Govt. Polytechnic for Women, Sirsa
E-Contents

SUBJECT: DIGITAL ELECTRONICS

BRANCH: COMPUTER ENGINEERING

SEM: 3rd

UNIT 1
NUMBER SYSTEMS AND CODES

Arithmetic operations using decimal numbers are quite common. However, in logical design it is necessary to perform manipulations in the so-called binary system of num- bers because of the on-off nature of the physical devices used. The present chapter is intended to acquaint the reader with the fundamental concepts involved in dealing with number systems other than decimal. In particular, the binary system is covered in con- siderable detail.

POSITIONAL NOTATION

An ordinary decimal number can be regarded as a polynomial in powers of 10. For example, 423.12 can be regarded as $4 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 1 \times 10^{-1} + 2 \times 10^{-2}$. Decimal numbers like this are said to be expressed in a number system with **base**, or **radix**, 10 because there are 10 basic digits (0, 1, 2, …, 9) from which the number system is formulated. In a similar fashion we can express any number N in a system using any base b. We shall write such a number as $(N)_b$. Whenever $(N)_b$ is written, the convention of always expressing b in base 10 will be followed. Thus $(N)_b = (p_n p_{n-1} \ldots p_1 p_0 . p_{-1} p_{-2} \ldots p_{-m})_b$ where b is an integer greater than 1 and $0 \leq p_i \leq b-1$. The value of a number represented in this fashion, which is called **positional notation**, is given by

$$(N)_b = p_n b_n + p_{n-1} b_{n-1} + \ldots + p_0 b_0 + p_{-1} b_{-1} + p_{-2} b_{-2} + \ldots + p_{-m} b_{-m} \qquad (1.1\text{-}1)$$

$$(N)_b = \sum_{i=-m}^{n} p_i b_i \qquad (1.1\text{-}2)$$

For decimal numbers, the symbol "." is called the **decimal point**; for more gen- eral base-*b* numbers, it is called the **radix point**. That portion of the number to the right of

the radix point ($p_{-1} p_{-2} \ldots p_{-m}$) is called the **fractional part**, and the portion to the left of the radix point ($p_n p_{n-1} \ldots p_0$) is called the **integral part**.

Numbers expressed in base 2 are called **binary numbers**. They are often used in computers since they require only two coefficient values. The integers from 0 to 15 are given in Table 1.1-1 for several bases. Since there are no coefficient values for the range 10 to $b-1$ when $b > 10$, the letters A, B, C, . . . are used. Base-8 numbers are called **octal numbers,** and base-16 numbers are called **hexadecimal numbers**. Octal and hexadecimal numbers are often used as a shorthand for binary numbers. An octal number can be converted into a binary number by converting each of the octal co- efficients individually into its binary equivalent. The same is true for hexadecimal numbers. This property is true because 8 and 16 are both powers of 2. For numbers with bases that are not a power of 2, the conversion to binary is more complex.

1.1-1 Conversion of Base

To make use of nondecimal number systems, it is necessary to be able to convert a number expressed in one base into the correct representation of the number in another base. One way of doing this makes direct use of the polynomial expression (1.1-1). For example, consider the binary number $(1011.101)_2$. The corresponding polynomial expression is

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

or 8 + 2 + 1 + 1/2 + 1/8 or 11 + 5/8 = 11.625

TABLE 1.1-1 Integers in various bases

$(N)_b$	2	3	4	5	...	8	...	10	11	12
			...	16						
	0001	001	01	01	01	01	01	01	1	
	0010	002	02	02	02	02	02	02	2	
	0011	010	03	03	03	03	03	03	3	
	0100	011	10	04	04	04	04	04	4	
	0101	012	11	10	05	05	05	05	5	
	0110	020	12	11	06	06	06	06	6	
	0111	021	13	12	07	07	07	07	7	
	1000	022	20	13	10	08	08	08	8	
	1001	100	21	14	11	09	09	09	9	
	1010	101	22	20	12	10	0A	0A	A	
	1011	102	23	21	13	11	10	0B	B	

1100	110	30	22		14		12	11	10		C
1101	111	31	23		15		13	12	11		D
1110	112	32	24		16		14	13	12		E
1111	120	33	30		17		15	14	13		F

(

This technique of directly evaluating the polynomial expression for a number is a general method for converting from an arbitrary base b_1 to another arbitrary base b_2. For convenience, it will be called the **polynomial method.** This method consists in:

1. Expressing the number $(N)_{b_1}$ as a polynomial, with base-b_2 numbers used in the polynomial.
2. Evaluating the polynomial, base-b_2 arithmetic being used.

This polynomial method is most often used by human beings whenever a number is to be converted to base 10, since it is then possible to use decimal arithmetic.

This method for converting numbers from one base to another is the first example of one of the major goals of this book: the development of algorithms. In general terms, an algorithm is a list of instructions specifying a sequence of operations which will give the answer to any problem of a given type. The important characteristics of an algorithm are: (1) that it is fully specified and does not rely on any skill or intuition on the part of the person applying it and (2) that it always works, (i.e., that a correct answer is always obtained.) The notion of an algorithm is discussed in more detail in Section 1.1 of [Knuth 68].

It is not always convenient to use base-b_2 arithmetic in converting from base-b_1 to base-b_2 . An algorithm for carrying out this conversion by using base-b_1 arithmetic will be discussed next. This discussion is specifically for the situation in which b_1 = 10, but it can be extended easily to the more general case. This will be called the **it- erative method**, since it involves iterated multiplication or division.

In converting $(N)_{10}$ to $(N)_b$ the fraction and integer parts are converted separately. First, consider the integer part (portion to the left of the decimal point). The general conversion procedure is to divide $(N)_{10}$ by b, giving $(N)_{10}/b$ and a remainder. The remainder, call it p_0 , is the least significant (rightmost) digit of $(N)_b$. The next least significant digit, p_1 , is the remainder of $(N)_{10}/b$ divided by b, and succeeding digits are obtained by continuing this process. A convenient form for carrying out this conversion is illustrated in the following example.

Example 1.1-1

(a) $(23)_{10}$ = $(10111)_2$

	2 | 23	(Remainder)
1	2 | 11	
1	2 | 5	
1	2 | 2	
0	2 | 1	
1	0	

(b) $(23)_{10}$ = $(27)_8$

	8 | 23	(Remainder)
7	8 | 2	
2	0	

1.1 Positional Notation

(c) $(410)_{10}$ = $(3120)_5$

	5 | 410	(Remainder)
0	5 | 82	
2	5 | 16	
1	5 | 3	
3	0	

Now consider the portion of the number to the right of the decimal point, i.e., the fractional part. The procedure for converting this is to multiply $(N)_{10}$ (fractional) by b. If the resulting product is less than 1, then the most significant (leftmost) digit of the fractional part is 0. If the resulting product is greater than 1, the most significant digit of the fractional part is the integral part of the product. The next most significant digit is formed by multiplying the fractional part of this product by b and taking the integral part. The remaining digits are formed by repeating this process. The process may or may not terminate. A convenient form for carrying out this conversion is illustrated be- low.

Example 1.1-2.

(c) $(27.68)_{10} = (11011.101011\ldots)_2 = (33.53\ldots)_8$

0.68 x 2 = 1.36

0.1

2 27

(a) $(0.625)_{10} = (0.5)_8$

0.625 x 8 = 5.000

0.5

(b) $(0.23)_{10} = (0.001110\ldots)_2$ 0.23 x 2 = 0.46 0.0

0.46 x 2 = 0.92 0.00

0.92 x 2 = 1.84 0.001

0.84 x 2 = 1.68 0.0011

0.68 x 2 = 1.36 0.00111

0.36 x 2	13	1	0.36 x 2 = 0.72	= 0.72 0.001110 ...
	6	1	0.72 x 2 = 1.44	
2 0.10	3	0	0.44 x 2 = 0.88	
2 0.101	1	1	0.88 x 2 = 1.76	
2				0.1010
2				0.10101
	0	1	0.76 x 2 = 1.52	0.101011 ...

8	27		0.68 x 8 = 5.44 0.5
8	3	3	0.44 x 8 = 3.52 0.53 …
	0	3	

This example illustrates the simple relationship between the base-2 (binary) sys- tem and the base-8 (octal) system. The binary digits, called **bits**, are taken three at a time in each direction from the binary point and are expressed as decimal digits to give the corresponding octal number. For example, 101 in binary is equivalent to 5 in decimal; so the octal number in part (c) above has a 5 for the most significant digit of the fractional part. The conversion between octal and binary is so simple that the octal expression is sometimes used as a convenient shorthand for the corresponding binary number.

When a fraction is converted from one base to another, the conversion may not terminate, since it may not be possible to represent the fraction exactly in the new base with a finite number of digits. For example, consider the conversion of $(0.1)_3$ to a base-10 fraction. The result is clearly $(0.333\ldots)_{10}$, which can be written as $(0.\overline{3})_{10}$ to indicate that the 3's are repeated indefinitely. It is always possible to represent the result of a

conversion of base in this notation, since the nonterminating fraction must consist of a group of digits which are repeated indefinitely. For example, $(0.2)_{11} = 2 \times 11^{-1} = (0.1818\ldots)_{10} = (0.0\overline{18})_{10}$.

It should be pointed out that by combining the two conversion methods it is pos- sible to convert between any two arbitrary bases by using only arithmetic of a third base. For example, to convert $(16)_7$ to base 3, first convert to base 10,

$$(16)_7 = 1 \times 7^1 + 6 \times 7^0 = 7 + 6 = (13)_{10}$$ Then

convert $(13)_{10}$ to base 3,

3	13	(Remainder)		
3	4		1	$(16)_7 = (13)_{10} = (111)_3$
3	1	1		
3	0	1		

For more information about positional number systems, the following references are good sources: [Chrystal 61] and [Knuth 69].

BINARY ARITHMETIC

Many modern digital computers employ the binary (base-2) number system to represent numbers, and carry out the arithmetic operations using binary arithmetic. While a de- tailed treatment of computer arithmetic is not within the scope of this book, it will be useful to have the elementary techniques of binary arithmetic available. In performing decimal arithmetic it is necessary to memorize the tables giving the results of the elemen- tary arithmetic operations for pairs of decimal digits. Similarly, for binary arithmetic the tables for the elementary operations for the binary digits are necessary.

1.2-1 Binary Addition

The binary addition table is as follows:

Sum	Carry
0 + 0 = 0	0
0 + 1 = 1	0
1 + 0 = 1	0
1 + 1 = 0	1

Addition is performed by writing the numbers to be added in a column with the binary points aligned. The individual columns of binary digits, or **bits**, are added in the usual order according to the above addition table. Note that in adding a column of

bits, there is a 1 carry for each pair of 1's in that column. These 1 carries are treated as bits to be added in the next column to the left. A general rule for addition of a column of numbers (using any base) is to add the column decimally and divide by the base. The remainder is entered as the sum for that column, and the quotient is carried to be added in the next column.

Example 1.2-1

Base 2

Carries: *10011 11*

1001.011 = $(9.375)_{10}$
1101.101 = $(13.625)_{10}$
10111.000 = $(23)_{10}$ = Sum

1.2-2 Binary Subtraction

The binary subtraction table is as follows:

	Difference	Borrow
0 − 0 = 0	0	
0 − 1 = 1	1	
1 − 0 = 1	0	
1 − 1 = 0	0	

Subtraction is performed by writing the minuend over the subtrahend with the binary points aligned and carrying out the subtraction according to the above table. If a borrow occurs and the next leftmost digit of the minuend is a 1, it is changed to a 0 and the process of subtraction is then continued from right to left.

	Base 2	Base 10
Borrow:	*1*	
	0	
Minuend	1\0	2
Subtrahend	−01	−1
Difference	01	1

If a borrow occurs and the next leftmost digit of the minuend is a 0, then this 0 is changed to a 1, as is each successive minuend digit to the left which is equal to 0. The first minuend digit to the left which is equal to 1 is changed to 0, and then the subtrac- tion process is resumed.

	Base 2	Base 10
Borrow:	*1*	
	011	
Minuend	11\0\0\0	24
Subtrahend	□10001	□17
Difference	00111	7
Borrow:	*1 1*	
Minuend	01011	
	1\0\1\0\0\0	40
Subtrahend	□011001	□25
Difference	001111	15

1.2-3 Complements

It is possible to avoid this subtraction process by using a complement representation for negative numbers. This will be discussed specifically for binary **fractions**, although it is easy to extend the complement techniques to integers and mixed numbers. The **2's complement** (^2B) of a binary fraction B is defined as follows:

$$^2B = (2 - B)_{10} = (10 - B)_2$$

Thus, $^2(0.1101) = 10.0000 - 0.1101 = 1.0011$. A particularly simple means of carry- ing out the subtraction indicated in the expression for $^2(0.1101)$ is obtained by noting that $10.0000 = 1.1111 + 0.0001$. Thus, $10.0000 - 0.1101 = (1.1111 - 0.1101) + 0.0001$. The subtraction $1.1111 - 0.1101$ is particularly easy, since all that is neces- sary is to reverse each of the digits of 0.1101 to obtain 1.0010. Finally, the addition of 0.0001 is also relatively simple, and yields 1.0011. In general, the process of forming ^2B involves reversing the digits of B and then adding 0.00 ... 01.

The usefulness of the 2's complement stems from the fact that it is possible to obtain the difference $A - B$ by adding ^2B to A. Thus, $A + {}^2B = (A + 10 - B)_2 = (10 + (A - B))_2$. If $(A - B) > 0$, then $(10 + A - B)_2$ will be 10 plus the positive fraction $(A - B)$. It is thus possible to obtain $A - B$ by dropping the leftmost 1 in $A + {}^2B$. For example,

$$\begin{array}{r} 1.0011 \\ \hline 10.0001 \end{array}$$

$A =$	0.1110	$A =$	0.1110
$-B =$	−0.1101	$+ {}^2B =$	
	0.0001		

If $(A - B) < 0$, then $A + {}^2B = (10 \ldots 0 - |A - B|)_2$, which is just equal to ${}^2(A - B)$, the 2's-complement representation of $A - B$. For example,

$A =$	0.1101	$A =$	0.1101	
$-B =$	-0.1110	$+ {}^2B =$	1.0010	
	-0.0001		1.1111	${}^2(0.0001) = 1.1111$

The ^{1}'s complement is also very commonly used. This is defined as

If $A + {}^1B$ is formed, the result is $(A - B + 10 - 0.000 \ldots 1)_2$. If $(A - B) > 0$, this can be converted to $A - B$ by removing the $(10)_2$ and adding a 1 to the least significant digit of $A + {}^1B$. This is called an **end-around carry**. For example:

			+0.0001	
			0.0001	
$A =$	0.1110	$A =$	0.1110	
$-B =$	-0.1101	$+ {}^1B =$	+1.0010	
	0.0001	$A + {}^1B =$	10.0000	

End-around carry

so that $A - B =$ 0.0001

If $(A - B) < 0$, then $A + {}^1B$ will be the 1's complement of $|A - B|$. For example,

$A =$	0.1101	$A =$	0.1101
$-B =$	-0.1110	${}^1B =$	1.0001

---

[1] ${}^1B = (10 - 0.000 \ldots 1 - B)_2$

where the location of the 1 in 0.000 ... 1 corresponds to the least significant digit of B. Since $(10 - 0.000 \ldots 1)_2$ is equal to 01.111 ... 1, it is possible to form ${}^1B$ by revers- ing the digits of B and adding a 1 before the radix point. Thus, ${}^1(0.1101) = 1.0010$.

-0.0001 $A + {}^{1}B =$ 1.1110 $^{1}(0.0001) = 1.1110$

The **radix complement** of a base-b fraction F is defined as

$$^{b}F = (10 - F)_b$$

and the **diminished radix complement** is defined as

$$^{b-1}F = (10 - F - 0.000 \ldots 1)_b$$

Similar procedures hold for the formation of the complements and their use for subtraction.

When integers or mixed numbers are involved in the subtractions, the definitions of the complements must be generalized to

$$^{b}N = (100 \ldots 0. - N)_b$$

and

$$^{b-1}N = (100 \ldots 0. - N - 0.00 \ldots 1)_b$$

where $100 \ldots 0$ contains two more digits than any integer to be encountered in the subtractions. For example, if $(N)_2 = 11.01$, then

$$\begin{aligned} {}^{2}(N)_2 &= 1000.00 - 11.01 \\ &= 111.11 - 11.01 + 0.01 \\ &= 100.10 + 0.01 \\ &= 100.11 \end{aligned}$$

$$\begin{array}{rr} M = & 11.10 \\ -N = & \underline{-11.01} \\ & 0.01 \end{array} \qquad \begin{array}{rr} M = & 11.10 \\ {}_{2}N = & \underline{100.11} \\ & 1000.01 \\ & --- \\ & -- \\ & \text{Discard} \end{array}$$

1.2-4 Shifting

In carrying out multiplication or division there are intermediate steps which require that numbers be shifted to the right or the left. Shifting a base-b number k places to the right has the effect of multiplying the number by b^{-k}, and shifting k places to the left is equivalent to multiplication by b^{+k}. Thus, if n

$$(N)_b = \sum_{i=-m}^{n} p_i b^i = (p_n p_{n-1} \ldots p_1 p_0 . p_{-1} p_{-2} \ldots p_{-m})_b$$

shifting $(N)_b$ k places to the left yields

$$(p_n p_{n-1} \ldots p_1 p_0 p_{-1} \ldots p_{-k} . p_{-k-1} \ldots p_{-m})_b = \sum_{i=-m}^{n} p_i b^{i+k}$$

and

$$\sum_{i=-m}^{n} p_i b^{i+k} = b^k \sum_{i=-m}^{n} p_i b^i = b^k (N)_b$$

A similar manipulation shows the corresponding situation for right shifts. Shifting the binary point k places (k positive for right shifts and negative for left shifts) in a bi- nary number multiplies the value of the number by 2^k. For example,

$$(110.101)_2 = (6.625)_{10}$$

$$(1.10101)_2 = 2^{-2} (6.625)_{10} = \left(6.62\right)^{\frac{5}{4}}{}_{10} = (1.65625)_{10}$$

$$(11010.1)_2 = 2^{+2} (6.625)_{10} = (4 \times 6.625)_{10} = (26.5)_{10}$$

1.2-5 Binary Multiplication

The binary multiplication table is as follows:

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

The process of binary multiplication is illustrated by the following example:

110.10 Multiplicand
10.1 Multiplier

```
   11010      Partial Product
  00000       Partial Product
 11010
------
10000.010     Partial Product
```


For every digit of the multiplier which is equal to 1, a partial product is formed consisting of the multiplicand shifted so that its least significant digit is aligned with the 1 of the multiplier. An all-zero partial product is formed for each 0 multiplier digit. Of course, the all-zero partial products can be omitted. The final product is formed by summing all the partial products. The binary point is placed in the product by using the same rule as for decimal multiplication: the number of digits to the right of the binary point of the product is equal to the sum of the numbers of digits to the right of the binary points of the multiplier and the multiplicand.

The simplest technique for handling the multiplication of negative numbers is to use the process just described to multiply the magnitudes of the numbers. The sign of the product is determined separately, and the product is made negative if either the multiplier or the multiplicand, but not both, are negative. It is possible to carry out multiplication directly with negative numbers represented in complement form. This is usually done using a recoding scheme called Booth's Algorithm, [Waser 82], which also speeds up the multiplication.

1.2-6 Binary Division

Division is the most complex of the four basic arithmetic operations. Decimal long division as taught in grade school is a trial-and-error process. For example, in dividing 362 by 46 one must first recognize that 46 is larger than 36 and then must guess how many times 46 will go into 362. If an initial guess of 8 is made and the multiplication8 x 46 = 368 is carried out, the result is seen to be larger than 362 so that the 8 must be replaced by a 7. This process of trial and error is simpler for binary division because there are fewer possibilities in the binary case.

To implement binary division in a digital computer a division algorithm must be specified. Two different algorithms, called restoring and nonrestoring division, are used.

Restoring division is carried out as follows: In the first step, the divisor is subtracted from the dividend with their leftmost digits aligned. If the result is positive, a 1 is entered as the quotient digit corresponding to the rightmost digit of the dividend from which a digit of the divisor was subtracted. The next rightmost digit of the dividend is appended to the result, which then becomes the next partial dividend. The divisor is then shifted one place to the right so that its least significant digit is aligned with the rightmost digit of the partial dividend, and the process just described is repeated.

