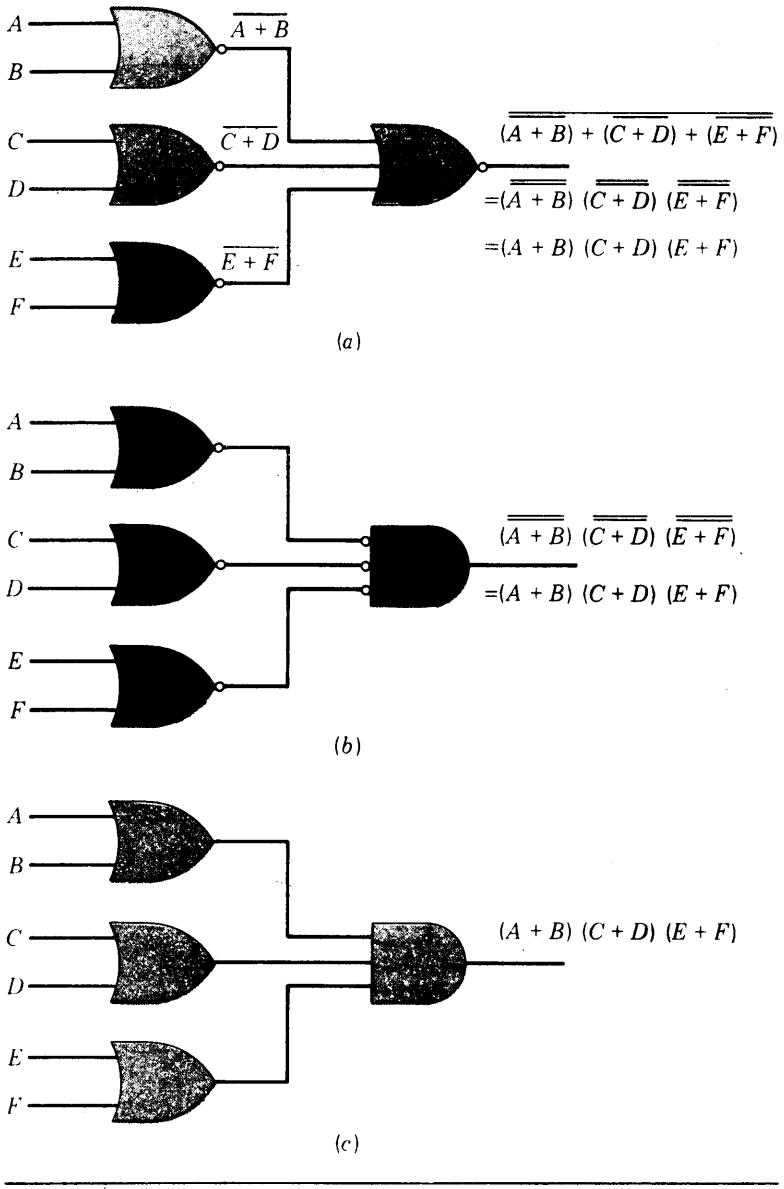


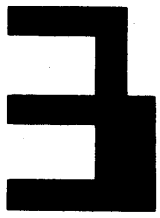
FIGURE 3.29

NOR-to-NOR gate network analysis.

NAND-TO-AND AND NOR-TO-OR GATE NETWORKS

3.23 In the two preceding sections, we showed how to analyze and design networks using NAND and NOR gates in NAND-to-NAND and NOR-to-NOR forms. Two other forms are in common usage: the NAND-to-AND and the NOR-to-OR forms.

Since NAND gates are quite popular and since the outputs from NAND gates



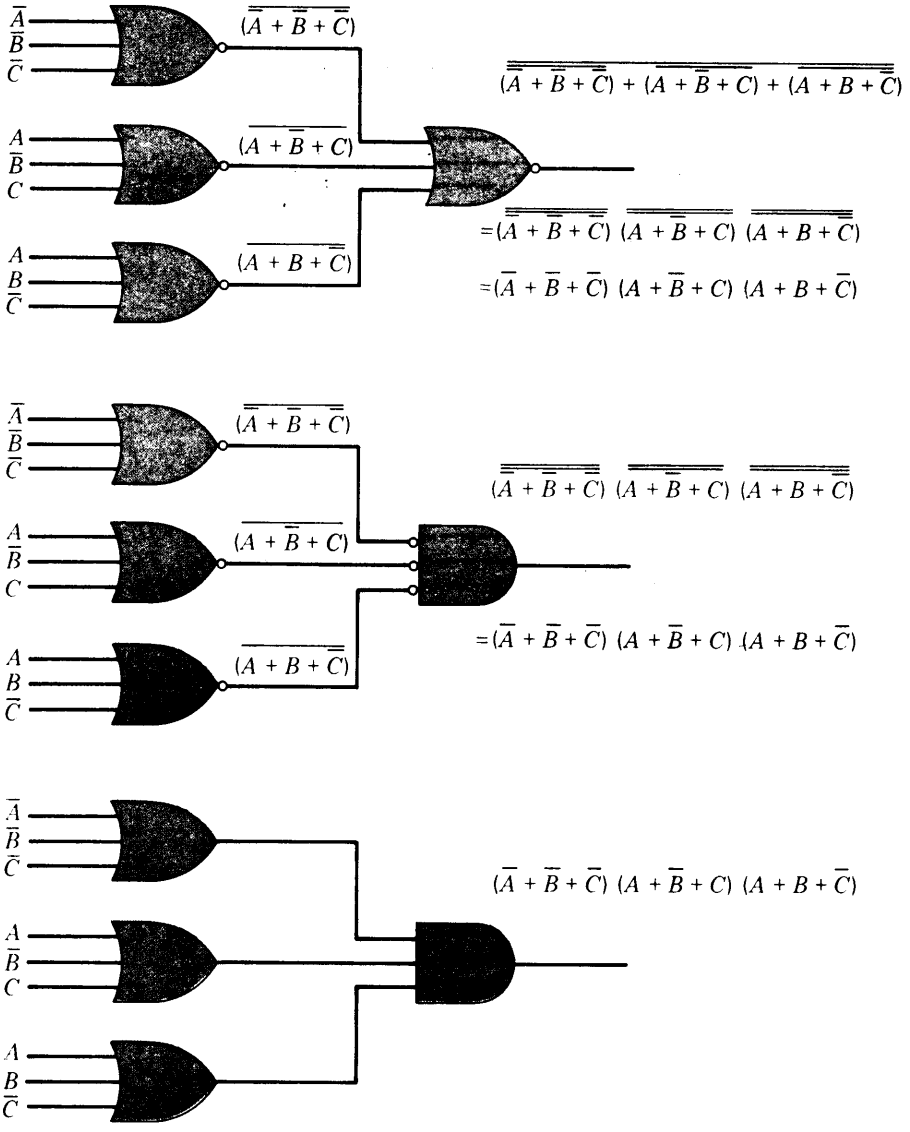
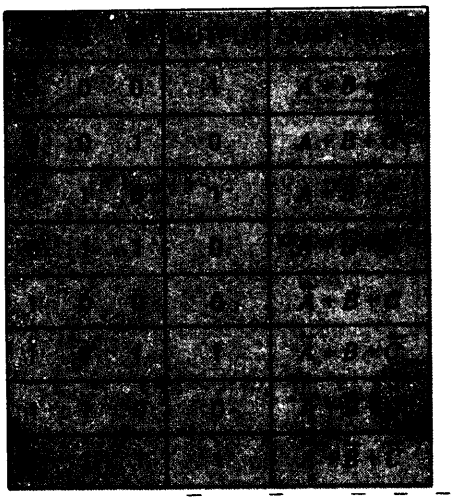


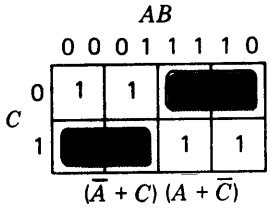
FIGURE 3.31

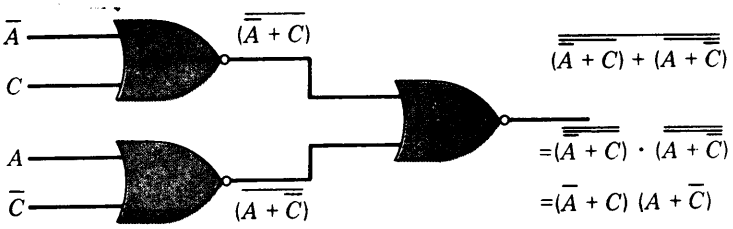
NOR gate network analysis.

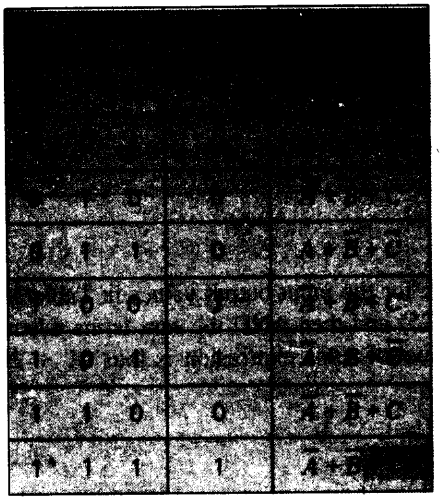
sometimes can be ANDed by a simple connection, as we show later, we first present the analysis and design procedures for NAND-to-AND gate networks.

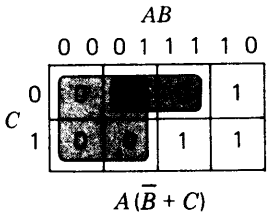
Figure 3.33(*a*) shows a NAND-to-AND gate network with inputs *A*, *B*, *C*, *D*, and *E*. Figure 3.33(*b*) shows the same configuration but with the NAND gates replaced by the equivalent NAND gate symbol from Fig. 3.24. This shows a NAND-to-AND network functions like an OR-to-AND network with each input complemented and leads to this design rule.

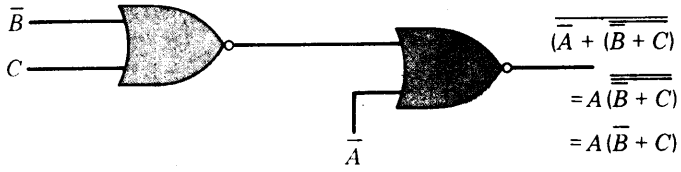








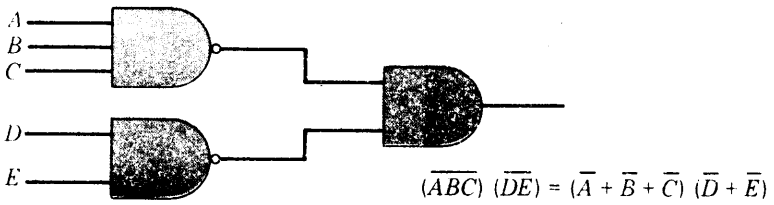




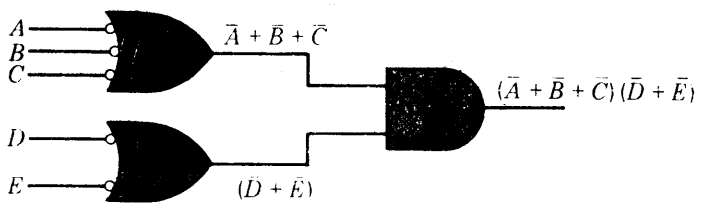
NAND-TO-AND AND NOR-TO-OR GATE NETWORKS

FIGURE 3.32

Two NOR gate designs.



a. Conventional NAND to AND gate network

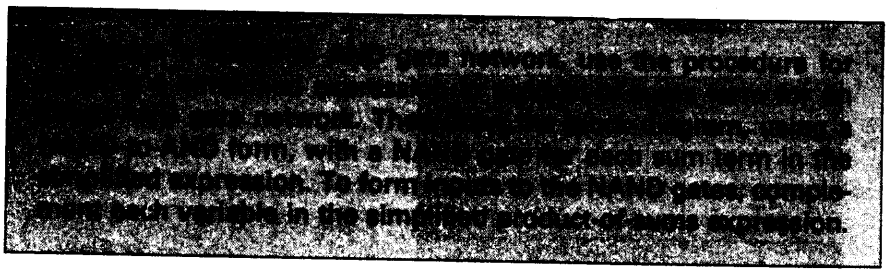


b. NAND-to-AND in (a) but with equivalent gates substituted for NANDs.

FIGURE 3.33

NAND-to-AND gate network. (*a*) Conventional NAND-to-AND gate network. (*b*) NAND-to-AND with equivalent gates substituted.

DESIGN RULE



Example

Design a NAND-to-AND gate network for the input-output values in Table 3.24. We add a sum-term column (Table 3.25) and then AND the sum terms where 0s appear in the output values. Our product-of-sums expression is thus $(A + B +$

TABLE 3.24

A	B	C	OUTPUT

TABLE 3.25

A	B	C	OUTPUT	SUM TERM

$\overline{C})(A + \overline{B} + C)(\overline{A} + B + C)(\overline{A} + B + \overline{C})$. This must be simplified. The simplified expression is $(A + \overline{B} + C)(\overline{A} + B)(B + \overline{C})$. The rule states that we must now form a NAND-to-AND gate network, but that each input should be complemented. This means each variable in $(A + \overline{B} + C)(\overline{A} + B)(B + \overline{C})$ must be complemented. The inputs for one NAND gate thus will be $\overline{A}$, B, and $\overline{C}$ which are from the first sum term $(A + \overline{B} + C)$. The inputs to the second NAND gate will be A and $\overline{B}$ from the term $\overline{A} + B$, and the third NAND gate will have as inputs $\overline{B}$ and C from the sum term $(B + \overline{C})$. The resulting block diagram is seen in Fig. 3.34.

NOR-to-OR gate networks are also widely used because NORs are the natural gates for emitter-coupled logic (ECL) circuits, a major circuit line. Figure 3.35(*a*) shows a NOR-to-OR gate network with four inputs and the output boolean algebra expressions. Figure 3.35(*b*) shows the same configuration but with equivalent gates from Fig. 3.29 substituted for the NOR gates. This shows that the basic form for the boolean expression realized is AND-to-OR but with each input variable complemented. Thus the design rule for a NOR-to-OR gate network is as follows:

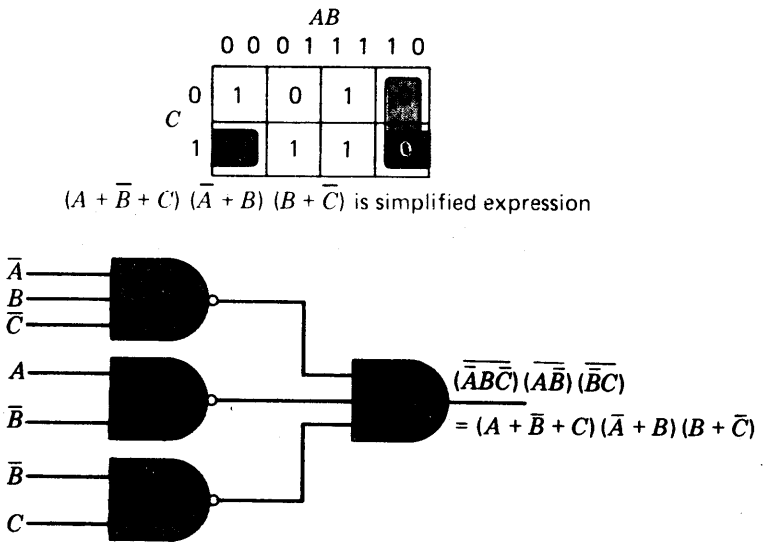


FIGURE 3.34

Design of a NAND-to-AND network.

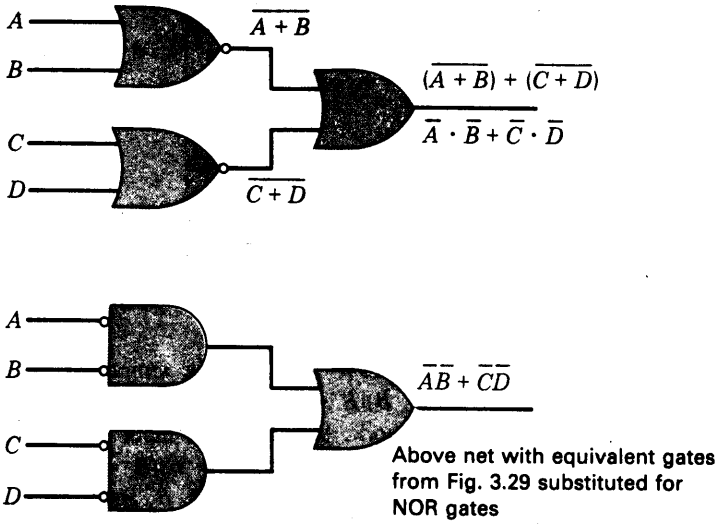


FIGURE 3.35

NOR-to-OR gate network and equivalent network.

DESIGN RULE

To design a NOR-to-OR gate network, develop and simplify the sum-of-products expression for the described function. Then draw a NOR-to-OR gate network with a NOR gate for each product term, but complement each input in the sum-of-products expression to form the input to the NOR gates.

Table 3.26 shows a table of combinations to be realized as a NOR-to-OR gate network. The product terms are added to the table, and then the boolean algebra expression is derived for the problem: $\overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + A\overline{B}\overline{C} + AB\overline{C} + ABC$. This expression is then simplified, giving $AB + BC + A\overline{C}$.

The design rule says that to realize a NOR-to-OR gate network we use the above expression but complement each input. This means the first NOR gate will

TABLE 3.26

A	B	C	OUTPUT	PRODUCT TERMS

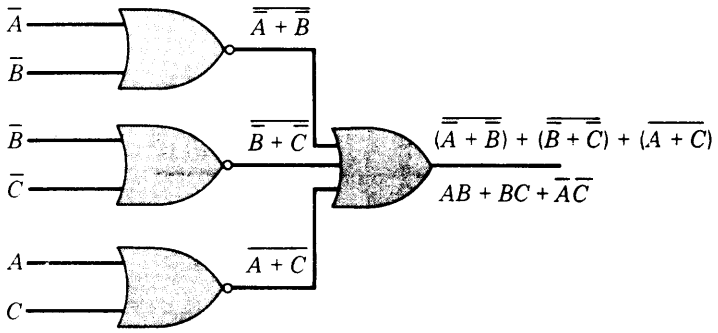


FIGURE 3.36

Design for NOR-to-OR gate network.

have as inputs $\overline{A}$ and $\overline{B}$ from the product term AB; the second NOR gate will have as inputs $\overline{B}$ and $\overline{C}$ from the product term BC; and the third NOR gate will have as inputs A and C from the product term $\overline{A}\,\overline{C}$. Figure 3-36 shows this design.

