

## APPENDIX A: DIGITAL LOGIC

### A.1 Introduction

In this appendix, we take a look at a few basic principles of digital logic that we can apply in the design of a digital computer. We start by studying **combinational logic** in which logical decisions are made based only on combinations of the inputs. We then look at **sequential logic** in which decisions are made based on combinations of the current inputs as well as the past history of inputs. With an understanding of these underlying principles, we can design digital logic circuits from which an entire computer can be constructed. We begin with the fundamental building block of a digital computer, the **combinational logic unit (CLU)**.

### A.2 Combinational Logic

A combinational logic unit translates a set of inputs into a set of outputs according to one or more mapping functions. The outputs of a CLU are strictly functions of the inputs, and the outputs are updated immediately after the inputs change. A basic model of a CLU is shown in Figure A-1. A set of inputs  $i_0 - i_n$  is

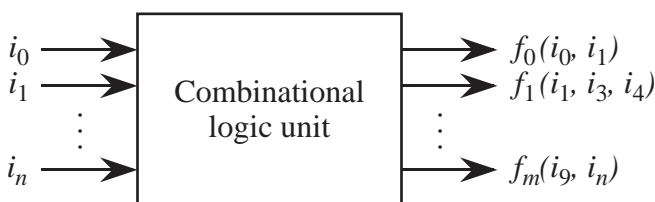


Figure A-1 External view of a combinational logic unit.

presented to the CLU, which produces a set of outputs according to mapping functions  $f_0 - f_m$ . There is no feedback from the outputs back to the inputs in a combinational logic circuit (we will study circuits with feedback in Section

A.11.)

Inputs and outputs for a CLU normally have two distinct values: high and low. When signals (values) are taken from a finite set, the circuits that use them are referred to as being **digital**. A digital electronic circuit receives inputs and produces outputs in which 0 volts (0 V) is typically considered to be a low value and +5 V is considered to be a high value. This convention is not used everywhere: high speed circuits tend to use lower voltages; some computer circuits work in the **analog** domain, in which a continuum of values is allowed; and digital optical circuits might use phase or polarization in which high or low values are no longer meaningful. An application in which analog circuitry is appropriate is in flight simulation, since the analog circuits more closely approximate the mechanics of an aircraft than do digital circuits.

Although the vast majority of digital computers are binary, **multi-valued** circuits also exist. A wire that is capable of carrying more than two values can be more efficient at transmitting information than a wire that carries only two values. A digital multi-valued circuit is different from an analog circuit in that a multi-valued circuit deals with signals that take on one of a finite number of values, whereas an analog signal can take on a continuum of values. The use of multi-valued circuits is theoretically valuable, but in practice it is difficult to create reliable circuitry that distinguishes between more than two values. For this reason, multi-valued logic is currently in limited use.

In this text, we are primarily concerned with digital binary circuits, in which exactly two values are allowed for any input or output. Thus, we will consider only binary signals.

### A.3 Truth Tables

In 1854 George Boole published his seminal work on an algebra for representing logic. Boole was interested in capturing the mathematics of thought, and developed a representation for factual information such as “The door is open.” or “The door is not open.” Boole’s algebra was further developed by Shannon into the form we use today. In Boolean algebra, we assume the existence of a basic postulate, that a binary variable takes on a single value of 0 or 1. This value corresponds to the 0 and +5 voltages mentioned in the previous section. The assignment can also be done in reverse order for 1 and 0, respectively. For purposes of understanding the behavior of digital circuits, we can abstract away the physical correspondence to voltages and consider only the symbolic values 0 and 1.

A key contribution of Boole is the development of the **truth table**, which captures logical relationships in a tabular form. Consider a room with two 3-way switches *A* and *B* that control a light *Z*. Either switch can be up or down, or both switches can be up or down. When exactly one switch is up, the light is on. When both switches are up or down, the light is off. A truth table can be constructed that enumerates all possible settings of the switches as shown in Figure

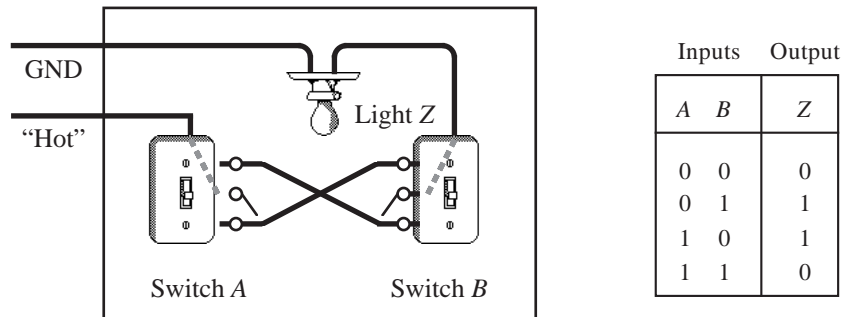


Figure A-2 A truth table relates the states of 3-way switches *A* and *B* to light *Z*.

A-2. In the table, a switch is assigned the value 0 if it is down, otherwise it is assigned the value 1. The light is on when  $Z = 1$ .

In a truth table, all possible input combinations of binary variables are enumerated and a corresponding output value of 0 or 1 is assigned for each input combination. For the truth table shown in Figure A-2, the output function *Z* depends upon input variables *A* and *B*. For each combination of input variables there are two values that can be assigned to *Z*: 0 or 1. We can choose a different assignment for Figure A-2, in which the light is on only when both switches are up or both switches are down, in which case the truth table shown in Figure A-3 enu-

Inputs		Output
<i>A</i>	<i>B</i>	<i>Z</i>
0	0	1
0	1	0
1	0	0
1	1	1

Figure A-3 Alternate assignments of outputs to switch settings.

merates all possible states of the light for each switch setting. The wiring pattern would also need to be changed to correspond. For two input variables, there are  $2^2 = 4$  input combinations, and  $2^4 = 16$  possible assignments of outputs to input

combinations. In general, since there are  $2^n$  input combinations for  $n$  inputs, there are  $2^{2^n}$  possible assignments of output values to input combinations.

#### A.4 Logic Gates

If we enumerate all possible assignments of switch settings for two input variables, then we will obtain the 16 assignments shown in Figure A-4. We refer to

Inputs		Outputs							
$A$	$B$	<i>False</i>	<i>AND</i>	$\overline{AB}$	$A$	$\overline{A}$	$B$	<i>XOR</i>	<i>OR</i>
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Inputs		Outputs							
$A$	$B$	<i>NOR</i>	<i>XNOR</i>	$\overline{B}$	$A + \overline{B}$	$\overline{A}$	$\overline{A} + B$	<i>NAND</i>	<i>True</i>
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

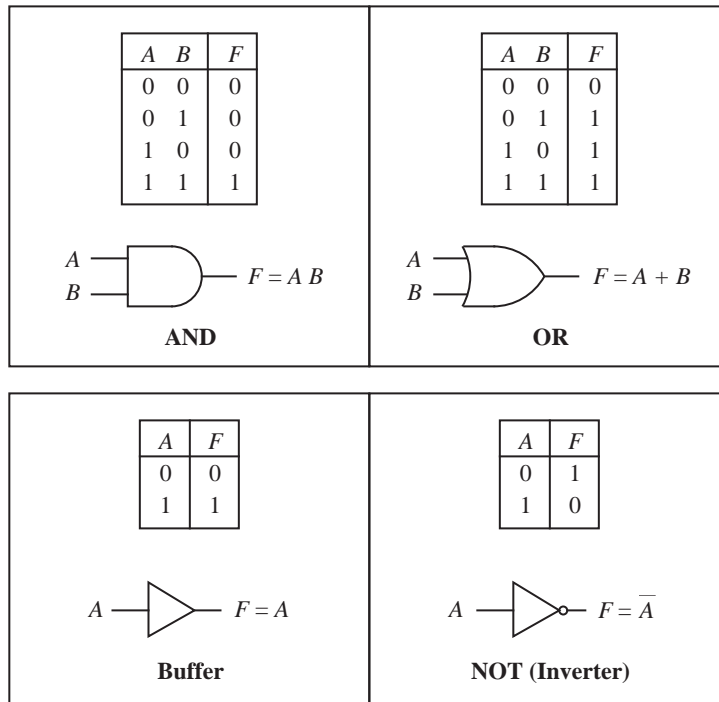
Figure A-4 Truth tables showing all possible functions of two binary variables.

these functions as **Boolean logic functions**. A number of assignments have special names. The *AND* function is true (produces a 1) only when  $A$  and  $B$  are 1, whereas the *OR* function is true when either  $A$  or  $B$  is 1, or when both  $A$  and  $B$  are 1. A function is false when its output is 0, and so the *False* function is always 0, whereas the *True* function is always 1. The plus signs '+' in the Boolean expressions denote logical *OR*, and do not imply arithmetic addition. The juxtaposition of two variables, as in  $AB$ , denotes logical *AND* among the variables.

The  $A$  and  $B$  functions simply repeat the  $A$  and  $B$  inputs, respectively, whereas the  $\overline{A}$  and  $\overline{B}$  functions **complement**  $A$  and  $B$ , by producing a 0 where the uncomplemented function is a 1 and by producing a 1 where the uncomplemented function is a 0. In general, a bar over a term denotes the complement operation, and so the *NAND* and *NOR* functions are complements to *AND* and *OR*, respectively. The *XOR* function is true when either of its inputs, but not

both, is true. The *XNOR* function is the complement to *XOR*. The remaining functions are interpreted similarly.

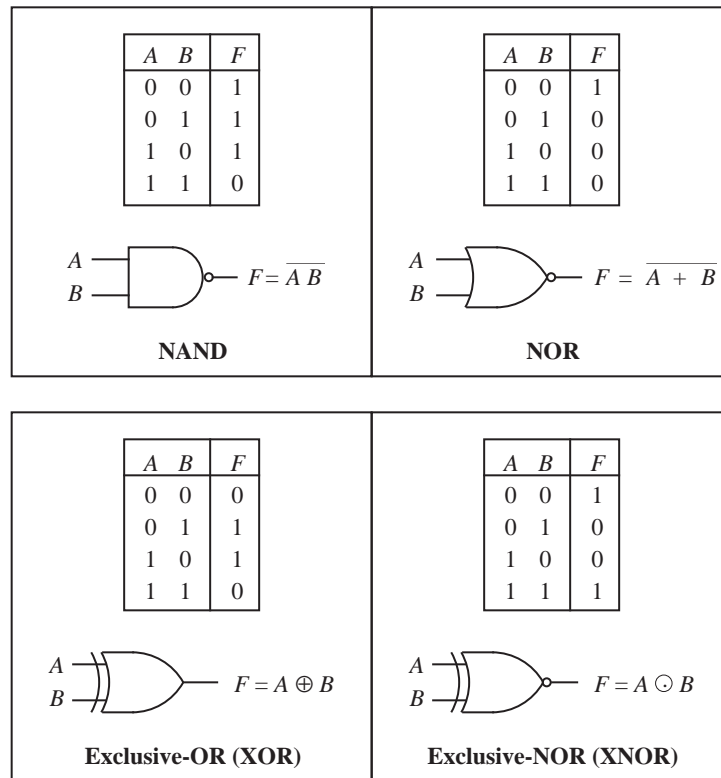
A **logic gate** is a physical device that implements a simple Boolean function. The functions that are listed in Figure A-4 have representations as logic gate symbols, a few of which are shown in Figure A-5 and Figure A-6. For each of the func-



**Figure A-5** Logic gate symbols for AND, OR, buffer, and NOT Boolean functions.

tions, *A* and *B* are binary inputs and *F* is the output.

In Figure A-5, the AND and OR gates behave as previously described. The output of the AND gate is true when both of its inputs are true, and is false otherwise. The output of the OR gate is true when either or both of its inputs are true, and is false otherwise. The buffer simply copies its input to its output. Although the buffer has no logical significance, it serves an important practical role as an amplifier, allowing a number of logic gates to be driven by a single signal. The NOT gate (also called an **inverter**) produces a 1 at its output for a 0 at its input, and produces a 0 at its output for a 1 at its input. Again, the inverted output signal is referred to as the complement of the input. The circle at the output of the

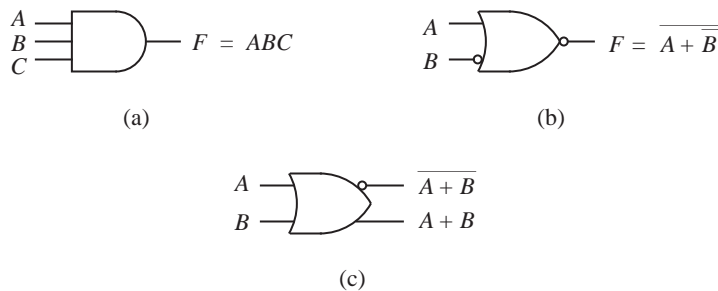


**Figure A-6** Logic gate symbols for NAND, NOR, XOR, and XNOR Boolean functions.

NOT gate denotes the complement operation.

In Figure A-6, the NAND and NOR gates produce complementary outputs to the AND and OR gates, respectively. The exclusive-OR (XOR) gate produces a 1 when either of its inputs, but not both, is 1. In general, XOR produces a 1 at its output whenever the number of 1's at its inputs is odd. This generalization is important in understanding how an XOR gate with more than two inputs behaves. The exclusive-NOR (XNOR) gate produces a complementary output to the XOR gate.

The logic symbols shown in Figure A-5 and Figure A-6 are only the basic forms, and there are a number of variations that are often used. For example, there can be more inputs, as for the three-input AND gate shown in Figure A-7a. The circles at the outputs of the NOT, NOR, and XNOR gates denote the complement operation, and can be placed at the inputs of logic gates to indicate that



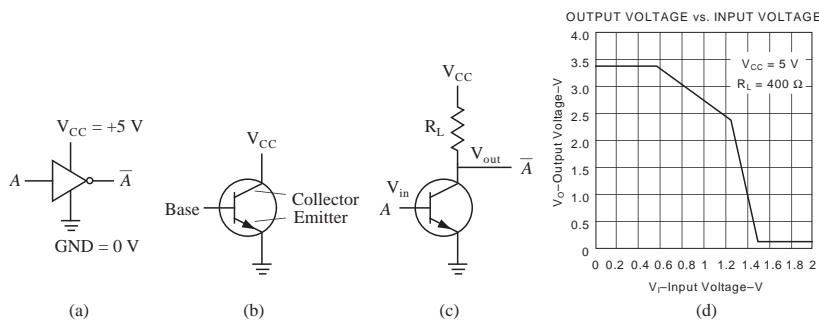
**Figure A-7** Variations of the basic logic gate symbols for (a) three inputs; (b) a negated output; and (c) complementary outputs.

the inputs are inverted upon entering the gate, as shown in Figure A-7b. Depending on the technology used, some logic gates produce complementary outputs. The corresponding logic symbol for a complementary logic gate indicates both outputs as illustrated in Figure A-7c.

Physically, logic gates are not magical, although it may seem that they are when a device like an inverter can produce a logical 1 (+5 V) at its output when a logical 0 (0 V) is provided at the input. The next section covers the underlying mechanism that makes electronic logic gates work.

#### A.4.1 ELECTRONIC IMPLEMENTATION OF LOGIC GATES

Electrically, logic gates have power terminals that are not normally shown. Figure



**Figure A-8** (a) Power terminals for an inverter made visible; (b) schematic symbol for a transistor; (c) transistor circuit for an inverter; (d) static transfer function for an inverter.

A-8a illustrates an inverter in which the +5 V and 0 V (GND) terminals are made visible. The +5 V signal is commonly referred to as  $V_{CC}$  for “voltage collector-collector.” In a physical circuit, all of the  $V_{CC}$  and GND terminals are con-

nected to the corresponding terminals of a power supply.

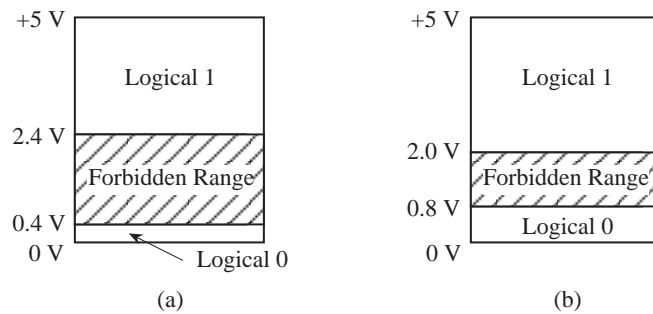
Logic gates are composed of electrical devices called **transistors**, which have a fundamental switching property that allows them to control a strong electrical signal with a weak signal. This supports the process of amplification, which is crucial for cascading logic gates. Without amplification, we would only be able to send a signal through a few logic gates before the signal deteriorates to the point that it is overcome by noise, which exists at every point in an electrical circuit to some degree.