If the result of subtracting the divisor from the dividend is negative, a 0 is entered in the quotient and the divisor is added back to the negative result so as to **restore** the original

dividend. The divisor is then shifted one place to the right, and a subtraction is carried out again. The process of restoring division is illustrated in the following example at the top of the next page:

Divisor = 1 1 1 1 Dividend = 1 1 0 0

q_0 q_{-1} q_{-2} q_{-3} q_{-4} q_{-5}

0 .1 1 0 0 1

1 1 1 1 \r(1 1 0 0 .0 0 0 0 0)

Subtract		1 1 1 1
Negative result	$q_0 = 0$	−0 0 1 1
Restore		+1 1 1 1
		1 1 0 0 0
Subtract		1 1 1 1
Positive result	$q_{-1} = 1$	1 0 0 1 0
Subtract		1 1 1 1
Positive result	$q_{-2} = 1$	0 0 0 1 1 0
Subtract		1 1 1 1
Negative result	$q_{-3} = 0$	−1 0 0 1
Restore		+ 1 1 1 1
		0 1 1 0 0
Subtract		1 1 1 1
Negative result	$q_{-4} = 0$	−0 0 1 1
Restore		+ 1 1 1 1
		1 1 0 0 0
Subtract		1 1 1 1
Positive result	$q_{-5} = 1$	1 0 0 1 (remainder)

In **nonrestoring division**, the step of adding the divisor to a negative partial dividend is omitted, and instead the shifted divisor is added to the negative partial divi- dend. This step of adding the shifted divisor replaces the two steps of adding the divi- sor and then subtracting the shifted divisor. This can be justified as follows: If X rep- resents the negative partial dividend and Y the divisor, then $1/2Y$ represents the divi- sor shifted one place to the right. Adding the divisor and then subtracting the shifted divisor yields $X + Y - 1/2Y = X + 1/2Y$, while adding the shifted divisor yields the same result, $X + 1/2Y$. The steps which occur in using nonrestoring division to divide 1100 by 1111 are shown in the following example at the top of the next page:

Divisor = 1 1 1 1 Dividend = 1 1 0 0

q_0 $q_{□□}$ $q_{□□}$ $q_{□□}$ $q_{□□}$ $q_{□□}$

0 .1 1 0 0 1

1 1 1 1 \r(1 1 0 0 .0 0 0 0 0 0)

Subtract		1 1 1 1
Negative result	q_0 = 0	□0 0 1 1 0
Shift and add		+ 1 1 1 1
Positive result	$q_{□□}$ = 1	+ 1 0 0 1 0
Shift and subtract		□ 1 1 1 1
Positive result	$q_{□□}$= 1	+ 0 0 1 1 0
Shift and subtract		□ 1 1 1 1
Negative result	$q_{□□}$= 0	□ □ □ □ □ □□
Shift and add		□1 1 1 1
Negative result	$q_{□□}$= 0	□0 0 1 1 0
Shift and add		+ 1 1 1 1
Positive result	$q_{□□}$= 1	+ 1 0 0 1 (remainder)

□

An important technique for improving the performance of digital arithmetic circuitry is the use of more sophisticated algorithms for the basic arithmetic operations. A discussion of these methods is beyond the scope of this book. The interested reader is referred to [Waser 82], [Hwang 78], or Chapter 2 and Section 8.1 in [Gschwind 75] for more details on arithmetic.

BINARY CODES

The binary number system has many advantages and is widely used in digital systems. However, there are times when binary numbers are not appropriate. Since we think much more readily in terms of decimal numbers than binary numbers, facilities are usually provided so that data can be entered into the system in decimal form, the con- version to binary being performed automatically inside the system. In fact, many com- puters have been designed which work entirely with decimal numbers. For this to be possible, a scheme for representing each of the 10 decimal digits as a sequence of bi- nary digits must be used.

1.3-1 Binary-Coded-Decimal Numbers

To represent 10 decimal digits, it is necessary to use at least 4 binary digits, since there are 2^4 , or 16, different combinations of 4 binary digits but only 2^3, or 8, different combinations of 3 binary digits. If 4 binary digits, or **bits**, are used and only one combination of bits is used to represent each decimal digit, there will be six unused or

invalid code words. In general, any arbitrary assignment of combinations of bits to digits can be used so that there are 16!/6! or approximately 2.9 □ 10^{10}

TABLE 1.3-1 Some common 4-bit decimal codes

Decimal digit	8 b_3	4 b_2	2 b_1	1 b_0	8	4	-2	-1	2	4	2	1	Excess-3			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	1	1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	1	0	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	0	1	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	1	1	1	0	1	1	1	0	0	0
6	0	1	1	0	1	0	1	0	1	1	0	0	1	0	0	1
7	0	1	1	1	1	0	0	1	1	1	0	1	1	0	1	0
8	1	0	0	0	1	0	0	0	1	1	1	0	1	0	1	1
9	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0

possible codes. Only a few of these codes have ever been used in any system, since the arithmetic operations are very difficult in almost all of the possible codes. Several of the more common 4-bit decimal codes are shown in Table 1.3-1.

The 8,4,2,1 code is obtained by taking the first 10 binary numbers and assigning them to the corresponding decimal digits. This code is an example of a **weighted code**, since the decimal digits can be determined from the binary digits by forming the sum $d = 8b_3 + 4b_2 + 2b_1 + b_0$. The coefficients 8, 4, 2, 1 are known as the **code weights**. The number 462 would be represented as 0100 0110 0010 in the 8,4,2,1 code. It has been shown in [White 53] that there are only 17 different sets of weights possible for a positively weighted code: (3,3,3,1), (4,2,2,1), (4,3,1,1), (5,2,1,1), (4,3,2,1), (4,4,2,1), (5,2,2,1), (5,3,1,1), (5,3,2,1), (5,4,2,1), (6,2,2,1), (6,3,1,1), (6,3,2,1), (6,4,2,1), (7,3,2,1), (7,4,2,1), (8,4,2,1).

It is also possible to have a weighted code in which some of the weights are negative, as in the 8,4,□2,□1 code shown in Table 1.3-1. This code has the useful property of being **self-complementing**: if a code word is formed by complementing each bit individually (changing 1's to 0's and 0's to 1's), then this new code word represents the 9's complement of the digit to which the original code word corresponds. For example, 0101 ' represents denotes the 3 in complement of the 8,4, □2,□1 code,b_i , then a and 1010code is represents self-complementing if, 6 in this code.

In general, if b_i ' ' ' '

for any code word $b_3b_2b_1b_0$ representing a digit d_i , the code word $\overset{b}{}_3\ \overset{b}{}_2\ \overset{b}{}_1\ \overset{b}{}_0$ represents 9□d_i. The 2,4,2,1 code of Table 1.3-1 is an example of a self-complementing code having all positive weights, and the excess-3 code is an example of a code which is self-complementing but not weighted. The excess-3 code is obtained from the 8,4,2,1 code by adding (using binary arithmetic) 0011 (or 3) to each 8,4,2,1 code word to obtain the corresponding excess-3 code word.

Although 4 bits are sufficient for representing the decimal digits, it is sometimes expedient to use more than 4 bits in order to achieve arithmetic simplicity or ease in er- ror detection. The 2-out-of-5 code shown in Table 1.3-2 has the property that each code word has exactly two 1's. A single error which complements 1 of the bits will

TABLE 1.3-2 Some decimal codes using more than 4 bits.

Decimal digit	2-out-of-5	Biquinary 5043210
0	00011	0100001
1	00101	0100010
2	00110	0100100
3	01001	0101000
4	01010	0110000
5	01100	1000001
6	10001	1000010
7	10010	1000100
8	10100	1001000
9	11000	1010000

always produce an invalid code word and is therefore easily detected. This is an unweighted code. The biquinary code shown in Table 1.3-2 is a weighted code in which 2 of the bits specify whether the digit is in the range 0 to 4 or the range 5 to 9 and the other 5 bits identify where in the range the digit occurs.

GEOMETRIC REPRESENTATION OF BINARY NUMBERS

An *n*-bit binary number can be represented by what is called a **point in *n*- space**. To see just what is meant by this, consider the set of 1-bit binary numbers, that is, 0 and 1. This set can be represented by two points in 1-space, i.e., by two points on a line. Such a presentation is called a **1-cube** and is shown in Fig.1.4-1b.
(A **0-cube** is a single point in 0-space.)

Now consider the set of 2-bit binary numbers, that is, 00, 01, 10, 11 (or, deci- mally, 0, 1, 2, 3). This set can be represented by four points (also called **vertices**, or **nodes**) in 2-space. This representation is called a **2-cube** and is shown in Fig.1.4-1c. Note that this figure can be obtained by projecting the 1-cube (i.e., the horizontal line with two points) downward and by prefixing a 0 to 0 and 1 on the original 1-cube and a 1 to 0 and 1 on the projected 1-cube. A similar projection procedure can be followed in obtaining any next-higher-dimensional figure. For example, the representation for the

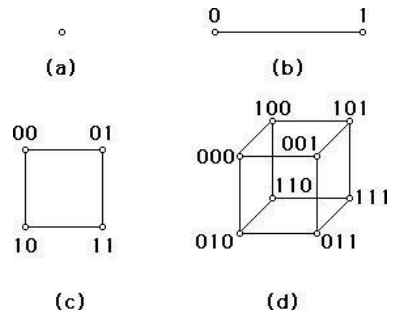


Figure 1.4-1 *n*-Cubes for *n* = 0, 1, 2, 3: (a) 0-cube; (b) 1-cube; (c) 2- cube; (d) 3-cube.

set of 3-bit binary numbers is obtained by projecting the 2-cube representation of Fig.1.4-1c. A 0 is prefixed to the bits on the original 2-cube, and a 1 is prefixed to the bits on the projection of the 2-cube. Thus, the 3-bit representation, or **3-cube**, is shown in Fig. 1.4-1d.

A more formal statement for the projection method of defining an *n*-cube is as follows:

1. A 0-cube is a single point with no designation.
2. An *n*-cube is formed by projecting an (*n*-1)-cube. A 0 is prefixed to the designations of the points of the original (*n*-1)-cube, and a 1 is prefixed to the designations of the points of the projected (*n*-1)-cube.

There are 2^n points in an *n*-cube. A ***p*-subcube** of an *n*-cube. ($p \leq n$) is de- fined as a collection of any 2^p points which have exactly ($n-p$) corresponding bits all the same. For example, the points 100, 101, 000, and 001 in the 3-cube (Fig.1.4-1d) form a 2-subcube, since there are $2^2 = 4$ total points and $3-2 = 1$ of the bits (the sec-

ond*p*-subcubes) is the sam in ane fo *n*-rcube all fou, sincr pointse ther.e arIne general,(C_n

) = there ($n!/($ are$n-p)!n!2p!)^{n-p}$ (^{p}number)/[($n-p$ of)!p ways!] different of se-

$n-p$

lecting *n* things taken $n-p$ at a time) ways in which $n-p$ of the bits may be the same, and there are 2^{n-p} combinations which these bits may take on. For example, there are ($3!2^2$)/(2!1!) = 12 1-subcubes (line segments) in a 3-cube, and there are ($3!2^1$)/(1!2!) = 6 2-subcubes ("squares") in a 3-cube.

Besides the form shown in Fig.1.4-1, there are two other methods of drawing an *n*-cube which are frequently used. The first of these is shown in Fig.1.4-2 for the 3- and 4-cubes. It is seen that these still agree with the projection scheme and are merely a particular way of drawing the cubes. The lines which are dotted are usually omitted for convenience in drawing.

If in the representation of Fig.1.4-2 we replace each dot by a square area, we have what is known as an ***n*-cube map**. This representation is shown for the 3- and 4- cubes in Fig. 1.4-3. Maps will be of considerable use to us later. Notice that the appropriate entry for each cell of the maps of Fig.1.4-3 can be determined from the corresponding row and column labels.

It is sometimes convenient to represent the points of an *n*-cube by the decimal equivalents of their binary designations. For example, Fig.1.4-4 shows the 3- and 4- cube maps represented this way. It is of interest to note that, if a point has the decimal equivalent N_i in an *n*-cube, in an (n + 1)-cube this point and its projection (as defined) become N_i and $N_i + 2^n$.

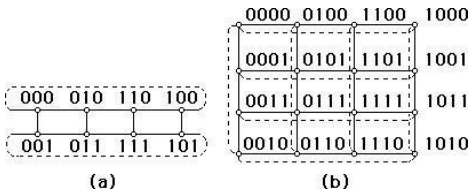


Figure 1.4-2 Alternative representa- tions: (a) 3-cube; (b) 4-cube.

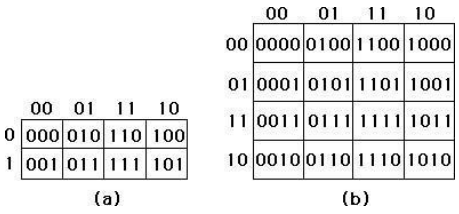


Figure 1.4-3 *n*-Cube maps for *n* = 3 (a) and *n* = 4 (b).

1.4-1 Distance

A concept which will be of later use is that of the distance between two points on an n-cube. Briefly, the **distance** between two points on an *n*-cube is simply the number of coordinates (bit positions) in which the binary representations of the two points differ. This is also called the **Hamming distance**. For example, 10110 and 01101 differ in all but the third coordinate (from left or right). Since the points differ in four coordi- nates, the distance between them is 4. A more formal definition is as follows: First, define the **mod** 2 **sum** of two bits, $a \oplus b$, by

$$0 \oplus 0 = 0 \qquad 1 \oplus 0 = 1$$

$$0 \oplus 1 = 1 \qquad 1 \oplus 1 = 0$$

That is, the sum is 0 if the 2 bits are alike, and it is 1 if the 2 bits are different. Now consider the binary representations of two points, $P_i = (a_{n-1} a_{n-2} \ldots a_0)$ and $P_j = (b_{n-1} b_{n-2} \ldots b_0)$, on the *n*-cube. The mod 2 sum of these two points is defined as

$$P_k = P_i \oplus P_j = (a_{n-1} \oplus b_{n-1}, a_{n-2} \oplus b_{n-2}, \ldots a_0 \oplus b_0)$$

This sum P_k is the binary representation of another point on the *n*-cube. The number of 1's in the binary representation P_i is defined as the **weight** of P_i and is given the sym- bol $|P_i|$. Then the distance (or **metric**) between two points is defined as

$$D(P_i, P_j) = |P_i \oplus P_j|$$

The distance function satisfies the following three properties:

$D(P_i, P_j) = 0$ if and only if $P_i = P_j$

$D(P_i, P_j) = D(P_j, P_i) > 0$ if $P_i \neq P_j$

$D(P_i, P_j) + D(P_j, P_k) \geq D(P_i, P_k)$ Triangle inequality

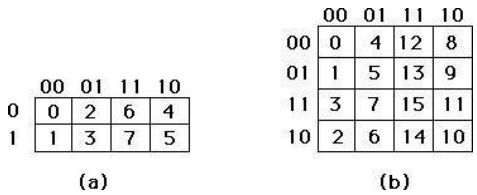


Figure 1.4-4 Decimal labels in *n*-cube maps: (a) 3-cube map; (b) 4-cube map.

To return to the more intuitive approach, since two adjacent points (connected by a single line segment) on an *n*-cube form a 1-subcube, they differ in exactly one coordi- nate and thus are distance 1 apart. We see then that, to any two points which are dis- tance *D* apart, there corresponds a **path** of *D* connected line segments on the *n*-cube joining the two points. Furthermore, there will be more than one path of length *D* con- necting the two points (for $D > 1$ and $n \geq 2$), but there will be no path shorter than length *D* connecting the two points. A given shortest path connecting the two points, thus, cannot intersect itself, and $D + 1$ nodes (including the end points) will occur on the path.

1.4-2 Unit-distance Codes

In terms of the geometric picture, a code is simply the association of the decimal inte- gers (0,1,2,...) with the points on an *n*-cube. There are two types of codes which are best described in terms of their geometric properties. These are the so-called **unit- distance codes** and **error-detecting** and **error-correcting codes**.

A unit-distance code is simply the association of the decimal integers (0,1,2,**...**) with the points on a connected path in the *n*-cube such that the distance is 1 between the point corresponding to any integer *i* and the point corresponding to integer $i + 1$ (see Fig. 1.4-5). That is, if P_i is the binary-code word for decimal integer *i*, then we must have

$$D(Pi\ ,\ Pi + 1) = 1 \qquad i = 0, 1, 2, \ldots$$

Unit-distance codes are used in devices for converting analog or continuous sig- nals such as voltages or shaft rotations into binary numbers which represent the magni- tude of the signal. Such a device is called an **analog-digital converter**. In any such device there must be boundaries between successive digits, and it is always possible for there to be some misalignment among the different bit positions at such a boundary. For example, if the seventh position is represented by 0111 and the eighth position by 1000, misalignment

could cause signals corresponding to 1111 to be gen- erated at the boundary between 7 and 8. If binary numbers were used for such a de- vice, large errors could thus occur. By using a unit-distance code in which adjacent positions differ only in 1 bit, the error due to misalignment can be eliminated.

The highest integer to be encoded may or may not be required to be distance 1 from the code word for 0. If it is distance 1, then the path is closed. Of particular interest is the case of a closed nonintersecting path which goes through all 2^n points of the n-cube. In graph theory such a path is known as a (closed) **Hamilton line**. Any unitdistance code associated with such a path is sometimes called a **Gray code**, although this term is usually reserved for a particular one of these codes. To avoid

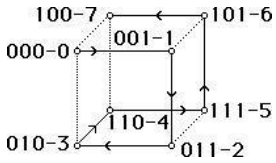


Figure 1.4-5 Path on a 3-cube corresponding to a unit-distance code.

TABLE 1.4-1 Unit-dis- tance code of Fig. 1.4-5

0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

confusing terminology, we shall refer to a unit-distance code which corresponds to a closed Hamilton line as a **closed *n* code**. This is a unit-distance code containing 2^n code words in which the code word for the largest integer (2^n □ 1) is distance 1 from the code word for the least integer (0). An **open *n* code** is similar except that the code words for the least and largest integer, respectively, are not distance 1 apart.

The most useful unit distance code is the Gray code which is shown in Table 1.4- 2. The attractive feature of this code is the simplicity of the algorithm for translating from the binary number system into the Gray code. This algorithm is described by the expression

$g_i = b_i \square b_{i+1}$

TABLE 1.4-2 The Gray code

	Binary				Gray			
Decimal	b_3	b_2	b_1	b_0	g_3	g_2	g_1	g_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1

2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

Thus, the Gray code word corresponding to 1100 in binary is formed as follows: $g_0 = b_0 \oplus b_1 = 0 \oplus 0 = 0$ $g_1 = b_1 \oplus b_2 = 0 \oplus 1 = 1$ $g_2 = b_2 \oplus b_3 = 1 \oplus 1 = 0$

$g_3 = b_3 \oplus b_4 = b_3 = 1$ b_4 understood to be 0

1.4-3 Symmetries of the n-Cube

A **symmetry** of the *n*-cube is defined to be any one-to-one translation of the binary point representations on the *n*-cube which leaves all pairwise distances the same. If we consider the set of binary numbers, we see that there are only two basic translation schemes which leave pairwise distances the same. (1) The bits of one coordinate may be interchanged with the bits of another coordinate in all code words. (2) The bits of one coordinate may be complemented (i.e., change 1's to 0's and 0's to 1's) in all code words. Since there are $n!$ translation schemes possible using (1), and since there are $2n$ ways in which coordinates may be complemented, there are $2n$ translation schemes possible using (2). Thus, in all there are $2^n(n!)$ symmetries of the *n*-cube. This means that for any *n*-bit code there are $2^n(n!) - 1$ rather trivial modifications of the original code (in fact, some of these may result in the original code) which can be obtained by interchanging and complementing coordinates. The pairwise distances are the same in all these codes.

It is sometimes desired to ennumerate the different types of a class of codes. Two codes are said to be of the same **type** if a symmetry of the *n*-cube translates one code into the other (i.e., by interchanging and complementing coordinates). As an example, we might ask: What are the types of closed *n* codes? It turns out that for $n < 4$ there is just one type, and this is the type of the conventional Gray code. For $n = 4$, there are nine types. Rather than specify a particular code of each type, we can list these types by specifying the sequence of coordinate changes for a closed path of that type. On the as- sumption that the coordinates are numbered (3210), the nine types are shown in Table 1.4-3.