WIRED OR AND WIRED AND GATES

***3.24** In certain integrated-circuit technologies, it is possible to form OR and AND gates by means of a simple connection. Figure 3.37(*a*) shows a NAND-to-AND gate combination in which the AND gate is formed by simply connecting the NAND gate outputs. The wired AND gate in Fig. 3.37(*a*) requires no additional circuitry beyond that required for the NAND gates. This is shown by the dotted lines used in the NAND symbol.

Only certain NAND gates can have their outputs connected in this way and still form an AND gate. The designer of the NAND gates arranges for this feature, and the manufacturer will indicate on the specification sheet when this can be done. For example, when transistor-transistor logic (TTL) circuits are used, the specification sheets sometimes refer to the gates as having "open collectors," which means they can be formed into NAND-to-AND nets by simply connecting their outputs. In effect, the circuits are designed so that the output level of all gates when the gates are connected will be the lowest level any gate would output if the gates were operated singly.

Figure 3.37(*b*) and (*c*) shows examples of NAND-to-wired-AND nets which correspond in function to those in Figs. 3.33 and 3.34. Again, we emphasize that not all NAND gates can be wire-ANDed by using a simple connection. When this is possible, however, the saving in circuitry and speed improvement makes the configuration desirable.

An important observation should be made here: If inputs are wire-ANDed by using a simple connection, a single variable cannot be tied to the AND connection. A single input NAND gate (inverter) must be used. Refer to Fig. 3.38, which shows a design where a single variable B occurs in the minimal expression.

To explain this problem, if in Fig. 3.38 A and C are each 1 and B is 0, then the NAND gate output should be 0, while the value of $\overline{B}$ is 1. What would the value at the wired AND junction be? Will the NAND gate output pull $\overline{B}$ down, or

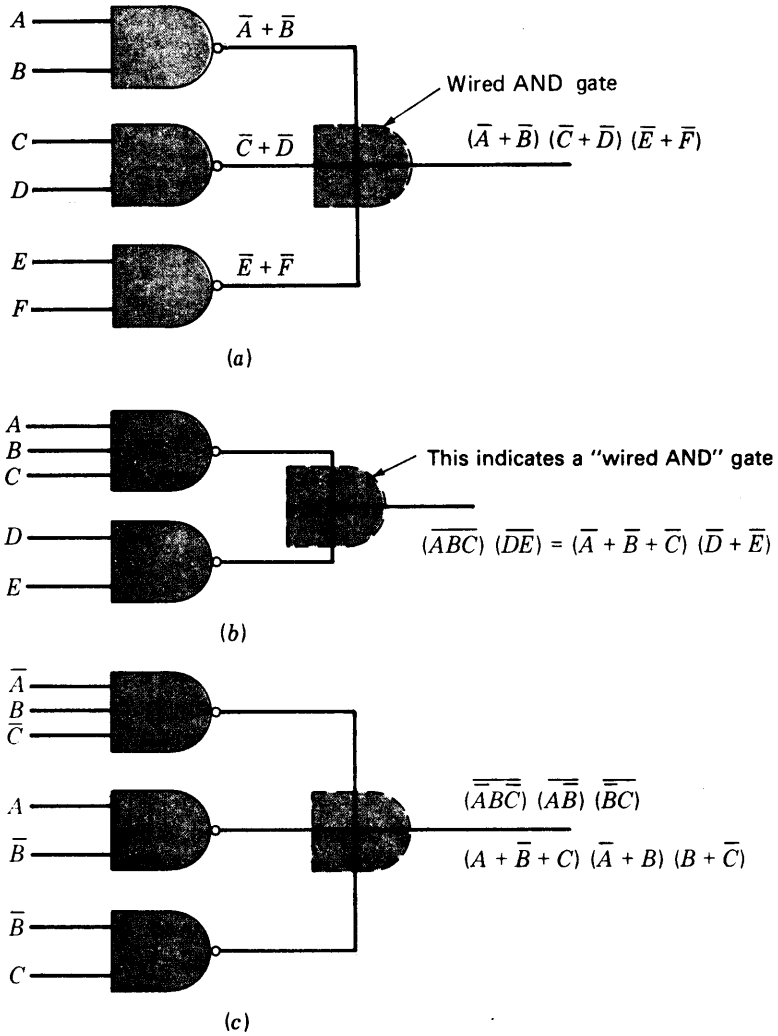


FIGURE 3.37

NAND to wired AND networks. (*a*) NAND-to-AND with wired AND gate. (*b*) NAND-to-AND with wired AND for Fig. 3.33(*a*). (*c*) NAND to wired AND for Fig. 3.34.

will the 1 on $\overline{B}$ force the level up? The situation is to use an AND gate, not a wired AND, or to use a NAND gate with the ability to have its output wire-ANDed, as shown in Fig. 3.38.

Some NOR gates will form an OR gate at their output when they are connected. Figure 3.39 shows a NOR-to-wired-OR net with output function $(\overline{A + B}) + (\overline{C + D}) = \overline{A}\cdot\overline{B} + \overline{C}\cdot\overline{D}$. This expression, $\overline{A}\cdot\overline{B} + \overline{C}\cdot\overline{D}$, shows us that the NOR-to-OR gate network functions as an AND-to-OR gate network but with each variable complemented. Again, the dotted symbol shows the gate is wired OR.

The above result shows that we can design for NOR-to-wired-OR networks just as for NOR-to-OR networks.

Again, note that only certain NOR gates can be connected at their outputs to

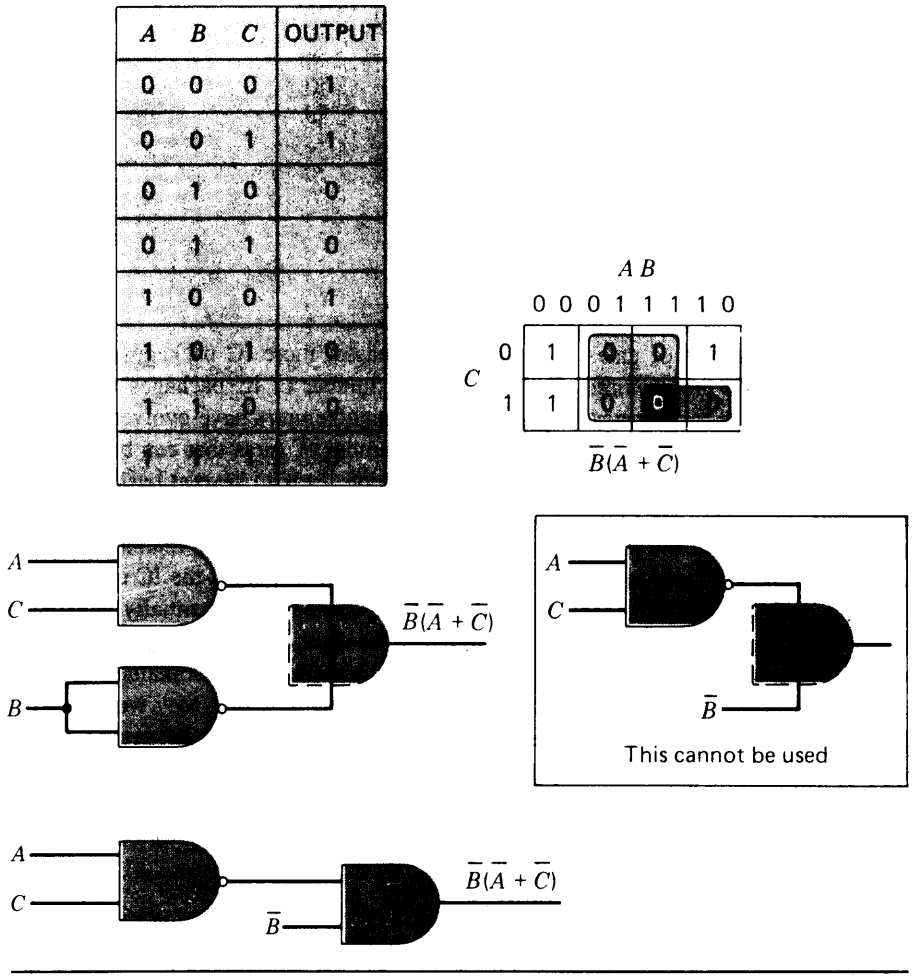


FIGURE 3.38

NAND-to-AND gate design with single variable.

form wired ORs. Some ECL circuits make this possible, and the manufacturer notes this on the specification sheets.

PLAs AND PALs

***3.25** When a design has been made for a gate network, the next step is to implement the design using integrated circuits. As has been mentioned, the most used IC line for gate networks has been a line called *transistor-transistor-logic* (TTL). Figure 3.40(*a*) shows an IC container, and Fig. 3.40(*b*) shows the gate layout in that container. This is called the *pin-out* for the IC package. The package in Fig. 3.40(*b*) is one of several hundred different gate layouts from which a designer can choose. By using this particular IC package, the NAND-to-NAND gate network for $A\bar{B} + BC$ can be realized by connecting the pins of the package as shown in Fig. 3.40(*c*). These connections are often made as conducting metallic strips on printed-circuit boards on which the IC containers are mounted.

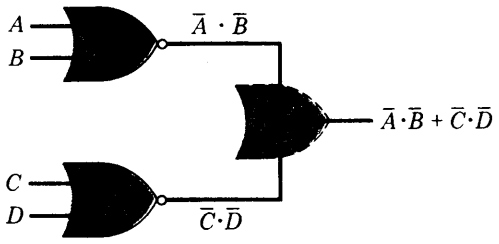


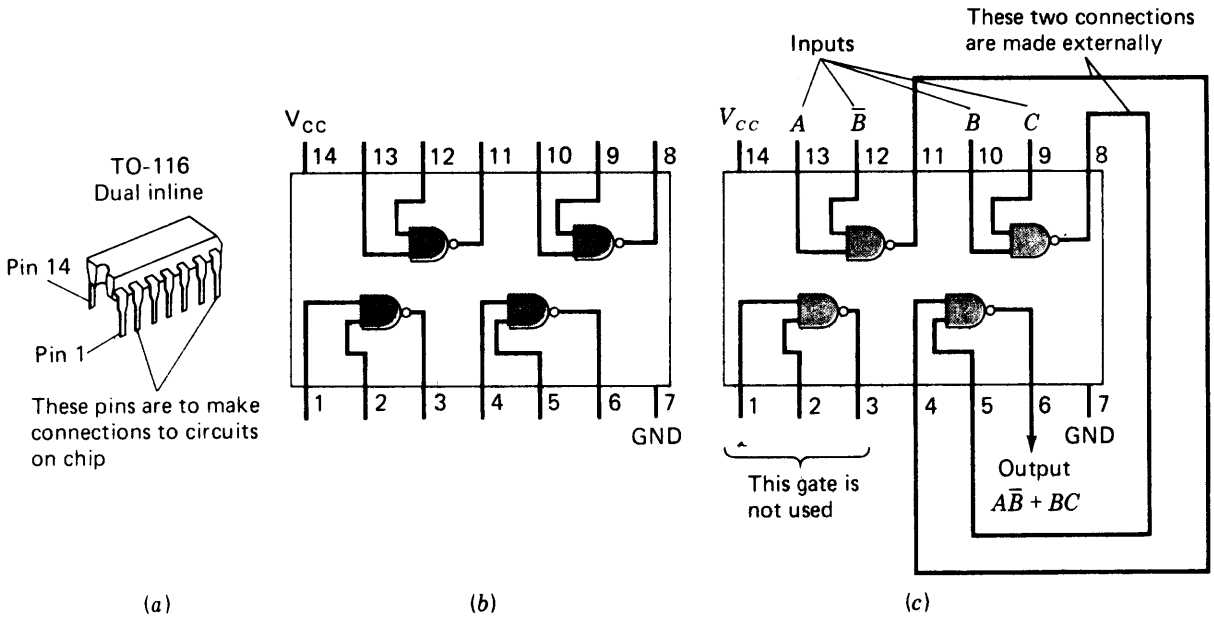
FIGURE 3.39

NOR-to-wired-OR gate network.

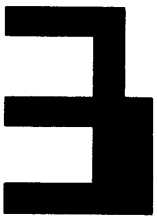
As the number of gates in a network increases, more IC packages such as that in Fig. 3.40 are required. To decrease the number of IC packages required and simplify interconnecting the packages, IC manufacturers have evolved manufacturing processes which greatly increase the number of gates that can be placed in a single IC container. This large-scale integration leads to several basic design problems, however. For example, the inputs and outputs to the gates in Fig. 3.40 are all available and the gates can be interconnected in any desired manner, but if more gates are placed in a single container, the number of pins in the IC container must be increased. This increases the cost of the container substantially and decreases the ability of the designer to select just the right combination of gates for the network. Also connections must still be made outside the IC container. If the same connections are made inside the container (on the IC chip), they would cost less and be more reliable. This leads to the idea of a chip with a specific gate layout in which the gates are interconnected on the chip. An IC chip with a specific gate layout made for a particular design is called a *custom* chip. Unfortunately,

FIGURE 3.40

Integrated-circuit container and pin-out. (*a*) Integrated circuit container. (*b*) Pin-out showing gate layout in container in (*a*). (*c*) NAND-to-NAND gate set realizing $A\overline{B} + BC$.



(*a*) (*b*) (*c*)



generating a complete design for a new individual custom IC chip[12] can be very expensive (costs can be from $50 to several hundreds of thousands of dollars). This means that start-up costs for a computer design that requires a number of custom chips can be very high. Once custom chips are made, however, for large runs, the cost per manufactured chip is low.

The high start-up costs for custom chips have caused designers to use IC packages with only a few gates per package, as in Fig. 3.40, and form the gate networks by interconnecting the gates outside the IC packages (using a printed-circuit board, as previously noted), particularly when small numbers of the design are to be made. However, although this approach is practical and economical for small production runs, it does not utilize the level of integration[13] possible for present ICs.

To aid designers in using fewer chips, IC manufacturers make *semicustom* chips in IC containers in which a basic two-level gate network with many gates is produced and the gates can be interconnected on the chip as desired. These are called *programmable logic arrays* (PLAs) or *programmable array logic* (PALs).[14] Figure 3.41 shows a layout for a small PLA. This particular array has three AND gates and two OR gates. (In actual practice, an array would have several hundred or more gates.) Note that the connections from inputs A, B, C, to the AND gates are not complete and that the AND gate outputs are not connected to the OR gates. These connections are made as desired by the gate network designer.

Figure 3.42 shows a design which uses the PLA in Fig. 3.41 and which realizes the two boolean algebra expressions $AB\overline{C} + \overline{A}\overline{B}$ (for output 1) and $\overline{A}\overline{B} + AC$ (for output 2).

These PLAs are manufactured in two different ways. In the first, the manufacturer places a fused connection at every intersection point in the PLA between the inputs and the AND gates and between the AND and OR gates. Thus every possible connection is made when the PLA is manufactured, and then the undesired connections are removed by blowing the fuses.[15] This type of PLA is often called a *field-programmable logic array* (FPLA).

In the second manufacturing technique, the desired connections are made during manufacture. The manufacturer originally makes the IC array layout so that any desired connections can be made, and the logic designer tells the manufacturer which connections to make for a particular design. Then the manufacturer creates a *mask*, which generates the desired connections when layers of metallization are added to the chip during manufacture. Setting up this mask costs far less (several hundred dollars or less) than designing an entire new chip with the precise logic

---

[12]A custom IC chip is one made from scratch for a particular purpose. A particular gate configuration can be manufactured into a chip by developing the masks used to produce the chip design.

[13]The level of integration is the complexity of the chip in terms of gates per chip. *Small-scale integration* (SSI) is roughly 1 to 20 gates per chip, *medium-scale integration* (MSI) is 20 to 100 gates per chip, *large-scale integration* (LSI) is 100 to 1000 gates per chip, and *very large-scale integration* (VLSI) is more than 1000 gates.

[14]PAL is a registered trademark of Monolithic Memories.

[15]The fuses are blown by selecting a fuse using logic levels at the inputs and then applying a relatively high voltage to a pin on the IC container. Electronic instruments can be purchased which blow selected fuses on a PLA. This is called *programming* the PLA.

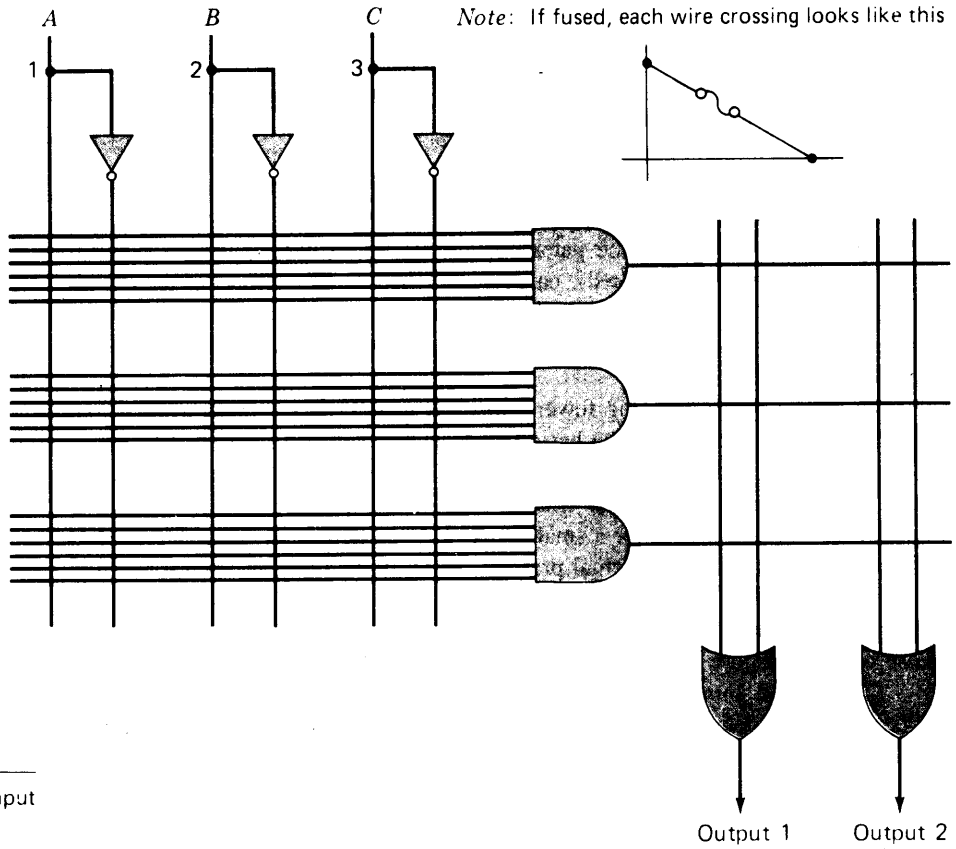


FIGURE 3.41

Layout for three-input two-output PLA.

array desired by the logic designer. And production runs of these chips are inexpensive.