The schematic symbol for a transistor is shown in Figure A-8b. When there is no positive voltage on the base, then a current will not flow from  $V_{CC}$  to GND. Thus, for an inverter, a logical 0 (0 V) on the base will produce a logical 1 (+5 V) at the collector terminal as illustrated in Figure A-8c. If, however, a positive voltage is applied to  $V_{in}$ , then a current will flow from  $V_{CC}$  to GND, which prevents  $V_{out}$  from producing enough signal for the inverter output to be a logical 1. In effect, when +5 V is applied to  $V_{in}$ , a logical 0 appears at  $V_{out}$ . The input-output relationship of a logic gate follows a nonlinear curve as shown in Figure A-8d for transistor-transistor logic (TTL). The nonlinearity is an important gain property that makes cascable operation possible.

A useful paradigm is to think of current flowing through wires as water flowing through pipes. If we open a connection on a pipe from  $V_{CC}$  to GND, then the water flowing to  $V_{out}$  will be reduced to a great extent, although some water will still make it out. By choosing an appropriate value for the resistor  $R_L$ , the flow can be restricted in order to minimize this effect.

Since there will always be some current that flows even when we have a logical 0 at  $V_{out}$ , we need to assign logical 0 and 1 to voltages using safe margins. If we assign logical 0 to 0 V and logical 1 to +5 V, then our circuits may not work properly if .1 V appears at the output of an inverter instead of 0 V, which can happen in practice. For this reason, we design circuits in which assignments of logical 0 and 1 are made using **thresholds**. In Figure A-9a, logical 0 is assigned to the voltage range [0 V to 0.4 V] and logical 1 is assigned to the voltage range [2.4 V to +5 V]. The ranges shown in Figure A-9a are for the output of a logic gate. There may be some attenuation (a reduction in voltage) introduced in the connection between the output of one logic gate and the input to another, and for that reason, the thresholds are relaxed by 0.4 V at the input to a logic gate as shown in Figure A-9b. These ranges can differ depending on the logic family. The output ranges only make sense, however, if the gate inputs settle into the

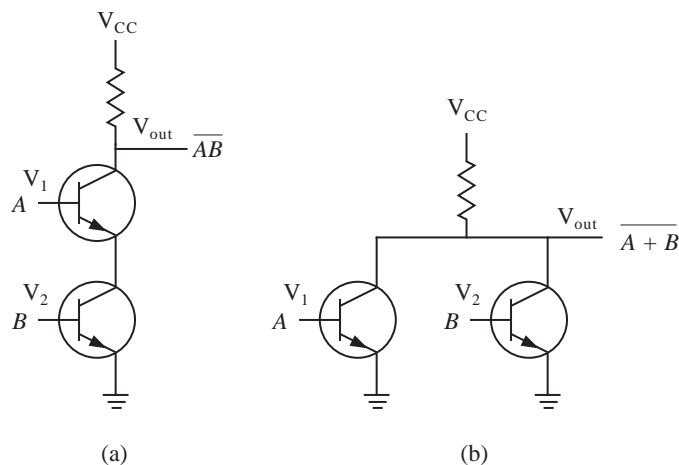




**Figure A-9** Assignments of logical 0 and 1 to voltage ranges (a) at the output of a logic gates, and (b) at the input to a logic gate.

logical 0 or 1 ranges at the input. For this reason, inputs to a logic gate should never be left “floating” – disconnected from a gate output,  $V_{CC}$ , or GND.

Figure A-10 shows transistor circuits for two-input NAND and NOR gates. For



**Figure A-10** Transistor circuits for (a) a two-input NAND gate and (b) a two-input NOR gate.

the NAND case, both of the  $V_1$  and  $V_2$  inputs must be in the logical 1 region in order to produce a voltage in the logical 0 region at  $V_{out}$ . For the NOR case, if either or both of the  $V_1$  and  $V_2$  inputs are in the logical 1 region, then a voltage in the logical 0 region will be produced at  $V_{out}$ .

A.4.2 TRI-STATE BUFFERS

A **tri-state buffer** behaves in a similar manner to the ordinary buffer that was introduced earlier in this appendix, except that a control input is available to disable the buffer. Depending on the value of the control input, the output is either 0, 1, or *disabled*, thus providing three output states. In Figure A-11, when the

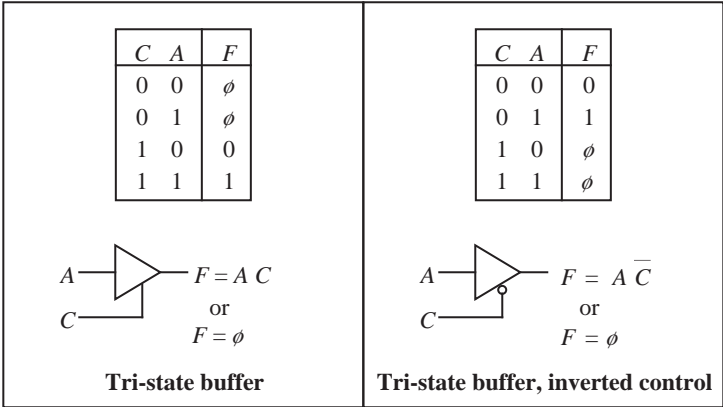


Figure A-11 Tri-state buffers.

control input  $C$  is 1, the tri-state buffer behaves like an ordinary buffer. When  $C$  is 0, then the output is electrically disconnected and no output is produced. The  $\phi$ 's in the corresponding truth table entries mark the disabled (disconnected) states. The reader should note that the disabled state,  $\phi$ , represents neither a 0 nor a 1, but rather the absence of a signal. In electrical circuit terms, the output is said to be in **high impedance**. The inverted control tri-state buffer is similar to the tri-state buffer, except that the control input  $C$  is complemented as indicated by the bubble at the control input.

An electrically disconnected output is different than an output that produces a 0, in that an electrically disconnected output behaves as if no output connection exists whereas a logical 0 at the output is still electrically connected to the circuit. The tri-state buffer allows the outputs from a number of logic gates to drive a common line without risking electrical shorts, provided that only one buffer is enabled at a time. The use of tri-state buffers is important in implementing **registers**, which are described later in this appendix.

A.5 Properties of Boolean Algebra

Table A.1 summarizes a few basic properties of Boolean algebra that can be applied to Boolean logic expressions. The postulates (known as "Huntington's

	Relationship	Dual	Property
Postulates	$A B = B A$	$A + B = B + A$	Commutative
	$A (B + C) = A B + A C$	$A + B C = (A + B) (A + C)$	Distributive
	$1 A = A$	$0 + A = A$	Identity
	$A \bar{A} = 0$	$A + \bar{A} = 1$	Complement
Theorems	$0 A = 0$	$1 + A = 1$	Zero and one theorems
	$A A = A$	$A + A = A$	Idempotence
	$A (B C) = (A B) C$	$A + (B + C) = (A + B) + C$	Associative
	$\bar{\bar{A}} = A$		Involution
	$\overline{A B} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A} \bar{B}$	DeMorgan's Theorem
	$AB + \bar{A}C + BC$ $= AB + \bar{A}C$	$(A + B)(\bar{A} + C)(B + C)$ $= (A + B)(\bar{A} + C)$	Consensus Theorem
	$A (A + B) = A$	$A + A B = A$	Absorption Theorem

**Table A.1 Basic properties of Boolean algebra.**

postulates”) are basic axioms of Boolean algebra and therefore need no proofs. The theorems can be proven from the postulates. Each relationship shown in the table has both an AND and an OR form as a result of the **principle of duality**. The dual form is obtained by changing ANDs to ORs, and changing ORs to ANDs.

The **commutative** property states that the order that two variables appear in an AND or OR function is not significant. By the principle of duality, the commutative property has an AND form ( $AB = BA$ ) and an OR form ( $A + B = B + A$ ). The **distributive** property shows how a variable is distributed over an expression with which it is ANDed. By the principle of duality, the dual form of the distributive property is obtained as shown.

The **identity** property states that a variable that is ANDed with 1 or is ORed with 0 produces the original variable. The **complement** property states that a variable that is ANDed with its complement is logically false (produces a 0, since at least one input is 0), and a variable that is ORed with its complement is logically true (produces a 1, since at least one input is 1).

The **zero** and **one** theorems state that a variable that is ANDed with 0 produces a 0, and a variable that is ORed with 1 produces a 1. The **idempotence** theorem

states that a variable that is ANDed or ORed with itself produces the original variable. For instance, if the inputs to an AND gate have the same value or the inputs to an OR gate have the same value, then the output for each gate is the same as the input. The **associative** theorem states that the order of ANDing or ORing is logically of no consequence. The **involution** theorem states that the complement of a complement leaves the original variable (or expression) unchanged.

**DeMorgan's theorem**, the **consensus theorem**, and the **absorption theorem** may not be obvious, and so we prove DeMorgan's theorem for the two-variable case using perfect induction (enumerating all cases), and leave the proofs of the consensus theorem and the absorption theorem as exercises (see problems A.24 and A.25.) Figure A-12 shows a truth table for each expression that appears in

$A$	$B$	$\overline{A B} = \overline{A} + \overline{B}$		$\overline{A + B} = \overline{A} \overline{B}$	
0	0	1	1	1	1
0	1	1	1	0	0
1	0	1	1	0	0
1	1	0	0	0	0

Figure A-12 DeMorgan's theorem is proven for the two-variable case.

either form of DeMorgan's theorem. The expressions that appear on the left and right sides of each form of DeMorgan's theorem produce equivalent outputs, which proves the theorem for two variables.

Not all of the logic gates discussed so far are necessary in order to achieve **computational completeness**, meaning that any digital logic circuit can be created from these gates. Three sets of logic gates that are computationally complete are: {AND, OR, NOT}, {NAND}, and {NOR} (there are others as well).

As an example of how a computationally complete set of logic gates can implement other logic gates that are not part of the set, consider implementing the OR function with the {NAND} set. DeMorgan's theorem can be used to map an OR gate onto a NAND gate, as shown in Figure A-13. The original OR function ( $A + B$ ) is complemented twice, which leaves the function unchanged by the involution property. DeMorgan's theorem then changes OR to AND, and distributes the innermost overbar over the terms  $A$  and  $B$ . The inverted inputs can also be implemented with NAND gates by the property of idempotence, as shown in Figure A-14. The OR function is thus implemented with NANDs. Functional

DeMorgan's theorem:  $A + B = \overline{\overline{A + B}} = \overline{\overline{A} \overline{B}}$



Figure A-13 DeMorgan's theorem is used in mapping an OR gate onto a NAND gate.

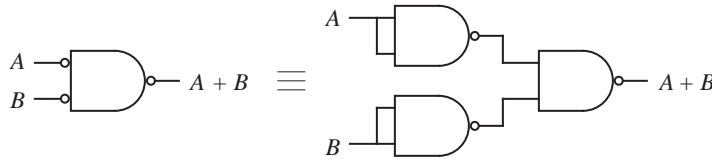


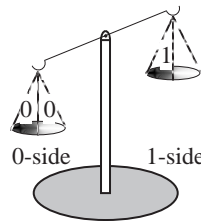
Figure A-14 Inverted inputs to a NAND gate implemented with NAND gates.

equivalence among logic gates is important for practical considerations, because one type of logic gate may have better operating characteristics than another for a given technology.

## A.6 The Sum-of-Products Form, and Logic Diagrams

Suppose now that we need to implement a more complex function than just a simple logic gate, such as the three-input **majority** function described by the truth table shown in Figure A-15. The majority function is true whenever more

Minterm Index	A	B	C	F
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1



A balance tips to the left or right depending on whether there are more 0's or 1's.

Figure A-15 Truth table for the majority function.

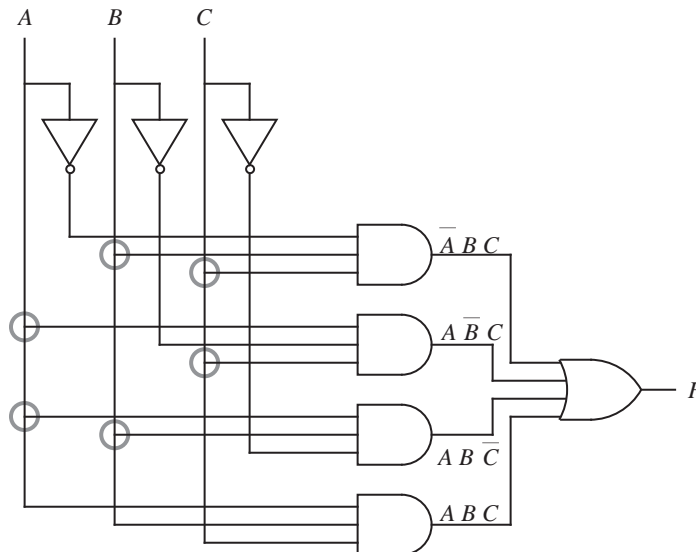
than half of its inputs are true, and can be thought of as a balance that tips to the left or right depending on whether there are more 0's or 1's at the input. This is a common operation used in fault recovery, in which the outputs of identical circuits operating on the same data are compared, and the greatest number of similar values determine the output (also referred to as "voting" or "odd one out").

Since no single logic gate discussed up to this point implements the majority function directly, we transform the function into a two-level AND-OR equation, and then implement the function with an arrangement of logic gates from the set {AND, OR, NOT} (for instance). The two levels come about because exactly one level of ANDed variables is followed by exactly one OR level. The Boolean equation that describes the majority function is true whenever  $F$  is true in the truth table. Thus,  $F$  is true when  $A=0$ ,  $B=1$ , and  $C=1$ , or when  $A=1$ ,  $B=0$ , and  $C=1$ , and so on for the remaining cases.

One way to represent logic equations is to use the **sum-of-products (SOP)** form, in which a collection of ANDed variables are ORed together. The Boolean logic equation that describes the majority function is shown in SOP form in Equation A.1. Again, the '+' signs denote logical OR and do not imply arithmetic addition.

$$F = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC \quad (\text{A.1})$$

By inspecting the equation, we can determine that four three-input AND gates will implement the four **product terms**  $\bar{A}BC$ ,  $A\bar{B}C$ ,  $AB\bar{C}$ , and  $ABC$ , and then the outputs of these four AND gates can be connected to the inputs of a four-input OR gate as shown in Figure A-16. This circuit performs the majority



**Figure A-16** A two-level AND-OR circuit implements the majority function. Inverters at the inputs are not included in the two-level count.

function, which we can verify by enumerating all eight input combinations and observing the output for each case.

The circuit diagram shows a commonly used notation that indicates the presence or absence of a connection, which is summarized in Figure A-17. Two lines that

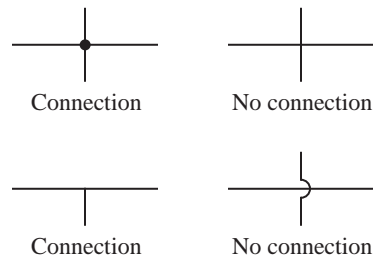


Figure A-17 Four notations used at circuit intersections.

pass through each other do not connect unless a darkened circle is placed at the intersection point. Two lines that meet in a  $\top$  are connected as indicated by the six highlighted intersections, and so darkened circles do not need to be placed over those intersection points.

When a product term contains exactly one instance of every variable, either in true or complemented form, it is called a **minterm**. A minterm has a value of 1 for exactly one of the entries in the truth table. That is, a *minimum* number of terms (one) will make the function true. As an alternative, the function is sometimes written as the logical sum over the true entries. Equation A.1 can be rewritten as shown in Equation A.2, in which the indices correspond to the minterm indices shown at the left in Figure A-15.

$$F = \sum \langle 3, 5, 6, 7 \rangle \quad (\text{A.2})$$

This notation is appropriate for the **canonical** form of a Boolean equation, which contains only minterms. Equations A.1 and A.2 are both said to be in “canonical sum-of-products form.”

