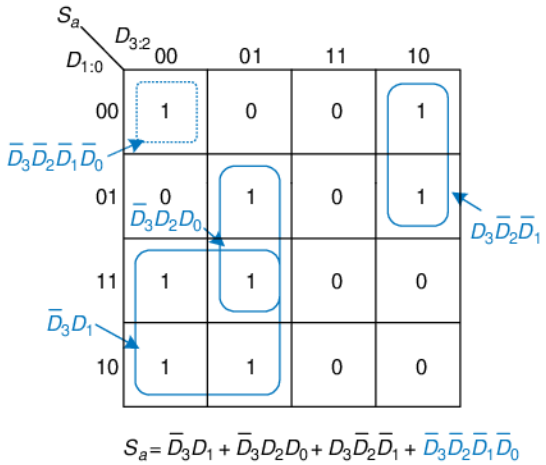


Figure 2.52 **Alternative K-map for S_a showing incorrect nonprime implicant**

In a K-map, X's allow for even more logic minimization. They can be circled if they help cover the 1's with fewer or larger circles, but they do not have to be circled if they are not helpful.

Example 2.11 SEVEN-SEGMENT DISPLAY DECODER WITH DON'T CARES

Repeat Example 2.10 if we don't care about the output values for illegal input values of 10 to 15.

Solution: The K-map is shown in Figure 2.53 with X entries representing don't care. Because don't cares can be 0 or 1, we circle a don't care if it allows us to cover the 1's with fewer or bigger circles. Circled don't cares are treated as 1's, whereas uncircled don't cares are 0's. Observe how a 2×2 square wrapping around all four corners is circled for segment S_a. Use of don't cares simplifies the logic substantially.

2.7.4 The Big Picture

Boolean algebra and Karnaugh maps are two methods for logic simplification. Ultimately, the goal is to find a low-cost method of implementing a particular logic function.

In modern engineering practice, computer programs called *logic synthesizers* produce simplified circuits from a description of the logic function, as we will see in Chapter 4. For large problems, logic synthesizers are much more efficient than humans. For small problems, a human with a bit of experience can find a good solution by inspection. Neither of the authors has ever used a Karnaugh map in real life to

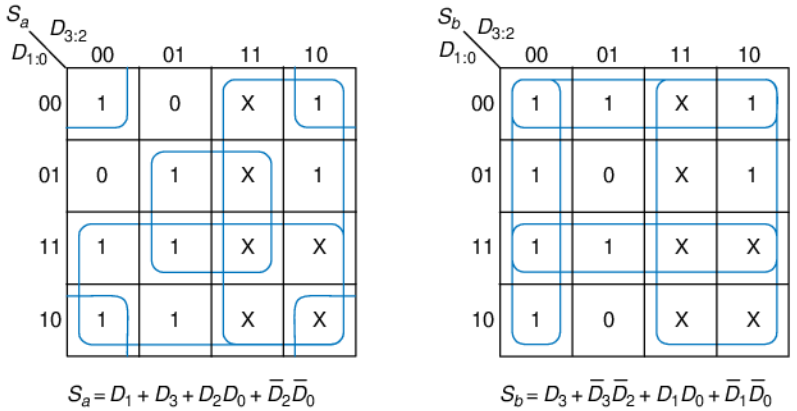


Figure 2.53 K-map solution with don't cares

solve a practical problem. But the insight gained from the principles underlying Karnaugh maps is valuable. And Karnaugh maps often appear at job interviews!

2.8 COMBINATIONAL BUILDING BLOCKS

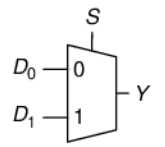
Combinational logic is often grouped into larger building blocks to build more complex systems. This is an application of the principle of abstraction, hiding the unnecessary gate-level details to emphasize the function of the building block. We have already studied three such building blocks: full adders (from Section 2.1), priority circuits (from Section 2.4), and seven-segment display decoders (from Section 2.7). This section introduces two more commonly used building blocks: multiplexers and decoders. Chapter 5 covers other combinational building blocks.

2.8.1 Multiplexers

Multiplexers are among the most commonly used combinational circuits. They choose an output from among several possible inputs based on the value of a *select* signal. A multiplexer is sometimes affectionately called a *mux*.

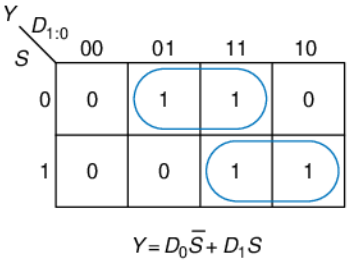
2:1 Multiplexer

Figure 2.54 shows the schematic and truth table for a 2:1 multiplexer with two data inputs, D_0 and D_1, a select input, S, and one output, Y. The multiplexer chooses between the two data inputs based on the select: if $S = 0$, $Y = D_0$, and if $S = 1$, $Y = D_1$. S is also called a *control signal* because it controls what the multiplexer does.



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2.54 2:1 multiplexer symbol and truth table



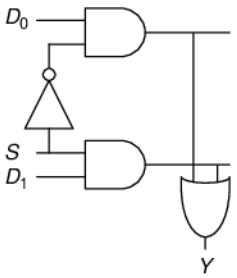


Figure 2.55 **2:1 multiplexer implementation using two-level logic**

A 2:1 multiplexer can be built from sum-of-products logic as shown in Figure 2.55. The Boolean equation for the multiplexer may be derived with a Karnaugh map or read off by inspection (Y is 1 if $S = 0$ AND D_0 is 1 OR if $S = 1$ AND D_1 is 1).

Alternatively, multiplexers can be built from tristate buffers, as shown in Figure 2.56. The tristate enables are arranged such that, at all times, exactly one tristate buffer is active. When $S = 0$, tristate T0 is enabled, allowing D_0 to flow to Y. When $S = 1$, tristate T1 is enabled, allowing D_1 to flow to Y.

Shorting together the outputs of multiple gates technically violates the rules for combinational circuits given in Section 2.1. But because exactly one of the outputs is driven at any time, this exception is allowed.

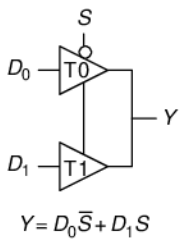


Figure 2.56 **Multiplexer using tristate buffers**

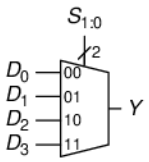


Figure 2.57 **4:1 multiplexer**

Wider Multiplexers

A 4:1 multiplexer has four data inputs and one output, as shown in Figure 2.57. Two select signals are needed to choose among the four data inputs. The 4:1 multiplexer can be built using sum-of-products logic, tristates, or multiple 2:1 multiplexers, as shown in Figure 2.58.

The product terms enabling the tristates can be formed using AND gates and inverters. They can also be formed using a decoder, which we will introduce in Section 2.8.2.

Wider multiplexers, such as 8:1 and 16:1 multiplexers, can be built by expanding the methods shown in Figure 2.58. In general, an N:1 multiplexer needs $\log_2 N$ select lines. Again, the best implementation choice depends on the target technology.

Multiplexer Logic

Multiplexers can be used as *lookup tables* to perform logic functions. Figure 2.59 shows a 4:1 multiplexer used to implement a two-input

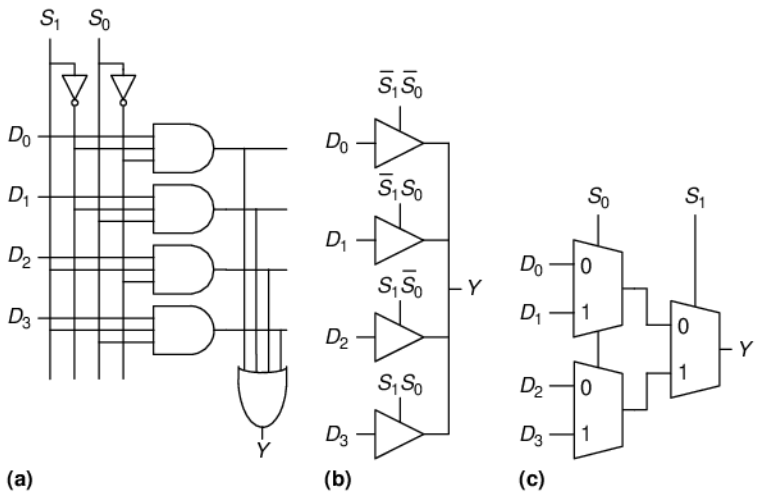


Figure 2.58 4:1 multiplexer implementations: (a) two-level logic, (b) tristates, (c) hierarchical

AND gate. The inputs, A and B, serve as select lines. The multiplexer data inputs are connected to 0 or 1 according to the corresponding row of the truth table. In general, a 2^N-input multiplexer can be programmed to perform any N-input logic function by applying 0's and 1's to the appropriate data inputs. Indeed, by changing the data inputs, the multiplexer can be reprogrammed to perform a different function.

With a little cleverness, we can cut the multiplexer size in half, using only a 2^{N-1}-input multiplexer to perform any N-input logic function. The strategy is to provide one of the literals, as well as 0's and 1's, to the multiplexer data inputs.

To illustrate this principle, Figure 2.60 shows two-input AND and XOR functions implemented with 2:1 multiplexers. We start with an ordinary truth table, and then combine pairs of rows to eliminate the rightmost input variable by expressing the output in terms of this variable. For example, in the case of AND, when $A = 0$, $Y = 0$, regardless of B. When $A = 1$, $Y = 0$ if $B = 0$ and $Y = 1$ if $B = 1$, so $Y = B$. We then use the multiplexer as a lookup table according to the new, smaller truth table.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$

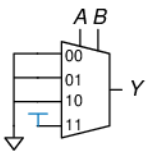


Figure 2.59 4:1 multiplexer implementation of two-input AND function

---

Example 2.12 LOGIC WITH MULTIPLEXERS

Alyssa P. Hacker needs to implement the function $Y = A\overline{B} + \overline{B}\overline{C} + \overline{A}BC$ to finish her senior project, but when she looks in her lab kit, the only part she has left is an 8:1 multiplexer. How does she implement the function?

Solution: Figure 2.61 shows Alyssa's implementation using a single 8:1 multiplexer. The multiplexer acts as a lookup table where each row in the truth table corresponds to a multiplexer input.

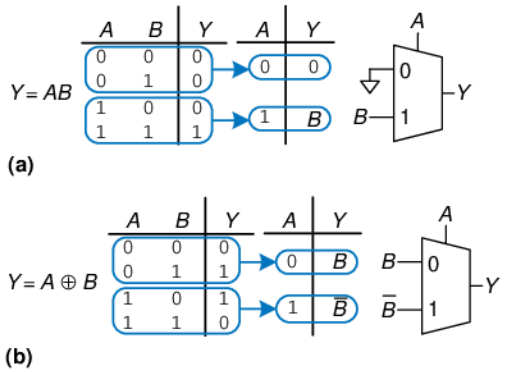


Figure 2.60 Multiplexer logic using variable inputs

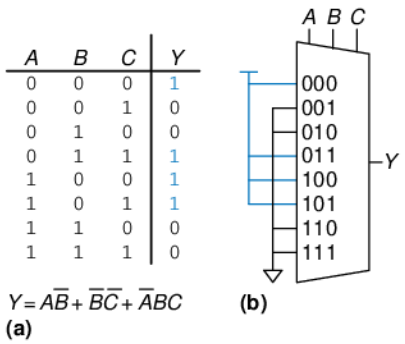


Figure 2.61 Alyssa's circuit: (a) truth table, (b) 8:1 multiplexer implementation

---

Example 2.13 LOGIC WITH MULTIPLEXERS, REPRISED

Alyssa turns on her circuit one more time before the final presentation and blows up the 8:1 multiplexer. (She accidently powered it with 20 V instead of 5 V after not sleeping all night.) She begs her friends for spare parts and they give her a 4:1 multiplexer and an inverter. Can she build her circuit with only these parts?

Solution: Alyssa reduces her truth table to four rows by letting the output depend on C. (She could also have chosen to rearrange the columns of the truth table to let the output depend on A or B.) Figure 2.62 shows the new design.

2.8.2 Decoders

A decoder has N inputs and 2^N outputs. It asserts exactly one of its outputs depending on the input combination. Figure 2.63 shows a 2:4 decoder. When $A_{1:0} = 00$, Y_0 is 1. When $A_{1:0} = 01$, Y_1 is 1. And so forth. The outputs are called *one-hot*, because exactly one is "hot" (HIGH) at a given time.

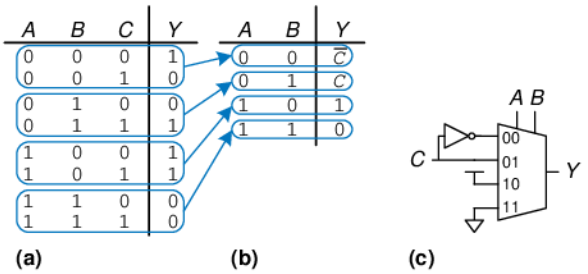
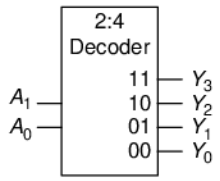


Figure 2.62 Alyssa's new circuit

Example 2.14 DECODER IMPLEMENTATION

Implement a 2:4 decoder with AND, OR, and NOT gates.

Solution: Figure 2.64 shows an implementation for the 2:4 decoder using four AND gates. Each gate depends on either the true or the complementary form of each input. In general, an N:2^N decoder can be constructed from 2^N N-input AND gates that accept the various combinations of true or complementary inputs. Each output in a decoder represents a single minterm. For example, Y_0 represents the minterm $\overline{A}_1\overline{A}_0$. This fact will be handy when using decoders with other digital building blocks.



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Figure 2.63 2:4 decoder

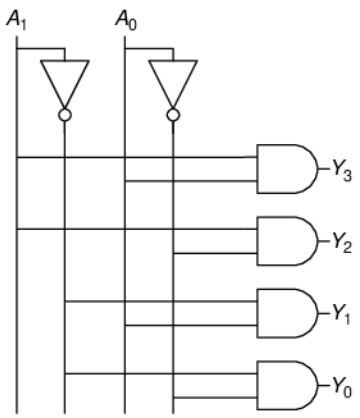


Figure 2.64 2:4 decoder implementation

Decoder Logic

Decoders can be combined with OR gates to build logic functions. Figure 2.65 shows the two-input XNOR function using a 2:4 decoder and a single OR gate. Because each output of a decoder represents a single minterm, the function is built as the OR of all the minterms in the function. In Figure 2.65, $Y = \overline{A}\,\overline{B} + AB = \overline{A \oplus B}$.