Note that the AND gate which generates $\overline{A}\overline{B}$ in Fig. 3.42 has its output connected to two OR gates. This is sometimes a useful and desirable feature for PLAs enabling a single AND gate to be used for two outputs.

Most larger PLAs contain several hundred gates, 15 to 25 inputs, and 5 to 15 outputs. This offers the logic designer great flexibility. The low cost per unit of these IC gate networks has led to widespread use of PLAs.

To design these large arrays, a simplifying symbology has proved useful. Figure 3.43 shows this for the array in Fig. 3.42. The crosses drawn on the function indicate ANDs, and the squares indicate ORs. The figure also shows that the AND can be realized by a single semiconductor junction (called a *diode*), and the OR can be realized by a single junction pointed the other way. In practice, the manufacturer lays out the chip with junctions at every intersection of the lines, and only the desired diode connections are made.[16] Note: This is simply a redrawing of Fig.

[16]For field-programmable logic arrays (FPLAs) the diodes are fused so they can be blown by an instrument called an *array programmer*. This sets the FPLA as desired.

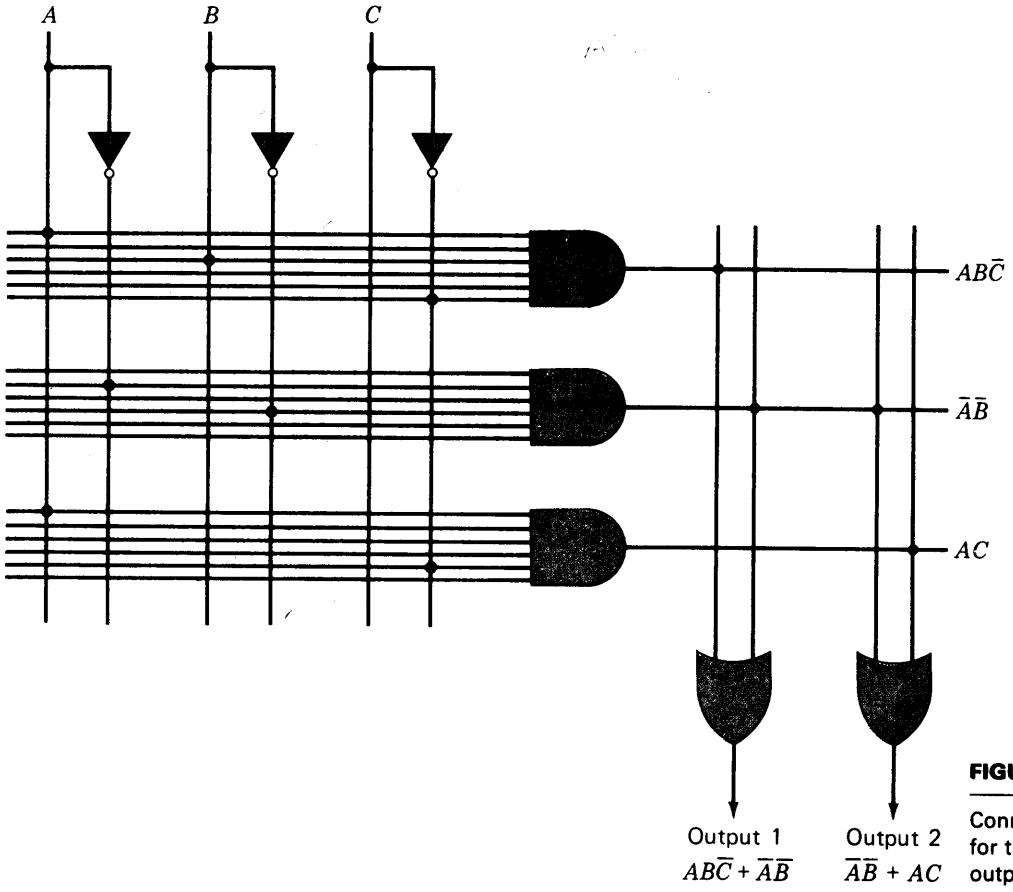


FIGURE 3.42

Connection design for three-input two-output PLA.

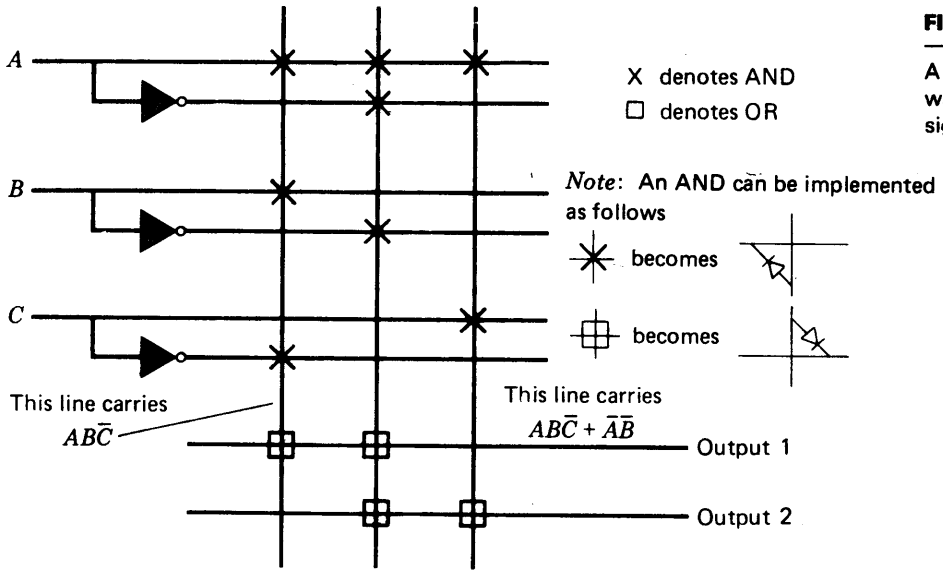
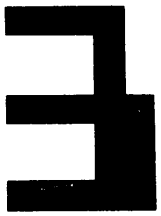


FIGURE 3.43

A frequently used way to draw PLA designs.



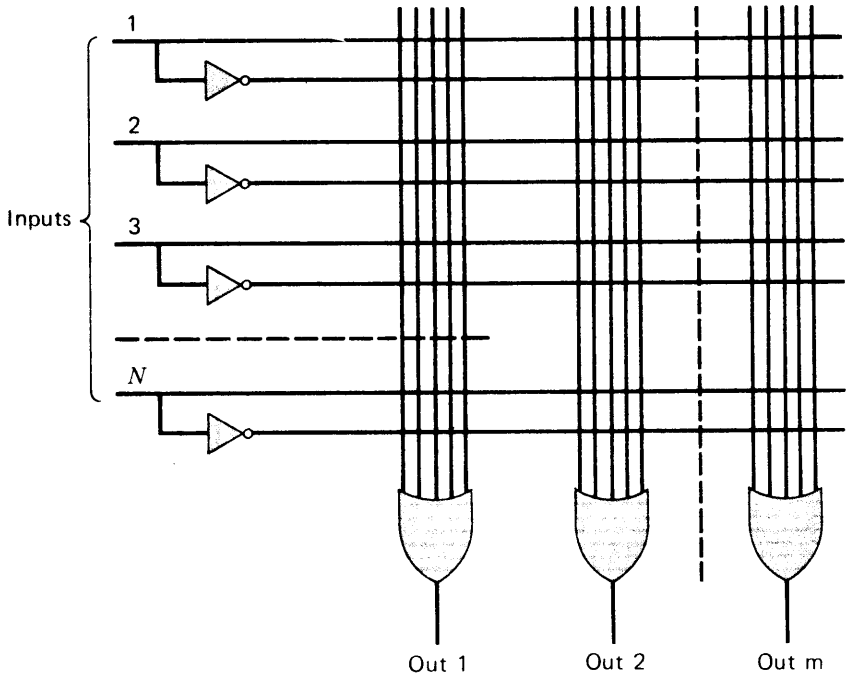


FIGURE 3.44

Layout for PAL.

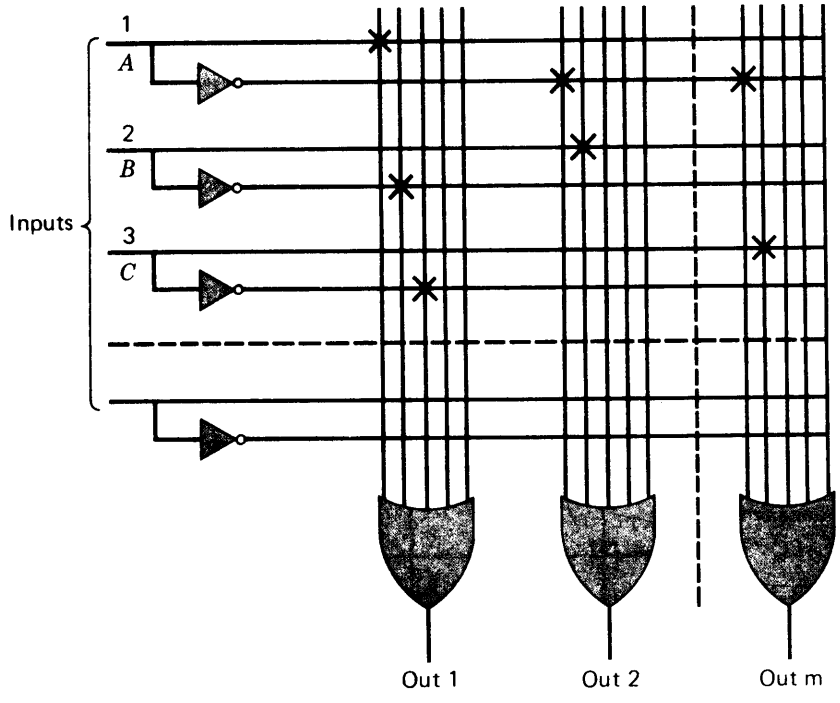


FIGURE 3.45

PAL design for $A\overline{B}\overline{C} + \overline{A}B + \overline{A}C$.

16X48X8 FPLA PROGRAM TABLE

THIS PORTION TO BE COMPLETED BY SIGNETICS

CF (XXXX) _____
CUSTOMER SYMBOLIZED PART # _____
DATE RECEIVED _____
COMMENTS _____

CUSTOMER NAME _____
PURCHASE ORDER # _____
SIGNETICS DEVICE # _____
TOTAL NUMBER OF PARTS _____
PROGRAM TABLE # _____
REV _____ DATE _____

PROGRAM TABLE ENTRIES

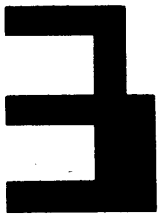
INPUT VARIABLE			OUTPUT FUNCTION		OUTPUT ACTIVE LEVEL	
Im	$\overline{Im}$	Don't Care	Prod Term Present in FP	Prod Term Not Present in FP	Active High	Active Low
H	L	— (dash)	A	• (period)	H	L
NOTE Enter '—' for unused inputs of used P-terms			NOTES 1 Entries independent of output polarity 2 Enter 'A' for unused outputs of used P-terms		NOTES 1 Polarity programmed once only 2 Enter 'H' for all unused outputs	

PRODUCT TERM[1]																	ACTIVE LEVEL[1]							
NO	INPUT VARIABLE																OUTPUT FUNCTION							
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0																								
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								
21																								
22																								
23																								
24																								
25																								
26																								
27																								
28																								
29																								
30																								
31																								
32																								
33																								
34																								
35																								
36																								
37																								
38																								
39																								
40																								
41																								
42																								
43																								
44																								
45																								
46																								
47																								

(1) Input and Output fields of unused P-terms can be left blank. Unused inputs and outputs are FPLA terminals left floating.

FIGURE 3.46

Program table for a PLA.



Pin configuration for 82S100

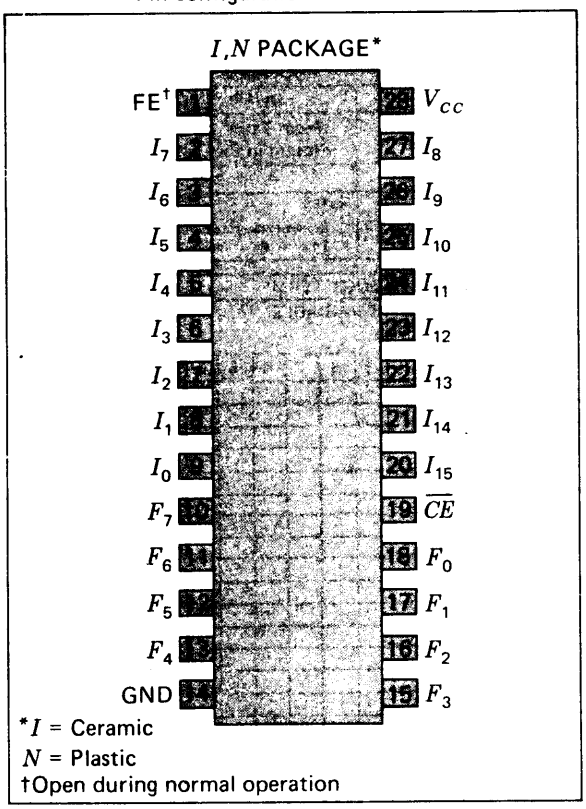


FIGURE 3.47

Pin-out for PLA. (*Signetics Corp.*)

3.42 with a different symbology. This symbology becomes very useful when there are many inputs and gates, however.

Figure 3.44 shows the layout for a version of the PLA which has become popular and which is called *programmable array logic*. The PALs are very similar to PLAs except that the OR gates are fixed and permanently connected to a set of AND gate output lines. As a result, AND gates cannot be shared, but the fixed OR gate connections lead to an ease of manufacture which has proved popular. Figure 3.45 shows a PAL design using the AND symbology (crosses) at intersections as in Fig. 3.43.

The current nomenclature calls the version of Fig. 3.44 in which the AND element connections are fused a PAL, but calls the version in which the manufacturer makes the connections *hard array logic* (HAL).

EXAMPLE OF DESIGN USING A PLA

3.26 Since PLAs and PALs are widely used because of their economy and speed of operation, we examine the design of a small network employing a widely used table listing.

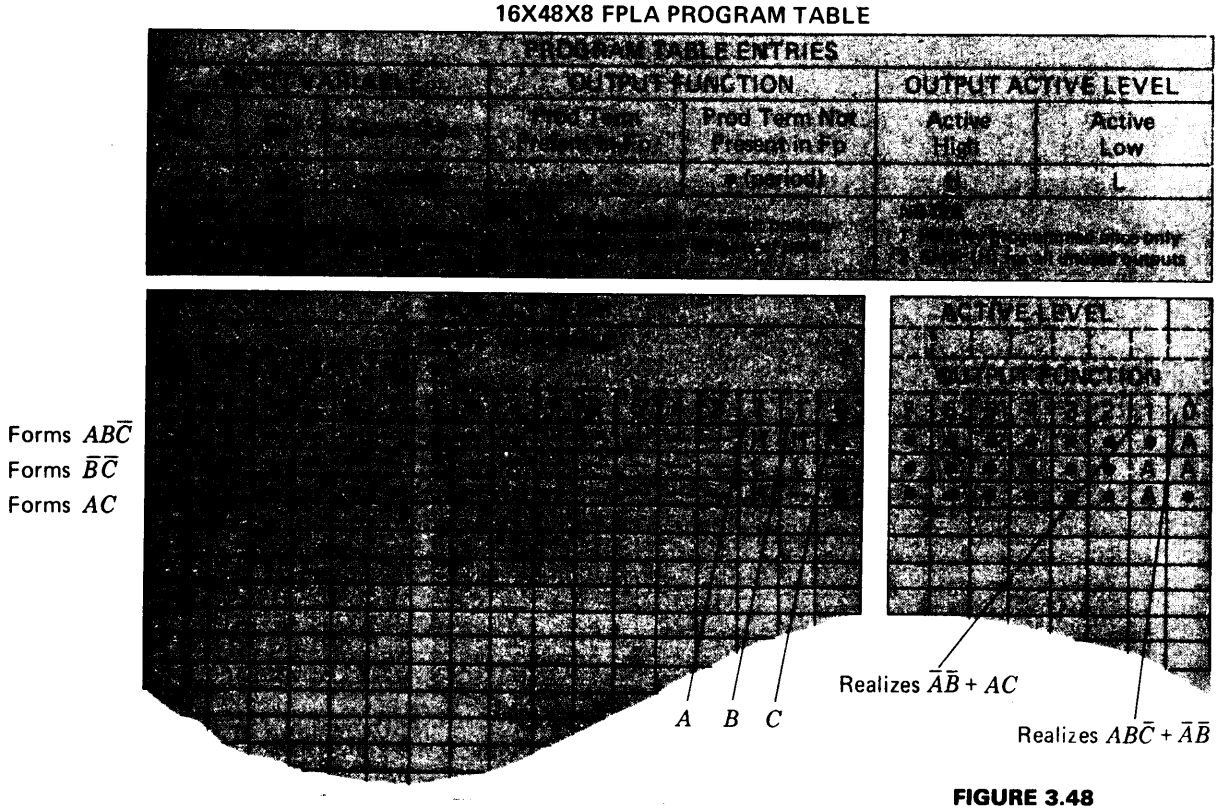


FIGURE 3.48

Program table for PLA design in Fig. 3.42.

Figure 3.46 shows a program table for a PLA manufactured by Signetics. This PLA has 16 input variables and 8 outputs from OR gates. Also, 48 AND gates can be formed on the chip. This PLA is packaged in a 28-pin IC container, and the pin-out is shown in Fig. 3.47.

The table in Fig. 3.46 can be filled out to describe a particular gate network and then mailed to an IC manufacturer, who will then produce chips with a gate network corresponding to the table. Although a Signetics table is used here, the table is typical and other manufacturers provide the same service.

The table is filled out as follows:

First, the logic input variables are identified with the INPUT VARIABLE number 0 to 15 on the table. We fill out the table for Fig. 3.42 as a (small) example. To do this, we identify A with input 2, B with input 1, and C with input 0 on the program table. We now wish to form the product terms $AB\overline{C}$, $\overline{A}\overline{B}$, and AC. The rule is that if an input is not complemented (inverted), an H is written in the table; if the input is to be complemented, an L is written; and if the input is not used, a — is written. To form $AB\overline{C}$, then, we form a row in the table containing dashes everywhere except in INPUT VARIABLE columns 2, 1, and 0, and in these we write H, H, and L. This is shown in Fig. 3.48.

