

e-16 (hex).  
quad numbers.  
can represent  
can represent  
ion of  $n$ )?  
42?  
number 123.3?  
ing point  
and produces  
r of unique  
bits of  
d 3 bits of  
floating point  
mber.)

101001 000110 0011  
110111 101010 0011

Int. Addressing, Pent. 4  
C Pentium 4.1.1.1  
2



LDR R0, R5, #5  
LDR R1, R5, #4  
ADD R2, R0, R1  
STR R2, R6, #0  
REV



## Digital Logic Structures

In Chapter 1, we stated that computers were built from very large numbers of very simple structures. For example, Intel's Pentium IV microprocessor, first offered for sale in 2000, was made up of more than 42 million MOS transistors. The IBM Power PC 750 FX, released in 2002, consists of more than 38 million MOS transistors. In this chapter, we will explain how the MOS transistor works (as a logic element), show how these transistors are connected to form logic gates, and then show how logic gates are interconnected to form larger units that are needed to construct a computer. In Chapter 4, we will connect those larger units into a computer.

But first, the transistor.

### 3.1 The Transistor

Most computers today, or rather most microprocessors (which form the core of the computer) are constructed out of MOS transistors. MOS stands for *metal-oxide semiconductor*. The electrical properties of metal-oxide semiconductors are well beyond the scope of what we want to understand in this course. They are below our lowest level of abstraction, which means that if somehow transistors start misbehaving, we are at their mercy. It is unlikely that we will have any problems from the transistors.

However, it is useful to know that there are two types of MOS transistors: p-type and n-type. They both operate "logically," very similar to the way wall switches work.

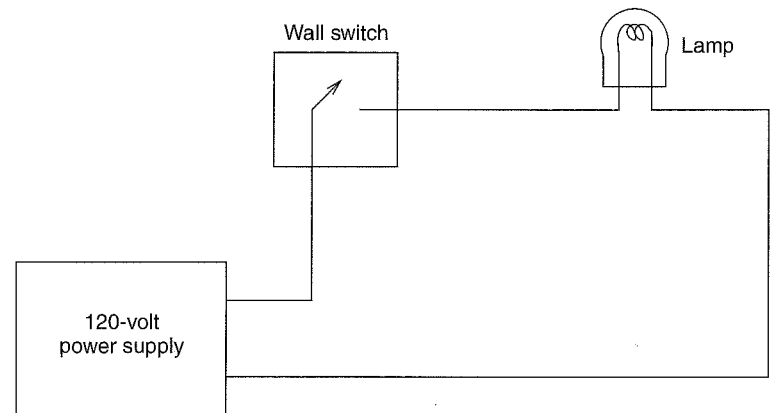


Figure 3.1 A simple electric circuit showing the use of a wall switch

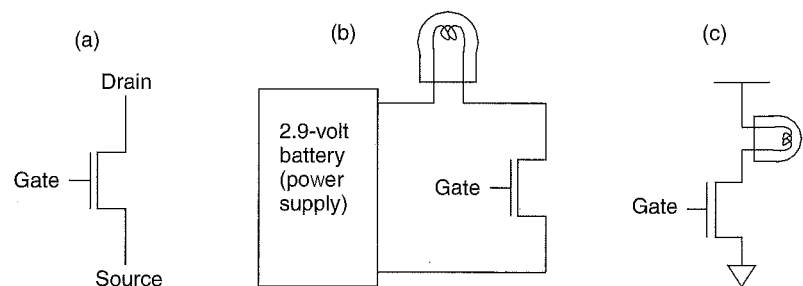


Figure 3.2 The n-type MOS transistor

Figure 3.1 shows the most basic of electrical circuits: a power supply (in this case, the 120 volts that come into your house), a wall switch, and a lamp (plugged into an outlet in the wall). In order for the lamp to glow, electrons must flow; in order for electrons to flow, there must be a closed circuit from the power supply to the lamp and back to the power supply. The lamp can be turned on and off by simply manipulating the wall switch to make or break the closed circuit.

Instead of the wall switch, we could use an n-type or a p-type MOS transistor to make or break the closed circuit. Figure 3.2 shows a schematic rendering of an n-type transistor (a) by itself, and (b) in a circuit. Note (Figure 3.2a) that the transistor has three terminals. They are called the *gate*, the *source*, and the *drain*. The reasons for the names source and drain are not of interest to us in this course. What is of interest is the fact that if the gate of the n-type transistor is supplied with 2.9 volts, the connection from source to drain acts like a piece of wire. We say (in the language of electricity) that we have a *closed circuit* between the source and drain. If the gate of the n-type transistor is supplied with 0 volts, the connection between the source and drain is broken. We say that between the source and drain we have an *open circuit*.

Figure 3.2b shows the n-type transistor in a circuit with a battery and a bulb. When the gate is supplied with 2.9 volts, the transistor acts like a piece of wire,

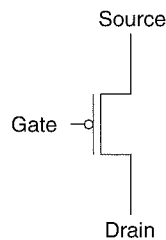


Figure 3.3 A p-type MOS transistor

completing the circuit and causing the bulb to glow. When the gate is supplied with 0 volts, the transistor acts like an open circuit, breaking the circuit, and causing the bulb not to glow.

Figure 3.2c is a shorthand notation for describing the circuit of Figure 3.2b. Rather than always showing the power supply and the complete circuit, electrical engineers usually show only the terminals of the power supply. The fact that the power supply itself provides the completion of the completed circuit is well understood, and so is not usually shown.