TABLE 1.4-3 Nine different types of unit-distance 4-bit code

Type																
1 (Gray)	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	3
2	1	0	1	3	1	0	1	2	0	1	0	3	0	1	0	2
3	1	0	1	3	0	1	0	2	1	0	1	3	0	1	0	2
4	1	0	1	3	2	3	1	0	1	3	1	0	2	0	1	3
5	1	0	1	3	2	0	1	3	1	0	1	3	2	0	1	3
6	1	0	1	3	2	3	1	3	2	0	1	2	1	3	1	2
7	1	0	1	3	2	0	2	1	0	2	0	3	0	1	0	2
8	1	0	1	3	2	1	2	0	1	2	1	3	0	1	0	2
9	1	0	1	3	2	3	1	0	3	0	2	0	1	2	3	2

ERROR-DETECTING AND ERROR-CORRECTING CODES

Special features are included in many digital systems for the purpose of increasing system reliability. In some cases circuits are included which indicate when an error has occurred—error detection—and perhaps provide some information as to where the error is—error diagnosis. Sometimes it is more appropriate to provide **error correction**: circuits not only detect a malfunction but act to automatically correct the erroneous indications caused by it. One technique used to improve reliability is to build two duplicate systems and then to run them in parallel, continually comparing the outputs of the two systems, [Burks 62]. When a mismatch is detected, actions are initiated to determine the source of the error and to correct it, [Keister 64]. Another approach uses three copies of each system module and relies on voter elements to select the correct output in case one of the three copies has a different output from the other two, ([von Neumann 56], [Lyons 62]). This technique is called triple modular redundancy (TMR). Such costly designs are appropriate either when the components are not sufficiently reliable [Burks 62] or in systems where reliability is very important as in real-time applications such as telephony, [Keister 64], airline reservations, [Perry 61], or space vehicles, [Dickinson 64].

In many other applications where such massive redundancy is not justified it is still important to introduce some (less costly) techniques to obtain some improvement in reliability. A very basic and common practice is to introduce some redundancy in encoding the information manipulated in the system. For example, when the 2-out-of- 5 code is used to represent the decimal digits, any error in only one bit is easily detected since if any single bit is changed the resulting binary word no longer contains exactly two 1's. While it is true that there are many 2-bit errors which will not be detected by this code, it is possible to argue that in many situations multiple errors are so much less likely than single errors that it is reasonable to ignore all but single errors.

Suppose it is assumed that the probability of any single bit being in error is *p* and that this probability is independent of the condition of any other bits. Also suppose that *p* is very much less than one, (i.e., that the components are very reliable). Then the

probability of all 5 bits representing one digit being correct is $P_0 = (1-p)^5$, the probability of exactly one error is $P_1 = 5(1-p)^4 p$ and the probabilty of two errors is $P_2 = 10(1-p)^3 p^2$. Taking the ratio $P_2/P_1 = 2p/(1-p) \approx 2p/(1+p) << 1$, showing that the probabilty of a double error is much smaller than that of a single error. Arguments such as this are the basis for the very common emphasis on handling only single errors.

It is possible to easily convert any of the 4-bit decimal codes to single-error-detecting codes by the addition of a single bit—a parity bit as is illustrated for the 8421 code in Table 1.5-1. The **parity bit** p is added to each code word so as to make the total number of 1's in the resultant 5-bit word even; i.e., $p = b_0 \oplus b_1 \oplus b_2 \oplus b_3$ If any one bit is reversed it will change the overall parity (number of 1's) from even to odd and thus provide an error indication.

This technique of adding a parity bit to a set of binary words is not peculiar to binary-coded-decimal schemes but is generally applicable. It is common practice to add a parity bit to all information recorded on magnetic tapes.

TABLE 1.5-1 8421 code with parity bit added

Decimal	8	4	2	1	Parity, digit b_3 b_2 b_1 b_0 p
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	0

The 8421 code with a parity bit added is shown plotted on the 5-cube map of Fig.1.5-1. Inspection of this figure shows that the minimum distance between any two words is two as must be true for any single-error-detecting code.

In summary, *any single-error-detecting code must have a minimum distance between any two code words of at least two*, and *any set of binary words with minimum distance between words of at least two can be used as a single-error-detecting code.* Also the addition of a parity bit to any set of binary words will guarantee that the minimum distance between any two words is at least two.

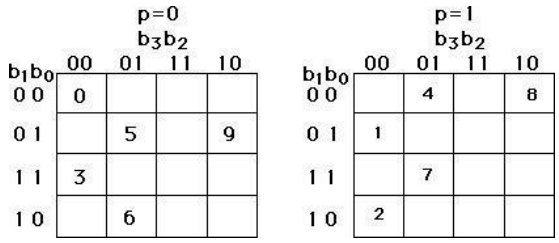


Figure 1.5-1 Five-cube map for the 8421 BCD code with parity bit p

1.5-1 Single-Error-Correcting Codes

A parity check over all the bits of a binary word provides an indication if one of the bits is reversed; however, it provides no information about which bit was changed—all bits enter into the parity check in the same manner. If it is desired to use parity checks to not only detect an altered bit but also to identify the altered bit, it is necessary to resort to several parity checks—each checking a different set of bits in the word. For example, consider the situation in Table 1.5-2 in which there are three bits, M_1, M_2, and M_3, which are to be used to represent eight items of information and there are two parity check bits C_1 and C_2. The information bits, M_i, are often called **message bits** and the C_i bits **check bits**. As indicated in the table C_1 is obtained as a parity check over

TABLE 1.5-2 A parity check table

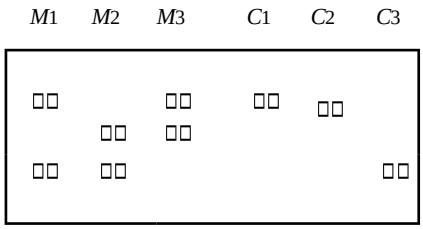
$C_1 = M_1 \square M_3$, $C_2 = M_2 \square M_3$

bits M_1 and M_3, while C_2 checks bits M_2 and M_3.

At first glance it might seem that this scheme might result in a single-error-correcting code since an error in M_3 alters both parity checks while an error in M_1 or M_2 each alters a distinct single parity check. This reasoning overlooks the fact that it is possible to have an error in a check bit as well as an error in a message bit. Parity check one could fail as a result of an error either in message bit M_1 or in check bit C_1. Thus in this situation it would not be clear whether M_1 should be changed or not. In order to obtain a true single-error-correcting code it is necessary to add an additional check bit as in Table 1.5-3.

TABLE 1.5-3 Eight-word single-error-correcting code: (a) Parity check table; (b) parity check equations; (c) Single-error-correcting code

(a)



(b)

$C_1 = M_1 \square M_3$
$C_2 = M_2 \square M_3$
$C_3 = M_1 \square M_2$

(c)

`$M_1 M_2 M_3$ $C_1 C_2 C_3$
a 0 0 0 0 0 0
b 0 0 1 1 1 0
c 0 1 0 0 1 1
d 0 1 1 1 0 1
e 1 0 0 1 0 1
f 1 0 1 0 1 1
g 1 1 0 1 1 0
h 1 1 1 0 0 0

Inspection of the parity check table in Table 1.5-3a shows that an error in any one of the check bits will cause exactly one parity check violation while an error in any one of the message bits will cause violations of a distinct pair of parity checks. Thus it is possible to uniquely identify any single error. The code words of Table 1.5-3c are shown plotted on the 6-cube map of Fig. 1.5-2. Each code word is indicated by the cor-

responding letter and all cells distance 1 away from a code word are marked with an □. The fact that no cell has more than one □ shows that no cell is distance one away from two code words. Since a single error changes a code word into a new word distance one away and each of such words is distance one away from only one code word it is possible to correct all single errors. A necessary consequence of the fact that no word is distance one away from more than one code word is the fact that the minimum distance between any pair of code words is three. In fact the *necessary and sufficient conditions for any set of binary words to be a single-error-correcting code is that the minimum distance between any pair of words be three*.

A single error correcting code can be obtained by any procedure which results in a set of words which are minimum distance three apart. The procedure illustrated in Table 1.5-3 is due to [Hamming 50] and due to its systematic nature is almost univer- sally used for single-error-codes.

With three parity check bits it is possible to obtain a single-error-correcting code of more than eight code words. In fact up to sixteen code words can be obtained. The parity check table for a code with three check bits, C_1, C_2, and C_4, and four message bits M_3 , M_5, M_6 and M_7 is shown in Table 1.5-4. The peculiar numbering of the bits has been adopted to demonstrate the fact that it is possible to make a correspondence between the bit positions and the entries of the parity check table. If the blanks in the table are replaced by 0's and the □ 's by 1's then each column will be a binary number which is the equivalent of the subscript on the corresponding code bit. The check bits are placed in the bit positions corresponding to binary powers since they then enter into only one parity check making the formation of the parity check equations very straight- forward.

The fact that Table 1.5-4 leads to a single-error-correcting code follows from the fact that each code bit enters into a unique set of parity checks. In fact, *the necessary and sufficient conditions for a parity check table to correspond to a single-error-correct- ing code are that each column of the table be distinct (no repeated columns) and that*

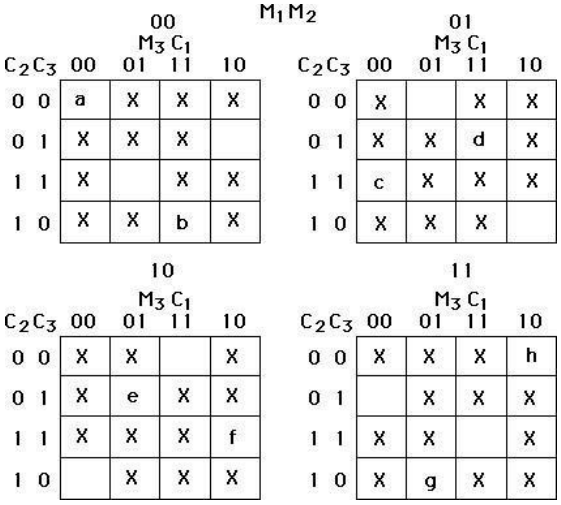
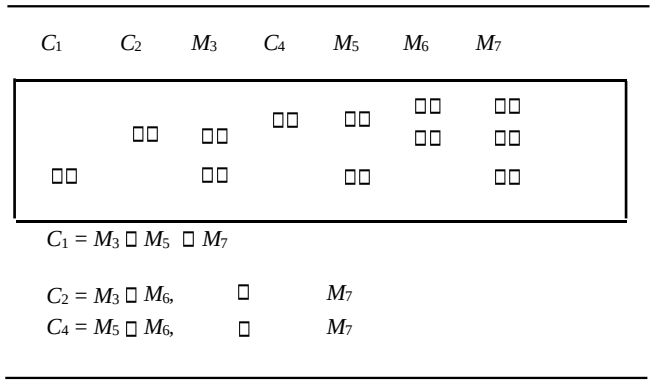


Figure 1.5-2 Six-cube map for the code of Table 1.5-3c.

TABLE 1.5-4 Parity check table for a single-error-correcting code with 3 check bits and 4 message bits



each column contain at least one entry. It follows from this that with K check bits it is possible to obtain a single-error-correcting code having at most 2^K total bits.[1] There are 2^K different columns possible but the empty column must be excluded leaving 2^K columns.

1.5-2 Double-Error-Detecting Codes

If a code such as that generated by Table 1.5-4 is being used and a double error occurs, a correction will be carried out but the wrong code word will be produced. For exam- ple, suppose that bits C_1 and C_2 were in error, the first two parity checks would be vio- lated and it would appear as if message bit M_3 had been in error. Similarly, errors in bit M_3 and M_6 would result in violations of the first and third parity checks,[2] and an indication of M_5 being in error would be produced. It is possible to add the ability to **detect** double errors as well as correct single errors by means of one addition parity check over all the bits. This is illustrated in Table 1.5-5. Any single error in the resulting code will result in the same parity check violations as without P and in addition will violate the P parity check. Any double error will **not** violate the P parity check but will violate some of the C parity checks thus providing an indication of the double error.

A code that detects double errors as well as correcting single errors must consist of binary words having a minimum distance of four. This situation is illustrated by Fig.1.5-3. Both the single-error codes and the double-error-detecting codes are in use in contemporary systems [Hsiao 70]. Many more sophisticated error-correcting codes have been studied ([Peterson 72], [Berlekamp 68]).

[1] In Table 1.5-4, K=3, $2^K - 1 = 7$ and the table does indeed have a total of 7 bits. [2] The two changes in parity check two would cancel.

TABLE 1.5-5 Parity check table for a code to detect all double errors and correct all single errors

C_1	C_2	M_3	C_4	M_5	M_6	M_7	P
	□□	□□	□□	□□	□□ □□	□□ □□	
□□		□□		□□		□□	
□□	□□	□□	□□	□□	□□	□□	□□

$C_1 = M_3 \square M_5 \square M_7$

$C_2 = M_3 \square M_6, \quad \square M_7$

$C_4 = M_5 \square M_6, \quad \square M_7$

$P = C_1 \square C_2 \square M_3 \square C_4 \square M_5 \square M_6 \square M_7$

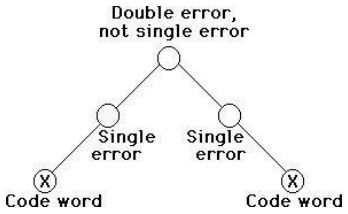


Figure 1.5-3 Fragment of an *N*-cube illustrating the distance between code words in a double-error-detecting, single- error-correcting code.

REFERENCES

[BERLEKAMP 68] Berlekamp, E.R., *Algebraic Coding Theory*, McGraw-Hill Book Company, New York 1968.

[BURKS 62] Burks, A.W., H.H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," *Datamation*, Section 6.5, pp. 39-40, October 1962.

[CHRYSTAL 61] Chrystal, G., *Algebra; an Elementary Text-book* , pt.I, Dover Publications, Inc., New York, 1961.

[DICKINSON 64] Dickinson, M.M., J.B. Jackson, and G.C. Randa, "Saturn V Launch Vehicle Digital Computer and Data Adapter," *Proc., AFIPS Fall Joint Computer Conf.*,Vol.26, Part 1, pp.501- 516, 1964.

[GSCHWIND 75] Gschwind, H.W., and E.J. McCluskey, *Design of Digital Computers*, Springer- Verlag, New York, 1975.

[HAMMING 50] Hamming, R.W., "Error Detecting and Error Correcting Codes," *BSTJ* , Vol.29, pp.147160, April 1950.

[HSIAO 70] Hsiao, M.Y., "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes, *IBM J. Res. and Devel*.,Vol.14, No.4, pp.395-401, July 1970.

[HWANG 78] Hwang, K., *Computer Arithmetic: Principles, Architecture, and Design*, J. Wiley and Sons, Inc., New York, 1978.

[KEISTER 64] Keister, W., R.W. Ketchledge and H.E. Vaughan, "No. 1 ESS: System Organization and Objectives," *Bell Syst. Tech. J*,Vol.43, No.5, Sec. 4, pp. 1841-1842, Sept. 1964.

[KNUTH 68] D.E. Knuth, *Fundamental Algorithms*, Addison-Wesley Publishing Company, Inc., Reading, Mass., 1968.

[KNUTH 69] D.E. Knuth, *Seminumerical Algorithms*, Addison-Wesley Publishing Company, Inc., Reading, Mass., 1969.

[LYONS 62] Lyons, R.E., and W. Vanderkulk, "The Use of Triple-modular Redundancy to Improve Computer Reliability," *IBM J. Res. and Devel* ., Vol. 6, No. 2, pp. 200-209, 1962.

[PERRY 61] Perry, M.N., and W.R. Plugge, "American Airlines SABRE Electronics Reservations System," *Proc., 9th Western Joint Computer Conf*., Los Angeles, CA, pp. 593, May 1961.

[PETERSON 72] Peterson, W.W., and E.J. Weldon, Jr., *Error-correctingCodes*, The MIT Press, Cambridge, MA., 2nd Ed., John Wiley & Sons, Inc., New York, 1972.

[VON NEUMANN 56] von Neumann, J., "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Automata Studies*, C.E. Shannon and J. McCarthy, eds., Annals of Math Studies No. 34, pp. 43-98, Princeton University Press, 1956.

[WASER 82] Waser, S., and M.J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, Holt, Rinehart and Winston, New York, 1982.

[WHITE 53] White, G.S., "Coded Decimal Number Systems for Digital Computers," *Proc., IRE*, Vol.41, No.10, pp. 1450-1452, October, 1953.

PROBLEMS

Convert:

(a) $(523.1)_{10}$ to base 8 **(e)** $(1100.11)_2$ to base 7

(b) $(523.1)_{10}$ to base 2 **(f)** $(101.11)_2$ to base 4 **(c)** $(101.11)_2$ to base 8 **(g)** $(321.40)_6$ to base 7

(d) $(101.11)_2$ to base 10 **(h)** $(25/3)_{10}$ to base 2

In base 10 the highest number which can be obtained by multiplying together two single digits is $9 \times 9 = 81$, which can be expressed with two digits. What is the maximum number of digits required to express the product of two single digits in an arbitrary base-b system?

Given that $(79)_{10} = (142)_b$, determine the value of b.

Given that $(301)_b = (I^2)_b$, where I is an integer in base b and I^2 is its square, determine the value of b.

Let

$$N^* = (n_4\, n_3 n_2 n_1 n_0)^* = 2 \times 3 \times 4 \times 5 \times n_4 + 3 \times 4 \times 5 \times n_3 + 4 \times 5 \times n_2 + 5 \times n_1 + n_0$$

$$= 120n_4 + 60n_3 + 20n_2 + 5n_1 + n_0 \text{ where}$$

$0 \le n_0 \le 4 \quad 0 \le n_1 \le 3 \quad 0 \le n_2 \le 2 \quad 0 \le n_3 \le 1 \quad 0 \le n_4 \le 1$ with all the ni positive integers.

(a) Convert (11111)* to base 10.

(b) Convert (11234)* to base 10.

(c) Convert $(97)_{10}$ to its equivalent $(n_4\, n_3 n_2 n_1 n_0)^*$.

(d) Which decimal numbers can be expressed in the form $(n_4\, n_3 n_2 n_1 n_0\,)^*$?

In order to write a number in base 16 the following symbols will be used for the numbers from 10 to 15:

10 *t* 12 *w* 14 *u*

(a) Convert 11 *e* 13 *h* 15 *f* $(4tu)_{16}$ to base 10.

(b) Convert $(2tfu)_{16}$ to base 2 directly (without first converting to base 10).

Convert $(1222)_3$ to base 5, $(N)_5$, using only binary arithmetic:

(a) Convert $(1222)_3$ to $(N)_2$. **(b)** Convert ($N)_2$ to $(N)_5$.

Perform the following binary-arithmetic operations: **(a)** 11.10 + 10.11 + 111.00 + 110.11 + 001.01 = ?

(b) 111.00 □011.11 = ?

(c) 011.11 □111.00 = ?

(d) 111.001 □1001.1 = ?

(e) 101011.1 + 1101.11 = ?

Form the radix complement and the diminished radix complement for each of the following numbers: **(a)** $(.10111)_2$

(b) $(.110011)_2$

(c) $(0.5231)_{10}$

(d) $(0.32499)_{10}$

(e) $(0.3214)_6$

(f) $(032456)_7$

(a) Write out the following weighted decimal codes:

(i) 7, 4, 2, □1

(ii) 8, 4, □2, □1

(iii) 4, 4, 1, □2

(iv) 7, 5, 3, □6

(v) 8, 7, □4, □2

(b) Which codes of part (a) are self-complementing?

(c) If a weighted binary-coded-decimal code is self-complementing, what necessary condition is placed on the sum of the weights?

(d) Is the condition of part (c) sufficient to guarantee the self-complementing property? Give an example to justify your answer.

Write out the following weighted decimal codes: (7, 3, 1, □2), (8, 4, □3, □2), (6, 2, 2, 1). Which of these, if any, are self-complementing?

Sketch a 4-cube, and label the points. List the points in the *p*-subcubes for p=2,3.

Compute all the pairwise distances for the points in a 3-cube. Arrange these in a matrix form where the rows and columns are numbered 0,1,...,7, corresponding to the points of the 3-cube. The 0-, 1-, and 2-cube pairwise distances are given by submatrices of this matrix. By observing the relationship between these matrices, what is a scheme for going from the *n*-cube pairwise-distance matrix to the (*n*+1)- cube pairwise-distance matrix?

What is a scheme for going from the Gray code to the ordinary binary code using addition mod 2 only?

For the Gray code, a weighting scheme exists in which the weights associated with the bits are constant except for sign.The signs alternate with the occurrence of 1's,

left to right. What is the weighting scheme?

List the symmetries of the 2-cube.

Write out a typical type-6 closed-unit-distance 4 code (Table 1.4-3).

Write out two open unit-distance 4 codes of different type (i.e., one is not a symmetry of the other).

Write out a set of six code words which have and single-error-correcting property.

A closed error-detecting unit-distance code is defined as follows: There are k ($k<2^n$) ordered binary n-bit code words with the property that changing a single bit in any word will change the original word into either its predecessor or its successor in the list (the first word is considered the successor for the last word) or into some other n-bit word **not** in the code. Changing a single bit cannot transform a code word into any code word other than its predecessor or successor. List the code word for such a code with $k = 6$, $n = 3$. Is there more than one symmetry type of code for these specifications? Why?

