A · B =	0	0	1	0

A =	1	0	1	0
B =	0	1	1	0
F = A⊕B =	1	1	0	0

(*c*)

FIGURE 1-22 Arithmetic logic unit (ALU). (*a*) Schematic symbol. (*b*) Truth table. (*c*) Sample AND, OR, and XOR operations.

input is asserted low, then a carry will be added to the results of the operation on A and B. For example, if the M input is low, S3 is high, S2 is low, S1 is low, S0 is high, and $\overline{C}_N$ is low, the F outputs will have the sum of A plus B plus a carry.

The real importance of an ALU such as the 74LS181 is that it can be programmed with a binary instruction applied to its mode and select inputs to perform many different functions on two binary words applied to its data inputs. In other words, instead of having to build a different circuit to perform each of these functions, we have one programmable device. We can perform any of the operations that we want in a computer with a sequence of simple operations such as those of the 74LS181. Therefore, an ALU is a very important part of the microprocessors and microcomputers that we discuss in the next chapter.

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms or concepts in this list, use the index to find them in the chapter.

Binary, bit, nibble, byte, word, doubleword

LSB, MSB, LSD, MSD

Hexadecimal, standard BCD, Gray code

7-segment display code

Alphanumeric codes: ASCII, EBCDIC

Parity bit, odd parity, even parity

Converting between binary, decimal, hexadecimal, BCD

Arithmetic with binary, hexadecimal, BCD

BCD decimal adjust operation

Signed numbers, sign bit

2's complement sign-and-magnitude form

Signal assertion level

Inverting and noninverting buffers

Symbols and truth tables for AND, NAND, OR, NOR, XOR logic gates

FPLA, PROM, PAL

D latch, D flip-flop

Register, shift register, binary counter

ROM: address lines, data lines, bus lines, three-state outputs and enable input

PROM, EPROM, EEPROM, flash EPROM

RAM: static, dynamic

ALU

REVIEW QUESTIONS AND PROBLEMS

1. Write the decimal equivalent for each integral power of 2 from 2^0 to 2^{20}.
2. Convert the following decimal numbers to binary:
   a. 22
   b. 76
   c. 500
3. Convert the following binary numbers to decimal:
   a. 1011
   b. 11010001
   c. 1110111001011001
4. Convert to hexadecimal:
   a. 53 decimal
   b. 756 decimal
   c. 01101100010 binary
   d. 11000010111 binary
5. Convert to decimal:
   a. D3H
   b. 3FEH
   c. 44H
6. Convert the following decimal numbers to BCD:
   a. 86
   b. 62
   c. 33
7. The L key is depressed on an ASCII-encoded keyboard. What pattern of 1's and 0's would you expect to find on the seven parallel data lines coming from the keyboard? What pattern would a carriage return, CR, give?
8. Define *parity* and describe how it is used to detect an error in transmitted data.
9. Show addition of:
   a. $1001 1_2$ and $101 1_2$ in binary
   b. 37_{10} and 25_{10} in BCD
   c. 4AH and 77H
10. Express the following decimal numbers in 8-bit sign-and-magnitude form:
    a. +26
    b. −7
    c. −26
    d. −125
11. Show the subtraction, in binary, of the following decimal numbers using both the pencil method and the 2's-complement addition method:
    a. 7 − 4
    b. 37 − 26
    c. 125 − 93
12. Show the multiplication of 1001 and 011 by the pencil method. Do the same for 11010 and 101.
13. Show the division of 1100100 by 1010 using the pencil method.

14. Perform the indicated operations on the following numbers:
    a. 3AH + 94H
    b. 17AH − 4CH
    c. 0101 1001 BCD
       + 0100 0010 BCD
    d. 0111 1001 BCD
       + 0100 1001 BCD
    e. 0101 1001 BCD
       − 0010 0110 BCD
    f. 0110 0111 BCD
       − 0011 1001 BCD

15. For the circuit in Figure 1-23:
    a. Is the Y output active high or active low?
    b. Is the C signal active high or active low?
    c. What input conditions on A, B, and C will cause the Y output to be asserted?

16. Desoribe how a D latch responds to a positive pulse on its CK input and how a D flip-flop responds to a positive pulse on its CK input.

17. The National Semiconductor INS8298 is a 65,536-bit ROM organized as 8192 words or bytes of 8 bits. How many address lines are required to address one of the 8192 bytes?

18. Why do most ROMs and RAMs have three-state outputs?

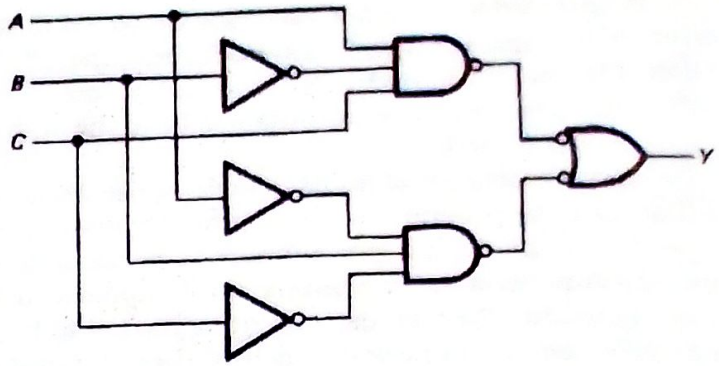


FIGURE 1-23 Circuit for problem 15.

19. Using Figure 1-22*b*, show the programming of the select and mode inputs the 74181 requires to perform the following arithmetic functions:
    a. A + B
    b. A − B − 1
    c. AB + A

20. Show the output word produced when the following binary words are ANDed with each other and when they are ORed with each other:
    a. 1010 and 0111
    b. 1011 and 1100
    c. 11010111 and 111000
    d. ANDing an 8-bit binary number with 1111 0000 is sometimes referred to as "masking" the lower 4 bits. Why?

Main frames: IBM - 4381 — GA 64 Register
Cray Y-MP/832

Minicomputers: Digital Equipment corporation's VAX-6360