## A.7 The Product-of-Sums Form

As a dual to the sum-of-products form, a Boolean equation can be represented in the **product-of-sums (POS)** form. An equation that is in POS form contains a collection of ORed variables that are ANDed together. One method of obtaining the POS form is to start with the complement of the SOP form, and then apply

DeMorgan's theorem. For example, referring again to the truth table for the majority function shown in Figure A-15, the complement is obtained by selecting input terms that produce 0's at the output, as shown in Equation A.3:

$$\bar{F} = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} \quad (\text{A.3})$$

Complementing both sides yields equation A.4:

$$F = \overline{\bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C}} \quad (\text{A.4})$$

Applying DeMorgan's theorem in the form  $\overline{W + X + Y + Z} = \bar{W}\bar{X}\bar{Y}\bar{Z}$  at the outermost level produces equation A.5:

$$F = (\bar{A}\bar{B}\bar{C})(\bar{A}\bar{B}C)(\bar{A}B\bar{C})(A\bar{B}\bar{C}) \quad (\text{A.5})$$

Applying DeMorgan's theorem in the form  $\overline{WXYZ} = \bar{W} + \bar{X} + \bar{Y} + \bar{Z}$  to the parenthesized terms produces equation A.6:

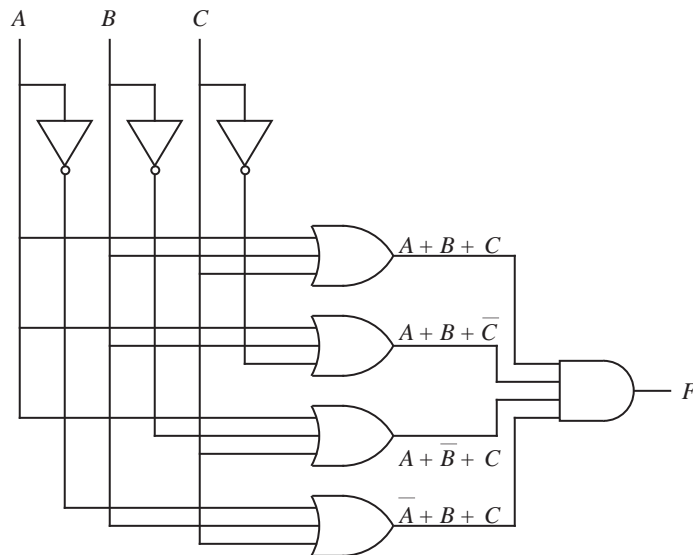
$$F = (A + B + C)(A + B + \bar{C})(A + \bar{B} + C)(\bar{A} + B + C) \quad (\text{A.6})$$

Equation A.6 is in POS form, and contains four **maxterms**, in which every variable appears exactly once in either true or complemented form. A maxterm, such as  $(A + B + C)$ , has a value of 0 for only one entry in the truth table. That is, it is true for the *maximum* number of truth table entries without reducing to the trivial function of always being true. An equation that consists of only maxterms in POS form is said to be in "canonical product-of-sums form." An OR-AND circuit that implements Equation A.6 is shown in Figure A-18. The OR-AND form is logically equivalent to the AND-OR form shown in Figure A-16.

One motivation for using the POS form over the SOP form is that it may result in a smaller Boolean equation. A smaller Boolean equation may result in a simpler circuit, although this does not always hold true since there are a number of considerations that do not directly depend on the size of the Boolean equation, such as the complexity of the wiring topology.

The **gate count** is a measure of circuit complexity that is obtained by counting all of the logic gates. The **gate input count** is another measure of circuit complexity that is obtained by counting the number of inputs to all of the logic gates. For the circuits shown in Figure A-16 and Figure A-18, a gate count of eight and





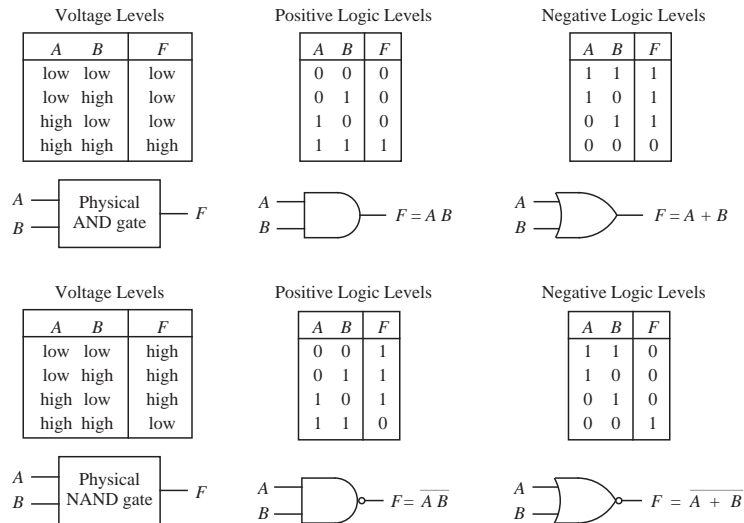
**Figure A-18** A two-level OR-AND circuit that implements the majority function. Inverters are not included in the two-level count.

a gate input count of 19 are obtained for both the SOP and POS forms. For this case, there is no difference in circuit complexity between the SOP and POS forms, but for other cases the differences can be significant. There is a variety of methods for reducing the complexity of digital circuits, a few of which are presented in Appendix B.

### A.8 Positive vs. Negative Logic

Up to this point we have assumed that high and low voltage levels correspond to logical 1 and 0, or TRUE and FALSE, respectively, which is known as **active high** or **positive logic**. We can make the reverse assignment instead: low voltage for logical 1 and high voltage for logical 0, which is known as **active low** or **negative logic**. The use of negative logic is sometimes preferred to positive logic for applications in which the logic inhibits an event rather than enabling an event.

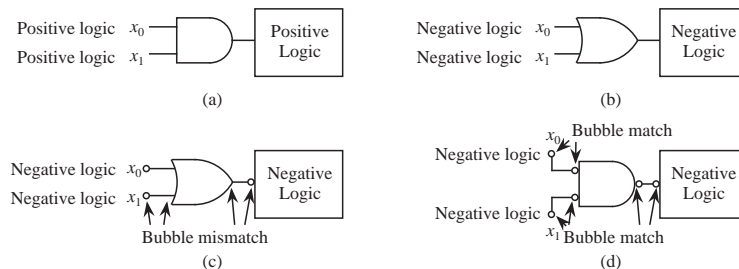
Figure A-19 illustrates the behavior of AND-OR and NAND-NOR gate pairs for both positive and negative logic. The positive logic AND gate behaves as a negative logic OR gate. The physical logic gate is the same regardless of the positive or negative sense of the logic – only the interpretation of the signals is changed.



**Figure A-19** Positive and negative logic assignments for AND-OR and NAND-NOR duals.

The mixing of positive and negative logic in the same system should be avoided to prevent confusion, but sometimes, it cannot be avoided. For these cases, a technique known as “bubble matching” helps keep the proper logic sense correct. The idea is to assume that all logic is asserted high (positive logic) and to place a bubble (denoting logical inversion) at the inputs or outputs of any negative logic circuits. Note that these bubbles are the same in function as the bubbles that appear at the complemented outputs of logic gates such as NOR and NAND. That is, the signal that leaves a bubble is the complement of the signal that enters it.

Consider the circuit shown in Figure A-20a, in which the outputs of two positive



**Figure A-20** The process of bubble matching.

logic circuits are combined through an AND gate that is connected to a positive logic system. A logically equivalent system for negative logic is shown in Figure A-20b. In the process of bubble matching, a bubble is placed on each active low input or output as shown in Figure A-20c.

To simplify the process of analyzing the circuit, active low input bubbles need to be matched with active low output bubbles. In Figure A-20c there are bubble mismatches because there is only one bubble on each line. DeMorgan's theorem is used in converting the OR gate in Figure A-20c to the NAND gate with complemented inputs in Figure A-20d, in which the bubble mismatches have been fixed.

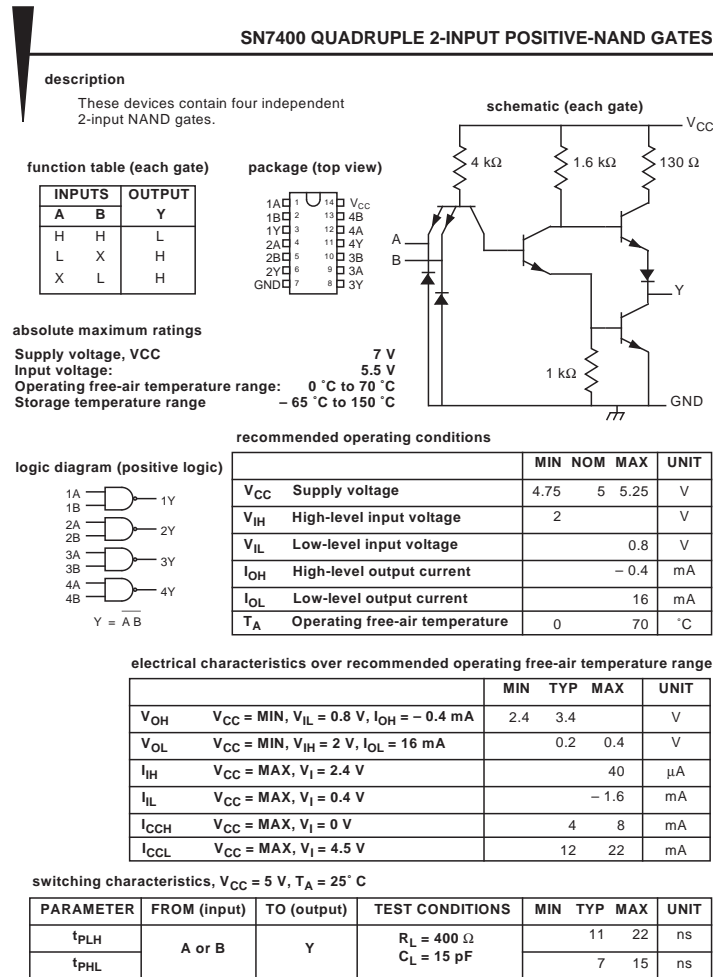
## A.9 The Data Sheet

Logic gates and other logic components have a great deal of technical specifications that are relevant to the design and analysis of digital circuits. The **data sheet**, or “spec sheet,” lists technical characteristics of a logic component. An example of a data sheet is shown in Figure A-21. The data sheet starts with a title for the component, which for this case is the SN7400 NAND gate. The description gives a functional description of the component in textual form.

The package section shows the pin layout and the pin assignments. There can be several package types for the same component. The function table enumerates the input-output behavior of the component from a functional perspective. The symbols “H” and “L” stand for “high” and “low” voltages respectively, to avoid confusion with the sense of positive or negative logic. The symbol “X” indicates that the value at an input does not influence the output. The logic diagram describes the logical behavior of the component, using positive logic for this case. All four NAND gates are shown with their pin assignments.

The schematic shows the transistor level circuitry for each gate. In the text, we treat this low level circuitry as an abstraction that is embodied in the logic gate symbols.

The “absolute maximum ratings” section lists the range of environmental conditions in which the component will safely operate. The supply voltage can go as high as 7 V and the input voltage can go up to 5.5 V. The ambient temperature should be between 0° C and 70° C during operation, but can vary between –65° C and 150° C when the component is not being used.



**Figure A-21** Simplified data sheet for 7400 NAND gate, adapted from Texas Instruments *TTL Databook* [Texas Instruments, 1988].

Despite the absolute maximum rating specifications, the recommended operating conditions should be used during operation. The recommended operating conditions are characterized by minimum (MIN), normal (NOM), and maximum (MAX) ratings.

The electrical characteristics describe the behavior of the component under certain operating conditions.  $V_{OH}$  and  $V_{OL}$  are the minimum output high voltage and the maximum output low voltage, respectively.  $I_{IH}$  and  $I_{IL}$  are the maximum

currents into an input pin when the input is high or low, respectively.  $I_{CCH}$  and  $I_{CCL}$  are the package's power supply currents when all outputs are high or low, respectively.

This data can be used in determining maximum **fan-outs** under the given conditions. Fan-out is a measure of the number of inputs that a single output can drive, for logic gates implemented in the same technology. That is, a logic gate with a fan-out of 10 can drive the inputs of 10 other logic gates of the same type. Similarly, **fan-in** is a measure of the number of inputs that a logic gate can accept (simply, the number of input lines to that gate). The absolute value of  $I_{OH}$  must be greater than or equal to the sum of all  $I_{IH}$  currents that are being driven, and  $I_{OL}$  must be greater than or equal to the sum of all  $I_{IL}$  currents (absolute values) that are being driven. The absolute value of  $I_{OH}$  for a 7400 gate is .4 mA (or 400  $\mu$ A), and so a 7400 gate output can thus drive ten 7400 inputs ( $I_{IH} = 40 \mu$ A per input).

The switching characteristics show the propagation delay to switch the output from a low to a high voltage ( $t_{PLH}$ ) and the propagation delay to switch the output from a high to a low voltage ( $t_{PHL}$ ). The maximum ratings show the worst cases. A circuit can be safely designed using the typical case as the worst case, but only if a test-and-select-the-best approach is used. That is, since  $t_{PLH}$  varies between 11 ns and 22 ns and  $t_{PHL}$  varies between 7 ns and 15 ns from one packaged component to the next, components can be individually tested to determine their true characteristics. Not all components of the same type behave identically, even under the most stringent fabrication controls, and the differences can be reduced by testing and selecting the best components.

## A.10 Digital Components

High level digital circuit designs are normally made using collections of logic gates referred to as **components**, rather than using individual logic gates. This allows a degree of circuit complexity to be abstracted away, and also simplifies the process of modeling the behavior of circuits and characterizing their performance. A few of the more common components are described in the sections that follow.

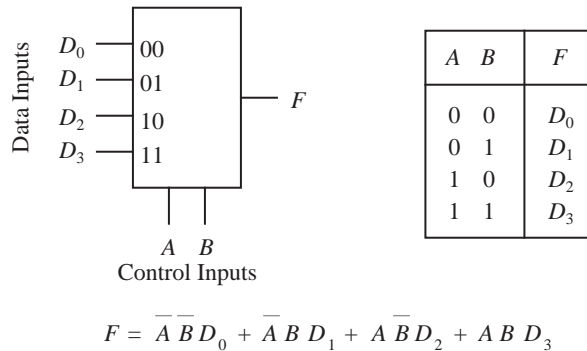
### A.10.1 LEVELS OF INTEGRATION

Up to this point, we have focused on the design of combinational logic units. Since we have been working with individual logic gates, we have been working at

the level of **small scale integration** (SSI), in which there are 10 – 100 components per chip. (“Components” has a different meaning in this context, referring to transistors and other discrete elements.) Although we sometimes need to work at this low level in practice, typically for high performance circuits, the advent of microelectronics allows us to work at higher levels of integration. In **medium scale integration** (MSI), approximately 100 – 1000 components appear in a single chip. **Large scale integration** (LSI) deals with circuits that contain 1000 – 10,000 components per chip, and **very large scale integration** (VLSI) goes higher still. There are no sharp breaks between the classes of integration, but the distinctions are useful in comparing the relative complexity of circuits. In this section we deal primarily with MSI components.

#### A.10.2 MULTIPLEXERS

A **multiplexer** (MUX) is a component that connects a number of inputs to a single output. A block diagram and the corresponding truth table for a 4-to-1 MUX are shown in Figure A-22. The output  $F$  takes on the value of the data input that

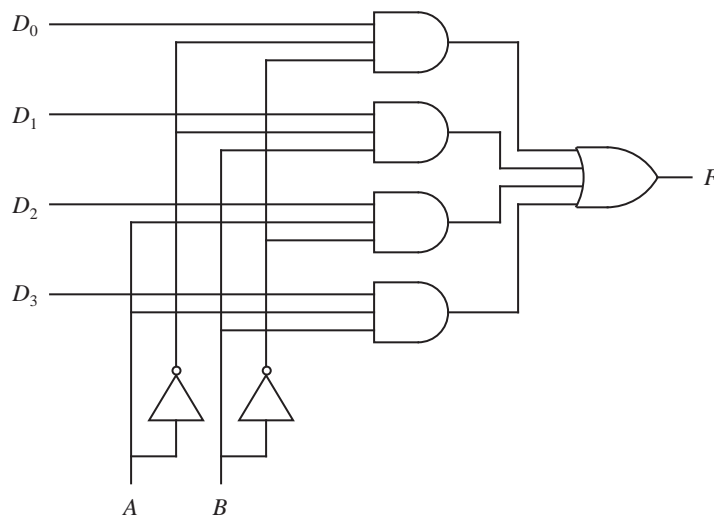


**Figure A-22** Block diagram and truth table for a 4-to-1 MUX.

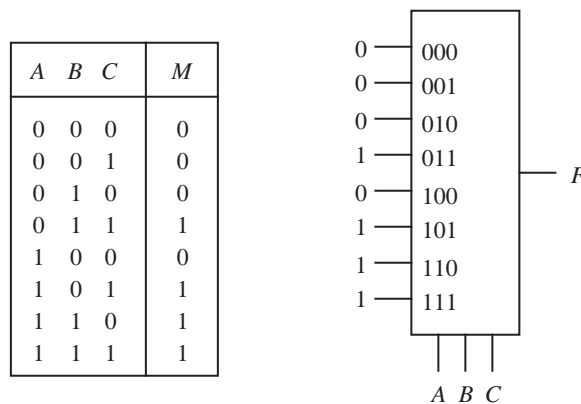
is selected by control lines  $A$  and  $B$ . For example, if  $AB = 00$ , then the value on line  $D_0$  (a 0 or a 1) will appear at  $F$ . The corresponding AND-OR circuit is shown in Figure A-23.

