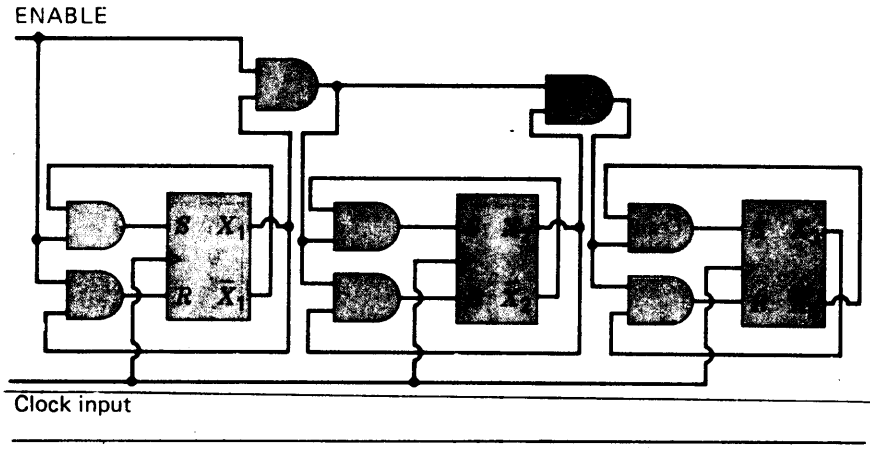


FIGURE 4.12

Gated-clocked binary counter.

ations are *enabled* by logic levels and activated by clock signals. The preceding counter is called a *ripple counter* because changes ripple down the flip-flop chain.

In Fig. 4.12 the ENABLE input to the first flip-flop in the chain X_1 goes to two AND gates, which also have the outputs of the flip-flop as inputs. Note: if the ENABLE signal is a 0, the two AND gates will have 0 outputs and X_1 will remain in the same state regardless of how many clock pulses occur.

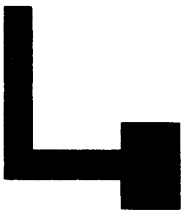
When the ENABLE signal is a 1, however, the outputs from flip-flop X_1 cause that flip-flop to always change values when a clock pulse occurs. Thus the counter records the number of clock pulses that occur while the ENABLE is on. Then the flip-flop X_2 will change only when X_1 is a 1, the ENABLE signal is a 1, and a positive-going clock signal occurs. Similarly, X_3 will change states only when X_1 and X_2 are 1s, the ENABLE is a 1, and a clock positive edge occurs.

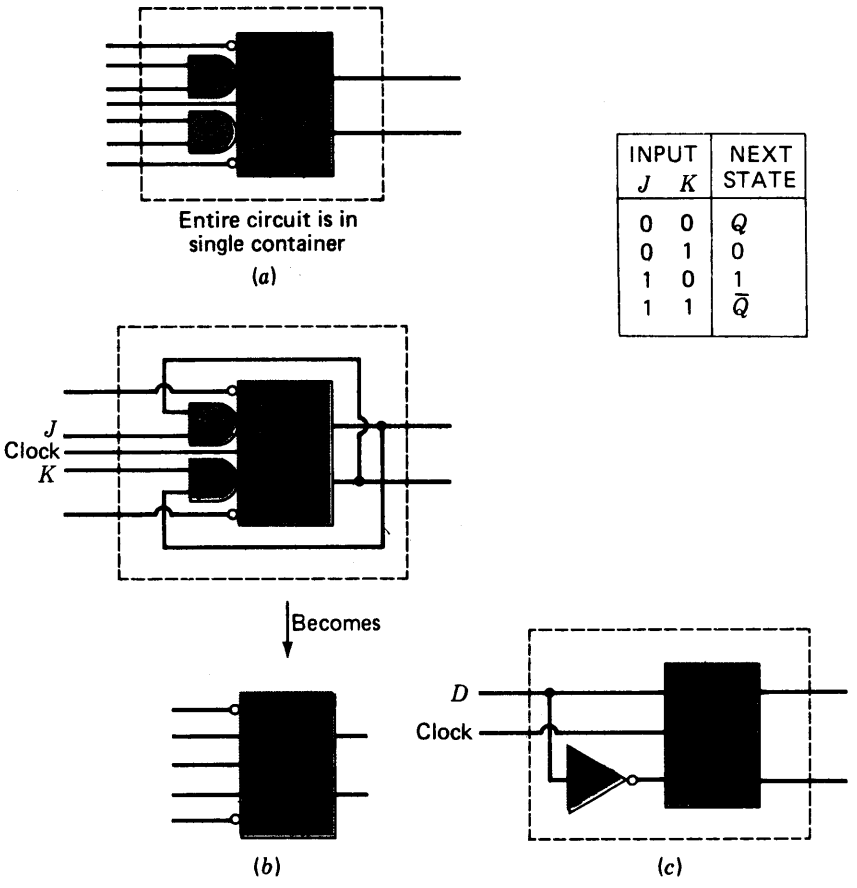
The two AND gates combined with an *RS* flip-flop in Fig. 4.12 are so useful that most popular lines of flip-flops contain in a single integrated-circuit container the flip-flop and its two AND gates, as shown in Fig. 4.13(*a*).

Figure 4.13(*b*) shows another very popular and useful flip-flop, which consists of the *RS* flip-flop and its two AND gates, but with the AND gates having the cross-coupling already permanently made. In this form the two lines taken outside are called *J* and *K*, and the flip-flop is called a *JK flip-flop*. Analysis of this flip-flop indicates that the *J* and *K* inputs act just as *RS* inputs for two 0 inputs—in this case the flip-flop never changes states. Also, with a 0 on *J* and a 1 on *K*, the flip-flop goes to the 0 state when a clock positive edge appears; and with a 1 on *J* and a 0 on *K*, the flip-flop goes to 1 when a clock positive edge appears. The significant fact is that when both *J* and *K* are 1s, the flip-flop always changes states when a clock positive edge appears.

The flip-flops in Fig. 4.13(*a*) and (*b*) both have DC RESET and DC SET inputs. The bubbles at the input on the block diagram indicate that these are activated by 0 inputs and are normally held at a 1 level. When, for instance, a 0 is placed on the DC SET input, the flip-flop goes to a 1 level regardless of the clock or other inputs. DC SET and DC RESET should not be 0s at the same time, because this is forbidden and leads to an undetermined next state.

It is a general rule that bubbles, or small circles, at the DC SET and DC RESET inputs mean that these inputs are activated by 0 levels. The absence of





INPUT J	INPUT K	NEXT STATE
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

FIGURE 4.13

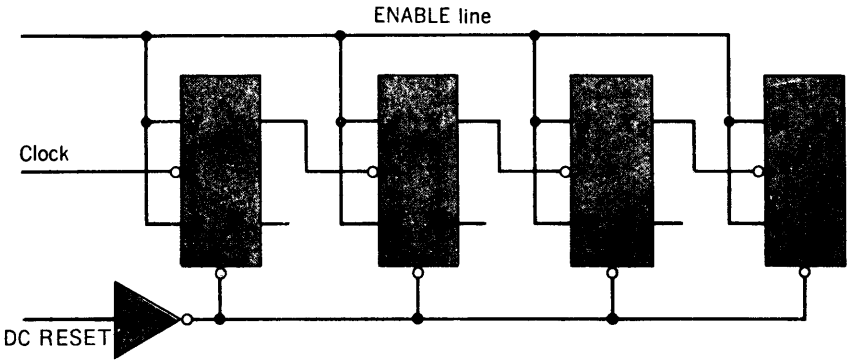
JK and *D* flip-flops. (*a*) *RS* flip-flop with AND gates. (*b*) How a *JK* flip-flop is made from an *RS* flip-flop. (*c*) *D* flip-flop.

these bubbles would mean that the inputs were activated by 1 levels and were normally at 0.

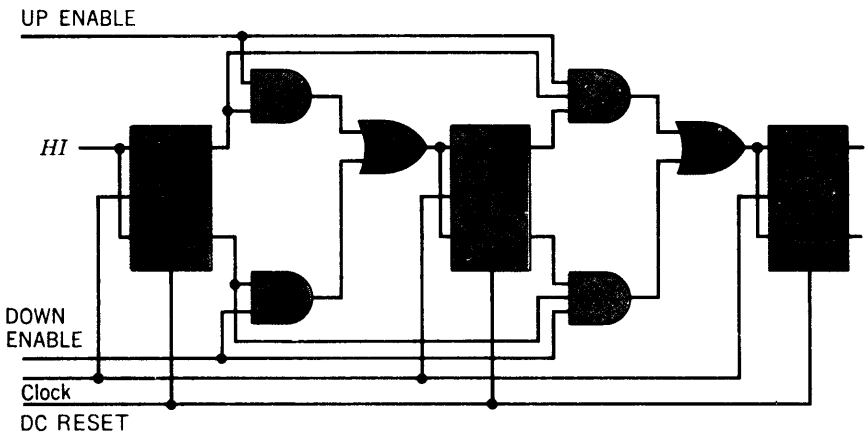
There is one other type of edge-triggered flip-flop now in general use, the *D flip-flop*. This flip-flop simply takes the value at its input when a clock pulse appears and remains in its same state until the next clock pulse appears. As shown in Fig. 4.13(*c*), the *D* flip-flop can be made from an *RS* flip-flop and an inverter. The operation is essentially the same as that of the *D* latch previously explained, except the *D* flip-flop operates on a clock edge and the latch is activated by a clock level.

The *D* flip-flop is very useful because when clocked, it takes the state on its input and holds it until clocked again. Only a single input line is needed for a transfer, whereas the *RS* or *JK* flip-flops require two input lines.

An example of the use of *JK* flip-flops is shown in Fig. 4.14(*a*) and (*b*). Figure 4.14(*a*) shows the simplicity of a gated binary counter with *JK* flip-flops. Figure 4.14(*b*) shows a block diagram for a *binary up-down counter*. When the UP ENABLE line is high or a 1, the counter will count up, that is, 0, 1, 2, 3,



(*a*)



(*b*)

FIGURE 4.14

Binary counters with *JK* flip-flops. (*a*) Gated ripple counter. (*b*) Up-down counter.

4, ... When the DOWN ENABLE line is a 1, the counter will count down, that is, 6, 5, 4, ... In general, the counter will increase its value by 1 if the UP ENABLE line is a 1 and a clock pulse arrives, or will decrease its value by a 1 if the DOWN ENABLE input is a 1 and a clock pulse occurs.

A RESET line is provided which is used to DC RESET the counter to 0. This is activated by a 1 on the RESET line.

BCD COUNTERS

4.9 The binary counters considered so far all count to their limit before resetting to all 0s. Often it is desired to have counters count in binary-coded decimal (BCD). Figure 4.15(*a*) shows a typical BCD counter. Examination of this counter shows that it counts normally until it reaches 1001; that is, the sequence until that time is as follows:

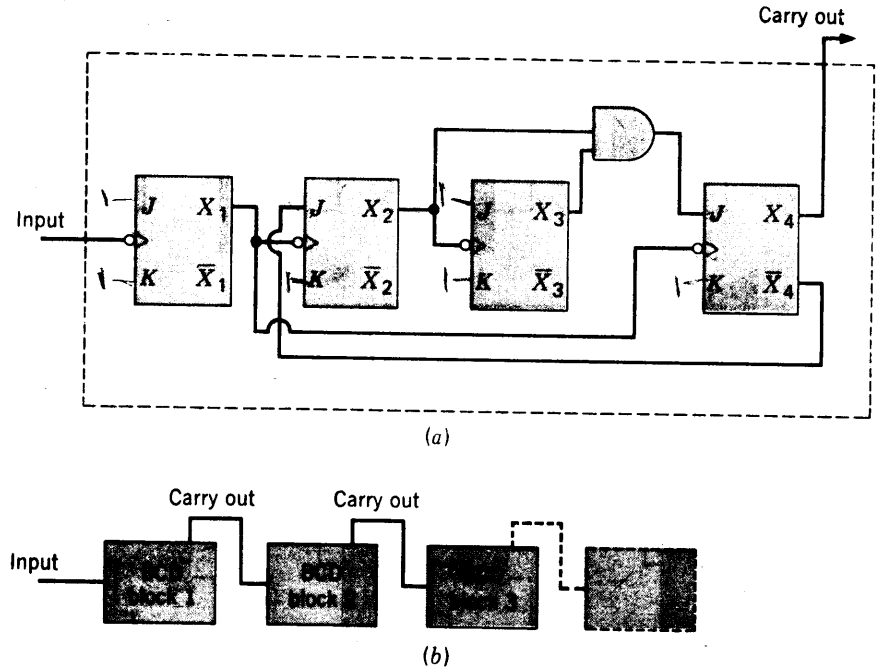


FIGURE 4.15

BCD counters. (*a*) Decade, or BCD, counter. (*Note:* Unconnected inputs are 1s.) (*b*) Cascading BCD counter blocks.

X_4	X_3	X_2	X_1
0	0	0	0
0	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	0	0	0
1	0	0	1

When the next negative-going edge at the input occurs, however, the BCD counter returns to all 0s. At the same time (that is, during the interval when the counter goes from 9 to 0) a negative-going signal edge occurs at the CARRY output. This CARRY output can be connected to the INPUT of another BCD counter, which will then be stepped by 1 when the first BCD stage goes from 9 to 0. This is shown in Fig. 4.15(*b*); where several four-flip-flop BCD stages are combined to make a large counter.

If we consider just two of the "BCD boxes," we find the sequence to be as follows:

8 Y_4	4 Y_3	2 Y_2	1 Y_1	8 X_4	4 X_3	2 X_2	1 X_1	value of bits
0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	1	
0	0	0	0	0	0	1	0	
0	0	0	0	0	0	1	1	
0	0	0	0	0	1	0	0	
0	0	0	0	0	1	0	1	
0	0	0	0	0	1	1	0	
0	0	0	0	0	1	1	1	
0	0	0	0	1	0	0	0	
0	0	0	0	1	0	0	1	
0	0	0	1	0	0	0	0	
0	0	0	1	0	0	0	1	
.	.	.	.	.	.	.	.	
0	0	0	1	1	0	0	1	
0	0	1	0	0	0	0	0	
0	0	1	0	0	0	0	1	
.	.	.	.	.	.	.	.	

Here we have counted to 21. This would continue until the counter reached 99, when the Y part would put out a signal which could be used to gate another stage to form a counter that could count to 999.

This sort of repetition of various "boxes," or "modules," such as a BCD counter, is facilitated by manufacturers placing an entire four-stage BCD counter in a single integrated-circuit container.

One thing should again be noted about the block diagram in Fig. 4.15(*a*). The flip-flops are activated by negative-going shifts in input levels at the input. This is indicated by the small circles or bubbles at the inputs. As a result, a flip-flop such as X_3 is activated when X_2 goes from a 1 to a 0, that is, when the 1 output makes a negative transition.

Also note that unconnected inputs, such as the K inputs of all the flip-flops and the J inputs of X_1 and X_3 are at 1 levels. This is due to the circuit construction.

INTEGRATED CIRCUITS

4.10 The flip-flops and gates used in modern computing machines—which range from calculators and microcomputers through the large high-speed computers—are constructed and packaged by using what is called *integrated-circuit technology*. When integrated circuits are used, one or more complete gates or flip-flops are packaged in a single integrated-circuit (IC) container. The IC containers provide input and output pins or connections which are then interconnected by plated strips on circuit boards, wires, or other means to form complete computing devices.

In earlier computers, flip-flop and gate circuits were constructed by using discrete electrical components such as resistors, capacitors, transistors, and, before that, vacuum tubes and relays. Individual components were interconnected to form

flip-flops and gates which were then interconnected to form computers. With the present-day IC technology, flip-flops and gates are fabricated in containers, and only the IC containers (or "cans") need be interconnected.

Two typical IC containers are shown in Fig. 4.16(*a*). One is called a *dual inline package* (in the trade it is called a "coffin," or a DIP), and this particular package has 14 pins which provide for external connections. For years this 14-pin package was a standard in the industry, and plastic and ceramic DIPs of this sort were the largest-selling IC package for some years.

There has been a tendency as IC technology improved, however, to increase the number of pins per package. Packages with 16 to 40 pins are becoming popular, and up to 100 pins per package can now be found in some IC manufacturers' products.

Figure 4.16(*b*) through (*e*) shows how several gates and flip-flops are packaged in a single container. The inputs and outputs are numbered, and each number refers to an external pin on the IC container. A ground connection and a positive power voltage are both required for each container, so that only 12 pins remain to be used for the actual inputs and outputs to gates and flip-flops. (For these circuits, each V_{CC} pin is connected to a 5.5-V power supply and GND to 0 V or system ground.)

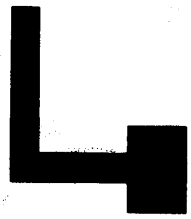
The particular circuits in Fig. 4.16 are called *transistor-transistor logic* (TTL) circuits and are widely used high-speed circuits. These circuits have a 3.5-V level for a 1 and 0.2-V level for a 0. The particular configurations shown with identical pin connections are manufactured by just about every major IC manufacturer, and packages from one manufacturer can be fairly easily substituted for another manufacturer's packages (provided the speed requirements or loading capabilities are not violated). There are many other packages with, for instance, three three-input NAND gates, two *RS* flip-flops, exclusive OR gates, etc.

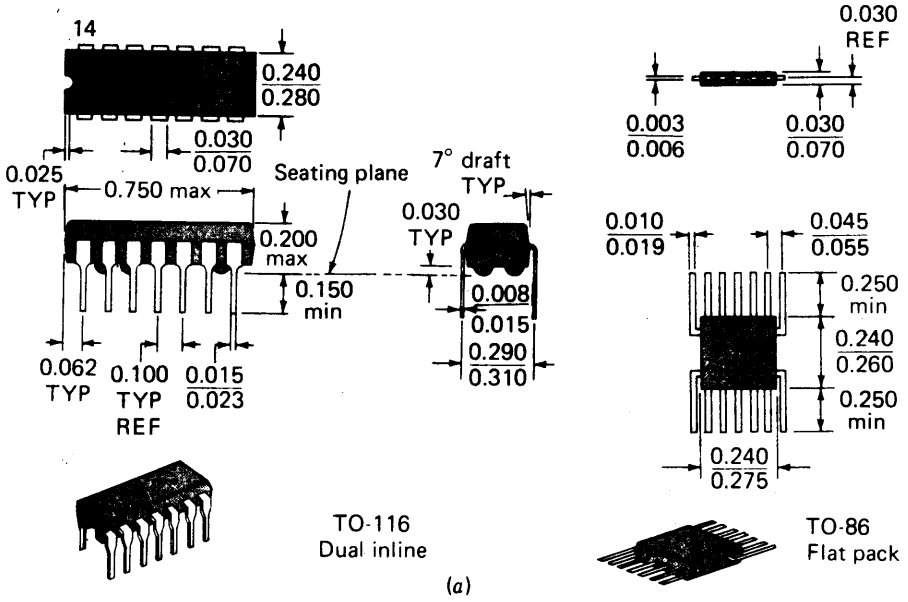
To illustrate the use of IC packages in logic design, we now examine an implementation of Fig. 4.17, using the packages shown in Fig. 4.16. The logic circuit in Fig. 4.17 is called a *shift register with feedback,*[4] for it consists of four flip-flops connected in a shift-register configuration and "feedback" from these

---

[4]This particular type of shift register with feedback is so widely used that complete books have been written about it. It is sometimes called a *linear shift register*, a *random sequence generation*, or a *linear recurring sequence generator*. With similar feedback connections, a register can be made with as many flip-flops as desired, thus forming counters with sequences of $2^N - 1$ for any reasonable N (where N is the number of flip-flops).