The OR gate inputs are written as follows:

If the AND term in a particular row is to be used in an OR output, an A (for

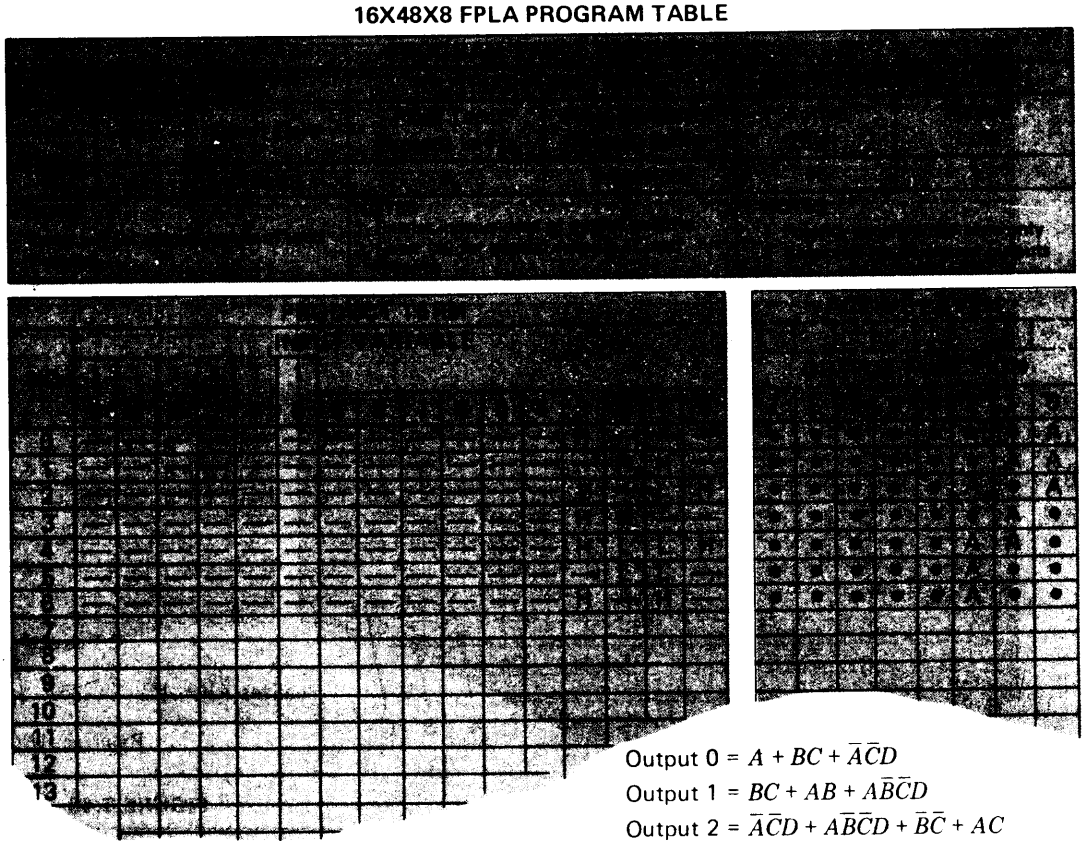


FIGURE 3.49

Program table for three AND-to-OR gate networks.

active) is written in the row; if not, a • is written. In Fig. 3.48, the OUTPUT FUNCTION line 0 in the table is associated with OUTPUT 1 in Fig. 3.42 and OUTPUT FUNCTION line 1 is associated with OUTPUT 2 in Fig. 3.42.

A final example is shown in Fig. 3.49. In this case, there are three output lines 0, 1, and 2. We associated A with INPUT VARIABLE 3 on the table, B with 2, C with 1, and D with 0. The functions formed are

$$\text{OUTPUT LINE 0} = A + BC + \bar{A}\bar{C}D$$
$$\text{OUTPUT LINE 1} = AB + BC + A\bar{B}\bar{C}D$$
$$\text{OUTPUT LINE 2} = \bar{B}\bar{C} + \bar{A}\bar{C}D + A\bar{B}\bar{C}D + AC$$

Clearly PLAs provide a convenient way to fabricate IC chips with gate networks. The fact that field-programmable arrays with a given design can be made by blowing selected fuses and then the computer design can be tested by using these trial chips is very convenient. Later, for production runs, the chips made by a manufacturer from the table can be used.

This part of the IC business is sufficiently developed that PLA designs can be punched into cards and sent to a manufacturer, punched into tapes and sent to a manufacturer, and even sent over the telephone line from a terminal connected to a tape reader. (Manufacturers will provide a long-distance number you can call to phone in designs.)

SUMMARY

3.27 This chapter presented the basic idea of a gate and showed how gates can be interconnected to form logic networks. The basic types of gates were introduced: AND, OR, NAND, and NOR gates and inverters.

The design of logic networks for computers is greatly facilitated by boolean algebra. This subject was introduced, and tables of combination, theorems of the algebra, and algebraic reduction of expressions were all explained.

The map method for simplifying boolean algebra expressions was presented. This makes it possible to minimize boolean expressions, thereby also simplifying the logic networks that realize these expressions. The various network forms such as AND-to-OR, NAND-to-NAND, OR-to-AND, NOR-to-NOR, NAND-to-AND, and NOR-to-OR networks were explained, and the design procedure for each was presented.

Gating networks can now be made with many gates in a single container by using LSI manufacturing techniques. The general layout for several such integrated circuits was presented, as was a special representation which is commonly used. This makes possible design of relatively large arrays of gates using PLAs, PALs, and HALs.

QUESTIONS

3.1 Prepare a truth table for the following boolean expressions:
(*a*) $XYZ + \bar{X}\bar{Y}\bar{Z}$ (*b*) $ABC + A\bar{B}\bar{C} + \bar{A}\bar{B}\bar{C}$
(*c*) $A(B\bar{C} + \bar{B}C)$ (*d*) $(A + B)(A + C)(\bar{A} + \bar{B})$

3.2 Prepare a table of combinations for the following boolean algebra expressions:
(*a*) $\bar{X}\bar{Y} + \bar{X}Y$ (*b*) $XY\bar{Z} + \bar{X}\bar{Y}Z$ (*c*) $\bar{X}Y\bar{Z} + X\bar{Y}$
(*d*) $\bar{X}\bar{Y}\bar{Z} + X\bar{Y}\bar{Z} + \bar{X}Y\bar{Z}$ (*e*) $\bar{X}\bar{Y} + \bar{Y}\bar{Z}$ (*f*) $\bar{A}B(\bar{A}\bar{B}\bar{C} + \bar{B}C)$

3.3 Prepare a truth table for the following boolean expressions:
(*a*) $A\bar{B} + \bar{A}B$ (*b*) $A\bar{B} + B\bar{C}$
(*c*) $A\bar{C} + AC$ (*d*) $\bar{A}\bar{B}C + AB\bar{C} + \bar{A}BC$
(*e*) $AB(\bar{A}\bar{B}C + A\bar{B}\bar{C} + AB\bar{C})$

3.4 Prepare a table of combinations for the following boolean algebra expressions:
(*a*) $X(\bar{Y} + \bar{Z}) + X\bar{Y}$ (*b*) $X\bar{Y}(Z + Y\bar{Z}) + \bar{Z}$
(*c*) $[X(Y + \bar{Y}) + \bar{X}(\bar{Y} + Y)]\cdot\bar{Z}$ (*d*) $AB(\bar{A}B + A\bar{B})$
(*e*) $A[(\bar{B} + C) + \bar{C}]$ (*f*) $\bar{A}\bar{B}\bar{C}(\bar{A}B\bar{C} + A\bar{B}\bar{C})$

3.5 Prepare a table of combinations for the following boolean algebra expressions:
(*a*) $XY + \bar{X}\bar{Y}Z$ (*b*) $ABC + \bar{A}\bar{B} + \bar{A}B$
(*c*) $ABC + \bar{A}\bar{C}$

3

BOOLEAN ALGEBRA AND GATE NETWORKS

3.6 Prepare a table of combinations for the following boolean algebra expressions:
(*a*) $A\bar{B}\bar{C} + \bar{A}B$ (*b*) $\bar{A}\bar{B}\bar{C} + AC + AB$
(*c*) $XZ + X\bar{Y} + \bar{X}\bar{Z}$

3.7 Simplify the following expressions, and draw a block diagram of the circuit for each simplified expression, using AND and OR gates. Assume the inputs are from flip-flops.
(*a*) $A\bar{B}\bar{C} + \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C$
(*b*) $ABC + \bar{A}BC + A\bar{B}C + AB\bar{C} + A\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}\bar{C}$
(*c*) $A(A + B + C)(\bar{A} + B + C)(A + \bar{B} + C)(A + B + \bar{C})$
(*d*) $(A + B + C)(A + \bar{B} + \bar{C})(A + B + \bar{C})(A + \bar{B} + C)$

3.8 Simplify the expressions in Question 3.4 and draw block diagrams of gating networks for your simplified expressions, using AND gates, OR gates, and inverters.

3.9 Simplify the following expressions:
(*a*) $ABC(AB\bar{C} + A\bar{B}C + \bar{A}BC)$ (*b*) $AB + A\bar{B} + \bar{A}C + \bar{A}\bar{C}$
(*c*) $XY + XY\bar{Z} + XY\bar{Z} + XZY$ (*d*) $XY(\bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + \bar{X}\bar{Y}\bar{Z})$

3.10 Simplify the expressions in Question 3.6 and draw block diagrams of gating networks for your simplified expressions, using AND gates, OR gates, and inverters.

3.11 Form the complements of the following expressions. For instance, the complement of $(XY + XZ)$ is equal to $\overline{(XY + XZ)} = (\bar{X} + \bar{Y})(\bar{X} + \bar{Z}) = \bar{X} + \bar{Y}\bar{Z}$.
(*a*) $(A + BC + AB)$ (*b*) $(A + B)(B + C)(A + C)$
(*c*) $AB + \bar{B}C + CD$ (*d*) $AB(\bar{C}D + \bar{B}C)$
(*e*) $A(B + C)(\bar{C} + \bar{D})$

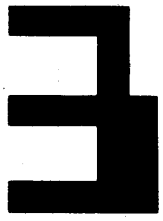
3.12 Complement the following expressions (as in Question 3.11):
(*a*) $\bar{X}\bar{Y} + X\bar{Y}$ (*b*) $X\bar{Y}Z + \bar{X}Y$
(*c*) $\bar{X}(Y + \bar{Z})$ (*d*) $X(Y\bar{Z} + \bar{Y}Z)$
(*e*) $XY(\bar{Y}Z + X\bar{Z})$ (*f*) $XY + \bar{X}\bar{Y}(Y\bar{Z} + \bar{X}\bar{Y})$

3.13 Prove the two basic De Morgan theorems, using the proof by perfect induction.

3.14 Prove the following rules using the proof by perfect induction:
(*a*) $X\bar{Y} + XY = X$
(*b*) $X + \bar{X}Y = X + Y$

3.15 Convert the following expressions to sum-of-products form:
(*a*) $(A + B)(\bar{B} + C)(\bar{A} + C)$
(*b*) $(\bar{A} + C)(\bar{A} + \bar{B} + \bar{C})(A + \bar{B})$
(*c*) $(A + C)(A\bar{B} + AC)(\bar{A}\bar{C} + \bar{B})$

3.16 Convert the following expressions to sum-of-products form:
(*a*) $(\bar{A} + \bar{B})(\bar{C} + B)$ (*b*) $\bar{A}B(\bar{B}C + \bar{B}C)$
(*c*) $(A + B\bar{C})(\bar{A}\bar{B} + \bar{A}B)$ (*d*) $AB(A\bar{B}\bar{C} + \bar{A}C)$
(*e*) $(\bar{A} + B)[A\bar{C} + (B + C)]$ (*f*) $(\bar{A} + C)(AB + \bar{A}\bar{B} + AC)$



QUESTIONS

3.17 Which rule is the dual of rule 12 in Table 3.10?

3.18 Give a dual of the rule of $X + \bar{X}Y = X + Y$.

3.19 Multiply the following sum terms, forming a sum-of-products expression in each case. Simplify while multiplying when possible.

(*a*) $(A + C)(B + D)$
(*b*) $(A + C + D)(B + D + C)$
(*c*) $(AB + C + DC)(AB + BC + D)$
(*d*) $(A\bar{B} + \bar{A}B + A\bar{C})(\bar{A}\bar{B} + A\bar{B} + A\bar{C})$

3.20 Convert the following expressions to product-of-sums form:

(*a*) $A + \bar{A}B + \bar{A}\bar{C}$
(*b*) $BC + \bar{A}B$
(*c*) $A\bar{B}(\bar{B} + \bar{C})$
(*d*) $\bar{A}\bar{B}(\bar{B}\bar{C} + \bar{B}\bar{C})$
(*e*) $(A + \bar{B} + C)(AB + \bar{A}C)$
(*f*) $(\bar{A} + \bar{B})A\bar{B}C$

3.21 Write the boolean expression (in sum-of-products form) for a logic circuit that will have a 1 output when $X = 0, Y = 0, Z = 1$ and $X = 1, Y = 1, Z = 0$; and a 0 output for all other input states. Draw the block diagram for this circuit, assuming that the inputs are from flip-flops.

3.22 Convert the following to product-of-sums form

(*a*) $AB + \bar{A}(B + \bar{C})(D + \bar{B})$
(*b*) $(B + C)[(\bar{B} + \bar{C})(A + \bar{C})(B + C)]$

3.23 Convert the following to product-of-sums form

(*a*) $AB\bar{C} + A\bar{B}C + \bar{A}BC$
(*b*) $ABC + \bar{B}\bar{C}(A + CD)(B + C)$

3.24 Prove the following theorem, using the rules in Table 3.10:

$$(X + Y)(X + \bar{Y}) = X$$

3.25 Write the boolean expression (in sum-of-products form) for a logic network that will have a 1 output when $X = 1, Y = 0, Z = 0$; $X = 1, Y = 1, Z = 0$; $X = 1, Y = 1, Z = 0$; and $X = 1, Y = 1, Z = 1$. The circuit will have a 0 output for all other sets of input values. Simplify the expression derived and draw a block diagram for the simplified expression.

3.26 Derive the boolean algebra expression for a gating network that will have outputs 0 only when $X = 1, Y = 1, Z = 1$; $X = 0, Y = 0, Z = 0$; $X = 1, Y = 0, Z = 0$. The outputs are to be 1 for all other cases.

3.27 Prove rule 18 in Table 3.10, using the proof by perfect induction.

3.28 Develop sum-of-products and product-of-sums expressions for F_1, F_2, and F_3 in Table 3.27.

3.29 Develop both the sum-of-products and the product-of-sums expressions that describe Table 3.28. Then simplify both expressions. Draw a block diagram for logical circuitry that corresponds to the simplified expressions, using only NAND gates for the sum-of-products and NOR gates for product-of-sums expression.

TABLE 3.27

INPUTS			OUTPUTS		
X	Y	Z	F_1	F_2	F_3
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	1	1	1
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	1	1	1
1	1	1	1	0	1

TABLE 3.28

INPUTS			OUTPUT
X	Y	Z	A
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

3.30 Draw block diagrams for the F_1, F_2, and F_3 in Question 3.28, using only NAND gates.

3.31 Write the boolean algebra expressions for Tables 3.29 to 3.31, showing expressions in sum-of-products form. Then simplify the expressions and draw a block diagram of the circuit corresponding to each expression.

3.32 Draw block diagrams for the F_1, F_2, and F_3 in Question 3.28, using only NOR gates.

3.33 Draw block diagrams for F_1, F_2, and F_3 in Question 3.28, using OR-to-NAND networks.

3.34 Draw block diagrams for F_1, F_2, and F_3 in Question 3.28, using AND-to-NOR gate networks.

3.35 Draw Karnaugh maps for the expressions in Question 3.2.

3.36 Draw Karnaugh maps for the expressions in Question 3.3.

3.37 For a four-variable map in W, X, Y, and Z draw the subcubes for:
(*a*) $WX\overline{Y}$ (*b*) WX (*c*) $XY\overline{Z}$ (*d*) Y

3.38 For a four-variable map in W, X, Y, and Z draw the subcubes for:
(*a*) $\overline{W}X\overline{Y}\overline{Z}$ (*b*) $W\overline{Z}$ (*c*) $\overline{W}Z$ (*d*) $\overline{Y}$

TABLE 3.29

INPUTS			OUTPUT
A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

3.39 Draw maps of the expressions in Question 3.40; then draw the subcubes for the shortened terms you found.

3.40 Apply the rule $AY + A\bar{Y} = A$, where possible, to the following expressions:

(*a*) $X\bar{Y} + \bar{X}\bar{Y}$ (*b*) $\bar{A}\bar{B}C + A\bar{B}\bar{C}$

(*c*) $A\bar{B}C + ABC$ (*d*) $ABC + A\bar{B}\bar{C} + A\bar{B}C + AB\bar{C}$

(*e*) $ABC + \bar{A}\bar{B}\bar{C} + A\bar{B}C$ (*f*) $ABC + \bar{A}BC + \bar{A}\bar{B}\bar{C}$

Note: There is a technique for writing minterms that is widely used. It consists in writing the letter *m* (to represent *minterm*) along with the value of the binary number given by the row of the table and combinations in which the minterm lies. For instance, in the variables *X*, *Y*, *Z* we have the unfinished table of combinations given in Table 3.32.

TABLE 3.30

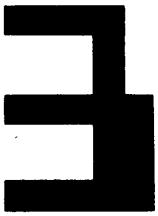
INPUTS			OUTPUT
A	B	C	Z
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

TABLE 3.31

INPUTS			OUTPUT
X	Y	Z	P
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

TABLE 3.32

INPUT			OUTPUT	PRODUCT TERMS	DESIGNATION AS m_i
X	Y	Z			
0	0	0		$\bar{X}\bar{Y}\bar{Z}$	m_0
0	0	1		$\bar{X}\bar{Y}Z$	m_1
0	1	0		$\bar{X}Y\bar{Z}$	m_2
0	1	1		$\bar{X}YZ$	m_3
1	0	0		$X\bar{Y}\bar{Z}$	m_4
1	0	1		$X\bar{Y}Z$	m_5
1	1	0		$XY\bar{Z}$	m_6
1	1	1		XYZ	m_7



BOOLEAN ALGEBRA AND GATE NETWORKS

For this table $m_0 = \bar{X}\bar{Y}\bar{Z}$, $m_1 = \bar{X}\bar{Y}Z$, $m_2 = \bar{X}Y\bar{Z}$, $m_3 = \bar{X}YZ$, and so to $m_7 = XYZ$. Now we can substitute m_i's for actual terms and shorten the writing of expressions. For instance, $m_1 + m_2 + m_4$ means $\bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z}$. Similarly, $m_0 + m_3 + m_5 + m_7$ means $\bar{X}\bar{Y}\bar{Z} + \bar{X}YZ + X\bar{Y}Z + XYZ$.

