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

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

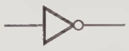
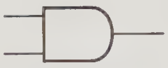
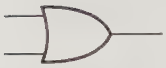
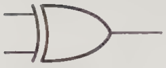
Name	NOT	AND	OR	XOR
Graphical symbol				
Operator sign and alternate(s)	x' $\neg x$ or $\overline{x}$	xy $x \wedge y$	$x \vee y$ $x + y$	$x \oplus y$ $x \not\equiv y$
Output is 1 iff:	Input is 0	Both inputs are 1s	At least one input is 1	Inputs are not equal
Arithmetic expression	$1 - x$	$x \times y$ or xy	$x + y - xy$	$x + y - 2xy$

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

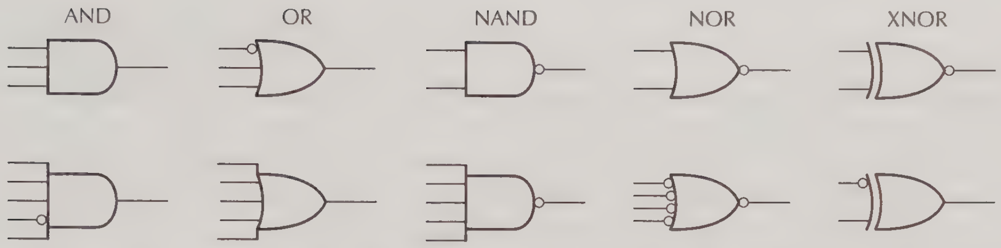


Figure 1.2 Gates with more than two inputs and/or with inverted signals at input or output.

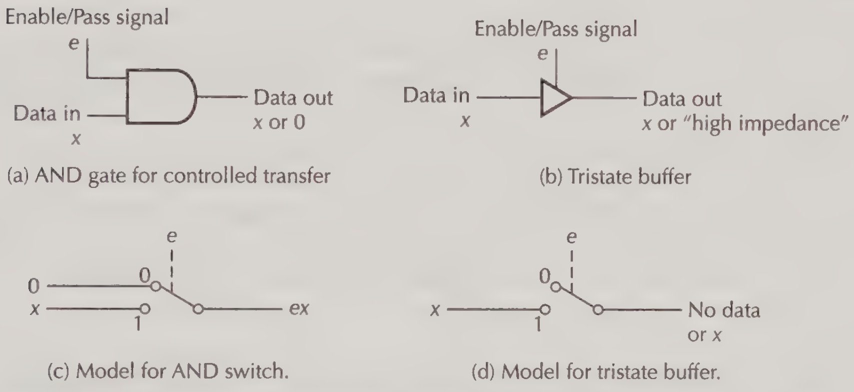
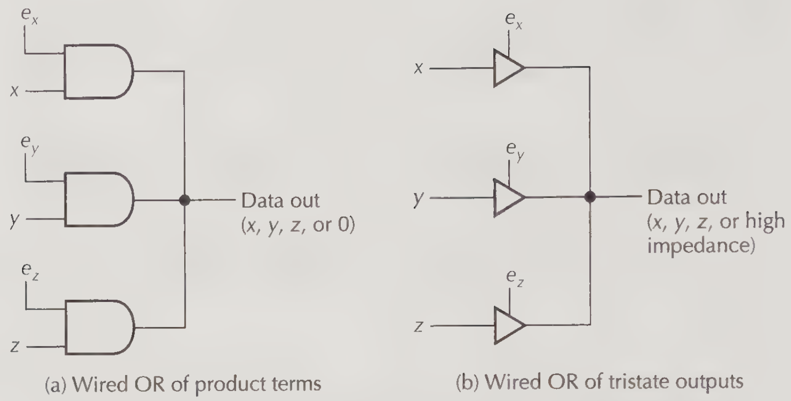


Figure 1.3 An AND gate and a tristate buffer can act as controlled switches or valves. An inverting buffer is logically the same as a NOT gate.

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 whenever



e_x
x
e_y
y
e_z
z
Data out
(x, y, z, or 0)
(a) Wired OR of product terms
e_x
x
e_y
y
e_z
z
Data out
(x, y, z, or high impedance)
(b) Wired OR of tristate outputs

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

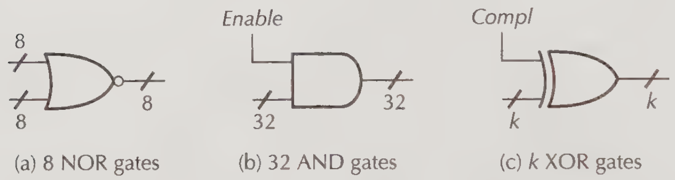


Figure 1.5 Arrays of logic gates represented by a single gate symbol.

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	$x \vee 0 = x$	$x1 = x$
One/Zero	$x \vee 1 = 1$	$x0 = 0$
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

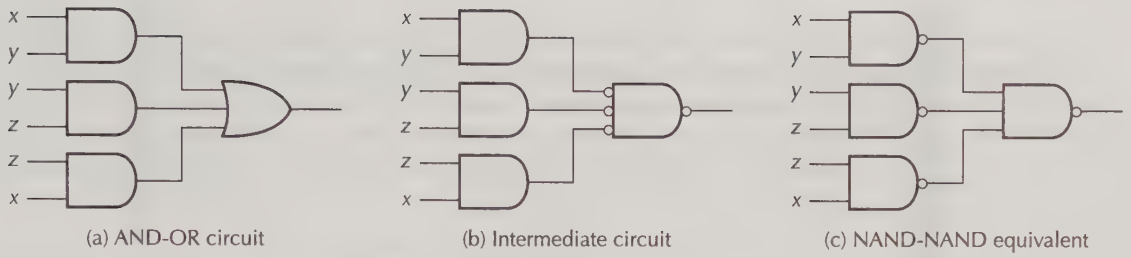


Figure 1.6 A two-level AND-OR circuit and two equivalent circuits.

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.

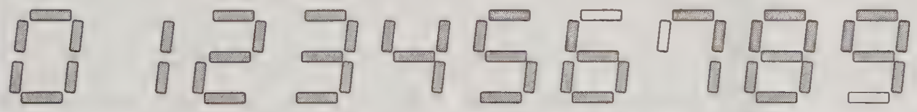


Figure 1.7 Seven-segment display of decimal digits. The three open segments may be optionally used. The digit 1 can be displayed in two ways, with the more common right-side version shown.

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.

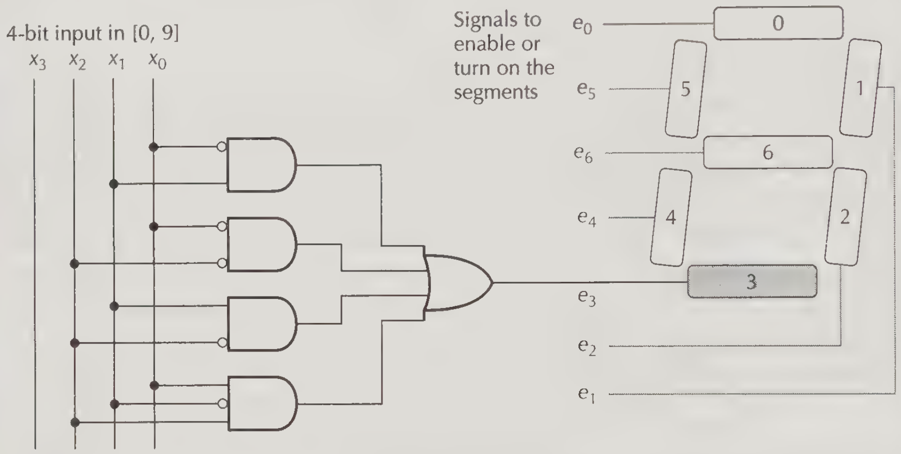


Figure 1.8 The logic circuit that generates the enable signal for the lowermost segment (number 3) in a seven-segment display unit.