Consider the set of consecutive states taken by X_4 in the shift register in Fig. 4.17 to be its output sequence. Each nonzero 4-tuple occurs once in any 15-bit segment of this sequence, each nonzero 3-tuple occurs twice, etc. Adding a 15-bit segment of this sequence to another 15-bit segment bit by bit mod 2 (exclusive OR) will give still another 15-bit segment. These sequences are used in instruments to form random number generators and to generate bandpass noise; in radar for interplanetary observations; in communications systems to generate noise and encode or encrypt; and for many other purposes. See Birkhoff and Bartee for more information and references.





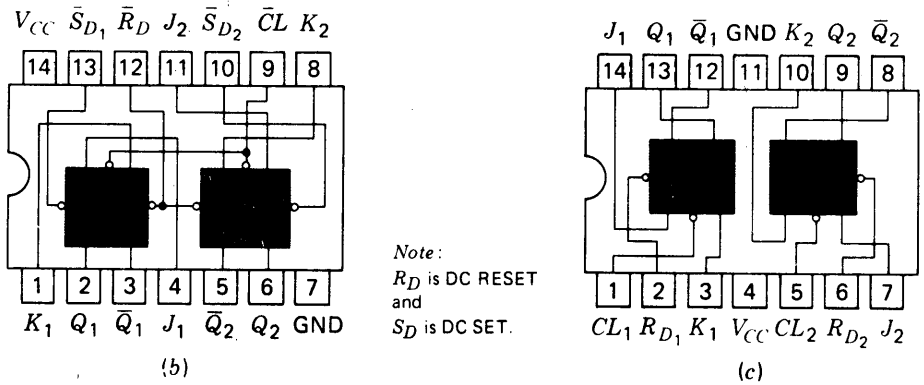


FIGURE 4.16

IC containers and flip-flop and gate circuits. (*a*) Dual inline and flat-pack IC containers. (*b*) Dual *JK* flip-flop with common clock and resets and separate sets. (*c*) Dual *JK* flip-flop with separate resets and clocks. (*d*) Dual four-input NAND gates. (*e*) Quadruplex two-input NAND gates.

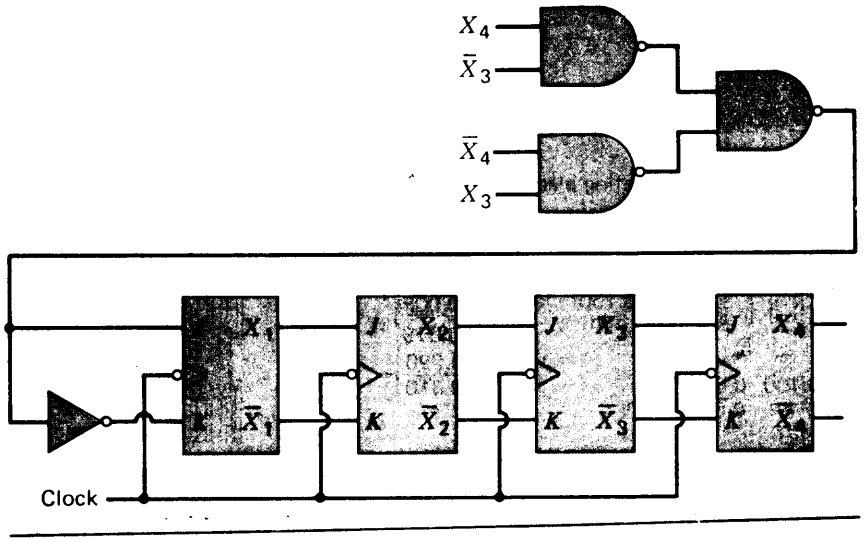


FIGURE 4.1

Shift register with feedback.

four flip-flops to the first flip-flop's inputs. This particular counter is started by setting a 1 in X_1 and 0s in X_2, X_3, and X_4. The sequence of states taken is then

1	0	0	0	
0	1	0	0	
0	0	1	0	
1	0	0	1	
1	1	0	0	
0	1	1	0	
1	0	1	1	
0	1	0	1	basic sequence which repeats
1	0	1	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	
0	1	1	1	
0	0	1	1	
0	0	0	1	
1	0	0	0	
0	1	0	0	

Notice that this sequence contains 15 of the 16 possible 4-bit numbers that might be taken by this circuit. (Only the all-0 combination is excluded.) This is a widely used sequence which occurs in many instruments and has many uses in radar systems, sonar systems, coding encryption boxes, etc.

Quite often the sequence of states taken by a logic circuit is written in a

counter table. The counter table for the above sequence is

X_1	X_2	X_3	X_4
1	0	0	0
0	1	0	0
0	0	1	0
1	0	0	1
1	1	0	0
0	1	1	0
1	0	1	1
0	1	0	1
1	0	1	0
1	1	0	1
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1

In the counter table, the flip-flops' names are first listed, followed by the starting states. Then the successive states taken are listed in order, and the final line contains the state preceding the starting state.

There is a straightforward technique for designing a logic circuit to realize a counter table; this technique is developed in Sec. 4.12. For now we return to the implementation of the counter in Fig. 4.17.

In order to implement this counter, we require four flip-flops and a gate circuit which will yield the $\bar{X}_3X_4 + X_3\bar{X}_4$. As shown, this can be made with a NAND-to-NAND gate network with three two-input NAND gates. An inverter is also required. (A NAND gate can be used for this by connecting both inputs.)

One problem remains: We need to start the counter with X_1 in state 1 and the other three flip-flops in state 0. Since DC RESET inputs are connected on the flip-flops [see Fig. 4.16(*b*)], it is necessary to use a trick for flip-flop X_1. This simply involves renaming the J and K inputs and the two outputs so that J becomes K, K becomes J, and the two output names are reversed. The DC RESET input then becomes a DC SET input for the new (renamed) flip-flop.

Figure 4.18 shows the circuit as finally designed. Notice how X_1 differs in connections from X_2 and X_3.

The logic circuit in Fig. 4.18 could be implemented by using a printed-circuit board to make the connections between IC containers. Or the connections could be made by individual wires by using any of a number of interconnection boards manufactured by various companies. Placing a 0 (ground) on the DC RESET input sets the flip-flops to the desired starting conditions, and the circuit will then step through the desired states.

There are several major lines of integrated circuits now being produced in substantial quantities. Table 4.1 lists several basic lines and gives some of the characteristics of each line. The first three IC lines in the table are called *bipolar logic* because they utilize conventional transistors in the IC packages, and the next three of the lines use what are called *field-effect transistors* (FETs) and are fabricated by using metal-oxide semiconductor (MOS) technology. IIL is also bipolar

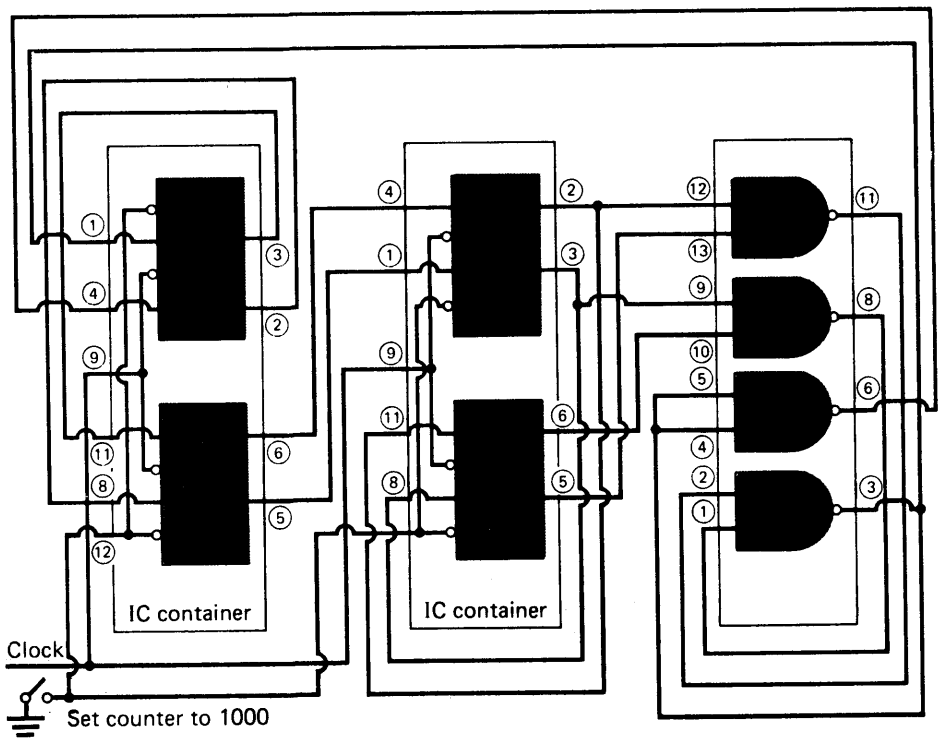


FIGURE 4.18

Design of shift register in Fig. 4.17 using TTL. Circled numbers are pin numbers on IC containers.

but nonstandard in operation. The bipolar logic lines are widely used for constructing configurations on circuit boards which realize high-speed logic. Generally there are not so many gates and flip-flops in a package using bipolar logic, but these lines are fast and can be interconnected more readily than the MOS lines. The reason is that the bipolar logic lines use more power (primarily more current) for each gate or flip-flop and can, as a result, produce more current drive and therefore drive long cables, long wires, and, in general, more other circuits.

Associated with each gate and flip-flop in a line of integrated circuits are data concerning the gates or flip-flop's ability to drive other circuits and be driven by other circuits. Typically the manufacturer gives data concerning the delays through the circuit, rise and fall times for output waveforms, the circuit's ability to drive other electrical loads, circuits, and long wires or cables. The manufacturer also generally provides information on how many other inputs to gates of a similar type a given gate can drive. In its simplest form, every input to every gate and flip-flop is the same, and the manufacturer simply stipulates how many inputs can be connected to a given output. Each input is then called a *standard load*, and an output is said to be able to drive, for instance, eight standard loads. For some circuit lines, different gates and flip-flop inputs present different loads, and so an input to a particular kind of gate might have a number such as 2 or 3 associated with it and an output drive number such as 12. Then the designer must see that the sum of the input loads does not exceed the output drive number for a given output.

TABLE 4.1 INTEGRATED-CIRCUIT LINES

NAME OF CIRCUIT LINE	ACRONYM	SPEED (DELAY PER GATE), NS	POWER PER GATE	GOOD FEATURES	PROBLEMS
Transistor-transistor logic	TTL	4 (3 ns for Schottky clamped)	10 mW (20 mW for Schottky clamped)	Very popular, widely represented. Easy to interconnect. Wide selection of functions available. MSI packages relatively inexpensive.	Generates noise spikes, relatively high power dissipation, modest packing density.
Low-power transistor-transistor logic	LPTL	10	1 mW	A (low-power) TTL and TTL developed for calculators and other portable applications, easy to use and interconnect.	Low speed.
Emitter-coupled logic	ECL	0.3	60 mW	Highest speed, generates little noise internally.	Difficult to interconnect. Low packing density. Difficult to cool.
p-channel metal-oxide silicon	PMOS	25	0.1 mW	Low power, good packing density, easy to manufacture, inexpensive.	Slow and delicate, has limited ability to drive lines and to interface with other circuit lines.
n-channel metal-oxide silicon	NMOS	7	0.1 mW	Faster than PMOS, relatively low power, good packing density, inexpensive, and relatively easy to manufacture.	Has limited ability to drive lines and to interface with other circuits.
Complementary metal-oxide silicon	CMOS	7	10 nW	Very low standby power required. Medium speed and packing density, reasonably priced.	Power consumption increases when switched at high speeds.
Integrated injection logic	IIL	1	0.1 mW	Fast, low power.	Difficult to manufacture. Non-standard logic gate formations.

Note: In this table W stands for watts, n for nano ($= 10^{-9}$), m for milli ($= 10^{-3}$), and s for seconds.

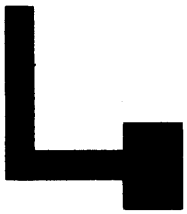


Figure 4.19 shows a binary counter which is packaged in a single IC container with 16 pins. The counter has several features:

1 The counter counts up (from 0000 to 1111) if $U/\overline{D}$ is a 1 and down (from 1111 to 0000) if $U/\overline{D}$ is a 0.

2 The four flip-flops can be "loaded" from the four DATA inputs by making the LOAD line a 1 when a clock pulse occurs (the LOAD line is normally a 0, so making it a 1 causes the upper DATA input values to be taken by the upper flip-flops, etc.).

3 The counter can be gated on or off by the two ENABLE lines.

MEDIUM-, LARGE-, AND VERY LARGE-SCALE INTEGRATION

4.11 Most circuits are now fabricated by using the general technology of integrated circuitry. In this case the transistors, diodes, resistors, and any other components are fabricated together, by using solid-state physics techniques, in a single container. In the most common technology, called *monolithic integrated circuitry*, a single semiconductor wafer is processed by photomasking, etching, diffusions, and other steps, thus producing a complete array of diodes, transistors, and resistors already interconnected to form one or more logic gates or flip-flops.

When more than a few flip-flops or a few gates are packaged in a single container, the process is called *medium-scale integration* (MSI). Notice that medium-scale integration still refers to integrated circuits, except that even more circuits are housed in a single container. There are no fixed specific rules, but generally if more than 10 but less than 100 gates or flip-flops are in a single package, the manufacturer will refer to it as MSI.

When more than 100 gates or flip-flops are manufactured in a single small container, the process is called *large-scale integration* (LSI). Some ideas of the complexity of arrays of this sort are found in later chapters where memories and arithmetic-logic units consisting of thousands of flip-flops and gates in a single package are studied.

Finally, there is very large-scale integration (VLSI) in which 50,000 to several hundred thousand gates and flip-flops are packaged in a single package. The memories and microprocessors in later chapters will illustrate this.

Despite the various levels of integration, the circuits are surprisingly similar in principle, except that VLSI tends to use a technology based on MOS, while MSI, LSI, and "conventional" integrated circuits use "conventional" *npn* and sometimes *pnp* transistors fabricated on silicon chips. There are good reasons for this. MOS circuits require very small areas on a chip and use very little power, which is quite important given the volume/complexity factor. However, conventional bipolar circuits are faster and more readily interconnected. As a result, the MOS technology is used more often for larger arrays which can be treated only as complete single units rather than on a circuit-by-circuit basis. MOS is more likely to be used in large memories and microprocessors, for example.

Figure 4.20(*a*) shows a typical MSI package containing a complete BCD counter. This counter steps from 0 to 9 and then resets to 0 when X_1 (which is pin

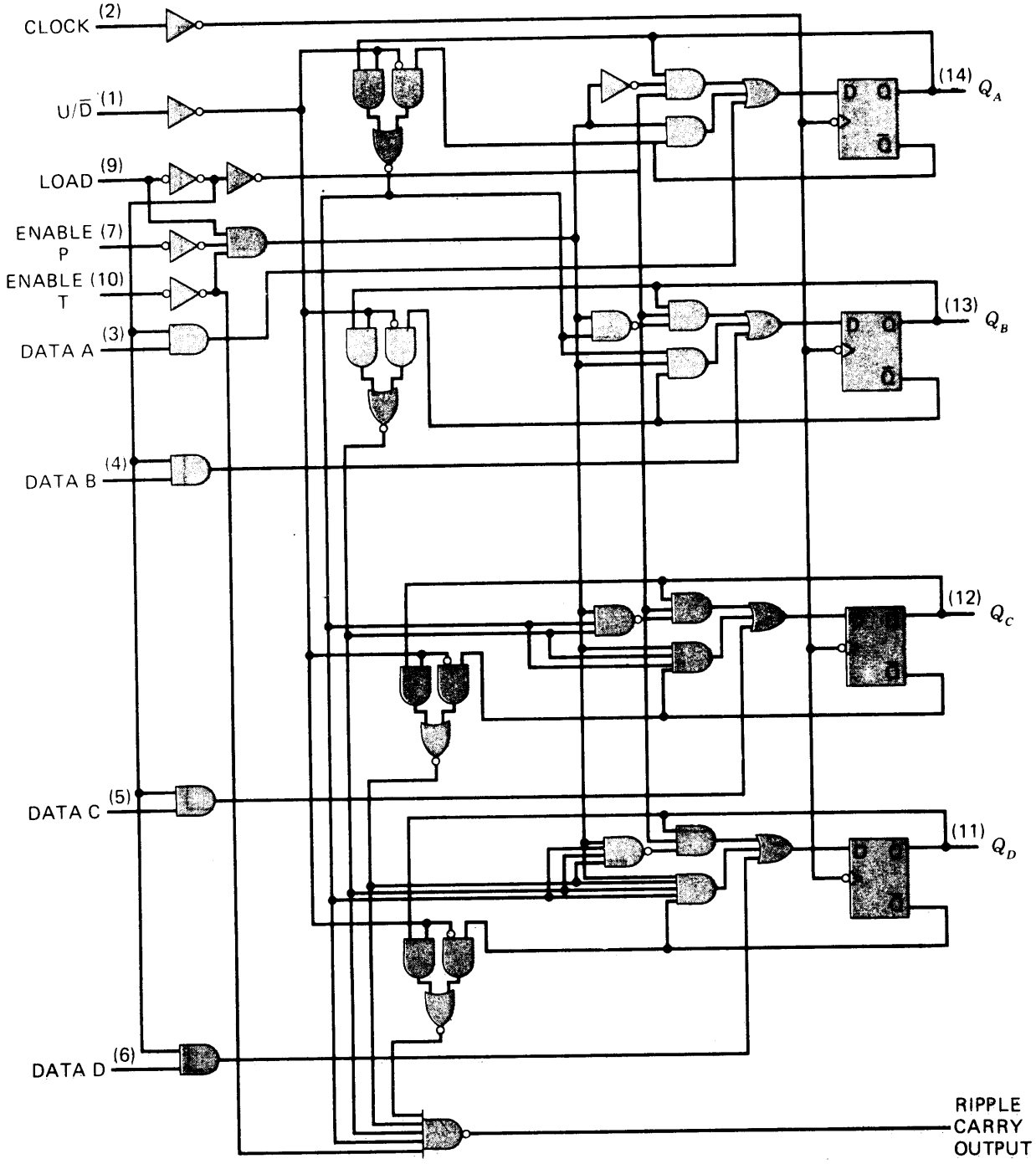


FIGURE 4.19

Binary counter in IC container.

