PART ONE

BACKGROUND AND MOTIVATION

"Architecture is the learned game, correct and magnificent, of forms assembled in the light."
—Le Corbusier

"I think that the future computer architect is a systems architect, not simply a processor architect; so one must bring together software technology, systems applications, arithmetic, all in a complex system which has a statistical behavior that is not immediately or simply analyzed. . . ."
—Michael J. Flynn, looking forward, circa 1998

TOPICS IN THIS PART

1. Combinational Digital Circuits
2. Digital Circuits with Memory
3. Computer System Technology
4. Computer Performance

Computer architecture encompasses a set of core ideas that are applicable to the design or understanding of virtually any digital computer, from the tiniest embedded systems to the largest supercomputers. Computer architecture isn't just for computer designers; even simple users benefit from a firm grasp of the core ideas and an awareness of the more advanced concepts in this field. Certain key realizations, such as the fact that a $2x$ GHz processor is not necessarily twice as fast as an x GHz model, require a basic training in computer architecture.

We begin this part by reviewing hardware components used in the design of digital circuits and subsystems. Combinational elements, including gates, multiplexers, demultiplexers, decoders, and encoders, are covered in Chapter 1, while sequential circuits, exemplified by register files and counters, constitute the topic of Chapter 2. In Chapter 3, we present an overview of developments in computer technology, and its current state. This is followed by a discussion of absolute and relative performance of computer systems in Chapter 4, perhaps the single most important chapter, setting the stage for performance enhancement methods that are presented throughout the rest of the book.

CHAPTER 1

COMBINATIONAL DIGITAL CIRCUITS

"We used to think that if we know one, we knew two, because one and one are two. We are finding that we must learn a great deal more about 'and'."
—*Sir Arthur Eddington*

"This product contains minute electrically charged particles moving at velocities in excess of 500 million miles per hour. Handle with extreme care."
—*Proposed truth-in-product-labeling warning to be put on all digital systems (source unknown)*

TOPICS IN THIS CHAPTER

1.1 Signals, Logic Operators, and Gates
1.2 Boolean Functions and Expressions
1.3 Designing Gate Networks
1.4 Useful Combinational Parts
1.5 Programmable Combinational Parts
1.6 Timing and Circuit Considerations

Familiarity with digital design is required for studying computer architecture and is assumed of the reader of this book. The capsule review presented in this and the following chapter is intended to refresh the reader's memory and to provide a basis for understanding the terminology and designs in the rest of the book. In this chapter, we review some of the key concepts of combinational (memoryless) digital circuits and introduce a number of very useful components that are found in many diagrams in this book. Examples include tristate buffers (regular or inverting), multiplexers, decoders, and encoders. This review is continued in Chapter 2, which deals with sequential digital circuits (with memory). Readers who have trouble understanding the material in these two chapters should consult any of the logic design textbooks listed at the end of the chapter.

1.1 Signals, Logic Operators, and Gates

All information elements in digital computers, including instructions, numbers, and symbols, are encoded as electronic signals that are almost always *two-valued*. Even though *multivalued signals* and associated logic circuits are feasible and occasionally used, modern digital

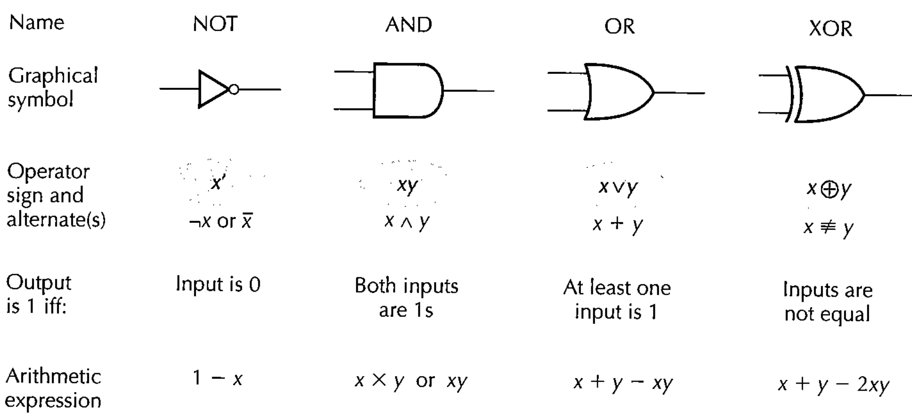
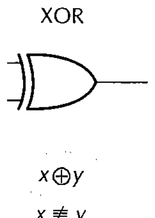


Figure 1.1 Some basic elements of digital logic circuits, with operator signs used in this book highlighted.

computers are predominantly binary. *Binary signals* can be represented by the presence or absence of some electrical property such as voltage, current, field, or charge. We refer to the two values of a binary signal as "0" and "1." These values can represent the digits of a radix-2 number in the natural way or be used to denote states (off/on), conditions (false/true), options (path A/path B), and the like. The assignment of 0 and 1 to binary states or conditions is arbitrary, but having 0 represent "off" or "false" and 1 correspond to "on" or "true" is more common. When binary signals are represented by high/low voltage, assigning high voltage to 1 leads to *positive logic* and the opposite is considered to be *negative logic*.

Logic operators are abstractions for specifying transformations of binary signals. There are $2^2 = 4$ possible single-input operators, because the truth table of such an operator has two entries (corresponding to the input being 0 or 1) and each entry can be filled with 0 or 1. A two-input operator with binary inputs can be defined in $2^4 = 16$ different ways, depending on whether it produces a 0 or 1 output for each of the four possible combinations of input values. Figure 1.1 depicts the single-input operator known as NOT (*complementer* or *inverter*) and three of the most commonly used two-input operators: AND, OR, and XOR (exclusive OR). For each of these operators, the sign used in logical expressions, and alternate form favored in books on logic design, are given. The operator signs used in this books are highlighted in Figure 1.1. In particular, we use "$\vee$" instead of the more common "+" for OR because we also talk a great deal about addition and in fact on occasion addition and OR are used in the same paragraph or diagram. For AND, on the other hand, simply juxtaposing the operands does not give rise to any problem because AND is identical to multiplication for binary signals.

Figure 1.1 also relates logic operators to arithmetic operators. For example, complementing or inverting a signal x yields $1 - x$. Because both AND and OR are *associative,* meaning that $(xy)z = x(yz)$ and $(x \vee y) \vee z = x \vee (y \vee z)$, these operators can be defined with more than two inputs, without causing any ambiguity about their outputs. Also, given that the graphical symbol for NOT consists of a triangle that represents the identity operation (or no operation at all) and a small "bubble" that signifies inversion, logic diagrams can be made simpler and less cluttered by allowing inversion bubbles on inputs or outputs of logic gates. For example, an AND gate and an inverter connected to its output can be merged into a single NAND gate, drawn as an AND gate with a bubble placed on its output line. Similarly, NOR



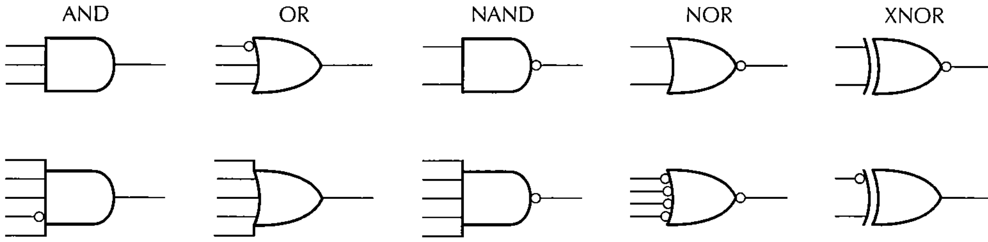


Figure 1.2 Gates with more than two inputs and/or with inverted signals at input or output.