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: $r_{\text{gum}}(r_{\text{bar}})$, 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 $s_{\text{lever}} = 1$. 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 $(y_{a-1} \ldots y_1 y_0)_{\text{two}}$. 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

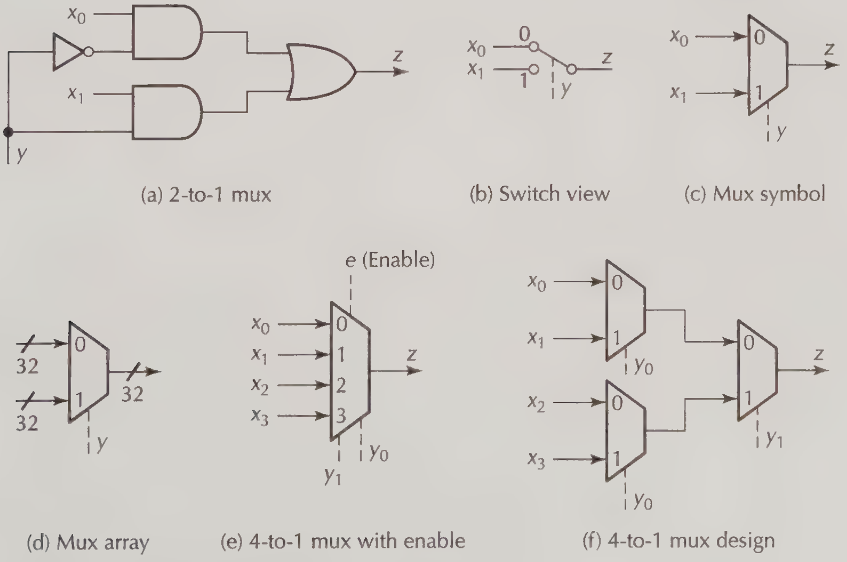


Figure 1.9 A multiplexer (mux), or selector, allows one of several inputs to be selected and routed to output depending on the binary value of a set of selection or address signals provided to it.

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

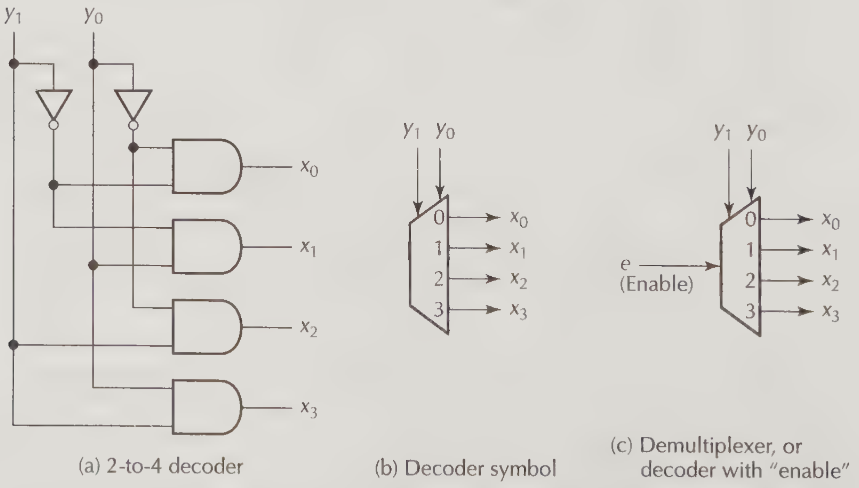


Figure 1.10 A decoder allows the selection of one of 2^a options using an a-bit address as input. A demultiplexer (demux) is a decoder that only selects an output if its enable signal is asserted.

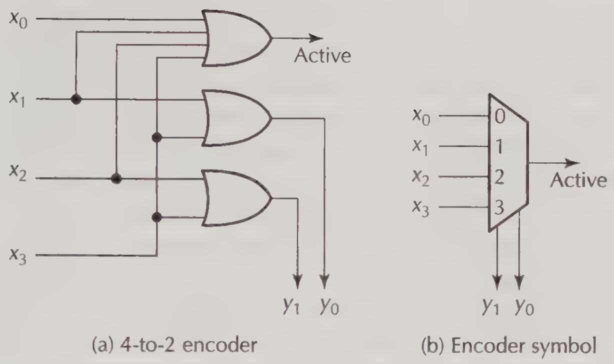


Figure 1.11 A 2^a-to-a encoder outputs an a-bit binary number equal to the index of the single 1 among its 2^a inputs.

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.

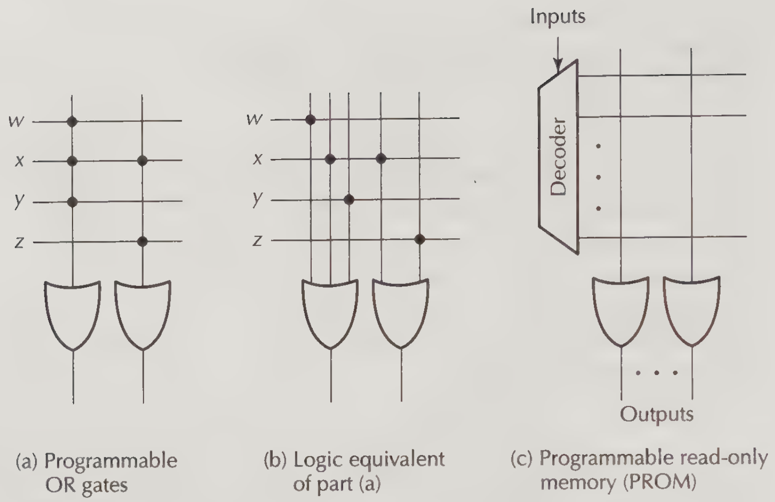

Inputs
w
x
y
z
Decoder
Outputs
(a) Programmable OR gates
(b) Logic equivalent of part (a)
(c) Programmable read-only memory (PROM)

Figure 1.12 Programmable connections and their use in a PROM.

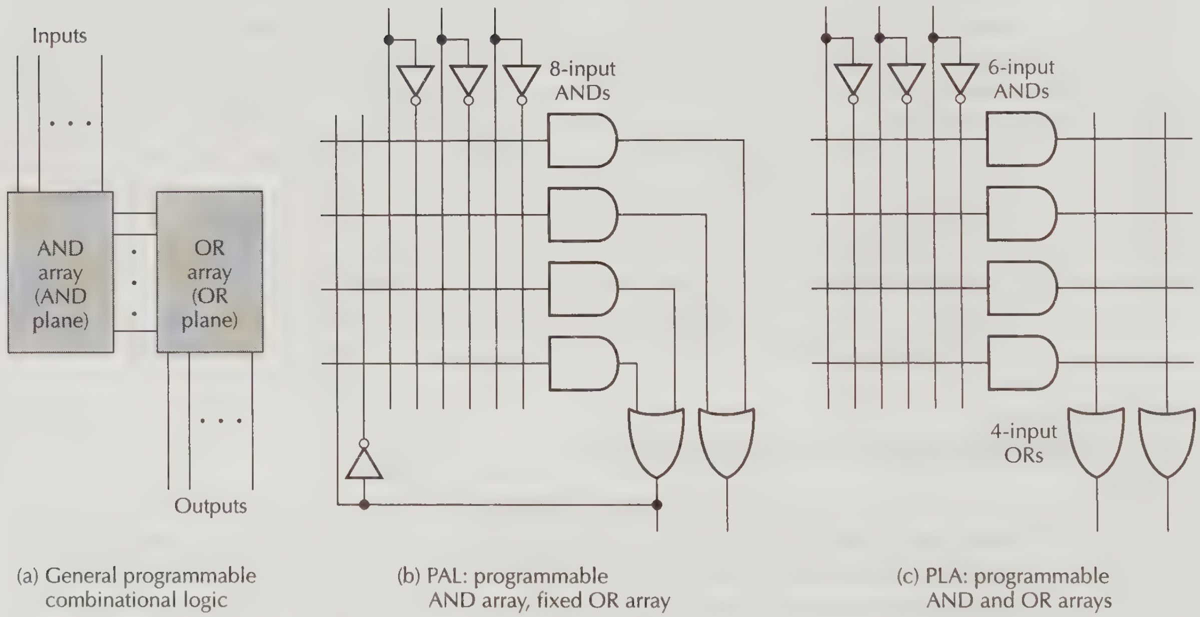

Inputs
AND array (AND plane)
OR array (OR plane)
Outputs
8-input ANDs
6-input ANDs
4-input ORs
(a) General programmable combinational logic
(b) PAL: programmable AND array, fixed OR array
(c) PLA: programmable AND and OR arrays