CHAPTER 2 PREVIEW

- Counting in Decimal and Binary
- Place Value
- Binary to Decimal Conversion
- Decimal to Binary Conversion
- Electronic Translators
- Hexadecimal Numbers
- Octal Numbers

COUNTING IN DECIMAL AND BINARY

- **Number System -**

  Code using symbols that refer to a number of items.

- **Decimal Number System - Uses ten symbols (base 10 system)**

- **Binary System -**

  Uses two symbols (base 2 system)

PLACE VALUE

- **Numeric value of symbols in different positions.**
- ***Example* - Place value in binary system:**

Place Value	**8s**	**4s**	**2s**	**1s**
Binary	**Yes**	**Yes**	**No**	**No**
Number	**1**	**1**	**0**	**0**

RESULT: Binary 1 00 = decimal 8 + 4 + 0 + 0 = decimal 12

BINARY TO DECIMAL CONVERSION

Convert Binary Number 110011 to a Decimal Number:

Binary	1	1	0	0	1	1

Decimal $32 + 16 + 0 + 0 + 2 + 1 =$ **51**

Convert the following binary numbers into decimal numbers:

Binary 1001 = 9

Binary 1 1 = 15

Binary 0010 = 2

DECIMAL TO BINARY CONVERSION

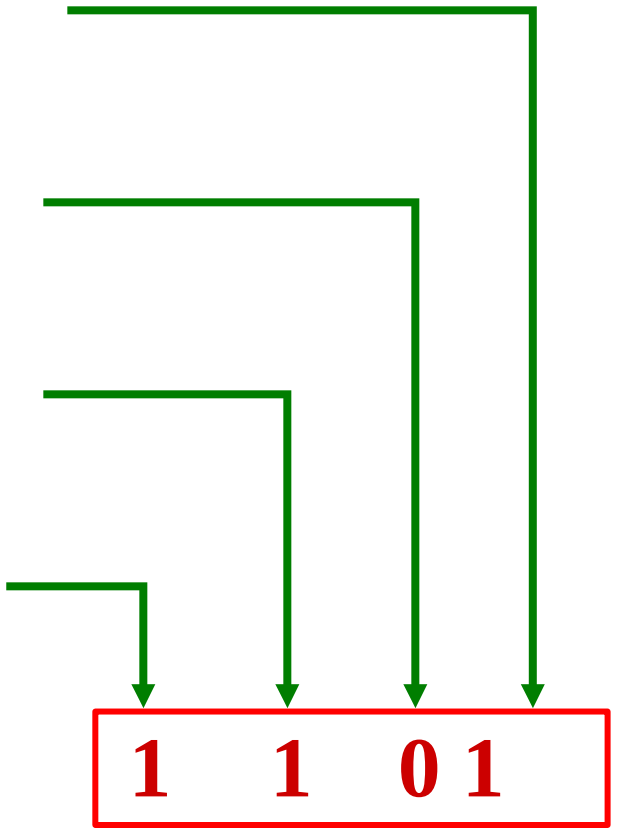
Divide by 2 Process

Decimal # 13 ÷ 2 = 6 remainder 1

6 ÷ 2 = 3 remainder 0

3 ÷ 2 = 1 remainder 1

1 ÷ 2 = 0 remainder 1



Convert the following decimal numbers into binary:

Decimal 1 = 101

Decimal 4 = 0100

Decimal 17 = 10001

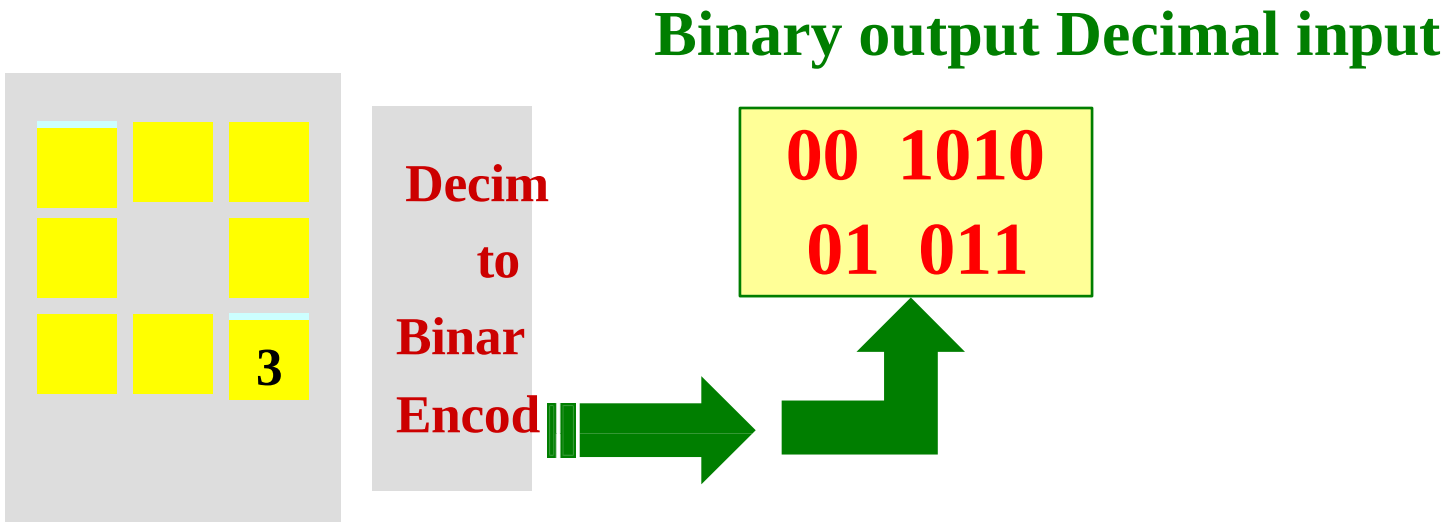
ELECTRONIC TRANSLATORS

Devices that convert from decimal to binary numbers and from binary to decimal numbers.

Encoders - translates from decimal to binary

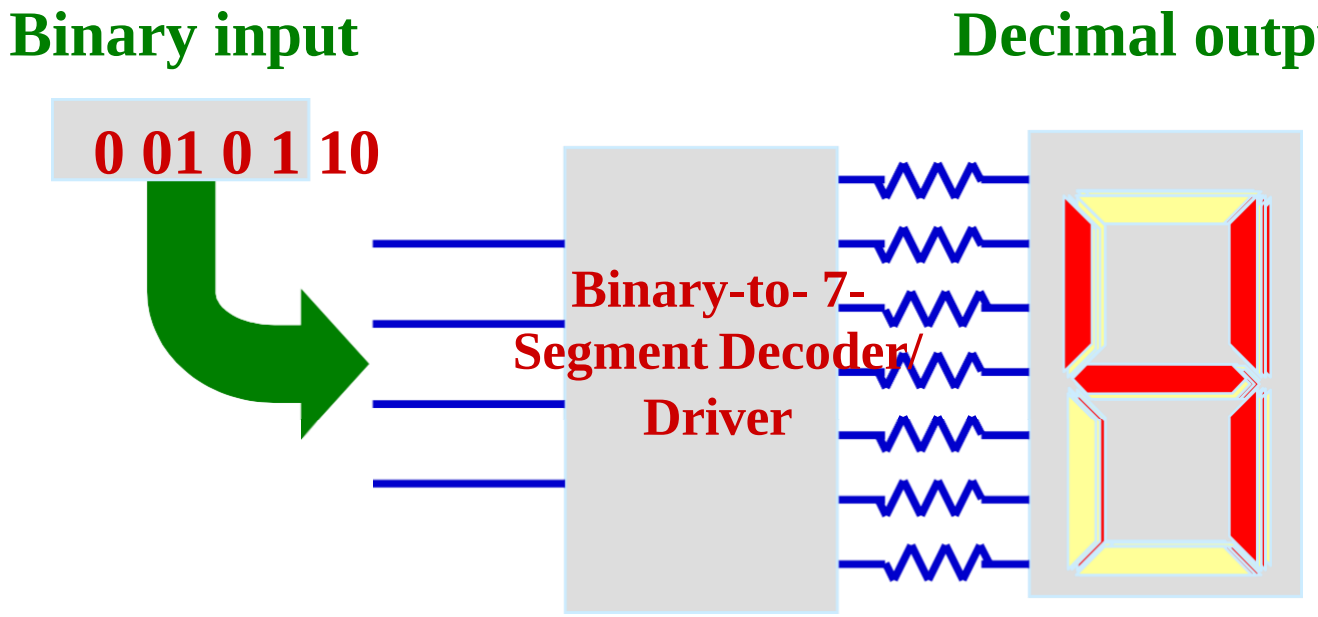
Decoders - translates from binary to decimal

ELECTRONIC ENCODER - DECIMAL TO BINARY



- **Encoders are available in IC form.**
- **This encoder translates from decimal input to binary (BCD) output.**

ELECTRONIC DECODING: BINARY TO DECIMAL



- Electronic decoders are available in IC form.
- This decoder translates from binary to decimal.
- Decimals are shown on an 7-segment LED display.

- **This decoder also drives the 7-segment display.**

HEXADECIMAL NUMBER SYSTEM

Uses 16 symbols -Base 16 System

0-9, A, B, C, D, E, F

Decimal	Binary	Hexadecimal
1	0001	1
9	1001	9
10	1010	A

15 **1 1** **F**

16 **10000** **10**

HEXADECIMAL AND BINARY CONVERSIONS

- **Hexadecimal to Binary Conversion**

Hexadecimal	**C**	**3**
	↓	↓
Binary	**1100**	**0011**

- **Binary to Hexadecimal Conversion**

Binary	**1 10**	**1010**

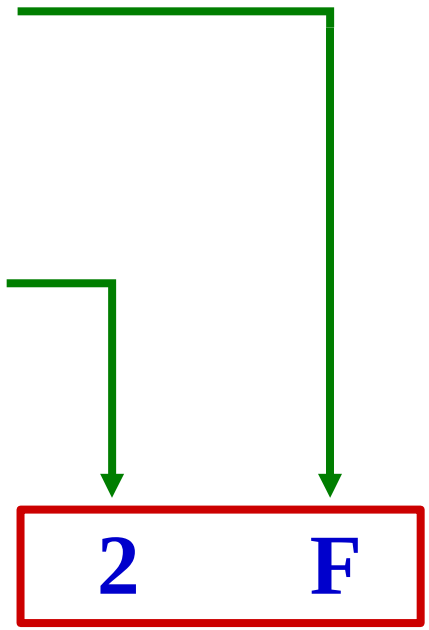
Hexadecimal E A

DECIMAL TO HEXADECIMAL CONVERSION

Divide by 16 Process

Decimal # **47 ÷ 16 = 2 remainder 15**

2 ÷ 16 = 0 remainder 2



HEXADECIMAL TO DECIMAL CONVERSION

Convert hexadecimal number
2DB to a decimal number

Place Value 256s 16s 1s

Hexadecimal	**2**		**D**		**B**		
	(256 x 2)		**(16 x 13)**		**(1 x 11)**		
Decimal	**512**	**+**	**208**	**+**	**1**	**=**	**731**

TEST

Convert Hexadecimal number A6 to Binary

A6 = 1010 0110 (Binary)

Convert Hexadecimal number 16 to Decimal

16 = 22 (Decimal)

Convert Decimal 63 to Hexadecimal

63 = 3F (Hexadecimal)

OCTAL NUMBERS

Uses 8 symbols -Base 8 System

0, 1, 2, 3, 4, 5, 6, 7

	Decimal	Binary	Octal
	1	001	1
6	1 0 6		
7	1 1 7		
8	001 000	10	
9	001 001	1	

PRACTICAL SUGGESTION ON NUMBER SYSTEM CONVERSIONS

- Use a scientific calculator

- **Most scientific calculators have DEC, BIN, OCT, and HEX modes and can either convert between codes or perform arithmetic in different number systems.**
- **Most scientific calculators also have other functions that are valuable in digital electronics such as AND, OR, NOT, XOR, and XNOR logic functions.**

ECE _ Digital Electronics

Basic Logic Operations,
Boolean Expressions,
and
Boolean Algebra

Basic Logic Operations

Basic Logic Operations

- AND
- OR
- NOT (Complement)

- Order of Precedence
    1. NOT
    2. AND
    3. OR
    - can be modified using parenthesis

Basic Logic Operations

Table 1.8
Truth Tables of Logical Operations

AND

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

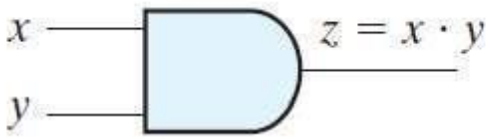
OR

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

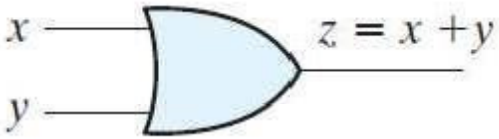
NOT

x	x'
0	1
1	0

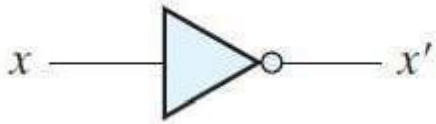
Basic Logic Operations



(a) Two-input AND gate



(b) Two-input OR gate

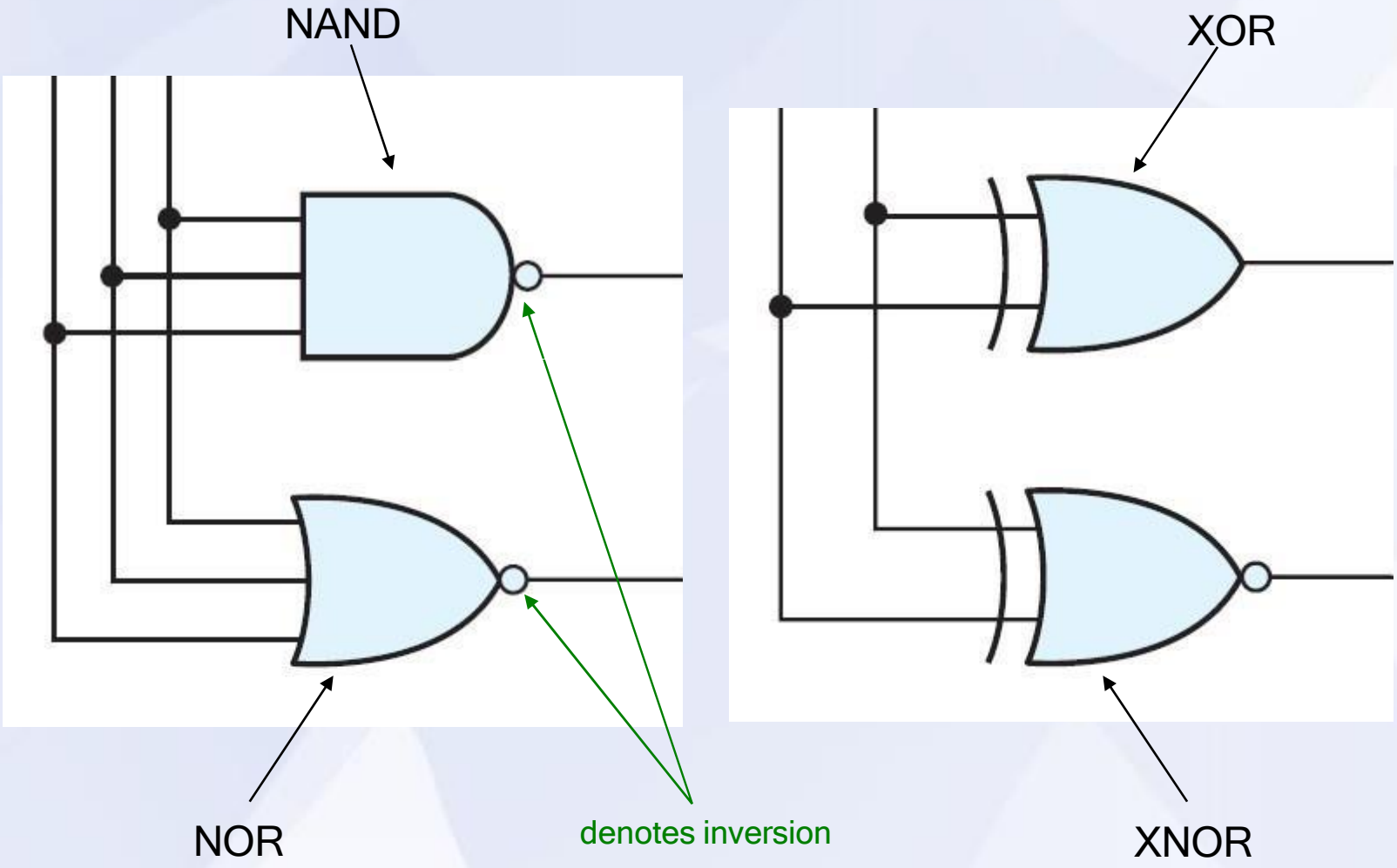


(c) NOT gate or inverter

Additional Logic Operations

- NAND
  - F = (A . B)'
- NOR
  - F = (A + B)'
- XOR
  - Output is 1 iff either input is 1, but not both.
- XNOR (aka. Equivalence)
  - Output is 1 iff bot inputs are 1 or both inputs are 0.

Additional Logic Operations



Additional Logic Operations

Exer ise:

Derive the Truth ta le for each of the following logi operations:

1. 2-input NAND
2. 2-input NOR

Additional Logic Operations

Exer ise:

Derive the Truth ta le for each of the following logi operations:

1. 2-input XOR
2. 2-input XNOR

Truth ables

Truth T ables

- Used to describe the functional behavior of a Boolean expression and/or Logic circuit.

- Each row in the truth table represents a unique combination of the input variables.

For n input variables there are 2^n rows.

- The output of the logic function is defined for each row.

- Each row is assigned a numerical value, with the rows listed in ascending order.
- The order of the input variables defined in the logic function is important.

3- input Truth Table

F(A,B,C) = Boolean expression

4- input Truth Table

F(A,B,C,D) = Boolean expression

Boolean E pressions

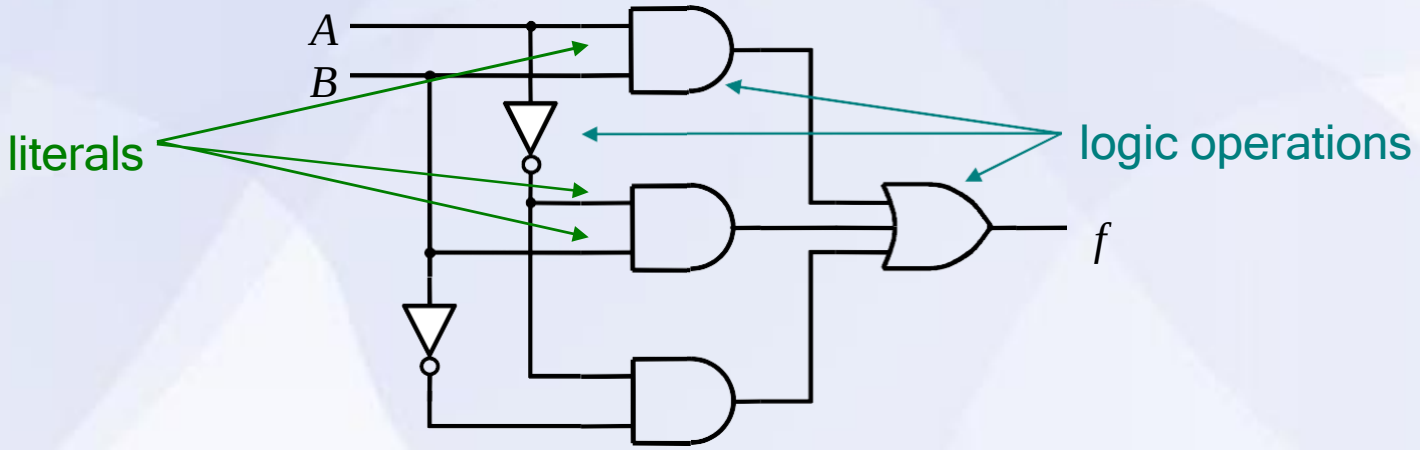
Boolean Expressions

- Boolean expressions are composed of
  - Literals – variables and their complements
  - Logical operations
- Examples
  - F = A.B'.C + A'.B. ' + A.B.C + A'.B'.C'

  - F = (A+B+C').(A'+ '+C).(A+B+C)
  - F = A.B'.C' + A.(B.C' + B'.C)

Boolean Expressions

- Boolean expressions are realized using a network (or combination) of logic gates.
  - Each logic gate i plements one of the logic operations in the[Boolean expression
  - Each input to a lo ic gate represents one of the literals in the[oolean expression



Boolean Expressions

- Boolean expressions are evaluated by
- Substituting a 0 or 1 for each literal
- Calculating the logical value of the expression

  - A Truth Table specifies the value of the Boolean expression for every combination of the variables in the Boolean expression.