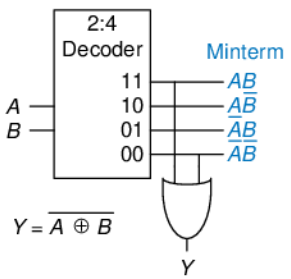


Figure 2.65 Logic function using decoder

When using decoders to build logic, it is easiest to express functions as a truth table or in canonical sum-of-products form. An N-input function with M 1's in the truth table can be built with an $N{:}2^N$ decoder and an M-input OR gate attached to all of the minterms containing 1's in the truth table. This concept will be applied to the building of Read Only Memories (ROMs) in Section 5.5.6.

2.9 TIMING

In previous sections, we have been concerned primarily with whether the circuit works—ideally, using the fewest gates. However, as any seasoned circuit designer will attest, one of the most challenging issues in circuit design is *timing:* making a circuit run fast.

An output takes time to change in response to an input change. Figure 2.66 shows the *delay* between an input change and the subsequent output change for a buffer. The figure is called a *timing diagram;* it portrays the *transient response* of the buffer circuit when an input changes. The transition from LOW to HIGH is called the *rising edge*. Similarly, the transition from HIGH to LOW (not shown in the figure) is called the *falling edge*. The blue arrow indicates that the rising edge of Y is caused by the rising edge of A. We measure delay from the *50% point* of the input signal, A, to the 50% point of the output signal, Y. The 50% point is the point at which the signal is half-way (50%) between its LOW and HIGH values as it transitions.

When designers speak of calculating the *delay* of a circuit, they generally are referring to the worst-case value (the propagation delay), unless it is clear otherwise from the context.

2.9.1 Propagation and Contamination Delay

Combinational logic is characterized by its *propagation delay* and *contamination delay*. The propagation delay, t_{pd}, is the maximum time from when an input changes until the output or outputs reach their final value. The contamination delay, t_{cd}, is the minimum time from when an input changes until any output starts to change its value.

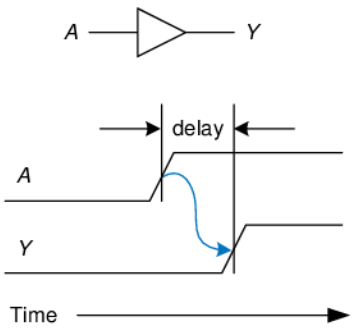


Figure 2.66 Circuit delay

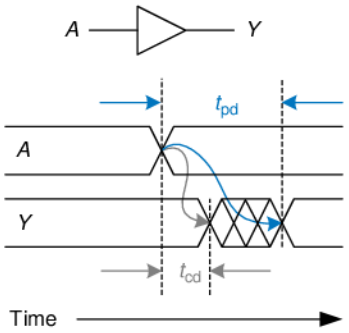


Figure 2.67 Propagation and contamination delay

Figure 2.67 illustrates a buffer's propagation delay and contamination delay in blue and gray, respectively. The figure shows that A is initially either HIGH or LOW and changes to the other state at a particular time; we are interested only in the fact that it changes, not what value it has. In response, Y changes some time later. The arcs indicate that Y may start to change t_{cd} after A transitions and that Y definitely settles to its new value within t_{pd}.

The underlying causes of delay in circuits include the time required to charge the capacitance in a circuit and the speed of light. t_{pd} and t_{cd} may be different for many reasons, including

- different rising and falling delays
- multiple inputs and outputs, some of which are faster than others
- circuits slowing down when hot and speeding up when cold

Circuit delays are ordinarily on the order of picoseconds (1 ps = 10^{-12} seconds) to nanoseconds (1 ns = 10^{-9} seconds). Trillions of picoseconds have elapsed in the time you spent reading this sidebar.

Calculating t_{pd} and t_{cd} requires delving into the lower levels of abstraction beyond the scope of this book. However, manufacturers normally supply data sheets specifying these delays for each gate.

Along with the factors already listed, propagation and contamination delays are also determined by the *path* a signal takes from input to output. Figure 2.68 shows a four-input logic circuit. The *critical path*, shown in blue, is the path from input A or B to output Y. It is the longest, and therefore the slowest, path, because the input travels

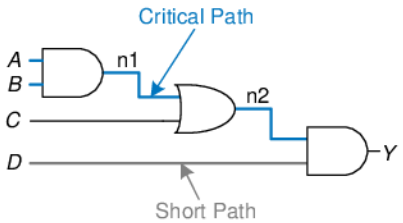


Figure 2.68 Short path and critical path

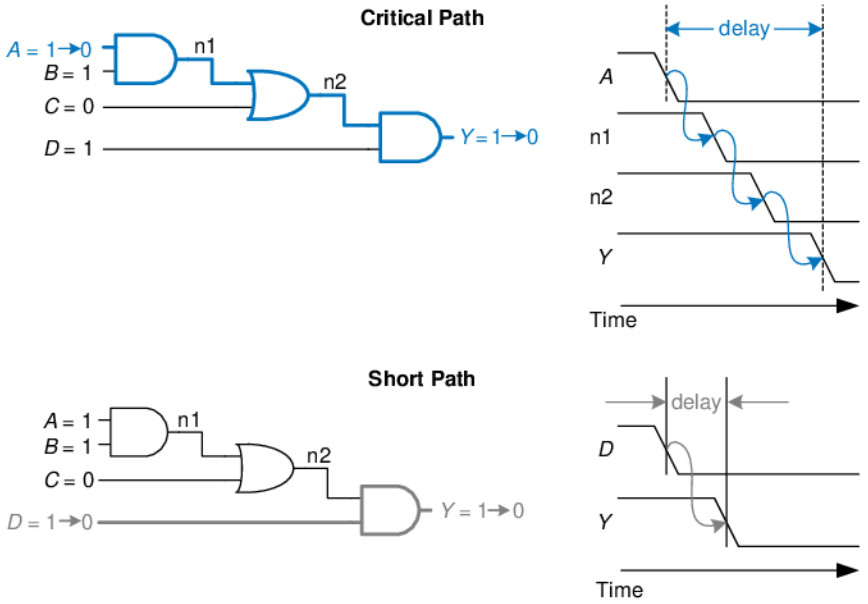


Figure 2.69 **Critical and short path waveforms**

through three gates to the output. This path is critical because it limits the speed at which the circuit operates. The *short path* through the circuit, shown in gray, is from input D to output Y. This is the shortest, and therefore the fastest, path through the circuit, because the input travels through only a single gate to the output.

The propagation delay of a combinational circuit is the sum of the propagation delays through each element on the critical path. The contamination delay is the sum of the contamination delays through each element on the short path. These delays are illustrated in Figure 2.69 and are described by the following equations:

$$t_{pd} = 2t_{pd\_\mathrm{AND}} + t_{pd\_\mathrm{OR}} \tag{2.7}$$

$$t_{cd} = t_{cd\_\mathrm{AND}} \tag{2.8}$$

Although we are ignoring wire delay in this analysis, digital circuits are now so fast that the delay of long wires can be as important as the delay of the gates. The speed of light delay in wires is covered in Appendix A.

Example 2.15 FINDING DELAYS

Ben Bitdiddle needs to find the propagation delay and contamination delay of the circuit shown in Figure 2.70. According to his data book, each gate has a propagation delay of 100 picoseconds (ps) and a contamination delay of 60 ps.

Solution: Ben begins by finding the critical path and the shortest path through the circuit. The critical path, highlighted in blue in Figure 2.71, is from input A

or B through three gates to the output, Y. Hence, t_{pd} is three times the propagation delay of a single gate, or 300 ps.

The shortest path, shown in gray in Figure 2.72, is from input C, D, or E through two gates to the output, Y. There are only two gates in the shortest path, so t_{cd} is 120 ps.

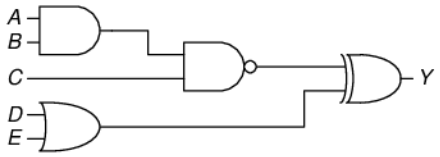


Figure 2.70 Ben's circuit

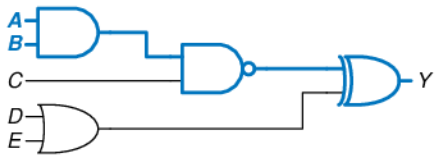


Figure 2.71 Ben's critical path

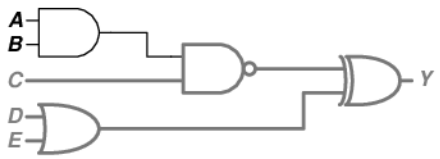


Figure 2.72 Ben's shortest path

Example 2.16 MULTIPLEXER TIMING: CONTROL-CRITICAL VS. DATA-CRITICAL

Compare the worst-case timing of the three four-input multiplexer designs shown in Figure 2.58 in Section 2.8.1. Table 2.7 lists the propagation delays for the components. What is the critical path for each design? Given your timing analysis, why might you choose one design over the other?

Solution: One of the critical paths for each of the three design options is highlighted in blue in Figures 2.73 and 2.74. t_{pd_sy} indicates the propagation delay from input S to output Y; t_{pd_dy} indicates the propagation delay from input D to output Y; t_{pd} is the worst of the two: $\max(t_{pd\_sy}, t_{pd\_dy})$.

For both the two-level logic and tristate implementations in Figure 2.73, the critical path is from one of the control signals, S, to the output, Y: $t_{pd} = t_{pd\_sy}$. These circuits are *control critical*, because the critical path is from the control signals to the output. Any additional delay in the control signals will add directly to the worst-case delay. The delay from D to Y in Figure 2.73(b) is only 50 ps, compared with the delay from S to Y of 125 ps.

Figure 2.74 shows the hierarchical implementation of the 4:1 multiplexer using two stages of 2:1 multiplexers. The critical path is from any of the D inputs to the output. This circuit is *data critical*, because the critical path is from the data input to the output: ($t_{pd} = t_{pd\_dy}$).

If data inputs arrive well before the control inputs, we would prefer the design with the shortest control-to-output delay (the hierarchical design in Figure 2.74). Similarly, if the control inputs arrive well before the data inputs, we would prefer the design with the shortest data-to-output delay (the tristate design in Figure 2.73(b)).

The best choice depends not only on the critical path through the circuit and the input arrival times, but also on the power, cost, and availability of parts.

Table 2.7 Timing specifications for multiplexer circuit elements

Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35

2.9.2 Glitches

So far we have discussed the case where a single input transition causes a single output transition. However, it is possible that a single input transition can cause *multiple* output transitions. These are called *glitches* or *hazards*. Although glitches usually don't cause problems, it is important to realize that they exist and recognize them when looking at timing diagrams. Figure 2.75 shows a circuit with a glitch and the Karnaugh map of the circuit.

Hazards have another meaning related to microarchitecture in Chapter 7, so we will stick with the term *glitches* for multiple output transitions to avoid confusion.

The Boolean equation is correctly minimized, but let's look at what happens when $A = 0$, $C = 1$, and B transitions from 1 to 0. Figure 2.76 (see page 90) illustrates this scenario. The short path (shown in gray) goes through two gates, the AND and OR gates. The critical path (shown in blue) goes through an inverter and two gates, the AND and OR gates.

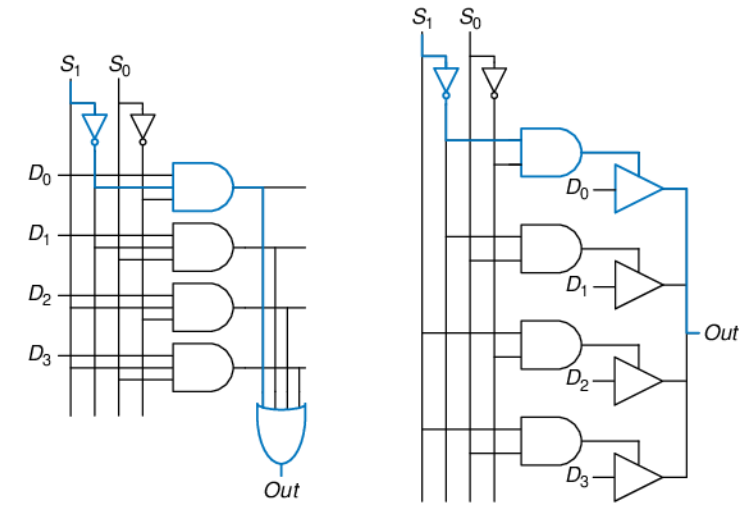


Figure 2.73 4:1 multiplexer propagation delays: (a) two-level logic, (b) tristate

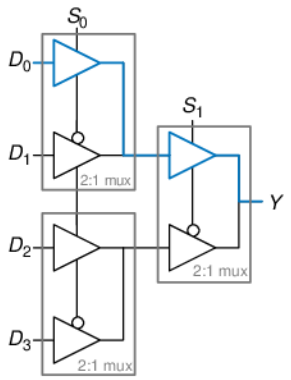


Figure 2.74 4:1 multiplexer propagation delays: hierarchical using 2:1 multiplexers

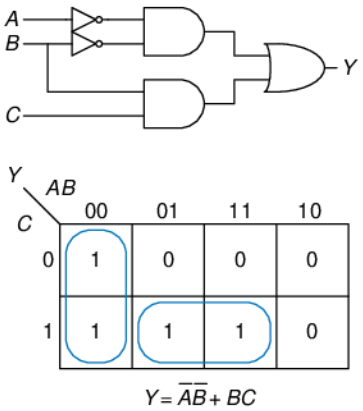


Figure 2.75 Circuit with a glitch

As B transitions from 1 to 0, n2 (on the short path) falls before n1 (on the critical path) can rise. Until n1 rises, the two inputs to the OR gate are 0, and the output Y drops to 0. When n1 eventually rises, Y returns to 1. As shown in the timing diagram of Figure 2.76, Y starts at 1 and ends at 1 but momentarily glitches to 0.