When we design circuits with MUXes, we normally use the “black box” form shown in Figure A-22, rather than the more detailed form shown in Figure A-23. In this way, we can abstract away detail when designing complex circuits.

Multiplexers can be used to implement Boolean functions. In Figure A-24, an 8-to-1 MUX implements the majority function. The data inputs are taken directly from the truth table for the majority function, and the control inputs are



**Figure A-23** An AND-OR circuit implements a 4-to-1 MUX.



**Figure A-24** An 8-to-1 multiplexer implements the majority function.

assigned to the variables  $A$ ,  $B$ , and  $C$ . The MUX implements the function by passing a 1 from the input of each true minterm to the output. The 0 inputs mark portions of the MUX that are not needed in implementing the function, and as a result, a number of logic gates are underutilized. Although portions of MUXes are almost always unused in implementing Boolean functions, multiplexers are widely used because their generality simplifies the design process, and their modularity simplifies the implementation.

As another case, consider implementing a function of three variables using a 4-to-1 MUX. Figure A-25 shows a three-variable truth table and a 4-to-1 MUX

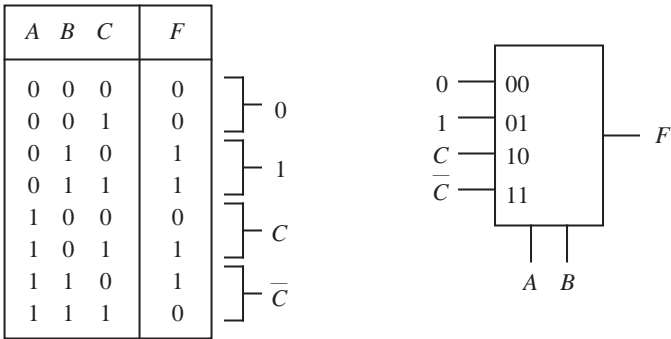


Figure A-25 A 4-to-1 MUX implements a three-variable function.

that implements function  $F$ . We allow data inputs to be taken from the set  $\{0, 1, C, \bar{C}\}$ , and the groupings are obtained as shown in the truth table. When  $AB = 00$ , then  $F = 0$  regardless of whether  $C = 0$  or  $C = 1$ , and so a 0 is placed at the corresponding 00 data input line on the MUX. When  $AB = 01$ , then  $F = 1$  regardless of whether  $C = 0$  or  $C = 1$ , and so a 1 is placed at the 01 data input. When  $AB = 10$ , then  $F = C$  since  $F$  is 0 when  $C$  is 0 and  $F$  is 1 when  $C$  is 1, and so  $C$  is placed at the 10 input. Finally, when  $AB = 11$ , then  $F = \bar{C}$ , and so  $\bar{C}$  is placed at the 11 input. In this way, we can implement a three-variable function using a two-variable MUX.

A.10.3 DEMULTIPLEXERS

A **demultiplexer** (DEMUX) is the converse of a MUX. A block diagram of a 1-to-4 DEMUX with control inputs  $A$  and  $B$  and the corresponding truth table are shown in Figure A-26. A DEMUX sends its single data input  $D$  to one of its

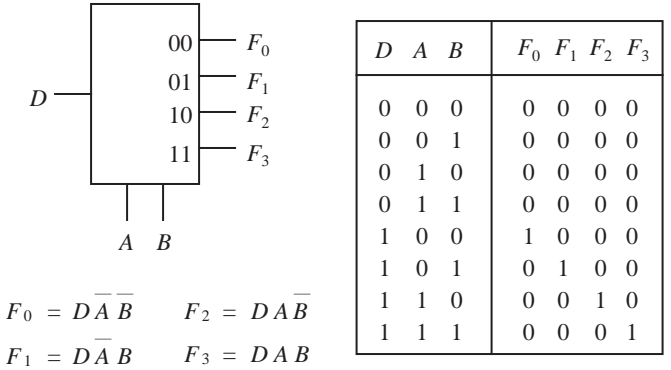


Figure A-26 Block diagram and truth table for a 1-to-4 DEMUX.



outputs  $F_i$  according to the settings of the control inputs. A circuit for a 1-to-4 DEMUX is shown in Figure A-27. An application for a DEMUX is to send data

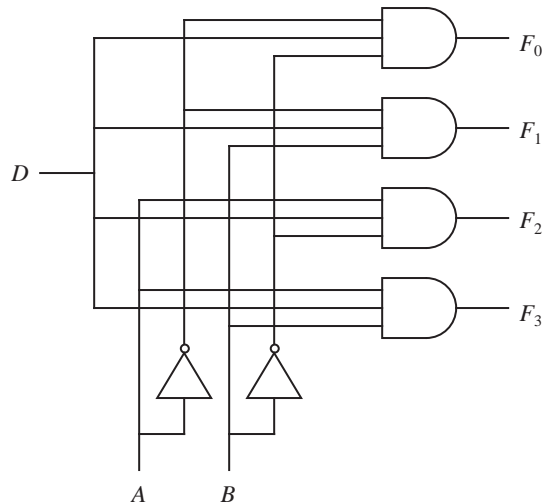


Figure A-27 A circuit for a 1-to-4 DEMUX.

from a single source to one of a number of destinations, such as from a call request button for an elevator to the closest elevator car. The DEMUX is not normally used in implementing ordinary Boolean functions, although there are ways to do it (see problem A.17).

#### A.10.4 DECODERS

A **decoder** translates a logical encoding into a spatial location. Exactly one output of a decoder is high (logical 1) at any time, which is determined by the settings on the control inputs. A block diagram and a truth table for a 2-to-4 decoder with control inputs  $A$  and  $B$  are shown in Figure A-28. A corresponding

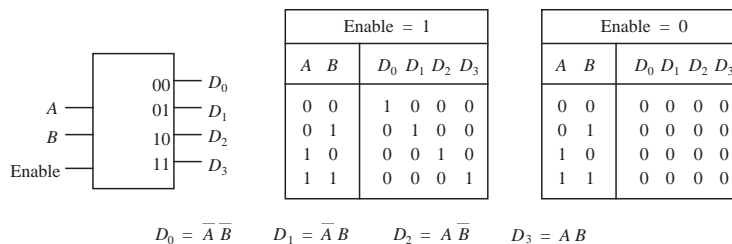
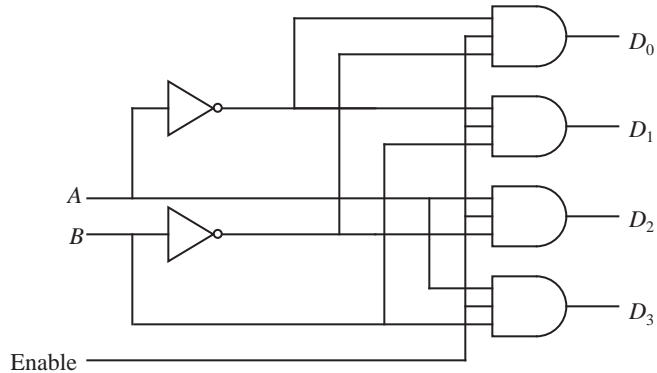


Figure A-28 Block diagram and truth table for a 2-to-4 decoder.

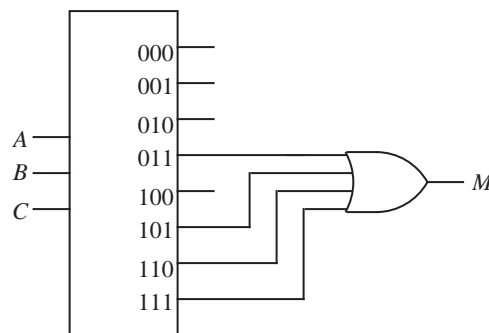
logic diagram that implements the decoder is shown in Figure A-29. A decoder



**Figure A-29** An AND circuit for a 2-to-4 decoder.

may be used to control other circuits, and at times it may be inappropriate to enable any of the other circuits. For that reason, we add an enable line to the decoder, which forces all outputs to 0 if a 0 is applied at its input. (Notice the logical equivalence between the DEMUX with an input of 1 and the decoder.)

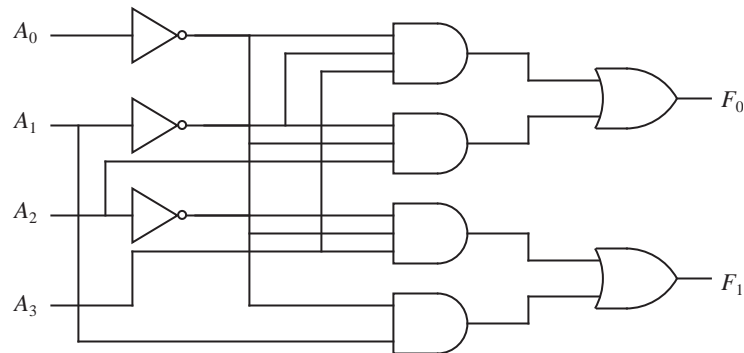
One application for a decoder is in translating memory addresses into physical locations. Decoders can also be used in implementing Boolean functions. Since each output line corresponds to a different minterm, a function can be implemented by logically ORing the outputs that correspond to the true minterms in the function. For example, in Figure A-30, a 3-to-8 decoder implements the



**Figure A-30** A 3-to-8 decoder implements the majority function.

majority function. Unused outputs remain disconnected.

A **programmable logic array** (PLA) is a component that consists of a customizable AND matrix followed by a customizable OR matrix. A PLA with three inputs and two outputs is shown in Figure A-33. The three inputs  $A$ ,  $B$ , and  $C$



**Figure A-32** Logic diagram for a 4-to-2 priority encoder.

and their complements are available at the inputs of each of eight AND gates that generate eight product terms. The outputs of the AND gates are available at the inputs of each of the OR gates that generate functions  $F_0$  and  $F_1$ . A programmable fuse is placed at each crosspoint in the AND and OR matrices. The matrices are customized for specific functions by disabling fuses. When a fuse is disabled at an input to an AND gate, then the AND gate behaves as if the input is tied to a 1. Similarly, a disabled input to an OR gate in a PLA behaves as if the input is tied to a 0.

As an example of how a PLA is used, consider implementing the majority function on a  $3 \times 2$  PLA (three input variables  $\times$  two output functions). In order to simplify the illustrations, the form shown in Figure A-34 is used, in which it is understood that the single input line into each AND gate represents six input lines, and the single input line into each OR gate represents eight input lines. Darkened circles are placed at the crosspoints to indicate where connections are made. In Figure A-34, the majority function is implemented using just half of the PLA, which leaves the rest of the PLA available for another function.

PLAs are workhorse components that are used throughout digital circuits. An advantage of using PLAs is that there are only a few inputs and a few outputs, while there is a large number of logic gates between the inputs and outputs. It is important to minimize the number of connections at the circuit edges in order to modularize a system into discrete components that are designed and implemented separately. A PLA is ideal for this purpose, and a number of automated programs exist for designing PLAs from functional descriptions. In keeping with this concept of modularity, we will sometimes represent a PLA as a black box as shown in Figure A-35, and assume that we can safely leave the design of the internals of the PLA to an automated program.

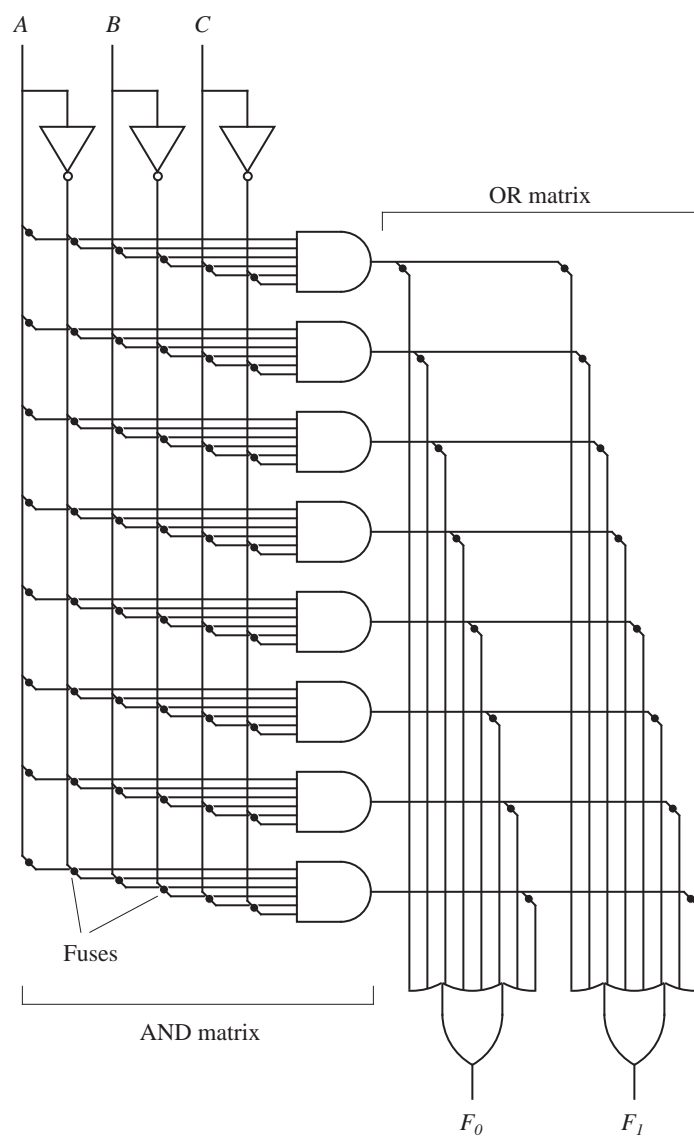


Figure A-33 A programmable logic array.

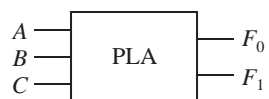


Figure A-35 Black box representation of a PLA.

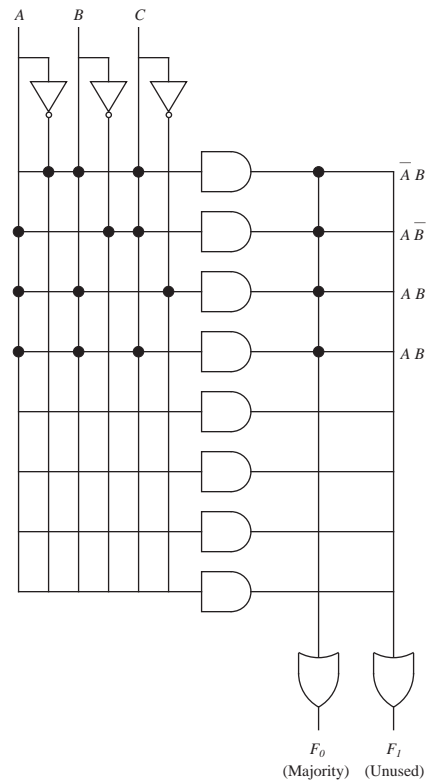


Figure A-34 Simplified representation of a PLA.

## EXAMPLE: A RIPPLE-CARRY ADDER

As an example of how PLAs are used in the design of a digital circuit, consider designing a circuit that adds two binary numbers. Binary addition is performed similar to the way we perform decimal addition by hand, as illustrated in Figure

Carry In	→	0	0	0	0	1	1	1	1
Operand A	→	0	0	1	1	0	0	1	1
Operand B	→	+ 0	+ 1	+ 0	+ 1	+ 0	+ 1	+ 0	+ 1
		<u>0 0</u>	<u>0 1</u>	<u>0 1</u>	<u>1 0</u>	<u>0 1</u>	<u>1 0</u>	<u>1 0</u>	<u>1 1</u>
		0 0	0 1	0 1	1 0	0 1	1 0	1 0	1 1
Carry Out	↑								
Sum	↑								

Example:	
Carry	1 0 0 0
Operand A	0 1 0 0
Operand B	+ 0 1 1 0
Sum	<u>1 0 1 0</u>

Figure A-36 Example of addition for two unsigned binary numbers.

A-36. Two binary numbers  $A$  and  $B$  are added from right to left, creating a sum and a carry in each bit position. Two input bits and a carry-in must be summed at each bit position, so that a total of eight input combinations must be considered as shown in the truth table in Figure A-37.

$A_i$	$B_i$	$C_i$	$S_i$	$C_{i+1}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

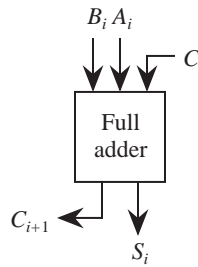


Figure A-37 Truth table for a full adder.

The truth table in Figure A-37 describes an element known as a **full adder**, which is shown schematically in the figure. A **half adder**, which could be used for the rightmost bit position, adds two bits and produces a sum and a carry, whereas a full adder adds two bits and a carry and produces a sum and a carry. The half adder is not used here in order to keep the number of different components to a minimum. Four full adders can be cascaded to form an adder large enough to add the four-bit numbers used in the example of Figure A-36, as shown in Figure A-38. The rightmost full adder has a carry-in ( $c_0$ ) of 0.