The p-type transistor works in exactly the opposite fashion from the n-type transistor. Figure 3.3 shows the schematic representation of a p-type transistor. When the gate is supplied with 0 volts, the p-type transistor acts (more or less) like a piece of wire, closing the circuit. When the gate is supplied with 2.9 volts, the p-type transistor acts like an open circuit. Because the p-type and n-type transistors act in this complementary way, we refer to circuits that contain both p-type and n-type transistors as CMOS circuits, for *complementary metal-oxide semiconductor*.

## 3.2 Logic Gates

One step up from the transistor is the logic gate. That is, we construct basic logic structures out of individual MOS transistors. In Chapter 2, we studied the behavior of the AND, the OR, and the NOT functions. In this chapter we construct transistor circuits that implement each of these functions. The corresponding circuits are called AND, OR, and NOT gates.

### 3.2.1 The NOT Gate (Inverter)

Figure 3.4 shows the simplest logic structure that exists in a computer. It is constructed from two MOS transistors, one p-type and one n-type. Figure 3.4a is the schematic representation of that circuit. Figure 3.4b shows the behavior of the circuit if the input is supplied with 0 volts. Note that the p-type transistor conducts and the n-type transistor does not conduct. The output is, therefore, connected to 2.9 volts. On the other hand, if the input is supplied with 2.9 volts, the p-type transistor does not conduct, but the n-type transistor does conduct. The output in this case is connected to ground (i.e., 0 volts). The complete behavior

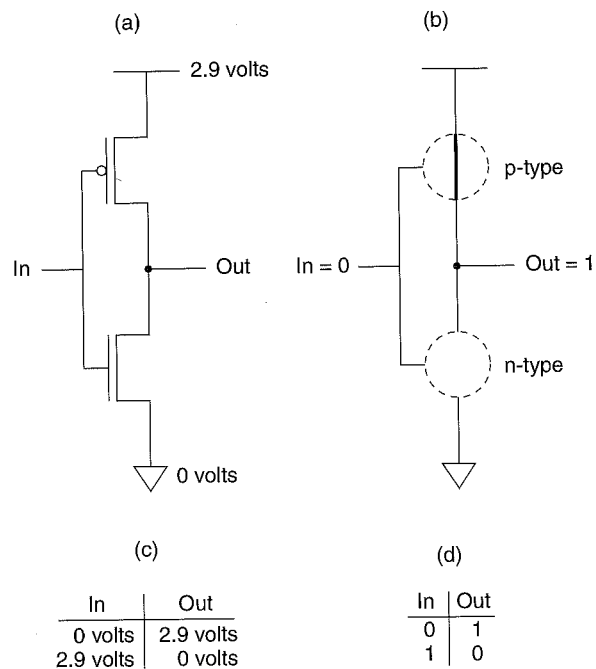


Figure 3.4 A CMOS inverter

of the circuit can be described by means of a table, as shown in Figure 3.4c. If we replace 0 volts by the symbol 0 and 2.9 volts by the symbol 1, we have the truth table (Figure 3.4d) for the complement or NOT function, which we studied in Chapter 2.

In other words, we have just shown how to construct an electronic circuit that implements the NOT logic function discussed in Chapter 2. We call this circuit a *NOT gate*, or an *inverter*.

### 3.2.2 OR and NOR Gates

Figure 3.5 illustrates a NOR gate. Figure 3.5a is a schematic of a circuit that implements a NOR gate. It contains two p-type and two n-type transistors.

Figure 3.5b shows the behavior of the circuit if *A* is supplied with 0 volts and *B* is supplied with 2.9 volts. In this case, the lower of the two p-type transistors produces an open circuit, and the output *C* is disconnected from the 2.9-volt power supply. However, the leftmost n-type transistor acts like a piece of wire, connecting the output *C* to 0 volts.

Note that if both *A* and *B* are supplied with 0 volts, the two p-type transistors conduct, and the output *C* is connected to 2.9 volts. Note further that there is no ambiguity here, since both n-type transistors act as open circuits, and so *C* is disconnected from ground.

If either *A* or *B* is supplied with 2.9 volts, the corresponding p-type transistor results in an open circuit. That is sufficient to break the connection from *C* to

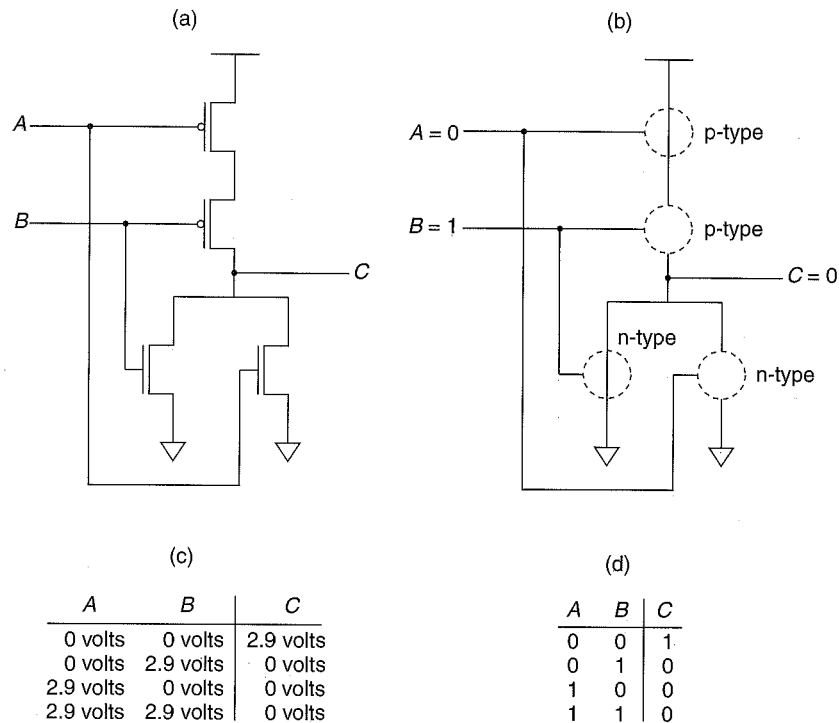


Figure 3.5 The NOR gate

the 2.9-volt source. However, 2.9 volts supplied to the gate of one of the n-type transistors is sufficient to cause that transistor to conduct, resulting in  $C$  being connected to ground (i.e., 0 volts).