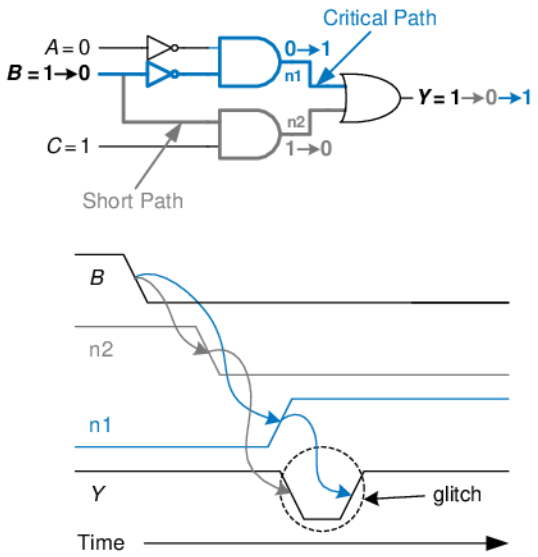


Figure 2.76 Timing of a glitch

As long as we wait for the propagation delay to elapse before we depend on the output, glitches are not a problem, because the output eventually settles to the right answer.

If we choose to, we can avoid this glitch by adding another gate to the implementation. This is easiest to understand in terms of the K-map. Figure 2.77 shows how an input transition on B from $ABC = 001$ to $ABC = 011$ moves from one prime implicant circle to another. The transition across the boundary of two prime implicants in the K-map indicates a possible glitch.

As we saw from the timing diagram in Figure 2.76, if the circuitry implementing one of the prime implicants turns *off* before the circuitry of the other prime implicant can turn *on*, there is a glitch. To fix this, we add another circle that *covers* that prime implicant boundary, as shown in Figure 2.78. You might recognize this as the consensus theorem, where the added term, $\overline{A}\,\overline{C}$, is the consensus or redundant term.

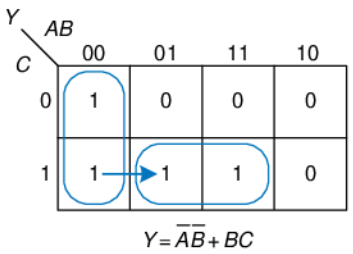


Figure 2.77 Input change crosses implicant boundary

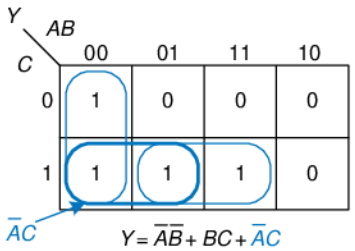


Figure 2.78 K-map without glitch

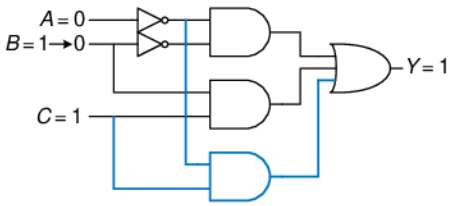


Figure 2.79 Circuit without glitch

Figure 2.79 shows the glitch-proof circuit. The added AND gate is highlighted in blue. Now a transition on B when $A = 0$ and $C = 1$ does not cause a glitch on the output, because the blue AND gate outputs 1 throughout the transition.

In general, a glitch can occur when a change in a single variable crosses the boundary between two prime implicants in a K-map. We can eliminate the glitch by adding redundant implicants to the K-map to cover these boundaries. This of course comes at the cost of extra hardware.

However, simultaneous transitions on multiple variables can also cause glitches. These glitches cannot be fixed by adding hardware. Because the vast majority of interesting systems have simultaneous (or near-simultaneous) transitions on multiple variables, glitches are a fact of life in most circuits. Although we have shown how to eliminate one kind of glitch, the point of discussing glitches is not to eliminate them but to be aware that they exist. This is especially important when looking at timing diagrams on a simulator or oscilloscope.

2.10 SUMMARY

A digital circuit is a module with discrete-valued inputs and outputs and a specification describing the function and timing of the module. This chapter has focused on combinational circuits, circuits whose outputs depend only on the current values of the inputs.

The function of a combinational circuit can be given by a truth table or a Boolean equation. The Boolean equation for any truth table can be obtained systematically using sum-of-products or product-of-sums form. In sum-of-products form, the function is written as the sum (OR) of one or more implicants. Implicants are the product (AND) of literals. Literals are the true or complementary forms of the input variables.

Boolean equations can be simplified using the rules of Boolean algebra. In particular, they can be simplified into minimal sum-of-products form by combining implicants that differ only in the true and complementary forms of one of the literals: $PA + P\overline{A} = P$. Karnaugh maps are a visual tool for minimizing functions of up to four variables. With practice, designers can usually simplify functions of a few variables by inspection. Computer-aided design tools are used for more complicated functions; such methods and tools are discussed in Chapter 4.

Logic gates are connected to create combinational circuits that perform the desired function. Any function in sum-of-products form can be built using two-level logic with the literals as inputs: NOT gates form the complementary literals, AND gates form the products, and OR gates form the sum. Depending on the function and the building blocks available, multilevel logic implementations with various types of gates may be more efficient. For example, CMOS circuits favor NAND and NOR gates because these gates can be built directly from CMOS transistors without requiring extra NOT gates. When using NAND and NOR gates, bubble pushing is helpful to keep track of the inversions.

Logic gates are combined to produce larger circuits such as multiplexers, decoders, and priority circuits. A multiplexer chooses one of the data inputs based on the select input. A decoder sets one of the outputs HIGH according to the input. A priority circuit produces an output indicating the highest priority input. These circuits are all examples of combinational building blocks. Chapter 5 will introduce more building blocks, including other arithmetic circuits. These building blocks will be used extensively to build a microprocessor in Chapter 7.

The timing specification of a combinational circuit consists of the propagation and contamination delays through the circuit. These indicate the longest and shortest times between an input change and the consequent output change. Calculating the propagation delay of a circuit involves identifying the critical path through the circuit, then adding up the propagation delays of each element along that path. There are many different ways to implement complicated combinational circuits; these ways offer trade-offs between speed and cost.

The next chapter will move to sequential circuits, whose outputs depend on previous as well as current values of the inputs. In other words, sequential circuits have *memory* of the past.

Exercises

Exercise 2.1 Write a Boolean equation in sum-of-products canonical form for each of the truth tables in Figure 2.80.

(a)

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(c)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(d)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

(e)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 2.80 Truth tables

Exercise 2.2 Write a Boolean equation in product-of-sums canonical form for the truth tables in Figure 2.80.

Exercise 2.3 Minimize each of the Boolean equations from Exercise 2.1.

Exercise 2.4 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.3. Reasonably simple means that you are not wasteful of gates, but you don't waste vast amounts of time checking every possible implementation of the circuit either.

Exercise 2.5 Repeat Exercise 2.4 using only NOT gates and AND and OR gates.

Exercise 2.6 Repeat Exercise 2.4 using only NOT gates and NAND and NOR gates.

Exercise 2.7 Simplify the following Boolean equations using Boolean theorems. Check for correctness using a truth table or K-map.

(a) $Y = AC + \overline{A}\,\overline{B}C$

(b) $Y = \overline{A}\,\overline{B} + \overline{A}B\overline{C} + \overline{(A + \overline{C})}$

(c) $Y = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C} + A\overline{B}C\overline{D} + ABD + \overline{A}\,\overline{B}C\overline{D} + B\overline{C}D + \overline{A}$

Exercise 2.8 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.7.

Exercise 2.9 Simplify each of the following Boolean equations. Sketch a reasonably simple combinational circuit implementing the simplified equation.

(a) $Y = BC + \overline{A}\overline{B}\overline{C} + B\overline{C}$

(b) $Y = \overline{A + \overline{A}B + \overline{A}\overline{B}} + \overline{A + \overline{B}}$

(c) $Y = ABC + ABD + ABE + ACD + ACE + (\overline{A + D + E}) + \overline{B}\,\overline{C}D + \overline{B}\,\overline{C}E + \overline{B}\,\overline{D}\,\overline{E} + \overline{C}\,\overline{D}\,\overline{E}$

Exercise 2.10 Give an example of a truth table requiring between 3 billion and 5 billion rows that can be constructed using fewer than 40 (but at least 1) two-input gates.

Exercise 2.11 Give an example of a circuit with a cyclic path that is nevertheless combinational.

Exercise 2.12 Alyssa P. Hacker says that any Boolean function can be written in minimal sum-of-products form as the sum of all of the prime implicants of the function. Ben Bitdiddle says that there are some functions whose minimal equation does not involve all of the prime implicants. Explain why Alyssa is right or provide a counterexample demonstrating Ben's point.

Exercise 2.13 Prove that the following theorems are true using perfect induction. You need not prove their duals.

(a) The idempotency theorem (T3)

(b) The distributivity theorem (T8)

(c) The combining theorem (T10)

Exercise 2.14 Prove De Morgan's Theorem (T12) for three variables, B_2, B_1, B_0, using perfect induction.

Exercise 2.15 Write Boolean equations for the circuit in Figure 2.81. You need not minimize the equations.

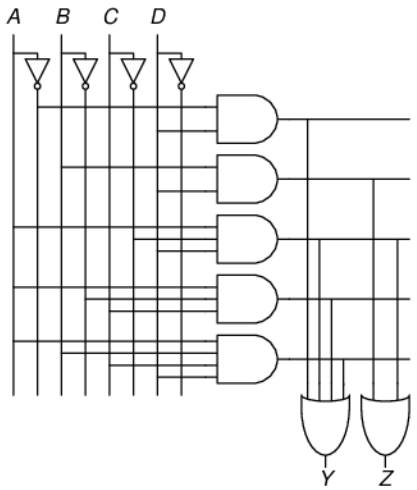


Figure 2.81 Circuit schematic

Exercise 2.16 Minimize the Boolean equations from Exercise 2.15 and sketch an improved circuit with the same function.

Exercise 2.17 Using De Morgan equivalent gates and bubble pushing methods, redraw the circuit in Figure 2.82 so that you can find the Boolean equation by inspection. Write the Boolean equation.

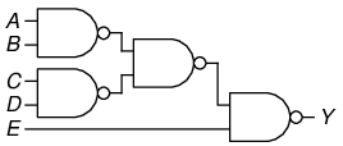


Figure 2.82 Circuit schematic

Exercise 2.18 Repeat Exercise 2.17 for the circuit in Figure 2.83.

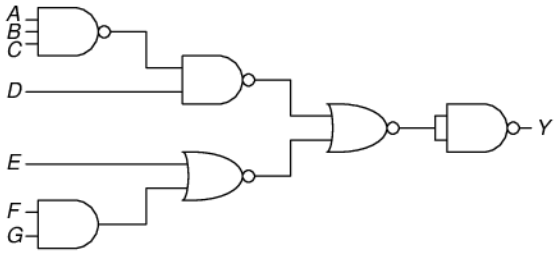


Figure 2.83 Circuit schematic

Exercise 2.19 Find a minimal Boolean equation for the function in Figure 2.84. Remember to take advantage of the don't care entries.

A	B	C	D	Y
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	0
0	1	0	0	0
0	1	0	1	X
0	1	1	0	0
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Figure 2.84 Truth table

Exercise 2.20 Sketch a circuit for the function from Exercise 2.19.

Exercise 2.21 Does your circuit from Exercise 2.20 have any potential glitches when one of the inputs changes? If not, explain why not. If so, show how to modify the circuit to eliminate the glitches.

Exercise 2.22 Ben Bitdiddle will enjoy his picnic on sunny days that have no ants. He will also enjoy his picnic any day he sees a hummingbird, as well as on days where there are ants and ladybugs. Write a Boolean equation for his enjoyment (E) in terms of sun (S), ants (A), hummingbirds (H), and ladybugs (L).

Exercise 2.23 Complete the design of the seven-segment decoder segments S_c through S_g (see Example 2.10):

(a) Derive Boolean equations for the outputs S_c through S_g assuming that inputs greater than 9 must produce blank (0) outputs.

(b) Derive Boolean equations for the outputs S_c through S_g assuming that inputs greater than 9 are don't cares.

(c) Sketch a reasonably simple gate-level implementation of part (b). Multiple outputs can share gates where appropriate.

Exercise 2.24 A circuit has four inputs and two outputs. The inputs, $A_{3:0}$, represent a number from 0 to 15. Output P should be TRUE if the number is prime (0 and 1 are not prime, but 2, 3, 5, and so on, are prime). Output D should be TRUE if the number is divisible by 3. Give simplified Boolean equations for each output and sketch a circuit.

Exercise 2.25 A *priority encoder* has 2^N inputs. It produces an N-bit binary output indicating the most significant bit of the input that is TRUE, or 0 if none of the inputs are TRUE. It also produces an output *NONE* that is TRUE if none of the input bits are TRUE. Design an eight-input priority encoder with inputs $A_{7:0}$ and outputs $Y_{2:0}$ and *NONE*. For example, if the input is 00100000, the output Y should be 101 and *NONE* should be 0. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.26 Design a modified priority encoder (see Exercise 2.25) that receives an 8-bit input, $A_{7:0}$, and produces two 3-bit outputs, $Y_{2:0}$ and $Z_{2:0}$. Y indicates the most significant bit of the input that is TRUE. Z indicates the second most significant bit of the input that is TRUE. Y should be 0 if none of the inputs are TRUE. Z should be 0 if no more than one of the inputs is TRUE. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.27 An M-bit *thermometer code* for the number k consists of k 1's in the least significant bit positions and $M - k$ 0's in all the more significant bit positions. A *binary-to-thermometer code converter* has N inputs and 2^N-1 outputs. It produces a 2^N-1 bit thermometer code for the number specified by the input. For example, if the input is 110, the output should be 0111111. Design a 3:7 binary-to-thermometer code converter. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.28 Write a minimized Boolean equation for the function performed by the circuit in Figure 2.85.

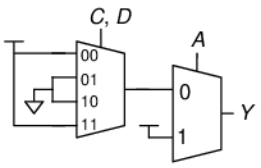


Figure 2.85 Multiplexer circuit

Exercise 2.29 Write a minimized Boolean equation for the function performed by the circuit in Figure 2.86.