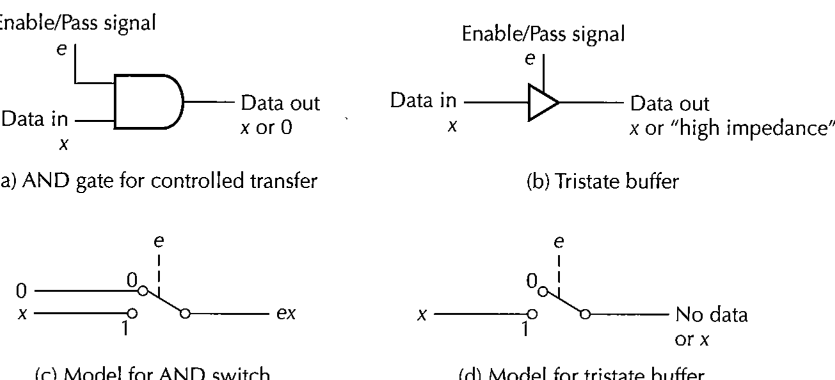
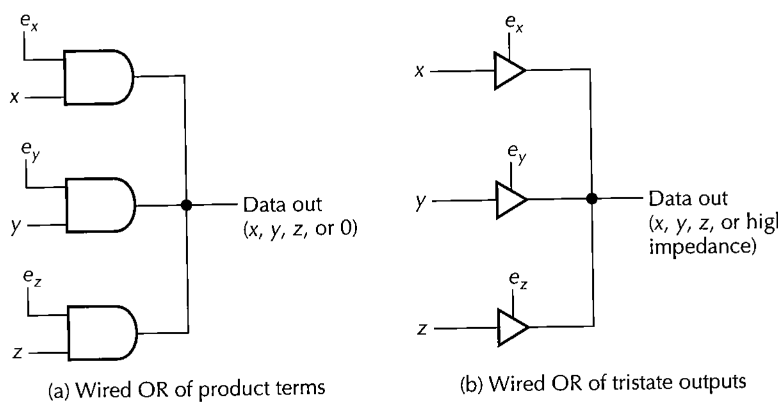


Figure 1.3 An AND gate and a tristate buffer can act as controlled switches or valves. An inverting buffer is logically the same as a NOT gate.

and XNOR gates can be defined (Figure 1.2). Bubbles can also be placed on a gate's inputs, leading to the graphical representation of operations such as $x' \vee y \vee z$ with one gate symbol.

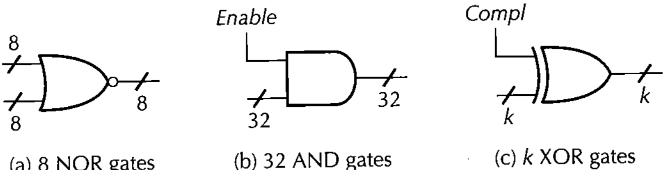
Much as variables in a program are named, the name of a logic signal must be chosen with care to convey useful information about the signal's role. Names that are very short or very long must be avoided if possible. A control signal whose value is 1 is referred to as "asserted," while a 0 signal is deasserted. Asserting a control signal is a common way of causing an action or event to occur. If signal names are chosen carefully, a signal named "*sub*" will likely cause a subtraction operation to be performed when asserted, while a 3-bit signal bundle "*oper*" may encode one of eight possible operations to be performed by some unit. When it is the deassertion of a signal that triggers an event, the signal name should appear in complemented form for clarity; for example, the signal *add'*, when deasserted, may cause addition to be performed. It is also possible to apply a name such as *add'sub* to a signal that causes two different actions depending on its value.

If one input of a two-input AND gate is viewed as a control signal and the other as a data signal, one can say that assertion of the control signal allows the data signal to propagate to the output, whereas deassertion of the control signal forces the output to 0, independently of the input data (Figure 1.3). Thus, an AND gate can act as a switch or data valve that is controlled by an *enable* or *pass* signal. An alternate mechanism for this purpose, also shown in Figure 1.3, is a *tristate buffer* whose output is equal to the data input x when the control signal e is asserted and assumes an indeterminate value (high impedance in electrical terms) when e is deasserted. A tristate buffer effectively isolates the output from the input wheneve



(a) Wired OR of product terms (b) Wired OR of tristate outputs

Figure 1.4 Wired OR allows tying together of several controlled signals.



(a) 8 NOR gates (b) 32 AND gates (c) k XOR gates

Figure 1.5 Arrays of logic gates represented by a single gate symbol.

the control signal is deasserted. An XOR gate with one control and one data signal can be viewed as a *controlled inverter* that inverts the data if its control is asserted and lets it through unchanged, otherwise.

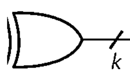
Outputs of several AND switches, tristate buffers, or inverting buffers can be connected to each other for an implicit or wired OR function. In fact, a primary application of tristate buffers is to connect a possibly large number of data sources (such as memory cells) to a common data line through which data travels to a receiver. In Figure 1.4, when only one of the enable signals is asserted, the corresponding data passes through and prevails at the output side. When no enable signal is asserted, then the output will be 0 (for AND gates) or high impedance (for tristate buffers). When more than one enable signal is asserted, the logical OR of the associated data inputs prevails at the output side, although this situation is often avoided.

We frequently use an array of identical gates to combine bundles of signals. In depicting such an arrangement, we draw just one gate and indicate, by using a tick mark and an integer next to it, how many signals or gates are involved. For example, Figure 1.5a shows bitwise NOR operation performed on two 8-bit bundles. If the input bundles are x and y and the output bundle z, then this is equivalent to setting $z_i = (x_i \vee y_i)'$ for each i. Similarly, we can have an array of 32 AND switches, all tied to the same *Enable* signal, to control the flow of a 32-bit data word from the input side to the output side (Figure 1.5b). As a final example, an array of k XOR gates can be used to invert all bits in a k-bit bundle whenever *Compl* is asserted (Figure 1.5c).

1.2 Boolean Functions and Expressions

A signal that can be either 0 or 1 is a *Boolean variable*. An n-variable *Boolean function* depends on n Boolean variables and produces a result in $\{0, 1\}$. Boolean functions are of interest to us because a network of logic gates with n inputs and one output implements an n-variable Boolean function. There are various ways for specifying Boolean functions.

a. A *truth table* is a listing of the function results for all combinations of input values. The truth table for an n-variable Boolean function has n input columns, an output column, and 2^n rows. A truth table with m output columns might be used to specify m Boolean functions of the same variables at once (see, e.g., Table 1.1). A *don't-care* entry "x" in an output column means that the function result is of no interest in that row, perhaps because that combination of input values is not expected to ever arise. An "x" in an input column means that the function result does not depend on the value of the particular variable involved.



b. A *logic expression* is made of Boolean variables, logic operators, and parentheses. In the absence of parentheses, NOT takes precedence over AND, which takes precedence over OR/XOR. For a given assignment of values to variables, a logic expression can be evaluated to yield a Boolean result. Logic expressions can be manipulated using laws of Boolean algebra (Table 1.2). Usually, the goal of this process is to obtain an *equivalent* logic expression that is in some way simpler or more suitable for hardware realization.

TABLE 1.1 Three 7-variable Boolean functions specified in a compact truth table with don't-care entries in both input and output columns.