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 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

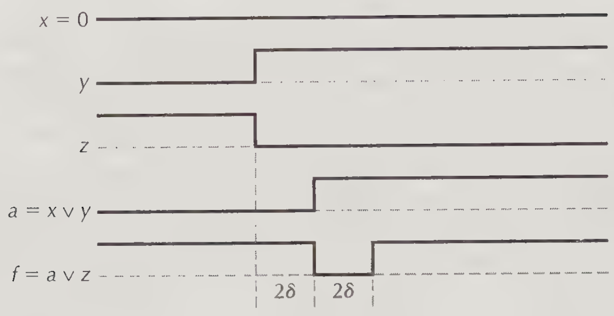
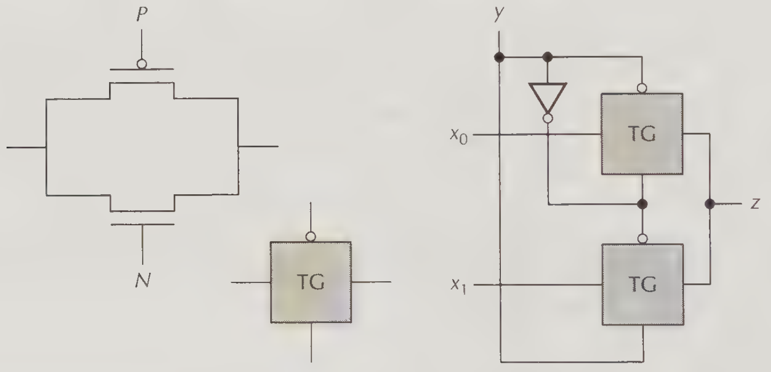


Figure 1.14 Timing diagram for a circuit that exhibits glitching.



(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.

CHAPTER 2

DIGITAL CIRCUITS WITH MEMORY

"Microprocessors were not a product of the computer industry at all. They were the outcome of the desire—and the imperative need—of the young semiconductor industry to find a profitable application for early VLSI. . . . The Intel engineers knew little about computer architecture. Their immediate objective was to make programmable devices that would replace random logic."
—Maurice Wilkes, Computing Perspectives

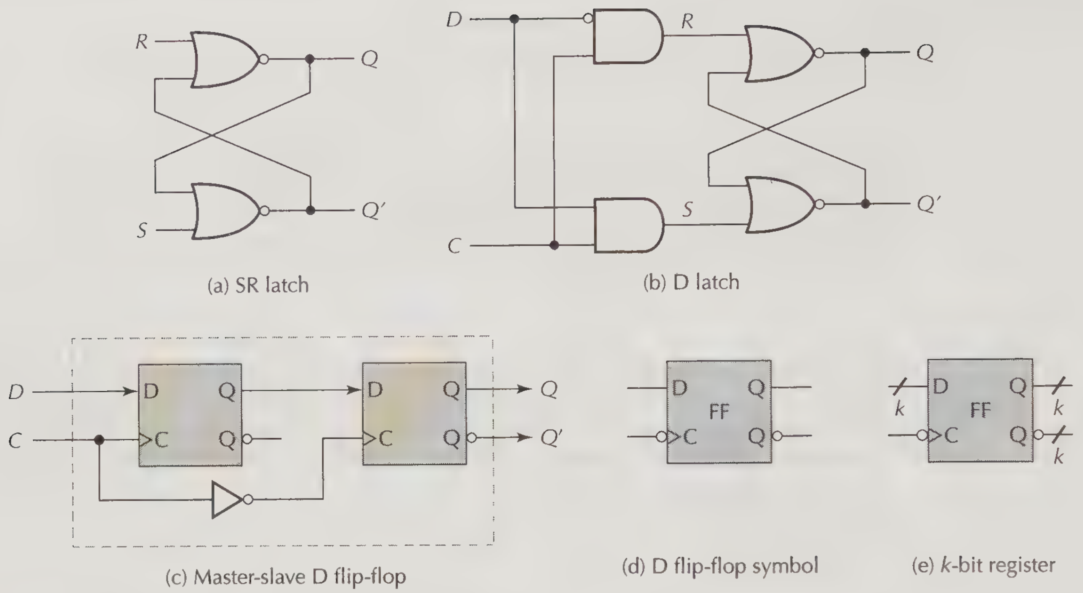
"The days of the digital watch are numbered."
—Tom Stoppard

TOPICS IN THIS CHAPTER

The behavior of a combinational (memoryless) circuit depends only on its current inputs, not on past history. A sequential digital circuit, on the other hand, has a finite amount of memory whose content, determined by past inputs, affects the current input/output behavior. In this chapter, we present a capsule review of methods for defining and realizing such sequential circuits by means of storage elements (latches, flip-flops, registers) and combinational logic. We also introduce a number of very useful components that are found in many diagrams in this book. Examples include register files, shift registers, and counters. As in Chapter 1, readers who have trouble understanding this material should consult any of the logic design textbooks listed at the end of the chapter.

2.1 Latches, Flip-Flops, and Registers

The design of sequential circuits exhibiting memory requires the use of storage elements capable of holding information. The simplest storage element is capable of holding a single bit and can be *set* to 1 or *reset* to 0 at will. The SR latch, depicted in Figure 2.1a, is one such



(a) SR latch (b) D latch (c) Master-slave D flip-flop (d) D flip-flop symbol (e) *k*-bit register

Figure 2.1 Latches, flip-flops, and registers.

element. When both R and S inputs are 0, the latch is in one of two stable states corresponding to $Q = 0$ ($Q' = 1$) or $Q = 1$ ($Q' = 0$). Asserting the R input resets the latch to $Q = 0$, while asserting S sets the latch to $Q = 1$. Either state then persists after the asserted input has been deasserted. Adding two AND gates to an SR latch, as in Figure 2.1b, produces a D latch with data (D) and clock (C) inputs. When C is asserted, the output Q of the SR latch follows D (the latch is set if $D = 1$ and reset if $D = 0$). We say that the latch is open or *transparent* for $C = 1$. Once C has been deasserted, the SR latch closes and maintains at its output the last value of D before deassertion of C.

Two D latches can be connected as in Figure 2.1c to form a master-slave D flip-flop. When C is asserted, the master latch is open and its output follows D, while the slave latch is closed, maintaining its state. Deassertion of C closes the master latch and causes its state to be copied into the slave latch. Figure 2.1d shows the shorthand notation for a D flip-flop. The bubble on the C input indicates that the new state of the flip-flop takes effect on the *negative edge* of the clock input, that is, when the clock goes down from 1 to 0. Such a flip-flop is said to be *negative-edge-triggered*. We can build a *positive-edge-triggered* D flip-flop by inverting the C input in Figure 2.1c. An array of k flip-flops, all tied to the same clock input, forms a k-bit register (Figure 2.1e). Because a master-slave flip-flop maintains its content as it is being modified, it is ideal for use in building registers that are often read from and written into in the same machine cycle.