Figure 3.5c summarizes the complete behavior of the circuit of Figure 3.5a. It shows the behavior of the circuit for each of the four pairs of voltages that can be supplied to  $A$  and  $B$ . That is,

$$\begin{aligned}
 A &= 0 \text{ volts}, & B &= 0 \text{ volts} \\
 A &= 0 \text{ volts}, & B &= 2.9 \text{ volts} \\
 A &= 2.9 \text{ volts}, & B &= 0 \text{ volts} \\
 A &= 2.9 \text{ volts}, & B &= 2.9 \text{ volts}
 \end{aligned}$$

If we replace the voltages with their logical equivalents, we have the truth table of Figure 3.5d. Note that the output  $C$  is exactly the opposite of the logical OR function that we studied in Chapter 2. In fact, it is the NOT-OR function, more typically abbreviated as NOR. We refer to the circuit that implements the NOR function as a NOR gate.

If we augment the circuit of Figure 3.5a by adding an inverter at the output, as shown in Figure 3.6a, we have at the output  $D$  the logical function OR. Figure 3.6a is the circuit for an OR gate. Figure 3.6b describes the behavior of this circuit if the input variable  $A$  is set to 0 and the input variable  $B$  is set to 1. Figure 3.6c shows the circuit's truth table.



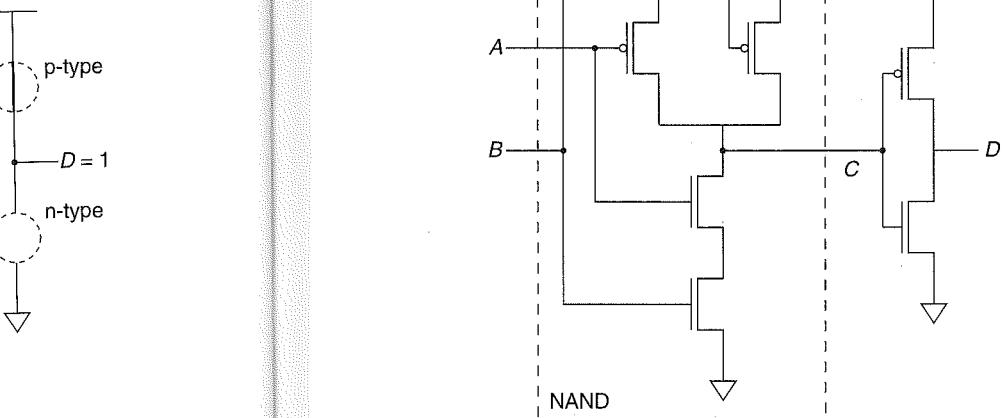


Figure 3.7 The AND gate

Figure 3.7b summarizes in truth table form the behavior of the circuit of Figure 3.7a. Note that the circuit is an AND gate. The circuit shown within the dashed lines (i.e., having output C) is a NOT-AND gate, which we generally abbreviate as NAND.

The gates just discussed are very common in digital logic circuits and in digital computers. There are millions of inverters (NOT gates) in the Pentium IV microprocessor. As a convenience, we can represent each of these gates by standard symbols, as shown in Figure 3.8. The bubble shown in the inverter, NAND, and NOR gates signifies the complement (i.e., NOT) function.

From now on, we will not draw circuits showing the individual transistors. Instead, we will raise our level of abstraction and use the symbols shown in Figure 3.8.

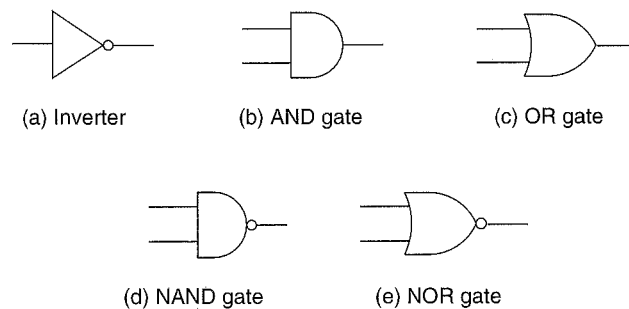


Figure 3.8 Basic logic gates

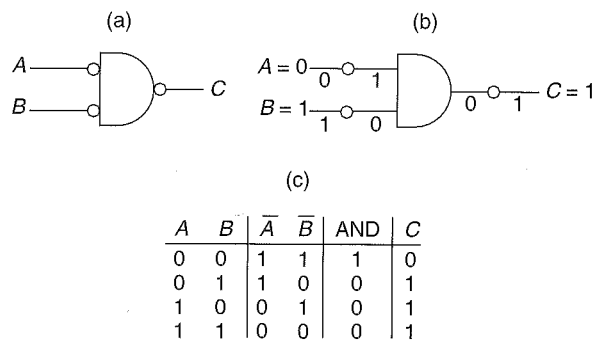


Figure 3.9 DeMorgan's law

### 3.2.4 DeMorgan's Law

Note (see Figure 3.9a) that one can complement an input before applying it to a gate. Consider the effect on the two-input AND gate if we apply the complements of  $A$  and  $B$  as inputs to the gate, and also complement the output of the AND gate. The “bubbles” at the inputs to the AND gate designate that the inputs  $A$  and  $B$  are complemented before they are used as inputs to the AND gate.