	Seven inputs							Three outputs		
Line #	s_{lever}	c_{25}	c_{10}	a_{gum}	a_{bar}	p_{gum}	p_{bar}	r_{coins}	r_{gum}	r_{bar}
1	0	x	x	x	x	x	x	0	0	0
2	1	0	0	x	x	x	x	x	0	0
3	1	0	1	x	x	x	x	1	0	0
4	1	1	0	x	x	x	x	1	0	0
5	1	1	1	x	x	0	0	1	0	0
6	1	1	1	x	x	1	1	1	0	0
7	1	1	1	x	0	0	1	1	0	x
8	1	1	1	x	1	0	1	0	0	1
9	1	1	1	0	x	1	0	1	x	0
10	1	1	1	1	x	1	0	0	1	0

TABLE 1.2 Laws (basic identities) of Boolean algebra.

Name of law	OR version	AND version
Identity	$x \vee 0 = x$	$x\,1 = x$
One/Zero	$x \vee 1 = 1$	$x\,0 = 0$
Idempotent	$x \vee x = x$	$xx = x$
Inverse	$x \vee x' = 1$	$xx' = 0$
Commutative	$x \vee y = y \vee x$	$xy = yx$
Associative	$(x \vee y) \vee z = x \vee (y \vee z)$	$(xy)z = x(yz)$
Distributive	$x \vee (yz) = (x \vee y)(x \vee z)$	$x(y \vee z) = (xy) \vee (xz)$
DeMorgan's	$(x \vee y)' = x'y'$	$(xy)' = x' \vee y'$

A logic expression formed by ORing several AND terms is in *(logical-)sum-of-products* form, for example, $xy \vee yz \vee zx$ or $w \vee x'yz$. Similarly, ANDing of several OR terms leads to a *product-of-(logical-)sums* expression, for example, $(x \vee y)(y \vee z)(z \vee x)$ or $w'(x \vee y \vee z)$.

c. A *word statement* can describe a simple logic function of a few Boolean variables. For example, a statement such as "The alarm will sound if the door is opened while the security system is engaged or when the smoke detector is triggered" corresponds to the Boolean function $e_{\text{alarm}} = (s_{\text{door}} s_{\text{security}}) \vee d_{\text{smoke}}$, which relates an enable signal to a pair of status signals and a detector signal.

d. A *logic diagram* is a graphical representation of a Boolean function that also carries information about its hardware realization. Deriving a logic diagram from any of the specification types just named is the *logic circuit synthesis* process. Going backward from a logic diagram to another form of specification is known as *logic circuit analysis*. In addition to gates and other elementary components, a logic diagram may include boxes of various shapes that represent standard building blocks or previously designed subcircuits.

We often use a combination of the preceding four methods, in a hierarchical scheme, to represent computer hardware. For example, a high-level logic diagram, composed of subcircuits and standard blocks, may provide the big picture. Each of the nonstandard elements, which is not simple enough to be described by a truth table, logic expression, or word statement, may in turn be specified through another diagram, and so on.

Example 1.1: Proving equivalence of logic expressions Prove that the following pairs of logic expressions are equivalent.

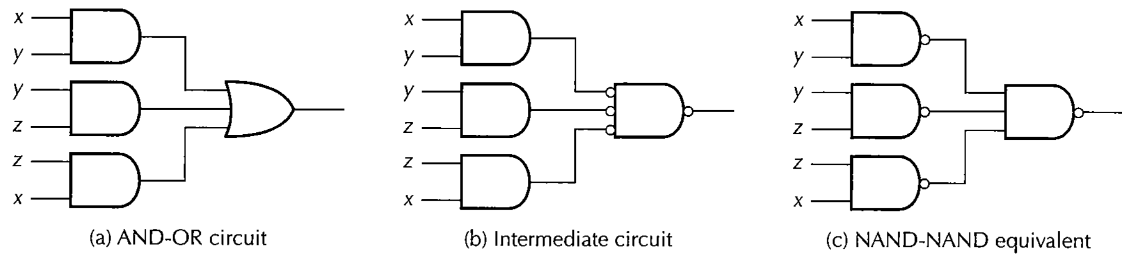
a. Distributive law, AND version: $x(y \vee z) \equiv (xy) \vee (xz)$
b. DeMorgan's law, OR version: $(x \vee y)' \equiv x'y'$
c. $xy \vee x'z \vee yz \equiv xy \vee x'z$
d. $xy \vee yz \vee zx \equiv (x \vee y)(y \vee z)(z \vee x)$

Solution: We prove each part by a different method to illustrate the range of possibilities.

a. Use the truth table method: form an 8-row truth table corresponding to all possible combinations of values for the three variables x, y, and z. Observe that the two expressions lead to the same value in each row. For example, $1(0 \vee 1) = (1\ 0) \vee (1\ 1) = 1$.
b. Use the arithmetic substitutions shown in Figure 1.1 to convert this logic equality problem into the easily proven algebraic equality $1 - (x + y - xy) = (1 - x)(1 - y)$.
c. Use case analysis: for example, derive simplified forms of the equality for $x = 0$ (prove $z \vee yz = z$) and $x = 1$ (prove $y \vee yz = y$). You may have to divide a more complex problem further.
d. Use logic manipulation to convert one expression into the other: $(x \vee y)(y \vee z)(z \vee x) = (xy \vee xz \vee yy \vee yz)(z \vee x) = (xz \vee y)(z \vee x) = xzz \vee xzx \vee yz \vee yx = xz \vee yz \vee yx$.

1.3 Designing Gate Networks

Any logic expression composed of NOT, AND, OR, XOR, and other types of gates is a specification for a gate network. For example, the logic expression $xy \vee yz \vee zx$ specifies the gate network of Figure 1.6a. This is a two-level AND-OR logic circuit with AND gates in level 1



(a) AND-OR circuit (b) Intermediate circuit (c) NAND-NAND equivalent

Figure 1.6 A two-level AND-OR circuit and two equivalent circuits.

and an OR gate in level 2. Because according to DeMorgan's law (Table 1.2, last row, middle column), an OR gate can be replaced by a NAND gates with complemented inputs, Figure 1.6b is readily seen to be equivalent to Figure 1.6a. Now, by moving the inversion bubbles at the inputs of the level-2 NAND gate in Figure 1.6b to the outputs of the AND gates in level 1, we derive the two-level NAND-NAND circuit of Figure 1.6c that realizes the same function. A similar process converts any two-level OR-AND circuit to an equivalent NOR-NOR circuit. In both cases, any signal that is input directly to a level-2 gate must be inverted (because the bubble remains).

Whereas the process of converting a logic expression to a logic diagram, and thus an associated hardware realization, is trivial, obtaining a logic expression that leads to the best possible hardware circuit is not. For one thing, the definition of "best" changes depending on the technology and implementation scheme being used (e.g., custom VLSI, programmable logic, discrete gates) and on the design goals (e.g., high speed, power economy, low cost). For another, the simplification process, if not done via automatic design tools, is not only cumbersome but also imperfect; for example, it might be based on minimizing the number of gates employed, without taking into account the speed and cost implications of wires that connect the gates together. In this book, we do not concern ourselves with the simplification process for logic expressions. This is because every logic function that we will encounter, when suitably divided into parts, is simple enough to allow the required parts to be realized by means of efficient logic circuits in a straightforward manner. We illustrate the process through two examples.

Example 1.2: BCD-to-seven-segment decoder Figure 1.7 shows how the decimal digits 0–9 might appear on a seven-segment display device. Design logic circuits to generate the enable signals that cause the segments to be lit or darkened, given a 4-bit binary representation of the decimal digit (binary-coded decimal or BCD code) to be displayed as input.