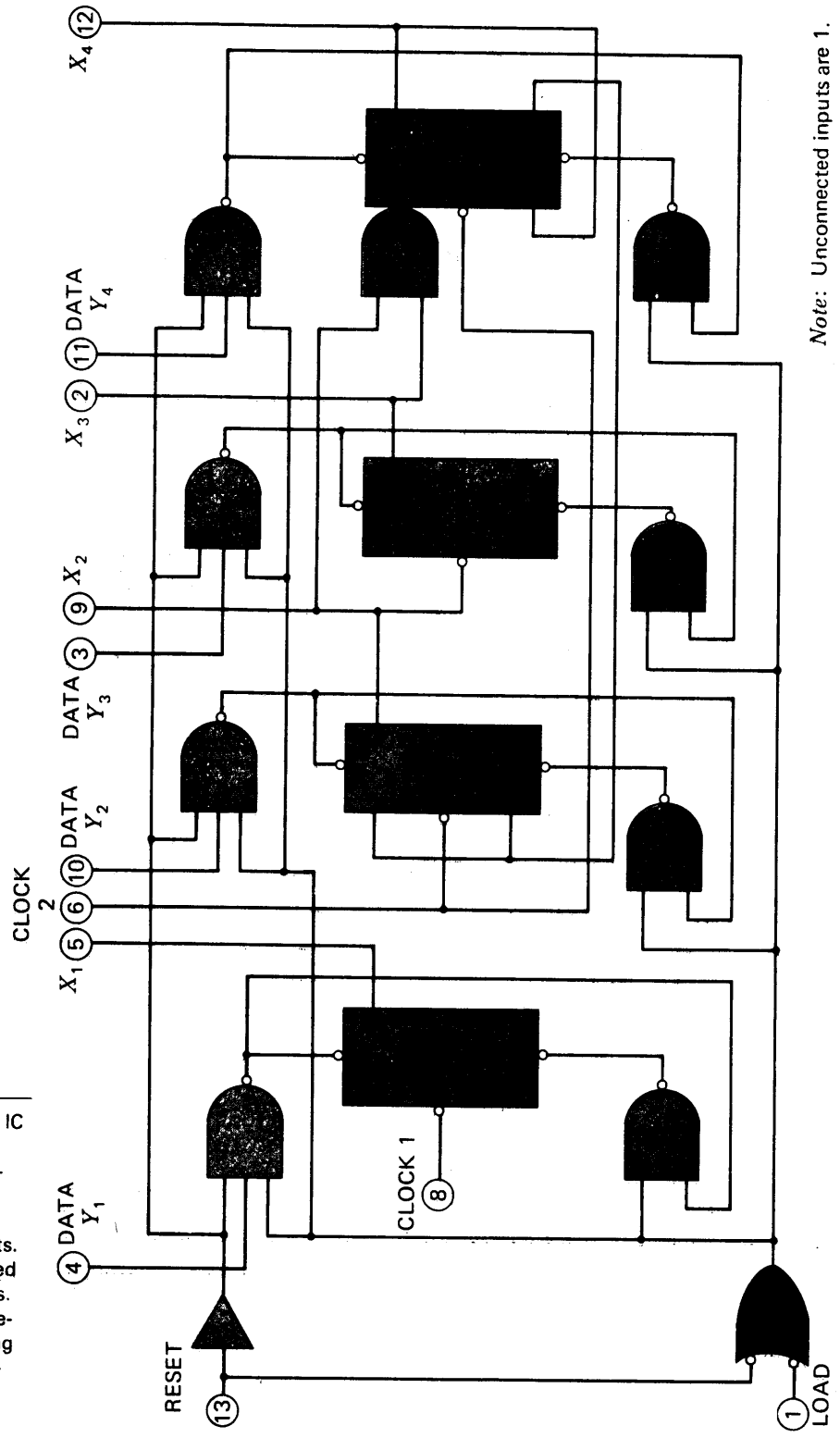
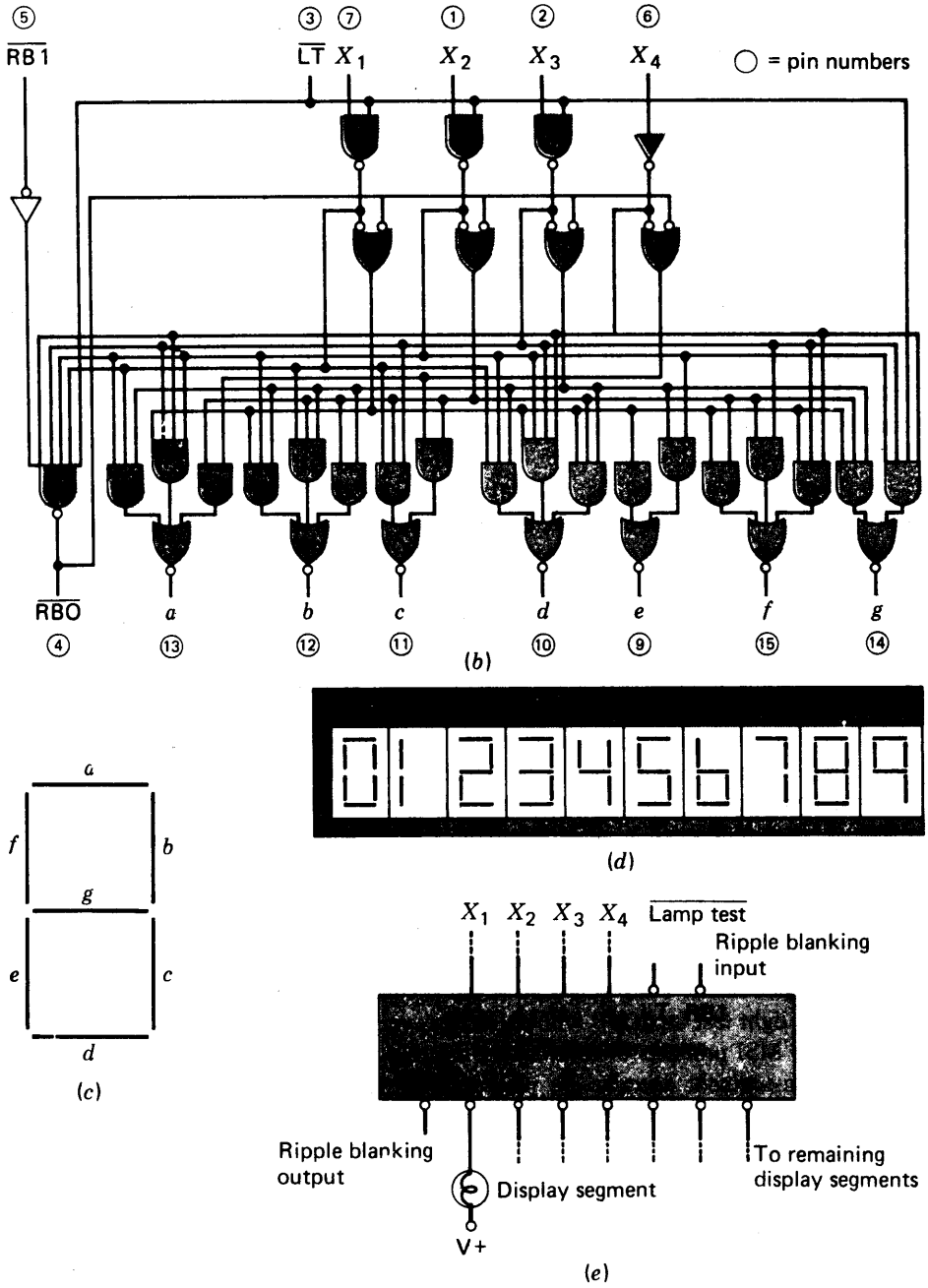


FIGURE 4.20

(*a*) BCD counter in IC package. (*b*) Logic diagram for seven-segment decoder. (*c*) Designation for the seven segments. (*d*) Numbers formed by seven segments. (*e*) General arrangement for connecting seven-segment decoder. (*Fairchild Semiconductor.*)



5) is connected to clock 2 (which is pin 6). The counter is stepped each time an input clock waveform connected to clock 1 (pin 8) goes negative (on negative edges). The counter can be reset to the all 0s by connecting a 0 to the reset line (pin 13). Data from four input wires connected to Y_1, Y_2, Y_3, and Y_4 will be loaded into flip-flops X_1, X_2, X_3, and X_4, respectively, if the LOAD input is pulled down to a 0. (It is normally a 1.)

An example of gate networks in MSI packages is shown in Fig. 4.20(*b*), which shows a *seven-segment decoder*. When decimal numbers are to be read from a digital calculator, instrument, microcomputer, etc., display devices using light-emitting diodes (LEDs) or liquid crystals are often used. Each digit of the display is formed from seven segments, each consisting of one light-emitting diode or crystal which can be turned on or off. A typical arrangement is shown in Fig. 4.20(*c*) which assigns the letters *a* through *g* to the segments. To make the digit 5, for example, segments *a*, *f*, *g*, *c*, and *d* are turned on. The set of digits as formed by these segments is shown in Fig. 4.20(*d*).

The seven-segment decoder in Fig. 4.20(*b*) can be connected to the outputs of the four flip-flops in the BCD counter in (*a*) by connecting the X_1, X_2, X_3, and X_4 outputs from (*a*) to the X_1, X_2, X_3, and X_4 inputs of (*b*). If the seven outputs *a* through *f* of Fig. 4.20(*b*) are then connected to a decimal digital display device, a counter with a decimal digit display, such as those in the familiar calculator, will be formed.

The BCD counter in Fig. 4.20(*a*) can be extended to several digits by connecting the X_4 output from one digit to the clock 1 input of the next-highest-order digit in the counter.

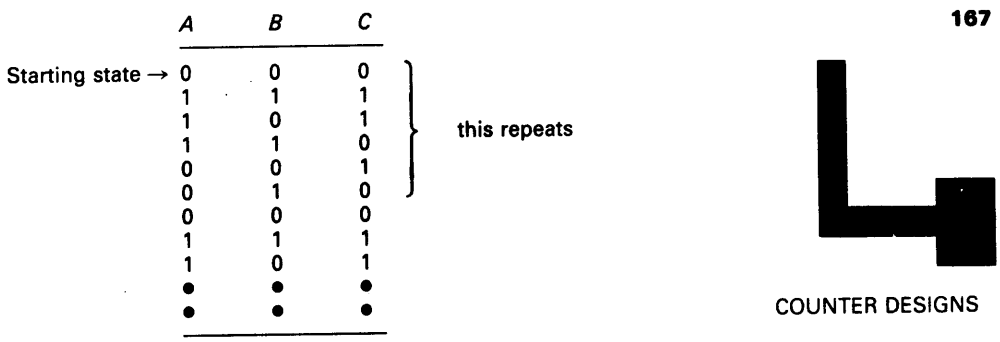
The seven-segment decoder in Fig. 4.20(*b*) has the ability to blank leading zeros in a multidigit display, which is commonly done on calculators. Consider that a multistage BCD counter has been connected to several seven-segment decoders with one decoder per BCD counter stage. If the ripple blanking output (RBO) of each seven-segment decoder is connected to the ripple blanking input (RBI) of the seven-segment decoder of the next-higher-order digit in the counter, and if the ripple blanking input of the most significant digit's seven-segment decoder is connected to a 0 input, then a blanking circuit will be formed. Then, for instance, in a four-stage counter the number 0014 will have the leading two 0s turned off, or *blanked*; the number 0005 will be displayed as simply 5, with the 0 displays not turned on, etc. In effect, the circuit tests for a 0 value at its input. If the value is 0 and all the digits to its left are 0, then it turns off all seven segments and generates a blanking signal for the next rightmost digit's light driver. The light test (LT) input can be used to test all seven segments simultaneously. Notice that making the light test input a 0 will cause all seven segments to go on.

In the MSI products of some manufacturers, both a four-stage BCD counter and a seven-segment decoder are placed in the same package complete with a blanking input and output for each digit. This gives some feeling for the more complex MSI packages.

COUNTER DESIGN

***4.12**[5] The design of a counter to sequence through a given set of states is straightforward, using the technique to be shown. First, a counter table is made up that lists the states to be taken. Assume that we wish a counter using three flip-flops to sequence as follows:

[5]Sections marked with asterisks can be omitted on a first reading without loss of continuity.



	A	B	C	
Starting state →	0	0	0	
	1	1	1	
	1	0	1	this repeats
	1	1	0	
	0	0	1	
	0	1	0	
	0	0	0	
	1	1	1	
	1	0	1	
	•	•	•	
	•	•	•	

This table shows that if the counter is in the state $A = 0, B = 0, C = 0$ and a clock pulse (edge) is applied, then the counter is to step to $A = 1, B = 1, C = 1$. As another example, if $A = 0, B = 1$, and $C = 0$ and a clock pulse occurs, then the counter is to step to $A = 0, B = 0, C = 0$. As can be seen, the counter "cycles" because after taking the state 010 it returns to 000 and then goes to 111, as before. If clock pulses continue, the counter will cycle through the six different states shown indefinitely.

We use *RS* flip-flops for our first design. Now each flip-flop has two inputs, an *R* input and an *S* input. So we give the *R* input to *A* the name A^R, the *S* input to *A* the name A^S, the *R* input to *B* the name B^R, and so on through C^S.

The problem is now to derive boolean algebra expressions for each of the six inputs to the flip-flops. To do this, we place the state table in a *counter design table*, listing the three flip-flops and their states and also listing the six inputs to the flip-flops. This is shown in Fig. 4.21(*a*).

The values for $A^R, A^S, B^R, B^S, C^R, C^S$ are then filled in by using the following rule.

DESIGN RULE

Consider a row in the table and a specific flip-flop:

1. If the flip-flop's state is a 0 in the row and a 0 in the next row, place a 0 in the *S* input column and a d in the *R* input column for the flip-flop inputs.
2. If the flip-flop is a 1 in a row and a 1 in the next row, place a 0 in the *R* input column and a d in the *S* input column.
3. If the flip-flop is a 0 in a row and changes to a 1 in the next row, place a 1 in the *S* column and a 0 in the *R* column.
4. If the flip-flop is a 1 in a row and changes to a 0 in the next row, place a 1 in the *R* column and a 0 in the *S* column.

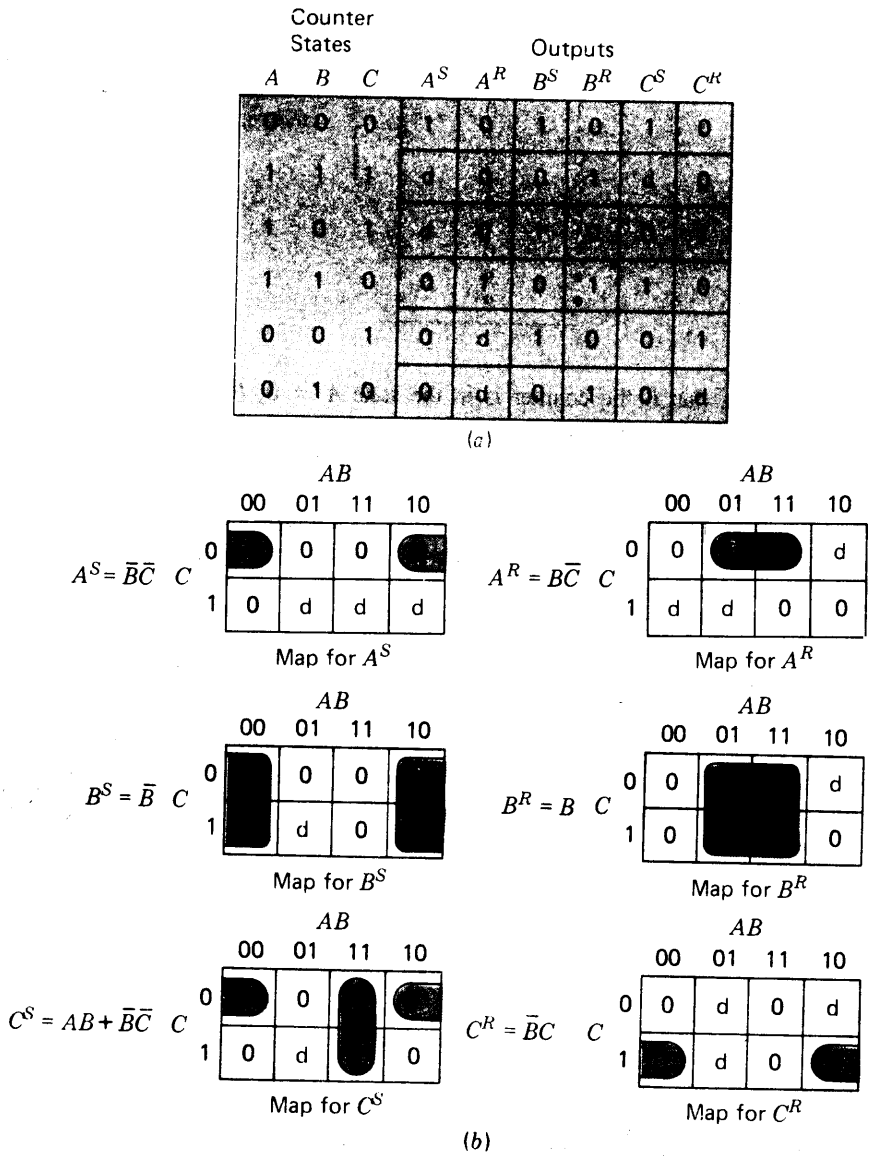
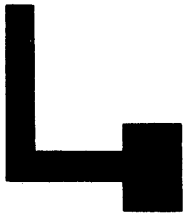


FIGURE 4.21

Designing a counter using *RS* flip-flops.

As an example, consider flip-flop A in Fig. 4.21(a). The flip-flop has value 0 in the first row and changes to a 1 in the second row. We therefore place a 1 in A^S in row 1 and a 0 in A^R in row 1. In row 2, A has value 1 and remains a 1 in row 3. So we fill in a d in A^S and a 0 in A^R.

The reasoning behind these rules is as follows: Suppose that flip-flop A is in the 0 state and should stay in the 0 state when the next clock pulse is applied. The S input must then be 0, and the R input can be a 1 *or* a 0. Thus we must have 0 at A^S, but can place a d (for don't-care) at the A^R input.



If A is a 0 and should change to a 1, however, the S input to A must be a 1 and the R input a 0 when the next clock pulse is applied. So a 1 is placed in A^S and 0 in A^R.

It is instructive to examine several of the entries in the counter table in Fig. 4.21(a) to see how this rule applies.

Our goal is to generate the flip-flop inputs (A^R, A^S, etc.) in a *given* row so that when the counter is in the state in that row, each input will take the value listed. Then the next clock pulse will cause the counter to step to the state in the next row below in the counter table.

The design of the counter now progresses as a boolean algebra expression is formed from this table for A^R, A^S, B^R, B^S, C^R, and C^S, the inputs to the flip-flops, and then each expression is minimized. This is shown in Fig. 4.21(b), which gives the maps for the flip-flops' inputs. Notice that any unused counter states can be included as d's in the map because the counter never uses them. The minimal expressions are shown beside each map.

The final step is to draw the block diagram for the counter using the minimal expressions. The final design for this counter is shown in Fig. 4.22.

Now suppose that we desire to design the above counter, using *JK* flip-flops. The procedure will be basically the same, except that the rules for filling in the counter design using *JK* flip-flops will be different.