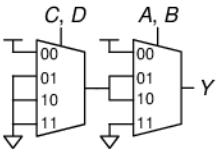


Figure 2.86 Multiplexer circuit

Exercise 2.30 Implement the function from Figure 2.80(b) using

(a) an 8:1 multiplexer

(b) a 4:1 multiplexer and one inverter

(c) a 2:1 multiplexer and two other logic gates

Exercise 2.31 Implement the function from Exercise 2.9(a) using

(a) an 8:1 multiplexer

(b) a 4:1 multiplexer and no other gates

(c) a 2:1 multiplexer, one OR gate, and an inverter

Exercise 2.32 Determine the propagation delay and contamination delay of the circuit in Figure 2.83. Use the gate delays given in Table 2.8.

Table 2.8 Gate delays for Exercises 2.32–2.35

Gate	t_{pd} (ps)	t_{cd} (ps)
NOT	15	10
2-input NAND	20	15
3-input NAND	30	25
2-input NOR	30	25
3-input NOR	45	35
2-input AND	30	25
3-input AND	40	30
2-input OR	40	30
3-input OR	55	45
2-input XOR	60	40

Exercise 2.33 Sketch a schematic for a fast 3:8 decoder. Suppose gate delays are given in Table 2.8 (and only the gates in that table are available). Design your decoder to have the shortest possible critical path, and indicate what that path is. What are its propagation delay and contamination delay?

Exercise 2.34 Redesign the circuit from Exercise 2.24 to be as fast as possible. Use only the gates from Table 2.8. Sketch the new circuit and indicate the critical path. What are its propagation delay and contamination delay?

Exercise 2.35 Redesign the priority encoder from Exercise 2.25 to be as fast as possible. You may use any of the gates from Table 2.8. Sketch the new circuit and indicate the critical path. What are its propagation delay and contamination delay?

Exercise 2.36 Design an 8:1 multiplexer with the shortest possible delay from the data inputs to the output. You may use any of the gates from Table 2.7 on page 88. Sketch a schematic. Using the gate delays from the table, determine this delay.

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 2.1 Sketch a schematic for the two-input XOR function using only NAND gates. How few can you use?

Question 2.2 Design a circuit that will tell whether a given month has 31 days in it. The month is specified by a 4-bit input, $A_{3:0}$. For example, if the inputs are 0001, the month is January, and if the inputs are 1100, the month is December. The circuit output, Y, should be HIGH only when the month specified by the inputs has 31 days in it. Write the simplified equation, and draw the circuit diagram using a minimum number of gates. (Hint: Remember to take advantage of don't cares.)

Question 2.3 What is a tristate buffer? How and why is it used?

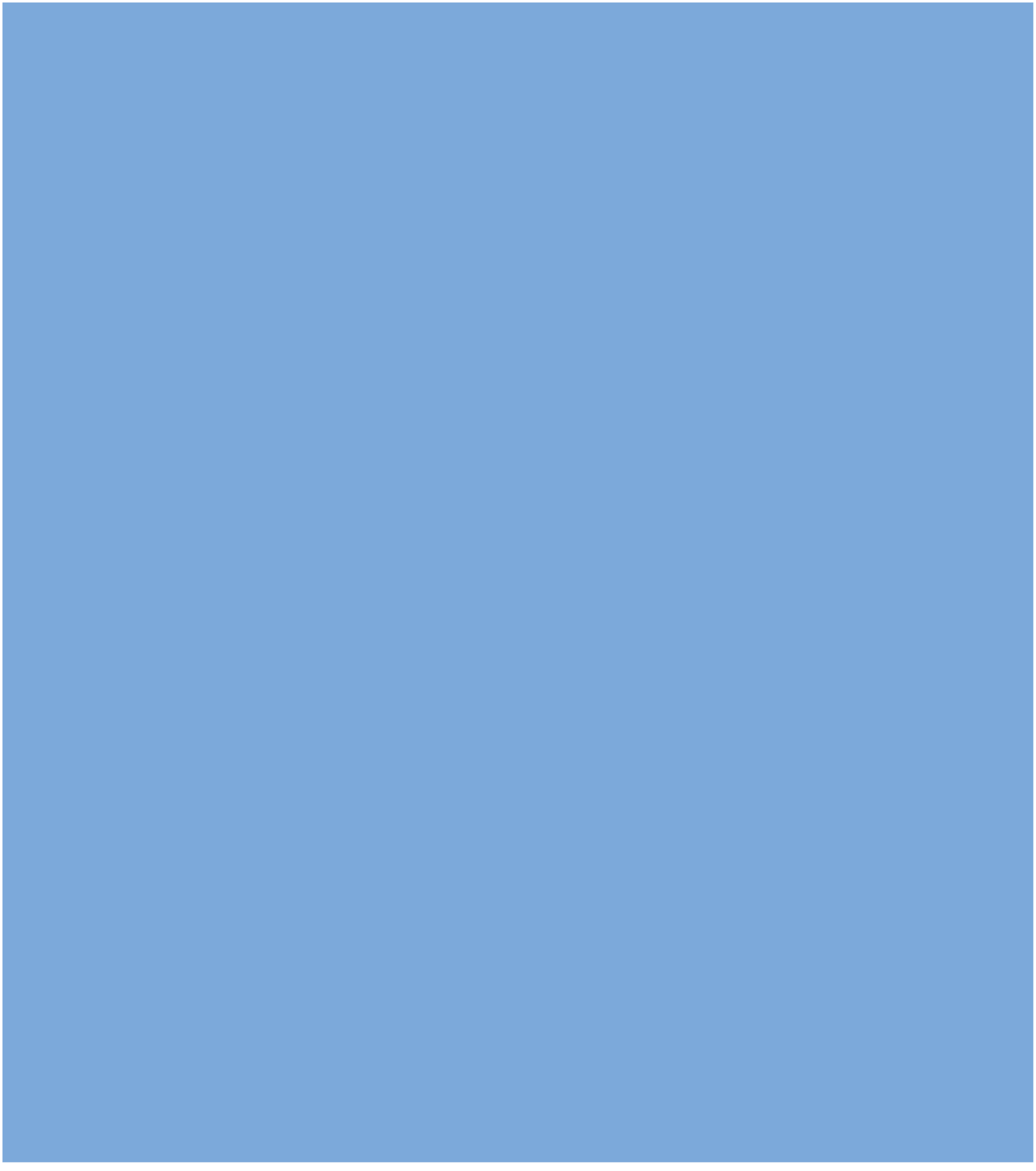
Question 2.4 A gate or set of gates is universal if it can be used to construct any Boolean function. For example, the set {AND, OR, NOT} is universal.

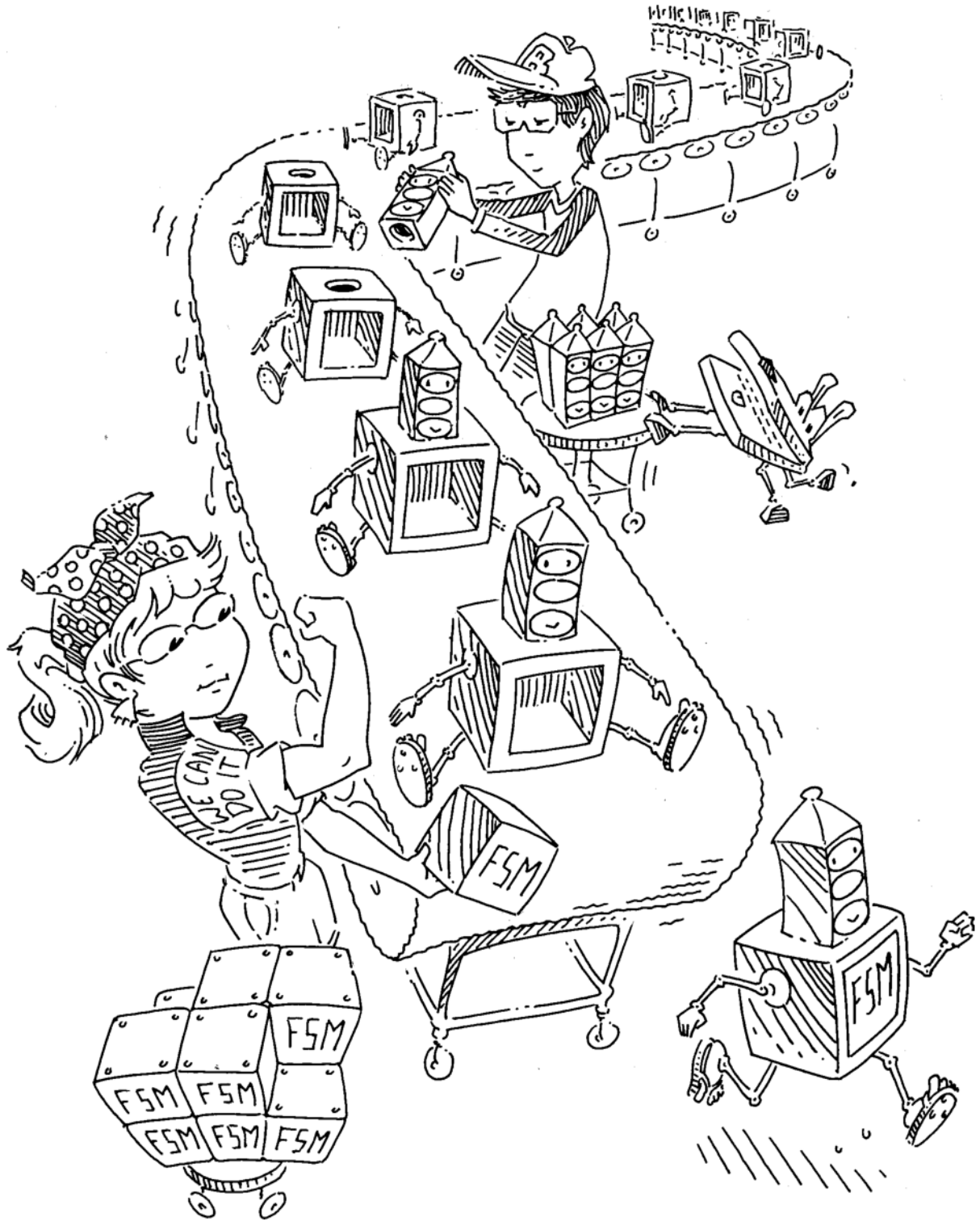
(a) Is an AND gate by itself universal? Why or why not?

(b) Is the set {OR, NOT} universal? Why or why not?

(c) Is a NAND gate by itself universal? Why or why not?

Question 2.5 Explain why a circuit's contamination delay might be less than (instead of equal to) its propagation delay.




WE CAN DO IT
FSM
FSM
FSM
FSM
FSM
FSM
FSM
FSM

Sequential Logic Design

3

3.1 INTRODUCTION

In the last chapter, we showed how to analyze and design combinational logic. The output of combinational logic depends only on current input values. Given a specification in the form of a truth table or Boolean equation, we can create an optimized circuit to meet the specification.

In this chapter, we will analyze and design *sequential* logic. The outputs of sequential logic depend on both current and prior input values. Hence, sequential logic has memory. Sequential logic might explicitly remember certain previous inputs, or it might distill the prior inputs into a smaller amount of information called the *state* of the system. The state of a digital sequential circuit is a set of bits called *state variables* that contain all the information about the past necessary to explain the future behavior of the circuit.

The chapter begins by studying latches and flip-flops, which are simple sequential circuits that store one bit of state. In general, sequential circuits are complicated to analyze. To simplify design, we discipline ourselves to build only synchronous sequential circuits consisting of combinational logic and banks of flip-flops containing the state of the circuit. The chapter describes finite state machines, which are an easy way to design sequential circuits. Finally, we analyze the speed of sequential circuits and discuss parallelism as a way to increase clock speed.

3.2 LATCHES AND FLIP-FLOPS

The fundamental building block of memory is a *bistable* element, an element with two stable states. Figure 3.1(a) shows a simple bistable element consisting of a pair of inverters connected in a loop. Figure 3.1(b) shows the same circuit redrawn to emphasize the symmetry. The inverters are *cross-coupled*, meaning that the input of I1 is the output of I2 and vice versa. The circuit has no inputs, but it does have two outputs,

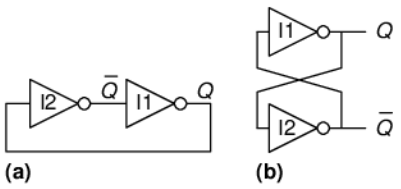


Figure 3.1 Cross-coupled inverter pair

Just as Y is commonly used for the output of combinational logic, Q is commonly used for the output of sequential logic.

Q and $\overline{Q}$. Analyzing this circuit is different from analyzing a combinational circuit because it is cyclic: Q depends on $\overline{Q}$, and $\overline{Q}$ depends on Q.

Consider the two cases, Q is 0 or Q is 1. Working through the consequences of each case, we have:

- *Case I:* $Q = 0$
  As shown in Figure 3.2(a), I2 receives a FALSE input, Q, so it produces a TRUE output on $\overline{Q}$. I1 receives a TRUE input, $\overline{Q}$, so it produces a FALSE output on Q. This is consistent with the original assumption that $Q = 0$, so the case is said to be *stable*.
- *Case II:* $Q = 1$
  As shown in Figure 3.2(b), I2 receives a TRUE input and produces a FALSE output on $\overline{Q}$. I1 receives a FALSE input and produces a TRUE output on Q. This is again stable.

Because the cross-coupled inverters have two stable states, $Q = 0$ and $Q = 1$, the circuit is said to be bistable. A subtle point is that the circuit has a third possible state with both outputs approximately halfway between 0 and 1. This is called a *metastable* state and will be discussed in Section 3.5.4.

An element with N stable states conveys $\log_2 N$ bits of information, so a bistable element stores one bit. The state of the cross-coupled inverters is contained in one binary state variable, Q. The value of Q tells us everything about the past that is necessary to explain the future behavior of the circuit. Specifically, if $Q = 0$, it will remain 0 forever, and if $Q = 1$, it will remain 1 forever. The circuit does have another node, $\overline{Q}$, but $\overline{Q}$ does not contain any additional information because if Q is known, $\overline{Q}$ is also known. On the other hand, $\overline{Q}$ is also an acceptable choice for the state variable.