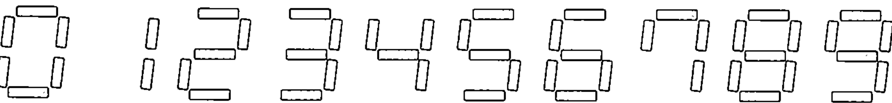


Figure 1.7 Seven-segment display of decimal digits. The three open segments may be optionally used. The digit 1 can be displayed in two ways, with the more common right-side version shown.

Solution: Figure 1.7 is a graphical representation of rows 0-9 of a 16-row truth table, where each of the rows 10–15 constitutes a don't-care condition. There are four input columns

x_3, x_2, x_1, x_0 and seven output columns e_0–e_6. Figure 1.8 shows the numbering of segments and the logic circuit that produces the enable signal for segment number 3. The truth table output column associated with e_3 contains the entries 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, x, x, x, x, x, x (9 displayed without segment 3). This is easily translated to the logic expression $e_3 = x_1x_0' \vee x_2'x_0' \vee x_2'x_1 \vee x_2x_1'x_0$. Note that e_3 is independent of x_3. Deriving the logic circuits for the remaining six segments is done similarly.

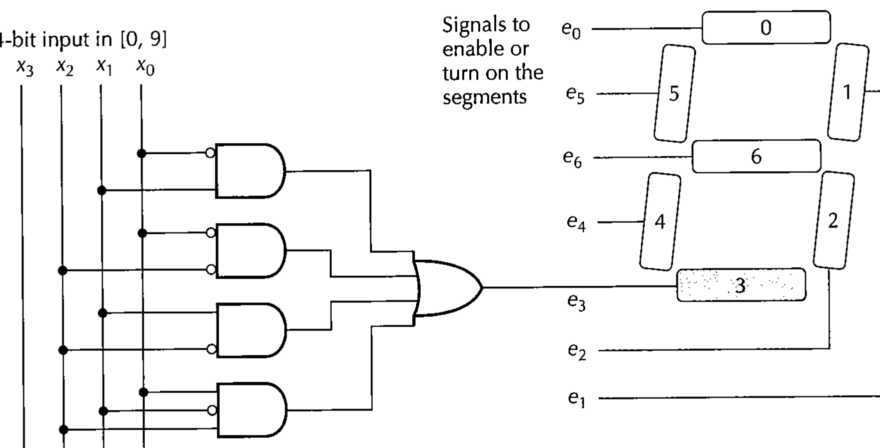
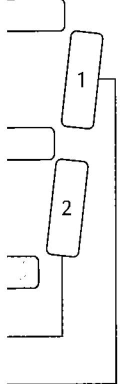


Figure 1.8 The logic circuit that generates the enable signal for the lowermost segment (number 3) in a seven-segment display unit.

Example 1.3: Simple vending machine actuator A small vending machine can dispense a pack of gum or a candy bar, each costing 35 cents. The customer must deposit exact change consisting of a quarter and a dime, indicate preference for one of the two items by pressing the corresponding pushbutton, and pull a lever to release the desired item into a bin. The actuator is a combinational circuit with three outputs: $r_{gum}(r_{bar})$, when asserted, causes a pack of gum (candy bar) to be released; assertion of r_{coins} causes the deposited coins to be returned when, for any reason, a sale cannot be completed. Inputs to the actuator are the following signals:

a. s_{lever} indicating the state of the lever (1 means that the lever has been pulled)
b. c_{25} and c_{10}, for quarter and dime, supplied by a coin detection module
c. a_{gum} and a_{bar} supplied by devices that sense the availability of the two items
d. p_{gum} and p_{bar} coming from two pushbuttons holding the customer's preference

Solution: Refer again to Table 1.1, which is the truth table for the vending machine actuator. When the lever has not been pulled, all outputs must be 0, regardless of the values of other inputs (line 1). The rest of the cases that follow correspond to $s_{lever} = 1$. When no coin has been deposited, neither item should be released; the value of r_{coins} is immaterial in this case, since there is no coin to be returned (line 2). When only one coin has been deposited, no item should be released and the coin must be returned to the customer (lines 3–4). The rest of the cases correspond to 35 cents having been deposited and the lever pulled. If the customer has made no



selection, or has selected both items, the coins must be returned and no item released (lines 5–6). If a candy bar has been selected, a candy bar is released or the coins are returned, depending on a_{bar} (lines 7–8). The case for the selection of a pack of gum is similar (lines 9–10). The following logic expressions for the three outputs are readily obtained by inspection: $r_{\text{gum}} = s_{\text{lever}} c_{25} c_{10} p_{\text{gum}}$, $r_{\text{bar}} = s_{\text{lever}} c_{25} c_{10} p_{\text{bar}}$, $r_{\text{coins}} = s_{\text{lever}} (c'_{25} \vee c'_{10} \vee p'_{\text{gum}} p'_{\text{bar}} \vee p_{\text{gum}} p_{\text{bar}} \vee a'_{\text{gum}} p_{\text{gum}} \vee a'_{\text{bar}} p_{\text{bar}})$.

1.4 Useful Combinational Parts

Certain combinational parts can be used in the synthesis of digital circuits, much as one utilizes prefabricated closets or bathroom fixtures in constructing a house. Such standard building blocks are numerous and include several arithmetic circuits to be discussed in Part III of the book. In this section, we review the design of three types of combinational components used primarily for control purposes: multiplexers, decoders, and encoders.

A 2^a-to-1 multiplexer, mux for short, has 2^a data inputs $x_0, x_1, x_2, \ldots$, a single output z, and a selection or address signals $y_{a-1}, \ldots, y_1, y_0$. The output z is equal to the input x_i whose index i has the binary representation $(y_{a-1} \cdots y_1 y_0)_{\text{two}}$. Examples include 2-to-1 (two-way) and 4-to-1 (four-way) multiplexers, depicted in Figure 1.9, which have one and two address inputs, respectively. Like arrays of gates, several muxes controlled by the same address lines can be used to select one bundle of signals over another (Figure 1.9d). An n-to-1 mux, where n is not a power of 2, can be built by simply pruning the unneeded parts of a larger mux with 2^a inputs, where $2^{a-1} < n < 2^a$. For example, the design in Figure 1.9f can be converted to a 3-input mux by simply removing the mux with inputs x_2 and x_3, and then connecting x_2 directly to the second-level mux. At any given time, the output z of a mux is equal to one of its

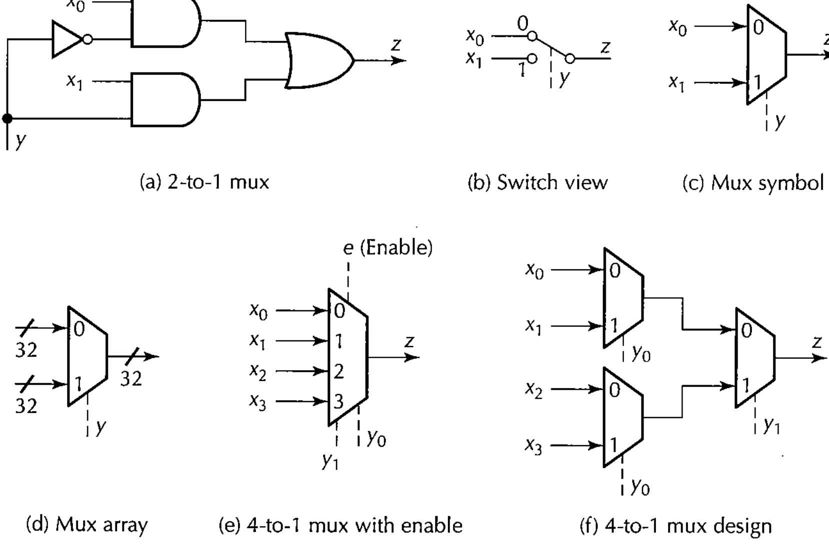


Figure 1.9 A multiplexer (mux), or selector, allows one of several inputs to be selected and routed to output depending on the binary value of a set of selection or address signals provided to it.

inputs. By providing a multiplexer with an enable signal e, which is supplied as an extra input to each of the AND gates in Figure 1.9a, we get the option of forcing the output to 0 independently of data and address inputs. This is essentially equivalent to none of the inputs being selected (Figure 1.9e).