Figure 2.2 depicts how changes in the D and C inputs affect the Q output in a D latch and a negative-edge-triggered D flip-flop. In a D latch, the output Q follows D whenever C is asserted and remains stable at its last value when C is deasserted. So, any change in Q coincides with a positive or negative edge of C or D; a small delay is involved owing to signal propagation time through gates. Arrows in Figure 2.2 indicate cause-effect relationships. In a D flip-flop, changes in Q coincide with negative edges of C; again, a small delay is involved. To avoid metastability, which causes improper flip-flop operation (see Section 2.6), the value of

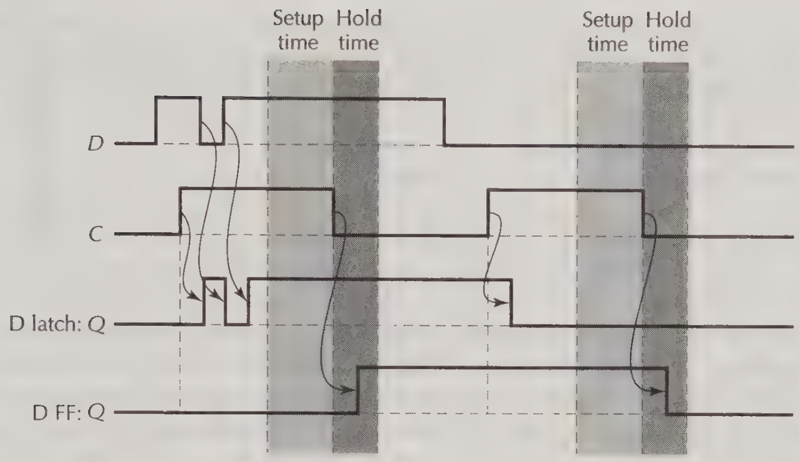


Figure 2.2 Operations of D latch and negative-edge-triggered D flip-flop.

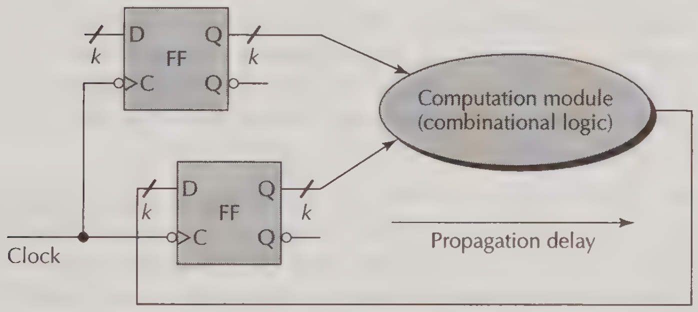


Figure 2.3 Register-to-register operation with edge-triggered flip-flops.

D should remain stable during a small window before and after a falling edge of *C*. The required stability time for *D* before the falling edge is known as *setup time,* while that after the falling edge is *hold time*.

Figure 2.3 shows a typical interconnection of registers and combinational components in synchronous sequential systems driven by a clock signal. One or more source registers provide the data used in a computation, and the result is stored in a destination register. Because of the master-slave register design leading to edge-triggered operation, the destination register may be the same as one of the source registers. As operand signals propagate through the computation module and begin to affect the input side of the destination register, the output of source registers remain stable, leading to the needed stability in the inputs of the destination register. As long as the clock period is greater than the sum of latch propagation delay, propagation delay through the combinational logic, and latch setup time, correct operation is ensured; as a rule, hold time can be ignored because it is usually smaller than latch propagation delay.

Because almost all the sequential circuits needed in this book use D flip-flops, while a few are based SR flip-flops that are easily derivable from the SR latch of Figure 2.1a, we do not cover other flip-flop types such as JK and T. Such flip-flops, described in virtually any textbook on logic design, lead to design simplifications in some cases; we avoid them, however, to keep the focus on notions in computer architecture rather than on details of logic circuit implementation.

2.2 Finite-State Machines

Just as Boolean functions and truth tables are abstract characterizations of combinational digital circuits, *finite-state machines* (or simply *state machines*) and *state tables* are used to specify the sequential behavior of a digital circuit with memory. For simple sequential circuits, we prefer the graphical representation of state tables, known as *state diagrams*. More complex circuits are described in an algorithmic fashion, given that both the state table and state diagram representations grow exponentially with the number of state variables. A finite-state machine with n bits of storage can have up to 2^n states; therefore, even a digital circuit with a single 32-bit register as memory is already too large to be described in state table form. We illustrate the process of deriving simple state tables through an example.

Example 2.1: Coin reception state machine A small vending machine can dispense items each costing 35 cents. Customers must use only quarters and dimes. The machine is not equipped to return change, but it is set to dispense the desired item when \$0.35 or more has been deposited. Derive a state table for the coin reception unit of this vending machine, assuming that coins are deposited, and thus detected, one at a time.

Solution: Figure 2.4 depicts a possible state table and the associated state diagram. The starting state is S_{00}. As coins are deposited, the unit moves from one state to another to "remember" the amount of money paid thus far. For example, the first dime deposit takes the unit from S_{00} to S_{10}, where the state name S_{10} is chosen to convey the amount deposited. The state S_{35} corresponds to the deposited amount being enough for dispensing an item. Once in the "sale

Current state ↓	-------Input-------		
	Dime	Quarter	Reset
S_{00}	S_{10}	S_{25}	S_{00}
S_{10}	S_{20}	S_{35}	S_{00}
S_{20}	S_{30}	S_{35}	S_{00}
S_{25}	S_{35}	S_{35}	S_{00}
S_{30}	S_{35}	S_{35}	S_{00}
S_{35}	S_{35}	S_{35}	S_{00}

Next state

S_{00} is the initial state
S_{35} is the final state

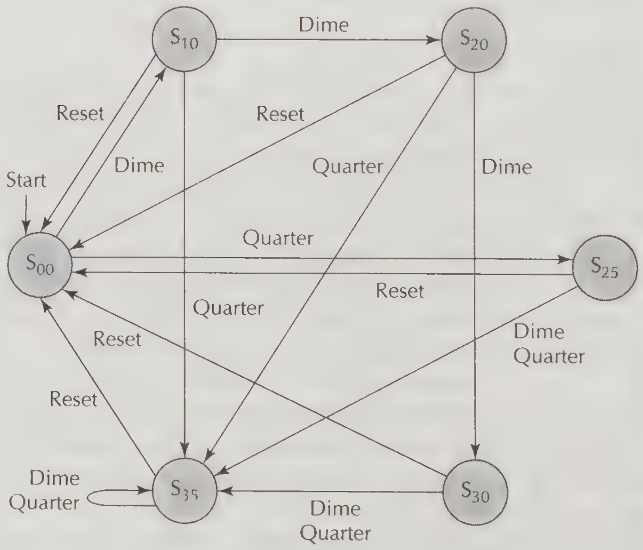


Figure 2.4 State table and state diagram for a vending machine coin reception unit.

enabled" state S_{35}, the finite-state machine undergoes no state change due to additional coin deposits. The machine has three inputs, which correspond to three events: insertion of a dime, insertion of a quarter, and reset (due to the sale conditions not being met, causing coin return, or sale completion). Because states S_{25} and S_{30} are completely equivalent from the viewpoint of the coin reception process (both need an additional dime or quarter to allow transition to S_{35}), they can be merged to obtain a simpler five-state machine.

The finite-state machine represented in Figure 2.4 is known as a *Moore machine* because the machine's output is associated with its states. In state S_{35}, dispensing of the selected item is enabled, whereas in all other states, it is disabled. In a *Mealy machine,* on the other hand, outputs are associated with the transitions between states, and thus the output depends on both the present state and the current input received. Figure 2.5 shows how Moore and Mealy machines may be realized in hardware. The machine's state is held in an l-bit register that allows up to 2^l states. The next-state logic circuit produces the excitation signals required to effect a change of state based on n inputs and l state variables, while the output logic circuit produces the machine's m output signals. When the state register is composed of D flip-flops, the number of next-state excitation signals is the same as the number of state variables, because each D flip-flop needs one data input. Outputs are derived based on state variables only (Moore machine) or state variables and inputs (Mealy machine).