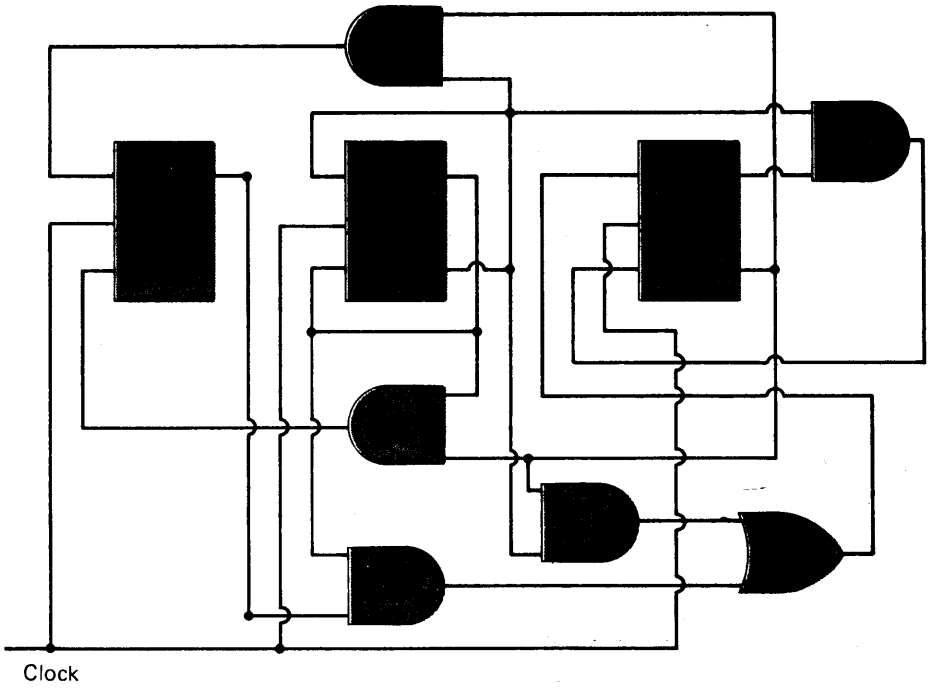


FIGURE 4.22

Counter with *RS* flip-flops.

The inputs will now be A^J, A^K, B^J, B^K, C^J, and C^K. The rules for *JK* flip-flops are as follows:

DESIGN RULE

For a given flip-flop in a selected row, the *J* and *K* inputs to the flip-flop are filled in as follows:

1. If the flip-flop is a 0 in a row and remains a 0 in the next row, place a 0 in the *J* input column and a d in the *K* input column.

2. If the flip-flop is a 1 in a row and remains a 1 in the next row, place a 0 in the *K* input column and a d in the *J* input column.

3. If the flip-flop is a 0 in a row and changes to a 1 in the next row, place a 1 in the *J* input column and a d in the *K* input column.

4. If the flip-flop is a 1 in a row and changes to a 0 in the next row, place a d in the *J* input column and a 1 in the *K* input column.

The reasoning behind the above rules is as follows: Suppose that a given flip-flop, say *A*, is in the 0 state and should stay in the same state when the next clock pulse occurs. The input A^K must be a 0 at that time, but A^J can be either a 0 at that time or a 1, so that A^J input is essentially a d (or don't-care) input. If *A* must go from a 0 to a 1, however, the A^K input *must* be a 1, but the A^J input can be either a 0 or a 1 (since the flip-flop will change states if both inputs are 1s). Notice that there are more d's in the rules for *JK* flip-flops than for *RS* flip-flops because of the ability of the flip-flops to change states when both inputs are 1s.

The maps for each input to the flip-flops A^J, A^K, B^J, B^K, C^J, and C^K are drawn as before, the expression for each flip-flop's input is minimized, and the block diagram for the counter is then drawn as in Fig. 4.23. Notice that fewer gates are used for the counter in Fig. 4.23 than for that in Fig. 4.22. This is because of the additional d's in the maps, and it will generally, although not always, be the case. (Sometimes the *RS* and *JK* designs will be the same; *JK* flip-flops cannot require more gates for a given counter sequence.)

STATE DIAGRAMS AND STATE TABLES

4.13 A set of interconnected gates with inputs and outputs is called a *combinational network*. The outputs from a combinational network at a given time are completely determined by the inputs at that time. As a result, the function of a combinational network can be described by using a table of combinations that simply lists the input-output values.

When flip-flops are combined with gates, a more complicated situation arises because the flip-flops progress through various states—depending on inputs—and the output can depend on the previous as well as the preceding inputs.

A	B	C	A^J	A^K	B^J	B^K	C^J	C^K
0	0	0	1	d	1	d	1	d
0	0	1	d	d	1	d	d	1
0	1	0	0	d	d	1	0	d
0	1	1	d	d	d	1	d	1
1	0	0	0	d	0	d	0	d
1	0	1	d	0	0	d	d	1
1	1	0	d	1	d	1	1	d
1	1	1	0	d	1	d	d	1

$A^J = \bar{B}\bar{C}$

C \ AB	00	01	11	10
0	1	0	d	d
1	0	d	d	d

$A^K = \bar{C}$

C \ AB	00	01	11	10
0	d	d	1	d
1	d	d	0	0

$B^J = 1$

C \ AB	00	01	11	10
0	1	d	d	d
1	1	d	d	1

$B^K = 1$

C \ AB	00	01	11	10
0	d	1	1	d
1	d	d	1	d

$C^J = A + \bar{B}$

C \ AB	00	01	11	10
0	1	0	1	d
1	d	d	d	d

$C^K = \bar{B}$

C \ AB	00	01	11	10
0	d	d	d	1
1	1	d	0	d

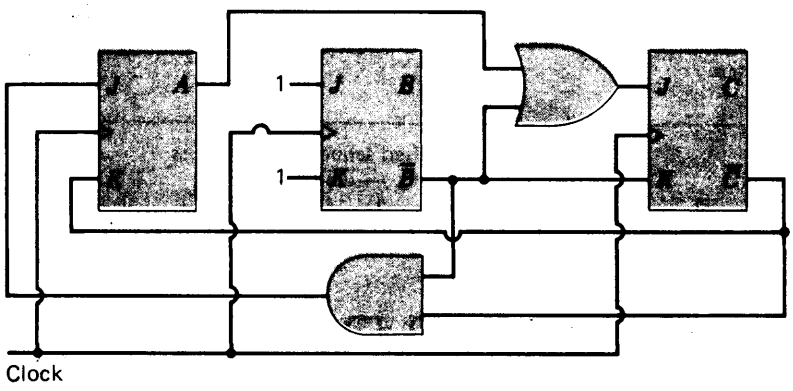


FIGURE 4.23

Design for a *JK* flip-flop counter.

To analyze and design with both flip-flops and gates, several techniques have been developed. The best known involves the use of *state diagrams* and *state tables*, which are the subject of this section.

A simple design problem which requires flip-flops as well as gates is the design of a *binary sequence detector*. A binary sequence detector has a single input line that it examines. The sequence detector looks for some specified sequence of inputs on this input line and outputs a 1 when this sequence is found.[6] An example of a specified sequence would be three consecutive 1s. In this case, if the sequence detector is present with the inputs 1001011101, the sequence detector will output a 0 at all times except immediately following the third 1, when it will output a 1. This is shown in Fig. 4.24. The sequence detector is like a lock which unlocks (outputs a 1) only when the combination (in this case, three consecutive 1s) appears. Sequence detectors can be designed to detect any specified sequence such as 11011 or 1110101 or any other.

Figure 4.25(*a*) shows a state diagram which describes a binary sequence detector that detects three consecutive 1s. A state diagram is formed from what mathematicians call a *directed graph*. State diagrams have *nodes*, which are the circles in Fig. 4.25(*a*), and *links*, which are the curved lines with arrowheads at one end. There are four nodes in Fig. 4.25(*a*) and eight links.

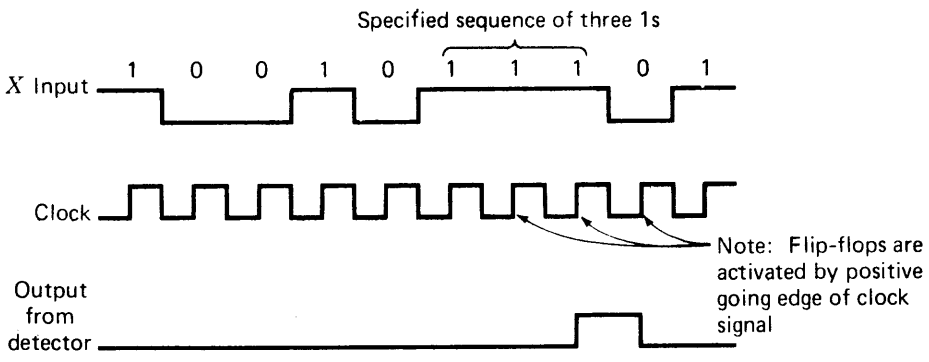
The nodes in a state diagram correspond to flip-flop states in the final design and so are also called *states* and given names. For Fig. 4.25(*a*), the states are named A, B, C, and D. To the right of each state name there is a comma, followed by the output value for that state. This corresponds to the output from the detector in Fig. 4.24. For this diagram, if the present state is A, the output is 0; if the state is D, the output is 1; etc. Each link of the graph is labeled with the input values $X = 1$ or $X = 0$. These links show how transitions are made from state to state. This X input in Fig. 4.25(*a*) corresponds to the X input in Fig. 4.24.

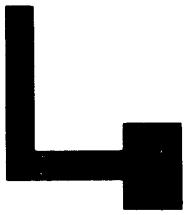
The interpretation of the state diagram corresponding to Fig. 4.25(*a*) is as follows. The machine is started in state A, at which time the output is a 0. If the input X is 0 when the first clock pulse arrives, the detector stays in state A and continues to output a 0. This is shown by the loop connected to A and labeled $X = 0$.

[6]The sequence of 1s and 0s on the input line occurs in time, and each 1 and 0 is generally clocked into the flip-flops.

FIGURE 4.24

Input and output waveforms for binary sequence detector.





If the detector is in state A and a 1 is input (when the clock pulse arrives), the system goes to state B and continues to output a 0.

With the detector in state B, if a 0 is input, the detector goes back to state A and continues to output a 0. If a 1 is input with the detector in state B, the detector goes to C and continues to output a 0.

This analysis of the detector's operation can be continued. The important thing is that if the detector is in state C and an input of 1 is given, the detector goes to state D and outputs a 1. If more 1s are input with the machine in state D, it remains in that state and continues to output a 1. If a 0 is input, the detector returns to A.

As can be seen, the detector outputs a 0 until three successive 1s are input, at which time it outputs a 1 and this 1 output is continued until a 0 is input.

Figure 4.25(b) is a *state table* which represents the same sequence detector as the state diagram in Fig. 4.25(a). There are three major columns in the table: Present state, Output, and Next state. The interpretation of this table is as follows. If the detector is in present state A and a 0 is input, the next state will be A and a 0 will be output. If the system is in state A and a 1 is input, the detector will go to state B and a 0 will be output while in that state.

If the detector is in state A and two successive 1s are input, the resulting state will be C. If another 1 is input, the detector will go to the D state and a 1 will be output. While in D, the 1 inputs will keep the system in D and the outputs will continue to be 1s until a 0 is input; then the next state will be A and a 0 will be output. The state diagram in Fig. 4.25(a) and the state table in Fig. 4.25(b) should be compared to see how they describe the same operations.

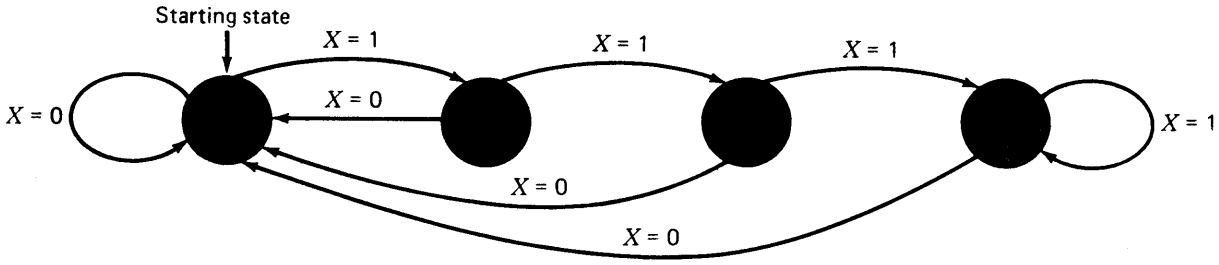
Having described the sequence detector by using a state diagram and a state table, we now make a design using flip-flops and gates which will realize the state diagram and table and which can be constructed by using integrated circuits.

Since flip-flops are to take states corresponding to states A, B, C, and D in the state diagram and table, two flip-flops will be required to take the four states. We can make a preliminary drawing of the layout for the sequence detector. Figure 4.25(c) shows the overall layout with two flip-flops, a set of gates, and an input X and an output Z. What remains is to design the gate network in Fig. 4.25(c). First, however, it is necessary to assign values to the flip-flops for each of states A, B, C, and D.

A natural assignment of flip-flop values is to let $Q_1 = 0$, $Q_2 = 0$ represent state A; $Q_2 = 0$, $Q_2 = 1$ represent B; $Q_1 = 1$, $Q_2 = 0$ represent C; and $Q_1 = 1$, $Q_2 = 1$ represent D. Replacing the A, B, C, and D in Fig. 4.25(b) with this assignment of values leads to the table in Fig. 4.25(d), which is otherwise the same as Fig. 4.25(b).

It is now possible to design the actual gate structure. There are three inputs to the gating network: Q_1 and Q_2, the flip-flop outputs, and the X input. There are also three outputs from the gate network: the D inputs to Q_1 and Q_2 and the Z output. Since there are three outputs, three maps are required. The maps for D_1 and D_2 (the inputs to Q_1 and Q_2) will have three inputs, Q_1, Q_2, and X. However, the map for the output has only two inputs, Q_1 and Q_2, since the output Z is determined by the present state of the system and not the current input.

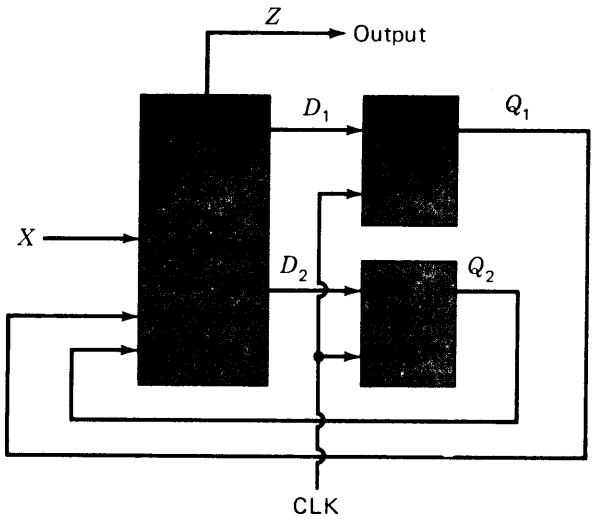
The maps for the system in Fig. 4.25(a) through (d) are shown in Fig. 4.25(e), and the complete design is seen in Fig. 4.25(f). A RESET has been added



(a)

Present state	Output Z	Next state	
		Input X	
		0	1
A	0	A	B
B	0	A	C
C	0	A	D
D	1	A	D

(b)



(c)

Present state Q_1 Q_2	Output Z	Next state	
		Input X	
		0	1
00	0	00	01
01	0	00	10
10	0	00	11
11	1	00	11

(d)

X \ Q_1 Q_2	00	01	11	10
0	0	0	0	0
1	0	1	1	1

$D_1 = Q_1 X + Q_2 X$

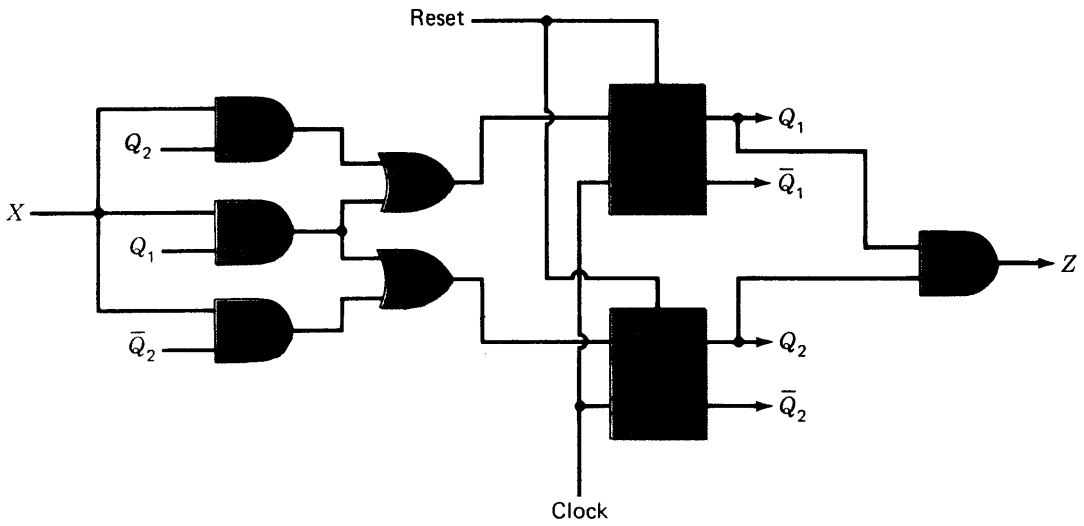
X \ Q_1 Q_2	00	01	11	10
0	0	0	0	0
1	1	0	1	1

$D_2 = Q_1 X + \overline{Q}_2 X$

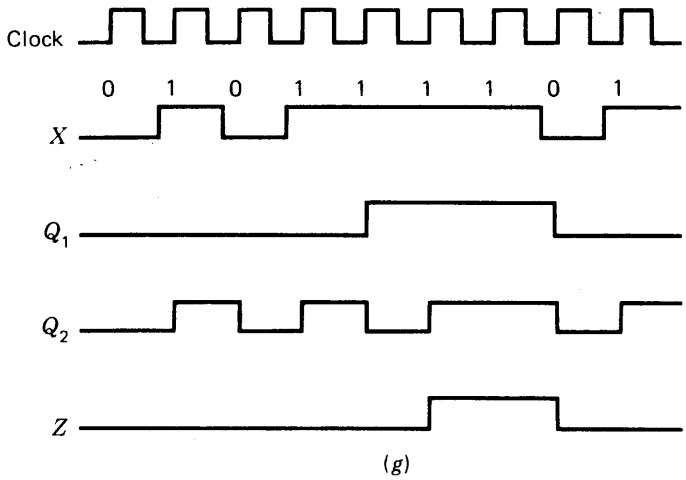
Q_2 \ Q_1	0	1
0	0	0
1	0	1

$Z = Q_1 Q_2$

(e)



(*f*)



(*g*)

FIGURE 4.25

(*a*) State diagram. (*b*) State table. (*c*) Circuit for (*a*) and (*b*). (*d*) State table with assigned values to state flip-flops. (*e*) Maps for design. (*f*) Gates and flip-flops for sequence detector. (*g*) Waveforms for design.

which can be used to start the machine. The operation of this design should be checked by noting the resulting states of the flip-flops for several sequences of X inputs. Figure 4.25(*g*) shows a set of waveforms.