Multiplexers are versatile building blocks. Any a-variable Boolean function can be implemented by means of a 2^a-to-1 mux, where the variables are connected to the address inputs and each of the 2^a data inputs carries a constant value 0 or 1 according to the function's truth table value for that particular row. In fact, if the complement of one of the variables is available at input, then a smaller 2^{a-1}-to-1 mux suffices. As a concrete example, to implement the function e_3 defined in Example 1.2, one can use an 8-to-1 mux with address inputs connected to x_2, x_1, x_0 and data input carrying 1, 0, 1, 1, 0, 1, 1, 0, from top to bottom. Alternatively, one can use a 4-to-1 mux, with address lines connected to x_1 and x_0 and the data lines carrying x_2', x_2, 1, and x_2', again from top to bottom. The latter four terms are easily derived from the expression for e_3 by successively fixing the value of x_1x_0 at 00, 01, 10, and 11.

An a-to-2^a (a-input) decoder asserts one and only one of its 2^a output lines. The output x_i that is asserted has an index i whose binary representation matches the value on the a address lines. The logic diagram for a 2-to-4 decoder is shown in Figure 1.10a, with its shorthand symbol given in Figure 1.10b. If outputs of such a decoder are used as enable signals for four different elements or units, then the decoder allows us to choose which one of the four unit is enabled at any given time. If we want to have the option of not enabling any of the four units, then a decoder with an enable input (Figure 1.10c), also known as a *demultiplexer* or *demux,* might be used, where the enable input e is supplied as an additional input to each of the four AND gates in Figure 1.10a (more generally 2^a AND gates). The name demultiplexer indicates that this circuit performs the opposite function of a mux: whereas a mux selects one of its inputs and routes it to the output, a demux receives an input e and routes it to a selected output.

The function of an *encoder* is exactly the opposite of a decoder. When one, and only one, input of a 2^a-input (2^a-to-a) encoder is asserted, its a-bit output supplies the index of the

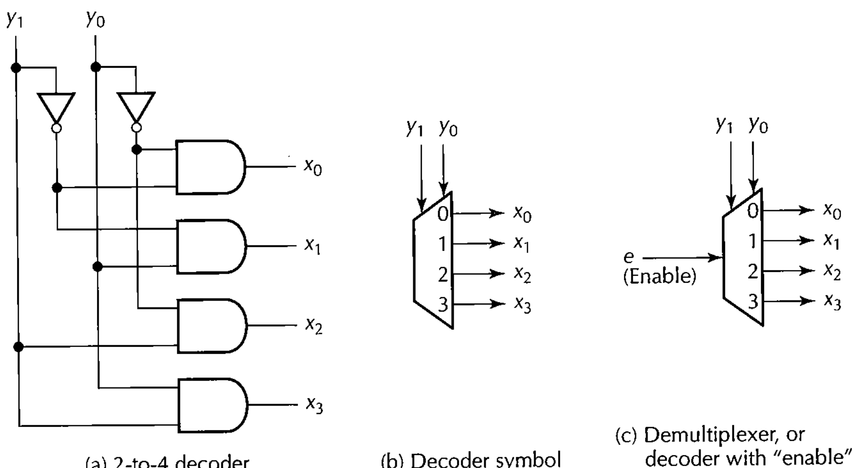
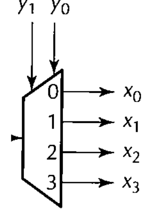
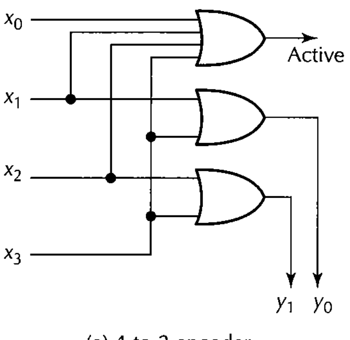
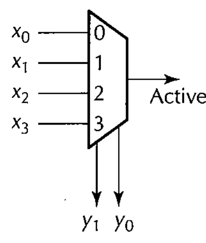


Figure 1.10 A decoder allows the selection of one of 2^a options using an a-bit address as input. A demultiplexer (demux) is a decoder that only selects an output if its enable signal is asserted.





(a) 4-to-2 encoder



(b) Encoder symbol

Figure 1.11 A 2^a-to-a encoder outputs an a-bit binary number equal to the index of the single 1 among its 2^a inputs.

asserted input in the form of a binary number. The logic diagram for a 4-to-2 encoder is shown in Figure 1.11a, with its shorthand symbol given in Figure 1.11b. More generally, the number n of inputs need not be a power of 2. In this case, the $\lceil \log_2 n \rceil$-bit encoder output is the binary representation of the index for the single asserted input, that is, a number between 0 and $n - 1$. If a decoder is designed as a collection of OR gates, as in Figure 1.11a, it produces the all-0s output when no input is asserted or when input 0 is asserted. These two cases are thus indistinguishable at the encoder's output. If we lift the restriction that at most one input of the encoder can be asserted and design the circuit to output the index of the asserted input with the lowest index, a *priority encoder* results. For example, assuming that inputs x_1 and x_2 are asserted, the encoder of Figure 1.11a produces the output 11 (index = 3, which does not correspond to any asserted input), whereas a priority encoder would output 01 (index = 1, the smallest of the indices for asserted inputs). The "Active" signal allows us to differentiate between the cases of none of the inputs being asserted and x_0 being asserted.

Both decoders and encoders are special cases of *code converters*. A decoder converts an a-bit binary code into a 1-out-of-2^a code, a code with 2^a codewords each of which is composed of a single 1 and all other bits set to 0. An encoder converts a 1-out-of-2^a code to a binary code. In Example 1.2, we designed a BCD-to-seven-segment code converter.

1.5 Programmable Combinational Parts

To avoid having to use a large number of small-scale integrated circuits for implementing a Boolean function of several variables, IC manufacturers offer large arrays of gates whose connections can be customized by the process known as programming. With respect to the programming mechanism, there are two types of such circuits. In one type, all connections of potential interest are already made but can be selectively removed. Such connections are made via *fuses* that can be blown open by passing a sufficiently large current through them. In another type of programmable circuit, *antifuse* elements are used to selectively establish connections where desired. In logic diagrams, the same convention is used for both types: a connection that is left in place, or is established, appears as a heavy dot on crossing lines, whereas for a connection that is blown open, or not established, there is no such dot. Figure 1.12a shows how the two functions $w \vee x \vee y$ and $x \vee z$ can be implemented by programmable OR gates. An array of such OR gates, connected to the outputs of an a-to-2^a decoder allows us to implement several functions of a input variables at once (Figure 1.12c). This arrangement is known as programmable read-only memory or PROM.