This can be extended to four or more variables. An expression in W, X, Y, Z can be written as $m_0 + m_{13} + m_{15} = \bar{W}\bar{X}\bar{Y}\bar{Z} + WX\bar{Y}Z + WXYZ$. Or $m_2 + m_5 + m_9 = \bar{W}\bar{X}Y\bar{Z} + \bar{W}X\bar{Y}Z + W\bar{X}\bar{Y}Z$. As can be seen, to change a minterm to its m_i, simply make uncomplemented variables 1s and complemented variables 0s. Thus $\bar{W}XY\bar{Z}$ would be 0110, or 6 decimal; $\bar{W}X\bar{Y}\bar{Z}$ would be 0010, or 2 decimal. These two terms would then be written m_6 and m_2. (Notice that we must know how many variables a minterm is in.)

3.41 Draw the Karnaugh maps in X, Y, Z for:

(*a*) $m_0 + m_1 + m_5 + m_7$ (*b*) $m_1 + m_3 + m_5 + m_4$
(*c*) $m_1 + m_2 + m_3 + m_5$ (*d*) $m_0 + m_5 + m_7$

3.42 Draw the subcubes for a three-variable map in X, Y, Z for:

(*a*) $m_1 + m_3 + m_5 + m_0$ (*b*) $m_4 + m_7$ (*c*) $m_0 + m_3$

3.43 Find the maximal subcubes for the maps drawn for Question 3.42.

3.44 Find minimal expressions for the maps drawn in Question 3.42.

3.45 Using maps, simplify the following expressions in four variables, W, X, Y, and Z:

(*a*) $m_2 + m_3 + m_5 + m_6 + m_7 + m_9 + m_{11} + m_{13}$
(*b*) $m_0 + m_2 + m_4 + m_8 + m_9 + m_{10} + m_{11} + m_{12} + m_{13}$

3.46 Using maps, simplify the following expressions in four variables W, X, Y, and Z:

(*a*) $m_1 + m_3 + m_5 + m_7 + m_{12} + m_{13} + m_8 + m_9$
(*b*) $m_0 + m_5 + m_7 + m_8 + m_{11} + m_{13} + m_{15}$

3.47 Using maps, derive minimal product-of-sums expressions for the functions given in Question 3.46.

3.48 Using maps, derive minimal product-of-sums expressions for the functions given in Question 3.42.

3.49 Using maps, simplify the following expressions, using sum-of-products form:

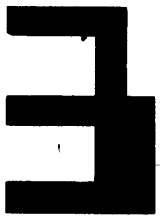
(*a*) $\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + \overbrace{ABC + \bar{A}B\bar{C} + \bar{A}\bar{B}C}^{\text{don't-cares}}$

(*b*) $ABC + \bar{A}\bar{B}\bar{C} + \overbrace{AB\bar{C} + A\bar{B}C}^{\text{don't-cares}}$

(*c*) $ABCD + \bar{A}\bar{B}\bar{C}D + \bar{A}BCD + \overbrace{A\bar{B}CD + \bar{A}\bar{B}CD + AB\bar{C}\bar{D}}^{\text{don't-cares}}$

3.50 Using maps, derive minimal product-of-sums expressions for the functions given in Question 3.49.

3.51 Using maps, simplify the following expressions, using sum-of-products form:



(a) $ABC + \overline{A}\overline{B}C + \overbrace{\overline{A}BC + A\overline{B}C + AB\overline{C}}^{\text{don't-cares}}$

(b) $ABCD + \overline{A}\overline{B}CD + \overbrace{\overline{A}BCD + A\overline{B}CD + \overline{A}B\overline{C}D}^{\text{don't-cares}}$

(c) $A\overline{B}C\overline{D} + \overline{A}\overline{B}CD + \overbrace{\overline{A}B\overline{C}D + \overline{A}BC\overline{D}}^{\text{don't-cares}}$

3.52 (*a*) Design an AND-to-OR gate combinational network for the boolean algebra expression

$$\text{ABCD} + AB\overline{C}\overline{D} + \overline{A}\overline{B}CD + \overline{A}B\overline{C}D + ABC\overline{D} + \overline{A}BCD$$

Use as few gates as you can.

(*b*) Design a NOR gate combinational network for the boolean algebra function in part (*a*), again using as few gates as you can.

3.53 The following is a NAND-to-NAND gate network. Draw a block diagram for a NOR-to-NOR gate network that realizes the same function, using as few gates as possible.

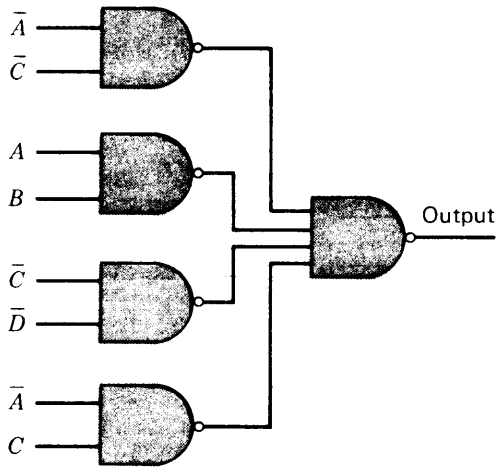


FIGURE Q3.53

3.54 (*a*) Derive a boolean algebra expression for the output Y of the network shown.

(*b*) Convert the expression for Y derived in (*a*) to product-of-sums form.

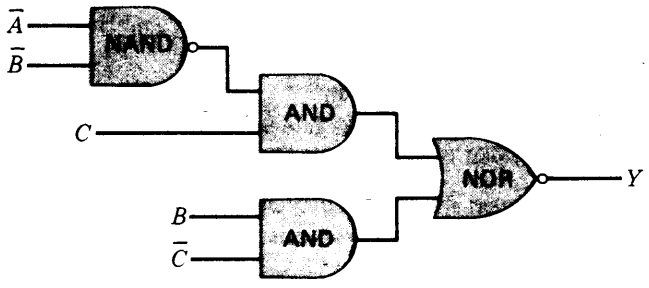
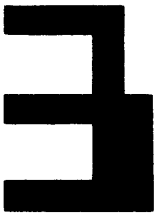


FIGURE Q3.54



3.55 (*a*) Design an OR-to-AND gate combinational network for the boolean algebra expression

$$ABCD + \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + \bar{A}BC\bar{D} + (\bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}CD)$$

The two terms in parentheses are don't-care terms.

(*b*) Using only NAND gates, design a combinational network for the boolean algebra function given in part (*a*).

3.56 (*a*) Design an OR-to-AND gate combinational network for the boolean algebra expression

$$\bar{A}BCD + \bar{A}BC\bar{D} + \bar{A}B\bar{C}D + \bar{A}B\bar{C}\bar{D} + \bar{A}\bar{B}CD)$$

The two terms in parentheses are don't-care terms.

(*b*) Using only NOR gates, design a combinational network for the boolean algebra function given in part (*a*).

3.57 The following NAND-to-AND gate network is to be redesigned using a NOR-to-OR gate configuration. Make the change, using as few gates as possible.

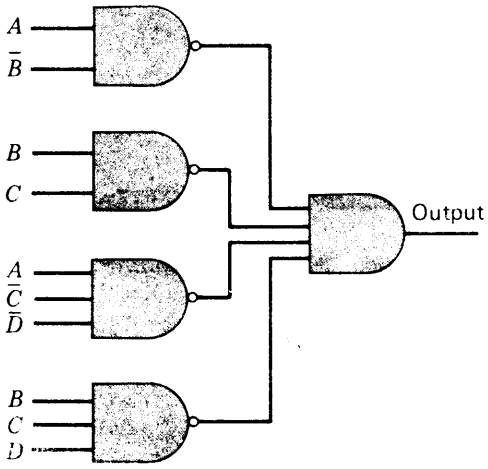


FIGURE Q3.57

3.58 (*a*) Design an AND-to-OR gate combinational network for the boolean algebra expression

$$ABCD + AB\bar{C}D + \bar{A}\bar{B}C\bar{D} + AB\bar{C}\bar{D} + AB\bar{C}\bar{D} + \bar{A}B\bar{C}\bar{D}$$

Use as few gates as you can.

(*b*) Design a NOR gate combinational network for the boolean algebra function in part (*a*), again using as few gates as you can.

3.59 Convert the following NOR-to-OR gate network to a NAND-to-AND gate network. Use as few gates as possible.

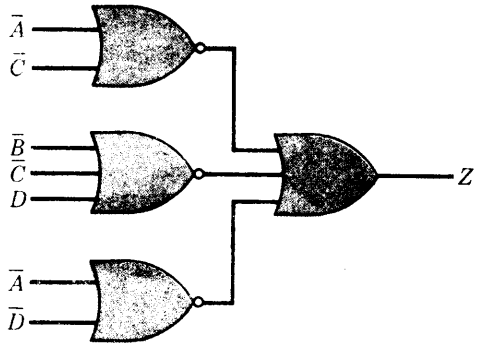


FIGURE Q3.59

3.60 (*a*) Design an AND-to-OR gate combinational network for the boolean algebra expression

$$A\overline{B}CD + AB\overline{C}D + \overline{A}\overline{B}CD + AB\overline{C}\overline{D} + A\overline{B}\overline{C}\overline{D} + \overline{A}BCD$$

Use as few gates as you can.

(*b*) Design a NOR-to-NOR gate combinational network for the boolean algebra function in part (*a*), again using as few gates as you can.

3.61 A combinational network has three control inputs C_1, C_2, and C_3; three data inputs A_1, A_2, and A_3; and a single output Z. (And $\overline{A}_1$, $\overline{A}_2$, $\overline{A}_3$, $\overline{C}_1$, $\overline{C}_2$, and $\overline{C}_3$ are also available as inputs.) Each input is a binary-valued signal. Only one of the control inputs can be a 1 at any given time, and all three can be 0s simultaneously. When C_1 is a 1, the value of Z is to be the value of A_1; when C_2 is a 1, the value of Z is to be the value of A_2; and when C_3 is a 1, the value of the output is to be the value of A_3. If C_1, C_2, and C_3 are 0s, the output Z is to have value 0. Design this network, using only NOR gates. Make the network have two levels, and use as few gates as possible.

3.62 The following NAND-to-AND gate network is to be redesigned by using a NOR-to-OR gate configuration. Make the change, using as few gates as possible.

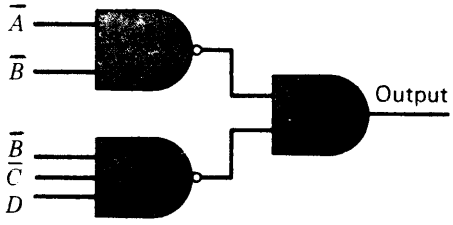


FIGURE Q3.62

3.63 Convert the following NOR-to-NOR gate network to a NAND-to-AND gate network. Use as few gates as possible.

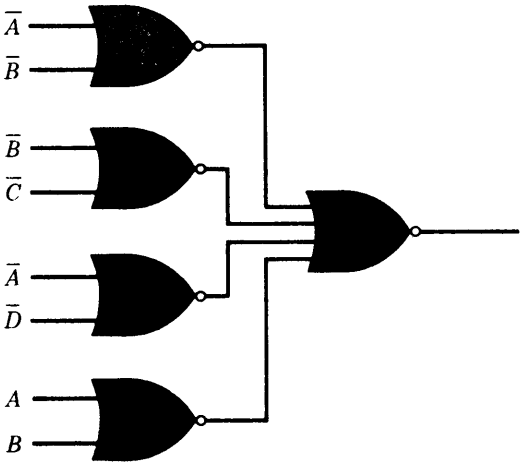


FIGURE Q3.63

3.64 Will the minimal expression of the function in Table 3.33 require fewer NAND gates or NOR gates? (d means don't-care.) Assume complements are available. How many gates for each? Give your minimal expressions.

3.65 The following NAND-to-AND gate network must be converted to a NOR-to-OR gate network. Make the conversion, using as few gates as possible in your final design.

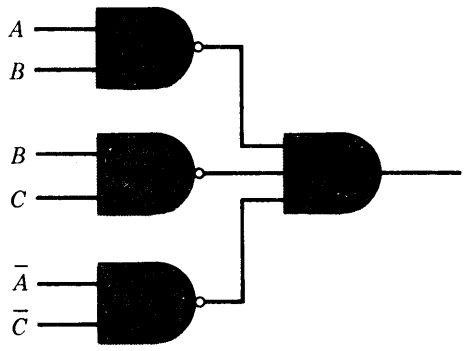


FIGURE Q3.65

3.66 Will the minimal expression for the function in Table 3.34 require fewer NAND gates or NOR gates? (d means don't-care.) Assume complements are available. How many gates for each? Give your minimal expressions.

3.67 Simplify:

(*a*) $(\overline{W} + \overline{X} + Y + Z)\,(\overline{W} + X + \overline{Y} + Z)\,(\overline{W} + X + Y + Z)$

$$\overbrace{(W + X + \overline{Y} + Z)\,(W + \overline{X} + Y + Z)\,(\overline{W} + \overline{X} + Y + \overline{Z})\,(W + X + Y + Z)}^{\text{don't-cares}}$$

(*b*) $\overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}B\overline{C}D + ABC\overline{D} +$

$$\overbrace{\overline{A}\overline{B}C\overline{D} + \overline{A}BCD + ABCD + \overline{A}BC\overline{D} + A\overline{B}C\overline{D}}^{\text{don't-cares}}$$

(*c*) For parts (*a*) and (*b*), design block diagrams for the logical circuitry of

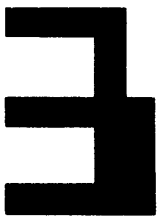


TABLE 3.33

X_1	X_2	X_3	X_4	OUTPUT
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	d
0	1	1	0	1
0	1	1	1	d
1	0	0	0	d
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	d
1	1	0	1	0
1	1	1	0	d
1	1	1	1	0

TABLE 3.34

X_1	X_2	X_3	X_4	OUTPUT
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	d
1	0	0	0	d
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	d
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

the simplified expressions, using either NAND gates only or NOR gates only. Assume that complements of the inputs are available. The same type gates do not have to be used for both (*a*) and (*b*).

3.68 Write a boolean algebra expression in sum-of-products form for a gating network with three inputs A, B, and C (and their complements $\overline{A}$, $\overline{B}$, and $\overline{C}$) that is to have a 1 output only when two or three of the inputs have a 1 value. Implement, using a NAND-to-wired-AND gate network.

3.69 Draw a block diagram for a gate network having a NOR-to-OR gate network with three inputs, A, B, and C (and their complements) that have a 1 output only when two or three of the inputs have a 1 value, as in Question 3.68.

3.70 Will the minimal expression for the function in Table 3.35 require fewer NAND gates or NOR gates? (d means don't-care.) Assume that complements are available. How many gates for each? Give your minimal expressions.

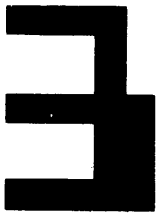


TABLE 3.35				
X_1	X_2	X_3	X_4	OUTPUT
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	d
0	1	0	1	0
0	1	1	0	1
0	1	1	1	d
1	0	0	0	d
1	0	0	1	1
1	0	1	0	1
1	0	1	1	d
1	1	0	0	d
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

3.71 The following is a NAND-to-NAND gate network. Draw a block diagram for a NOR-to-NOR gate network that realizes the same function, using as few gates as possible.

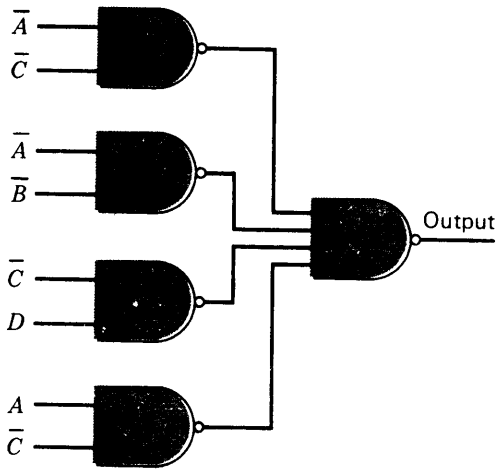


FIGURE Q3.71

3.72 Convert the following NOR-to-OR gate network to a NAND-to-NAND gate network. Use as few gates as possible.

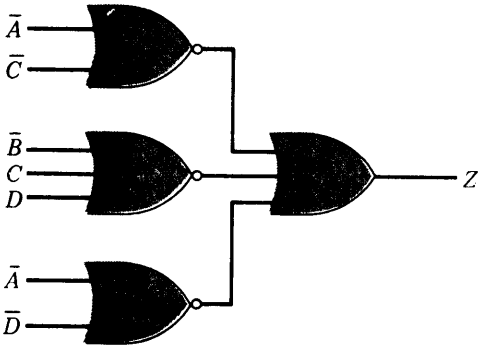
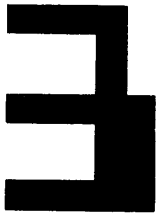


FIGURE Q3.72



QUESTIONS

3.73 (*a*) Design an AND-to-OR gate combinational network for the boolean algebra function.

$$F = \overline{W}\overline{X}\overline{Y}\overline{Z} + W\overline{X}\overline{Y}Z + W\overline{X}Y\overline{Z} + W\overline{X}YZ + \overline{W}X\overline{Y}\overline{Z}$$

Use as few gates as you can.

(*b*) Design a NOR gate combinational network for the boolean algebra function in part (*a*), again using as few gates as you can.

3.74 This chapter has explained a number of two-level networks that can be used to implement all possible functions of a given number of variables. There are also two-level networks that can implement only a few of the many functions possible. For instance, an AND-to-AND gate network is only the AND function as shown below:

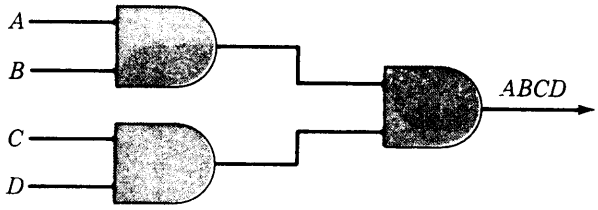


FIGURE Q3.74a

Similarly, a NAND-to-OR implements only an OR function with complemented inputs as shown below:

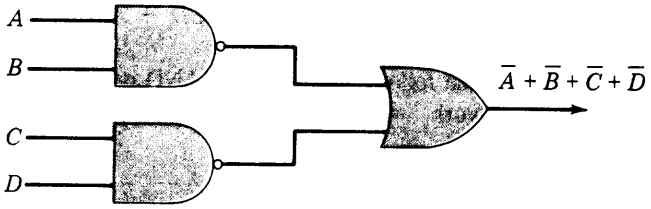


FIGURE Q3.74b

In all, 8 of the possible 16 two-level network arrangements that can be made with NOR, NAND, OR, and AND gates will realize all functions, while 8 are degenerate and yield only a few of the functions. Identify the degenerate forms and the forms that will yield all functions.

3.75 This chapter did not treat the two-level AND-to-NOR form. Derive a rule for designing AND-to-NOR gate networks, and show how it works for a problem of your choice.

3.76 This chapter did not treat OR-to-NAND gate networks, although all boolean functions can be realized by using that configuration. Derive a sample network, using your rule.