2.3 Designing Sequential Circuits

Hardware realization of sequential circuits, according to the general structure in Figure 2.5, begins with the selection of memory elements to be used. Here, we deal exclusively with D flip-flops, so this step of the design process is predetermined. Next, the states must be encoded using l state variables for a suitable value of l. The choice of l obviously affects the cost of memory elements. However, this does not automatically mean that we always aim to minimize l; often a sparser encoding of states, using more state variables, leads to simpler combinational circuits for generating the excitation signals and the outputs. A detailed review of this *state assignment* process and associated trade-offs is beyond the scope of our discussion. We note only that the choices range from the densest possible encoding, where 2^l is strictly less than twice the number of states, to the sparsest, where l is equal to the number of states and

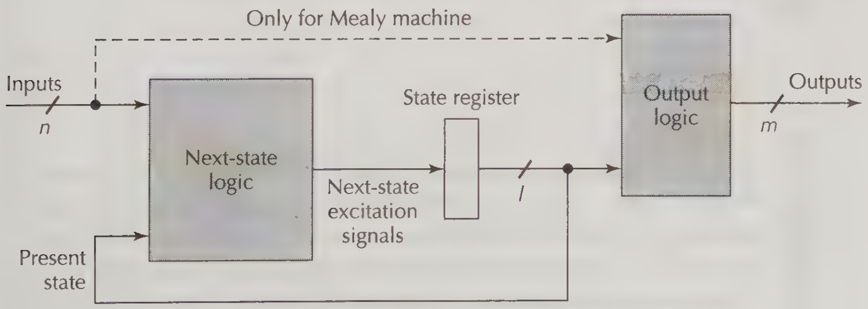


Figure 2.5 Hardware realization of Moore and Mealy sequential machines.

each state code is an l-bit string containing a single 1 (this is known as *one-hot encoding*). After state assignment, the truth tables for the two combinational blocks in Figure 2.5, and thus their circuit realizations, are easily derived. We illustrate the complete design process through two examples.

Example 2.2: Building a JK flip-flop Design a JK flip-flop out of a single D flip-flop and combinational logic elements. A JK flip-flop, a memory element with two inputs (J and K) and two outputs (Q and Q'), holds its state when $J = K = 0$, is reset to 0 when $J = 0$ and $K = 1$, is set to 1 when $J = 1$ and $K = 0$, and inverts its state (changes from 0 to 1 or 1 to 0) when $J = K = 1$.

Solution: The problem statement defines the state table for JK flip-flop (Table 2.1) which is essentially the truth table for a function D of three variables (J, K, Q). The following excitation input for the D flip-flop is easily derived from Table 2.1: $D = JQ' \vee K'Q$. The resulting circuit is depicted in Figure 2.6. Because the D flip-flop used in the design is negative-edge-triggered, so is the resulting JK flip-flop.

TABLE 2.1 State table for a JK flip-flop defined in Example 2.2.

Next state for → Current state ↓	$J=0$ $K=0$	$J=0$ $K=1$	$J=1$ $K=0$	$J=1$ $K=1$
0	0	0	1	1
1	1	0	1	0

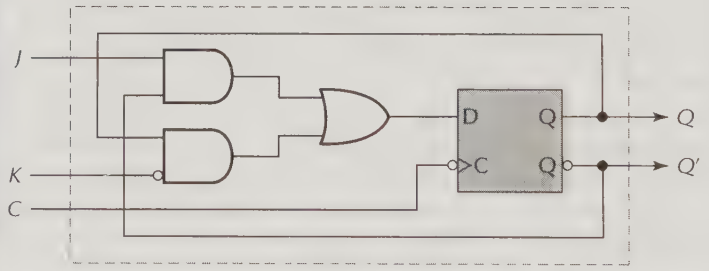


Figure 2.6 Hardware realization of a JK flip-flop (Example 2.2).

Example 2.3: Sequential circuit for a coin reception unit Consider the state table and state diagram for the vending machine coin reception unit derived in Example 2.1. Assume that states S_{25} and S_{30} are merged into state $S_{25/30}$, as suggested at the end of the solution for Example 2.1. Design a sequential circuit to implement the resulting five-state machine using D flip-flops with separate asynchronous reset inputs.

Solution: There are five states, only one of which allows a sale to be completed. We need at least 3 bits to encode the five states. Because only the state S_{35} allows the sale to proceed, it seems natural to distinguish that state by encoding it as $Q_2Q_1Q_0 = 1xx$, making Q_2 the "enable sale" output. For the remaining states, the following assignments might be used: 000 for S_{00} (because it is the reset state), 001 for S_{10}, 010 for S_{20}, and 011 for $S_{25/30}$. Let q and d be the inputs which, when asserted for one clock cycle, indicate the detection of a quarter and a dime, respectively. Because coins are inserted and detected one at a time, $q = d = 1$ is a don't-care condition. These choices lead to an encoded state table (Table 2.2), which is essentially the truth table for three functions (D_2, D_1, D_0) of five variables (q, d, Q_2, Q_1, Q_0). The following excitation inputs for the D flip-flops are easily derived: $D_2 = Q_2 \vee qQ_0 \vee qQ_1 \vee dQ_1Q_0$, $D_1 = q \vee Q_1 \vee dQ_0$, $D_0 = q \vee d'Q_0 \vee dQ_0'$. The resulting circuit is depicted in Figure 2.7.

TABLE 2.2 State table for a coin reception unit after the state assignment chosen in Example 2.3.

Next state for → **Current state ↓**	$q=0$ $d=0$	$q=0$ $d=1$	$q=1$ $d=0$	$q=1$ $d=1$
$S_{00} = 000$	000	001	011	xxx
$S_{10} = 001$	001	010	1xx	xxx
$S_{20} = 010$	010	011	1xx	xxx
$S_{25/30} = 011$	011	1xx	1xx	xxx
$S_{35} = 1xx$	1xx	1xx	1xx	xxx

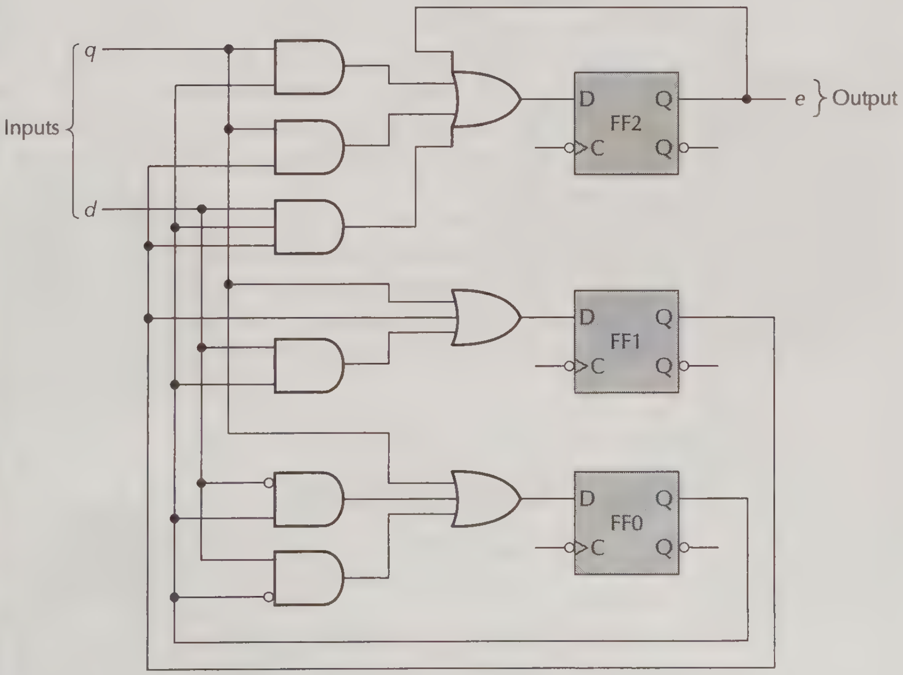


Figure 2.7 Hardware realization of a coin reception unit (Example 2.3).

2.4 Useful Sequential Parts

A register is an array of flip-flops with individual data inputs and common control signals. Certain special types of register are commonly used in building digital systems. For example, a shift register can be loaded with new content or with its old content shifted to the right or left. Conceptually, a shift register can be built of an ordinary register connected to a multiplexer. A register capable of single-bit left shift is shown in Figure 2.8. When the register is clocked, a new data word or the left-shifted version of the word stored in the register replaces its old content. By using a larger multiplexer and appropriate control signals, other types of shift can be accommodated. Multibit shifts are accommodated either by doing several single-bit shifts over successive clock cycles (which is rather slow) or by using a special combinational circuit that can shift by various amounts. The design of such a *barrel shifter,* which is primarily used in the alignment and normalization of floating-point operands and results, will be discussed in Chapter 12.