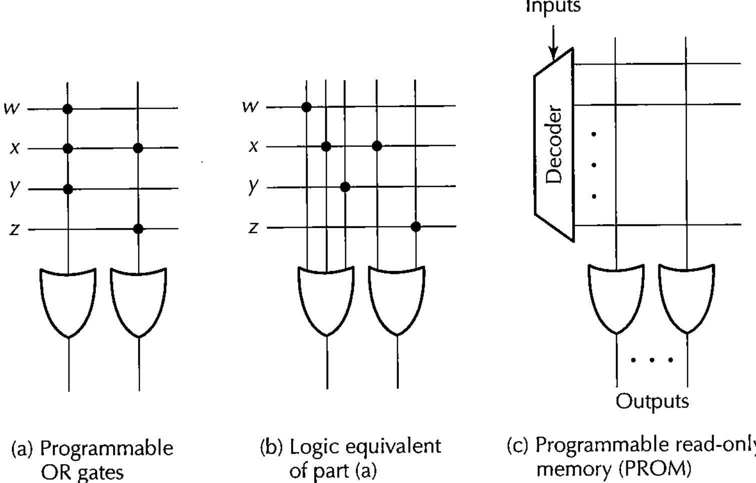


Figure 1.12 Programmable connections and their use in a PROM.

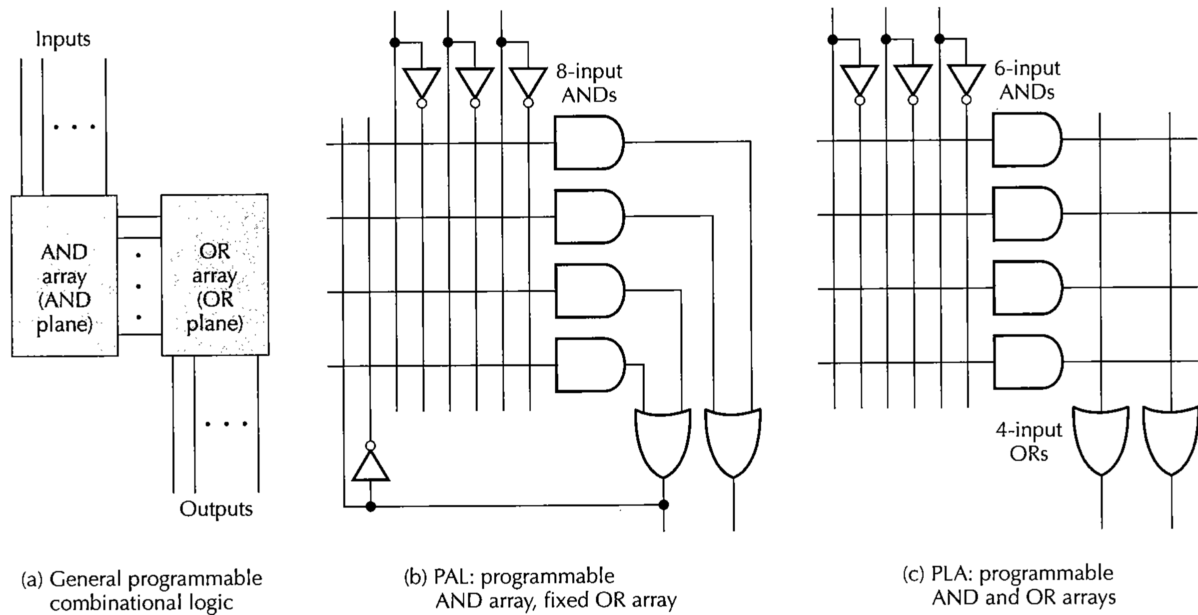
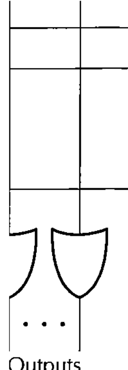


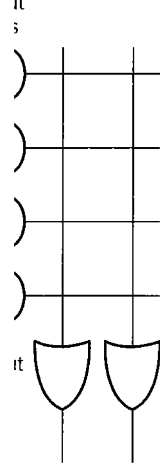
Figure 1.13 Programmable combinational logic: general structure and two classes known as PAL and PLA devices. Not shown is PROM with fixed AND array (a decoder) and programmable OR array.

Figure 1.13a shows a more general structure for programmable combinational logic circuits in which the decoder of Figure 1.12c has been replaced by an array of AND gates. The n inputs, and their complements formed internally, are provided to the AND array, which generates a number of product terms involving input variables and their complements. These product terms are input to an OR array that combines the appropriate product terms for each



nable read-only
(PROM)

OM.



ımable
. arrays

known as PAL
ıable OR array.

ınal logic cir-
) gates. The *n*
y, which gen-
ments. These
erms for each

of up to *m* functions of interest to be output. PROM is a special case of this structure where the AND array is a fixed decoder and the OR array can be arbitrarily programmed. When the OR array has fixed connections but the inputs to the AND gates can be programmed, the result is a programmable array logic or PAL device (Figure 1.13b). When both the AND and OR arrays can be programmed, the resulting circuit is known as programmable logic array or PLA (Figure 1.13c). PAL devices and PLAs are more efficient than PROMs because they generate far fewer product terms. PAL devices are more efficient, but less flexible, than PLAs.

In commercial PAL parts, because of the limited number of product terms that can be combined to form each output, a feedback mechanism is often provided that makes some of the outputs selectable as inputs to AND gates. In Figure 1.13b the left output is fed back to the AND array, where it can be used as an input into the AND gates contributing to the formation of the right output. Alternatively, such fed-back outputs can be used as primary inputs if additional inputs are needed. A commonly used PAL product is the PAL16L8 device. The numbers 16 and 8 in the device's name refer to input and output lines, respectively; the package has 20 pins for 10 inputs, 2 outputs, 6 bidirectional I/O lines, power, and ground. The programmable AND array of this device consists of 64 AND gates each with 32 inputs (all 16 inputs and their complements). The 64 AND gates are divided into eight groups of 8 gates. Within each group, 7 AND gates feed a 7-input OR gate producing one output, and the remaining AND gate generates an enable signal for an inverting tristate buffer.

PLAs are not used as commodity parts but as structures that allow regular and systematic implementation of logic functions on custom VLSI chips. For example, we will see later (in Chapter 13) that the instruction decoding logic of a processor is a natural candidate for PLA implementation.

1.6 Timing and Circuit Considerations

When the input signals to a gate vary, any requisite output change does not occur immediately but rather takes effect with some delay. The gate delay varies with the underlying technology, gate type, number of inputs (gate *fan-in*), supply voltage, operating temperature, and so on. However, as a first-order approximation, all gate delays can be considered equal and denoted by δ. A two-level logic circuit can then be said to have a delay of 2δ. For the CMOS (complementary metal-oxide semiconductor) technology used in the great majority of modern digital circuits, the *gate delay* may be as little as a fraction of a nanosecond. Signal propagation on wires that connect gates also contributes some delay, but again in the context of an approximate analysis, such delays can be ignored to simplify analyses; as circuit dimensions are scaled down, however, such an omission is becoming more and more problematic. The only accurate way for estimating the delay of a logic circuit is to run the complete design, with full details of logic elements and wiring, through a design tool. Even then, safety margins must be included in the timing estimates to account for process irregularities and other variations.