Figure 3.9b shows the behavior of this structure for the input combination  $A = 0$ ,  $B = 1$ . For ease of representation, we have moved the bubbles away from the inputs and the output of the AND gate. That way, we can more easily see what happens to each value as it passes through a bubble.

Figure 3.9c summarizes by means of a truth table the behavior of the logic circuit of Figure 3.9a for all four combinations of input values. Note that the NOT of  $A$  is represented as  $\bar{A}$ .

We can describe the behavior of this circuit algebraically:

$$\overline{\bar{A} \text{ AND } \bar{B}} = A \text{ OR } B$$

We can also state this behavior in English:

“It is not the case that both  $A$  and  $B$  are false” is equivalent to saying “At least one of  $A$  and  $B$  is true.”

This equivalence is known as DeMorgan's law. Is there a similar result if one inverts both inputs to an OR gate, and then inverts the output?

### 3.2.5 Larger Gates

Before we leave the topic of logic gates, we should note that the notion of AND, OR, NAND, and NOR gates extends to larger numbers of inputs. One could build a three-input AND gate or a four-input OR gate, for example. An  $n$ -input AND gate has an output value of 1 only if ALL  $n$  input variables have values of 1. If any of the  $n$  inputs has a value of 0, the output of the  $n$ -input AND gate is 0. An  $n$ -input OR gate has an output value of 1 if ANY of the  $n$  input variables has a value of 1. That is, an  $n$ -input OR gate has an output value of 0 only if ALL  $n$  input variables have values of 0.



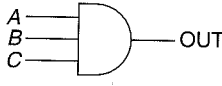
(a)				(b)
A	B	C	OUT	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	

Figure 3.10 A three-input AND gate

Figure 3.10 illustrates a three-input AND gate. Figure 3.10a shows its truth table. Figure 3.10b shows the symbol for a three-input AND gate.

Can you draw a transistor-level circuit for a three-input AND gate? How about a four-input AND gate? How about a four-input OR gate?

### 3.3 Combinational Logic Circuits

Now that we understand the workings of the basic logic gates, the next step is to build some of the logic structures that are important components of the microarchitecture of a computer.

There are fundamentally two kinds of logic structures, those that include the storage of information and those that do not. In Sections 3.4, 3.5, and 3.6, we will deal with structures that store information. In this section, we will deal with those that do not. These structures are sometimes referred to as *decision elements*. Usually, they are referred to as *combinational logic structures*, because their outputs are strictly dependent on the combination of input values that are being applied to the structure *right now*. Their outputs are not at all dependent on any past history of information that is stored internally, since no information can be stored internally in a combinational logic circuit.

We will next examine a decoder, a mux, and a full adder.

#### 3.3.1 Decoder

Figure 3.11 shows a logic gate description of a two-input decoder. A decoder has the property that exactly one of its outputs is 1 and all the rest are 0s. The one output that is logically 1 is the output corresponding to the input pattern that it is expected to detect. In general, decoders have  $n$  inputs and  $2^n$  outputs. We say the output line that detects the input pattern is *asserted*. That is, that output line has the value 1, rather than 0 as is the case for all the other output lines. In Figure 3.11, note that for each of the four possible combinations of inputs  $A$  and  $B$ , exactly one output has the value 1 at any one time. In Figure 3.11b, the input to the decoder is 10, resulting in the third output line being asserted.

The decoder is useful in determining how to interpret a bit pattern. We will see in Chapter 5 that the work to be carried out by each instruction in the LC-3 is

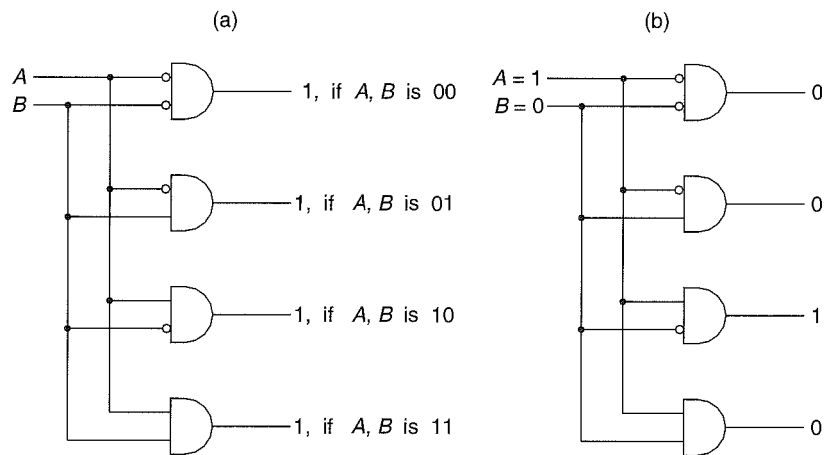


Figure 3.11 A two-input decoder

determined by a four-bit pattern, called an *opcode*, that is part of the instruction. A 4-to-16 decoder is a simple combinational logic structure for identifying what work is to be performed by each instruction.