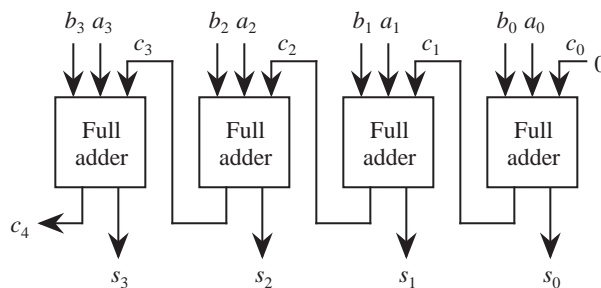


Figure A-38 A four-bit adder implemented with a cascade of full adders.

The reader will note that the value for a given sum bit cannot be computed until the carry-out from the previous full adder has been computed. The circuit is called a “ripple carry” adder because the correct values for the carry bits “ripple” through the circuit from right to left. The reader may also observe that even though the circuit looks “parallel,” in reality the sum bits are computed serially

from right to left. This is a major disadvantage to the circuit. We discuss ways of speeding up addition in Chapter 3, Arithmetic.

An approach to designing a full adder is to use a PLA, as shown in Figure A-39.

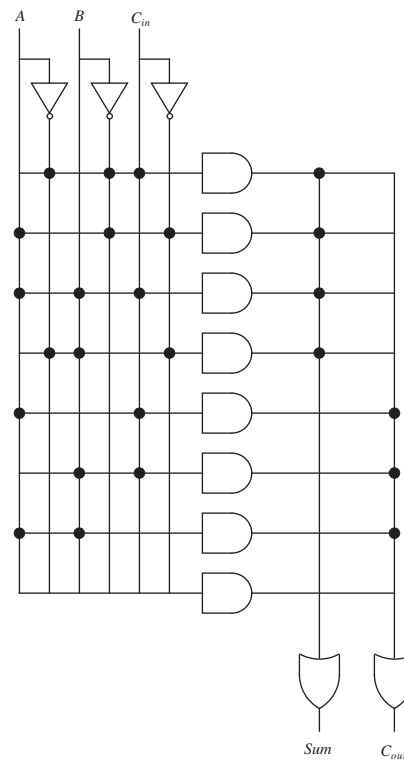


Figure A-39 PLA realization of a full adder.

The PLA approach is very general, and computer-aided design (CAD) tools for VLSI typically favor the use of PLAs over random logic or MUXes because of their generality. CAD tools typically reduce the sizes of the PLAs (we will see a few reduction techniques in Appendix B) and so the seemingly high gate count for the PLA is not actually so high in practice. ■

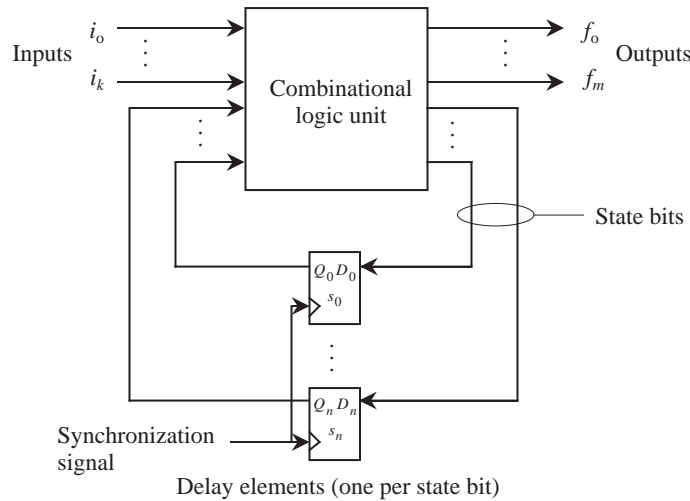
### A.11 Sequential Logic

In the earlier part of this appendix we explored combinational logic units, in which the outputs are completely determined by functions of the inputs. A sequential logic unit, commonly referred to as a **finite state machine** (FSM), takes an input and a current state, and produces an output and a new state. An FSM is distinguished from a CLU in that the past history of the inputs to the



FSM influences its state and output. This is important for implementing memory circuits as well as control units in a computer.

The classical model of a finite state machine is shown in Figure A-40. A CLU



**Figure A-40** Classical model of a finite state machine.

takes inputs from lines  $i_0 - i_k$  which are external to the FSM, and also takes inputs from state bits  $s_0 - s_n$  which are internal to the FSM. The CLU produces output bits  $f_0 - f_m$  and new state bits. Delay elements maintain the current state of the FSM, until a synchronization signal causes the  $D_i$  values to be loaded into the  $s_i$  which appear at  $Q_i$  as the new state bits.

#### A.11.1 THE S-R FLIP-FLOP

A **flip-flop** is an arrangement of logic gates that maintains a stable output even after the inputs are made inactive. The output of a flip-flop is determined by both the current inputs and the past history of inputs, and thus a combinational logic unit is not powerful enough to capture this behavior. A flip-flop can be used to store a single bit of information, and serves as a building block for computer memory.

If either or both inputs of a two-input NOR gate is 1, then the output of the NOR gate is 0, otherwise the output is 1. As we saw earlier in this appendix, the time that it takes for a signal to propagate from the inputs of a logic gate to the output is not instantaneous, and there is some delay  $\Delta\tau$  that represents the prop-

agation delay through the gate. The delay is sometimes considered lumped at the output of the gate for purposes of analysis, as illustrated in Figure A-41. The

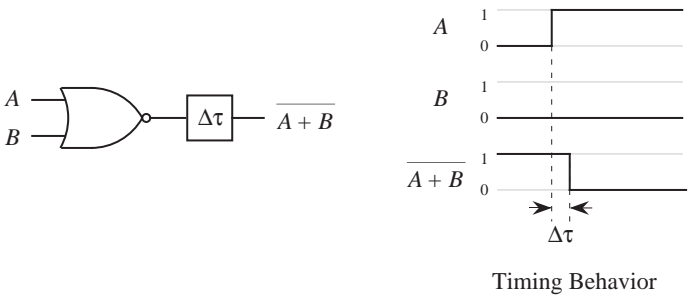


Figure A-41 A NOR gate with a lumped delay at its output.

lumped delay is not normally indicated in circuit diagrams but its presence is implied.

The propagation time through the NOR gate affects the operation of a flip-flop. Consider the **set-reset** (S-R) flip-flop shown in Figure A-42, which consists of

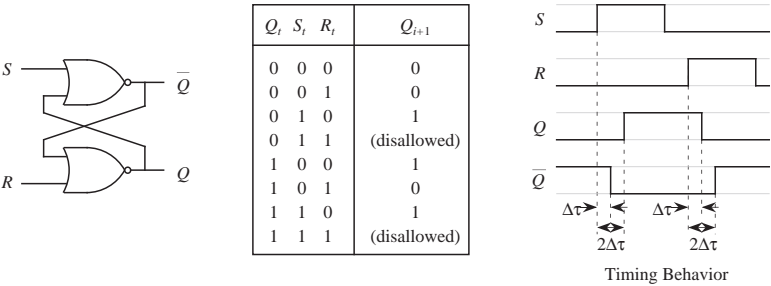
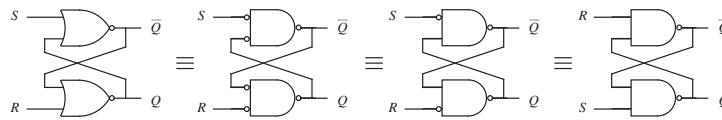


Figure A-42 An S-R flip-flop.

two cross-coupled NOR gates. If we apply a 1 to  $S$ , then  $\bar{Q}$  goes to 0 after a delay  $\Delta\tau$ , which causes  $Q$  to go to 1 (assuming  $R$  is initially 0) after a delay  $2\Delta\tau$ . As a result of the finite propagation time, there is a brief period of time  $\Delta\tau$  when both the  $Q$  and  $\bar{Q}$  outputs assume a value of 0, which is logically incorrect, but this condition will be fixed when the **master-slave** configuration is discussed later. If we now apply a 0 to  $S$ , then  $Q$  retains its state until some later time when  $R$  goes to 1. The S-R flip-flop thus holds a single bit of information and serves as an elementary memory element.

There is more than one way to make an S-R flip-flop, and the use of cross-coupled NOR gates is just one configuration. Two cross-coupled NAND gates can

also implement an S-R flip-flop, with  $S = R = 1$  being the quiescent state. Making use of DeMorgan's theorem, we can convert the NOR gates of an S-R flip-flop into AND gates as shown in Figure A-43. By “bubble pushing,” we change the



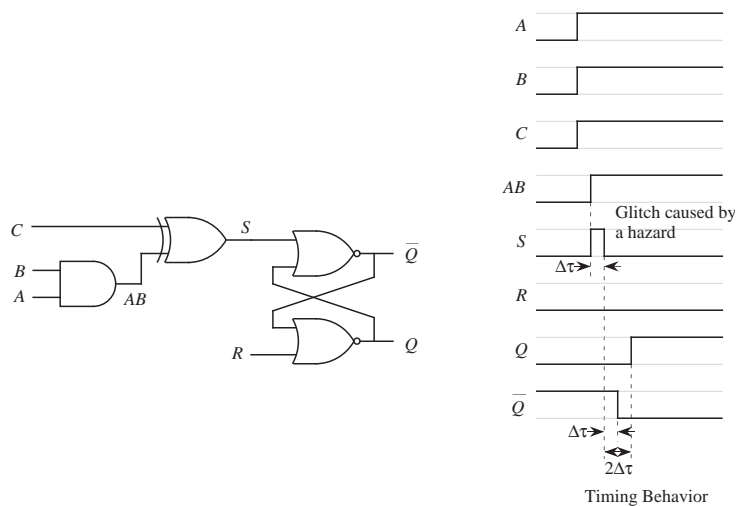
**Figure A-43** A NOR implementation of an S-R flip-flop is converted into a NAND implementation.

AND gates into NAND gates, and then reverse the sense of  $S$  and  $R$  to remove the remaining input bubbles.

#### A.11.2 THE CLOCKED S-R FLIP-FLOP

Now consider that the inputs to the S-R flip-flop may originate from the outputs of some other circuits, whose inputs may originate from the outputs of other circuits, forming a cascade of logic circuits. This mirrors the form of conventional digital circuits. A problem with cascading circuits is that transitions may occur at times that are not desired.

Consider the circuit shown in Figure A-44. If signals  $A$ ,  $B$ , and  $C$  all change from



**Figure A-44** A circuit with a hazard.

the 0 state to the 1 state, then signal  $C$  may reach the XOR gate before  $A$  and  $B$  propagate through the AND gate, which will momentarily produce a 1 output at  $S$ , which will revert to 0 when the output of the AND gate settles and is XORed with  $C$ . At this point it may be too late, however, since  $S$  may be in the 1 state long enough to set the flip-flop, destroying the integrity of the stored bit.

When the final state of a flip-flop is sensitive to the relative arrival times of signals, the result may be a **glitch**, which is an unwanted state or output. A circuit that can produce a glitch is said to have a **hazard**. The hazard may or may not manifest itself as a glitch, depending on the operating conditions of the circuit at a particular time.

In order to achieve synchronization in a controlled fashion, a **clock** signal is provided, to which every state-dependent circuit (such as a flip-flop) synchronizes itself by accepting inputs only at discrete times. A clock circuit produces a continuous stream of 1's and 0's, as indicated by the waveform shown in Figure

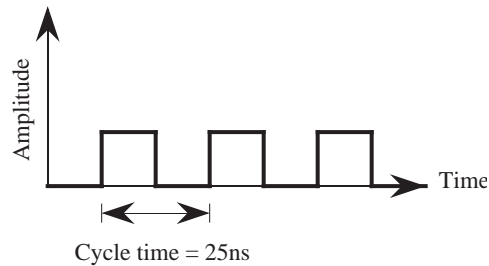


Figure A-45 A clock waveform.

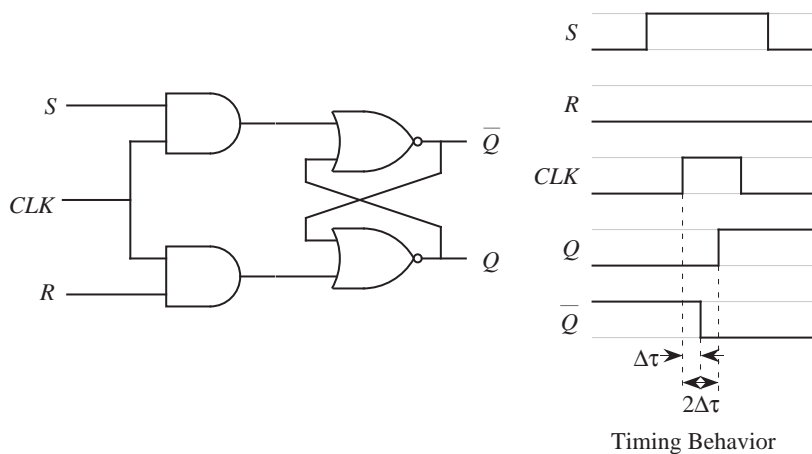
A-45. The time required for the clock to rise, then fall, then begin to rise again is called the **cycle time**. The square edges that are shown in the waveform represent an ideal square wave. In practice, the edges are rounded because instantaneous rise and fall times do not occur.

The **clock rate** is taken as the inverse of the cycle time. For a cycle time of 25 ns/cycle, the corresponding clock rate is  $1/25$  cycles/ns, which corresponds to 40,000,000 cycles per second, or 40 MHz (for 40 megahertz). A list of other abbreviations that are commonly used to specify cycle times and clock rates is shown in Table A.2.

We can make use of the clock signal to eliminate the hazard by creating a **clocked S-R flip-flop**, which is shown in Figure A-46. The symbol  $CLK$  labels the clock input. Now,  $S$  and  $R$  cannot change the state of the flip-flop until the

Prefix	Abbrev.	Quantity	Prefix	Abbrev.	Quantity
milli	m	$10^{-3}$	Kilo	K	$10^3$
micro	$\mu$	$10^{-6}$	Mega	M	$10^6$
nano	n	$10^{-9}$	Giga	G	$10^9$
pico	p	$10^{-12}$	Tera	T	$10^{12}$
femto	f	$10^{-15}$	Peta	P	$10^{15}$
atto	a	$10^{-18}$	Exa	E	$10^{18}$

**Table A.2** Standard scientific prefixes for cycle times and clock rates.



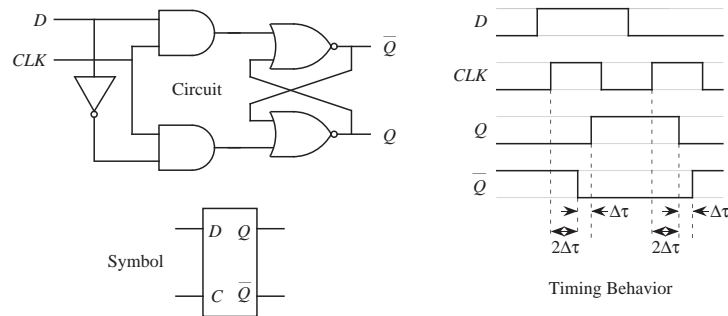
**Figure A-46** A clocked S-R flip-flop.

clock is high. Thus, as long as  $S$  and  $R$  settle into stable states while the clock is low, then when the clock makes a transition to 1, the stable value will be stored in the flip-flop.

### A.11.3 THE D FLIP-FLOP AND THE MASTER-SLAVE CONFIGURATION

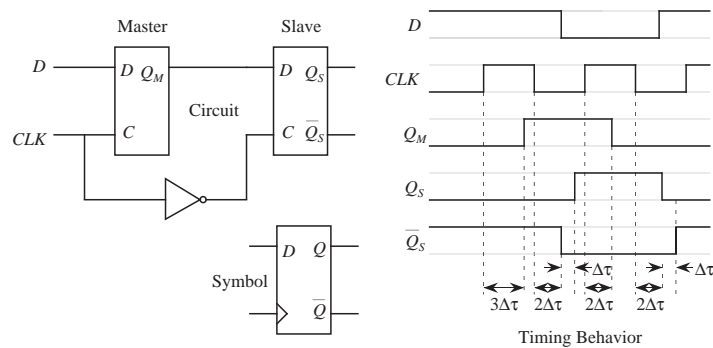
A disadvantage of the S-R flip-flop is that in order to store a 1 or a 0, we need to apply a 1 to a different input ( $S$  or  $R$ ) depending on the value that we want to store. An alternative configuration that allows either a 0 or a 1 to be applied at the input is the **D flip-flop** which is shown in Figure A-47. The D flip-flop is constructed by placing an inverter across the  $S$  and  $R$  inputs of an S-R flip-flop. Now, when the clock goes high, the value on the  $D$  line is stored in the flip-flop.