Just as a register is an array of flip-flops (Figure 2.1e), a *register file* is an array of registers (Figure 2.9). The index or address of a register is used to read from it or to write into it. For a register file with 2^h registers, the register address is an h-bit binary number. Because many operations require more than one operand, register files are almost always *multiported,* meaning that they are capable of supplying the data words stored in multiple registers at once, while at the same time writing into one or more registers. Registers built of negative-edge-triggered master-slave flip-flops can be read out and modified in the same clock cycle, with changes in content taking effect in the next clock cycle. Thus, an operation such as $B \leftarrow A + B$, where A and B are stored in registers, can be executed in a single clock cycle, given that writing of the new value of B does not interfere with reading of its present value (see Figure 2.3).

A *FIFO* (pronounced fie-foe) is a special *first-in, first-out* register file whose elements are accessed in the same order that they were put in. Thus, a FIFO, does not need an address input; rather, it is equipped with one read port, one write port, and special indicator signals designating "FIFO empty" and "FIFO full." As shown in Figure 2.9c, the write enable and read enable signals for a FIFO are called "push" and "pop," respectively, indicating whether data is to be pushed into the FIFO or popped from it. A FIFO is basically a hardware-implemented *queue* with a fixed maximum size.

An SRAM (static random-access memory) device is much like a register file, except that it is usually single-ported and much larger in capacity. A $2^h \times g$ SRAM chip (Figure 2.10) receives an h-bit address as input and supplies g bits of data as output during a read operation. For a write operation, g bits of input data are written into the selected location. Input and output data usually share the same pins, given that they are never used at the same time. The write

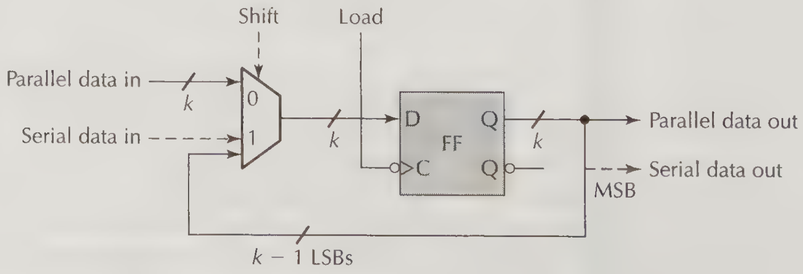
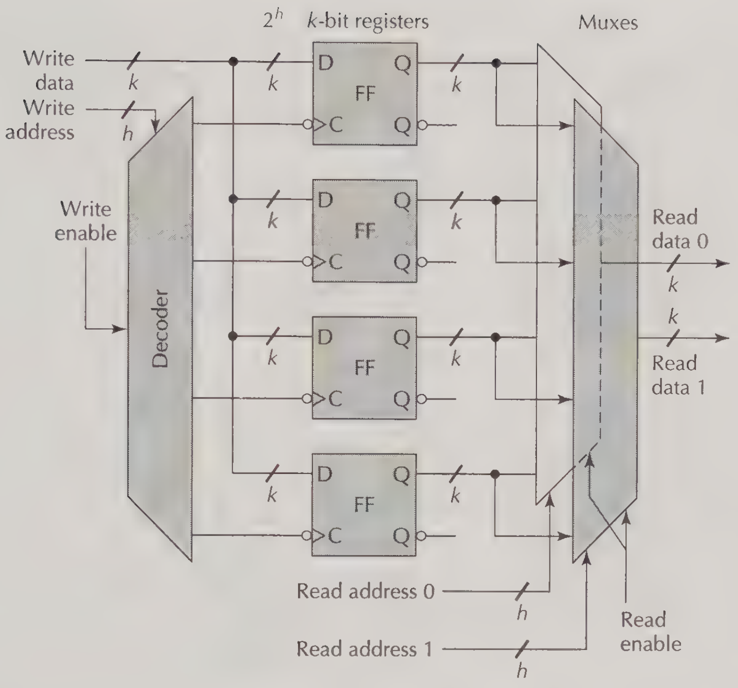
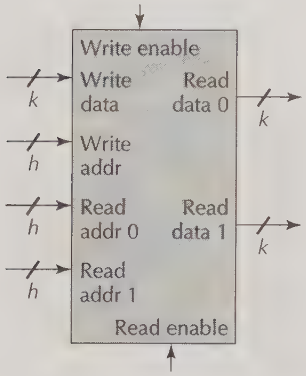


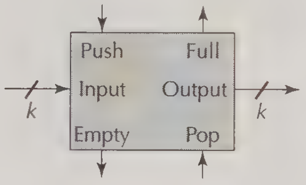
Figure 2.8 Register with single-bit left shift and parallel load capabilities. For logical left shift, the serial data in line is connected to 0.



(a) Register file with random access

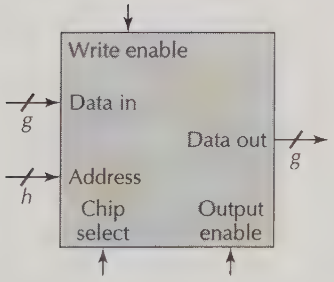


(b) Graphic symbol for register file

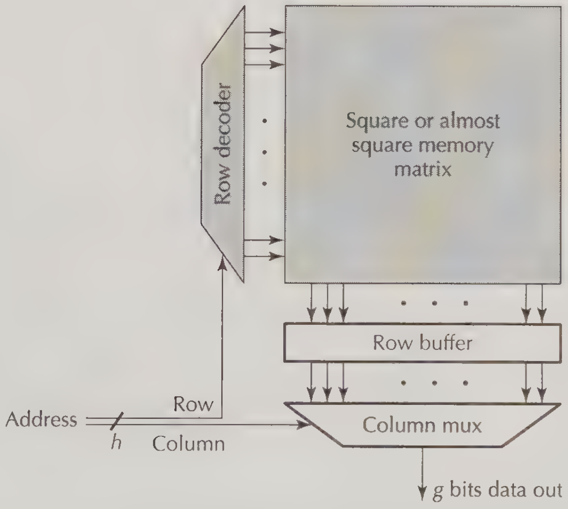


(c) FIFO symbol

Figure 2.9 Register file with random access and FIFO.



(a) SRAM block diagram



(b) SRAM read mechanism

Figure 2.10 SRAM memory is simply a large, single-port register file.

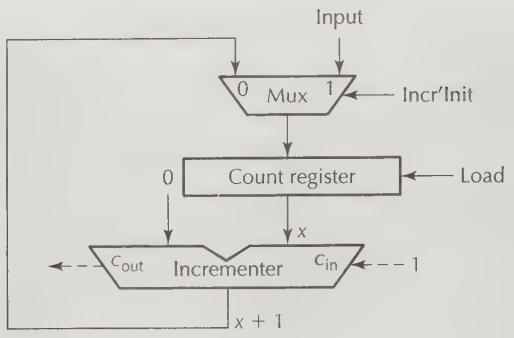


Figure 2.11 Synchronous binary counter with initialization capability.