### 3.3.2 Mux

Figure 3.12a shows a gate-level description of a two-input multiplexer, more commonly referred to as a *mux*. The function of a mux is to select one of the inputs and connect it to the output. The select signal ( $S$  in Figure 3.12) determines which input is connected to the output. The mux of Figure 3.12 works as follows: Suppose  $S = 0$ , as shown in Figure 3.12b. Since the output of an AND gate is 0 unless all inputs are 1, the output of the rightmost AND gate is 0. Also, the output of the leftmost AND gate is whatever the input  $A$  is. That is, if  $A = 0$ , then the output of the leftmost AND gate is 0, and if  $A = 1$ , then the output is 1. Since the output of the rightmost AND gate is 0, it has no effect on the OR gate. Consequently, the output at  $C$  is exactly the same as the output of the leftmost AND gate. The net result of all this is that if  $S = 0$ , the output  $C$  is identical to the input  $A$ .

On the other hand, if  $S = 1$ , it is  $B$  that is ANDed with 1, resulting in the output of the OR gate having the value of  $B$ .

In summary, the output  $C$  is always connected to either the input  $A$  or the input  $B$ —which one depends on the value of the select line  $S$ . We say  $S$  selects the source of the mux (either  $A$  or  $B$ ) to be routed through to the output  $C$ . Figure 3.12c shows the standard representation for a mux.

In general, a mux consists of  $2^n$  inputs and  $n$  select lines. Figure 3.13a shows a gate-level description of a four-input mux. It requires two select lines. Figure 3.13b shows the standard representation for a four-input mux.

Can you construct the gate-level representation for an eight-input mux? How many select lines must you have?

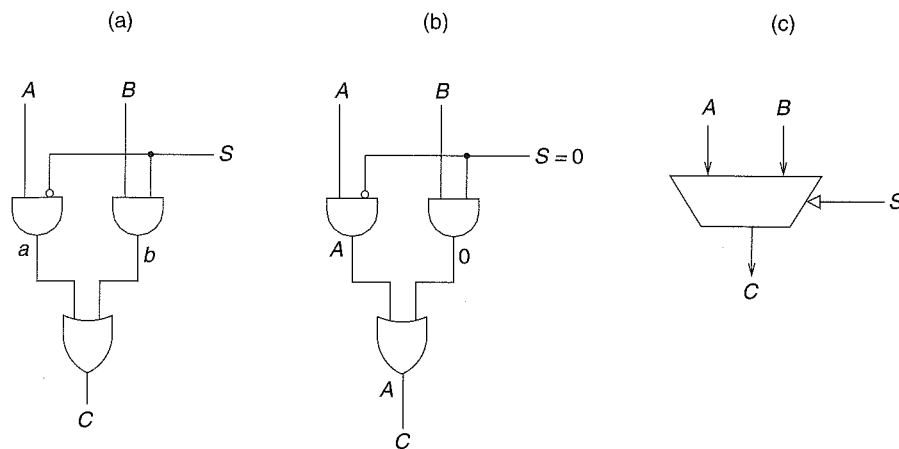


Figure 3.12 A 2-to-1 mux

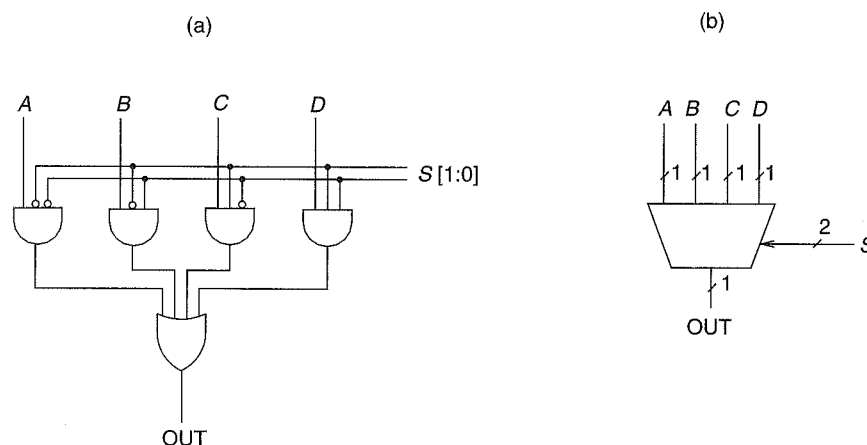


Figure 3.13 A four-input mux

### 3.3.3 Full Adder

In Chapter 2, we discussed binary addition. Recall that a simple algorithm for binary addition is to proceed as you have always done in the case of decimal addition, from right to left, one column at a time, adding the two digits from the two values plus the carry in, and generating a sum digit and a carry to the next column. The only difference is you get a carry after 1, rather than after 9.

Figure 3.14 is a truth table that describes the result of binary addition on *one column* of bits within two  $n$ -bit operands. At each column, there are three values that must be added: one bit from each of the two operands and the carry from the previous column. We designate these three bits as  $a_i$ ,  $b_i$ , and  $carry_i$ . There are two results, the sum bit ( $s_i$ ) and the carryover to the next column,  $carry_{i+1}$ . Note that if only one of the three bits equals 1, we get a sum of 1, and no carry (i.e.,  $carry_{i+1} = 0$ ). If two of the three bits equal 1, we get a sum of 0, and a carry

$a_i$	$b_i$	$carry_i$	$carry_{i+1}$	$s_i$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 3.14 A truth table for a binary adder

of 1. If all three bits equal 1, the sum is 3, which in binary addition corresponds to a sum of 1 and a carry of 1.

Figure 3.15 is the gate-level description of the truth table of Figure 3.14. Note that each AND gate in Figure 3.15 produces an output 1 for exactly one of the eight input combinations of  $a_i$ ,  $b_i$ , and  $carry_i$ . The output of the OR gate for  $C_{i+1}$  must be 1 in exactly those cases where the corresponding input combinations in Figure 3.14 produce an output 1. Therefore the inputs to the OR gate that generates  $C_{i+1}$  are the outputs of the AND gates corresponding to those input combinations. Similarly, the inputs to the OR gate that generates  $S_i$  are the outputs of the AND gates corresponding to the input combinations that require an output 1 for  $S_i$  in the truth table of Figure 3.14.