3.77 Show how the NOR-to-NAND gate network shown at the top of page 132 can be replaced by a single gate.

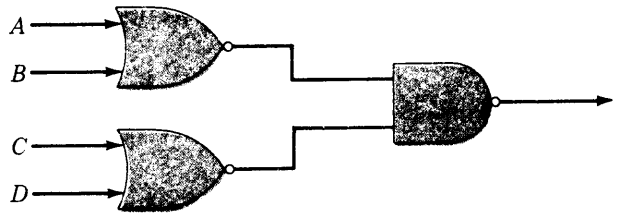


FIGURE Q3.77

3.78 Convert the following NAND-to-NAND gate network to a (two-level) NOR-to-OR gate network:

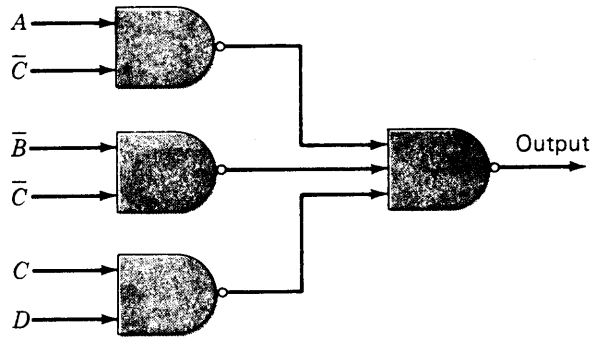


FIGURE Q3.78

3.79 Using as few gates as possible, design a NAND-to-AND gate network that realizes the following boolean algebra expression:

$$\overline{A}B\overline{C}D + AB\overline{C}\,\overline{D} + A\overline{B}C\overline{D} + ABC\overline{D} + A\overline{B}\,\overline{C}\,\overline{D}$$

3.80 Convert the following NAND-to-NAND gate network to a (two-level) NOR-to-OR gate network:

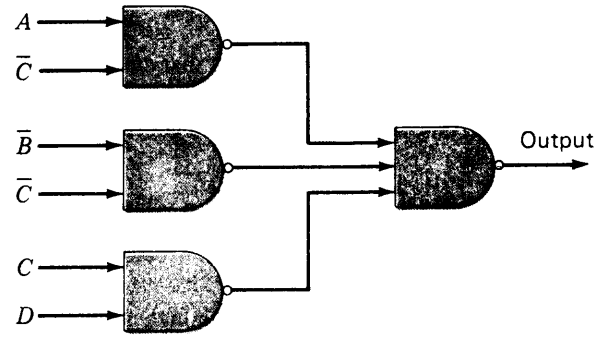


FIGURE Q3.80

3.81 Convert the following NAND-to-NAND gate network to a NOR-to-NOR gate network:

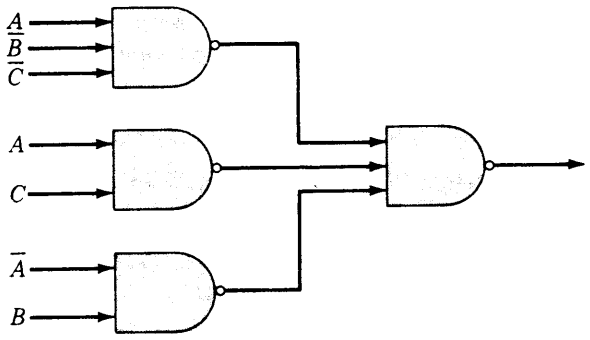


FIGURE Q3.81

3.82 Convert the following NOR-to-OR gate network to a NAND-to-AND gate network. Use as few gates as possible.

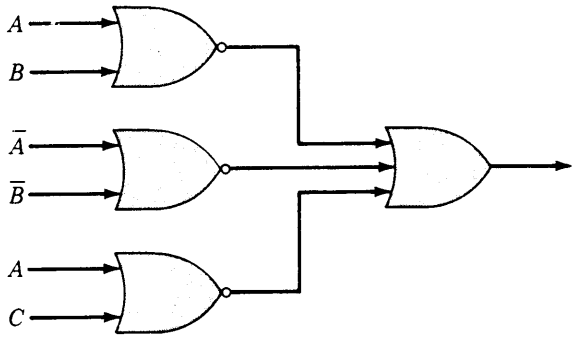


FIGURE Q3.82

3.83 Convert the following NAND-to-AND gate network to a NOR-to-OR gate network. Use as few gates as possible.

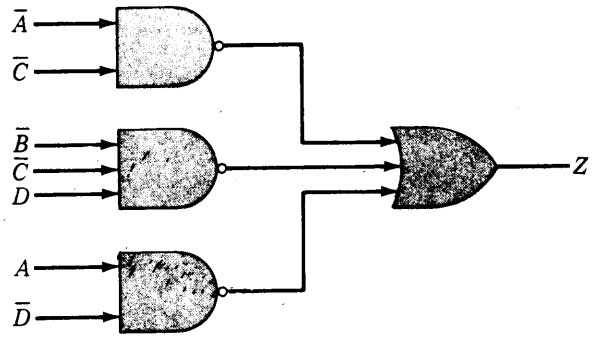
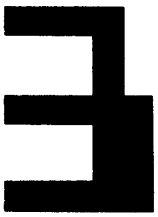


FIGURE Q3.83

3.84 Using a PLA table as in Fig. 3.46, design a three-input, six-output gate network that squares each input. (Inputs and outputs are unsigned binary integers.)

3.85 Using a PLA table as in Fig. 3.46, design a network that forms the 9s complement of a BCD number in 2, 4, 2, 1 form.



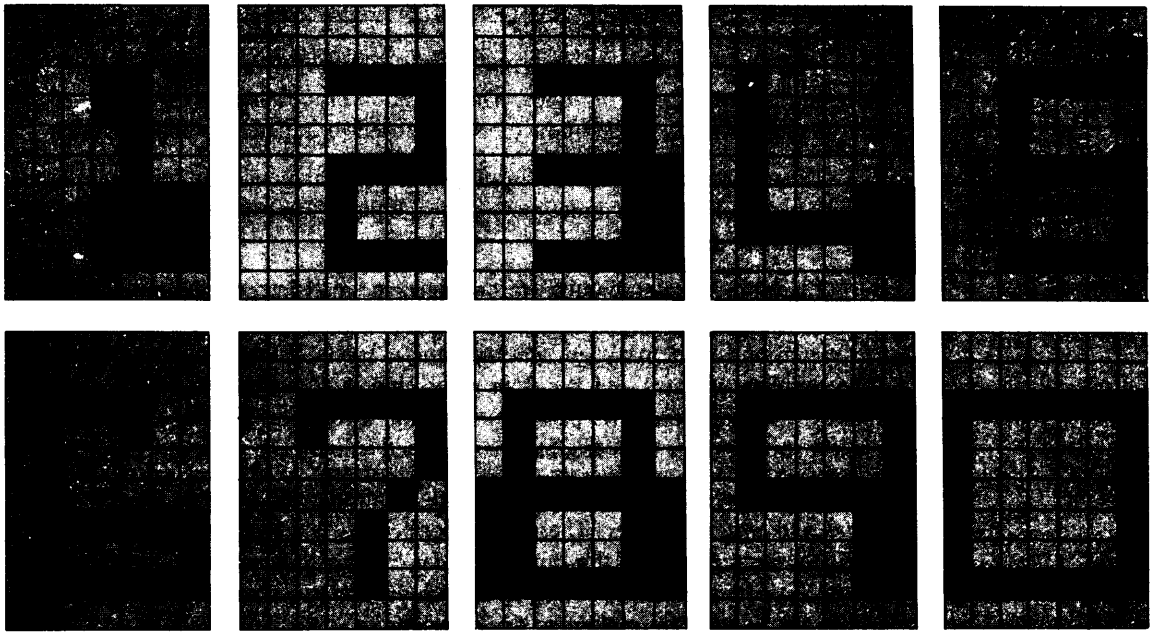
3.86 Using the PAL in Fig. 3.44, design a three-input gate network which finds the 2s complement of a positive input number.

3.87 Using the PLA symbology in Fig. 3.43, design a logic network that converts a BCD number in 2, 4, 2, 1 form to 8, 4, 2, 1 form.

3.88 Using a PLA table as in Fig. 3.46, design a four-input, five-output circuit that adds 3 to a BCD number in 8, 4, 2, 1 form.

3.89 Using the PLA in Fig. 3.41, show how to form the two outputs $X = \overline{A}\overline{B}C + A\overline{B}\overline{C}$ and $Y = \overline{A}\overline{B}C + AB$.

3.90 Using the symbology in Fig. 3.43, form a design for a PLA with two outputs $X = AB\overline{C} + A\overline{B}$ and $Y = AB\overline{C} + \overline{A}\overline{B}C + \overline{A}B$.



LOGIC DESIGN

Chapter 3 described gates and the analysis of gating networks by using boolean algebra. The basic devices used in the operational or calculating sections of digital computers consist of gates and devices called *flip-flops*. It is remarkable that even the largest of computers is primarily constructed of these devices. Accordingly, this chapter first describes flip-flops and their characteristics. From an intuitive viewpoint, flip-flops provide memory and gates provide operations on, or functions of, the values stored in these memory devices.

Following the introduction to flip-flops, the use of flip-flops and gates to perform several of the most useful functions in computers is presented. The particular functions described include counting in binary and binary-coded decimal, transferring values, and shifting or scaling values stored in flip-flops.

Several other names have been used instead of *flip-flop*. These include *binary* and *toggle*, but flip-flop has been the most frequently used. Also, there are several other types of memory devices in computers, and these are studied in Chap. 6. For actual operations, flip-flops remain dominant, however, because of their high speed, the ease with which they can be set or read, and the natural way gates and flip-flops can be interconnected.

This chapter also contains a section on clocks in digital computers. Computers do not run by taking steps at random times, but proceed from step to step at intervals precisely controlled by a clock which provides a carefully regulated time base for all operations. Some knowledge of the uses of clocks in computers is indispensable, and the subject is introduced here.

OBJECTIVES

1 The basic memory cells in the operational part of a digital computer are electronic circuits called flip-flops. The operation of these devices is described, as are the details of clocks and how clocks are used to initiate flip-flop operation.

2 Binary counters consist of flip-flops and gates and count up (or down) at the direction of a clock and (sometimes) control inputs. The basic types of binary and BCD counters are described.

3 Electronic digital circuits are packaged in small integrated-circuit (IC) containers. The number of gates and flip-flops in a single package determines the level of integration for the package, and the various levels of integration are discussed. (The number of gates and flip-flops on a chip is sometimes called the *packing density*.) Several actual IC packages from the most used IC lines are presented. Then a shift register with feedback is designed using these packages.

4 Two representation techniques for digital design and analysis are called *state tables* and *state diagrams*. These are described, as is a procedure for designing a particular arrangement of flip-flops and gates called a *state machine*. Several design examples are presented, followed by an introduction to a class of integrated circuits which can be used to implement these designs.

FLIP-FLOPS

4.1 The basic circuit for storing information in a digital machine is called a *flip-flop*. There are several fundamental types of flip-flops and many circuit designs. However, two characteristics are shared by all flip-flops.

1 The flip-flop is a bistable device, that is, a circuit with only two stable states, which we designate the 0 state and the 1 state.

The flip-flop circuit can remember, or store, a binary bit of information because of its bistable characteristic. The flip-flop responds to inputs. If an input causes it to go to its 1 state, it will remain there and "remember" a 1 until some signal causes it to go to the 0 state. Similarly, once placed in the 0 state, the flip-flop will remain there until it is told to go to the 1 state. This simple characteristic, the ability of the flip-flop to retain its state, is the basis for information storage in the operating or calculating sections of a digital computer.

2 The flip-flop has two output signals, one of which is the complement of the other.

Figure 4.1 shows the block diagram for a particular type of flip-flop, the *RS flip-flop*. There are two inputs, designated S and R, and two outputs, marked with X and $\overline{X}$. To describe and analyze flip-flop operation, there are several conventions that are standard in the computer industry.

1 Each flip-flop is given a "name." Convenient names are letters, such as X or Y or A or B; or letter-number combinations, such as A_1 or B_2; or sometimes,

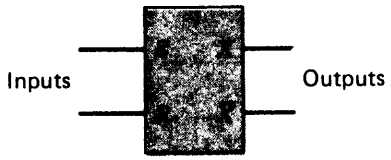


FIGURE 4.1

RS flip-flop.

because of difficulty in subscripting on typewriters or printers, simply $A1$ or $B2$. The flip-flop in Fig. 4.1 is called X. It has two outputs, the X output and the $\overline{X}$ output.

The X and $\overline{X}$ output lines are always complements; that is, if the X output line has a 1 signal, the $\overline{X}$ output line has a 0 signal; and if the X output line has a 0 signal, output line $\overline{X}$ has a 1 signal.

2 The state of the flip-flop is taken to be the state of the X output. Thus if the output line X has a 1 signal on it, we say that "flip-flop X is in the 1 state." Similarly, if the X line contains a 0 signal, we say that "flip-flop X is in the 0 state."

These conventions are very important and convenient. Note that when flip-flop X is in the 1 state, the output line $\overline{X}$ has a 0 on it; and when flip-flop X is in the 0 state, the output line $\overline{X}$ has a 1 on it.

There are two input lines to the *RS* flip-flop. These are used to control the state of the flip-flop. The rules are as follows:

1 As long as both input lines S and R carry 0 signals, the flip-flop remains in the same state, that is, it does not change state.

2 A 1 signal on the S line (the SET line) and a 0 signal on the R line cause the flip-flop to "set" to the 1 state.

3 A 1 signal on the R line (the RESET line) and a 0 signal on the S line cause the flip-flop to "reset" to the 0 state.

4 Placing a 1 on the S and a 1 on the R lines at the same time is forbidden. If this occurs, the flip-flop can go to either state. (This is, in effect, an ambiguous input in that it is telling the flip-flop to both SET and RESET at the same time.)

An example of a possible sequence of input signals and the resulting state of the flip-flop is as follows:

S	R	X	X is the state of the flip-flop after inputs S and R are applied.
1	0	1	
0	0	1	Flip-flop remains in same state.
0	0	1	
0	1	0	Flip-flop is reset.
0	0	0	
0	0	0	
0	1	0	Flip-flop is told to reset but is already reset.
0	0	0	
1	0	1	Flip-flop is set.
0	0	1	

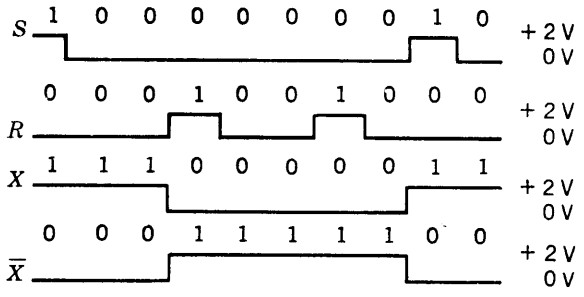


FIGURE 4.2

RS flip-flop waveforms.

Although the above conventions may seem formidable at first, they can be simply summarized by seeing that a 1 on the S line causes the flip-flop to SET (that is, assume the 1 state) and a 1 on the R line causes the flip-flop to RESET (that is, assume the 0 state). The flip-flop does nothing in the absence of 1 inputs and would be hopelessly confused by 1s on both S and R inputs.

It is very convenient to be able to draw graphs of the inputs and outputs from computer circuits to show how they act as inputs vary. We assume the convention that a 1 signal is a positive signal and a 0 signal a ground, or 0-V, signal. This is conventional in most present-day circuits and is called *positive logic*. Figure 4.2 shows several signals as they progress in time, with the current binary values of each signal written above it. The signals in Fig. 4.2 are the sequence of signals given in the list above along with both the X and $\overline{X}$ output line signals from the flip-flop. We have arbitrarily chosen +2 V for the 1 state of the signals and 0 V for the 0 state because these are very frequently used levels. Notice that the flip-flop changes only when the input levels command it to, and that it changes at once. (Actually, there would be a slight delay from when the flip-flop is told to change states and when it changes, since no physical device can respond instantly; so we assume that the flip-flop's delay in responding is quite small, perhaps a small fraction of a microsecond.)

TRANSFER CIRCUIT

4.2 The *RS* flip-flop, although simple in operation, is adequate for all purposes and is a basic flip-flop circuit. Let us examine the operation of this flip-flop in a configuration called a *transfer circuit*. Figure 4.3 shows two sets of flip-flops named X_1, X_2, and X_3 and Y_1, Y_2, and Y_3. The function of this configuration is to transfer the states, or *contents*, of Y_1 into X_1, Y_2 into X_2, and Y_3 into X_3 upon the TRANSFER command which consists of a 1 on the TRANSFER line.

Assume that Y_1, Y_2, and Y_3 have been set to some states that we want to remember, or store, in X_1, X_2, and X_3, while the Y flip-flops are used for further calculations. Placing a 1 on the TRANSFER line will cause this desired transfer of information. Understanding the transfer of the state of Y_1 into X_1 depends on seeing that if Y_1 is in the 0 state, the Y_1 output line has a 0 on it, and so the input line connected to the AND gate will be a 0 and the AND gate will place a 0 on the S input line of X_1, while the $\overline{Y}_1$ output from Y_1 will be a 1, causing, in the presence of a 1 on the TRANSFER line, a 1 on the R input of X_1. Similar reasoning

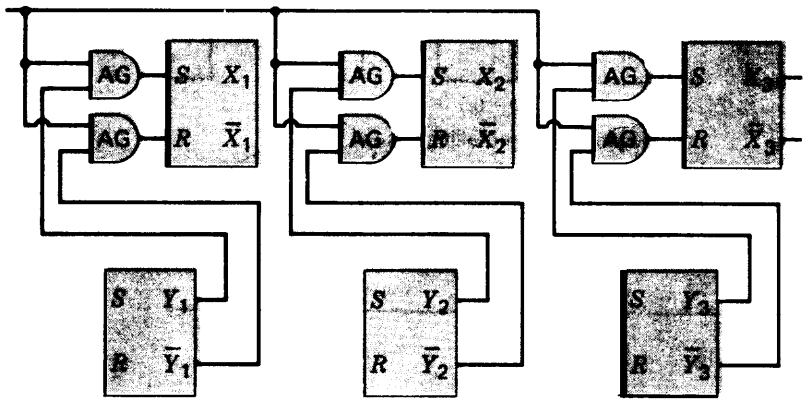


FIGURE 4.3

Transfer circuit.

will show that a 1 in Y_1 will cause a 1 to be placed in X_1 in the presence of a 1 on the TRANSFER line. As long as the TRANSFER line is a 0, both inputs to the X flip-flops will be 0s, and the flip-flop will remain in the last state it assumed.