  - For an n-variable Boolean expression, the truth table has 2^n rows (one for each combination).

Boolean Expressions

Example:

Evaluate the following Boolean expression, for all combination of inputs, using a Truth table.

F(A,B,C) = A'.B'.C' + A.B'.C' + A.C

Boolean E pressions

- Two Boolean expressi ns are equivalent if they have the same value for each combination of the variables in the Bo lean expression.
  - $F_1 = (A + B)'$
  - $F_2 = A'.B'$
- How do you prove that two Boolean expressions are equivalent?
  - Truth table
  - Boolean Algebra

Boolean E pressions

E a ple:

Using a Truth table, prove that the following two Boolean expr є sior s are equivalent.

$F_1 = (A + B)'$

$F_2 = A'.B'$

Boolean Algebra

Boolean Algebra

- George Boole developed an algebraic description for processes involving logical thought and reasoning.
  - Became known as Boolean Algebra
- Claude Shannon later dem onstrated that Boolean Algebra could be used to escribe switching circuits.
  - Switching circuits are circuits built from devices that switch between two states (e.g. 0 and 1).
  - Switching Algebra is a special case of Boolean Algebra in which all v ariables take on just two distinct values

- Boolean Algebra is a powerful tool for analyzing and designing logic circuits.

Basic Laws and Theorems

Commutative Law	A + B = B + A	A.B = B.A
Associative Law	A + (B + C) = (A + B) + C	A . (B . C) = (A . B) . C
Distributive Law	A.(B + C) = AB + AC	A + (B . C) = (A + B) . (A + C)
Null Elements	A + 1 = 1	A . 0 = 0
Identity	A + 0 = A	A . 1 = A
Idempotence	A + A = A	A . A = A
Complement	A + A' = 1	A . A' = 0
Involution	A'' = A	
Absorption (Covering)	A + AB = A	A . (A + B) = A
Simplification	A + A'B = A + B	A . (A' + B) = A . B
DeMorgan's Rule	(A + B)' = A'.B'	(A . B)' = A' + B'
Logic Adjacency (Combining)	AB + AB' = A	(A + B) . (A + B') = A
Consensus	AB + BC + A'C = AB + A'C	(A + B) . (B + C) . (A' + C) = (A + B) . (A' + C)

Idemp tence

A + = A

F = ABC + A C' + ABC

F = AB + ABC'

Note: terms can also be ac de using this theorem

A . = A

G = (A' + B + C').(A + B' + C).(A + B' + C)

G = (A' + B + C') + (A + B' + C)

Note: terms can also be added using this theorem

Complement

A + ' = 1

F = ABC'D + ABCD

F = ABD.(C' + C)

F = ABD

A . A' = 0

G = (A + B + C + D).(A + B' + C + D)

G = (A + C + D) + (B . B')

G = A C + D

Distributive Law

$A.(B + C) = AB + AC$

$F = WX.(Y + Z)$

$F = WXY + WXZ$

$G = B'.(AC + AD)$

$G = AB'C + AB'D$

$H = A.(W'X + WX' + YZ)$

$H = AW'X + AWX' + AYZ$

$A + (B.C) = (A + B).(A + C)$

$F = WX + (Y.Z)$

$F = (WX + Y).(WX + Z)$

$G = B' + (A.C.D)$

$G = (B' + A).(B' + C).(B' + D)$

$H = A + ((W'X).(WX'))$

$H = (A + W'X).(A + WX')$

Absorption (Covering)

$A + AB = A$

$A.(A + B) = A$

$F = A'BC + A'$

$F = A'$

$F = A'.(A' + BC)$

$F = A'$

$G = XYZ + XY'Z + X'Y'Z' + XZ$

$G = XYZ + XZ + X'Y'Z'$

$G = XZ + X'Y'Z'$

$G = XZ.(XZ + Y + Y')$

$G = XZ.(XZ + Y)$

$G = XZ$

$H = D + DE + DEF$

$H = D$

$H = D.(D + E + EF)$

$H = D$

Simplification

$$A + A'B = A + B$$

$$F = (XY + Z).(Y'W + Z'V') + (XY + Z)'$$

$$F = Y'W + Z'V' + (XY + Z)'$$

$$A.(A' + \quad) = A \cdot B$$

$$G = (X + Y).(\ (\quad + Y)' + (WZ)\ )$$

$$G = (X + Y) \cdot WZ$$

Logic Adjacen y (Combining)

$$A.B + \quad .B' = A$$

$$F = (X + Y).(W'X'Z) + (X + Y).(W'X'Z)'$$

$$F = (\quad + Y)$$

$$(A + B).(A + B') = A$$

$$G = (XY + X'Z').(XY + (X'Z')')$$

$$G = XY$$

Boolean Algebra

Exa ple:

Using Boolean Algebra, simplify the following Boolean e pression.

$$F(A,B,C) = A'.B.C + A.B'.C + A.B.C$$

Boolean Algebra

Exa ple:

Using Boolean Algebra, simplify the following Boolean e pression.

F(A,B,C) = (A'+B'+C').(A'+B+C').(A+B'+C')

DeMorgan's Laws

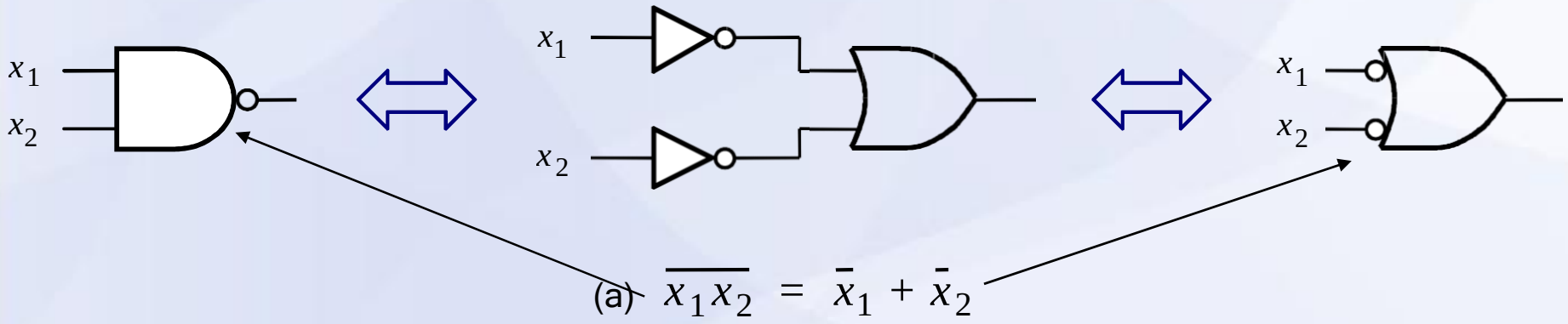
- Can be stated as follo s:
  - The complement of the product (AND) is the sum (OR) of the complements.
    - (X.Y)' = X' + Y'
  - The complement of the sum (OR) is the product (AND) of the complements.
    - (X + Y)' = X' . Y'
- Easily generalized to n variables.
- Can be proven using a Truth table

Proving DeMorgan's Law

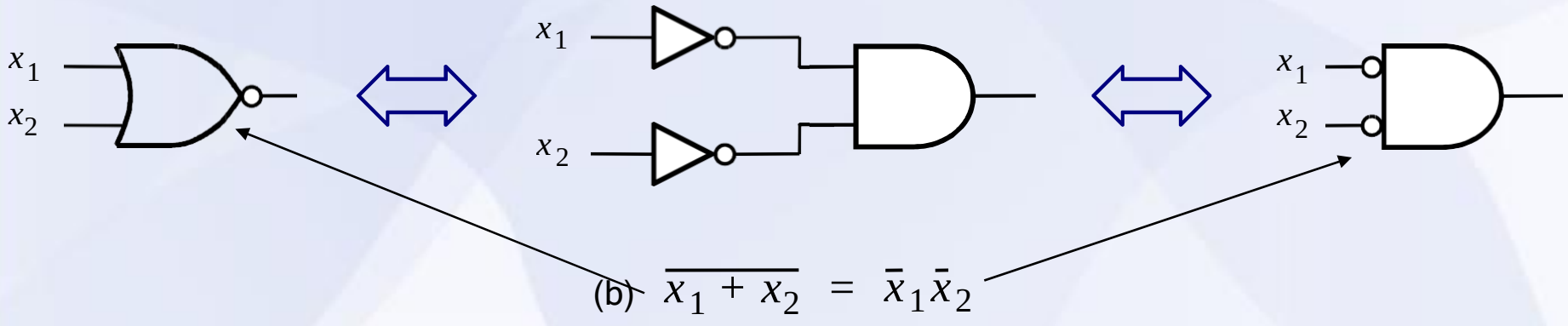
$$(X \cdot Y)' = X' + Y'$$

x	y	$x \cdot y$	$\overline{x \cdot y}$	$\overline{x}$	$\overline{y}$	$\overline{x} + \overline{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0
		LHS		RHS		

DeMorgan's Theorems



(a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$



(b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

Importance of Boolean Algebra

- Boolean Algebra is used to simplify Boolean expressions.
  - Through application of the Laws and Theorems discussed
- Simpler expressions lead to simpler circuit realiz tion, which, generally, reduces ost, area requirements, and power consumption.
- The objective of the digital circuit designer is to design and realize optimal digital ircuits.

Algebraic Simplification

- Justification for simplifying Boolean expressions:
  - Reduces the cost associated with realizing the expression using logic gates.
  - Reduces the area (i.e. silicon) required to fabricate the switching function.
  - Reduces the power c onsumption of the circuit.

- In general, there is no easy way to determine when a Boolean expression has been simplified to a minimum number of terms or minimum number of literals.
  - No unique solution

Algebraic Simplification

- Boolean (or Switching) expressions can be simplified using the following methods:

1. Multiplying out th e expression
2. Factoring the exp r ession

3. Combining terms of the expression
4. Eliminating terms in the expression
5. Eliminating literals in the expression
6. Adding redundant terms to the expression

As we shall see, there are other tools that can be used to simplify Boolean Expressions. Namely, **Karnaugh Maps**.

Digital Systems: Combinational Logic C rcuits

Objectives

- Convert a logic expression into a sum-of-products expression.
- Perform the necessary steps to reduce a sum-of-

products expression to its simplest form.

- Use Boolean algebra and the Karnaugh map as tools to simplify and design logic circuits.
- Explain the operation of both exclusive-OR and exclusive-NOR circuits.
- Design simple logic circuits without the help of a truth table.

Objectives (cont'd)

- Implement enable circuits.
- Cite the basic characteristics of TTL and CMOS digital ICs.

- Use the basic troubleshooting rules of digital systems.
- Deduce from observed results the faults of malfunctioning combinational logic circuits.
- Describe the fundamental idea of programmable logic devices (PLDs).
- Outline the steps involved in programming a PLD to perform a simple combinational logic function

Combinational Logic Circuits

- The logic level at the output depends on the combination of logic levels present at the inputs.

- A combinational circuit has no memory, so its output depends only on the current value of its inputs.
- We will not spend a great deal of time discussing how to troubleshoot the combinational circuits. (That's what the lab is for.)

Sum-of-Products Form

- Sum → OR
- Product → AND
- Each of the sum-of-products expression consists of two or more AND terms that are ORed together.
- Examples: ABC+A'BC'

  AB+A'BC'+C'D'+D
- Note that one inversion sign cannot cover more than one variable in a term. AB is not allowed.

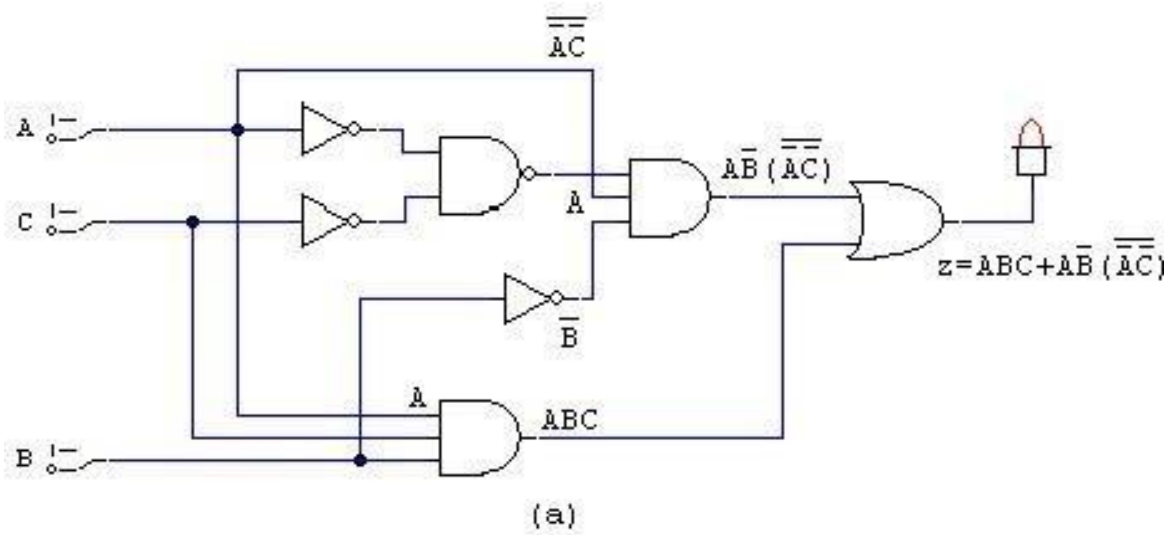
Product-of-Sums Form

- Each of the product-of-sums expression consists of two or more OR terms that are ANDed together.
- Examples: (A+B'+C)(A+C)
  (A+B')(C'+D)F
- Will use sum-of-products form in logic circuit simplification.

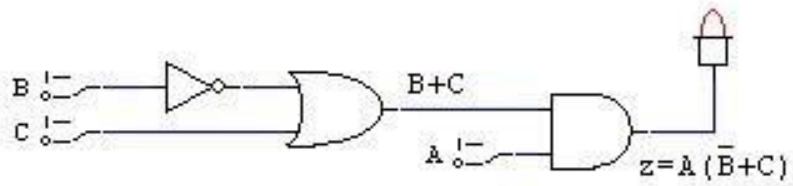
Simplifying Logic Circuits

- Goal: reduce the logic circuit expression to a simpler form so that fewer gates and connections are required to build the circuit.
- Example: 4.1(a) and 4.1(b) are equivalent, but 4-1(b) is much simpler.

Example 4.1



(a)



(b)

Circuit Simplification Methods

- Boolean algebra: greatly depends on inspiration and experience.
- Karnaugh map: systematic, step-by-step approach.
- Pros and Cons

Algebraic Simplification

- Use the Boolean algebra theorems introduced in Chapter 3 to help simplify the expression for a logic circuit.
- Based on experience, often becomes a trial-and-error process.
- No easy way to tell whether a simplified expression is in its simplest form.

Two Essential Steps

- The original expression is put into the sum-of-products form by repeated application of DeMorgan's theorem and multiplication of terms.
- The product terms are checked for common factors, and factoring is performed whenever possible.

Examples 4-1 to 4-4

Original	Simplified
ABC+AB’(A’C’)’	A(B’+C)

ABC+ABC’+AB’C	A(B+C)
A’C(A’BD)’+A’BC’D’+AB’C	B’C+A’D’(B+C)
(A’+B)(A+B+D)D’	BD’

Examples 4-5, 4-6

- (A’+B)(A+B’): equivalent form A’B’+AB
- AB’C+A’BD+C’D’: cannot be simplified further.

Designing Combinational Logic Circuits

1. Set up the truth table.
2. Write the AND term for each case where the output is a 1.
3. Write the sum-of-products expression for the output.
4. Simplify the output expression.
5. Implement the circuit for the final expression.

Example 4-8

- Design a logic circuit that is to produce a HIGH output when the voltage (represented by a four-bit binary number ABCD) is greater than 6V.

Example 4-9

- Generate the STOP signal and energize an indicator light whenever either of the following conditions exists: (1) there is no paper in the paper feeder tray; or (2) the two micro-switches in the paper path are activated, indicating a jam.

Karnaugh Map Method

- A graphical device to simplify a logic expression.
- Will only work on examples with up to 4 input variables.
- From truth table to logic expression to K map.
- Figure 4.11 shows the K map with 2,3 and 4 variables.

Looping

- The expression for output X can be simplified by properly combining those squares in the K map which contain 1s. The process of combining these 1s is

called **looping**.

- Looping groups of two (pairs) → eliminate 1 variable
- Looping groups of four (quads) → eliminate 2 variables
- Looping groups of eight (octets)→ eliminate 3 variables
- See Figure 4-12 to 4-14.

Complete Simplification Process

- Step 1: Construct the K map and places 1s in those squares corresponding to the 1s in the truth table. Places 0s in the other squares.
- Step 2: Examine the map for adjacent 1s and loop those 1s which are not adjacent to any other 1s. (isolated 1s)
- Step 3: Look for those 1s which are adjacent to only one other 1. Loop any pair containing such a 1.
- Step 4: Loop any octet even when it contains some 1s that have already been looped.

Complete Simplification Process

- Step 5: Loop any quad that contains one or more 1s that have not already been looped, making sure to use the minimum number of

loops.

- Step 6: Loop any pairs necessary to include any 1s have not already been looped, making sure to use the minimum number of loops.
- Step 7: Form the ORed sum of all the terms generated by each loop.

Filling K Map from Output Expression

- What to do when the desired output is presented as a Boolean expression instead of a truth table?

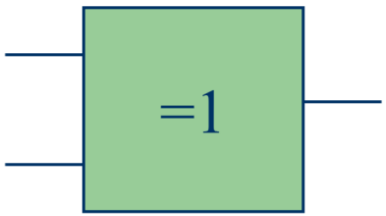
- Step 1: Convert the expression into SOP form.
- Step 2: For each product term in the SOP expression, place a 1 in each K-map square whose label contains the same combination of input values. Place a 0 in other squares.
- Example 4-14: y=C’(A’B’D’+D)+AB’C+D’

Don’t-Care Conditions

- Some logic circuits can be designed so that there are certain input conditions for which there are no specified output levels.
- A circuit designer is free to make the output for any don't care condition either a 0 or a 1 in order to produce the simplest output expression.
- Figures 4-18,19.