The design in Fig. 4.25(*f*) is an example of a state machine.[7] The same procedure involving state diagrams and state tables can be used to design many things, including interfaces and sections of a computer. In general, a state machine is simply a collection of interconnected flip-flops and gates with a set of inputs and

[7]State machines are often called *finite* state machines in computer science literature, but computer designs and IC manufacturers now generally use the shorter form, *state machine*.

outputs. This is a very general concept, and in following section we treat it in more detail.

DESIGN OF A SEQUENTIAL MAGNITUDE COMPARATOR

4.14 In the preceding section we described the design of a state machine with a single input and output. The same technique can be used to design a state machine with several inputs and outputs.

Consider the design of a *sequential comparator* which is to determine which of two binary numbers A and B, having the same number of bits, is larger. The most significant bits of each number are input to the comparator, followed by the second most significant bits and then the next most significant bits until finally the least significant bits are presented. (The numbers A and B could be stored in two shift registers.) There are to be two outputs, Z_1 and Z_2. If $A > B$, then Z_1 is to be a 1; if $A < B$, then Z_2 is to be a 1; and if $A = B$, then both Z_1 and Z_2 are to be 0s. Figure 4.26 shows a block diagram of the comparator and a set of waveforms for inputs and outputs.

The design of this comparator will be made by first developing a state diagram.

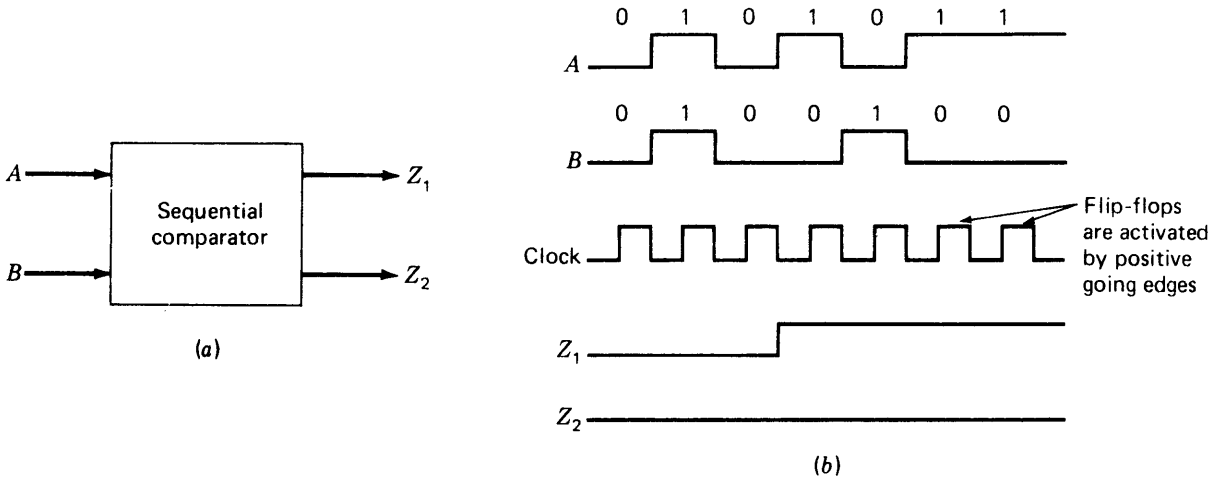
The most significant bits of A and B are to be presented first. If the most significant bit of A is 1 and of B is 0, then $A > B$. So we can draw a starting node K and a link to a state L with the input value 10 (for $A = 1$, $B = 0$); see Fig. 4.27(*a*). The output for the starting state will be $Z_1 = 0$, $Z_2 = 0$, or simply 00; and the output for state L will be $Z_1 = 1$ and $Z_2 = 0$.

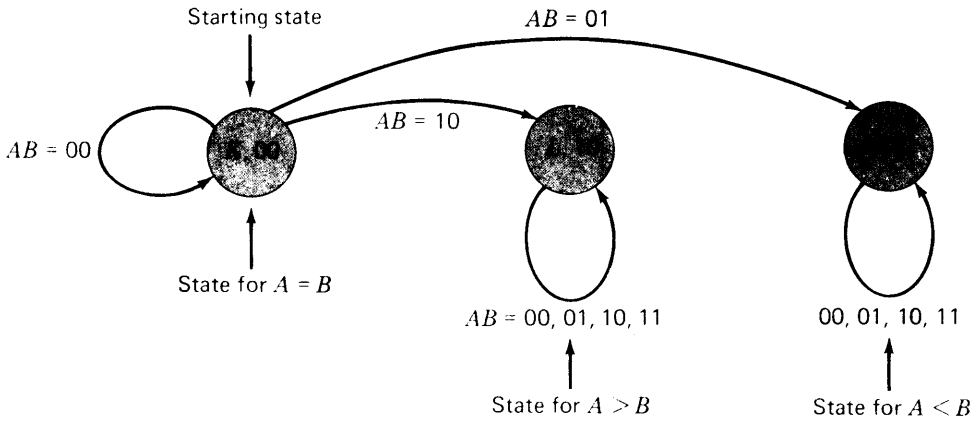
If the most significant bit of A is 0 and of B is 1, then $A < B$ and so a third state M is added with a link with inputs $AB = 01$ going to it from state K.

If the machine is in state L, then no sequence of inputs can change the fact that $A > B$. So a loop with 00, 01, 10, 11 is added to L, indicating it will stay in that state regardless of the inputs. Similarly, if the comparator is in state M, then $A < B$ and no sequence of inputs can change this relation. Thus 00, 01, 10, and 11 are placed on a loop leading from M to M.

FIGURE 4.26

Sequential comparator and waveforms. (*a*) Block diagram. (*b*) Waveforms for two 7-bit binary numbers $A = 0101011$ and $B = 0100100$.





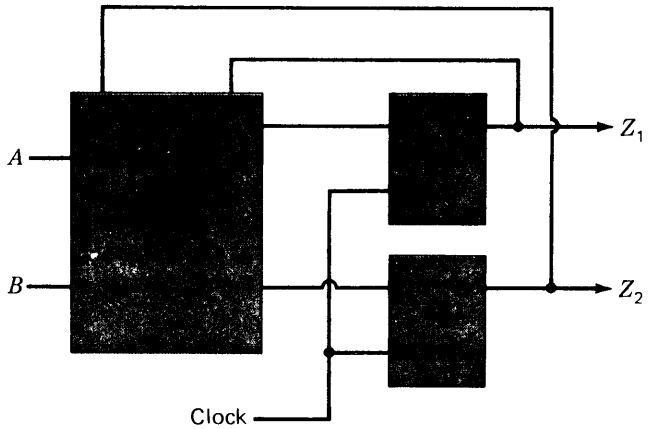
(a)

Present state	Output Z_1	Z_2	Next state inputs AB 00	01	10	11
K	0	0	K	M	L	K
L	0	1	L	L	L	L
M	1	0	M	M	M	M

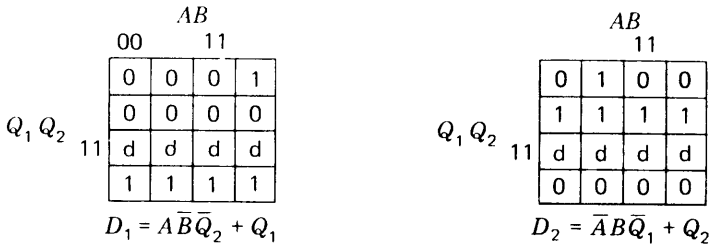
(b)

Present state Q_1 Q_2	Output Z_1	Z_2	Next state inputs A,B 00	01	10	11
00	00		00	01	10	00
01	01		01	01	01	01
10	10		10	10	10	10

(c)



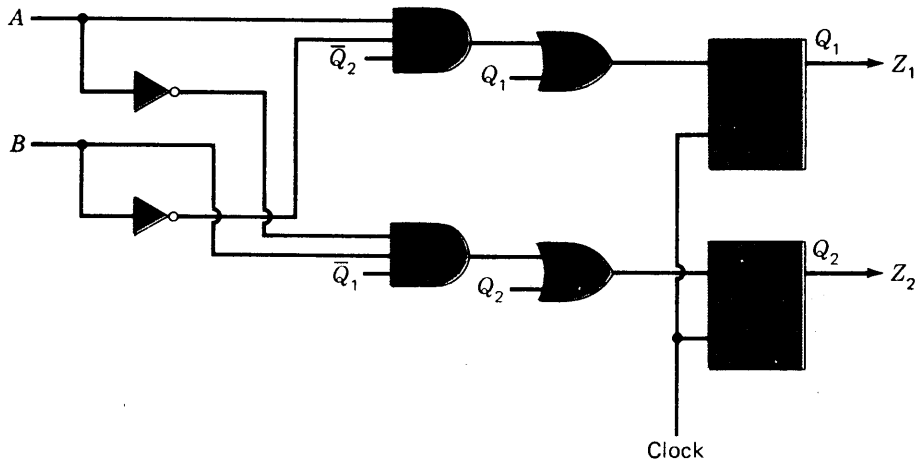
(d)



(e)

FIGURE 4.27

Design for sequential comparator. (*a*) State diagram. (*b*) State table for (*a*). (*c*) State table for sequential comparator with flip-flop values assigned. (*d*) Block diagram of layout for comparator. (*e*) Maps for sequential comparator. (*f*) Gate and flip-flop design for sequential comparator.



(*f*)

FIGURE 4.27

(*Continued*)

If the most significant bits of A and B are the same, then the numbers can be equal or either A or B can be larger. So for inputs of $A = 0$ and $B = 0$, or $A = 1$ and $B = 1$, the comparator has next state K and outputs $Z_1 Z_2 = 00$. We now see that this process will continue for the next significant bits used and all those which follow. So the state diagram is complete. As an example, if $A_2A_1A_0 = 101$, then A has value decimal 5 and if $B_2B_1B_0 = 110$, then B has value decimal 6, and the state diagram will output an 01 after the second bits arrive.

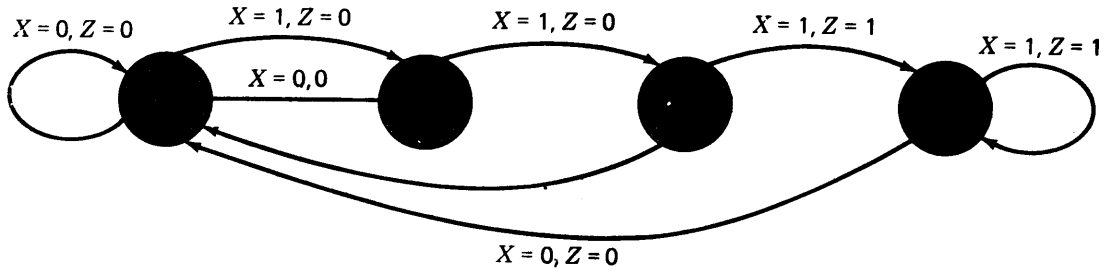
A state table can be made for the state diagram in Fig. 4.27(*a*). This table is shown in Fig. 4.27(*b*).

To design an actual logic network to realize the state table and state diagram for Fig. 4.27(*a*) and (*b*), it is necessary to use flip-flops. Since there are three states, two flip-flops, Q_1 and Q_2, will be used. A natural assignment of states to Q_1 and Q_2 is to let them have the same state as the outputs Z_1 and Z_2. Then no gates are required to produce Z_1 and Z_2 since the Q_1 and Q_2 outputs can be used for this purpose.

The resulting state table is shown in Fig. 4.27(*c*). The overall block diagram for the design is shown in Fig. 4.27(*d*). Only the gating network for the two D inputs to Q_1 and Q_2 needs to be designed since Z_1 and Z_2 are simply outputs from Q_1 and Q_2. The maps for these two inputs are shown in Fig. 4.27(*e*), and the final design is seen in Fig. 4.27(*f*).

COMMENTS—MEALY MACHINES

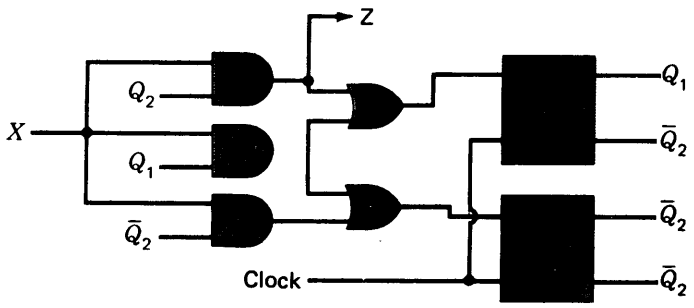
4.15 What have here been called state machines are also called *sequential machines, sequential systems, sequential circuits,* and *finite-state machines.* Many results in the theory of computing about what can and cannot be computed are concerned with what finite-state machines can compute. A finite-state machine



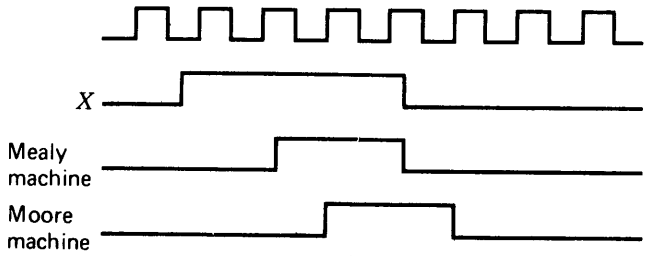
(a)

Present state	Next state		Output	
	X input		X input	
	0	1	0	1
A	A	B	0	0
B	A	C	0	0
C	A	D	0	1
D	A	D	0	1

(b)



(c)



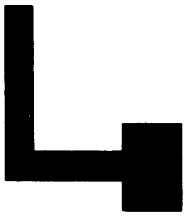
(d)

FIGURE 4.28

Mealy machine design. (*a*) State diagram. (*b*) State table. (*c*) Design for Mealy version of sequence detector. (*d*) Waveforms for Mealy and Moore machine designs.

which can read a tape and write on it is called a *Turing machine*. If the supply of tape for this Turing machine is limitless (the tape is potentially infinite in length), there are many interesting results concerning what the machine can compute, some of which are due to Turing, a brilliant early 20th-century mathematician. The references discuss these results.

There are several variations on the state diagrams and design procedures which have been described. When the outputs are determined by only the state of the flip-flops, as in the two previous designs, the machine is called a *Moore machine*, in honor of Edward Moore. When the outputs are determined by the input value as well as the internal state, the machine is called a *Mealy machine*. Figure 4.28 shows a Mealy machine state diagram, state table, block diagram, and some



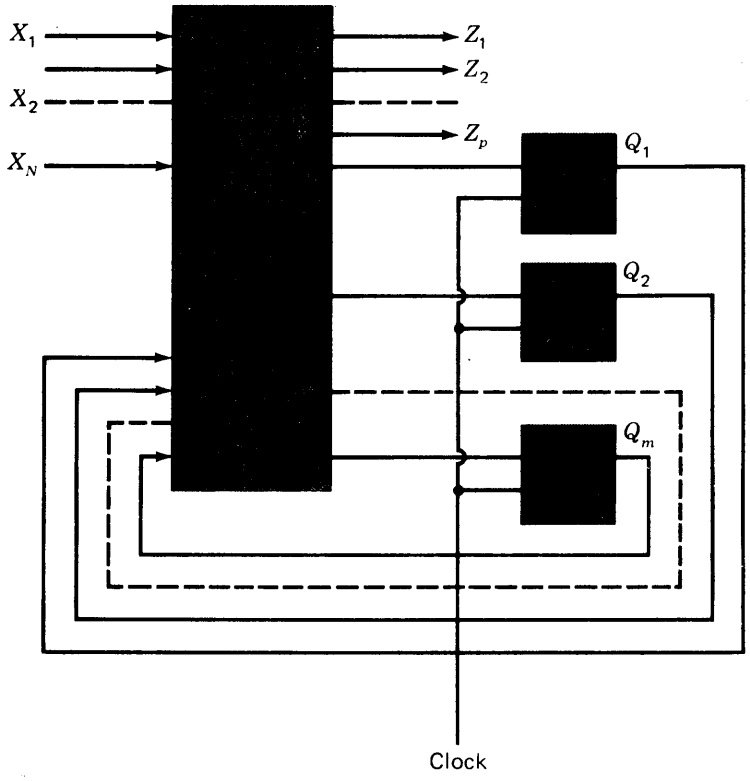


FIGURE 4.29

Block diagram of a state machine.

output waveforms for a machine which is a sequential detector that looks for three 1s (as in Fig. 4.24). Note the output values are on the links, not the nodes, in Fig. 4.28(*a*).

Occasionally it is possible to reduce the number of states in a first design. For completely specified tables, a technique for doing this was discovered by Edward Moore. If not all next states and outputs are specified, the problem is much harder, but Steve Unger solved this problem.[8]

The assignment of values to the flip-flops in the final design can influence the number of gates used, but no one knows how to make the best assignment short of trying every possible assignment. This is called the *assignment problem*.

PROGRAMMABLE ARRAYS OF LOGIC

***4.16** Integrated-circuit manufacturers have found natural ways to implement layouts for state machines on IC chips. The regular shape of the state machine with its gating array preceding the flip-flops makes for a rectangular and ordered

[8]Edward Moore did his work at Bell Laboratories and George Mealy at IBM. Both have changed jobs but are still around, as is Steve Unger who is at Columbia. Detailed descriptions of their work can be found in Birkhoff and Bartee or in Ed McCluskey's book on switching theory.