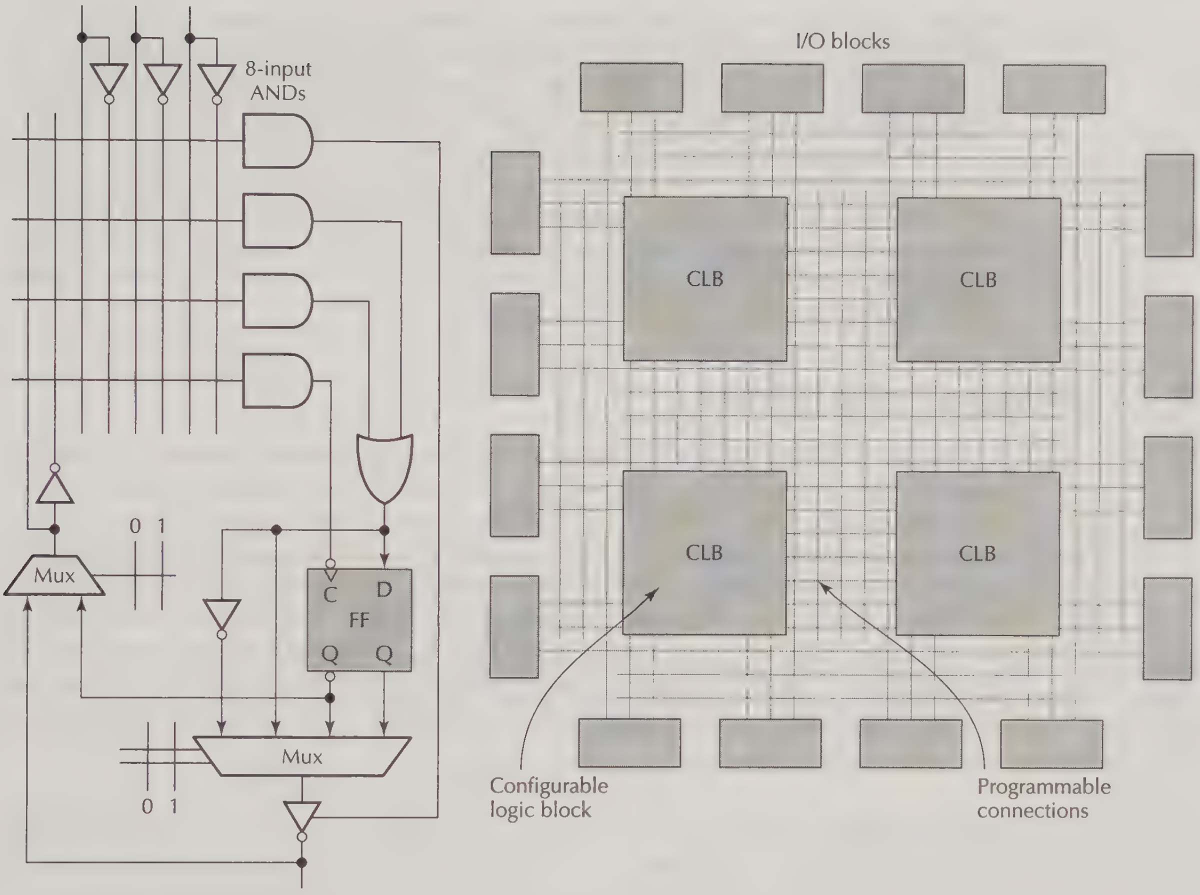
enable signal has the same role here as in register files. The output enable signal controls a set of tristate buffers supplying the output data, thus allowing us to connect a number of such chips to a single set of output or bus lines. To form a memory with k-bit words, where $k > g$, we connect k/g such chips in parallel, with all of them receiving the same address and control signals and each supplying or receiving g bits of the k-bit data word (in the rare event that g does not divide k, we use $\lceil k/g \rceil$ chips). Similarly, to form a memory with 2^m locations, where $m > h$, we use 2^{m-h} rows of chips, each containing k/g chips, with an external $(m - h)$-to-2^{m-h} decoder used to select the row of chips that should have its chip select signal asserted.

SRAM technology is used to implement small fast memories located near the processor (often on the same chip). The larger main memory of a computer is implemented in DRAM or dynamic random-access memory technology, to be discussed in Chapter 17. Briefly, DRAM requires the use of one transistor to store one bit of data, whereas SRAM needs several transistors per bit. This difference makes DRAM denser and cheaper, but also slower, than SRAM. *Read-only memories* (ROMs) are also commonly used in the design of digital system. A $2^h \times g$ ROM can be used to realize g Boolean functions of h variables and can thus be viewed as a form of programmable logic. ROM contents can be loaded permanently at the time of manufacture or be "programmed" into the memory by means of special ROM programming devices. In the latter case, we have a *programmable ROM* (PROM) device. An *erasable PROM* (EPROM) can be programmed more than once.

An *up counter* is built of a register and an incrementer, as shown in Figure 2.11. Similarly, a *down counter* is composed of a register and a decrementer. In the absence of explicit qualification, the term "counter" is used for an up counter. An *up/down counter* can count either up or down under the control of a direction signal. The counter design shown in Figure 2.11 is adequate for most applications. It can be made faster by using a fast incrementer with carry-lookahead feature similar to that used in fast adders, to be discussed in Chapter 10. If still higher speed is required, the counter can be divided into blocks. A short initial block (say, 3 bits wide) can easily keep up with the fast incoming signals. Increasingly wider blocks to the left of the initial block need not be as fast because they are adjusted less and less frequently.

2.5 Programmable Sequential Parts

Programmable sequential parts consist of programmable arrays of gates with strategically placed memory elements to hold data from one clock cycle to the next. For example, a commonly used form of PAL with memory elements has a structure similar to Figure 1.13b, but each OR gate output can be stored in a flip-flop and the device output is selectable from among



(a) Portion of PAL with storable output (b) Generic structure of an FPGA

Figure 2.12 Examples of programmable sequential logic.

the OR gate output, its complement, and the flip-flop outputs (Figure 2.12a). Either the OR gate output or the flip-flop output can be fed back into the AND array through a 2-to-1 multiplexer. The three signals controlling the multiplexers in Figure 2.12a can be linked to logic 0 or 1 through programmable connections.

Programmable circuits similar to the one depicted in Figure 2.12a provide all the elements necessary for implementing a sequential machine (Figure 2.5) and can also be used as a combinational parts if desired. Such devices have two distinct parts. The *gate array,* which is essentially a PAL or PLA like the ones shown in Figure 1.13, is provided for its ability to realize desired logic functions. The *output macrocell,* containing one or more flip-flops, multiplexers, tristate buffers, and/or inverters, forms the required outputs based on values derived in the current cycle and those derived in earlier cycles and stored in the various memory elements.

The ultimate in flexibility is offered by *field-programmable gate arrays* (FPGAs), depicted in simplified form in Figure 2.12b, which are composed of a large number of configurable logic blocks (CLBs) in the center, surrounded by I/O blocks at the edges. Programmable connections fill the spaces between the blocks. Each CLB is capable of realizing one or two arbitrary logic functions of a small number of variables and also has one or more memory elements. Each I/O block is similar to the output macrocell in the lower half of Figure 2.12a. Groups of CLBs and I/O blocks can be linked together via programmable interconnections to form complex digital systems. The memory elements (typically SRAM) that hold the connectivity pattern between cells can be initialized from ROMs upon start-up to define the system's functionality. They might also be loaded with new values at any time to effect *run-time reconfiguration*. Special software packages, supplied by manufacturers of FPGAs and independent vendors, allow automatic mapping onto FPGAs of algorithmically specified hardware functionality. A more detailed discussion of FPGAs and other sequential programmable parts can be found in the end-of-chapter references.

2.6 Clocks and Timing of Events

A clock is a circuit that produces a periodic signal, usually at a constant frequency or rate. The inverse of the clock rate is the clock period. For example, a one-gigahertz (GHz) clock has a frequency of 10^9 and a period of $1/10^9$ s $= 1$ ns. Typically, the clock signal is at 0 or 1 for about half the clock period (Figure 2.13). The operation of a synchronous sequential circuit is governed by a clock. With edge-triggered flip-flops (FFs), correct operation of the circuit shown in Figure 2.13 can be ensured by making the clock period long enough to accommodate the worst-case delay through the combinational logic, t_{comb}, while still leaving enough time for the setup time of FF2. Given that any change in FF1 is not instantaneous either but requires an amount of time known as *propagation time*, t_{prop}, we arrive at the following requirement:

$$\text{Clock period} \geq t_{\text{prop}} + t_{\text{comb}} + t_{\text{setup}} + t_{\text{skew}}$$

The extra term t_{skew}, for *clock skew*, accounts for the possibility that owing to signal propagation delays and certain anomalies, the clock signal controlling FF2 may arrive slightly ahead of that for FF1, thus effectively shortening the clock period. Note that the flip-flop hold time is absent from the foregoing inequality because it is almost always less than t_{prop} (i.e., during t_{prop}, a flip-flop's output remains at its previous value, thus satisfying the hold time requirement).