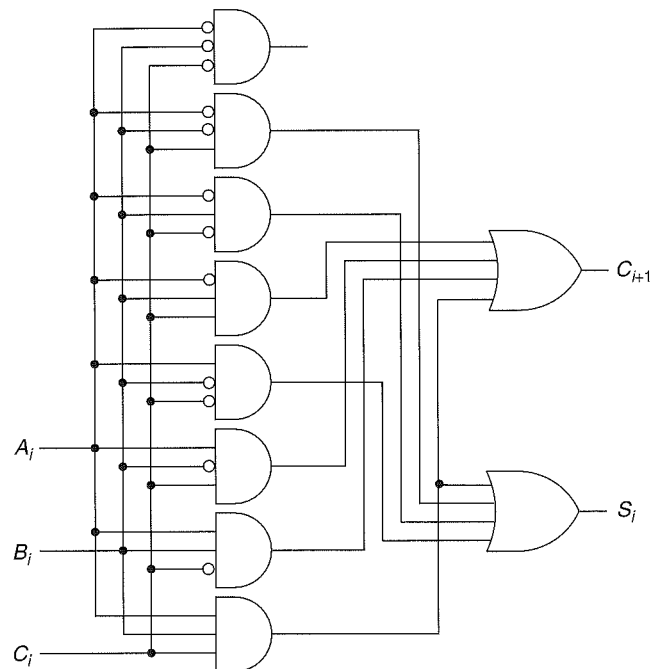


Figure 3.15 Gate-level description of a full adder

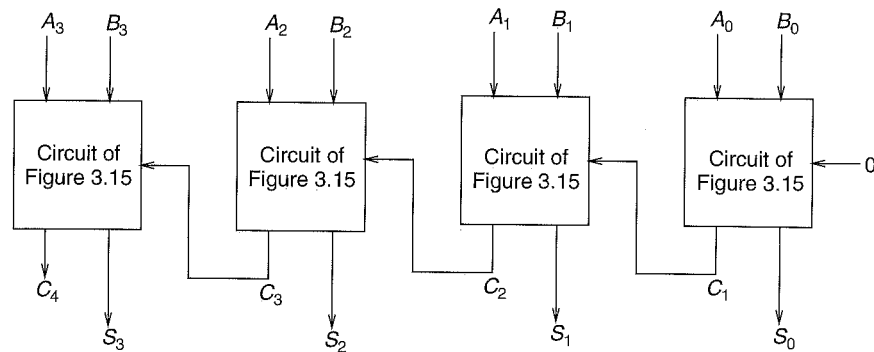


Figure 3.16 A circuit for adding two 4-bit binary numbers

Note that since the input combination 000 does not result in an output 1 for either  $C_{i+1}$  or  $S_i$ , its corresponding AND gate is not an input to either of the two OR gates.

We call the logic circuit of Figure 3.15 that provides three inputs ( $a_i$ ,  $b_i$ , and  $carry_i$ ) and two outputs (the sum bit  $s_i$  and the carryover to the next column  $carry_{i+1}$ ) a *full adder*.

Figure 3.16 illustrates a circuit for adding two 4-bit binary numbers, using four of the full adder circuits of Figure 3.15. Note that the carryout of column  $i$  is an input to the addition performed in column  $i + 1$ .

### 3.3.4 The Programmable Logic Array (PLA)

Figure 3.15 illustrates a very common building block for implementing any collection of logic functions one wishes to. The building block is called a programmable logic array (PLA). It consists of an array of AND gates (called an AND array) followed by an array of OR gates (called an OR array). The number of AND gates corresponds to the number of input combinations (rows) in the truth table. For  $n$  input logic functions, we need a PLA with  $2^n$   $n$ -input AND gates. In Figure 3.17, we have  $2^3$  3-input AND gates. The number of OR gates corresponds to the number of output columns in the truth table. The implementation algorithm is simply to connect the output of an AND gate to the input of an OR gate if the corresponding row of the truth table produces an output 1 for that output column. Hence the notion of programmable. That is, we say we program the connections from AND gate outputs to OR gate inputs to implement our desired logic functions.

Figure 3.15 showed eight AND gates connected to two OR gates since our requirement was to implement two functions (sum and carry) of three input variables. Figure 3.17 shows a PLA that can implement any four functions of three variables one wishes to, by appropriately connecting AND gate outputs to OR gate inputs.

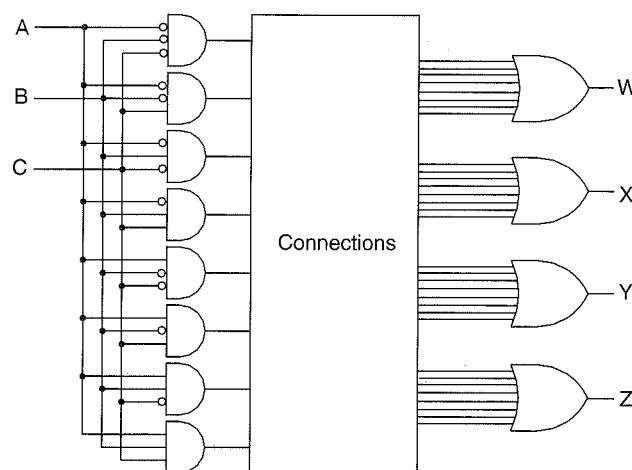


Figure 3.17 A programmable logic array

### 3.3.5 Logical Completeness

Before we leave the topic of combinational logic circuits, it is worth noting an important property of building blocks for logic circuits: logical completeness. We showed in Section 3.3.4 that any logic function we wished to implement could be accomplished with a PLA. We saw that the PLA consists of only AND gates, OR gates, and inverters. That means that any logic function we wish to implement can be accomplished, provided that enough AND, OR, and NOT gates are available. We say that the set of gates {AND, OR, NOT} is *logically complete* because we can build a circuit to carry out the specification of any truth table we wish without using any other kind of gate. That is, the set of gates {AND, OR, and NOT} is logically complete because a barrel of AND gates, a barrel of OR gates, and a barrel of NOT gates are sufficient to build a logic circuit that carries out the specification of any desired truth table. The barrels may have to be big ones, but the point is, we do not need any other kind of gate to do the job.