The above simple operation, the *transfer operation*, is quite important. Related sets of flip-flops in a computer are called *registers*, and the three flip-flops Y_1, Y_2, and Y_3 would be called simply *register Y*, and the three flip-flops, X_1, X_2, and X_3 would be called *register X*. Then a 1 on the TRANSFER line would transfer the contents of register Y into register X. This is an important concept.

CLOCKS

4.3 A very important fact about digital computers is that they are clocked. This means that there is some "master clock" somewhere sending out signals which are carefully regulated in time. These signals initiate the operations performed.

There are excellent reasons why computers are designed this way. The alternative way, with operations triggering other operations as they occur, is called *asynchronous operation* (the clocked way is called *synchronous operation*) and leads to considerable difficulty in design and maintenance. As a result, genuinely asynchronous operation is rarely used.

The clock is, therefore, the mover of the computer in that it carefully measures time and sends out regularly spaced signals which cause things to happen. We can examine the operation of the flip-flops and gates before and after the clock "initiates an action." Initiating signals are often called, for historical reasons, *clock pulses*.[1]

Figure 4.4 shows a typical clock waveform. The clock waveform in Fig. 4.4(*a*) and (*b*) is called a *square wave*. The figure shows two important portions of a square wave: the *leading edge*, or *rising edge*, or sometimes *positive-going edge;* and the *falling edge*, or *negative-going edge*. These are particularly important

[1]The term *clock pulse* has a historical origin. The early computers used short electric pulses to initiate operations, and these were naturally called *clock pulses*. Few circuits still use these narrow pulses, and the majority of circuits now respond to edges of square waves as in Fig. 4.4.

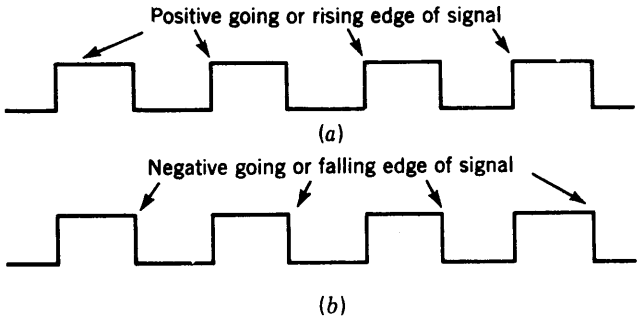


FIGURE 4.4

Clock waveforms.

since most flip-flops now in use respond to either (but not both) a falling edge or a rising edge. In effect, a system which responds to rising edges of the clock "rests" between such edges and changes state only when such positive-going edges occur. (The reason for the rest periods is to give the circuits time to assume their new states and to give all transients time to die down. The frequency at which such edges occur is generally determined by the speed with which the circuits can go to their new states, the delay times for the gates which must process the new signals, etc.)

Since clock signals are used to initiate flip-flop actions, a clock input is included on most flip-flops. This input is marked with a small triangle, as shown in Fig. 4.5(*a*). A clocked flip-flop can respond to either the positive-going edge of the clock signal or the negative-going edge.[2] If a given flip-flop responds to the positive-going edge of the signal, there is no "bubble" at the triangle or clock input on the block diagram, as in Fig. 4.5(*a*). If the flip-flop responds to a negative-going edge, or signal, a bubble is placed at the clock input, as in Fig. 4.5(*b*).

Sometimes the clock input is simply marked with a CL instead of the triangle. Manufacturers who adopt this practice will explain whether the flip-flop is or is not edge-triggered in the specifications sheet for the flip-flop.

It is important to understand the above convention because most clocked flip-flops actually respond to a *change* in clock input level, not to the level itself. This is shown in Fig. 4.5(*c*) and (*d*). The flip-flop in Fig. 4.5(*a*) responds to positive-going clock edges (positive shifts), and a typical set of signals for the *clocked RS flip-flop* in Fig. 4.5(*a*) is shown in Fig. 4.5(*c*).

The flip-flop is operated according to these rules:

1 If the S and R inputs are 0s when the clock edge (pulse) occurs, the flip-flop does not change states but remains in its present state.

2 If the S input is a 1 and the R input is a 0 when the clock pulse (positive-going edge) occurs, the flip-flop goes to the 1 state.

[2]A flip-flop which responds to a rising or falling clock signal (as opposed to responding to a dc level) is called an *edge-triggering*, or *master-slave*, flip-flop for reasons that will be explained.

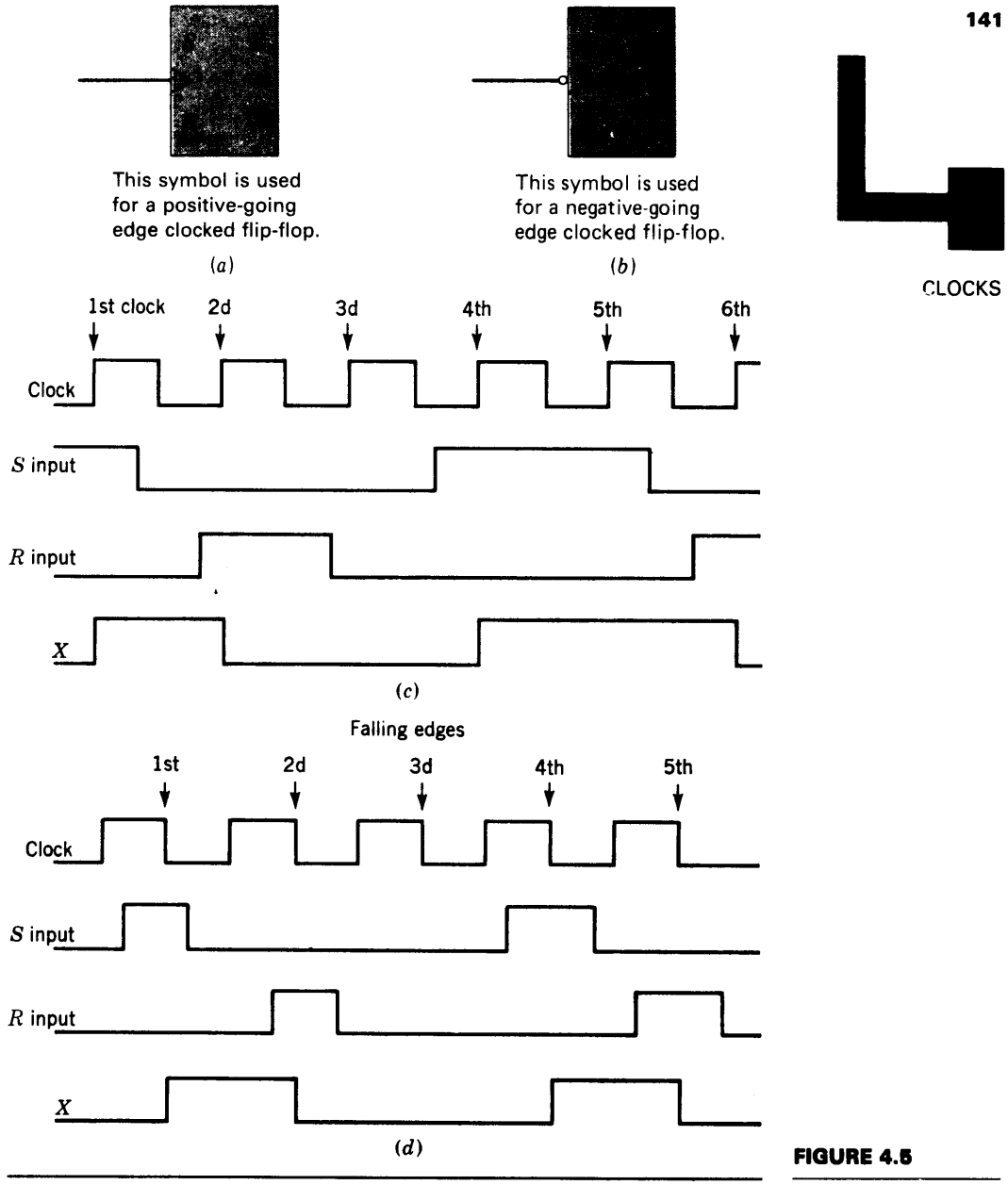


FIGURE 4.5

Clocked flip-flops and waveforms. (*a*) Positive-edge-triggering flip-flop. (*b*) Negative-edge-triggering flip-flop. (*c*) Waveforms for positive-edge-triggering flip-flop in (*a*). (*d*) Waveforms for flip-flop in (*b*).

3 If the S input is a 0 and the R input a 1 when the clock pulse occurs, the flip-flop is cleared to the 0 state.

4 Both the S and R inputs should not be 1s when the clock signal's positive-going edge occurs.

Of course, nothing happens to the flip-flop's state between occurrences of the initiating positive-going clock signal. Figure 4.5(*c*) shows this with a square-wave clock signal. The flip-flop is set to a 1 by the first clock positive-going edge and a 0 at the occurrence of the second clock signal. No change occurs at the third positive-going clock edge. The flip-flop is set to 1 again on the fourth edge and remains a 1 until the sixth clock edge occurs. Notice that the *S* and *R* inputs can be anything between the clock edges without affecting the operation of the flip-flop. (They can even both be 1s without effect, except when the positive-going edge occurs.)

Figure 4.5(*d*) shows typical waveforms for the flip-flop in Fig. 4.5(*b*). This flip-flop is *negative-edge triggering* because it responds to shifts in the clock level which are negative-going. The rules of operation are as before: 0s on *S* and *R* lead to no change; a 1 on *S* sets the flip-flop; and a 1 on *R* clears the flip-flop. The flip-flop responds to the *S* and *R* inputs only at the precise time the clock input goes negative.

FLIP-FLOP DESIGNS

4.4 Flip-flops can be made from gates; in fact, this is a common practice. Figure 4.6 shows two NOR gates cross-coupled to form an *RS* flip-flop. The cross-coupled NOR gates in Fig. 4.6(*a*) have two inputs, *S* and *R*, and two outputs, *Q*, and $\overline{Q}$. This configuration realizes the *RS* flip-flop in Fig. 4.6(*b*).

The operation of the NOR gates is as follows: Consider both *S* and *R* to be 0s. If *Q* is a 1, then the rightmost NOR gate has a 1 and a 0 input and its output will be a 0. This places a 0 on the $\overline{Q}$ output and two 0s at the input to the leftmost NOR gate which will have a 1 output, and the configuration will be stable. Similar reasoning will show that the configuration will be stable with a 1 on $\overline{Q}$ and a 0 on *Q*.

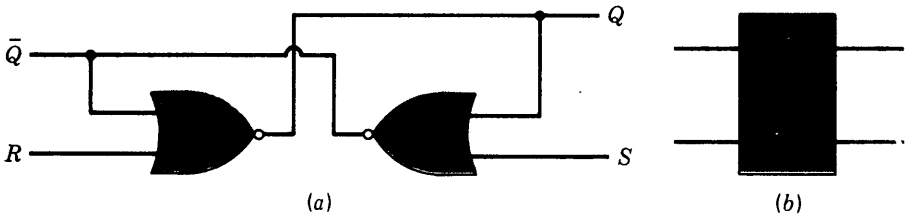
The *S* and *R* inputs work as follows: If a 1 is placed on the *R* input and a 0 on the *S* input, this will force the leftmost NOR gate to a 0 output, and this will cause the rightmost NOR gate to have two 0s as inputs and a 1 output. The flip-flop has now been cleared with a 0 on the *Q* output and a 1 on the $\overline{Q}$ output. Similar reasoning will show that a 1 on the *S* input and a 0 on the *R* input will force the NOR gate flip-flop to the 1 state with *Q* a 1 and $\overline{Q}$ a 0.

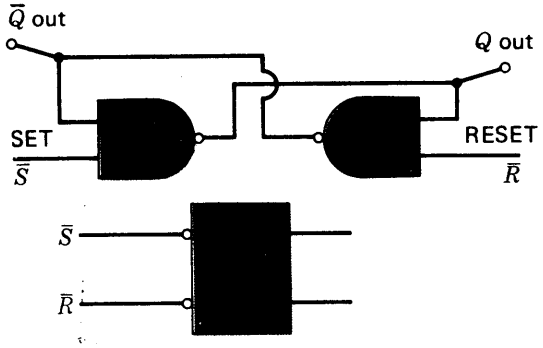
GATED FLIP-FLOP

4.5 Just as Fig. 4.6 showed that two NOR gates can be used to form an *RS* flip-flop, Fig. 4.7 shows that two NAND gates can be used to form an *RS* flip-flop

FIGURE 4.6

RS flip-flop formed by cross-coupling NOR gates. (*a*) Cross-coupled NOR gates. (*b*) *RS* flip-flop corresponding to (*a*).





$\bar{S}$	$\bar{R}$	NEXT STATE
0	0	NOT USED
0	1	1
1	0	0
1	1	NO CHANGE

FIGURE 4.7

Two NAND gates used to form an *RS* flip-flop.

(however, the inputs are complemented). In this case the inputs operate as follows: When both $\bar{S}$ and $\bar{R}$ are 1s, the flip-flop will remain in its present state, that is, it will not change states. If, however, the $\bar{R}$ input goes to a 0, the NAND gate connected to $\bar{R}$ will have a 1 output regardless of the other feedback input to the NAND gate. This will force the flip-flop to the 0 state (provided the $\bar{S}$ input is kept high or a 1).

Similar reasoning shows that making the $\bar{S}$ input a 0 will cause the NAND gate at the $\bar{S}$ input to have a 1 output, forcing the flip-flop to the 1 state (again provided the $\bar{R}$ input is kept high or 1).

If both inputs $\bar{R}$ and $\bar{S}$ are made 0s, the next state will depend on which input is returned to 1 first. If both are returned to 1 simultaneously, the resulting state of the flip-flop will be indeterminate. As a result, this is a "forbidden," or "restricted," input combination.

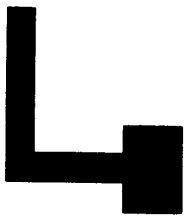
The block diagram in Fig. 4.7 shows the flip-flop to be a conventional *RS* flip-flop, except that the two inputs are inverted. This is shown by the two bubbles at the $\bar{R}$ and $\bar{S}$ inputs. The circuit is activated by 0s and inputs are normally at 1.

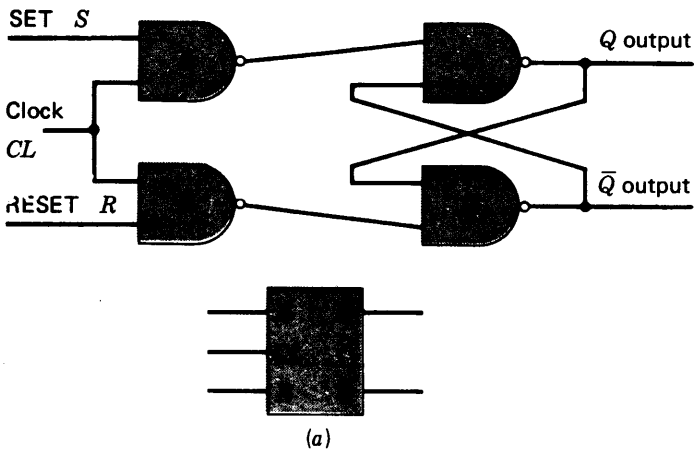
A limited form of clocked flip-flop called a *latch* can be formed by using four NAND gates, as shown in Fig. 4.8(*a*). The circuit has an *R* and an *S* input and also a clock input CL. This latch flip-flop is activated by a positive level on the clock input, and not by a positive transition. Thus the flip-flop "takes" its input levels during the positive portion of clock signals, not changes in clock levels. Let us see how the circuit works. If the clock signal is at the 0 level, both NAND gates *A* and *B* will have 1 outputs, and so the NAND gate inputs to *C* and *D* will be a 1 and, as before, the flip-flop will remain in its present state with either *C* or *D* on. (Both cannot be on because of the cross-coupling.)

If the clock signal goes to the 1 level and both inputs *R* and *S* are at the 0 level, the NAND gate outputs of *A* and *B* will still be 1s, and the flip-flop will remain in the same state.

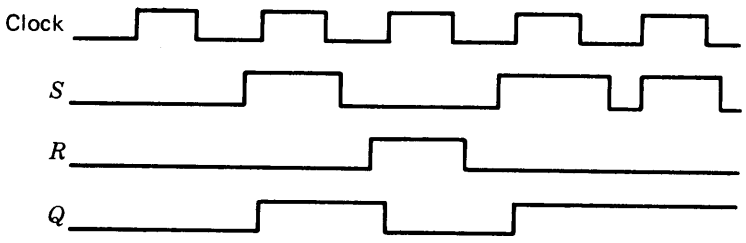
If the *R* input is a 1 and the *S* input a 0, when the clock input goes positive (a 1), the NAND gate connected to *R* will have a 0 output and the NAND gate connected to *S* a 1 output, forcing the flip-flop consisting of *C* and *D* to the 0 state.

Similarly, a 1 at the *S* input and a 0 at the *R* input will cause the *S* input gate *A* to have a 0 output and the *R* input gate *B* a 1 output, forcing the flip-flop to the 1 state.

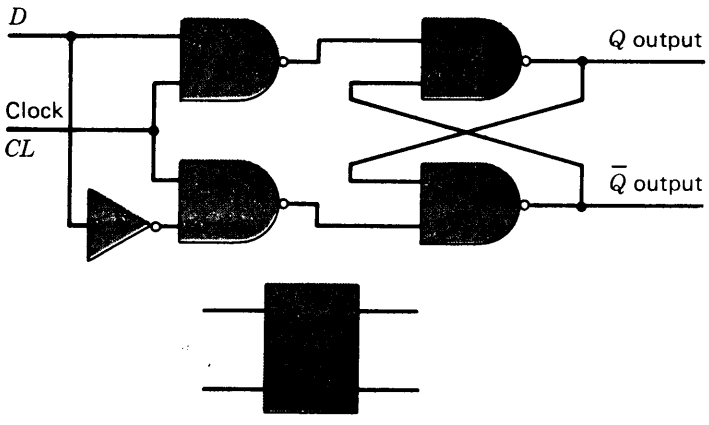




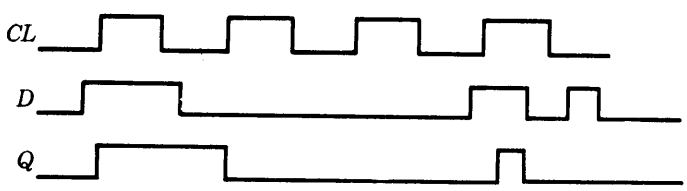
(a)



(b)



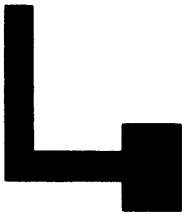
(c)



(d)

FIGURE 4.8

Latches. (*a*) *RS* latch. (*b*) *RS* flip-flop waveforms. (*c*) *D* latch. (*d*) Waveforms for *D* latch.



If both S and R are 1s and a clock pulse (0 level to 1 level and back to 0) occurs, the next state of the circuit is indeterminate.