The D flip-flop is commonly used in situations where there is feedback from the output back to the input through some other circuitry, and this feedback can



**Figure A-47** A clocked D flip-flop. The letter 'C' denotes the clock input in the symbol form.

sometimes cause the flip-flop to change states more than once per clock cycle. In order to ensure that the flip-flop changes state just once per clock pulse, we break the feedback loop by constructing a master-slave flip-flop as shown in Figure



**Figure A-48** A master-slave flip-flop.

A-48. The **master-slave flip-flop** consists of two flip-flops arranged in tandem, with an inverted clock used for the second flip-flop. The master flip-flop changes when the clock is high, but the slave flip-flop does not change until the clock is low, thus the clock must first go high and then go low before the input at  $D$  in the master is clocked through to  $Q_S$  in the slave. The triangle shown in the symbol for the master-slave flip-flop indicates that transitions at the output occur only on a rising (0 to 1 transition) or falling (1 to 0 transition) edge of the clock. Transitions at the output do not occur continuously during a high level of the clock as for the clocked S-R flip-flop. For the configuration shown in Figure A-48, the transition at the output occurs on the falling edge of the clock.

A **level-triggered** flip-flop changes state continuously while the clock is high (or low, depending on how the flip-flop is designed). An **edge-triggered** flip-flop

changes only on a high-to-low or low-to-high clock transition. Some textbooks do not place a triangle at the clock input in order to distinguish between level-triggered and edge-triggered flip-flops, and indicate one form or the other based on their usage or in some other way. In practice the notation is held somewhat loosely. Here, we will use the triangle symbol and will also make the flip-flop type clear from the way it is used.

#### A.11.4 J-K AND T FLIP-FLOPS

In addition to the S-R and D flip-flops, the **J-K** and **T flip-flops** are very common. The J-K flip-flop behaves similarly to an S-R flip-flop, except that it flips its state when both inputs are set to 1. The T flip-flop (for “toggle”) alternates states, as when the inputs to a J-K flip-flop are set to 1. Logic diagrams and symbols for the clocked J-K and T flip-flops are shown in Figure A-49 and Figure A-50,

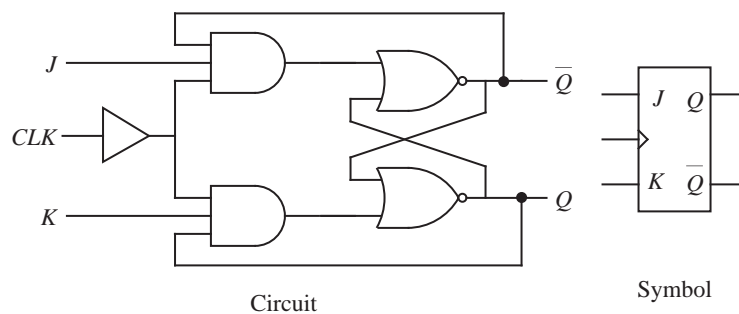


Figure A-49 Logic diagram and symbol for a basic J-K flip-flop.

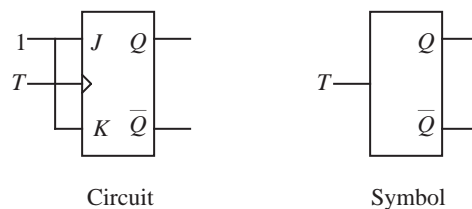
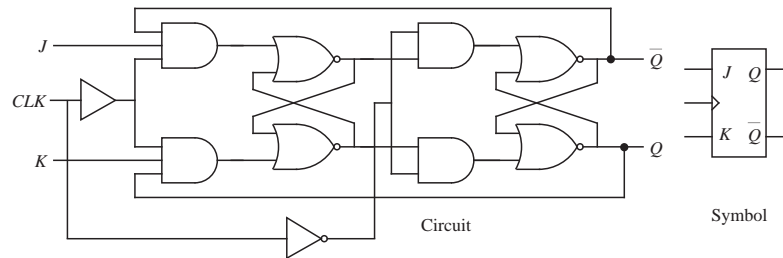


Figure A-50 Logic diagram and symbol for a T flip-flop.

respectively.

A problem with the toggle mode operation for the J-K flip-flop is that when  $J$  and  $K$  are both high when the clock is also high, the flip-flop may toggle more than once before the clock goes low. This is another situation in which a master-slave configuration is appropriate. A schematic diagram for a master-slave J-K

flip-flop is shown in Figure A-51. The “endless toggle” problem is now fixed with



**Figure A-51** Logic diagram and symbol for a master-slave J-K flip-flop.

this configuration, but there is a new problem of “one’s catching.” If an input is high for any time while the clock is high, and if the input is simply in a transition mode before settling, the flip-flop will “see” the 1 as if it was meant to be a valid input. The situation can be avoided if hazards are eliminated in the circuit that provides the inputs.

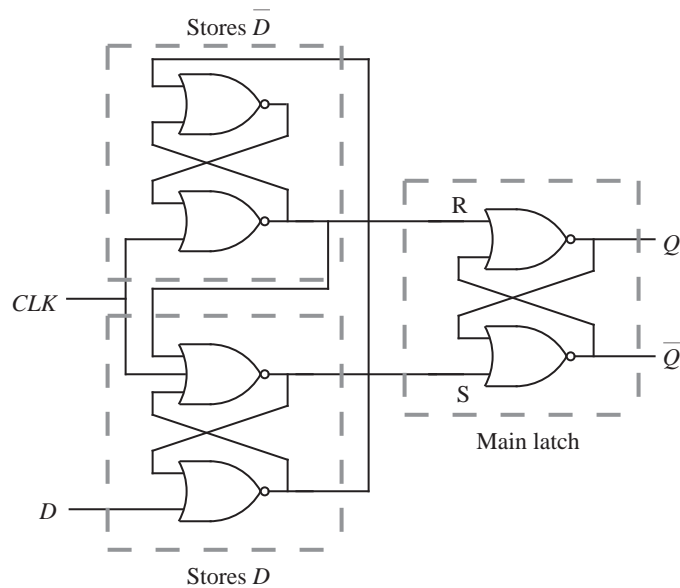
We can solve the one’s catching problem by constructing edge triggered flip-flops in which only the transition of the clock (low to high for positive edge triggered and high to low for negative edge triggered) causes the inputs to be sampled, at which point the inputs should be stable.

Figure A-52 shows a configuration for a negative edge triggered D flip-flop. When the clock is high, the top and bottom latches output 0’s to the main (output) S-R latch. The D input can change an arbitrary number of times while the clock is high without affecting the state of the main latch. When the clock goes low, only the settled values of the top and bottom latches affect the state of the main latch. While the clock is low, if the D input changes, the main flip-flop is not affected.

## A.12 Design of Finite State Machines

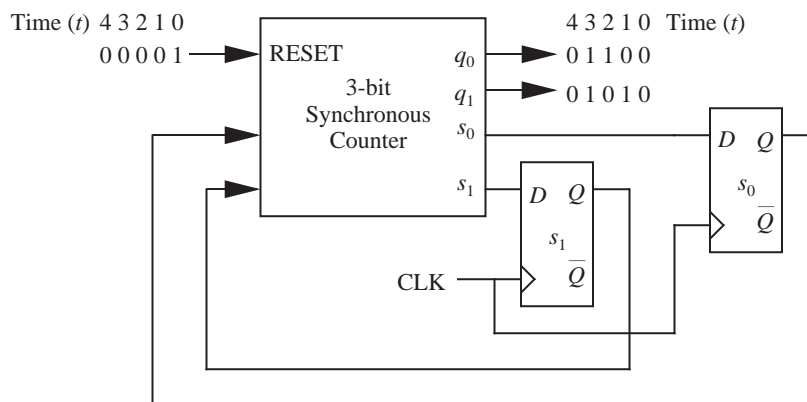
Refer again to the classical model of an FSM shown in Figure A-40. The delay elements can be implemented with master-slave flip-flops, and the synchronization signal can be provided by the clock. In general, there should be a flip-flop on each feedback line. Notice that we can label the flip-flops in any convenient way as long as the meaning is clear. In Figure A-40, the positions of the inputs  $D_i$  and the outputs  $Q_i$  have been interchanged with respect to the flip-flop figures in the previous section.





**Figure A-52** Negative edge triggered D flip-flop.

Consider a modulo(4) synchronous counter FSM, which counts from 00 to 11 and then repeats. A block diagram of a synchronous counter FSM is shown in Figure A-53. The RESET (positive logic) function operates synchronously with



**Figure A-53** A modulo(4) counter.

respect to the clock. The outputs appear as a sequence of values on lines  $q_0$  and  $q_1$  at time steps corresponding to the clock. As the outputs are generated, a new state  $s_1s_0$  is generated that is fed back to the input.

We can consider designing the counter by enumerating all possible input conditions and then creating four functions for the output  $q_1q_0$  and the state  $s_1s_0$ . The corresponding functions can then be used to create a combinational logic circuit that implements the counter. Two flip-flops are used for the two state bits.

How do we know that two state bits are needed on the feedback path? The fact is, we may not know in advance how many state bits are needed, and so we would like to have a more general approach to designing a finite state machine. For the counter, we can start by constructing a **state transition diagram** as shown in Figure A-54, in which each state represents a count from 00 to 11, and

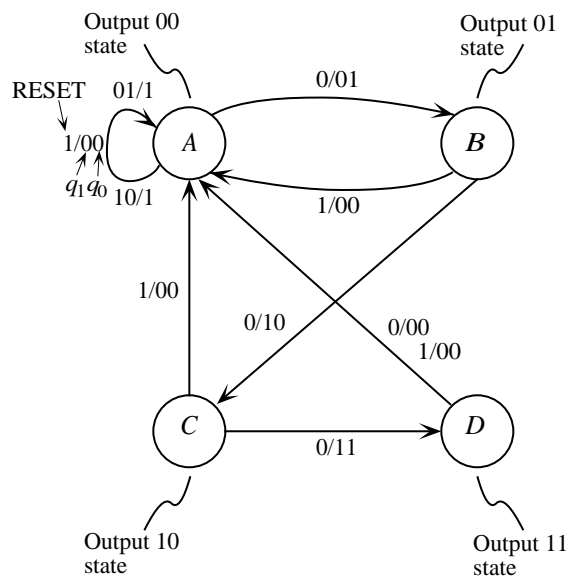


Figure A-54 State transition diagram for a modulo(4) counter.

the directed arcs represent transitions between states. State  $A$  represents the case in which the count is 00, and states  $B$ ,  $C$ , and  $D$  represent counts 01, 10, and 11 respectively.

Assume the FSM is initially in state  $A$ . There are two possible input conditions: 0 or 1. If the input (RESET) line is 0 then the FSM advances to state  $B$  and outputs 01. If the RESET line is 1, then the FSM remains in state  $A$  and outputs 00. Similarly, when the FSM is in state  $B$ , the FSM advances to state  $C$  and outputs 10 if the RESET line is 0, otherwise the FSM returns to state  $A$  and outputs 00. Transitions from the remaining states are interpreted similarly.

Once we have created the state transition diagram, we can rewrite it in tabular form as a **state table** as shown in Figure A-55. The present states are shown at the

Present state \ Input	RESET	
	0	1
A	B/01	A/00
B	C/10	A/00
C	D/11	A/00
D	A/00	A/00

Next state
Output

**Figure A-55** State table for a modulo(4) counter.

left, and the input conditions are shown at the top. The entries in the table correspond to next state/output pairs which are taken directly from the state transition diagram in Figure A-54. The highlighted entry corresponds to the case in which the present state is *B* and the input is 0. For this case, the next state is *C* and the next output is 10.

After we have created the state table, we encode the states in binary. Since there are four states, we need at least two bits to uniquely encode the states. We arbitrarily choose the encoding: *A* = 00, *B* = 01, *C* = 10, and *D* = 11, and replace every occurrence of *A*, *B*, *C*, and *D* with their respective encodings as shown in Figure A-56. In practice, the state encoding may affect the form of the resulting

Present state ( $S_t$ ) \ Input	RESET	
	0	1
A:00	01/01	00/00
B:01	10/10	00/00
C:10	11/11	00/00
D:11	00/00	00/00

**Figure A-56** State table with state assignments for a modulo(4) counter.

circuit, but the circuit will be logically correct regardless of the encoding.

From the state table, we can extract truth tables for the next state and output functions as shown in Figure A-57. The subscripts for the state variables indicate timing relationships.  $s_t$  is the present state and  $s_{t+1}$  is the next state. The subscripts are commonly omitted since it is understood that the present signals appear on the right side of the equation and the next signals appear on the left

<i>RESET</i> <i>r(t)</i>	<i>s<sub>1</sub>(t)</i>	<i>s<sub>0</sub>(t)</i>	<i>s<sub>1</sub>s<sub>0</sub>(t+1)</i>	<i>q<sub>1</sub>q<sub>0</sub>(t+1)</i>
0	0	0	01	01
0	0	1	10	10
0	1	0	11	11
0	1	1	00	00
1	0	0	00	00
1	0	1	00	00
1	1	0	00	00
1	1	1	00	00

$$s_0(t+1) = \overline{r(t)}\overline{s_1(t)}\overline{s_0(t)} + \overline{r(t)}s_1(t)\overline{s_0(t)}$$

$$s_1(t+1) = \overline{r(t)}s_1(t)s_0(t) + \overline{r(t)}s_1(t)\overline{s_0(t)}$$

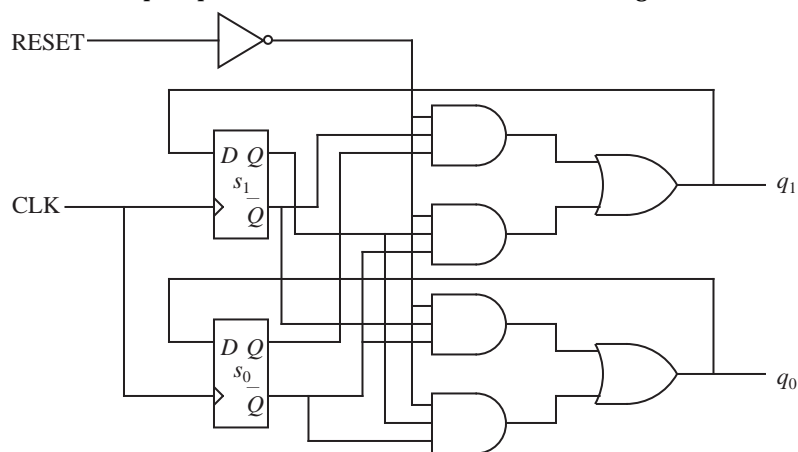
$$q_0(t+1) = \overline{r(t)}\overline{s_1(t)}\overline{s_0(t)} + \overline{r(t)}s_1(t)\overline{s_0(t)}$$

$$q_1(t+1) = \overline{r(t)}s_1(t)s_0(t) + \overline{r(t)}s_1(t)\overline{s_0(t)}$$

**Figure A-57 Truth table for the next state and output functions for a modulo(4) counter.**

side of the equation. Notice that  $s_0(t+1) = q_0(t+1)$  and  $s_1(t+1) = q_1(t+1)$ , so we only need to implement  $s_0(t+1)$  and  $s_1(t+1)$  and tap the outputs for  $q_0(t+1)$  and  $q_1(t+1)$ .

Finally, we implement the next state and output functions using logic gates and master-slave D flip-flops for the state variables as shown in Figure A-58.



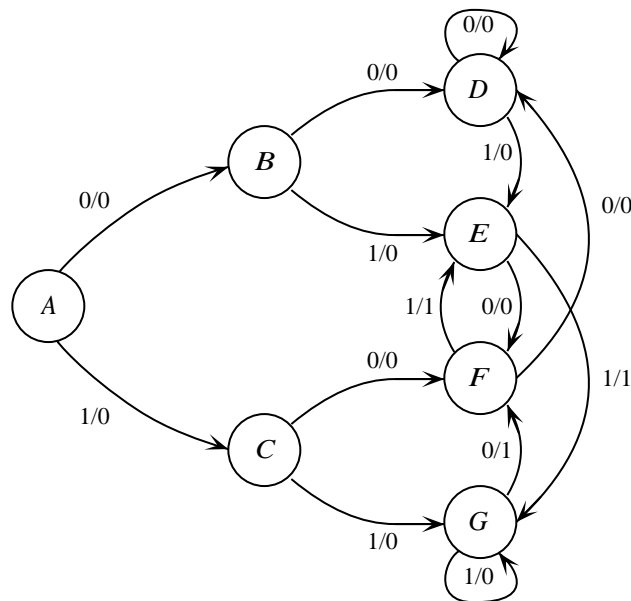
**Figure A-58 Logic design for a modulo(4) counter.**

## EXAMPLE: A SEQUENCE DETECTOR

As another example, we would like to design a machine that outputs a 1 when exactly two of the last three inputs are 1. For example, an input sequence of 011011100 produces an output sequence of 001111010. There is a one-bit serial

input line, and we can assume that initially no inputs have been seen. For this problem, we will use D flip-flops and 8-to-1 MUXes.