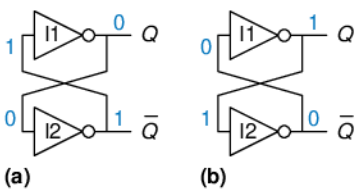


Figure 3.2 Bistable operation of cross-coupled inverters

When power is first applied to a sequential circuit, the initial state is unknown and usually unpredictable. It may differ each time the circuit is turned on.

Although the cross-coupled inverters can store a bit of information, they are not practical because the user has no inputs to control the state. However, other bistable elements, such as *latches* and *flip-flops*, provide inputs to control the value of the state variable. The remainder of this section considers these circuits.

3.2.1 SR Latch

One of the simplest sequential circuits is the *SR latch*, which is composed of two cross-coupled NOR gates, as shown in Figure 3.3. The latch has two inputs, S and R, and two outputs, Q and $\overline{Q}$. The SR latch is similar to the cross-coupled inverters, but its state can be controlled through the S and R inputs, which *set* and *reset* the output Q.

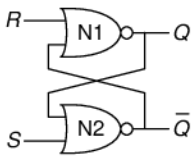


Figure 3.3 SR latch schematic

A good way to understand an unfamiliar circuit is to work out its truth table, so that is where we begin. Recall that a NOR gate produces a FALSE output when either input is TRUE. Consider the four possible combinations of R and S.

- *Case I:* $R = 1, S = 0$
  N1 sees at least one TRUE input, R, so it produces a FALSE output on Q. N2 sees both Q and S FALSE, so it produces a TRUE output on $\overline{Q}$.
- *Case II:* $R = 0, S = 1$
  N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output Q. N2 receives at least one TRUE input, S, so it produces a FALSE output on $\overline{Q}$. Now we can revisit N1, knowing that both inputs are FALSE, so the output Q is TRUE.
- *Case III:* $R = 1, S = 1$
  N1 and N2 both see at least one TRUE input (R or S), so each produces a FALSE output. Hence Q and $\overline{Q}$ are both FALSE.
- *Case IV:* $R = 0, S = 0$
  N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output. N2 receives inputs of 0 and Q. Because we don't yet know Q, we can't determine the output. Now we are stuck. This is reminiscent of the cross-coupled inverters. But we know that Q must either be 0 or 1. So we can solve the problem by checking what happens in each of these subcases.

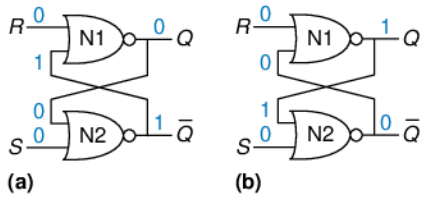


Figure 3.4 Bistable states of SR latch

- *Case IVa:* $Q = 0$
  Because S and Q are FALSE, N2 produces a TRUE output on $\overline{Q}$, as shown in Figure 3.4(a). Now N1 receives one TRUE input, $\overline{Q}$, so its output, Q, is FALSE, just as we had assumed.
- *Case IVb:* $Q = 1$
  Because Q is TRUE, N2 produces a FALSE output on $\overline{Q}$, as shown in Figure 3.4(b). Now N1 receives two FALSE inputs, R and $\overline{Q}$, so its output, Q, is TRUE, just as we had assumed.

Putting this all together, suppose Q has some known prior value, which we will call Q_{prev}, before we enter Case IV. Q_{prev} is either 0 or 1, and represents the state of the system. When R and S are 0, Q will remember this old value, Q_{prev}, and $\overline{Q}$ will be its complement, $\overline{Q}_{prev}$. This circuit has memory.

Case	S	R	Q	$\overline{Q}$
IV	0	0	Q_{prev}	$\overline{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Figure 3.5 SR latch truth table

The truth table in Figure 3.5 summarizes these four cases. The inputs S and R stand for *Set* and *Reset*. To *set* a bit means to make it TRUE. To *reset* a bit means to make it FALSE. The outputs, Q and $\overline{Q}$, are normally complementary. When R is asserted, Q is reset to 0 and $\overline{Q}$ does the opposite. When S is asserted, Q is set to 1 and $\overline{Q}$ does the opposite. When neither input is asserted, Q remembers its old value, Q_{prev}. Asserting both S and R simultaneously doesn't make much sense because it means the latch should be set and reset at the same time, which is impossible. The poor confused circuit responds by making both outputs 0.

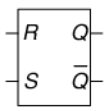


Figure 3.6 SR latch symbol

The SR latch is represented by the symbol in Figure 3.6. Using the symbol is an application of abstraction and modularity. There are various ways to build an SR latch, such as using different logic gates or transistors. Nevertheless, any circuit element with the relationship specified by the truth table in Figure 3.5 and the symbol in Figure 3.6 is called an SR latch.

Like the cross-coupled inverters, the SR latch is a bistable element with one bit of state stored in Q. However, the state can be controlled through the S and R inputs. When R is asserted, the state is reset to 0. When S is asserted, the state is set to 1. When neither is asserted, the state retains its old value. Notice that the entire history of inputs can be

accounted for by the single state variable Q. No matter what pattern of setting and resetting occurred in the past, all that is needed to predict the future behavior of the SR latch is whether it was most recently set or reset.

3.2.2 D Latch

The SR latch is awkward because it behaves strangely when both S and R are simultaneously asserted. Moreover, the S and R inputs conflate the issues of *what* and *when*. Asserting one of the inputs determines not only *what* the state should be but also *when* it should change. Designing circuits becomes easier when these questions of what and when are separated. The D latch in Figure 3.7(a) solves these problems. It has two inputs. The *data* input, D, controls what the next state should be. The *clock* input, CLK, controls when the state should change.

Again, we analyze the latch by writing the truth table, given in Figure 3.7(b). For convenience, we first consider the internal nodes $\overline{D}$, S, and R. If $CLK = 0$, both S and R are FALSE, regardless of the value of D. If $CLK = 1$, one AND gate will produce TRUE and the other FALSE, depending on the value of D. Given S and R, Q and $\overline{Q}$ are determined using Figure 3.5. Observe that when $CLK = 0$, Q remembers its old value, Q_{prev}. When $CLK = 1$, $Q = D$. In all cases, $\overline{Q}$ is the complement of Q, as would seem logical. The D latch avoids the strange case of simultaneously asserted R and S inputs.

Putting it all together, we see that the clock controls when data flows through the latch. When $CLK = 1$, the latch is *transparent*. The data at D flows through to Q as if the latch were just a buffer. When $CLK = 0$, the latch is *opaque*. It blocks the new data from flowing through to Q, and Q retains the old value. Hence, the D latch is sometimes called a *transparent latch* or a *level-sensitive* latch. The D latch symbol is given in Figure 3.7(c).

The D latch updates its state continuously while $CLK = 1$. We shall see later in this chapter that it is useful to update the state only at a specific instant in time. The D flip-flop described in the next section does just that.

Some people call a latch open or closed rather than transparent or opaque. However, we think those terms are ambiguous—does *open* mean transparent like an open door, or opaque, like an open circuit?

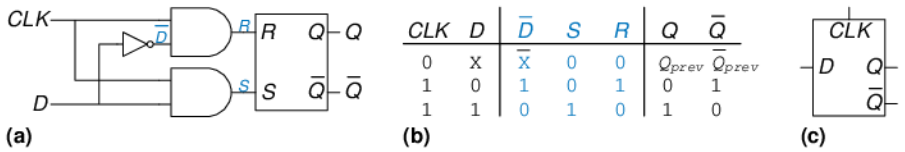


Figure 3.7 D latch: (a) schematic, (b) truth table, (c) symbol

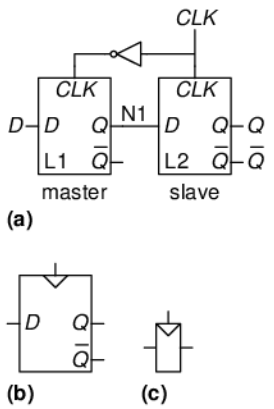


Figure 3.8 D flip-flop: (a) schematic, (b) symbol, (c) condensed symbol

The precise distinction between *flip-flops* and *latches* is somewhat muddled and has evolved over time. In common industry usage, a flip-flop is *edge-triggered*. In other words, it is a bistable element with a *clock* input. The state of the flip-flop changes only in response to a clock edge, such as when the clock rises from 0 to 1. Bistable elements without an edge-triggered clock are commonly called latches.

The term flip-flop or latch by itself usually refers to a *D flip-flop* or *D latch*, respectively, because these are the types most commonly used in practice.

3.2.3 D Flip-Flop

A *D flip-flop* can be built from two back-to-back D latches controlled by complementary clocks, as shown in Figure 3.8(a). The first latch, L1, is called the *master*. The second latch, L2, is called the *slave*. The node between them is named N1. A symbol for the D flip-flop is given in Figure 3.8(b). When the $\overline{Q}$ output is not needed, the symbol is often condensed as in Figure 3.8(c).

When $CLK = 0$, the master latch is transparent and the slave is opaque. Therefore, whatever value was at D propagates through to N1. When $CLK = 1$, the master goes opaque and the slave becomes transparent. The value at N1 propagates through to Q, but N1 is cut off from D. Hence, whatever value was at D immediately before the clock rises from 0 to 1 gets copied to Q immediately after the clock rises. At all other times, Q retains its old value, because there is always an opaque latch blocking the path between D and Q.

In other words, *a D flip-flop copies D to Q on the rising edge of the clock, and remembers its state at all other times*. Reread this definition until you have it memorized; one of the most common problems for beginning digital designers is to forget what a flip-flop does. The rising edge of the clock is often just called the *clock edge* for brevity. The D input specifies what the new state will be. The clock edge indicates when the state should be updated.

A D flip-flop is also known as a *master-slave flip-flop*, an *edge-triggered flip-flop*, or a *positive edge-triggered flip-flop*. The triangle in the symbols denotes an edge-triggered clock input. The $\overline{Q}$ output is often omitted when it is not needed.

Example 3.1 FLIP-FLOP TRANSISTOR COUNT

How many transistors are needed to build the D flip-flop described in this section?

Solution: A NAND or NOR gate uses four transistors. A NOT gate uses two transistors. An AND gate is built from a NAND and a NOT, so it uses six transistors. The SR latch uses two NOR gates, or eight transistors. The D latch uses an SR latch, two AND gates, and a NOT gate, or 22 transistors. The D flip-flop uses two D latches and a NOT gate, or 46 transistors. Section 3.2.7 describes a more efficient CMOS implementation using transmission gates.

3.2.4 Register

An N-bit register is a bank of N flip-flops that share a common CLK input, so that all bits of the register are updated at the same time.

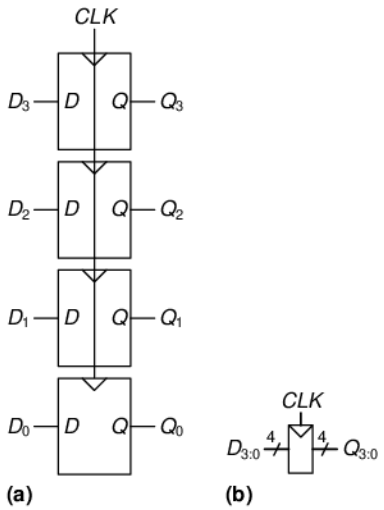


Figure 3.9 A 4-bit register: (a) schematic and (b) symbol

Registers are the key building block of most sequential circuits. Figure 3.9 shows the schematic and symbol for a four-bit register with inputs $D_{3:0}$ and outputs $Q_{3:0}$. $D_{3:0}$ and $Q_{3:0}$ are both 4-bit busses.

3.2.5 Enabled Flip-Flop

An *enabled flip-flop* adds another input called *EN* or ENABLE to determine whether data is loaded on the clock edge. When *EN* is TRUE, the enabled flip-flop behaves like an ordinary D flip-flop. When *EN* is FALSE, the enabled flip-flop ignores the clock and retains its state. Enabled flip-flops are useful when we wish to load a new value into a flip-flop only some of the time, rather than on every clock edge.

Figure 3.10 shows two ways to construct an enabled flip-flop from a D flip-flop and an extra gate. In Figure 3.10(a), an input multiplexer chooses whether to pass the value at *D*, if *EN* is TRUE, or to recycle the old state from *Q*, if *EN* is FALSE. In Figure 3.10(b), the clock is *gated*.

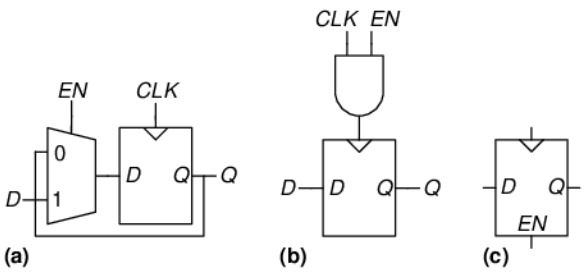


Figure 3.10 Enabled flip-flop: (a, b) schematics, (c) symbol

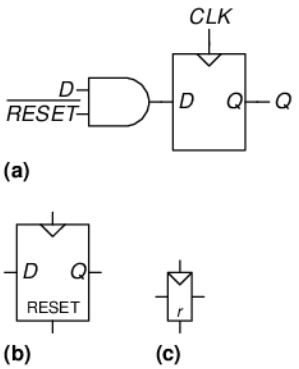


Figure 3.11 Synchronously resettable flip-flop: (a) schematic, (b, c) symbols

If *EN* is TRUE, the *CLK* input to the flip-flop toggles normally. If *EN* is FALSE, the *CLK* input is also FALSE and the flip-flop retains its old value. Notice that *EN* must not change while *CLK* = 1, lest the flip-flop see a clock *glitch* (switch at an incorrect time). Generally, performing logic on the clock is a bad idea. Clock gating delays the clock and can cause timing errors, as we will see in Section 3.5.3, so do it only if you are sure you know what you are doing. The symbol for an enabled flip-flop is given in Figure 3.10(c).

3.2.6 Resettable Flip-Flop

A *resettable flip-flop* adds another input called RESET. When RESET is FALSE, the resettable flip-flop behaves like an ordinary D flip-flop. When RESET is TRUE, the resettable flip-flop ignores *D* and resets the output to 0. Resettable flip-flops are useful when we want to force a known state (i.e., 0) into all the flip-flops in a system when we first turn it on.