When delays along various paths in a logic circuit are unequal, as they are bound to be owing to the unequal number of gates through which different signals pass and/or the aforementioned variations, a phenomenon known as *glitching* occurs. Suppose we were to implement the function $f = x \vee y \vee z$ using the circuit of Figure 1.13b. Because the OR gates in the target circuit have only two inputs, we must first generate $a = x \vee y$ on the left output and then use the result to form $f = a \vee z$ on the right output. Note that the signals x and y pass through four gate levels, whereas z passes through only two levels. This leads to the situation shown in the timing diagram of Figure 1.14 where $x = 0$ throughout, whereas y changes from

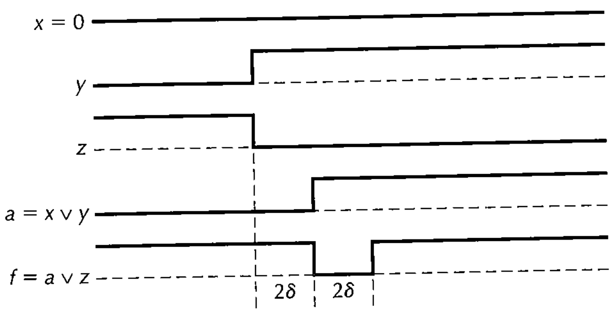
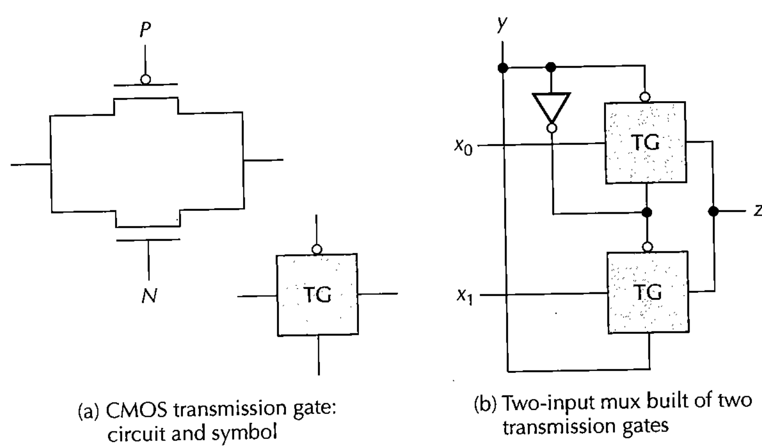


Figure 1.14 Timing diagram for a circuit that exhibits glitching.



(a) CMOS transmission gate: circuit and symbol

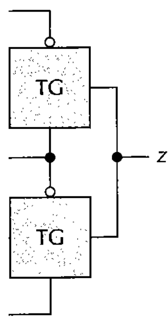
(b) Two-input mux built of two transmission gates

Figure 1.15 A CMOS transmission gate and its use in building a 2-to-1 mux.

0 to 1 at about the same time as z changing from 1 to 0. Theoretically, the output should be 1 at all times. However, because of unequal delays, the output assumes the value 0 for 2δ time units. This is one reason why we need accurate timing analyses and safety margins to ensure that adequate time is allowed for changes to fully propagate through the circuit, and for the outputs to assume their correct final values, before the generated results are used in other computations.

Even though in this book we deal exclusively with combinational logic circuits built of gates, we should mention for completeness that with CMOS technology, other circuit elements can be derived and used that do not directly correspond to a gate or gate network. Two examples are presented in Figure 1.15. The two-transistor circuit depicted in Figure 1.15a, alongside its symbolic representation, is known as a *transmission gate* (TG). It connects its two sides when the N control signal is asserted and disconnects them when P is asserted. If the signals N and P are complementary, the transmission gate behaves like a controlled switch. Two transmission gates and an inverter (another two-transistor CMOS circuit) can be used to form a 2-to-1 mux, as shown in Figure 1.15b. A 2^a-input mux can be built by using 2^a transmission gates and a decoder that converts the selection binary number $(y_{a-1} \cdots y_1 y_0)_{\text{two}}$ into a single asserted signal that feeds one of the transmission gates.

diagram for a
glitching.



nux built of two
ı gates

ng a 2-to-1 mux.

put should be 1
ıe 0 for 2δ time
ırgins to ensure
uit, and for the
ɜ used in other

ɜircuits built of
her circuit ele-
ɜ network. Two
ı Figure 1.15a,
It connects its
asserted. If the
ıtrolled switch.
can be used to
using 2^a trans-
$\cdots y_1 y_0)_{\text{two}}$ into

PROBLEMS

1.1 Universal logic elements

The three logic elements AND, OR, and NOT form a universal set because any logic function can be implemented by means of these elements, requiring nothing else.

a. Show that removing AND or OR from the set leaves the remaining set of two elements universal.
b. Show that the XOR forms a universal set with either AND or OR.
c. Show that the NAND gate is universal.
d. Show that the NOR gate is universal.
e. Show that the 2-to-1 multiplexer is universal.
f. Is there any other two-input element, besides NAND and NOR, that is universal? *Hint:* There are 10 functions of two variables that depend on both variables.

1.2 Vending machine actuator

Extend the vending machine actuator design of Example 1.3 in the following ways:

a. Each of the two items costs 65 cents.
b. There are four items to choose from.
c. Both changes of parts a and b.
d. Modify the design for part a to accept any amount up to one dollar (including a dollar bill) and to return change.

1.3 Realization of Boolean functions

Design a three-input, single-output logic circuit that implements any desired Boolean function of the three inputs based on a set of control inputs. *Hint:* There are 256 different three-variable Boolean functions, so the control input to the circuit (the opcode) must contain at least 8 bits. The truth table of a three-variable Boolean function has eight entries.

1.4 BCD-to-seven-segment decoder

Consider the BCD-to-seven-segment decoder partially designed in Example 1.2.

a. Complete the design, assuming that none of the three open segments in Figure 1.7 is included.
b. Redo the design using a 4-to-16 decoder and 7 OR gates.
c. Determine what will be displayed when the logic circuits derived in part a are fed with each of the 6 forbidden inputs 1010 through 1111.
d. Repeat part c for the design of part b.

1.5 BCD-to-seven-segment decoder

Design modified forms of the BCD-to-seven-segment decoder of Example 1.2 under each of the following sets of assumptions. In all case assume that "1" is represented by the two segments on the right edge of the panel.

a. Inputs are BCD digits; the digits 6, 7, and 9 are displayed with 6, 3, and 6 segments, respectively (see Figure 1.7).
b. Inputs are hexadecimal digits, with the added digits 10–15 represented as uppercase (A, C, E, F) or lowercase (b, d) letters. Use the assumptions of part a for 6, 7, and 9.
c. Inputs are BCD digits and the display shapes are chosen so that any single segment enable signal becoming permanently stuck on 0 does not lead to an undetectable display error.
d. Repeat part c for a single enable signal becoming permanently stuck on 1.

1.6 Parity generation and checking

An n-input *even-parity generator* produces a 1 output iff an odd number of its inputs are 1s. The circuit is so named because attaching the produced output to the n-bit input yields an even-parity $(n + 1)$-bit word. An n-input *even-parity checker* produces a 1 output (error signal) iff an odd number of its inputs are 1s.

a. Design an 8-input even-parity generator using only XOR gates.
b. Repeat part a for an odd-parity generator.
c. Design a 9-input even-parity checker using only XOR gates. How is your design related to those in parts a and b?
d. Repeat part c for an odd-parity checker.
e. Show that the fastest parity generator circuits can be built in a recursive manner, basing the design of

an n-input circuit on two $(n/2)$-input circuits of suitable types. What happens when n is odd?