A major problem with this circuit is that the R and S inputs should remain unchanged during the time the clock is a 1. This considerably limits the use of the circuit, leading to more complicated circuits that can offer the designer more flexibility. The primary value of the circuit is its simplicity.

The block diagram for a latch is the same as for an edge-triggered flip-flop except for the small triangle at the clock input for edge-triggered flip-flops and a CL for the latch.

Often designers specifically identify latches on block diagrams so that users will realize that the state taken by the flip-flop is determined by the R and S inputs during the positive clock level, and not at the edge of the clock signal.

Manufacturers often put a number of latches in a single IC package. In this case, a special kind of flip-flop called the D flip-flop is often used. A D-type latch flip-flop is shown in Fig. 4.8(c). The advantage of this is the single D input. The D flip-flop takes the value at its D input whenever the clock pulse input is high. It will effectively "track" input levels as long as the clock input is high, as shown in Fig. 4.8(d). If the clock input is lowered, the state will be the last state the flip-flop had when the clock input was high. If the clock input returns to 0 just before or during a transition in the input to the D latch, the final state the flip-flop takes will depend on the delay time for the flip-flop. Allowing input transitions during the time period when the clock is high is dangerous, unless there is assurance that the D input will not change for a safe period before the clock input is lowered.

MASTER-SLAVE FLIP-FLOP

4.6 To eliminate problems which arise with the latch type of flip-flop, more complicated flip-flop designs are used. The most popular uses edge triggering from the clock to initiate changes in the flip-flop's output and is based on the use of two single or latch flip-flops to form a single edge-triggered flip-flop.

The basic flip-flop design is shown in Fig. 4.9. Figure 4.9(a) shows that an edge-triggering RS flip-flop consists of two flip-flops plus some gating. The two flip-flops are called the master and slave. An expanded logic diagram for Fig. 4.9(a) is shown in Fig. 4.9(b) in which the master flip-flop is composed of the leftmost NAND gates and the slave flip-flop of the rightmost NAND gates.

The expanded diagram in Fig. 4.9(b) can be used to explain the flip-flop's operation. The flip-flop's output changes on the negative-going edge of the clock pulse. The basic timing is shown in Fig. 4.9(b). First, on the positive-going edge and during the positive section of the clock pulse, the master flip-flop is loaded by the two leftmost NAND gates. Then, during the negative-going edge of the clock signal, the two rightmost NAND gates load the contents of the master flip-flop into the slave flip-flop just after the two-input NAND gates are disabled. This means that the master flip-flop will not change in value while the clock is low (0); so the slave remains attached to a stable flip-flop, with a value taken during the positive section of the clock pulse.

A more detailed account of the action of the flip-flop is as follows: If the clock signal is low, the two-input NAND gates both have 1 outputs; so the master

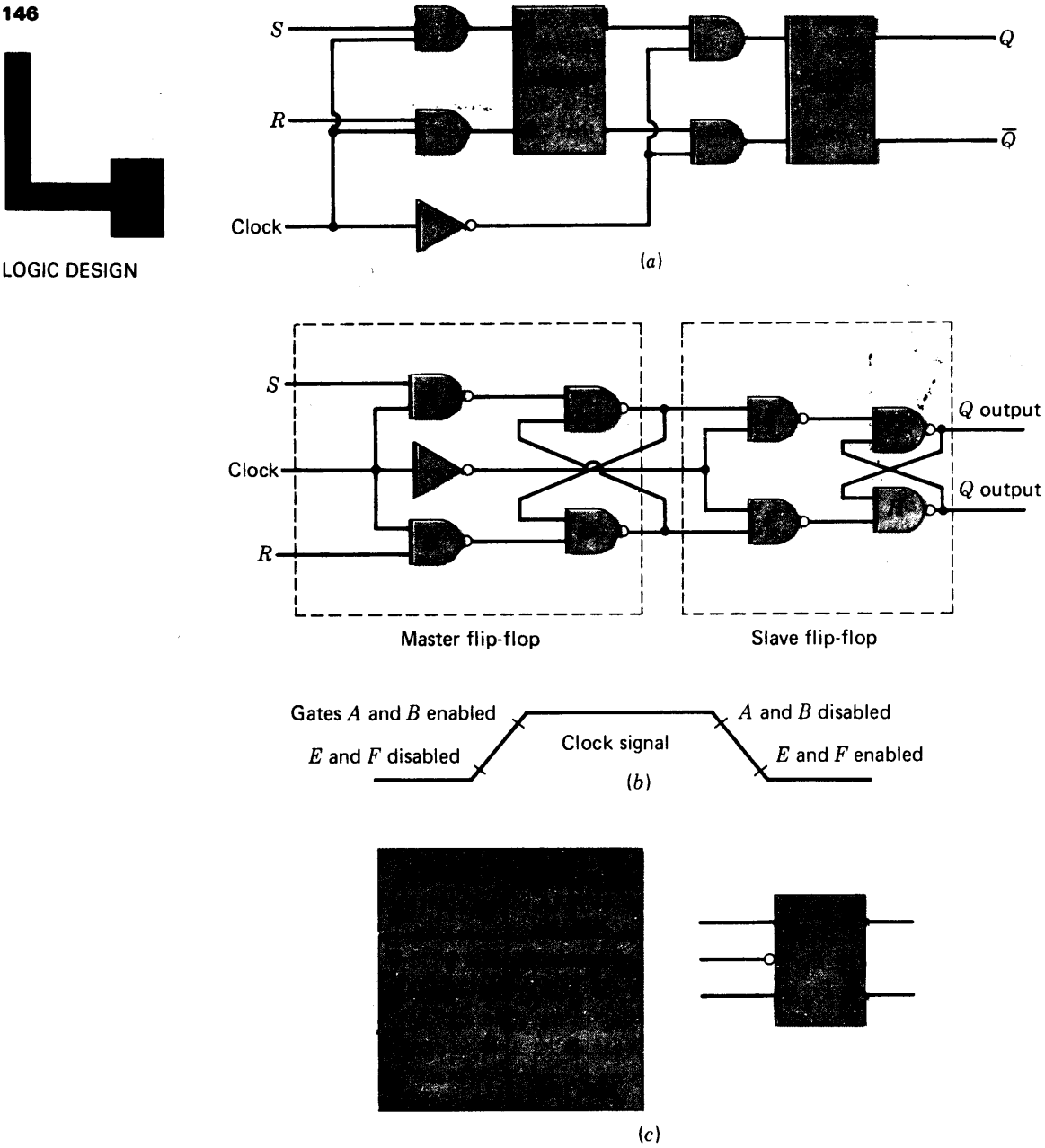


FIGURE 4.9

(*a*) *RS* clocked edge-triggering flip-flop. (*b*) Gate arrangement for master-slave flip-flop. (*c*) Master-slave flip-flop state table and symbol.

flip-flop does not change states since it is a NAND gate flip-flop and can be set or cleared only by 0 inputs.

At the same time, as long as the clock signal is low (a 0), the inverter causes the inputs to the *E* and *F* NAND gates to force the value of the master flip-flop

into the slave flip-flop. The situation is stable. The master cannot change, and the output flip-flop is "slaved" to the master.

When the clock starts positive, however, the E and F NAND gates are disabled, and the NAND gates A and B to the master flip-flop are then enabled.

When the input clock signal is a 1, then the master flip-flop will accept information from the S and R inputs, and the slave flip-flop is now isolated from the master and will not change states regardless of changes in the master.

The operation of the master flip-flop is according to the following rules when the clock level is a 1:

1 If both S and R are at 0 levels, the two input NAND gates will have 1 outputs and the master flip-flop will not change values.

2 If the S input is a 1 and the R input a 0, the master flip-flop will go to its 1 state, with the upper master NAND gate having a 1 output.

3 If the S input is a 0 and the R input a 1, the master flip-flop will go to its 0 state, with the upper NAND gate having a 0 output.

4 If both R and S are 1s, the final state is indeterminate.

When the clock signal goes negative to its 0 level, first the input NAND gates to the master flip-flop are disabled, that is, each output goes to a 1; then the E and F NAND gates are enabled (by the inverted clock signal). This causes the state of the master flip-flop to be transferred into the slave flip-flop.

The effects of all this are shown in the next-state table in Fig. 4.9(c), which indicates that the flip-flop is an *RS* flip-flop activated by a negative-going clock signal.

An edge-triggering flip-flop which triggers on positive edges can be made by adding an inverter at the CL input.

SHIFT REGISTER

4.7 Figure 4.10 shows a *shift register*. This circuit accepts information from some input source and then shifts this information along the chain of flip-flops, moving it one flip-flop each time a positive-going clock signal occurs.

Figure 4.10 also shows a typical sequence of input signals and flip-flop signals in the shift register. The input value is taken by X_1 when the first positive-going clock signal arrives.[3] Anything in this and the remaining flip-flops is shifted right at this time. We have assumed that all the flip-flops are initially in their 0 states. In the figure, the input waveform is at 1 when the first clock occurs; so X_1 goes to the 1 state.

---

[3]There is some delay from the time the positive-going edge of the clock signal tells a flip-flop to "go" until the flip-flop's outputs are able to change values. The clock signal itself will also require some small amount of time to rise, for physical reasons. For present systems the rise time on the clock signals, that is, the time to rise 90 percent of total rise, ranges from about 1×10^{-9} to about 50×10^{-9} s. The delay from the clock-signal change until a flip-flop's output changes 90 percent, which is called the *delay time*, ranges from 10.5×10^{-9} to 50×10^{-9} s for most circuits.

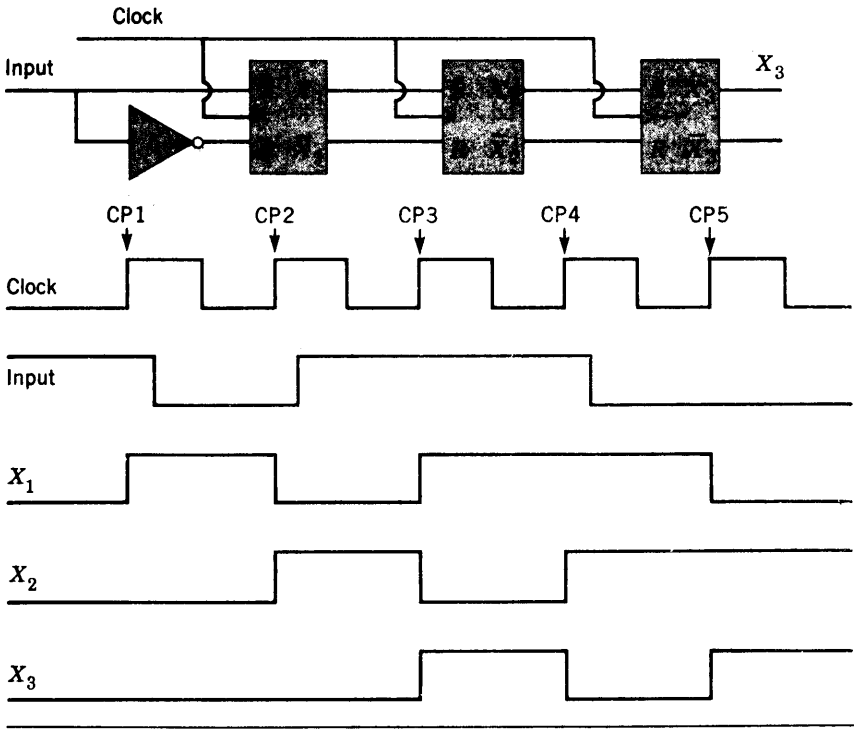


FIGURE 4.10

Shift register with waveforms.

When the second positive-going clock signal arrives, the input is at 0; so X_1 goes to the 0 state, but the 1 in X_1 is shifted into X_2. When the third clock edge appears, the input is a 1; so X_1 takes a 1, the 0 previously in X_1 is shifted into X_2, and the 1 in X_2 goes into X_3. This process continues. The values in X_3 are simply dropped off the end of the register.

Notice that each flip-flop takes the value in the flip-flop on its left when the shift register is stepped. The reasoning is as follows: If, for instance, X_1 is in the 1 state, then its X output line is a 1 and thus the S input to X_2 will be a 1; and the $\overline{X}$ output of X_1 will be a 0, and so the R input of X_2 will be a 0. This causes X_2 to take its 1 state when the clock pulse occurs. A 0 in X_1 will cause X_2 to go to 0 when the clock pulse occurs, and the reason for this should be analyzed.

There is one problem that could occur if certain design precautions were not taken with the flip-flops. If the flip-flop outputs changed too fast or if latches were used, a state could ripple, or race, down the chain. This is called the *race problem*. It is handled by designing the flip-flops so that they take the value at their inputs just as the clock positive-going edge occurs and not slightly after the clock's rise time. This leads to the complexity in flip-flop design, which has been discussed. Edge-triggered (master-slave) flip-flops are necessary for proper operation.

BINARY COUNTER

4.8 Inasmuch as the binary counter is one of the most useful of logical circuits, there are many kinds. The fundamental purpose of the binary counter is to record

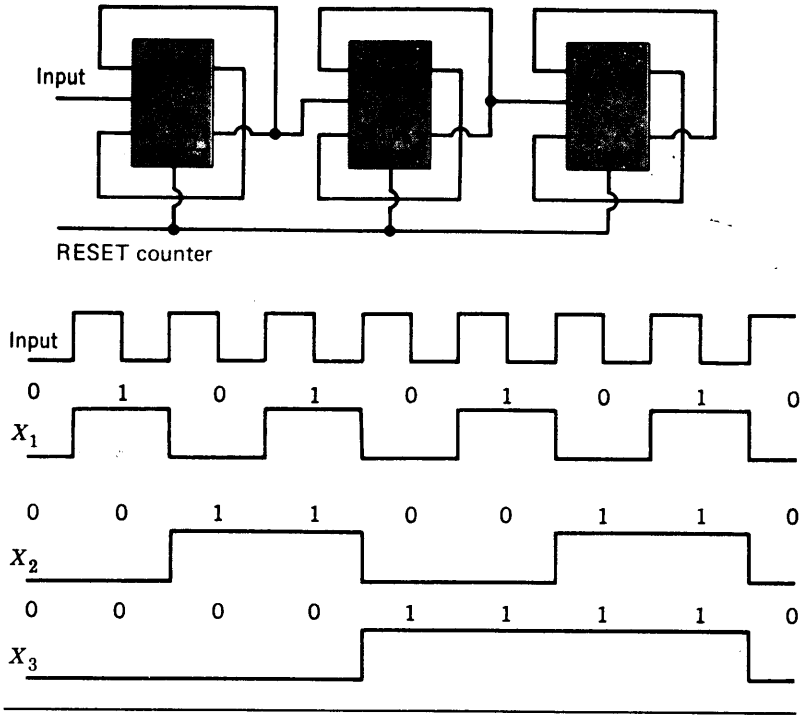


FIGURE 4.11

Binary counter.

the number of occurrences of some input. This is a basic function, that of counting, and it is used over and over.

The first type of binary counter to be explained is shown in Fig. 4.11. This counter records the number of occurrences of a positive-going edge (or pulse) at the input.

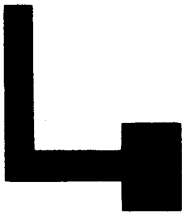
It is desirable to start this counter with 0s in all three flip-flops; so one further line is added to each flip-flop, a DC RESET line. This line is normally at the 0 level; when it goes positive, or 1, it places a 0 in the flip-flop. This action does not depend on the clock. When a DC RESET line is at the 1 level, the flip-flop goes to 0 regardless of any other input and in the absence or presence of a clock pulse.

It is quite common for flip-flops to have a DC RESET line. Notice that this input "overrides" all other inputs when it is a 1, forcing the flip-flop to the 0 state. A 0 on this line, however, does not affect flip-flop operation in any way.

Before counting begins, then, a 1 is placed temporarily on the RESET COUNTER line, and the three flip-flops are cleared to 0. The RESET COUNTER line is then returned to 0.

When the first clock positive edge occurs, the flip-flop X_1 goes to its 1 state. This is because when the flip-flop X_1 is in the 0 state, the $\overline{X}_1$ output is high, or 1, placing a 1 on the S input (refer to Fig. 4.11), and the X_1 output is low, or 0, placing a 0 on the R input; so a 1 goes into flip-flop X_1.

Flip-flops X_2 and X_3 are not affected by this change, for although the $\overline{X}_1$ output is connected to the clock input of X_2, the signal has gone from 1 to 0. This is a negative shift, which does nothing to X_2.



The counter now has $X_3 = 0$, $X_2 = 0$, $X_1 = 1$, or binary 001; so the first input clock edge has stepped the counter from 000 to 001.†

The occurrence of the second positive-going clock edge causes flip-flop X_1 to go from the 1 state to the 0 state. The reasoning is as follows: When X_1 is a 1, the X_1 output is a 1 and is connected to the R input, and the $\overline{X}_1$ output is a 0 and is connected to the S input. This tells the flip-flop to "go to 0," and when the second clock pulse occurs, it goes.

This is important: When a flip-flop is cross-coupled, that is, when its uncomplemented output is connected to its R input and its complemented output to its S input, the occurrence of a clock edge will always cause it to *complement*, or change values.

The change of value from 1 to 0 of flip-flop X_1 causes X_2 to change from a 0 to a 1. This is because $\overline{X}_1$'s output is connected to the CL input of X_2 and has gone from 0 to 1, a positive shift; and since X_2 is cross-coupled, it will complement (change values) and go from 0 to 1. This does not affect X_3, because the CL input of X_3 has gone from 1 to 0, a negative shift.

The counter has now progressed to $X_1 = 0$, $X_2 = 1$, and $X_3 = 0$; so the sequence of states has been 000, 001, 010.

Reasoning of this type will show that the progression of states by the counter will be as follows:

X_3	X_2	X_1
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
0	0	0
0	0	1
0	1	0

This is a list of binary numbers from 0 to 7, which repeats over and over. After five input pulses the counter contains 101, or binary 5; after seven pulses the counter contains 111, or binary 7. The maximum number of pulses this counter can handle, without ambiguity, is 7. After eight pulses the counter contains 0; after nine pulses, 1; etc. In the trade this is called a *modulo 8*, or *three-stage*, counter.

The counter can be extended by another flip-flop, X_4, which is cross-coupled and which has its CL input connected to the output of flip-flop $\overline{X}_3$. This forms a *four-stage*, or *modulo 16*, counter, which can handle up to 15 counts. A fifth flip-flop would form a counter which would count to 31, a sixth to 63, etc.

We now consider a *gated-clocked binary counter*. This is an exceedingly popular counter in modern computers, and it demonstrates the fact that most oper-

†Note that the binary numbers are written in the opposite direction from the block diagram layout, which has the least significant bit on the left. This makes for a neater block diagram and is frequently used. The standards, in fact, ask for left-to-right signal flow.