Such flip-flops may be *synchronously* or *asynchronously resettable*. Synchronously resettable flip-flops reset themselves only on the rising edge of *CLK*. Asynchronously resettable flip-flops reset themselves as soon as RESET becomes TRUE, independent of *CLK*.

Figure 3.11(a) shows how to construct a synchronously resettable flip-flop from an ordinary D flip-flop and an AND gate. When $\overline{\text{RESET}}$ is FALSE, the AND gate forces a 0 into the input of the flip-flop. When $\overline{\text{RESET}}$ is TRUE, the AND gate passes *D* to the flip-flop. In this example, $\overline{\text{RESET}}$ is an *active low* signal, meaning that the reset signal performs its function when it is 0, not 1. By adding an inverter, the circuit could have accepted an active high RESET signal instead. Figures 3.11(b) and 3.11(c) show symbols for the resettable flip-flop with active high RESET.

Asynchronously resettable flip-flops require modifying the internal structure of the flip-flop and are left to you to design in Exercise 3.10; however, they are frequently available to the designer as a standard component.

As you might imagine, settable flip-flops are also occasionally used. They load a 1 into the flip-flop when SET is asserted, and they too come in synchronous and asynchronous flavors. Resettable and settable flip-flops may also have an enable input and may be grouped into *N*-bit registers.

3.2.7 Transistor-Level Latch and Flip-Flop Designs*

Example 3.1 showed that latches and flip-flops require a large number of transistors when built from logic gates. But the fundamental role of a

latch is to be transparent or opaque, much like a switch. Recall from Section 1.7.7 that a transmission gate is an efficient way to build a CMOS switch, so we might expect that we could take advantage of transmission gates to reduce the transistor count.

A compact D latch can be constructed from a single transmission gate, as shown in Figure 3.12(a). When $CLK = 1$ and $\overline{CLK} = 0$, the transmission gate is ON, so D flows to Q and the latch is transparent. When $CLK = 0$ and $\overline{CLK} = 1$, the transmission gate is OFF, so Q is isolated from D and the latch is opaque. This latch suffers from two major limitations:

- *Floating output node:* When the latch is opaque, Q is not held at its value by any gates. Thus Q is called a *floating* or *dynamic* node. After some time, noise and charge leakage may disturb the value of Q.
- *No buffers:* The lack of buffers has caused malfunctions on several commercial chips. A spike of noise that pulls D to a negative voltage can turn on the nMOS transistor, making the latch transparent, even when $CLK = 0$. Likewise, a spike on D above V_{DD} can turn on the pMOS transistor even when $CLK = 0$. And the transmission gate is symmetric, so it could be driven backward with noise on Q affecting the input D. The general rule is that neither the input of a transmission gate nor the state node of a sequential circuit should ever be exposed to the outside world, where noise is likely.

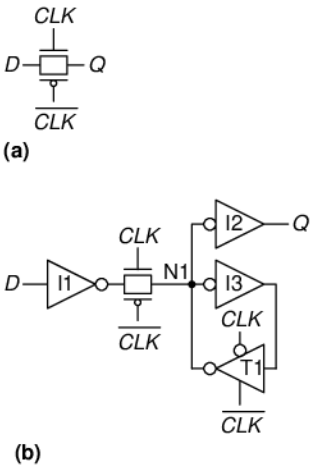


Figure 3.12 D latch schematic

Figure 3.12(b) shows a more robust 12-transistor D latch used on modern commercial chips. It is still built around a clocked transmission gate, but it adds inverters I1 and I2 to buffer the input and output. The state of the latch is held on node N1. Inverter I3 and the tristate buffer, T1, provide feedback to turn N1 into a *static node*. If a small amount of noise occurs on N1 while $CLK = 0$, T1 will drive N1 back to a valid logic value.

This circuit assumes CLK and $\overline{CLK}$ are both available. If not, two more transistors are needed for a CLK inverter.

Figure 3.13 shows a D flip-flop constructed from two static latches controlled by $\overline{CLK}$ and CLK. Some redundant internal inverters have been removed, so the flip-flop requires only 20 transistors.

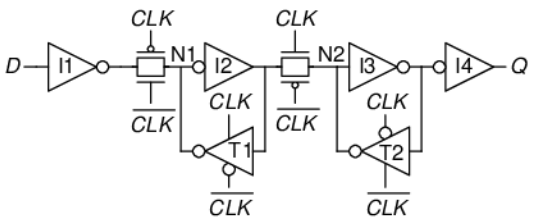


Figure 3.13 D flip-flop schematic

3.2.8 Putting It All Together

Latches and flip-flops are the fundamental building blocks of sequential circuits. Remember that a D latch is level-sensitive, whereas a D flip-flop is edge-triggered. The D latch is transparent when $CLK = 1$, allowing the input D to flow through to the output Q. The D flip-flop copies D to Q on the rising edge of CLK. At all other times, latches and flip-flops retain their old state. A register is a bank of several D flip-flops that share a common CLK signal.

Example 3.2 FLIP-FLOP AND LATCH COMPARISON

Ben Bitdiddle applies the D and CLK inputs shown in Figure 3.14 to a D latch and a D flip-flop. Help him determine the output, Q, of each device.

Solution: Figure 3.15 shows the output waveforms, assuming a small delay for Q to respond to input changes. The arrows indicate the cause of an output change. The initial value of Q is unknown and could be 0 or 1, as indicated by the pair of horizontal lines. First consider the latch. On the first rising edge of CLK, $D = 0$, so Q definitely becomes 0. Each time D changes while $CLK = 1$, Q also follows. When D changes while $CLK = 0$, it is ignored. Now consider the flip-flop. On each rising edge of CLK, D is copied to Q. At all other times, Q retains its state.

Figure 3.14 Example waveforms

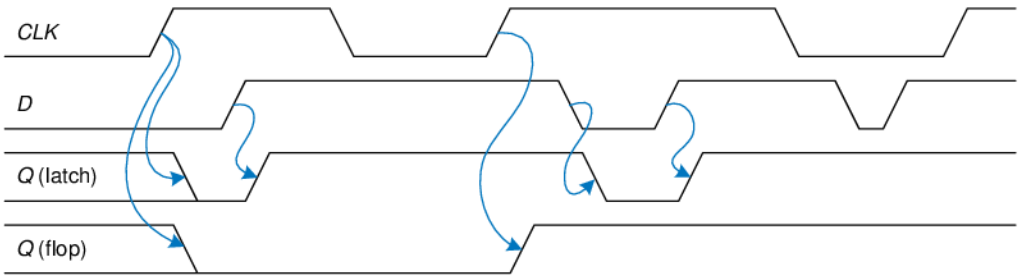


Figure 3.15 Solution waveforms

3.3 SYNCHRONOUS LOGIC DESIGN

In general, sequential circuits include all circuits that are not combinational—that is, those whose output cannot be determined simply by looking at the current inputs. Some sequential circuits are just plain kooky. This section begins by examining some of those curious circuits. It then introduces the notion of synchronous sequential circuits and the dynamic discipline. By disciplining ourselves to synchronous sequential circuits, we can develop easy, systematic ways to analyze and design sequential systems.

3.3.1 Some Problematic Circuits

Example 3.3 ASTABLE CIRCUITS

Alyssa P. Hacker encounters three misbegotten inverters who have tied themselves in a loop, as shown in Figure 3.16. The output of the third inverter is *fed back* to the first inverter. Each inverter has a propagation delay of 1 ns. Determine what the circuit does.

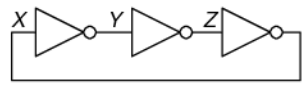


Figure 3.16 Three-inverter loop

Solution: Suppose node X is initially 0. Then $Y = 1$, $Z = 0$, and hence $X = 1$, which is inconsistent with our original assumption. The circuit has no stable states and is said to be *unstable* or *astable*. Figure 3.17 shows the behavior of the circuit. If X rises at time 0, Y will fall at 1 ns, Z will rise at 2 ns, and X will fall again at 3 ns. In turn, Y will rise at 4 ns, Z will fall at 5 ns, and X will rise again at 6 ns, and then the pattern will repeat. Each node oscillates between 0 and 1 with a *period* (repetition time) of 6 ns. This circuit is called a *ring oscillator*.

The period of the ring oscillator depends on the propagation delay of each inverter. This delay depends on how the inverter was manufactured, the power supply voltage, and even the temperature. Therefore, the ring oscillator period is difficult to accurately predict. In short, the ring oscillator is a sequential circuit with zero inputs and one output that changes periodically.

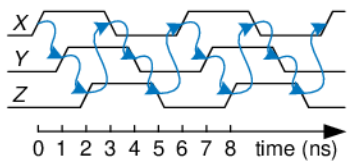


Figure 3.17 Ring oscillator waveforms

Example 3.4 RACE CONDITIONS

Ben Bitdiddle designed a new D latch that he claims is better than the one in Figure 3.17 because it uses fewer gates. He has written the truth table to find

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$

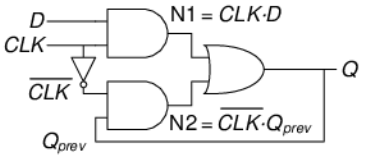


Figure 3.18 An improved (?) D latch

the output, Q, given the two inputs, D and CLK, and the old state of the latch, Q_{prev}. Based on this truth table, he has derived Boolean equations. He obtains Q_{prev} by feeding back the output, Q. His design is shown in Figure 3.18. Does his latch work correctly, independent of the delays of each gate?

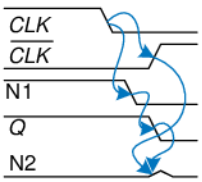


Figure 3.19 Latch waveforms illustrating race condition

Solution: Figure 3.19 shows that the circuit has a *race condition* that causes it to fail when certain gates are slower than others. Suppose $CLK = D = 1$. The latch is transparent and passes D through to make $Q = 1$. Now, CLK falls. The latch should remember its old value, keeping $Q = 1$. However, suppose the delay through the inverter from CLK to $\overline{CLK}$ is rather long compared to the delays of the AND and OR gates. Then nodes N1 and Q may both fall before $\overline{CLK}$ rises. In such a case, N2 will never rise, and Q becomes stuck at 0.

This is an example of *asynchronous* circuit design in which outputs are directly fed back to inputs. Asynchronous circuits are infamous for having race conditions where the behavior of the circuit depends on which of two paths through logic gates is fastest. One circuit may work, while a seemingly identical one built from gates with slightly different delays may not work. Or the circuit may work only at certain temperatures or voltages at which the delays are just right. These mistakes are extremely difficult to track down.

3.3.2 Synchronous Sequential Circuits

The previous two examples contain loops called *cyclic paths*, in which outputs are fed directly back to inputs. They are sequential rather than combinational circuits. Combinational logic has no cyclic paths and no races. If inputs are applied to combinational logic, the outputs will always settle to the correct value within a propagation delay. However, sequential circuits with cyclic paths can have undesirable races or unstable behavior. Analyzing such circuits for problems is time-consuming, and many bright people have made mistakes.

To avoid these problems, designers break the cyclic paths by inserting registers somewhere in the path. This transforms the circuit into a collection of combinational logic and registers. The registers contain the state of the system, which changes only at the clock edge, so we say the state is *synchronized* to the clock. If the clock is sufficiently slow, so that the inputs to all registers settle before the next clock edge, all races are eliminated. Adopting this discipline of always using registers in the feedback path leads us to the formal definition of a synchronous sequential circuit.

Recall that a circuit is defined by its input and output terminals and its functional and timing specifications. A sequential circuit has a finite set of discrete *states* $\{S_0, S_1, \ldots, S_{k-1}\}$. A *synchronous sequential circuit* has a clock input, whose rising edges indicate a sequence of times at which state transitions occur. We often use the terms *current state* and *next state* to distinguish the state of the system at the present from the state to which it will enter on the next clock edge. The functional specification details the next state and the value of each output for each possible combination of current state and input values. The timing specification consists of an upper bound, t_{pcq}, and a lower bound, t_{ccq}, on the time from the rising edge of the clock until the *output* changes, as well as *setup* and *hold* times, t_{setup} and t_{hold}, that indicate when the *inputs* must be stable relative to the rising edge of the clock.

t_{pcq} stands for the time of propagation from clock to Q, where Q indicates the output of a synchronous sequential circuit. t_{ccq} stands for the time of contamination from clock to Q. These are analogous to t_{pd} and t_{cd} in combinational logic.

The rules of *synchronous sequential circuit composition* teach us that a circuit is a synchronous sequential circuit if it consists of interconnected circuit elements such that

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register.

This definition of a synchronous sequential circuit is sufficient, but more restrictive than necessary. For example, in high-performance microprocessors, some registers may receive delayed or gated clocks to squeeze out the last bit of performance or power. Similarly, some microprocessors use latches instead of registers. However, the definition is adequate for all of the synchronous sequential circuits covered in this book and for most commercial digital systems.

Sequential circuits that are not synchronous are called *asynchronous*.

A flip-flop is the simplest synchronous sequential circuit. It has one input, D, one clock, CLK, one output, Q, and two states, $\{0, 1\}$. The functional specification for a flip-flop is that the next state is D and that the output, Q, is the current state, as shown in Figure 3.20.

We often call the current state variable S and the next state variable S'. In this case, the prime after S indicates next state, not inversion. The timing of sequential circuits will be analyzed in Section 3.5.

Two other common types of synchronous sequential circuits are called finite state machines and pipelines. These will be covered later in this chapter.

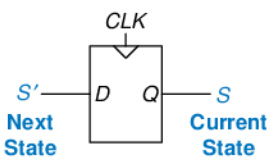


Figure 3.20 Flip-flop current state and next state

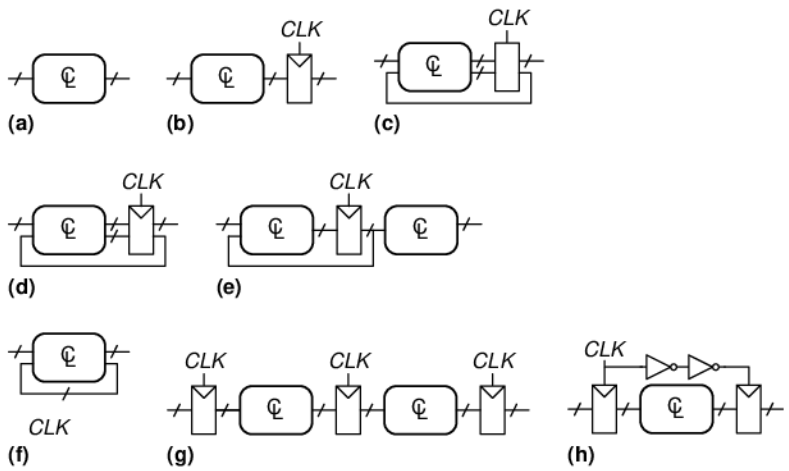


Figure 3.21 Example circuits

Example 3.5 SYNCHRONOUS SEQUENTIAL CIRCUITS

Which of the circuits in Figure 3.21 are synchronous sequential circuits?