We start by constructing a state transition diagram, as shown in Figure A-59.



**Figure A-59** State transition diagram for sequence detector.

There are eight possible three-bit sequences that our machine will observe: 000, 001, 010, 011, 100, 101, 110, and 111. State *A* is the initial state, in which we assume that no inputs have yet been seen. In states *B* and *C*, we have seen only one input, so we cannot output a 1 yet. In states *D*, *E*, *F*, and *G* we have only seen two inputs, so we cannot output a 1 yet, even though we have seen two 1's at the input when we enter state *G*. The machine makes all subsequent transitions among states *D*, *E*, *F*, and *G*. State *D* is visited when the last two inputs are 00. States *E*, *F*, and *G* are visited when the last two inputs are 01, 10, or 11, respectively.

The next step is to create a state table as shown in Figure A-60, which is taken directly from the state transition diagram. Next, we make a state assignment as shown in Figure A-61a. We then use the state assignment to create a truth table for the next state and output functions as shown in Figure A-61b. The last two entries in the table correspond to state 111, which cannot arise in practice,

Present state \ Input	X	
	0	1
A	B/0	C/0
B	D/0	E/0
C	F/0	G/0
D	D/0	E/0
E	F/0	G/1
F	D/0	E/1
G	F/1	G/0

Figure A-60 State table for sequence detector.

Present state \ Input	X		Input and state at time $t$				Next state and output at time $t+1$			
	0	1	$s_2$	$s_1$	$s_0$	$x$	$s_2$	$s_1$	$s_0$	$z$
$s_2s_1s_0$	$s_2s_1s_0z$	$s_2s_1s_0z$	0	0	0	0	0	0	1	0
A: 000	001/0	010/0	0	0	0	1	0	1	0	0
B: 001	011/0	100/0	0	0	1	0	0	1	1	0
C: 010	101/0	110/0	0	0	1	1	1	0	0	0
D: 011	011/0	100/0	0	1	0	0	1	0	1	0
E: 100	101/0	110/1	0	1	0	1	1	1	0	0
F: 101	011/0	100/1	0	1	1	0	0	1	1	0
G: 110	101/1	110/0	0	1	1	1	1	0	0	0
			1	0	0	0	1	0	1	0
			1	0	0	1	1	1	0	1
			1	0	1	0	0	1	1	0
			1	0	1	1	1	0	0	1
			1	1	0	0	1	0	1	1
			1	1	0	1	1	1	0	0
			1	1	1	0	d	d	d	d
			1	1	1	1	d	d	d	d

(a)

(b)

Figure A-61 State assignment and truth table for sequence detector.

according to the state table in Figure A-61a. Therefore, the next state and output entries do not matter, and are labeled as 'd' for **don't care**.

Finally, we create the circuit, which is shown in Figure A-62. There is one flip-flop for each state variable, so there are a total of three flip-flops. There are three next state functions and one output function, so there are four MUXes. Notice that the choice of  $s_2$ ,  $s_1$ , and  $s_0$  for the MUX control inputs is arbitrary. Any other grouping or ordering will also work.

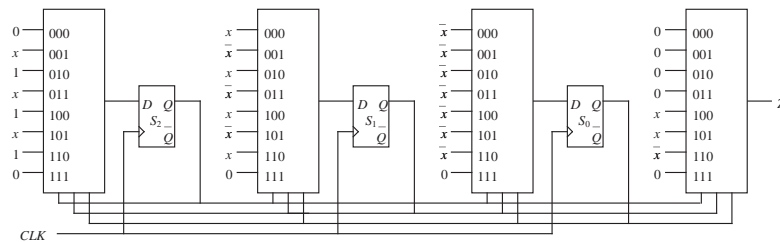


Figure A-62 Logic diagram for sequence detector.

## EXAMPLE: A VENDING MACHINE CONTROLLER

For this problem, we will design a vending machine controller using D flip-flops and a black box representation of a PLA (as in Figure A-35). The vending machine accepts three U.S. coins: the nickel (5¢), the dime (10¢), and the quarter (25¢). When the value of the inserted coins equals or exceeds 20¢, then the machine dispenses the merchandise, returns any excess money, and waits for the next transaction.

We begin by constructing a state transition diagram, as shown in Figure A-63. In

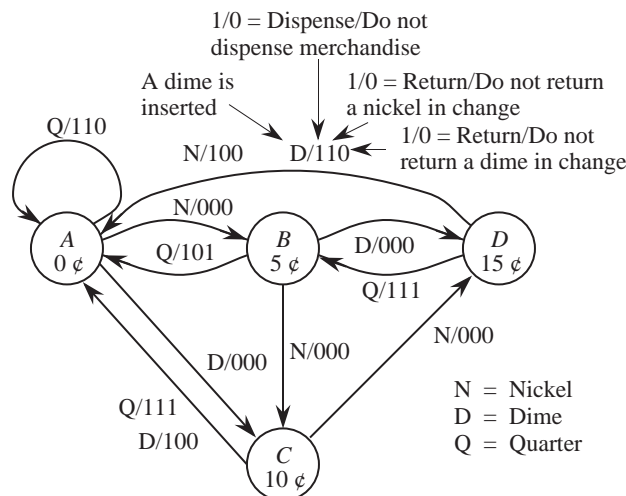


Figure A-63 State transition diagram for vending machine controller.

state *A*, no coins have yet been inserted, and so the money credited is 0¢. If a nickel or dime is inserted when the machine is in state *A*, then the FSM makes a transition to state *B* or state *C*, respectively. If a quarter is inserted, then the

money credited to the customer is 25¢. The controller dispenses the merchandise, returns a nickel in change, and remains in state *A*. This is indicated by the label “*Q/110*” on the state *A* self-loop. States *B* and *C* are then expanded, producing state *D* which is also expanded, producing the complete FSM for the vending machine controller.

Notice the behavior that is specified by the state transition diagram when a quarter is inserted when the FSM is in state *D*. Rather than dispensing product, returning 20¢, and returning to state *A*, the machine dispenses product, returns 15¢, and makes a transition to state *B*. The machine keeps the 5¢, and awaits the insertion of more money! In this case, the authors allowed this behavior for the sake of simplicity, as it keeps the number of states down.

From the FSM we construct the state table shown in Figure A-64a. We then

Input P.S.	N 00	D 01	Q 10
<i>A</i>	<i>B/000 C/000 A/110</i>		
<i>B</i>	<i>C/000 D/000 A/101</i>		
<i>C</i>	<i>D/000 A/100 A/111</i>		
<i>D</i>	<i>A/100 A/110 B/111</i>		

(a)

Input P.S.	N $x_1x_0$ 00	D $x_1x_0$ 01	Q $x_1x_0$ 10
$s_1s_0$	$s_1s_0 / z_2z_1z_0$		
<i>A:00</i>	01/000	10/000	00/110
<i>B:01</i>	10/000	11/000	00/101
<i>C:10</i>	11/000	00/100	00/111
<i>D:11</i>	00/100	00/110	01/111

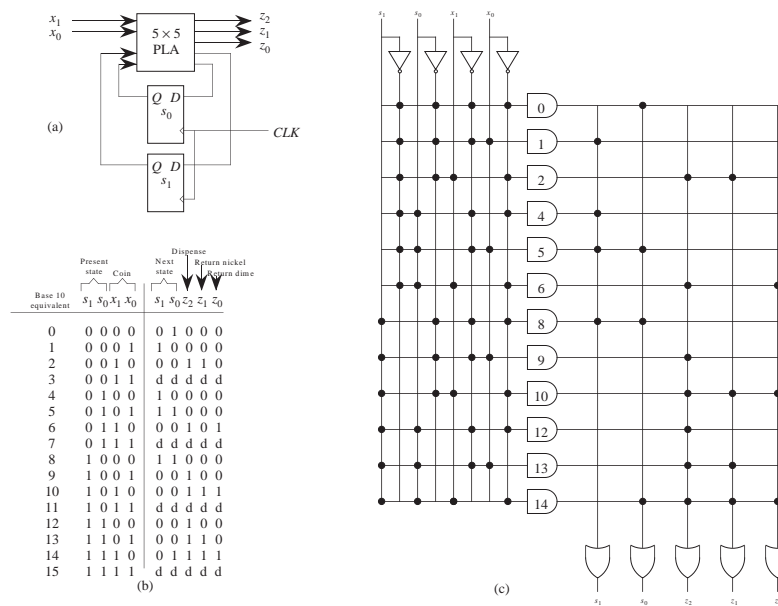
(b)

**Figure A-64** (a) State table for vending machine controller; (b) state assignment for vending machine controller.

make an arbitrary state assignment and encode the symbols *N*, *D*, and *Q* in binary as shown in Figure A-64b. Finally, we create a circuit diagram, which is shown in Figure A-65a. There are two state bits, so there are two D flip-flops. The PLA takes four inputs for the present-state bits and the  $x_1x_0$  coin bits. The PLA produces five outputs for the next-state bits and the dispense and return nickel/return dime bits. (We can assume that the clock input is asserted only on an event such as an inserted coin.)

Notice that we have not explicitly specified the design of the PLA itself in obtaining the FSM circuit in Figure A-65a. At this level of complexity, it is common to use a computer program to generate a truth table, and then feed the truth table to a PLA design program. We could generate the truth table and PLA design by hand, of course, as shown in Figure A-65b and Figure A-65c.





**Figure A-65** (a) FSM circuit, (b) truth table, and (c) PLA realization for vending machine controller.

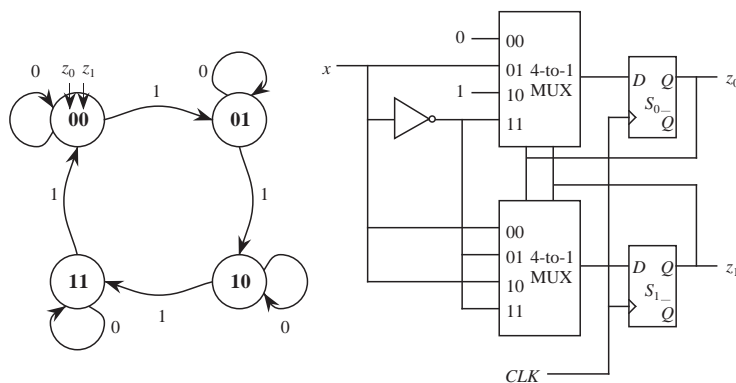
### A.13 Mealy vs. Moore Machines

The outputs of the FSM circuits we have studied so far are determined by the present states and the inputs. The states are maintained in falling edge triggered flip-flops, and so a state change can only occur on the falling edge of the clock. Any changes that occur at the inputs have no effect on the state as long as the clock is low. The inputs are fed directly through the output circuits, however, with no intervening flip-flops. Thus a change to an input at any time can cause a change in the output, regardless of whether the clock is high or low. In Figure A-65, a change at either the  $x_1$  or  $x_0$  inputs will propagate through to the  $z_2z_1z_0$  outputs independent of the level of the clock. This organization is referred to as the **Mealy** model of an FSM.

In the Mealy model, the outputs change as soon as the inputs change, and so there is no delay introduced by the clock. In the **Moore** model of an FSM, the outputs are embedded in the state bits, and so a change at the outputs occurs on the clock pulse *after* a change at the inputs. Both models are used by circuit designers, and either model may be encountered outside of this textbook. In this

section we simply highlight the differences through an example.

An example of a Moore FSM is shown in Figure A-66. The FSM counts from 0



**Figure A-66** A Moore binary counter FSM.

to 3 in binary and then repeats, similar to the modulo(4) counter shown in Figure A-58. The machine only counts when  $x = 1$ , otherwise the FSM maintains its current state. Notice that the outputs are embedded in the state variables, and so there is no direct path from the input to the outputs without an intervening flip-flop.

The Mealy model might be considered to be more powerful than the Moore model because in a single clock cycle, a change in the output of one FSM can ripple to the input of another FSM, whose output then changes and ripples to the next FSM, and so on. In the Moore model, lock-step synchronization is strictly maintained, and so this ripple scenario does not occur. Spurious changes in the output of an FSM thus have less influence on the rest of the circuit in the Moore model. This simplifies circuit analysis and hardware debugging, and for these situations, the Moore model may be preferred. In practice, both models are used.

#### A.14 Registers

A single bit of information is stored in a D flip-flop. A group of  $N$  bits, making up an  $N$ -bit word, can be stored in  $N$  D flip-flops organized as shown in Figure A-67 for a four-bit word. We refer to such an arrangement of flip-flops as a “register.” In this particular configuration, the data at inputs  $D_i$  are loaded into the register when the Write and Enable lines are high, synchronous with the clock.

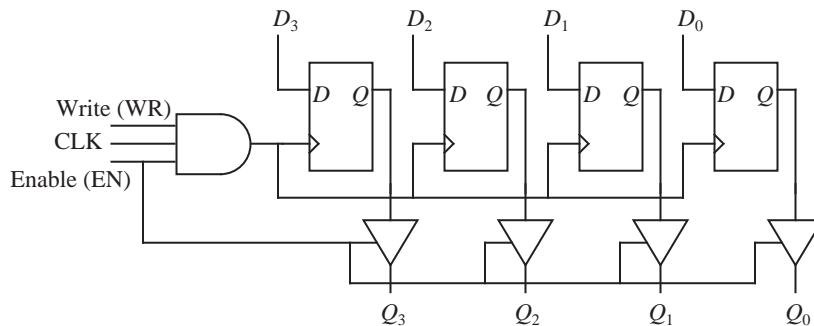


Figure A-67 A four-bit register.

The contents of the register can be read at outputs  $Q_i$  only if the Enable line is high, since the tri-state buffers are in the electrically disconnected state when the Enable line is low. We can simplify the illustration by just marking the inputs and outputs as shown in Figure A-68.

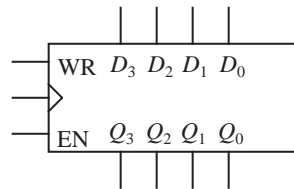


Figure A-68 Abstract representation of a four-bit register.

A **shift register** copies the contents of each of its flip-flops to the next, while accepting a new input at one end and “spilling” the contents at the other end, which makes cascading possible. Consider the shift register shown in Figure A-69. The register can shift to the left, shift to the right, accept a parallel load, or remain unchanged, all synchronous with the clock. The parallel load and parallel read capabilities allow the shift register to function as either a **serial-to-parallel converter** or as a **parallel-to-serial converter**.

### A.15 Counters

A **counter** is a different form of a register in which the output pattern sequences through a range of binary numbers. Figure A-70 shows a configuration for a modulo(8) counter that steps through the binary patterns: 000, 001, 010, 011, 100, 101, 110, 111 and then repeats. Three J-K flip-flops are placed in toggle mode, and each clock input is ANDed with the  $Q$  output from the previous stage, which successively halves the clock frequency. The result is a progression of

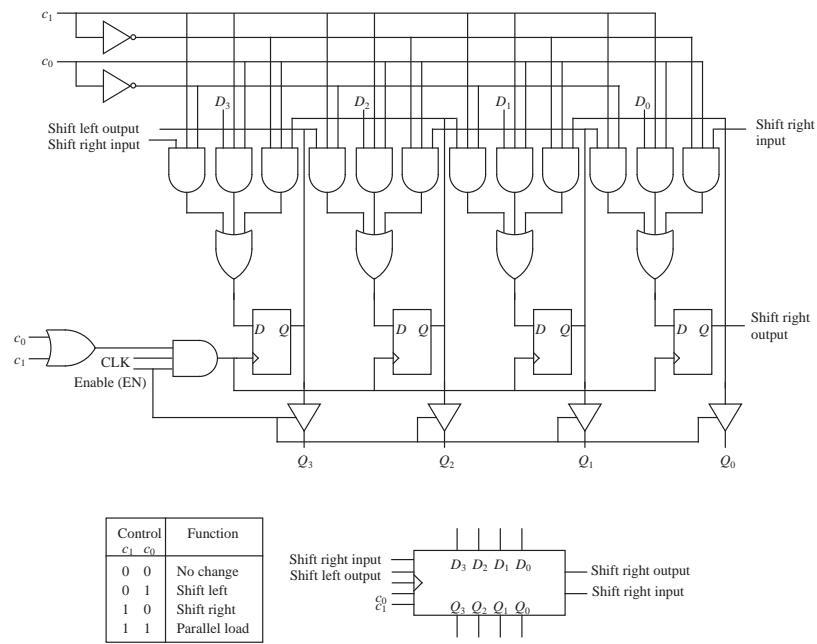


Figure A-69 Internal layout and block diagram for a left/right shifter with parallel read/write capabilities.