## 3.4 Basic Storage Elements

Recall our statement at the beginning of Section 3.3 that there are two kinds of logic structures, those that involve the storage of information and those that do not. We have discussed three examples of those that do not: the decoder, the mux, and the full adder. Now we are ready to discuss logic structures that do include the storage of information.

### 3.4.1 The R-S Latch

A simple example of a storage element is the R-S latch. It can store one bit of information. The R-S latch can be implemented in many ways, the simplest being

the one shown in Figure 3.18. Two 2-input NAND gates are connected such that the output of each is connected to one of the inputs of the other. The remaining inputs  $S$  and  $R$  are normally held at a logic level 1.

The R-S latch works as follows: We start with what we call the *quiescent* (or quiet) state, where inputs  $S$  and  $R$  both have logic value 1. We consider first the case where the output  $a$  is 1. Since that means the input  $A$  equals 1 (and we know the input  $R$  equals 1 since we are in the quiescent state), the output  $b$  must be 0. That, in turn, means the input  $B$  must be 0, which results in the output  $a$  equal to 1. As long as the inputs  $S$  and  $R$  remain 1, the state of the circuit will not change. We say the R-S latch stores the value 1 (the value of the output  $a$ ).

If, on the other hand, we assume the output  $a$  is 0, then the input  $A$  must be 0, and the output  $b$  must be 1. This, in turn, results in the input  $B$  equal to 1, and combined with the input  $S$  equal to 1 (again due to quiescence) results in the output  $a$  equal to 0. Again, as long as the inputs  $S$  and  $R$  remain 1, the state of the circuit will not change. In this case, we say the R-S latch stores the value 0.

The latch can be set to 1 by momentarily setting  $S$  to 0, provided we keep the value of  $R$  at 1. Similarly, the latch can be set to 0 by momentarily setting  $R$  to 0, provided we keep the value of  $S$  at 1. We use the term *set* to denote setting a variable to 0 or 1, as in "set to 0" or "set to 1." In addition, we often use the term *clear* to denote the act of setting a variable to 0.

If we clear  $S$ , then  $a$  equals 1, which in turn causes  $A$  to equal 1. Since  $R$  is also 1, the output at  $b$  must be 0. This causes  $B$  to be 0, which in turn makes  $a$  equal to 1. If we now return  $S$  to 1, it does not affect  $a$ , since  $B$  is also 0, and only one input to a NAND gate must be 0 in order to guarantee that the output of the NAND gate is 1. Thus, the latch continues to store a 1 long after  $S$  returns to 1.

In the same way, we can clear the latch (set the latch to 0) by momentarily setting  $R$  to 0.

We should also note that in order for the R-S latch to work properly, one must take care that it is never the case that both  $S$  and  $R$  are allowed to be set to 0 at the same time. If that does happen, the outputs  $a$  and  $b$  are both 1, and the final state of the latch depends on the electrical properties of the transistors making up the gates and not on the logic being performed. How the electrical properties of the transistors will determine the final state in this case is a subject we will have to leave for a later semester.

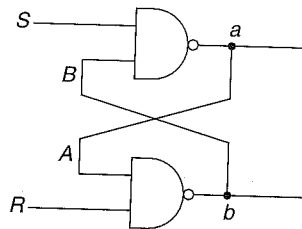


Figure 3.18 An R-S latch

### 3.4.2 The Gated D Latch

To be useful, it is necessary to control when a latch is set and when it is cleared. A simple way to accomplish this is with the gated latch.

Figure 3.19 shows a logic circuit that implements a gated  $D$  latch. It consists of the R-S latch of Figure 3.18, plus two additional gates that allow the latch to be set to the value of  $D$ , but *only* when  $WE$  is asserted.  $WE$  stands for *write enable*. When  $WE$  is not asserted (i.e., when  $WE$  equals 0), the outputs  $S$  and  $R$  are both equal to 1. Since  $S$  and  $R$  are also inputs to the R-S latch, if they are kept at 1, the value stored in the latch remains unchanged, as we explained in Section 3.4.1. When  $WE$  is momentarily asserted (i.e., set to 1), exactly one of the outputs  $S$  or  $R$  is set to 0, depending on the value of  $D$ . If  $D$  equals 1, then  $S$  is set to 0. If  $D$  equals 0, then both inputs to the lower NAND gate are 1, resulting in  $R$  being set to 0. As we saw earlier, if  $S$  is set to 0, the R-S latch is set to 1. If  $R$  is set to 0, the R-S latch is set to 0. Thus, the R-S latch is set to 1 or 0 according to whether  $D$  is 1 or 0. When  $WE$  returns to 0,  $S$  and  $R$  return to 1, and the value stored in the R-S latch persists.