Exclusive-OR

- Exclusive-OR (XOR)
  x = A'B+AB'
- Timing diagram

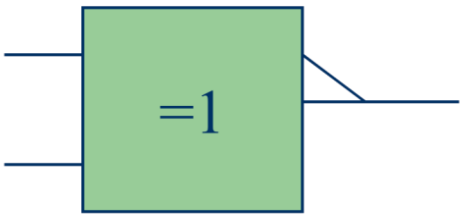


XOR		
A	B	x
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive-NOR

- Exclusive-NOR (XNOR)

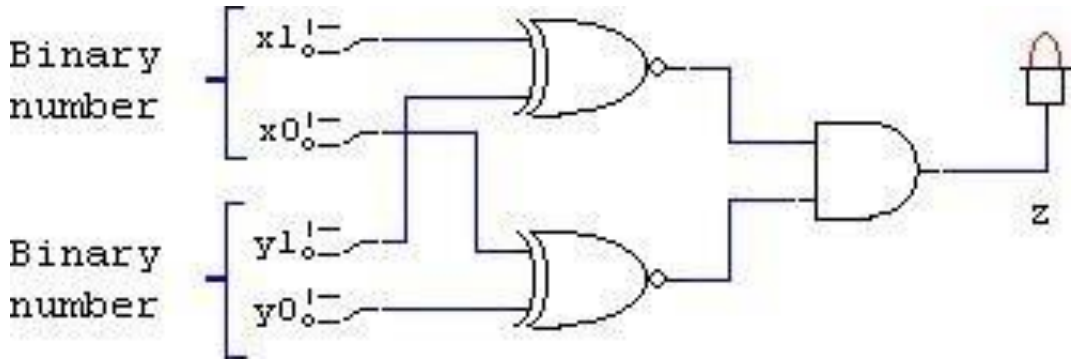
x = (A'B+AB')'



XNOR		
A	B	x
0	0	1
0	1	0
1	0	0
1	1	1

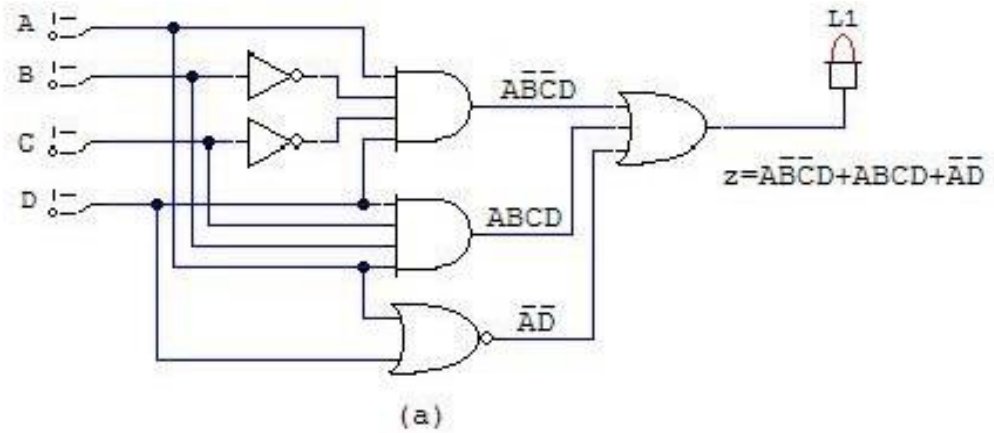
Example 4-17

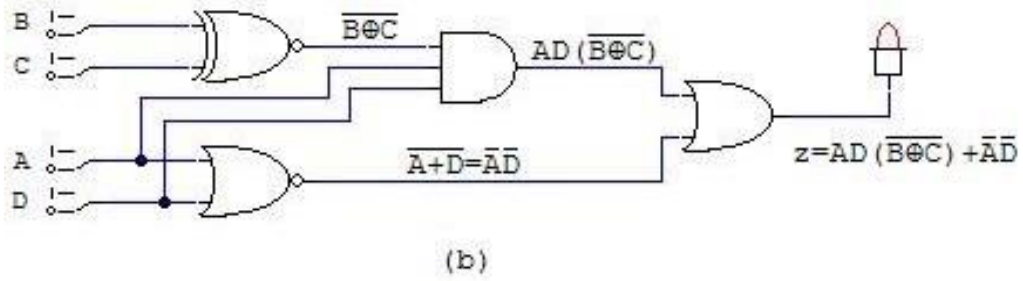
- Design a logic circuit, using x_1, x_0, y_1 and y inputs, whose output will be HIGH only when the two binary numbers x_1x_0 and y_1y_0 are equal.
- Hint: use XNOR gates (Figure 4-23)



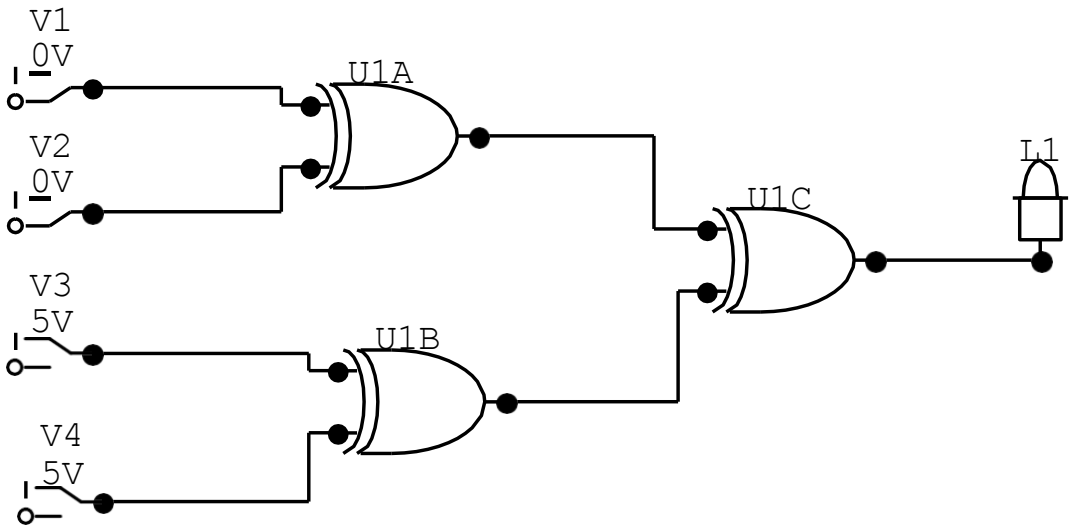
Using XNOR to Simplify Circuit Implementation

- Example 4-18

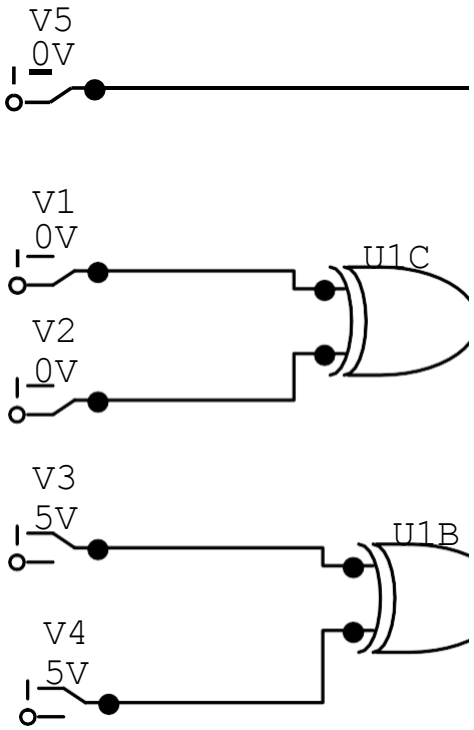




Parity Generator



Even-parity C



Error

Enable/Disable Circuits

- Each of the basic logic gates can be used to control the passage of an input logic signal through to the output.
- A: input, B: control (Figure 4-26)
- The logic level at the control input determines whether the input signal is enabled to reach the output or disabled from reaching the output.

Basic Characteristics of Digital ICs

- Digital ICs are a collection of resistors, diodes and transistor fabricated on a single piece of semiconductor material called a substrate, which is commonly referred to as a chip.
- The chip is enclosed in a package.
- Dual-in-line package (DIP)

Integrated Circuits

Complexity	Number of Gates
Small-scale integration(SSI)	<12
Medium-scale integration(MSI)	12 to 99
Large-scale integration(LSI)	100 to 9999
Very large-scale integration(VLSI)	10,000 to 99,999

Ultra large-scale integration(ULSI)	100,000 to 999,999
Giga-scale integration (GSI)	1,000,000 or more

Bipolar and Unipolar Digital ICs

- Categorized according to the principal type of electronic component used in their circuitry.
- Bipolar ICs are those that are made using the bipolar junction transistor (PNP or NPN).

- Unipolar ICs are those that use the unipolar field-effect transistors (P-channel and N-channel MOSFETs).

IC Families

- TTL Family: bipolar digital ICs (Table 4-6)
- CMOS Family: unipolar digital ICs (Tab le 4-
- TTL and CMOS dominate the field of S I a d MSI devices.

TTL Family

TTL Series	Prefix	Example IC
Standard TTL	74	7404 (hex inverter)
Schottky TTL	74S	74S04

Low-power Schottky TTL	74LS	74LS04
Advanced Schottky TTL	74AS	74AS04
Advanced low- power Schottky TTL	74ALS	74ALS04

CMOS Family

CMOS Series	Prefix	Example IC
Metal-gate CMOS	40	4001
Metal-gate, pin-compatible with TTL	74C	74C02
Silicon-gate, pin-compatible with TTL, high-speed	74HC	74HC02
Silicon-gate, high-speed, pin-compatible and electrically compatible with TTL	74HCT	74HCT02
Advanced-performance CMOS, not pin or electrically compatible with TTL	74AC	74AC02
Advanced-performance CMOS, not pin but electrically compatible with TTL	74ACT	74ACT02

Power and Ground

- To use digital IC, it is necessary to make proper connection to the IC pins.
- Power: labeled V_{cc} for the TTL circuit, labeled V_{DD} for CMOS circuit.
- Ground

Logic-level Voltage Ranges

- For TTL devices, V_{CC} is normally 5V.
- For CMOS circuits, V_{DD} can range from 3-18V.
- For TTL, logic 0 : 0-0,8V, logic 1:2-5V
- For CMOS, logic 0 : 0-1.5V, logic 1:3.5-5V

Unconnected Inputs

- Also called floating inputs.
- A floating TTL input acts like a logic 1, but measures a DC level of between 1.4 and 1.8V.
- A CMOS input cannot be left floating.

Logic-Circuit Connection Diagrams

- A connection diagram shows all electrical connections, pin numbers, IC numbers, component values, signal names, and power supply voltages.
- See Figure 4-32.

Troubleshooting Digital Systems

- Fault detection
- Fault isolation
- Fault correction
- Good troubleshooting techniques can be learned only through experimentation and actual troubleshooting of faulty circuits.

Troubleshooting Tools

- Logic probe
- Oscilloscope
- Logic pulser
- Current tracer
- … and your BRAIN!

Indicator Light	Logic Level
OFF	LOW
ON	HIGH
DIM	INTERMEDIATE
FLASHING	PULSING

Internal IC Faults

- Malfunction is the internal circuitry.
- Inputs or outputs shorted to ground or Vcc (Figure 4.34, 4-35)
- Inputs or outputs open-circuited (Figure 4.36)
- Short between two pins (other than ground or Vcc): whenever two signals that are supposed to be different show the same logic-level variations.

External Faults

- Open signal lines:Broken wire, Poor solder connection, Crack or cut trace on a printed circuit board, Bend or broken pin on a IC, faulty IC socket.

- Shorted signal lines: sloppy wiring, solder bridges, incomplete etching.
- Faulty power supply
- Output loading: when an output is connected to too many IC inputs.

Programmable Logic Device

- PLD is an integrated circuit that contains a particular arrangement of logic gates. (Figure 4.41)
- Useful in implementing complex circuits containing tens or thousands of logic gates.
- Sum-of-products form

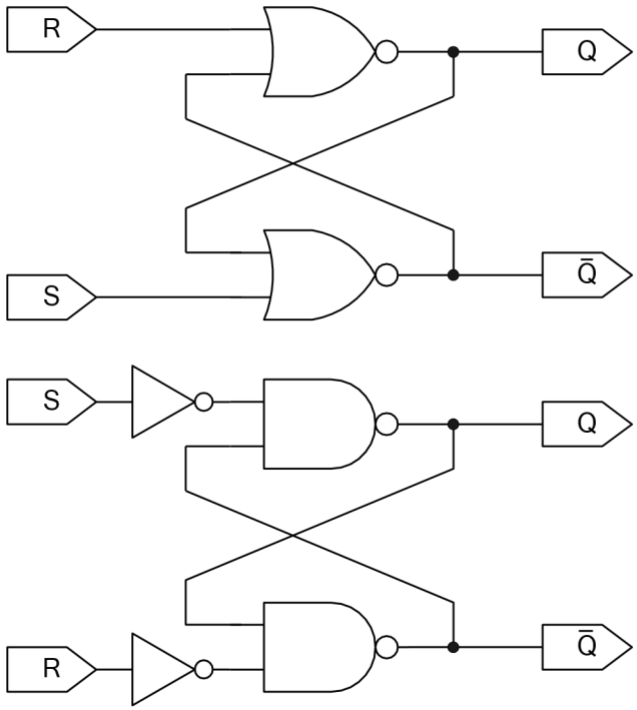
Sequential Digital Circuits

- Sequential circuits are digital circuits in which the outputs depend not only on the current inputs, but also on the previous state of the output.
- They basic sequential circuit elements can be divided in two categories:
- Level-sensitive (Latches) – High-level sensitive
  - Low-level sensitive
- Edge-triggered (Flip-flops)
  - Rising (positive) edge triggered
  - Falling (negative) edge triggered
  - Dual-edge triggered

The Set/Reset (SR) Latch

The Set/Reset latch is the most basic unit of sequential digital circuits. It has two inputs (S and R) and two outputs outputs Q and Q'. The two outputs must always be complementary, i.e if Q is 0 then Q' must be 1, and vice-versa. The S input sets the Q output to a logic 1. The R input resets the Q output to a logic 0.

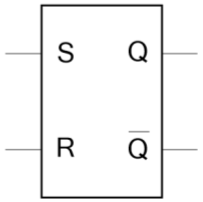
Circuit Diagram



Truth Table

S	R	Q+	Q'+	Function
0	0	Q	Q'	Latch
0	1	0	1	Reset
1	0	1	0	Set
1	1	0	0	Illegal

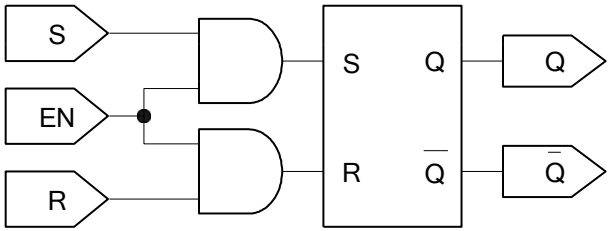
Logic Symbol



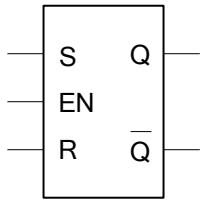
The Gated Set/Reset (SR) Latch

To be able to control when the S and R inputs of the SR latch can be applied to the latch and thus change the outputs, an extra input is used. This input is called the Enable. If the Enable is 0 then the S and R inputs have no effect on the outputs of the SR latch. If the Enable is 1 then the Gated SR latch behaves as a normal SR latch.

Circuit Diagram



Logic Symbol



Truth Table

EN	S	R	Q+
0	0	0	Q
0	0	1	Q
0	1	0	Q
0	1	1	Q
1	0	0	Q
1	0	1	0
1	1	0	1
1	1	1	U

Truth Table

EN	S	R	Q+	Function
0	X	X		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

for Computers

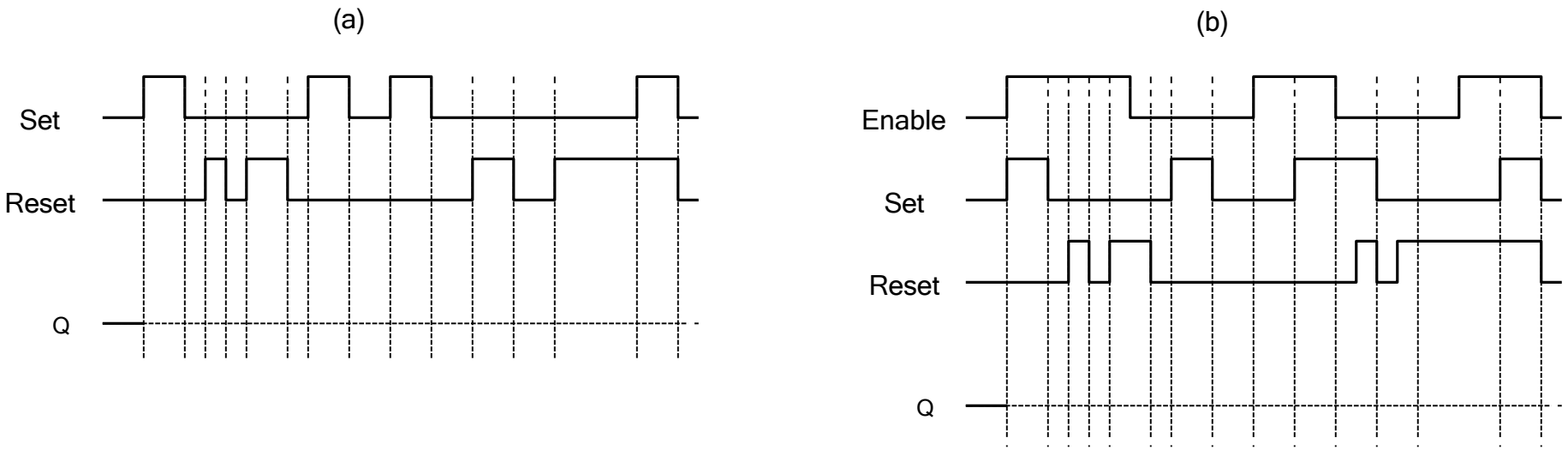
SR Latch :- Example

Complete the timing diagrams for :

(a) Simple SR Latch

(b) SR Latch with Enable input.

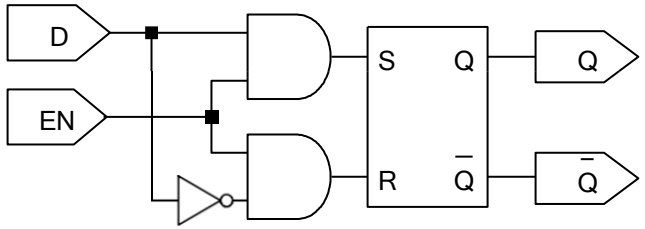
Assume that for both cases the Q output is initially at logic zero.



The Data (D) Latch

A problem with the SR latch is that the S and R inputs can not be at logic 1 at the same time. To ensure that this can not happen, the S and R inputs can by connected

through an inverter. In this case the Q output is always the same as the input, and the latch is called the Data or D latch. The D latch is used in Registers and memory devices.



Circuit Diagram

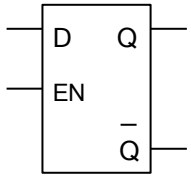
Truth Table

EN	D	Q	Q+
0	0	0	Q
0	0	1	Q
0	1	0	Q
0	1	1	Q
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Truth Table

EN	D	Q+	Function
0	0		
0	1		
1	0		
1	1		

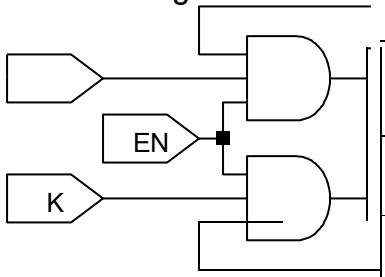
Logic Symbol



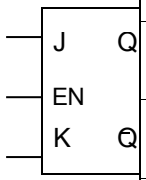
The JK Latch

Another way to ensure that the S and R inputs can not be at logic 1 simultaneously, is to cross connect the Q and Q' outputs with the S and R inputs through AND gates. The latch obtained is called the JK latch. In the J and K inputs are both 1 then the Q output will change state (Toggle) for as long as the Enable 1, thus the output will be unstable. This problem is avoided by ensuring that the Enable is at logic 1 only for a very short time, using edge detection circuits.

Circuit Diagram



Logic S mbol



Truth Table

EN	J	K
0	X	X
1	0	0
1	0	0
1	0	1
1	0	1
1	1	0
1	1	0
1	1	1
1	1	1

Truth Table

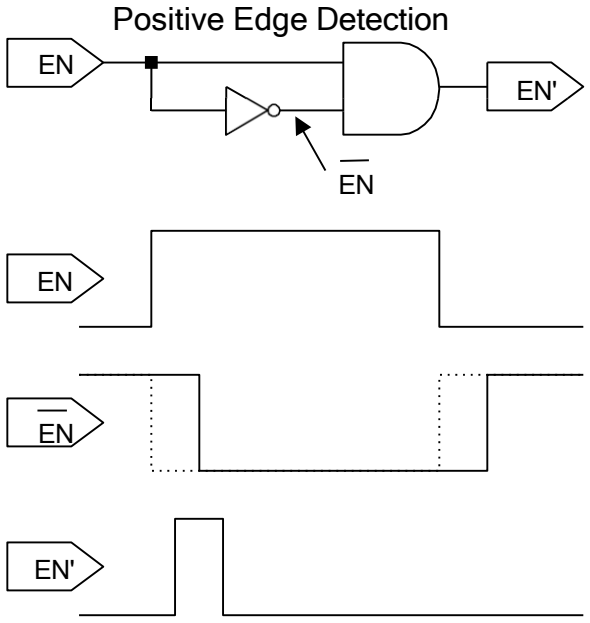
EN	J	K	Q+	Function
0	X	X		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

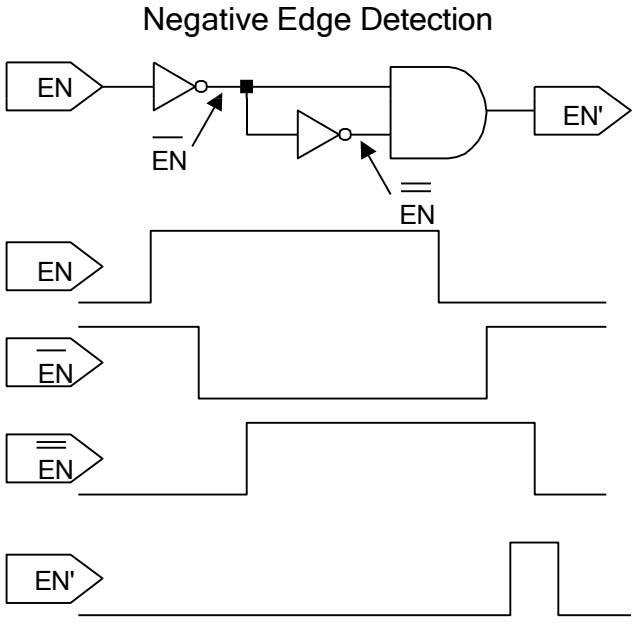
Latches and Flip-Flops

- **Latches** are also called **transparent** or **level triggered** flip flops, because the change on the outputs will follow the changes of the inputs as long as the Enable input is set.
- **Edge triggered** flip flops are the flip flops that change there outputs only at the transition of the Enable input. The enable is called the Clock input.