1.7 Multiplicity of Boolean functions

There are four single-variable Boolean functions of which only two actually depend of the input variable x (x and x'). Similarly, there are 16 two-variable function of which only 10 depend on both variables (exclude x, x', y, y', 0, and 1).

a. How many three-variable Boolean functions are there, and how many of them actually depend on all three variables?
b. Generalize the result of part a for n-variable functions. *Hint:* You must subtract from the number of n-variable functions all the ones that depend on $n - 1$ or fewer variables.

1.8 Equivalent logic expressions

Prove each of the four equivalences in Example 1.1 using the other three methods listed in the example.

1.9 Equivalent logic expressions

Use the arithmetic substitution method to prove the following pairs of logic expressions equivalent.

a. $xy' \vee x'z' \vee y'z' \equiv xy' \vee x'z'$
b. $xyz \vee x' \vee y' \vee z' \equiv 1$
c. $x \oplus y \oplus z \equiv xyz \vee xy'z' \vee x'yz' \vee x'y'z$
d. $xz \vee wy'z' \vee wxy' \vee w'xy \vee x'yz' \equiv$
   $xz \vee wy'z' \vee w'yz' \vee wx'z'$

1.10 Equivalent logic expressions

Prove that the following pairs of logic expressions are equivalent, first by the truth table method and then by means of symbolic manipulation (using the laws of Boolean algebra).

a. $xy' \vee x'z' \vee y'z' \equiv xy' \vee x'z'$
b. $xyz \vee x' \vee y' \vee z' \equiv 1$
c. $x \oplus y \oplus z \equiv xyz \vee xy'z' \vee x'yz' \vee x'y'z$
d. $xz \vee wy'z' \vee wxy' \vee w'xy \vee x'yz' \equiv$
   $xz \vee wy'z' \vee w'yz' \vee wx'z'$

1.11 Design of a 2 × 2 switch

Design a combinational circuit that acts as a 2 × 2 switch. The switch has data inputs a and b, one "cross" control signal c, and data outputs x and y. When $c = 0$, a is connected to x and b to y. When $c = 1$, a is connected to y and b to x (i.e., the inputs are crossed).

1.12 Numerical comparison circuits

Assume $x = (x_2x_1x_0)_{\text{two}}$ and $y = (y_2y_1y_0)_{\text{two}}$ are 3-bit unsigned binary numbers. Write down a logic expression in terms of the six Boolean variables x_2, x_1, x_0, y_2, y_1, y_0 that assumes the value 1 iff:

a. $x = y$
b. $x < y$
c. $x \leq y$
d. $x - y \leq 1$
e. $x - y$ is even
f. $x + y$ is divisible by 3

1.13 Numerical comparison circuits

Repeat Problem 1.12 (all parts), but assume that the 3-bit inputs x and y are 2's-complement numbers in the range -4 to $+3$.

1.14 Sum of products and product of sums

Express each of the following logical expressions in sum-of-products and product-of-sums forms.

a. $(x \vee y')(y \vee wz)$
b. $(xy \vee z)(y \vee wz')$
c. $x(x \vee y')(y \vee z') \vee x'$
d. $x \oplus y \oplus z$

1.15 Hamming SEC/DED code

A particular type of Hamming code has 8-bit codewords $P_8D_7D_6D_5P_4D_3P_2P_1$ that encode 16 different data values. The parity bits P_i are obtained from the data bits D_j according to the logic equations $P_1 = D_3 \oplus D_5 \oplus D_6$, $P_2 = D_3 \oplus D_5 \oplus D_7$, $P_4 = D_3 \oplus D_6 \oplus D_7$, $P_8 = D_5 \oplus D_6 \oplus D_7$.

a. Show that this code is capable of correcting any single-bit error and derive the correction rules. *Hint:* Think in terms of computing four parity check results, such as $C_1 = P_1 \oplus D_3 \oplus D_5 \oplus D_6$, all of which must yield 0 for an error-free codeword.
b. Show that the code detects all double-bit errors in addition to correcting single errors (thus it is a SEC/DED code).

c. Design the encoding circuit, using only 2-input NAND gates.
d. Design the decoding circuit that includes single-error correction and assertion of an error indicator signal in case of double errors.
e. Derive a PROM implementation of the decoding circuit of part d.

1.16 Mux-based implementation of logic functions

Show the inputs required to implement the following two logic functions using 4-to-1 multiplexers if only input x is available in complemented form.

a. $f(x, y, z) = y'z \vee x'y \vee xz'$
b. $g(w, x, y, z) = wx'z \vee w'yz \vee xz'$

1.17 Mux-based implementation of logic functions

a. Show that any three-variable logic function $f(x, y, z)$ can be realized using three 2-input multiplexers, assuming that x' is available as input.
b. Under what condition(s) can one realize a 3-variable function using three 2-input multiplexers, without requiring any complemented input?
c. Give an example of a three-variable function (truly depending on all three variables) that can be realized using only two 2-input multiplexers, with no complemented input available. Also show a diagram of the two-multiplexer realization of the function.
d. Can a three-variable function that is realizable with a single 2-input multiplexer truly depend on all three variables? Give an example or prove why such an arrangement is impossible.

1.18 Iterative number comparator

A iterative comparator for k-bit unsigned binary numbers consists of k cells arranged in a linear or cascade circuit. A cell receives one bit from each of the two operands x and y, and a pair of signals G_E ($x \geq y$ thus far) and L_E ($x \leq y$ thus far) from a neighboring cell, and produces G_E and L_E signals for the other neighbor. Note that $G_E = L_E = 1$ means that the two numbers are equal thus far.

a. Design the required cell assuming that G_E and L_E signals propagate from right to left.
b. Repeat part a for left-to-right signal propagation.
c. Show that the cells of part a or b can be connected into a tree structure, as opposed to linear array, to produce a faster comparator.

1.19 Arithmetic expressions for logic gates

The arithmetic expressions characterizing logic gates (Figure 1.1) can be extended to gates with more than two inputs. This is trivial for AND gates. Write the equivalent arithmetic expressions for 3- and 4-input OR gates. Generalize the expression to an h-input OR gate.

1.20 Programmable combinational parts

a. Show how to realize the logic function $f(x, y, z) = y'z \vee x'y \vee xz'$ on the PLA of Figure 1.13c.
b. Repeat part a on the PAL of Figure 1.13b.
c. Show that any function that can be realized by the PAL of Figure 1.13b is also realizable by the PLA of Figure 1.13c. Note that this is not a statement about PALs and PLAs in general but rather about the specific instances shown in Figure 1.13.

REFERENCES AND FURTHER READINGS

[Brow00] Brown, S., and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill, 2000.

[Erce99] Ercegovac, M. D., T. Lang, and J. H. Moreno, *Introduction to Digital Systems*, Wiley, 1999.

[Haye93] Hayes, J. P., *Introduction to Digital Logic Design*, Addison-Wesley, 1993.

[Katz94] Katz, R. H., *Contemporary Logic Design*, Benjamin/Cummings, 1994.

[Parh99] Parhami, B., and D.-M. Kwai, "Combinational Circuits," in *Encyclopedia of Electrical and Electronics Engineering,* Wiley, Vol. 3 (Ca-Co), 1999, pp. 562–569.

[Wake01] Wakerly, J. F., *Digital Design: Principles and Practices,* Prentice Hall, updated 3rd ed., 2001.

[WWW] Web names of some manufacturers of PALs and PLAs: altera.com, atmel.com, cypress.com, latticesemi.com, philips.com, vantis.com.