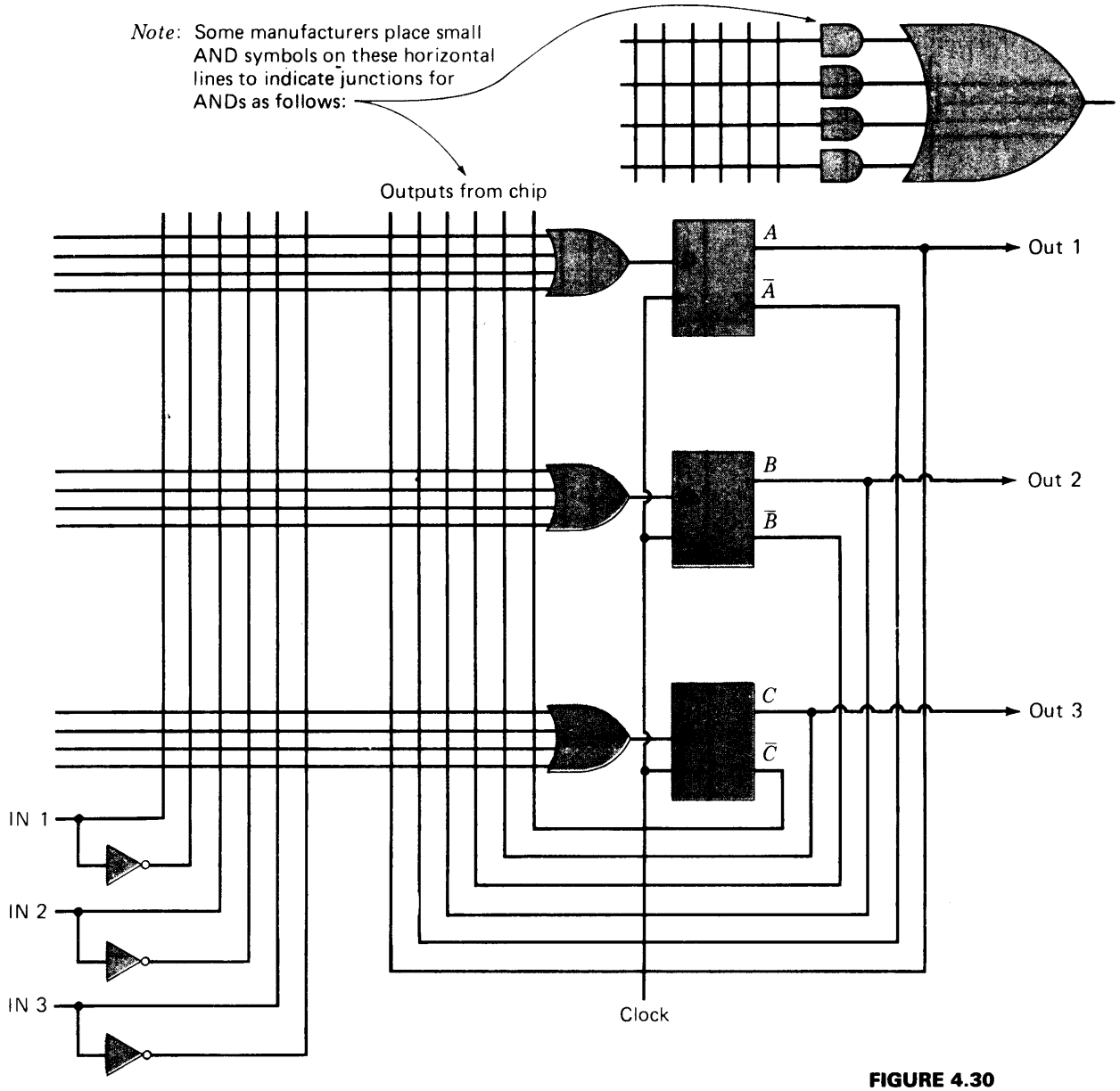


FIGURE 4.30

Programmable array of logic cells.

design for an integrated circuit. As a result, chips are available which can be used to implement state machine designs on a single chip (in a single IC package).

The basic idea is to present the designer with a layout for a design which is sufficiently flexible that most actual designs can be fabricated on this layout.

Figure 4.29 shows the basic block diagram for a state machine. What is needed is a set of flip-flops and a set of gates so that this general structure can be realized for many different designs. Figure 4.30 shows a programmable array of

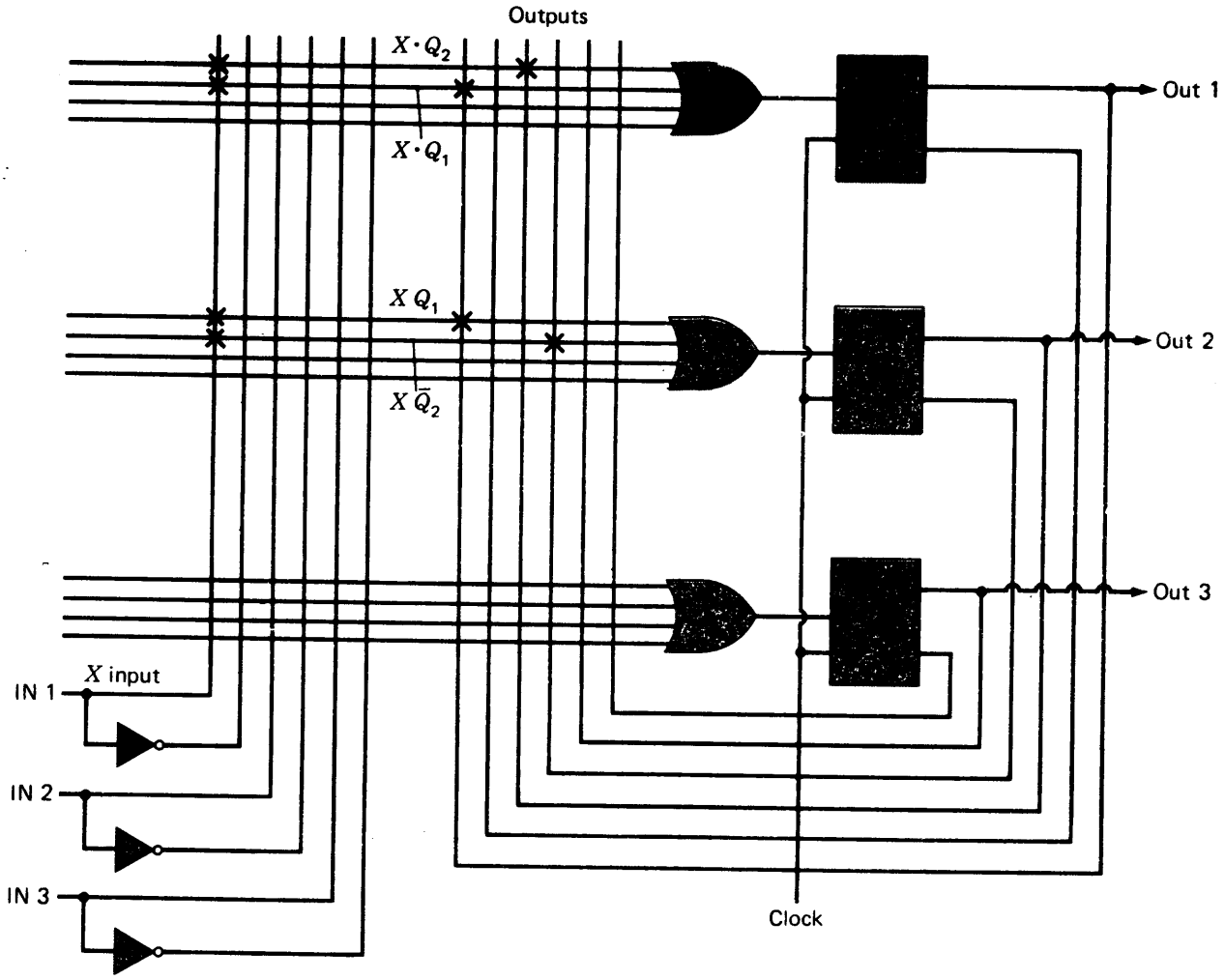


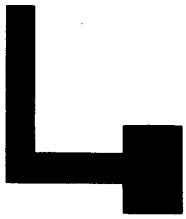
FIGURE 4.31

Design for state machine in Fig. 4.24 using PAL.

logic cells such that a state machine is formed by forming AND gates at the junctions of the vertical lines and connecting the horizontal lines to the OR gates. The particular array shown has three inputs, three flip-flops, and outputs from the flip-flops. The AND-to-OR gate networks are formed in the same way as for the PAL in Fig. 3.44; in fact, this particular array is called a PAL.

Figure 4.31 shows the design using the PAL in Fig. 4.30 for the sequence detector in Fig. 4.25. Since there are three flip-flops and three inputs in Fig. 4.31, only part of the logic in the figure is used. The idea, however, is that by working from a framework such as Fig. 4.31, most designs can be realized. Thus the manufacturer volume for the chip will be great enough that the cost per chip will be low enough to cost less than assembling a given design from several IC packages with fewer gates and flip-flops per package.

The actual IC packages on the market at present have several hundred gates



and flip-flops per package. Sometimes there are several arrays in a single package and, therefore, the possibility to form several machines in a single IC container. Figure 4.32 shows a PAL layout for a state machine from a particular manufacturer.

The gating connection required for a particular design using the layout in Fig. 4.32 can be implemented in two ways: (1) The manufacturer works from a design submitted by the designer, in which case a particular mask or pattern is made for the design and is used for each chip manufactured. (2) The chip can be a user-programmable chip in which every possible connection is made and undesired connections are destroyed by the user (using high currents) blowing fuses in the undesired connections.[9]

Arrays of logic cells such as shown in Fig. 4.32 are becoming increasingly popular and frequently are used in commercial and laboratory equipment. Such *semicustom fabrication processes* are very popular in IC manufacture because of the very high cost of designing and manufacturing completely original, or custom, chips.

SUMMARY

4.17 Flip-flop operation was described along with the operation of the clocks used to initiate operations in computers. Often used flip-flops are the *RS*, *JK*, and *D* flip-flops; each was discussed. Latches were introduced.

Binary counters perform many useful functions in digital machines, and these were discussed in some detail. Also, BCD counters were covered.

The various classes of integrated circuits—medium-scale, large-scale, very large-scale—were described and the circuit lines used in IC packages listed along with their characteristics. A design for a shift register with feedback was implemented by using ICs from a very popular circuit line, TTL.

Sometimes flip-flops are made from gates. We detailed how this can be done for several kinds of flip-flops, including the *JK* edge-triggered flip-flop.

A counter which will count through a given sequence of states can be designed by using the procedure which was presented along with several example designs.

The design and analysis of state machines are facilitated by the use of state tables and state diagrams. A design procedure which can be used to implement a given state table or state diagram was presented.

Some integrated circuits are now being manufactured which are widely used to implement state machine designs because of their particular layout. These ICs enable designers to implement a given design by programming chips (1) by removing undesired connections by blowing fuses in the connections or (2) by submitting a design to a manufacturer, who then implements the design (using IC metallization masks which result in the desired connections being made). This class of ICs has a much higher packing density than IC packages with a few gates or flip-flops per package, costs are lower than for an original design using custom IC packages.

---

[9]This is called *programming* the chip. There are instruments made which allow the introduction of a particular design into a chip by blowing the selected fuses.

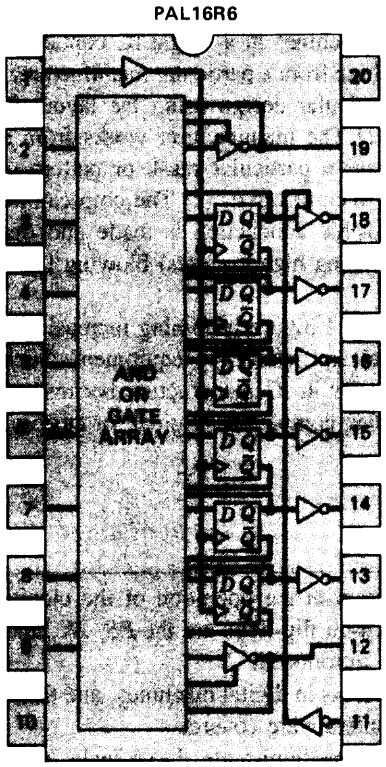
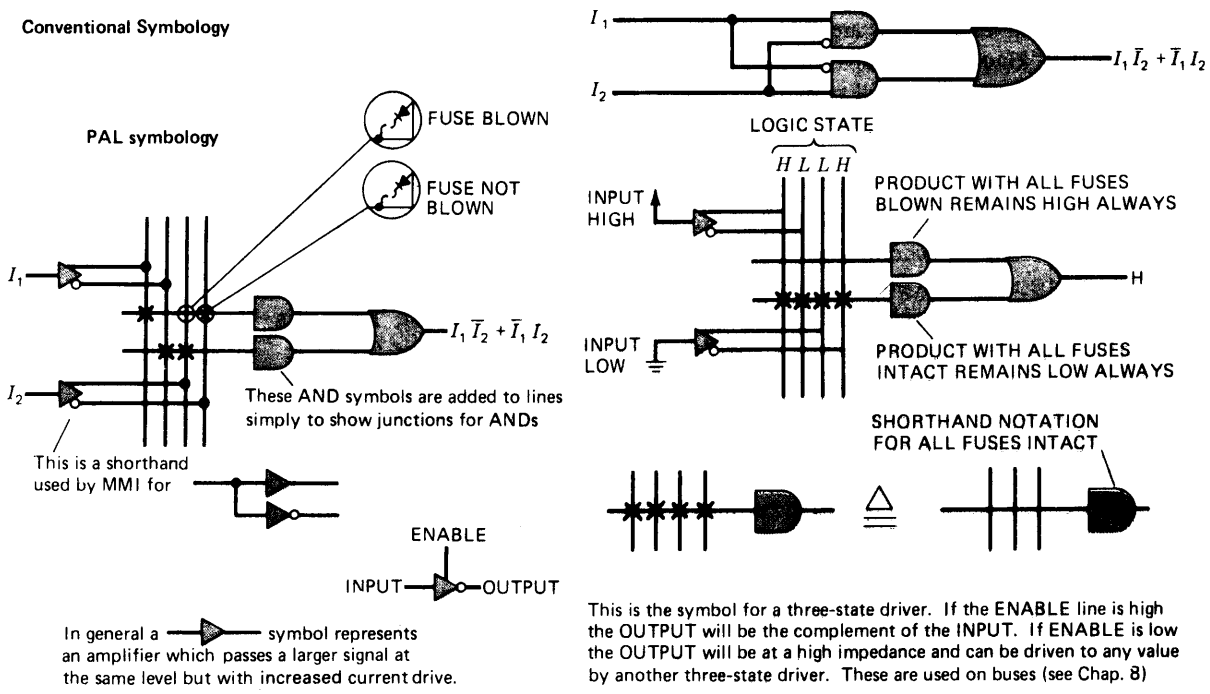
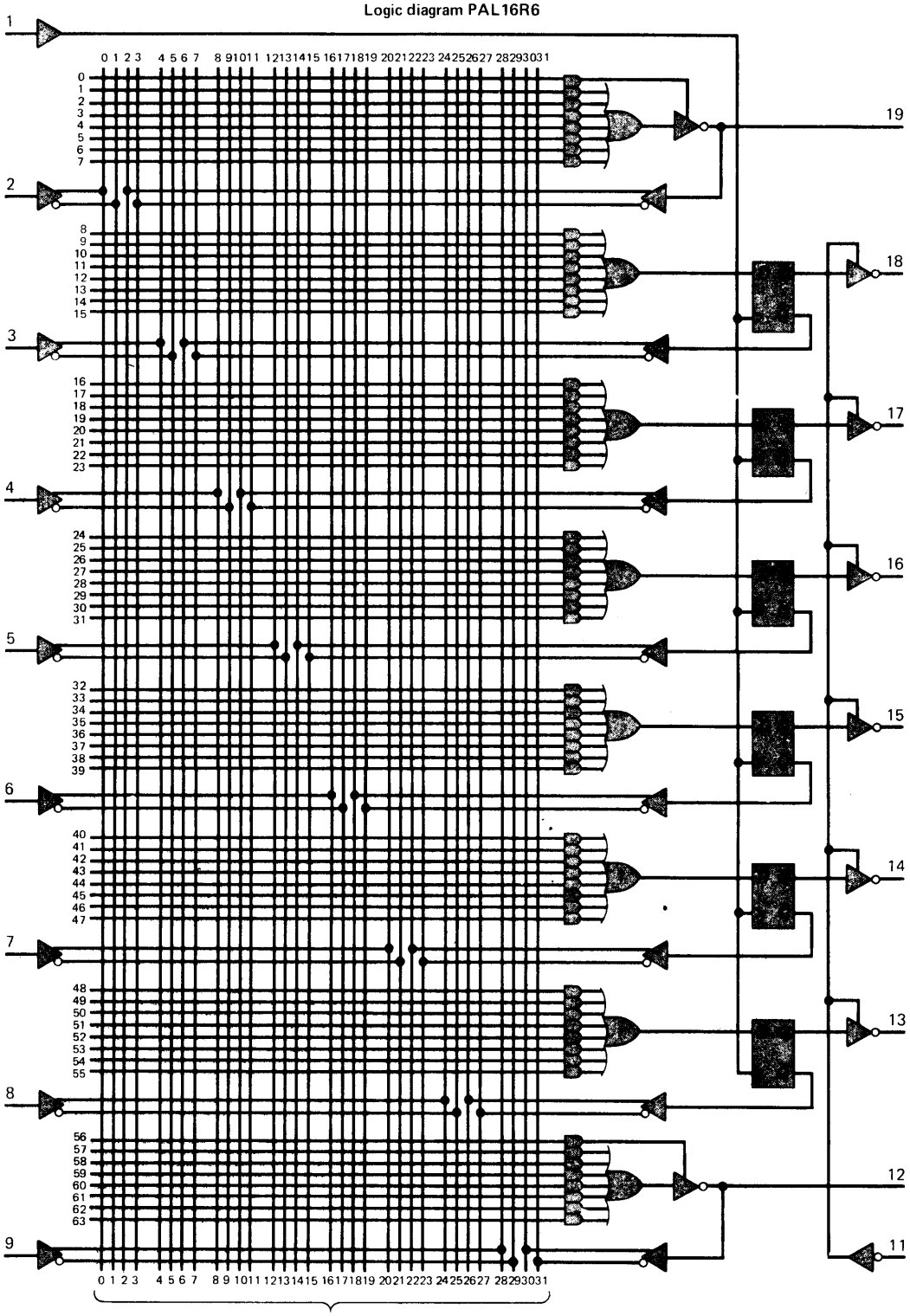


FIGURE 4.32

PAL with six flip-flops. (*Monolithic Memories.*)



Logic diagram PAL16R6



Note: ANDs can be formed at any junction in this area

QUESTIONS

4.1 Draw a set of waveforms for S and R and X and $\overline{X}$ (as in Fig. 4.2) so that the flip-flop in Fig. 4.1 will have the output signals 0011010 on the output line.

4.2 If the AND gate connected to the R input of X_1 in Fig. 4.3 fails so that its output is always 1, we would expect, after a few transfers, that X_1 would always be in what state? Why?

4.3 Draw a set of waveforms for S and R (as in Fig. 4.2) so that the flip-flop in Fig. 4.1 will have the output signals 101110001 on the X output line.

4.4 If the $\overline{X}$ output of flip-flop X is connected to an inverter, the inverter's output will always be the same as the X output of the flip-flop. True or false? Why?

4.5 Draw a set of waveforms as in Fig. 4.5(*d*) for the flip-flop in Fig. 4.5(*b*) so that the flip-flop will have the output signals 0010110 on its Y output line.

4.6 In Fig. 4.10 if flip-flop X_2 "sticks" (that is, fails) in its 0 state, will X_3 have a 1 output after (*a*) clock pulse 1, (*b*) clock pulse 2, (*c*) clock pulse 3 and for each clock pulse thereafter?

4.7 Draw an input waveform as in Fig. 4.10 so that X_3 will have the output signal 010011010 if X_1, X_2, and X_3 are started in the 0 state.

4.8 Draw a set of waveforms as in Fig. 4.5(*c*) for the flip-flop in Fig. 4.5(*a*) so that the flip-flop will have the output signals 10111001 on its X output line.

4.9 The binary counter in Fig. 4.11 uses flip-flops which act on positive transitions. Draw a block diagram of a binary counter with bubbles on the clock inputs (that is, with flip-flops which act on *negative*-going clock inputs).

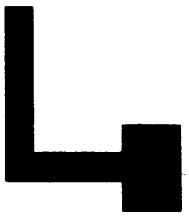
4.10 If the X_3 line (the output of X_3) is connected to the input line in Fig. 4.10, a *ring counter* is formed. If this circuit is started with $X_1 = 0$, $X_2 = 1$, and $X_3 = 1$, draw the waveform at X_1, X_2, and X_3 for six clock pulses.