Edge Detection Circuits

Edge detection circuits are used to detect the transition of the Enable from logic 0 to logic 1 (positive edge) or from logic 1 to logic 0 (negative edge). The operation of the edge detection circuits shown below is based on the fact that there is a time delay between the change of the input of a gate and the change at the output. This delay is in the order of a few nanoseconds. The Enable in this case is called the Clock (CLK)

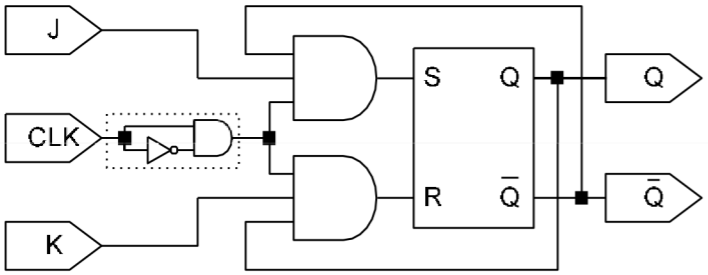




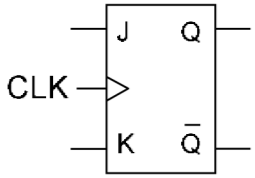
The JK Edge Triggered Flip Flop

The JK edge triggered flip flop can be obtained by inserting an edge detection circuit at the Enable (CLK) input of a JK latch. This ensures that the outputs of the flip flop will change only when the CLK changes (0 to 1 for +ve edge or 1 to 0 for –ve edge)

Positive Edge JK Flip Flop

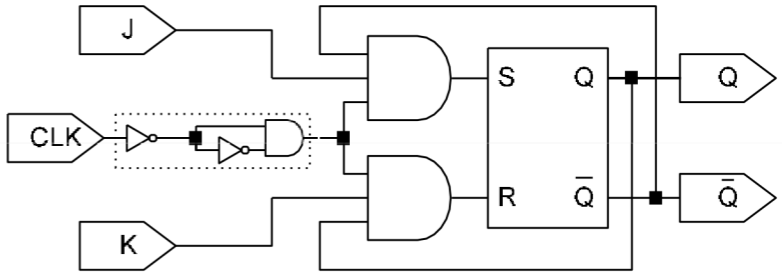


Logic Symbol

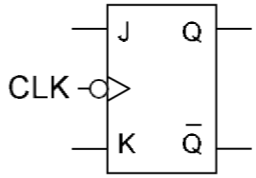


CLK	J	K	Q_{N+1}	Function
⌐	X	X	Q	
↑	0	0	Q	
↑	0	1	0	
↑	1	0	1	
↑	1	1	Q'	

Negative Edge JK Flip Flop



Logic Symbol



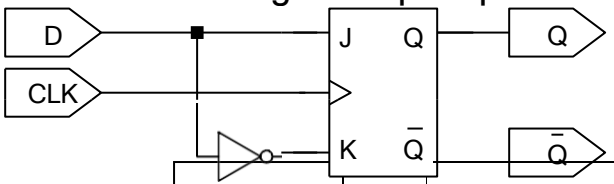
CLK	J	K	Q_{N+1}	Function
⌐	X	X		
↓	0	0		
↓	0	1		
↓	1	0		
↓	1	1		

for Computers

The D Edge Triggered Flip Flop

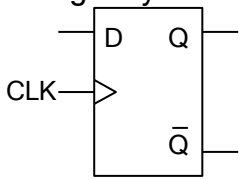
The D edge triggered flip flop can be obtained by connecting the J with the K inputs of a JK flip through an inverter as shown below. The D edge trigger can also be obtained by connecting the S with the R inputs of a SR edge triggered flip flop through an inverter.

Positive Edge D Flip Flop

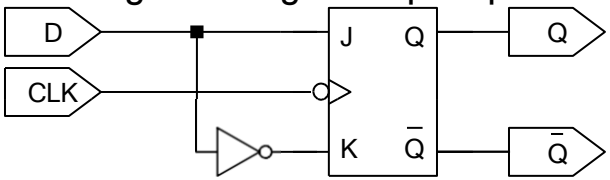


CLK	D	Q_{N+1}	Function
	X	Q	
↑	0	0	
↑	1	1	

Logic Symbol

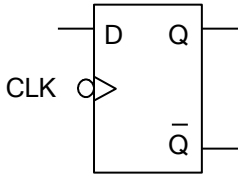


Negative Edge D Flip Flop



CLK	D	Q_{N+1}	Function
	X	Q	
↓	0	0	
↓	1	1	

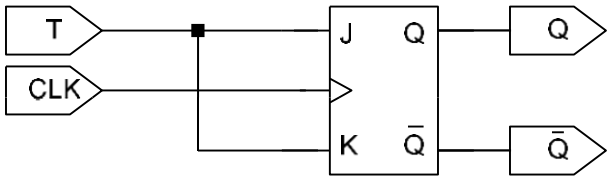
Logic Symbol



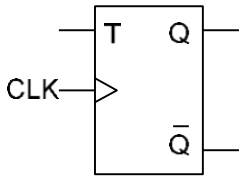
The Toggle (T) Edge Triggered Flip Flop

The T edge triggered flip flop can be obtained by connecting the J with the K inputs of a JK flip directly. When T is zero then both J and K are zero and the Q output does not change. When T is one then both J and K are one and the Q output will change to the opposite state, or toggle.

Positive Edge T Flip Flop

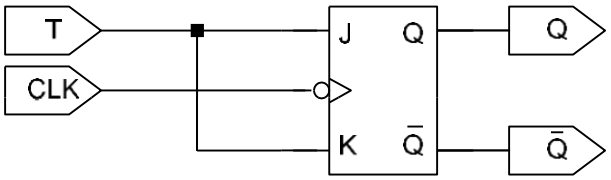


Logic Symbol

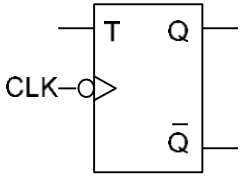


CLK	T	Q_{N+1}	Function
⌐	X	Q	
↑	0	Q	
↑	1	Q´	

Negative Edge T Flip Flop



Logic Symbol

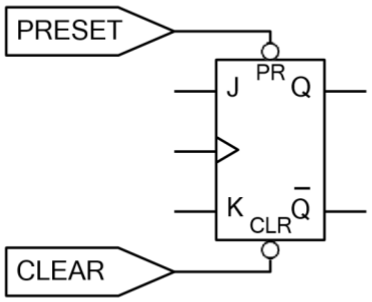


CLK	Q_{N+1}	Function
⌐ X	Q	
↓	Q	
↓	Q´	

Flip Flops with asynchronous inputs (Preset and Clear)

Two extra inputs are often found on flip flops, that either clear or preset the output. These inputs are effective at any time, thus are called asynchronous. If the Clear is at

Positive Edge JK Flip Flop with Preset and Clear



CLK	PR	CLR	J	K	Q_{N+1}	Function
⌐_	0	0	X	X		
	0	1	X	X	1	
	1	0	X	X	0	
	1	1	0	0	Q	
	1	1	0	1	0	
	1	1	1	0	1	
	1	1	1	1	Q’	

logic 0 then the output is forced to 0, irrespective of the other normal inputs. If the Preset is at logic 0 then the output is forced to 1, irrespective of the other normal inputs. The preset and the clear inputs can not be 0 simultaneously. In the Preset and Clear are both 1 then the flip flop behaves according to its normal truth table.

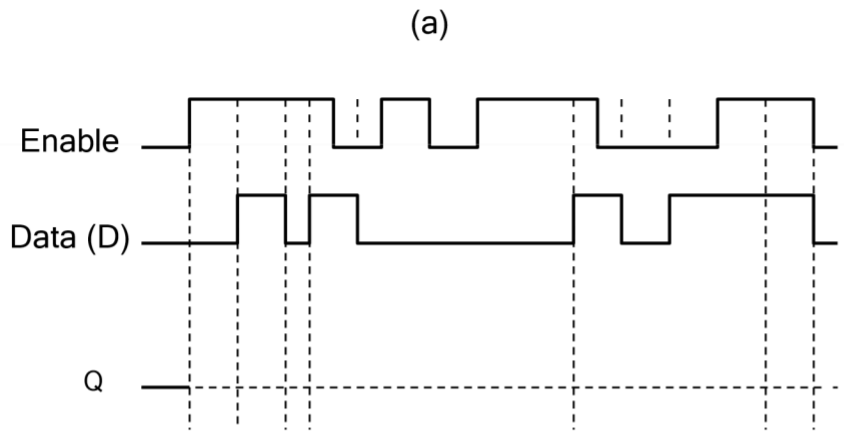
Data (D) Latch :- Example

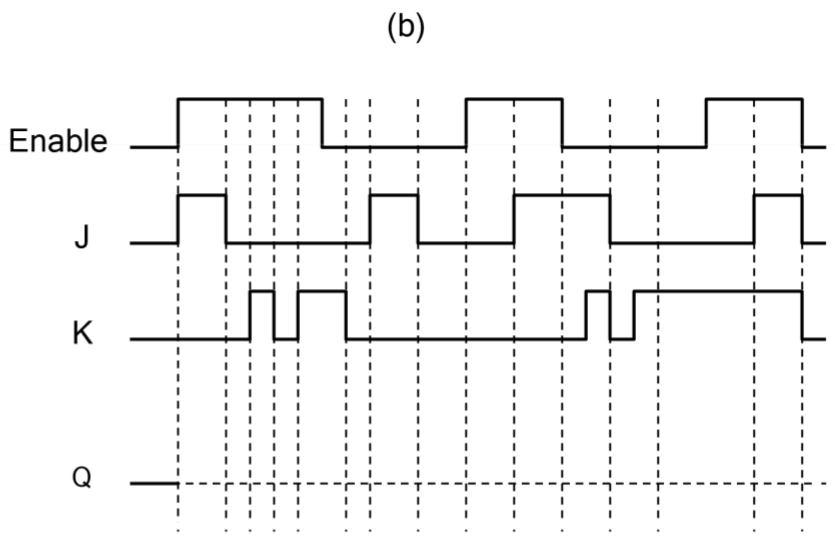
Complete the timing diagrams for :

(a) D Latch

(b) JK Latch

Assume that for both cases the Q output is initially at logic zero.





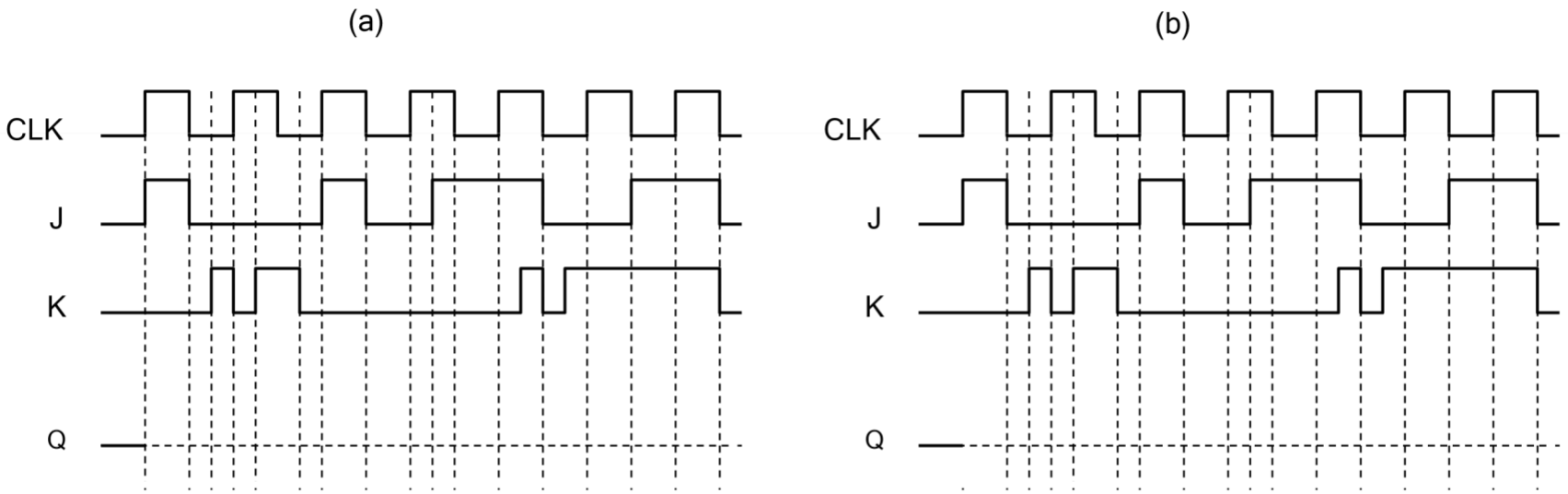
JK Edge Triggered Flip Flop :- Example

Complete the timing diagrams for :

(a) Positive Edge Triggered JK Flip Flop

(b) Negative Edge Triggered JK Flip Flop

Assume that for both cases the Q output is initially at logic zero.



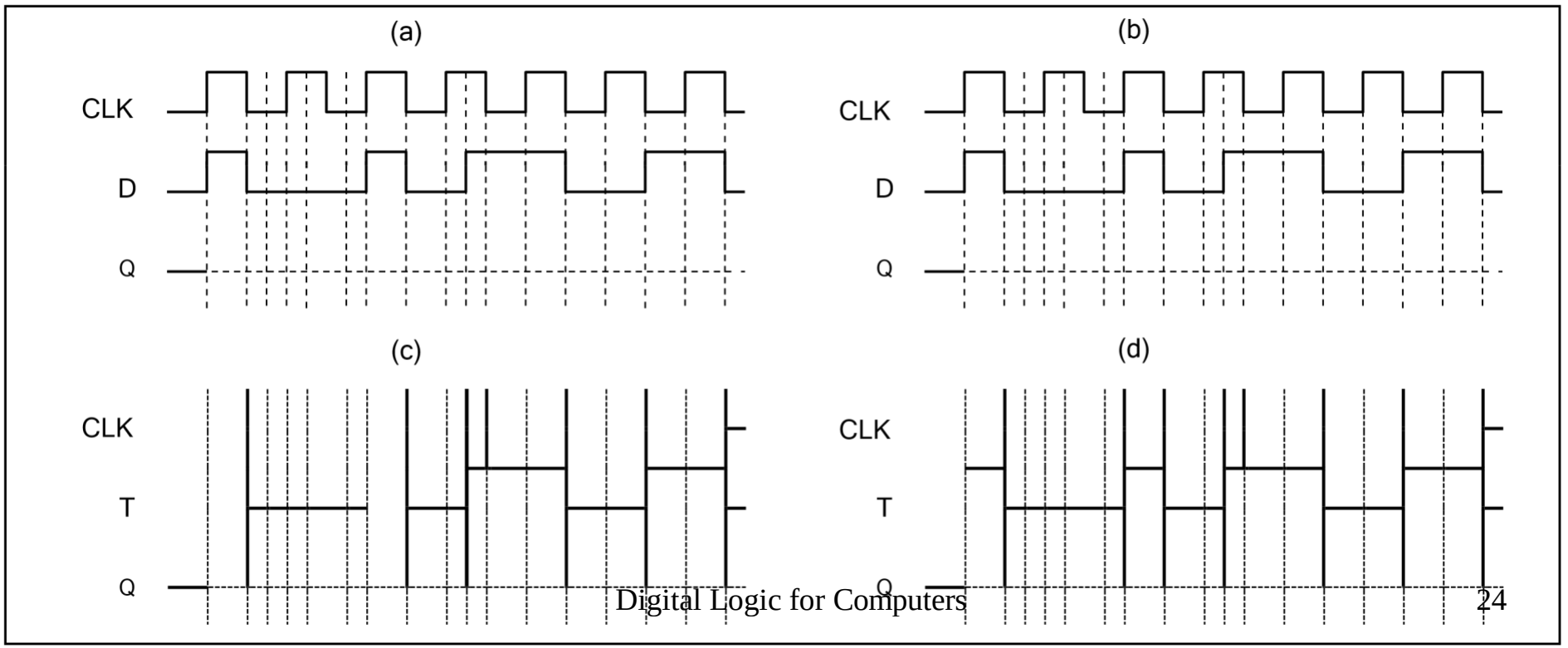
D and T Edge Triggered Flip Flops :- Example

Complete the timing diagrams for :

(a) Positive Edge Triggered D Flip Flop

(b) Positive Edge Triggered T Flip Flop

(c) Negative Edge Triggered T Flip Flop



(d) Negative Edge Triggered D Flip Flop

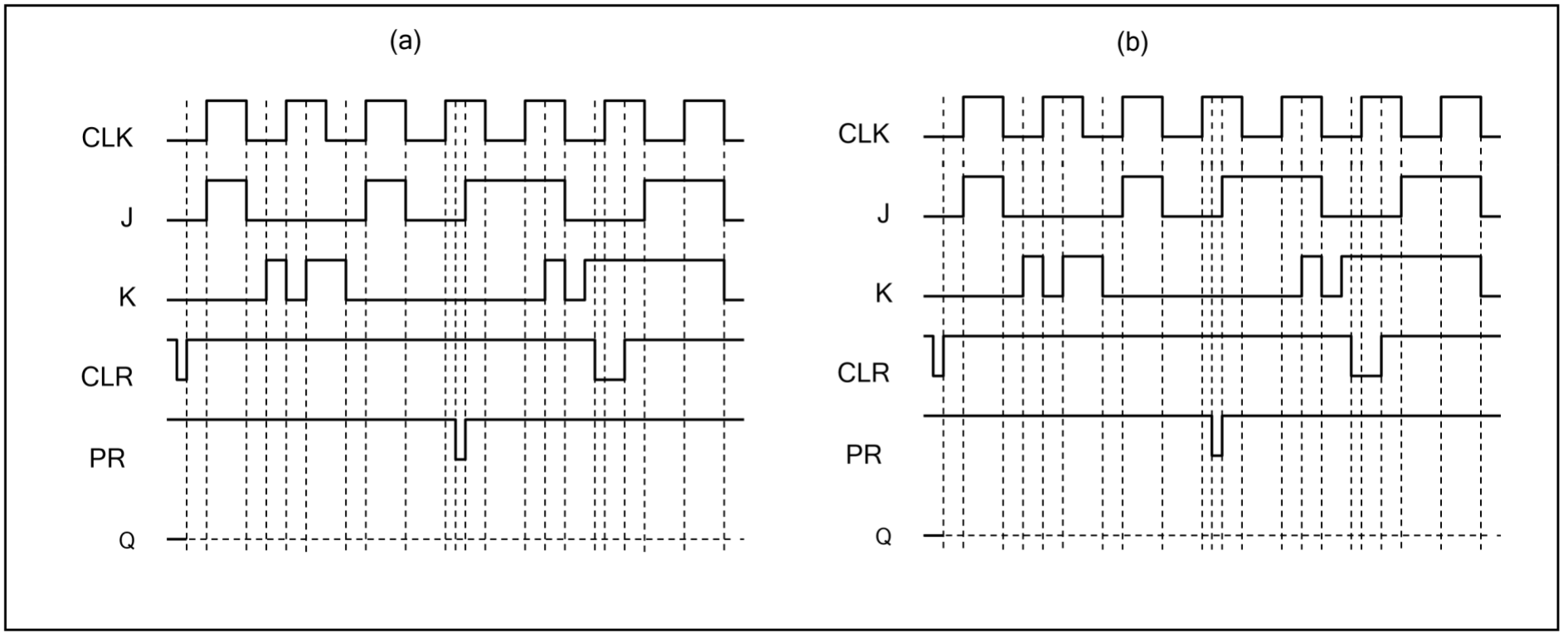
JK Flip Flop With Preset and Clear:- Example

Complete the timing diagrams for :

(a) Positive Edge Triggered JK Flip Flop

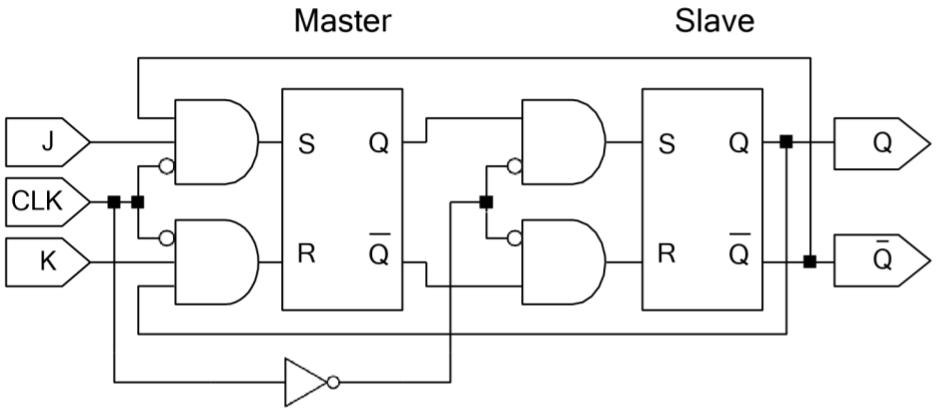
(b) Negative Edge Triggered JK Flip Flop.