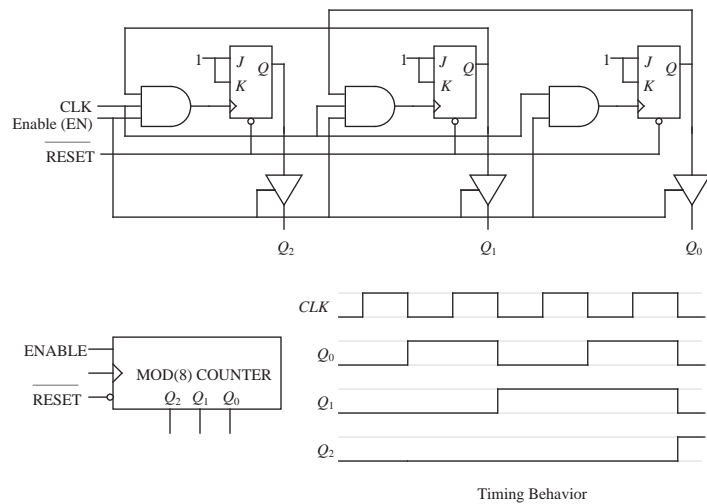


Figure A-70 A modulo(8) counter.

toggle flip-flops operating at rates that differ in powers of two, corresponding to

the sequence of binary patterns from 000 to 111.

Notice that we have added an active low asynchronous RESET line to the counter, which resets it to 000, independent of the states of the clock or enable lines. Except for the flip-flop in the least significant position, the remaining flip-flops change state according to changes in states from their neighbors to the right rather than synchronous with respect to the clock. It is similar in function to the modulo(4) counter in Figure A-58, but is more easily extended to large sizes because it is not treated like an ordinary FSM for design purposes, in which all states are enumerated. It is, nevertheless, an FSM.

## ■ SUMMARY

*In theory, any Boolean function can be represented as a truth table, which can then be transformed into a two-level Boolean equation and implemented with logic gates. In practice, collections of logic gates may be grouped together to form MSI components, which contain on the order of a few to a few dozen logic gates. MUXes and PLAs are two types of MSI components that are used for implementing functions. Decoders are used for enabling a single output line based on the bit pattern at the input, which translates a logical encoding into a spatial location. There are several other types of MSI components as well. We find ourselves using MSI components in abstracting away the gate level complexity of a digital circuit. LSI and VLSI circuits abstract away the underlying circuit complexity at higher levels still.*

*A finite state machine (FSM) differs from a combinational logic unit (CLU) in that the outputs of a CLU at any time are strictly a function of the inputs at that time whereas the outputs of an FSM are a function of its past history of inputs.*

## ■ Further Reading

Shannon's contributions to switching algebra (Shannon, 1938; Shannon, 1949) are based on the work of (Boole, 1854), and form the basis of switching theory as we now know it. There is a vast number of contributions to Boolean algebra that are too great to enumerate here. (Kohavi, 1978) is a good general reference for CLUs and FSMs. A contribution by (Davidson, 1979) covers a method of decomposing NAND based circuits, which is of interest because some computers

are composed entirely of NAND gates.

(Xilinx, 1992) covers the philosophy and practical aspects of the gate array approach, and describes configurations of the Xilinx line of **field programmable gate arrays** (FPGAs).

Some texts distinguish between a flip-flop and a latch. (Tanenbaum, 1990) distinguishes between the two by defining a flip-flop to be edge-triggered, whereas a latch is level-triggered. This may be the correct definition, but in practice, the terms are frequently interchanged and any distinction between the two is obscured.

Boole, G., *An Investigation of the Laws of Thought*, Dover Publications, Inc., New York, (1854).

Davidson, E. S., "An algorithm for NAND decomposition under network constraints," *IEEE Trans. Comp.*, **C-18**, (12), 1098, (1979).

Kohavi, Z., *Switching and Finite Automata Theory*, 2/e, McGraw-Hill, New York, (1978).

Shannon, C. E., "A Symbolic Analysis of Relay and Switching Circuits," *Trans. AIEE*, **57**, pp. 713-723, (1938).

Shannon, C. E., "The Synthesis of Two-Terminal Switching Circuits," *Bell System Technical Journal*, **28**, pp. 59-98, (1949).

Tanenbaum, A., *Structured Computer Organization*, 3/e, Prentice Hall, Englewood Cliffs, New Jersey, (1990).

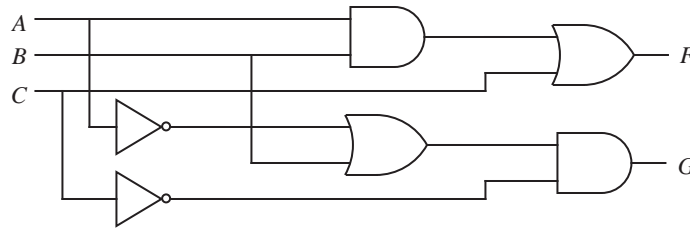
Xilinx, *The Programmable Gate Array Data Book*, Xilinx, Inc., 2100 Logic Drive, San Jose, California, (1992).

## ■ PROBLEMS

**A.1** Figure A-13 shows an OR gate implemented with a NAND gate and inverters, and Figure A-14 shows inverters implemented with NAND gates. Show the logic diagram for an AND gate implemented entirely with NAND gates.

**A.2** Draw logic diagrams for each member of the computationally complete set {AND, OR, NOT} using only the computationally complete set {NOR}.

**A.3** Given the logic circuit shown below, construct a truth table that describes its behavior.



**A.4** Construct a truth table for a three-input XOR gate.

**A.5** Compute the gate input count of the 4-to-2 priority encoder shown in Figure A-32. Include the inverters in your count.

**A.6** Design a circuit that implements function  $f$  below using AND, OR, and NOT gates.  $f(A, B, C) = \bar{A}BC + \bar{A}\bar{B}\bar{C} + ABC$

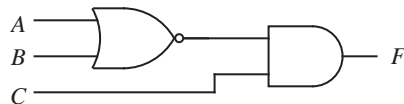
**A.7** Design a circuit that implements function  $g$  below using AND, OR, and NOT gates. Do not attempt to change the form of the equation.

$$g(A, B, C, D, E) = A(BC + \bar{B}\bar{C}) + B(CD + E)$$

**A.8** Are functions  $f$  and  $g$  shown below equivalent? Show how you arrive at your answer.

$$f(A, B, C) = ABC + \bar{A}\bar{B}\bar{C} \qquad g(A, B, C) = (A \oplus C)B$$

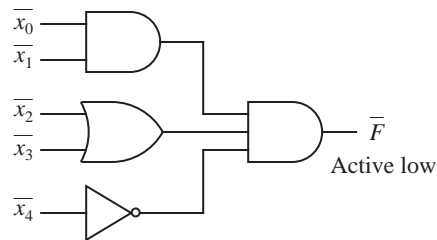
**A.9** Write a Boolean equation that describes function  $F$  in the circuit shown below. Put your answer in SOP form (without parentheses).



**A.10** A four-bit **comparator** is a component that takes two four-bit words as inputs and produces a single bit of output. The output is a 0 if the words are identical, and is a 1 otherwise. Design a four-bit comparator with any of the

logic gates you have seen in this appendix. Hint: Think of the four-bit comparator as four one-bit comparators combined in some fashion.

- A.11** Redraw the circuit shown below so that the bubble matching is correct. The overbars on the variable and function names indicate active low logic.



- A.12** Use two 4-to-1 MUXes to implement the functions:

$A$	$B$	$F_0$	$F_1$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- A.13** Use one 4-to-1 MUX to implement the majority function.

- A.14** Use a 2-to-4 decoder and an OR gate to implement the XOR of two inputs  $A$  and  $B$ .

- A.15** Draw a logic diagram that uses a decoder and two OR gates to implement functions  $F$  and  $G$  below. Be sure to label all lines in your diagram.

$$F(A, B, C) = \bar{A}B\bar{C} + \bar{A}\bar{B}C + A\bar{B}C + \bar{A}BC$$

$$G(A, B, C) = \bar{A}B\bar{C} + ABC$$

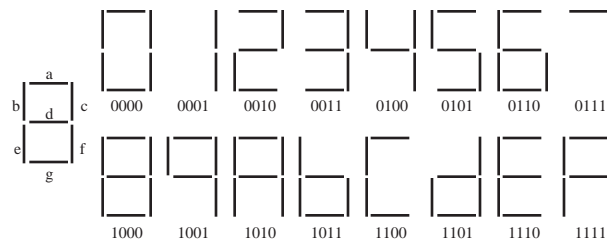
- A.16** Design a circuit using only 2-to-1 multiplexers that implements the function of an 8-to-1 multiplexer. Show your design in the form of a logic diagram, and label all of the lines.

- A.17** Since any combinational circuit can be constructed using only two-input



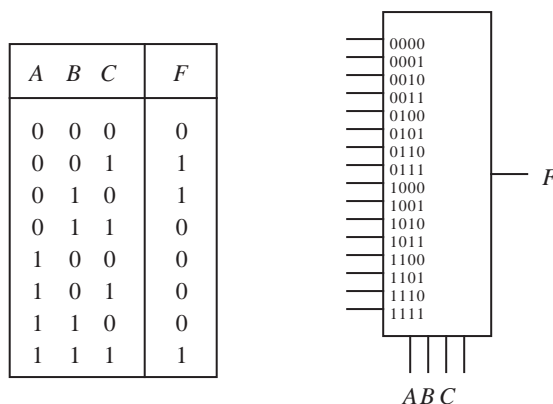
NAND gates, the two-input NAND is called a universal logic gate. The two-input NOR is also a universal logic gate; however, AND and OR are not. Since a two-input NAND can be constructed using only 4-to-1 MUXes (it can be done with one 4-to-1 MUX), any combinational circuit can be constructed using only 4-to-1 MUXes. Consequently, the 4-to-1 MUX is also a universal device. Show that the 1-to-2 DEMUX is a universal device by constructing a two-input NAND using only 1-to-2 DEMUXes. Draw a logic diagram. Hint: Compose the NAND from an AND and an inverter each made from 1-to-2 DEMUXes.

**A.18** A seven segment display, like you might find in a calculator, is shown below. The seven segments are labeled a through g. Design a circuit that takes as input a four-bit binary number and produces as output the control signal for just the b segment (not the letter 'b', which has the 1011 code). A 0 at the output turns the segment off, and a 1 turns the segment on. Show the truth table and an implementation using a single MUX, and no other logic components. Label all of the lines of the MUX.



**A.19** Implement function  $F$  shown in the truth table below using the 16-to-1

MUX shown. Label all of the lines, including the unmarked control line.



**A.20** A **strict encoder** takes  $2^N$  binary inputs, of which exactly one input is 1 at any time and the remaining inputs are 0, and produces an  $N$ -bit coded binary output that indicates which of the  $N$  inputs is high. For this problem, create a truth table for a 4-to-2 strict encoder in which there are four inputs:  $A$ ,  $B$ ,  $C$ , and  $D$ , and two outputs:  $X$  and  $Y$ .  $A$  and  $X$  are the most significant bits.

**A.21** Consider a combinational logic circuit with three inputs  $a$ ,  $b$ , and  $c$ , and six outputs  $u$ ,  $v$ ,  $w$ ,  $x$ ,  $y$ , and  $z$ . The input is an unsigned number between 0 and 7, and the output is the square of the input. The most significant bit of the input is  $a$ , and the most significant bit of the output is  $u$ . Create a truth table for the six functions.

**A.22** Consider the function  $f(a, b, c, d)$  that takes on the value 1 if and only if the number of 1's in  $b$  and  $c$  is greater than or equal to the number of 1's in  $a$  and  $d$ .

(a) Write the truth table for function  $f$

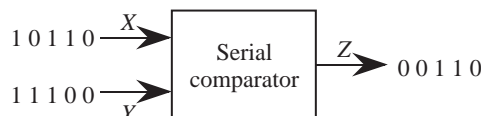
(b) Use an 8-to-1 multiplexer to implement function  $f$

**A.23** Create a truth table for a single digit ternary (base 3) comparator. The ternary inputs are  $A$  and  $B$  which are each a single ternary digit wide. The output  $Z$  is 0 for  $A < B$ , 1 for  $A = B$ , and 2 for  $A > B$ . Using this truth table as a guide, rewrite the truth table in binary using the assignment  $(0)_3 \rightarrow (00)_2$ ,  $(1)_3 \rightarrow$

$(01)_2$ , and  $(2)_3 \rightarrow (10)_2$ .

- A.24** Prove the consensus theorem for three variables using perfect induction.
- A.25** Use the properties of Boolean algebra to prove DeMorgan's theorem algebraically.
- A.26** Can an S-R flip-flop be constructed with two cross-coupled XOR gates? Explain your answer.
- A.27** Modify the state transition diagram in the Vending Machine example to provide more realistic behavior (that is, it returns *all* excess money) when a quarter is inserted in state D.
- A.28** Create a state transition diagram for an FSM that sorts two binary words  $A$  and  $B$ , most significant bit first, onto two binary outputs  $GE$  and  $LT$ . If  $A$  is greater than or equal to  $B$ , then  $A$  appears on the  $GE$  line and  $B$  appears on the  $LT$  line. If  $B$  is greater than  $A$ , then  $B$  appears on the  $GE$  line and  $A$  appears on the  $LT$  line.
- A.29** Design a circuit that produces a 1 at the  $Z$  output when the input  $X$  changes from 0 to 1 or from 1 to 0, and produces a zero at all other times. For the initial state, assume a 0 was last seen at the input. For example, if the input sequence is 00110 (from left to right), then the output sequence is 00101. Show the state transition diagram, the state table, state assignment, and the final circuit using MUXes.
- A.30** Design an FSM that outputs a 1 when the last three inputs are 011 or 110. Just show the state table. Do not draw a circuit.
- A.31** Design a finite state machine that takes two binary words  $X$  and  $Y$  in serial form, least significant bit (LSB) first, and produces a 1-bit output  $Z$  that is true when  $X > Y$  and is 0 for  $X \leq Y$ . When the machine starts, assume that  $X = Y$ . That is,  $Z$  produces 0's until  $X > Y$ . A sample input sequence and the corre-

sponding output sequence are shown below.



**A.32** Create a state transition diagram for an FSM that sorts two ternary inputs, most significant digit first, onto two ternary outputs *GE* and *LT*. If *A* is greater than or equal to *B*, then *A* appears on the *GE* line and *B* appears on the *LT* line, otherwise *B* appears on the *GE* line and *A* appears on the *LT* line. A sample input/output sequence is shown below. Use the ternary symbols 0, 1, and 2 when you label the arcs.

Input A:	0 2 1 1 2 0 1 2
Input B:	0 2 1 2 0 2 1 1
Output GE:	0 2 1 2 0 2 1 1
Output LT:	0 2 1 1 2 0 1 2
Time:	0 1 2 3 4 5 6 7

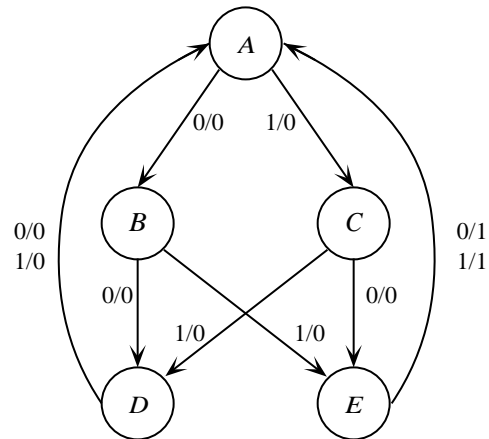
**A.33** Create a state transition diagram for a machine that computes an even parity bit *z* for its two-bit input  $x_1x_0$ . The machine outputs a 0 when all of the previous two-bit inputs collectively have an even number of 1's, and outputs a 1 otherwise. For the initial state, assume that the machine starts with even parity.

**A.34** Given the state transition diagram shown below,

(a) Create a state table.

(b) Design a circuit for the state machine described by your state table using D flip-flop(s), a single decoder, and OR gates. For the state assignment, use the bit pattern that corresponds to the position of each letter in the alphabet, starting from 0. For example, *A* is at position 0, so the state assignment is 000;

$B$  is at position 1, so the state assignment is 001, and so on.



**A.35** Redraw the circuit shown in Figure A-16 using AND and OR gates that have fan-in = 2.

**A.36** Suppose that you need to implement an  $N$ -input AND gate using only three-input AND gates. What is the minimum number of gate delays required to implement the  $N$ -input AND gate? A single AND gate has a gate delay of 1; two cascaded AND gates have a combined gate delay of 2, *etc.*