One way to allow higher clock rate, and thus greater computation throughput, is to use *pipelining*. In pipelined mode, the combinational logic of Figure 2.13 is divided into stages

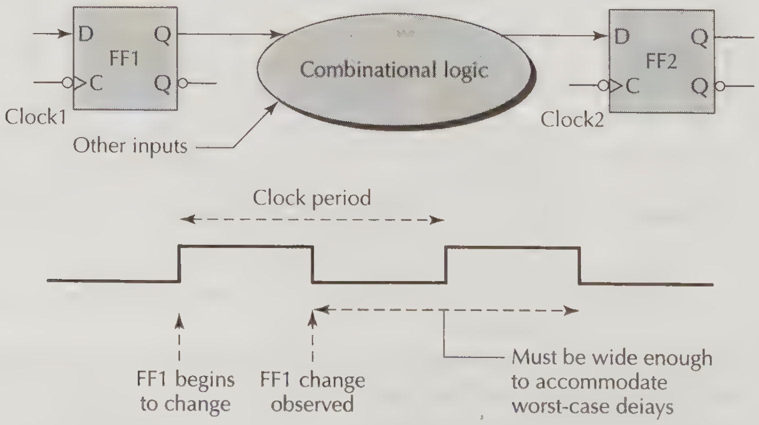


Figure 2.13 Determining the required length of the clock period.

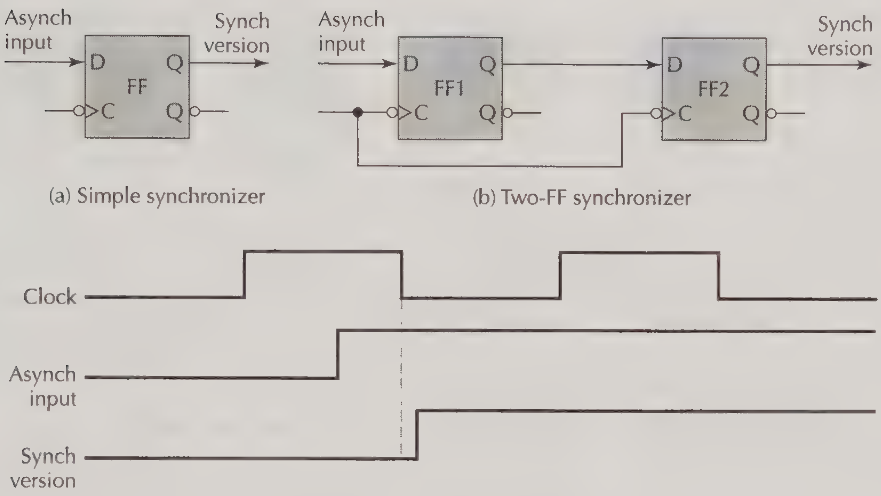


Figure 2.14 Synchronizers are used to prevent timing problems that might otherwise arise from untimely changes in asynchronous signals.

and storage elements inserted between stages to hold partial computation results. In this way, a new computation can begin as soon as results of the first stage are safely stored. However, this technique affects only the t_{comb} term in the preceding inequality. The clock period must still accommodate the other three terms. For this reason, there is a limit beyond which adding more pipeline stages will not be cost-effective.

Implicit in the foregoing inequality is the assumption that signals do not change within the clock period thus determined. In other words, if the constraint is barely satisfied, as is often the case when we try to use the fastest possible clock rate to maximize performance, then any change in signal values within the clock period could lead to a timing violation. Because many signals of interest may come from units that are not governed by the same clock (these are known as *asynchronous inputs*), their variations are beyond our control. For this reason, such signals are passed through special circuits known as *synchronizers* whose function is to ensure signal stability for the required duration of time. Figure 2.14a shows how a single flip-flop might be used to synchronize an asynchronous input. Any change in the asynchronous input can be observed only immediately after the next negative clock edge. There is a chance, however, that the change may occur too close to the next negative clock edge (Figure 2.14c), leading to a metastability condition where the observed signal is neither 0 nor 1; worse yet, the signal may appear as 0 to some units and as 1 to others. For this reason, two-FF synchronizers (Figure 2.14b) are preferred. While such synchronizers do not eliminate metastability altogether, they do make it so improbable that practical problems are avoided.

Somewhat faster operation results if we use latches and level-sensitive timing rather than edge-triggered flip-flops (to see this, compare the latch and flip-flop circuits in Figure 2.1). With level-sensitive operation, instead of abrupt changes that coincide with clock edges, a latch remains open for the entire time during which the clock signal is high. This can lead to problems if two successive latches are ever open at the same time. For this reason, level-sensitive timing is often used in conjunction with *two-phase clocking*. In this scheme, depicted in Figure 2.15, two nonoverlapping clock signals, ϕ_1 and ϕ_2 (for phase 1 and phase 2), are used to control successive latches in the computation path. This helps ensure that when a

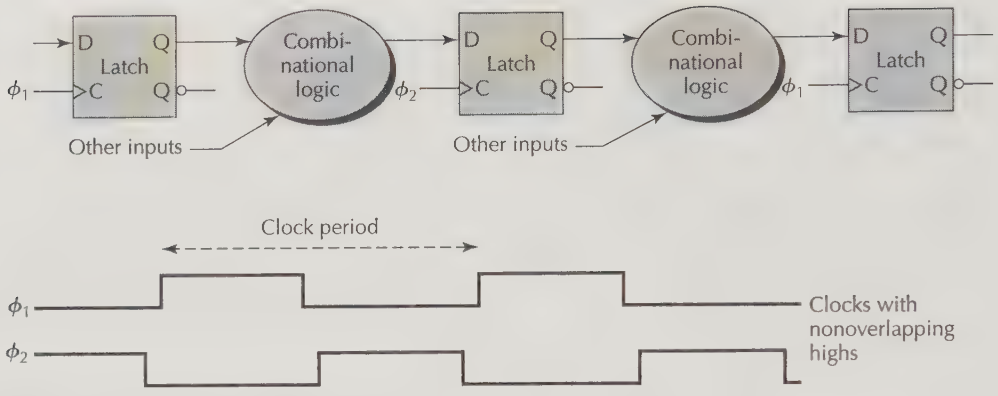


Figure 2.15 Two-phase clocking with nonoverlapping clock signals.

latch is open, the next latch downstream is always closed, and vice versa. As a result, signals propagate only from one latch to the next, as in the edge-triggered method. For simplicity, we always think of our circuits as having edge-triggered timing, knowing that we can convert any such design to level-sensitive operation through two-phase clocking.

PROBLEMS

2.1 Alternate forms of the D latch

a. By using the fact that an OR gate is replaceable by a NAND gate with inverted inputs, show that the D latch can be built of four NAND gates and an inverter.
b. Show that the inverter in the design of part a can be removed, leaving only four NAND gates, by feeding the output of one of the other gates to the gate that had the inverter on one input.

2.2 D latch vs D flip-flop

Characterize all input signal pairs, D and C, that when applied to a D latch and a D flip-flop lead to exactly the same observed output Q in terms of values and timing.

2.3 Other types of flip-flop

The negative-edge-triggered master-slave D flip-flop is but one of many types of flip-flop that can be built. Each flip-flop can be described by a characteristic table that relates its change of state to changes on its input line(s). The JK flip-flop case was discussed in Example 2.2. Show how each of the following flip-flop types can be built from a D flip-flop and a minimal amount of combinational logic.

a. An SR (set/reset) flip-flop that holds its state for $S = R = 0$, is set to 1 for $S = 1$, is reset to 0 for $R = 1$, and behaves unpredictably for $S = R = 1$. You can use the latter combination of input values as a don't-care, assuming that it will never occur in operation.
b. A T (toggle) flip-flop, with one input T, that holds its state for $T = 0$ and inverts its state for $T = 1$.
c. A D flip-flop with separate preset and clear inputs that essentially have the same effects as the S and R inputs of an SR flip-flop defined in part a. When either S or R is asserted, the flip-flop behaves like an SR flip-flop; otherwise, it behaves like a D flip-flop. The case $S = R = 1$ is considered a don't-care condition.