4.11 Redraw Fig. 4.11 as it is, but place bubbles on each clock input to X_1, X_2, and X_3 [that is, make the same drawing, but use the flip-flop in Fig. 4.5(*b*) instead of that in Fig. 4.5(*a*)]. Redraw the waveforms in Fig. 4.11 for this circuit.

4.12 Does the counter in Question 4.11 count up or down?

4.13 Make a single change in Fig. 4.11 by connecting the output of flip-flop X_2 to the clock input of X_3 instead of the $\overline{X}_2$ output of X_2. Now redraw the waveforms in Fig. 4.11 for this changed configuration.

4.14 After answering Question 4.13, using the flip-flop in Fig. 4.5(*a*), design a counter that counts as follows:



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

4.15 After answering Question 4.13, using only the flip-flop in Fig. 4.5(*a*), design a counter that counts as follows:

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

This counter counts *down*. Figure 4.11 counts *up*.

4.16 For Fig. 4.14(*b*) draw UP ENABLE, DOWN ENABLE, and clock waveforms so that the counter starts at 000 and counts as follows:

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

4.17 In Fig. 4.14(*a*), when the ENABLE line is a 1, the counter counts at the occurrence of a negative-going clock edge. Draw the output waveforms for FF0, FF1, and FF2 for these waveforms. Start the flip-flops at 0.

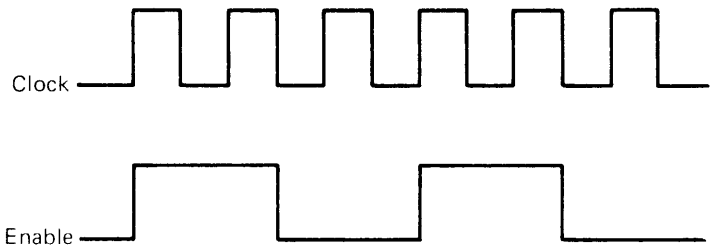


FIGURE Q4.17

4.18 Draw a clock waveform and waveforms for the output of the AND gate connected to the S input of X_1, the R input of X_2, and the 1 output of X_1, X_2, and X_3 for Fig. 4.12.

4.19 Suppose that the AND gate in Fig. 4.15(a) fails, so that its output is always a 0. Write the sequence or show the waveform through which this counter will go in response to clock signals.

4.20 Suppose that the AND gate's output in Fig. 4.15(a) is connected to the K input of X_4 instead of the J input. How will the computer count?

4.21 For Fig. 4.15(b), the carry-out from block 1 to block 2 goes from 1 to 0 every _____ clock pulses. The carry-out from block 3 to block 4 goes from 1 to 0 every _____ clock pulses.

4.22 (a) If we replace block 2 in Fig. 4.15(b) with the four-stage ripple counter in Fig. 4.14(a), using the 1 output of FF3 in that counter as the carry-out line, the carry-out line from block 2 will go from 1 to 0 after how many clock pulses?

(b) For the configuration in (a), after how many clock pulses will the carry-out from block 2 go from 0 to 1?

4.23 Using the circuits in Fig. 4.16, design a gated-clocked binary counter.

4.24 Design a BCD counter using the flip-flops in Fig. 4.16.

4.25 Using the circuits in Fig. 4.16, design a gating network with inputs A, B, and C which will have output 1 when $\bar{A}\bar{B}C$ or $A\bar{B}\bar{C}$ are 1s.

4.26 Using the circuits in Fig. 4.16, design a BCD counter.

4.27 Using the circuits in Fig. 4.16, design a gate network with inputs A, B, and C and output $A\bar{B} + \bar{A}C$.

4.28 Does a four-input NAND gate, as in Fig. 4.16, with the third input held at 1, act as a three-input gate would if only two inputs were used? Explain your answer.

4.29 Design the counter in Fig. 4.12, using the blocks in Fig. 4.16.

4.30 Give the states of the flip-flops in the following circuit after each of the first five clock signals (pulses) are applied. The circuit is started in the state $A_1 = 0$, $A_2 = 0$, $A_3 = 1$.

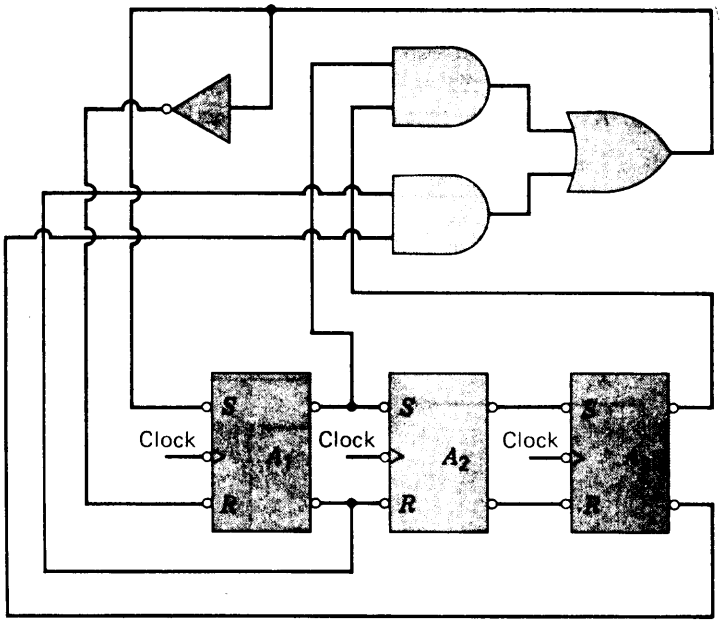


FIGURE Q4.30

4.31 Redesign the following circuit, using only *RS* flip-flops and NOR gates:

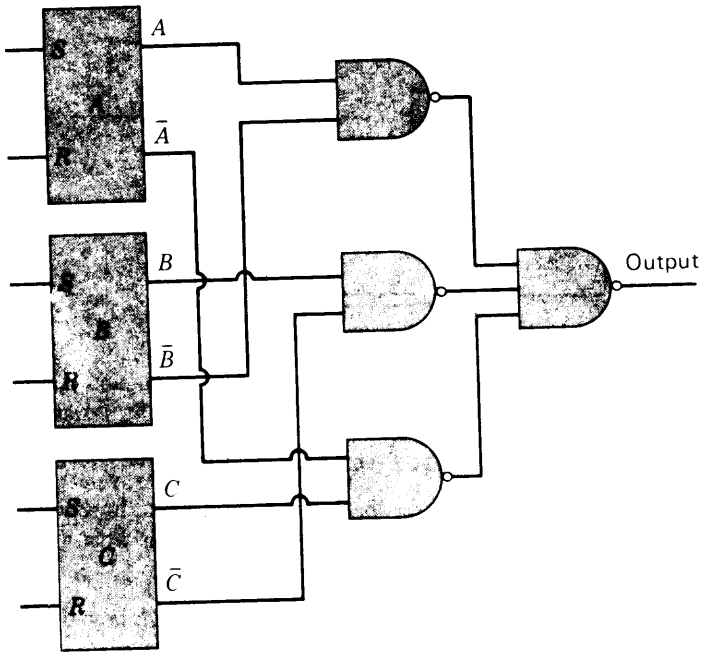


FIGURE Q4.31

4.32 The gate block in the following circuit is an "equal to" combinational network realizing the boolean function $\bar{X}_3\bar{X}_2 + X_3X_2$. If this set of three flip-flops is started in $X_1 = 1$, $X_2 = 0$, and $X_3 = 0$, what will the sequence of internal states be? As a start, the first three states can be listed as follows:

X_1	X_2	X_3	
1	0	0	
1	1	0	after first clock pulse
0	1	1	after second clock pulse
·	·	·	(you are to continue this list)

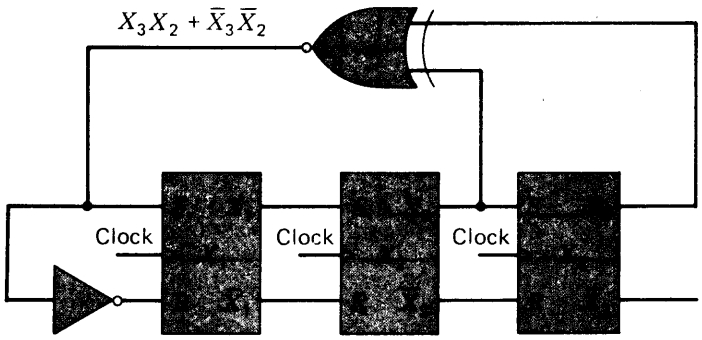


FIGURE Q4.32

4.33 The following circuit is started in $C_1 = 1$ and $C_2 = 0$. The circuit divides the number of positive-going input edges (positive pulses) by what number? (That is, every _____ input pulses the output will return to 0.) Justify your answer.

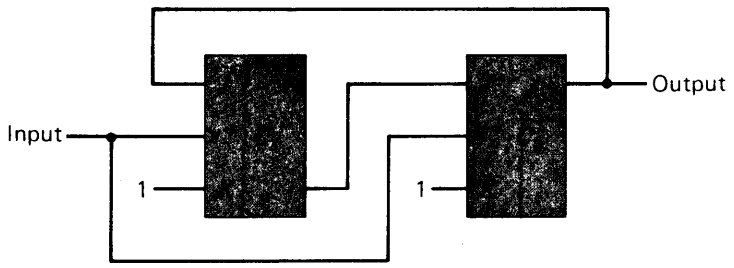


FIGURE Q4.33

4.34 In Fig. 4.9, write 0s and 1s for all gate inputs and outputs when the clock input is 1, R is a 1, S is a 1, and the flip-flop is in the 1 state.

4.35 In Fig. 4.9, why are E and F enabled at a lower level than A and B?

4.36 In Fig. 4.9, if the feedback connection from the C NAND gate output to the input of the D NAND gate is broken (open), in what state will we probably find the flip-flop?

4.37 Using your result from Question 4.34, explain how the flip-flop in Fig. 4.9 operates when R and S are 0s and a negative edge appears.

4.38 The following sequence is to be realized by a counter consisting of three *RS* flip-flops. Use AND and OR gates in your design.

	A_1	A_2	A_3	
	0	0	0	starting state
	0	1	0	
Sequence repeats after this segment	0	1	1	
	0	0	1	
	1	0	0	
	1	1	0	
	0	0	0	
	.	.	.	

4.39 Design a counter, using only *JK* flip-flops, AND gates, and OR gates which counts in the following sequence:

0	0	0	this repeats
0	1	0	
0	1	1	
1	0	0	
0	0	0	
0	1	0	
0	1	1	
1	0	0	
0	0	0	

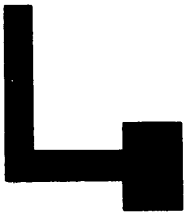
4.40 Design a counter, using three *JK* flip-flops X_1, X_2, and X_3 and whatever gates you would like, which counts as follows:

X_1	X_2	X_3	
0	0	0	starting state
0	1	1	after first clock pulse
0	1	0	after second clock pulse
1	1	1	after third clock pulse
1	0	1	after fourth clock pulse
0	0	0	after fifth clock pulse
0	1	1	
0	1	0	

4.41 The following sequence is to be realized by a counter consisting of three *JK* flip-flops. Use AND and OR gates in your design.

	A_1	A_2	A_3	
	0	0	0	starting state
	0	1	1	
Sequence repeats after this segment	0	1	0	
	0	0	1	
	1	0	1	
	1	1	0	
	0	0	0	

4.42 If the *S* and *R* inputs to Fig. 4.7 are both made 0s and the *S* is made a 1 followed by *R*, what will be the resulting state of the flip-flop?



4.43 The NAND gate flip-flop in Fig. 4.7 will have what outputs on the 0 and 1 lines if both SET and RESET are made 0s?

4.44 Design a counter, using three *JK* flip-flops X_1, X_2, and X_3 and whatever gates you would like, which counts as follows:

X_1	X_2	X_3	
0	0	1	starting state
0	1	1	after first clock pulse
0	1	0	after second clock pulse
1	1	1	after third clock pulse
1	0	1	after fourth clock pulse
0	0	1	after fifth clock pulse
0	1	1	
0	1	0	

4.45 The rules for designing counters using *JK* and *RS* flip-flops have been given. Derive the rules for designing a counter using *D* flip-flops.

4.46 Design a counter using *D* flip-flops which counts in the same manner as the example given for *JK* and *RS* flip-flops.

4.47 Design a binary sequence detector which recognizes four consecutive 1s. Display the state diagram, state table, and final design.

4.48 Using the PAL in Fig. 4.30, design a state machine which realizes this state diagram:

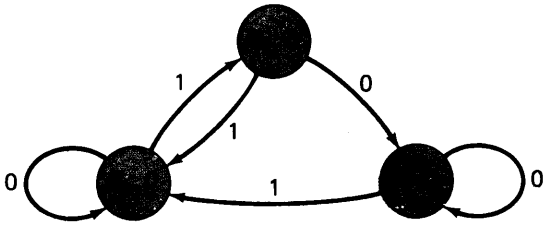


FIGURE Q4.48

4.49 Design a state machine which realizes this state diagram:

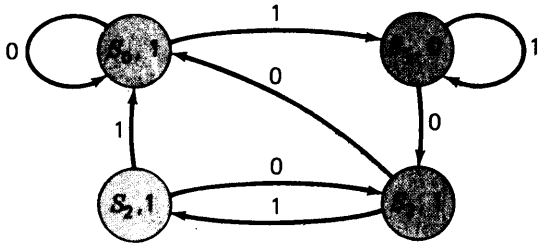
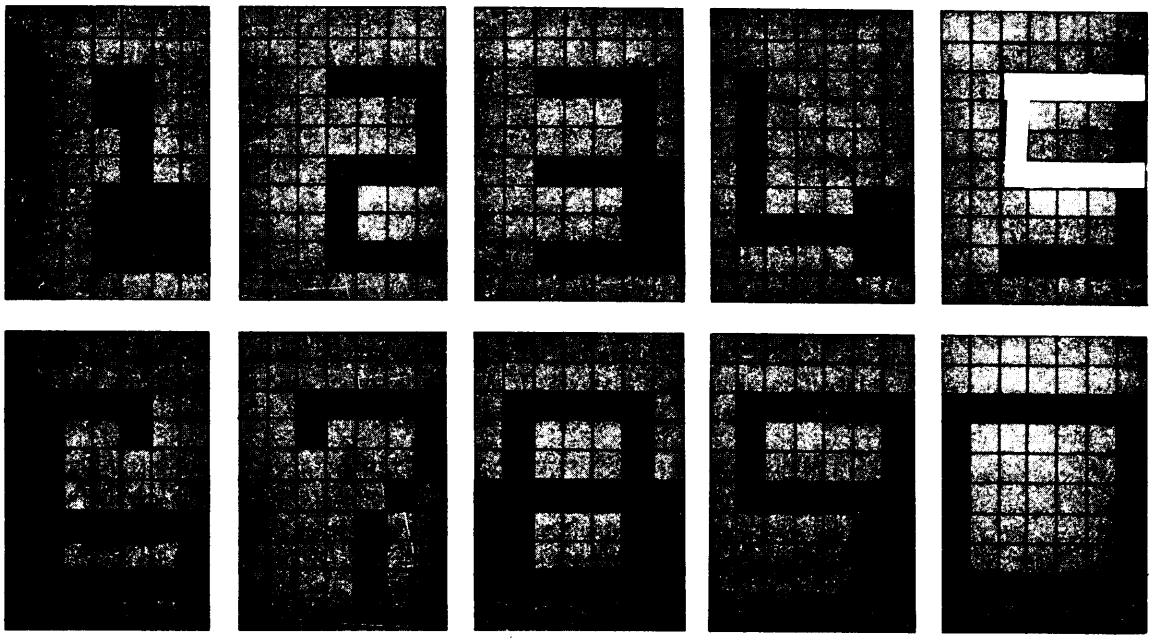


FIGURE Q4.49

4.50 Design a two-input magnitude comparator, but use the Mealy configuration instead of the Moore configuration shown in Figure 4.28.



THE ARITHMETIC-LOGIC UNIT

The arithmetic-logic unit (ALU) is the section of the computer that performs arithmetic and logical operations on the data. This section of the machine can be relatively small, consisting of as little as a part of a large-scale integration (LSI) chip, or, for large "number crunchers" (scientific-oriented computers), it can consist of a considerable array of high-speed logic components. Despite the variations in size and complexity, the small machines perform their arithmetic and logical operations using the same principles as the large machines. What changes is the speed of the logic gates and flip-flops used; also, special techniques are used for speeding up operations and performing several operations in parallel.

Although many functions can be performed by the ALUs of present-day machines, the basic arithmetic operations—addition, subtraction, multiplication, and division–continue to be "bread-and-butter" operations. Even the literature gives evidence of the fundamental nature of these operations, for when a new machine is described, the times required for addition and multiplication are always included as significant features. Accordingly, this chapter first describes the means by which a computer adds, subtracts, multiplies, and divides. Other basic operations, such as shifting, logical multiplication, and logical addition, are then described.

Remember that the control unit directs the operation of the ALU. What the ALU does is to add, subtract, shift, etc., when it is provided with the correct sequence of input signals. It is up to the control element to provide these signals, and it is the function of the memory units to provide the arithmetic element with the information that is to be used. These sections of the computer are discussed in Chaps. 6 and 9.

5

THE ARITHMETIC-LOGIC UNIT

OBJECTIVES

1 Most arithmetic operations are based on the use of a full-adder module which can add pairs of binary bits and initiate and propagate any carries that arise. The full-adder is explained, and several examples are given, including several popular IC adders.

2 The addition and subtraction of binary numbers can be effected by using adders and gates correctly connected. The layouts for 2s complement and 1s complement addition-subtraction units are shown and the general principles explained.

3 Binary-coded-decimal (BCD) adders and subtracters use a different layout than straight binary, and this subject is explained. Serial-parallel addition and subtraction using only a single BCD adder is often used in computers, and this subject is explained and examples are shown.

4 Multiplication and division are generally performed using three flip-flop registers and a sequence of addition, subtraction, and shift operations. The procedures for the operations are explained and examples given. High-speed multiplication using gate networks is covered also.

5 Arithmetic-logic units which can add, subtract, and perform logical operations form the backbone for the arithmetic and control operations in computers. The organization of these units is explained.

6 To perform scientific calculations, the floating-point number system is often used, particularly when high-level languages are written by the programmer. The structure of floating-point number systems is shown, and several example systems are explained.

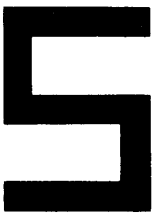
CONSTRUCTION OF THE ALU

5.1 The information handled in a computer is generally divided into "words," each consisting of a fixed number of bits.[1] For instance, the words handled by a given binary machine may be 32 bits in length. In this case, the ALU would have to be capable of adding, subtracting, etc., words 32 bits in length. The operands used are supplied to the ALU, and the control element directs the operations that are performed. If addition is to be performed, the addend and augend will be supplied to the ALU which must add the numbers and then, at least temporarily, store the results (sum).