Solution: Circuit (a) is combinational, not sequential, because it has no registers. (b) is a simple sequential circuit with no feedback. (c) is neither a combinational circuit nor a synchronous sequential circuit, because it has a latch that is neither a register nor a combinational circuit. (d) and (e) are synchronous sequential logic; they are two forms of finite state machines, which are discussed in Section 3.4. (f) is neither combinational nor synchronous sequential, because it has a cyclic path from the output of the combinational logic back to the input of the same logic but no register in the path. (g) is synchronous sequential logic in the form of a pipeline, which we will study in Section 3.6. (h) is not, strictly speaking, a synchronous sequential circuit, because the second register receives a different clock signal than the first, delayed by two inverter delays.

3.3.3 Synchronous and Asynchronous Circuits

Asynchronous design in theory is more general than synchronous design, because the timing of the system is not limited by clocked registers. Just as analog circuits are more general than digital circuits because analog circuits can use any voltage, asynchronous circuits are more general than synchronous circuits because they can use any kind of feedback. However, synchronous circuits have proved to be easier to design and use than asynchronous circuits, just as digital are easier than analog circuits. Despite decades of research on asynchronous circuits, virtually all digital systems are essentially synchronous.

Of course, asynchronous circuits are occasionally necessary when communicating between systems with different clocks or when receiving inputs at arbitrary times, just as analog circuits are necessary when communicating with the real world of continuous voltages. Furthermore, research in asynchronous circuits continues to generate interesting insights, some of which can improve synchronous circuits too.

3.4 FINITE STATE MACHINES

Synchronous sequential circuits can be drawn in the forms shown in Figure 3.22. These forms are called *finite state machines* (*FSMs*). They get their name because a circuit with k registers can be in one of a finite number (2^k) of unique states. An FSM has M inputs, N outputs, and k bits of state. It also receives a clock and, optionally, a reset signal. An FSM consists of two blocks of combinational logic, *next state logic* and *output logic*, and a register that stores the state. On each clock edge, the FSM advances to the next state, which was computed based on the current state and inputs. There are two general classes of finite state machines, characterized by their functional specifications. In *Moore machines*, the outputs depend only on the current state of the machine. In *Mealy machines*, the outputs depend on both the current state and the current inputs. Finite state machines provide a systematic way to design synchronous sequential circuits given a functional specification. This method will be explained in the remainder of this section, starting with an example.

Moore and Mealy machines are named after their promoters, researchers who developed *automata theory*, the mathematical underpinnings of state machines, at Bell Labs.

Edward F. Moore (1925–2003), not to be confused with Intel founder Gordon Moore, published his seminal article, *Gedanken-experiments on Sequential Machines* in 1956. He subsequently became a professor of mathematics and computer science at the University of Wisconsin.

George H. Mealy published *A Method of Synthesizing Sequential Circuits* in 1955. He subsequently wrote the first Bell Labs operating system for the IBM 704 computer. He later joined Harvard University.

3.4.1 FSM Design Example

To illustrate the design of FSMs, consider the problem of inventing a controller for a traffic light at a busy intersection on campus. Engineering students are moseying between their dorms and the labs on Academic Ave. They are busy reading about FSMs in their favorite

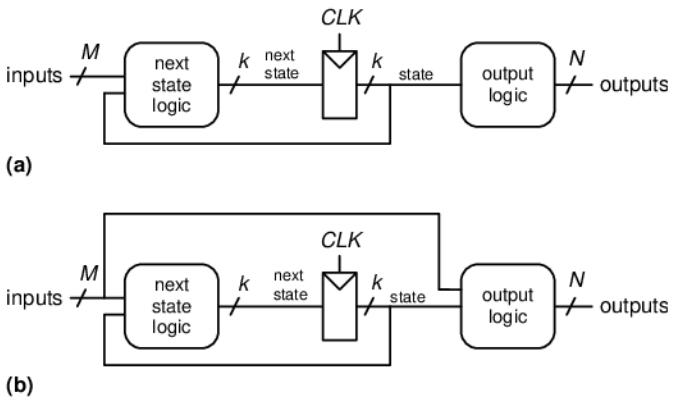


Figure 3.22 Finite state machines: (a) Moore machine, (b) Mealy machine

textbook and aren't looking where they are going. Football players are hustling between the athletic fields and the dining hall on Bravado Boulevard. They are tossing the ball back and forth and aren't looking where they are going either. Several serious injuries have already occurred at the intersection of these two roads, and the Dean of Students asks Ben Bitdiddle to install a traffic light before there are fatalities.

Ben decides to solve the problem with an FSM. He installs two traffic sensors, T_A and T_B, on Academic Ave. and Bravado Blvd., respectively. Each sensor indicates TRUE if students are present and FALSE if the street is empty. He also installs two traffic lights, L_A and L_B, to control traffic. Each light receives digital inputs specifying whether it should be green, yellow, or red. Hence, his FSM has two inputs, T_A and T_B, and two outputs, L_A and L_B. The intersection with lights and sensors is shown in Figure 3.23. Ben provides a clock with a 5-second period. On each clock tick (rising edge), the lights may change based on the traffic sensors. He also provides a reset button so that Physical Plant technicians can put the controller in a known initial state when they turn it on. Figure 3.24 shows a black box view of the state machine.

Ben's next step is to sketch the *state transition diagram*, shown in Figure 3.25, to indicate all the possible states of the system and the transitions between these states. When the system is reset, the lights are green on Academic Ave. and red on Bravado Blvd. Every 5 seconds, the controller examines the traffic pattern and decides what to do next. As long

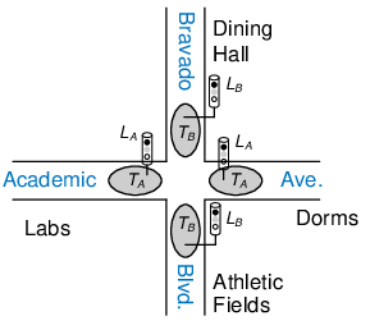


Figure 3.23 Campus map

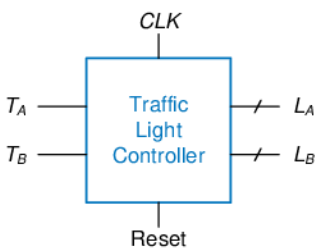


Figure 3.24 Black box view of finite state machine

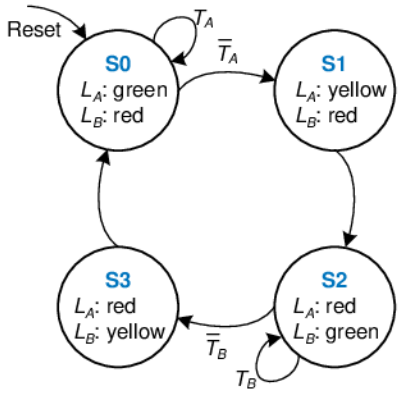


Figure 3.25 State transition diagram

as traffic is present on Academic Ave., the lights do not change. When there is no longer traffic on Academic Ave., the light on Academic Ave. becomes yellow for 5 seconds before it turns red and Bravado Blvd.'s light turns green. Similarly, the Bravado Blvd. light remains green as long as traffic is present on the boulevard, then turns yellow and eventually red.

In a state transition diagram, circles represent states and arcs represent transitions between states. The transitions take place on the rising edge of the clock; we do not bother to show the clock on the diagram, because it is always present in a synchronous sequential circuit. Moreover, the clock simply controls when the transitions should occur, whereas the diagram indicates which transitions occur. The arc labeled Reset pointing from outer space into state S0 indicates that the system should enter that state upon reset, regardless of what previous state it was in. If a state has multiple arcs leaving it, the arcs are labeled to show what input triggers each transition. For example, when in state S0, the system will remain in that state if T_A is TRUE and move to S1 if T_A is FALSE. If a state has a single arc leaving it, that transition always occurs regardless of the inputs. For example, when in state S1, the system will always move to S2. The value that the outputs have while in a particular state are indicated in the state. For example, while in state S2, L_A is red and L_B is green.

Ben rewrites the state transition diagram as a *state transition table* (Table 3.1), which indicates, for each state and input, what the next state, S', should be. Note that the table uses don't care symbols (X) whenever the next state does not depend on a particular input. Also note that Reset is omitted from the table. Instead, we use resettable flip-flops that always go to state S0 on reset, independent of the inputs.

The state transition diagram is abstract in that it uses states labeled {S0, S1, S2, S3} and outputs labeled {red, yellow, green}. To build a real circuit, the states and outputs must be assigned *binary encodings*. Ben chooses the simple encodings given in Tables 3.2 and 3.3. Each state and each output is encoded with two bits: $S_{1:0}$, $L_{A1:0}$, and $L_{B1:0}$.

Table 3.1 State transition table

Current State S	Inputs T_A	T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 State encoding

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Table 3.3 Output encoding

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

Ben updates the state transition table to use these binary encodings, as shown in Table 3.4. The revised state transition table is a truth table specifying the next state logic. It defines next state, S', as a function of the current state, S, and the inputs. The revised output table is a truth table specifying the output logic. It defines the outputs, L_A and L_B, as functions of the current state, S.

From this table, it is straightforward to read off the Boolean equations for the next state in sum-of-products form.

$$\begin{aligned} S_1' &= \overline{S}_1 S_0 + S_1 \overline{S}_0 \overline{T}_B + S_1 \overline{S}_0 T_B \\ S_0' &= \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B \end{aligned} \tag{3.1}$$

The equations can be simplified using Karnaugh maps, but often doing it by inspection is easier. For example, the T_B and $\overline{T}_B$ terms in the S'_1 equation are clearly redundant. Thus S'_1 reduces to an XOR operation. Equation 3.2 gives the *next state equations*.

Table 3.4 State transition table with binary encodings

Current State S_1	S_0	Inputs T_A	T_B	Next State S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Table 3.5 Output table

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$\begin{aligned} S_1' &= S_1 \oplus S_0 \\ S_0' &= \overline{S}_1 \overline{S}_0 T_A + S_1 \overline{S}_0 T_B \end{aligned} \tag{3.2}$$

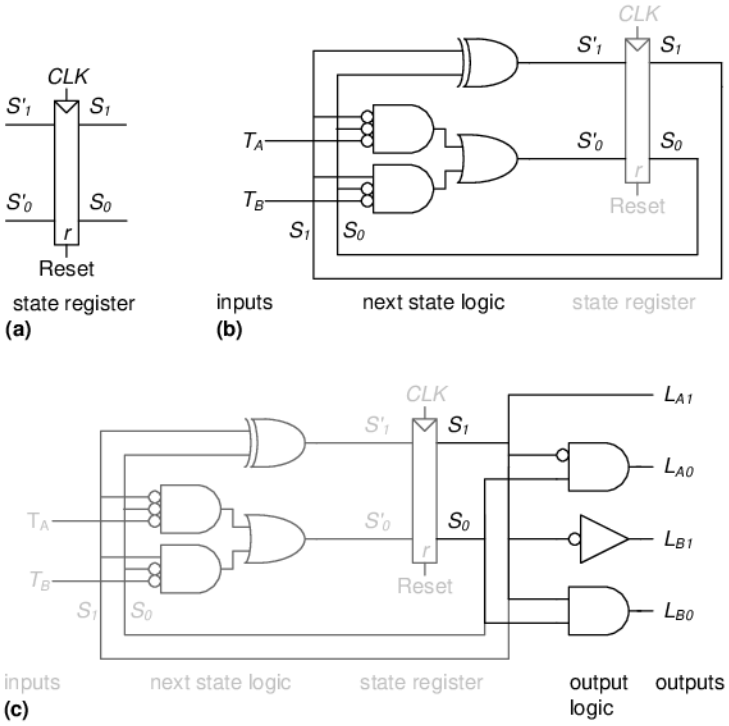
Similarly, Ben writes an *output table* (Table 3.5) indicating, for each state, what the output should be in that state. Again, it is straightforward to read off and simplify the Boolean equations for the outputs. For example, observe that L_{A1} is TRUE only on the rows where S_1 is TRUE.

$$\begin{aligned} L_{A1} &= S_1 \\ L_{A0} &= \overline{S}_1 S_0 \\ L_{B1} &= \overline{S}_1 \\ L_{B0} &= S_1 S_0 \end{aligned} \tag{3.3}$$

Finally, Ben sketches his Moore FSM in the form of Figure 3.22(a). First, he draws the 2-bit state register, as shown in Figure 3.26(a). On each clock edge, the state register copies the next state, $S'_{1:0}$, to become the state, $S_{1:0}$. The state register receives a synchronous or asynchronous reset to initialize the FSM at startup. Then, he draws the next state logic, based on Equation 3.2, which computes the next state, based on the current state and inputs, as shown in Figure 3.26(b). Finally, he draws the output logic, based on Equation 3.3, which computes the outputs based on the current state, as shown in Figure 3.26(c).

Figure 3.27 shows a timing diagram illustrating the traffic light controller going through a sequence of states. The diagram shows *CLK*, Reset, the inputs T_A and T_B, next state S', state S, and outputs L_A and L_B. Arrows indicate causality; for example, changing the state causes the outputs to change, and changing the inputs causes the next state to change. Dashed lines indicate the rising edge of *CLK* when the state changes.

The clock has a 5-second period, so the traffic lights change at most once every 5 seconds. When the finite state machine is first turned on, its state is unknown, as indicated by the question marks. Therefore, the system should be reset to put it into a known state. In this timing diagram,



This schematic uses some AND gates with bubbles on the inputs. They might be constructed with AND gates and input inverters, with NOR gates and inverters for the non-bubbled inputs, or with some other combination of gates. The best choice depends on the particular implementation technology.