Assume that for both cases the Q output is initially at logic zero.

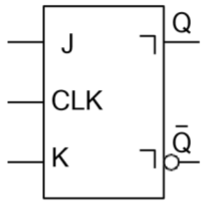

(a)
CLK
J
K
CLR
PR
Q
(b)
CLK
J
K
CLR
PR
Q

Level Triggered Master Slave JK Flip Flop

A Master Slave flip flop is obtained by connecting two SR latches as shown below. This flip flop reads the inputs when the clock is 1 and changes the output when the clock is at logic zero.



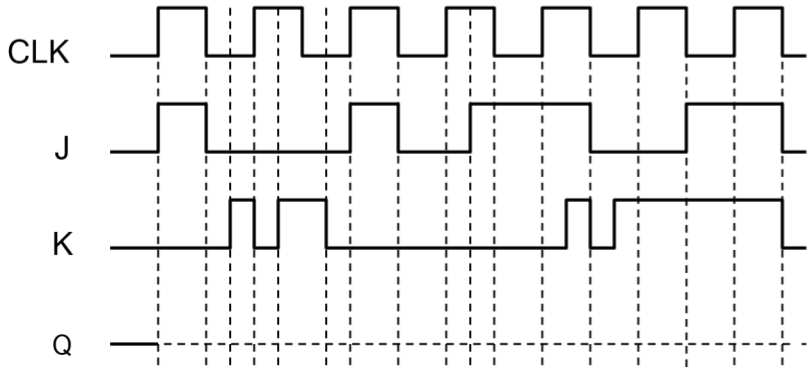
Logic Symbol



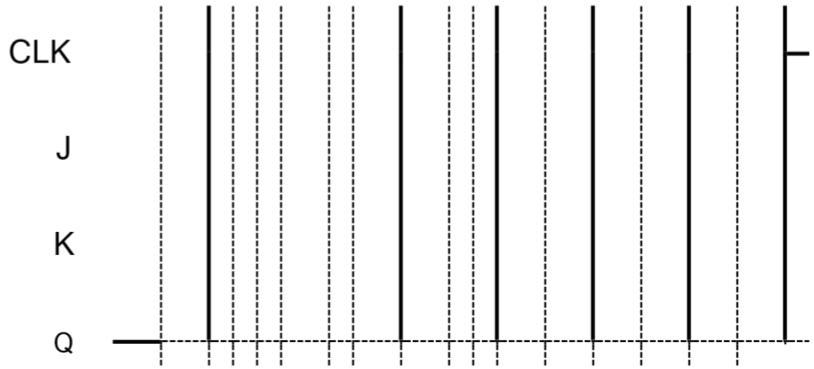
Truth Table

CLK	J	K	Q	Function
⎍	0	0		
⎍	0	1		
⎍	1	0		
⎍	1	1		

(a) Positive Master Slave JK Flip Flop

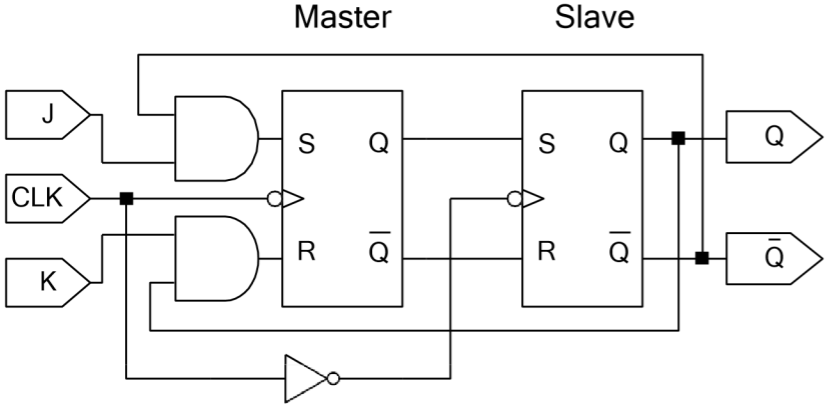


(b) Negative Master Slave JK Flip Flop

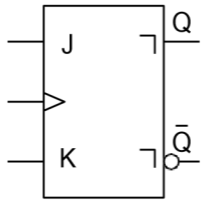


Edge Triggered Master Slave JK Flip Flop

A Master Slave flip flop is obtained by connecting two SR latches as shown below. This flip flop reads the inputs when the clock is 1 and changes the output when the clock is at logic zero.



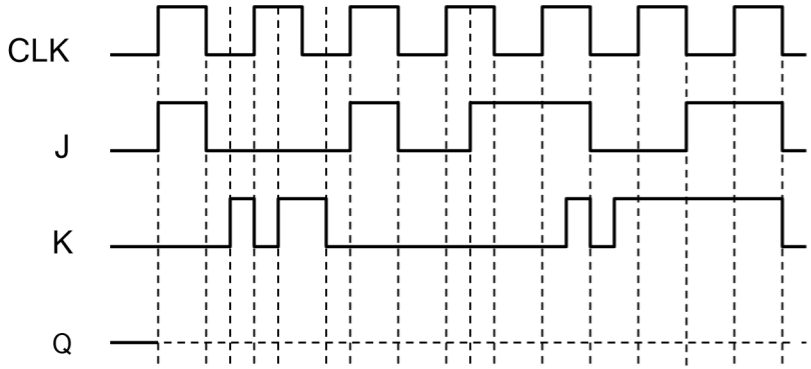
Logic Symbol



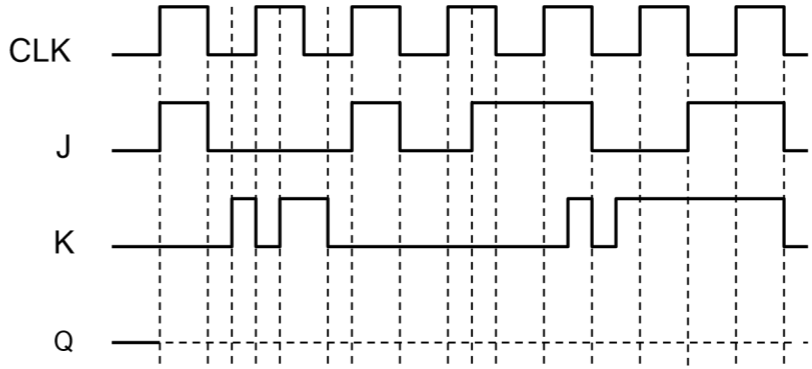
Truth Table

CLK	J	K	Q	Function
↓	0	0		
↓	0	1		
↓	1	0		
↓	1	1		

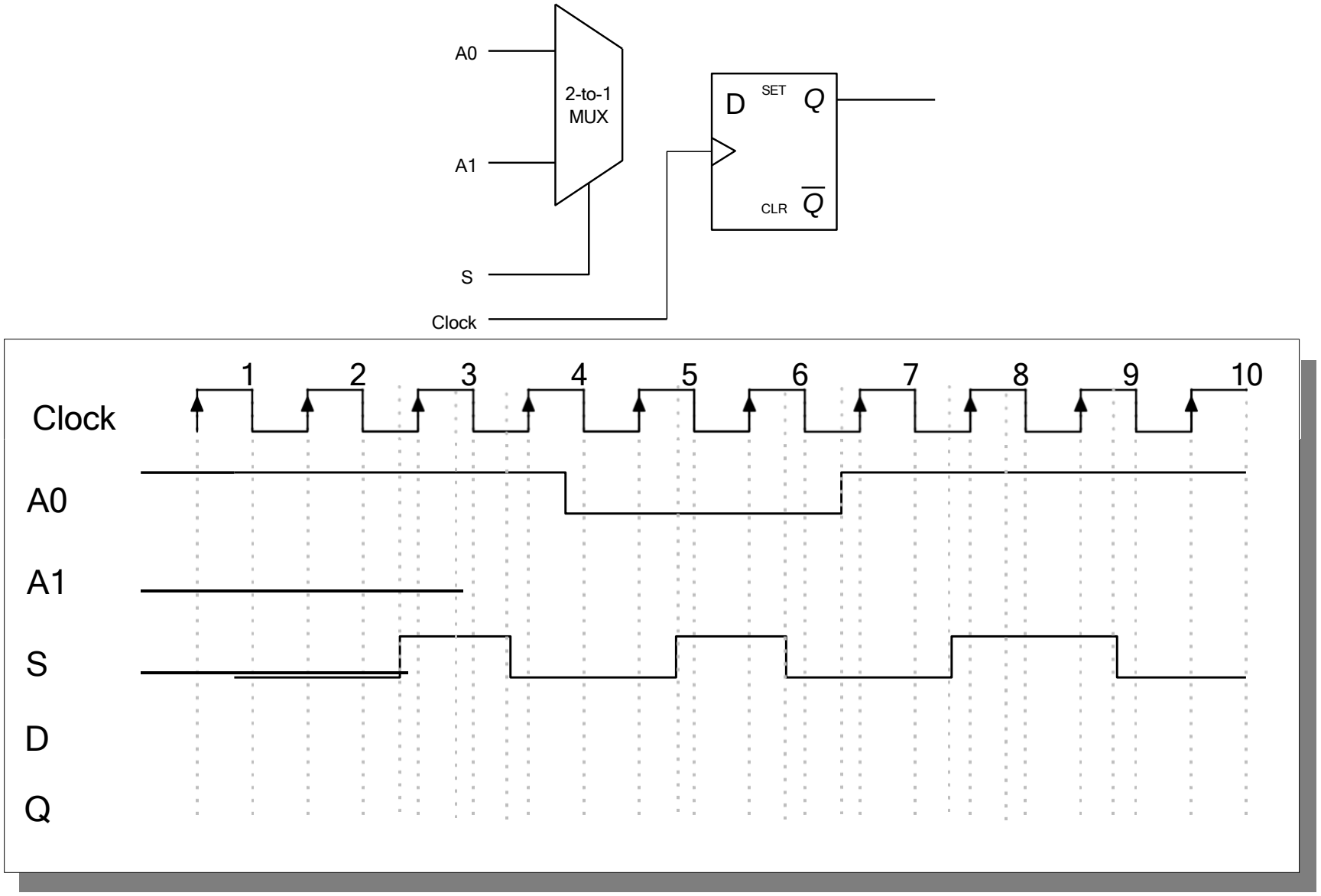
(a) Positive Master Slave JK Flip Flop



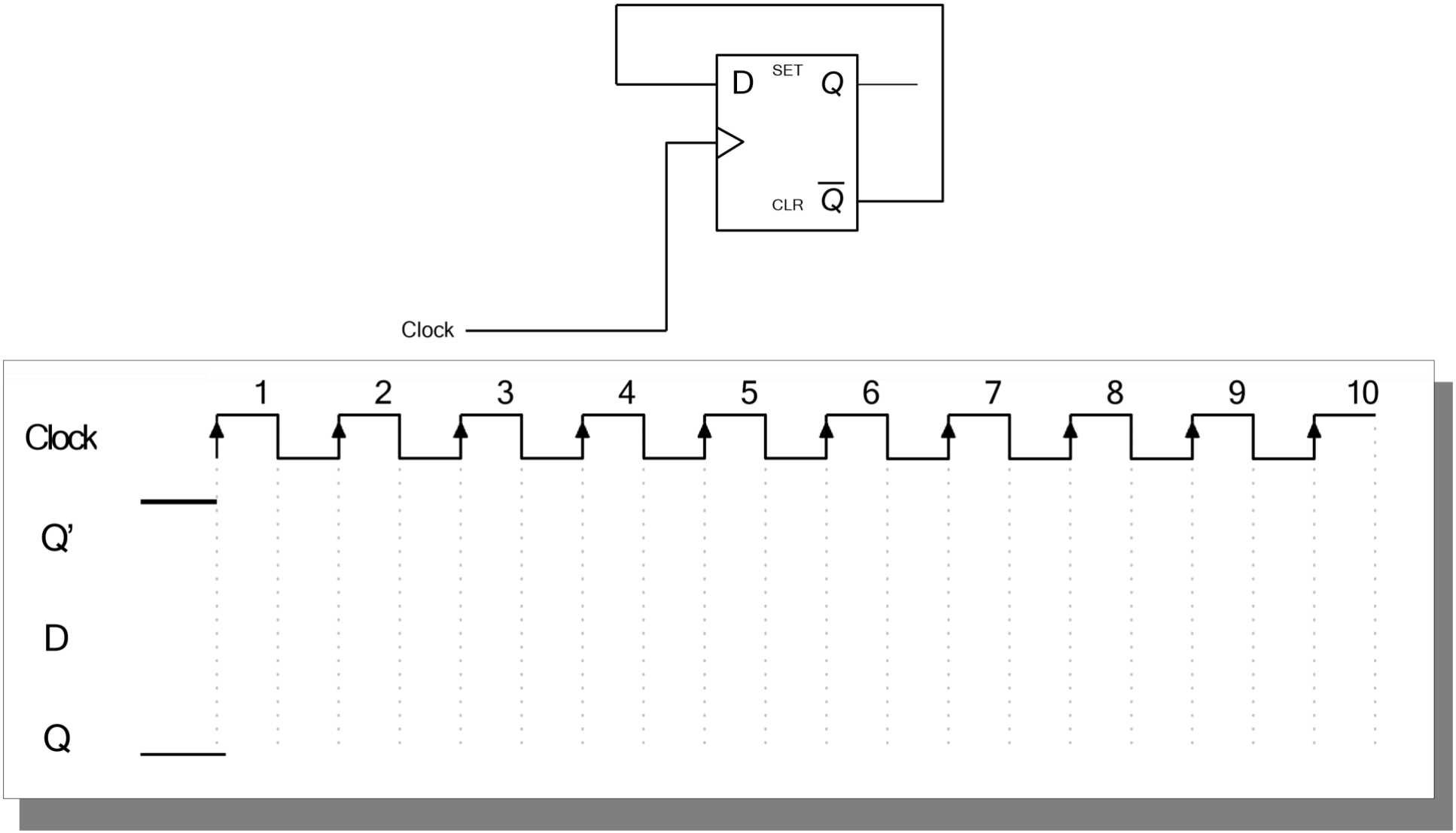
(b) Negative Master Slave JK Flip Flop



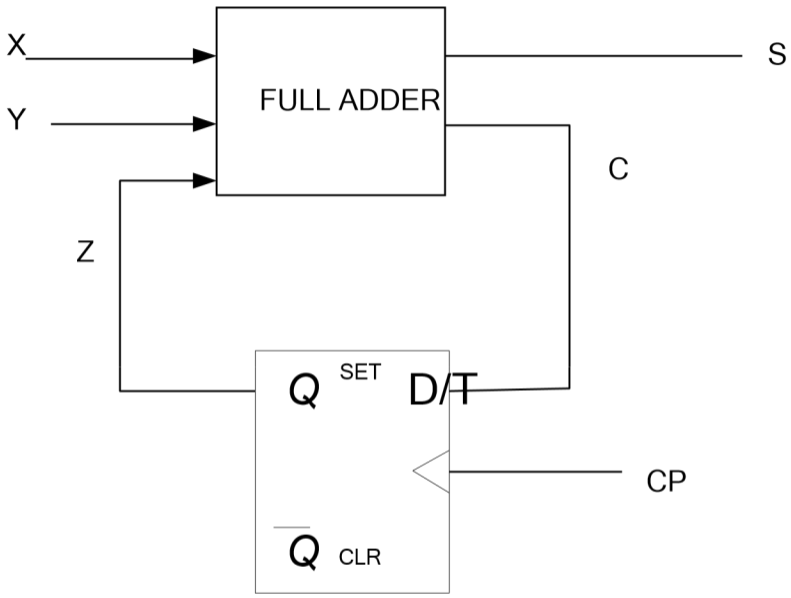
Sequential circuit example 1

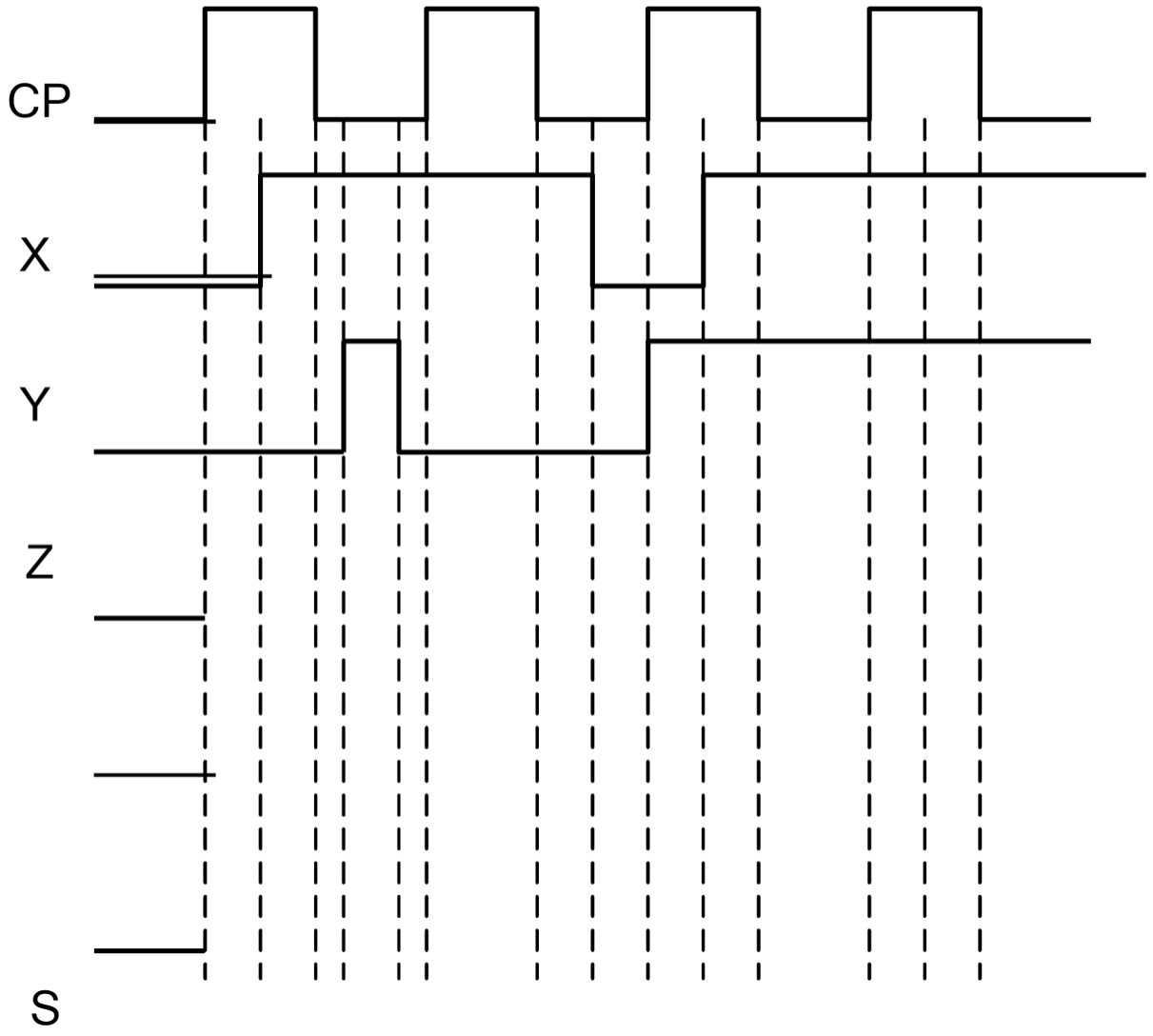

A0
A1
S
Clock
2-to-1
MUX
D
SET
Q
CLR
$\overline{Q}$
1
2
3
4
5
6
7
8
9
10
Clock
A0
A1
S
D
Q

Sequential circuit example 2



Sequential circuit example 3


X
Y
Z
FULL ADDER
S
C
Q
SET
D/T
CP
Q
CLR


CP
X
Y
Z
S

C

Digital Logic for Computers