To introduce several concepts, let us consider the construction of a typical computer ALU. The storage devices will consist of a set of flip-flop *registers*, each of which consists of one or more flip-flops. For convenience, the various registers of the ALU are generally given names such as X register, B register, MQ register, etc., and the flip-flops are then given the same names, so that the X register would contain flip-flops X_1, X_2, X_3, etc.

Many computers (especially microprocessors) have a register called an *accumulator* which is the principal register for arithmetic and logical operations. This

[1]Some computers also provide the ability to handle variable-length operands.



register stores the result of each arithmetic or logical operation, and gating circuitry is attached to the register so that the necessary operations can be performed on its contents and any other registers involved.

An accumulator is a basic storage register of the arithmetic element. If the machine is instructed to *load* the accumulator, the control element will first clear the accumulator of whatever may have been stored in it and then put the operand selected in storage into the accumulator register. If the computer is instructed to *add*, the number stored in the accumulator will represent the augend. Then the addend will be located in memory, and the computer's circuitry will add this number (the addend) to the number previously stored in the accumulator (the augend) and store the sum in the accumulator. Notice that the original augend will no longer be stored in the accumulator after the addition. Furthermore, the sum may either remain in the accumulator or be transferred to memory, depending on the type of computer. In this chapter we deal only with the processes of adding, subtracting, etc., and not the process of locating the number to be added in memory or the transferring of numbers to memory. These operations are covered in the following chapters.

There is now a tendency for computers to have more than one accumulator. When a computer has more than one, they are often named, for example, accumulator *A* and accumulator *B* (as in the 6800 microprocessor) or ACC1, ACC2, etc. (as in the Data General Corporation computers). When the number of registers provided to hold operands becomes larger than four, however, the registers are often called *general registers*, and individual registers are given names and numbers such as general register 4, general register 8, etc.

INTEGER REPRESENTATION

5.2 The numbers used in digital machines must be represented by using such storage devices as flip-flops. The most direct number representation system for binary-valued storage devices is an integer representation system. Figure 5.1(*a*) shows a register of four flip-flops, X_3, X_2, X_1, and X_0, used to store numbers.

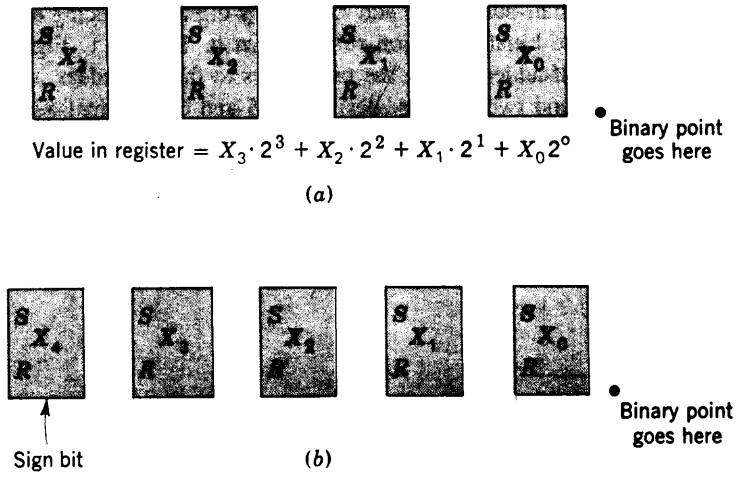
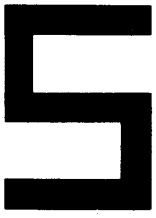


FIGURE 5.1

Representation systems. (*a*) Integer representation. (*b*) Sign-plus-magnitude system.



Simply writing the values or states of the flip-flops gives the number in integer form. Thus $X_3 = 1, X_2 = 1, X_1 = 0, X_0 = 0$ gives 1100, or decimal 12, whereas $X_3 = 0, X_2 = 1, X_1 = 0, X_0 = 1$ gives 0101, or decimal 5.

It is generally necessary to represent both positive and negative numbers; so an additional bit is required, called the *sign bit*. This is generally placed to the left of the magnitude bits. In Fig. 5.1(*b*) X_4 is the sign bit, and so X_3, X_2, X_1, and X_0 will give the magnitude. A 0 in X_4 means that the number is positive, and a 1 in X_4 means that the number is negative (this is the usual convention).[2] So $X_4 = 0$, $X_3 = 1, X_2 = 1, X_1 = 0$, and $X_0 = 1$ gives positive 1101, or +13 in decimal; and $X_4 = 1, X_3 = 1, X_2 = 1, X_1 = 0$, and $X_0 = 1$ gives negative 1101, or −13 in decimal.

This system is called the *signed-integer binary system*, or *signed-magnitude binary integer system*. If a register contains eight flip-flops, a signed binary number in the system would have 7 magnitude, or integer, bits and a single sign bit. So 00001111 would be +15, and 10001111 would be −15, since the leading 0 and 1 indicate the plus and minus signs only.

The magnitude of numbers which can be stored in the two representation systems in Fig. 5.1 are as follows:

1 For binary integer representation, an n-flip-flop register can store from (decimal) 0 to $2^n - 1$. A 6-bit register can therefore store from 000000 to 111111, where 111111 is 63, which is $2^6 - 1$, or $64 - 1$.

2 The signed binary integer representation system has a range of $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$ for a binary register. For instance, a seven-flip-flop register can store from −111111 to +111111, which is −63 to +63 $[(2^6 - 1)$ to $+ (2^6 - 1)]$.

In the following sections we learn how to perform various arithmetic and logical operations on registers.

BINARY HALF-ADDER

5.3 A basic module used in binary arithmetic elements is the *half-adder*. The function of the half-adder is to add two binary digits, producing a sum and a carry according to the binary addition rules shown in Table 5.1. Figure 5.2 shows a design for a half-adder. There are two inputs to the half-adder, designated X and Y in Fig. 5.2, and two outputs, designated S and C. The half-adder performs the binary addition operation for two binary inputs shown in Table 5.1. This is arithmetic addition, not logical or boolean algebra addition.

As shown in Fig. 5.2, there are two inputs to the half-adder and two outputs. If either of the inputs is a 1 but not both, then the output on the S line will be a 1. If both inputs are 1s, the output on the C (for carry) line will be a 1. For all

[2]Some companies number registers with the sign bit A_0 the most significant bit A, and so on to the least significant bit A_n is a register with $n + 1$ bits. IBM does this for some of its computers, for example.

TABLE 5.1	
INPUT	SUM BITS
0 + 0	0
0 + 1	1
1 + 0	1
1 + 1	0 with a carry of 1

other states, there will be a 0 output on the CARRY line. These relationships may be written in boolean form as follows:

$$S = X\bar{Y} + \bar{X}Y$$
$$C = XY$$

A *quarter-adder* consists of the two inputs to the half-adder and the S output only. The logical expression for this circuit is, therefore, $S = X\bar{Y} + \bar{X}Y$. This is also the *exclusive OR* relationship for boolean algebra (refer to Chap. 3).

FULL-ADDER

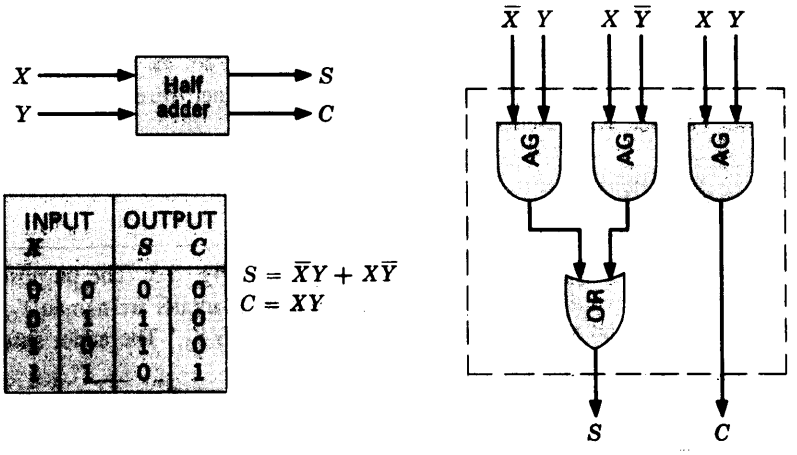
5.4 When more than two binary digits are to be added, several half-adders will not be adequate, for the half-adder has no input to handle carries from other digits.

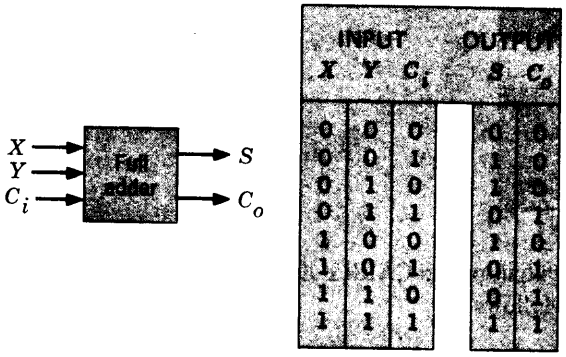
Consider the addition of the following two binary numbers:

1011		1011	
+ 1110		+ 1110	
11001	= sum	0101	= partial sum
		1 1	= carry bits
		11001	= complete sum

FIGURE 5.2

Half-adder.





$$S = \bar{X}\bar{Y}C_i + \bar{X}Y\bar{C}_i + X\bar{Y}\bar{C}_i + XYC_i$$

$$C_o = \bar{X}YC_i + X\bar{Y}C_i + XY\bar{C}_i + XYC_i$$

or

$$C_o = XC_i + XY + YC_i$$

FIGURE 5.3

Full-adder.

As shown, the carries generated in each column must be considered during the addition process. Therefore adder circuitry capable of adding the contents of two registers must include provision for handling carries as well as addend and augend bits. So there must be three inputs to each stage of a multidigit adder—except the stage for the least significant bits—one for each input from the numbers being added and one for any carry that might have been generated or propagated by the previous stage.

The block diagram symbol for a *full binary adder*, which will handle these carries, is illustrated in Fig. 5.3, as is the complete table of input-output relationships for the full-adder. There are three inputs to the full-adder: the X and Y inputs from the respective digits of the registers to be added and the C_i input, which is for any carry generated by the previous stage. The two outputs are S, which is the output value for that stage of the addition, and C_o, which produces the carry to be added into the next stage.[3] The boolean expressions for the input-output relationships for each of the two outputs are also presented in Fig. 5.3, as is the expression for the C_o output in simplified form.

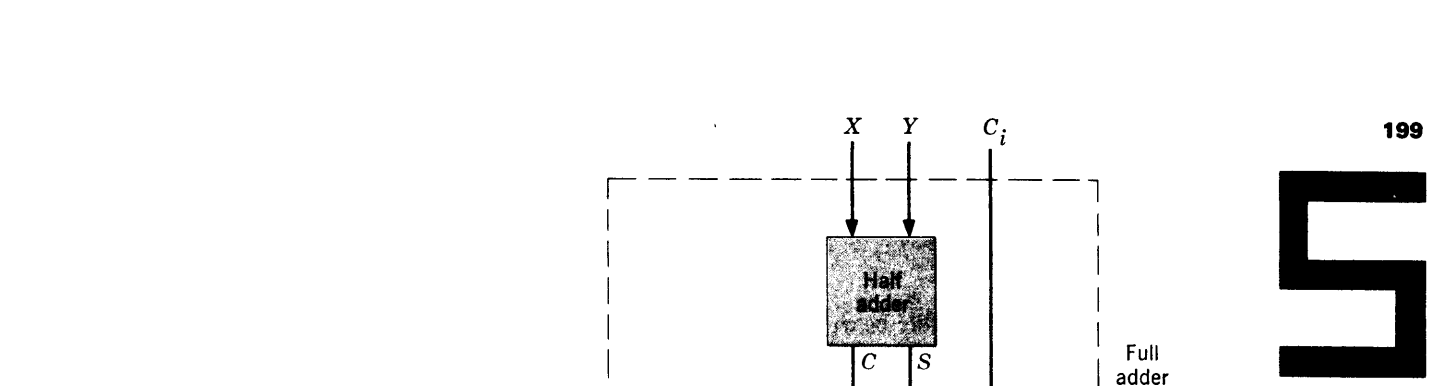
A full-adder may be constructed of two half-adders, as illustrated in Fig. 5.4. Constructing a full-adder from two half-adders may not be the most economical technique, however; generally full-adders are designed directly from the input-output relations illustrated in Fig. 5.3.

A PARALLEL BINARY ADDER

5.5 A 4-bit parallel binary adder is illustrated in Fig. 5.5. The purpose of this adder is to add two 4-bit binary integers. The addend inputs are named X_0 through X_3, and the augend bits are represented by Y_0 through Y_3.[4] The adder shown does

[3] C_i is for *carry-in* and C_o for *carry-out*.

[4] These inputs would normally be from flip-flop registers X and Y, and the adder would add the number in X to the number in Y, giving the sum, or S_0 through S_3.



A PARALLEL BINARY ADDER

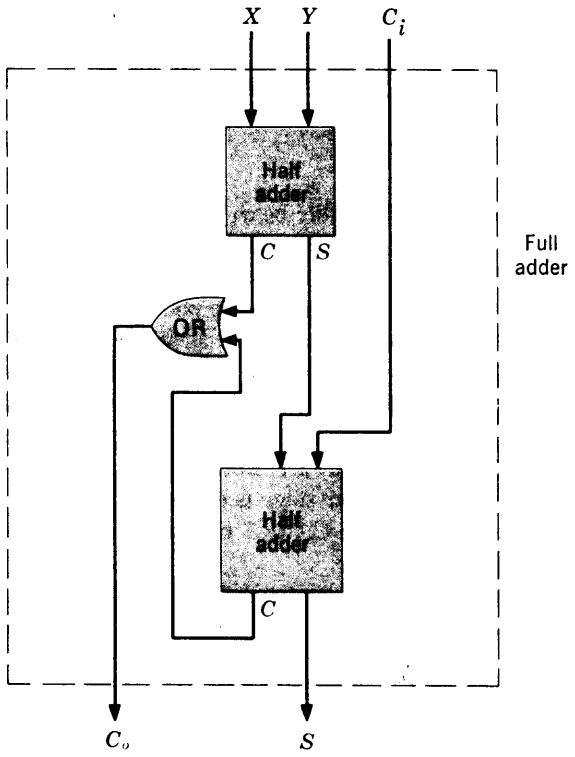


FIGURE 5.4

Half-adder and full-adder relations.

not possess the ability to handle sign bits for the binary words to be added, but only adds the magnitudes of the numbers stored. The additional circuitry needed to handle sign bits is dependent on whether negative numbers are represented in true magnitude or in the 1s or 2s complement systems, and this problem is described later.

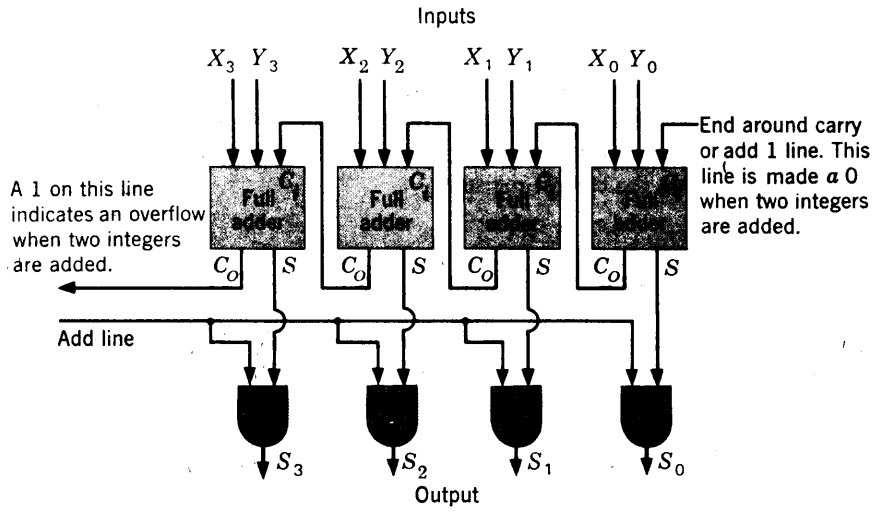
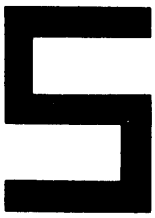


FIGURE 5.5

Parallel adder.



Consider the addition of the following two 4-bit binary numbers:

$$\begin{array}{rl} 0111 & \text{where } X_3 = 0,\ X_2 = 1,\ X_1 = 1, \text{ and } X_0 = 1 \\ \underline{0011} & \text{where } Y_3 = 0,\ Y_2 = 0,\ Y_1 = 1, \text{ and } Y_0 = 1 \\ \text{Sum} = 1010 & \end{array}$$

The sum should therefore be $S_3 = 1$, $S_2 = 0$, $S_1 = 1$, and $S_0 = 0$.

The operation of the adder may be checked as follows. Since X_0 and Y_0 are the least significant digits, they cannot receive a carry from a previous stage. In the problem above, X_0 and Y_0 are both 1s, their sum is therefore 0, and a carry is generated and added into the full-adder for bits X_1 and Y_1. Bits X_1 and Y_1 are also both 1s, as is the carry input to this stage. Therefore, the sum output line S_1 carries a 1, and the CARRY line to the next stage also carries a 1. Since X_2 is a 1, Y_2 is a 0, and the carry input is 1, the sum output line S_2 will carry a 0, and the carry to the next stage will be a 1. Both inputs X_3 and Y_3 are equal to 0, and the CARRY input line to this adder stage is equal to 1. Therefore, the sum output line S_3 will represent a 1, and the CARRY output line, designated as "overflow" in Fig. 5.5, will have a 0 output.

The same basic configuration illustrated in Fig. 5.5 may be extended to any number of bits. A 7-bit adder may be constructed by using 7 full-adders, and a 20-bit adder by using 20 full-adders.

Note that the OVERFLOW line could be used to enable the 4-bit adder in Fig. 5.5 to have a 5-bit output. This is not generally done, however, because the addend and augend both come from storage, and so their length is the length of the basic computer word, and a longer word cannot be readily stored by the machine. It was explained earlier that a machine with a word length of n bits (consisting of sign bit and $n - 1$ bits to designate the magnitude) could express binary numbers from $-2^{n-1} + 1$ to $2^{n-1} - 1$. A number within these limits is called *representable*. Since the simple 4-bit adder in Fig. 5.5 has no sign bit, it can represent only binary integers from 0 to 15. If 1100 and 1100 are added in the adder illustrated in Fig. 5.5, there will be a 1 output on the OVERFLOW line because the sum of these two numbers is 11000. This number is 24 decimal and cannot be represented in this system. Such a number is referred to as *nonrepresentable* for this particular very small register. When two integers are added such that their sum is nonrepresentable (that is, contains too many bits), then we say the sum *overflows*, or an *overflow* occurs and a 1 on the CARRY line for the full-adder connected to the most significant digits indicates this.

The AND gates connected to the S output lines from the four adders are used to gate the sum into the correct register.

POSITIVE AND NEGATIVE NUMBERS

5.6 When numbers are written in the decimal system, the common practice is to write the number as a magnitude preceded by a plus or minus sign, which indicates whether the number is positive or negative. Hence $+125$ is positive and -125 is negative. The same practice is generally used with binary numbers: $+111$ is positive 7, and -110 is negative 6. To handle both positive and negative num-