Figure 3.26 State machine circuit for traffic light controller

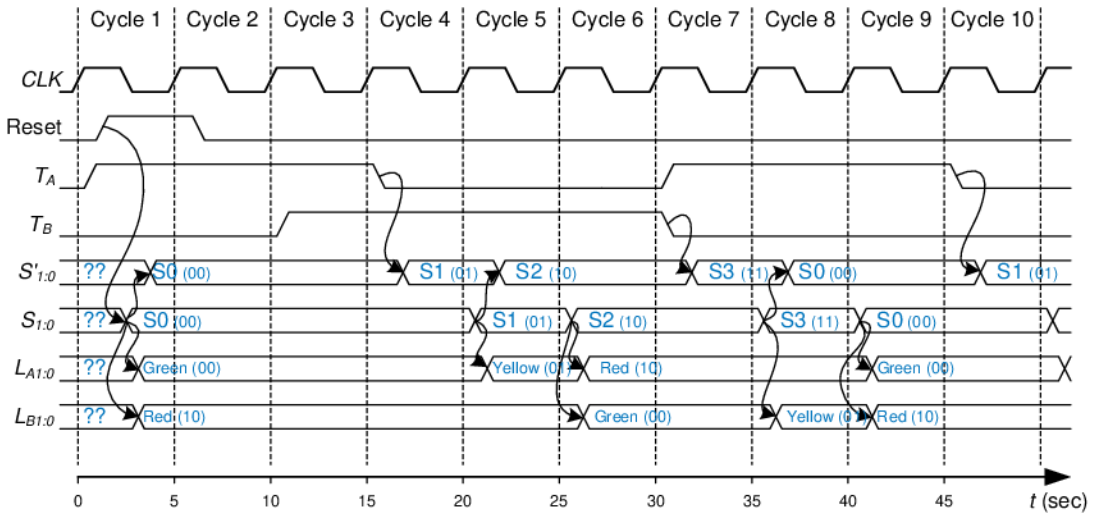


Figure 3.27 Timing diagram for traffic light controller

S immediately resets to S0, indicating that asynchronously resettable flip-flops are being used. In state S0, light L_A is green and light L_B is red.

In this example, traffic arrives immediately on Academic Ave. Therefore, the controller remains in state S0, keeping L_A green even though traffic arrives on Bravado Blvd. and starts waiting. After 15 seconds, the traffic on Academic Ave. has all passed through and T_A falls. At the following clock edge, the controller moves to state S1, turning L_A yellow. In another 5 seconds, the controller proceeds to state S2 in which L_A turns red and L_B turns green. The controller waits in state S2 until all the traffic on Bravado Blvd. has passed through. It then proceeds to state S3, turning L_B yellow. 5 seconds later, the controller enters state S0, turning L_B red and L_A green. The process repeats.

Despite Ben's best efforts, students don't pay attention to traffic lights and collisions continue to occur. The Dean of Students next asks him to design a catapult to throw engineering students directly from their dorm roofs through the open windows of the lab, bypassing the troublesome intersection all together. But that is the subject of another textbook.

3.4.2 State Encodings

In the previous example, the state and output encodings were selected arbitrarily. A different choice would have resulted in a different circuit. A natural question is how to determine the encoding that produces the circuit with the fewest logic gates or the shortest propagation delay. Unfortunately, there is no simple way to find the best encoding except to try all possibilities, which is infeasible when the number of states is large. However, it is often possible to choose a good encoding by inspection, so that related states or outputs share bits. Computer-aided design (CAD) tools are also good at searching the set of possible encodings and selecting a reasonable one.

One important decision in state encoding is the choice between binary encoding and one-hot encoding. With *binary encoding*, as was used in the traffic light controller example, each state is represented as a binary number. Because K binary numbers can be represented by $\log_2 K$ bits, a system with K states only needs $\log_2 K$ bits of state.

In *one-hot encoding*, a separate bit of state is used for each state. It is called one-hot because only one bit is "hot" or TRUE at any time. For example, a one-hot encoded FSM with three states would have state encodings of 001, 010, and 100. Each bit of state is stored in a flip-flop, so one-hot encoding requires more flip-flops than binary encoding. However, with one-hot encoding, the next-state and output logic is often simpler, so fewer gates are required. The best encoding choice depends on the specific FSM.

Example 3.6 FSM STATE ENCODING

A *divide-by-N counter* has one output and no inputs. The output Y is HIGH for one clock cycle out of every N. In other words, the output divides the frequency of the clock by N. The waveform and state transition diagram for a divide-by-3 counter is shown in Figure 3.28. Sketch circuit designs for such a counter using binary and one-hot state encodings.

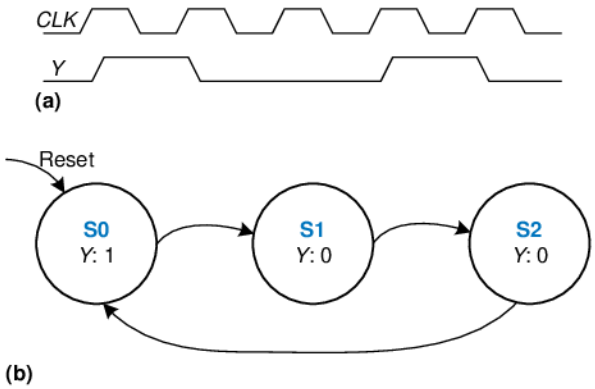


Figure 3.28 **Divide-by-3 counter (a) waveform and (b) state transition diagram**

Table 3.6 **Divide-by-3 counter state transition table**

Current State	Next State
S0	S1
S1	S2
S2	S0

Table 3.7 **Divide-by-3 counter output table**

Current State	Output
S0	1
S1	0
S2	0

Solution: Tables 3.6 and 3.7 show the abstract state transition and output tables before encoding.

Table 3.8 compares binary and one-hot encodings for the three states.

The binary encoding uses two bits of state. Using this encoding, the state transition table is shown in Table 3.9. Note that there are no inputs; the next state depends only on the current state. The output table is left as an exercise to the reader. The next-state and output equations are:

$$\begin{aligned} S_1' &= \overline{S}_1 S_0 \\ S_0' &= \overline{S}_1 \overline{S}_0 \end{aligned} \tag{3.4}$$

$$Y = \overline{S}_1 \overline{S}_0 \tag{3.5}$$

The one-hot encoding uses three bits of state. The state transition table for this encoding is shown in Table 3.10 and the output table is again left as an exercise to the reader. The next-state and output equations are as follows:

$$\begin{aligned} S_2' &= S_1 \\ S_1' &= S_0 \\ S_0' &= S_2 \end{aligned} \tag{3.6}$$

$$Y = S_0 \tag{3.7}$$

Figure 3.29 shows schematics for each of these designs. Note that the hardware for the binary encoded design could be optimized to share the same gate for Y and S'_0. Also observe that the one-hot encoding requires both settable (s) and resettable (r) flip-flops to initialize the machine to S0 on reset. The best implementation choice depends on the relative cost of gates and flip-flops, but the one-hot design is usually preferable for this specific example.

A related encoding is the *one-cold* encoding, in which K states are represented with K bits, exactly one of which is FALSE.

Table 3.8 Binary and one-hot encodings for divide-by-3 counter

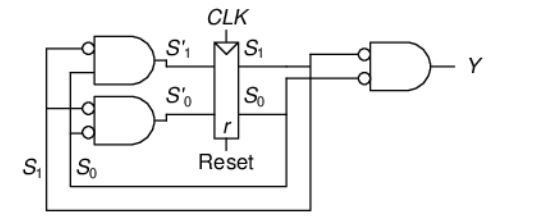
State	Binary Encoding S_2	S_1	S_0	One-Hot Encoding S_1	S_0
S0	0	0	1	0	1
S1	0	1	0	1	0
S2	1	0	0	0	0

Table 3.9 State transition table with binary encoding

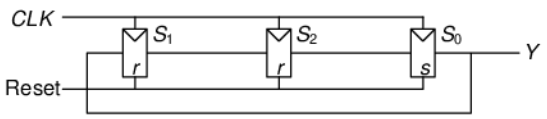
Current State S_1	S_0	Next State S'_1	S'_0
0	0	0	1
0	1	1	0
1	0	0	0

Table 3.10 State transition table with one-hot encoding

Current State S_2	S_1	S_0	Next State S'_2	S'_1	S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1



(a)



(b)

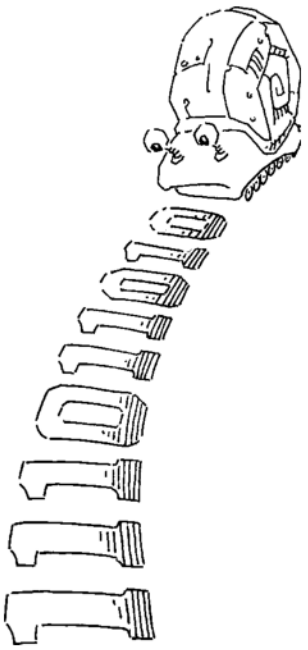
Figure 3.29 Divide-by-3 circuits for (a) binary and (b) one-hot encodings

3.4.3 Moore and Mealy Machines

An easy way to remember the difference between the two types of finite state machines is that a Moore machine typically has *more* states than a Mealy machine for a given problem.

So far, we have shown examples of Moore machines, in which the output depends only on the state of the system. Hence, in state transition diagrams for Moore machines, the outputs are labeled in the circles. Recall that Mealy machines are much like Moore machines, but the outputs can depend on inputs as well as the current state. Hence, in state transition diagrams for Mealy machines, the outputs are labeled on the arcs instead of in the circles. The block of combinational logic that computes the outputs uses the current state and inputs, as was shown in Figure 3.22(b).

Example 3.7 MOORE VERSUS MEALY MACHINES



Alyssa P. Hacker owns a pet robotic snail with an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 1's and 0's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last four bits that it has crawled over are, from left to right, 1101. Design the FSM to compute when the snail should smile. The input A is the bit underneath the snail's antennae. The output Y is TRUE when the snail smiles. Compare Moore and Mealy state machine designs. Sketch a timing diagram for each machine showing the input, states, and output as your snail crawls along the sequence 111011010.

Solution: The Moore machine requires five states, as shown in Figure 3.30(a). Convince yourself that the state transition diagram is correct. In particular, why is there an arc from S4 to S2 when the input is 1?

In comparison, the Mealy machine requires only four states, as shown in Figure 3.30(b). Each arc is labeled as A/Y. A is the value of the input that causes that transition, and Y is the corresponding output.

Tables 3.11 and 3.12 show the state transition and output tables for the Moore machine. The Moore machine requires at least three bits of state. Consider using a binary state encoding: S0 = 000, S1 = 001, S2 = 010, S3 = 011, and S4 = 100. Tables 3.13 and 3.14 rewrite the state transition and output tables with these encodings (These four tables follow on page 128).

From these tables, we find the next state and output equations by inspection. Note that these equations are simplified using the fact that states 101, 110, and 111 do not exist. Thus, the corresponding next state and output for the nonexistent states are don't cares (not shown in the tables). We use the don't cares to minimize our equations.

$$
\begin{aligned}
S_2' &= S_1 S_0 A \\
S_1' &= \overline{S}_1 S_0 A + S_1 \overline{S}_0 + S_2 A \\
S_0' &= \overline{S}_2 \overline{S}_1 \overline{S}_0 A + S_1 \overline{S}_0 \overline{A}
\end{aligned}
\tag{3.8}
$$

$$Y = S_2 \tag{3.9}$$

Table 3.15 shows the combined state transition and output table for the Mealy machine. The Mealy machine requires at least two bits of state. Consider using a binary state encoding: S0 = 00, S1 = 01, S2 = 10, and S3 = 11. Table 3.16 rewrites the state transition and output table with these encodings.

From these tables, we find the next state and output equations by inspection.

$$\begin{aligned} S_1' &= S_1\overline{S}_0 + \overline{S}_1 S_0 A \\ S_0' &= \overline{S}_1\overline{S}_0 A + S_1\overline{S}_0\overline{A} + S_1 S_0 A \end{aligned} \tag{3.10}$$

$$Y = S_1 S_0 A \tag{3.11}$$

The Moore and Mealy machine schematics are shown in Figure 3.31(a) and 3.31(b), respectively.

The timing diagrams for the Moore and Mealy machines are shown in Figure 3.32 (see page 131). The two machines follow a different sequence of states. Moreover, the Mealy machine's output rises a cycle sooner because it responds to the input rather than waiting for the state change. If the Mealy output were delayed through a flip-flop, it would match the Moore output. When choosing your FSM design style, consider when you want your outputs to respond.

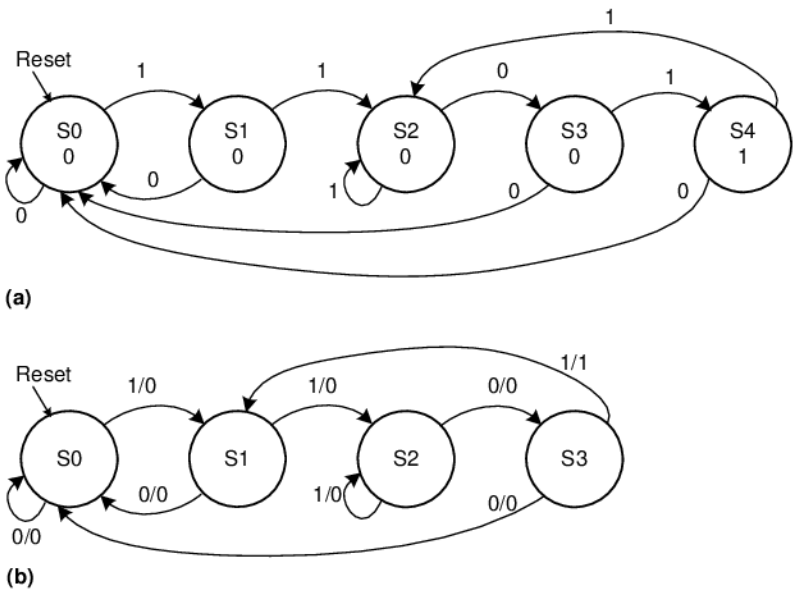


Figure 3.30 **FSM state transition diagrams: (a) Moore machine, (b) Mealy machine**