### 3.4.3 A Register

We have already seen in Chapter 2 that it is useful to deal with values consisting of more than one bit. In Chapter 5, we will introduce the LC-3 computer, where most values are represented by 16 bits. It is useful to be able to store these larger numbers of bits as self-contained units. The *register* is a structure that stores a number of bits, taken together as a unit. That number can be as large as is useful or as small as 1. In the LC-3, we will need many 16-bit registers, and also a few one-bit registers. We will see in Figure 3.33, which describes the internal structure of the LC-3, that PC, IR, and MAR are all 16-bit registers, and that  $N$ ,  $Z$ , and  $P$  are all one-bit registers.

Figure 3.20 shows a four-bit register made up of four gated  $D$  latches. The four-bit value stored in the register is  $Q_3, Q_2, Q_1, Q_0$ . The value  $D_3, D_2, D_1, D_0$  can be written into the register when  $WE$  is asserted.

*Note:* A common shorthand notation to describe a sequence of bits that are numbered as just described is  $Q[3:0]$ . That is, each bit is assigned its own bit number. The rightmost bit is bit [0], and the numbering continues from right to

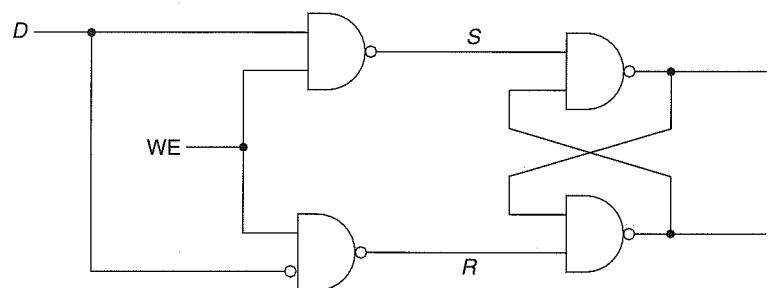


Figure 3.19 A gated D latch



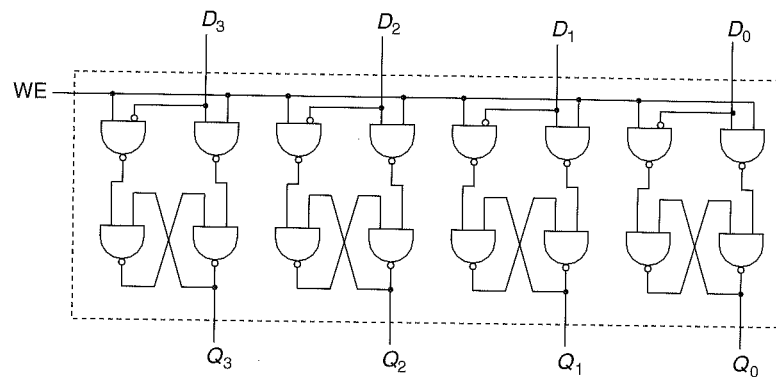


Figure 3.20 A four-bit register

left. If there are  $n$  bits, the leftmost bit is bit  $[n - 1]$ . For example, in the following 16-bit pattern,

0011101100011110

bit  $[15]$  is 0, bit  $[14]$  is 0, bit  $[13]$  is 1, bit  $[12]$  is 1, and so on.

We can designate a subunit of this pattern with the notation  $Q[l:r]$ , where  $l$  is the leftmost bit in the subunit and  $r$  is the rightmost bit in the subunit. We call such a subunit a *field*.

In this 16-bit pattern, if  $A[15:0]$  is the entire 16-bit pattern, then, for example:

$A[15:12]$  is 0011  
 $A[13:7]$  is 1110110  
 $A[2:0]$  is 110  
 $A[1:1]$  is 1

We should also point out that the numbering scheme from right to left is purely arbitrary. We could just as easily have designated the leftmost bit as bit  $[0]$  and numbered them from left to right. Indeed, many people do. So, it is not important whether the numbering scheme is left to right or right to left. But it is important that the bit numbering be consistent in a given setting, that is, that it is always done the same way. In our work, we will always number bits from right to left.

### 3.5 The Concept of Memory

We now have all the tools we need to describe one of the most important structures in the electronic digital computer, its *memory*. We will see in Chapter 4 how memory fits into the basic scheme of computer processing, and you will see throughout the rest of the book and indeed the rest of your work with computers how important the concept of memory is to computing.

Memory is made up of a (usually large) number of locations, each uniquely identifiable and each having the ability to store a value. We refer to the unique

identifier associated with each memory location as its *address*. We refer to the number of bits of information stored in each location as its *addressability*.

For example, an advertisement for a personal computer might say, "This computer comes with 16 megabytes of memory." Actually, most ads generally use the abbreviation 16 MB. This statement means, as we will explain momentarily, that the computer system includes 16 million memory locations, each containing 1 byte of information.

### 3.5.1 Address Space

We refer to the total number of uniquely identifiable locations as the memory's *address space*. A 16 MB memory, for example, refers to a memory that consists of 16 million uniquely identifiable memory locations.

Actually, the number *16 million* is only an approximation, due to the way we identify memory locations. Since everything else in the computer is represented by sequences of 0s and 1s, it should not be surprising that memory locations are identified by binary addresses as well. With  $n$  bits of address, we can uniquely identify  $2^n$  locations. Ten bits provide 1,024 locations, which is approximately 1,000. If we have 20 bits to represent each address, we have  $2^{20}$  uniquely identifiable locations, which is approximately 1 million. Thus 16 mega really corresponds to the number of uniquely identifiable locations that can be specified with 24 address bits. We say the address space is  $2^{24}$ , which is *exactly* 16,777,216 locations, rather than 16,000,000, although we colloquially refer to it as 16 million.

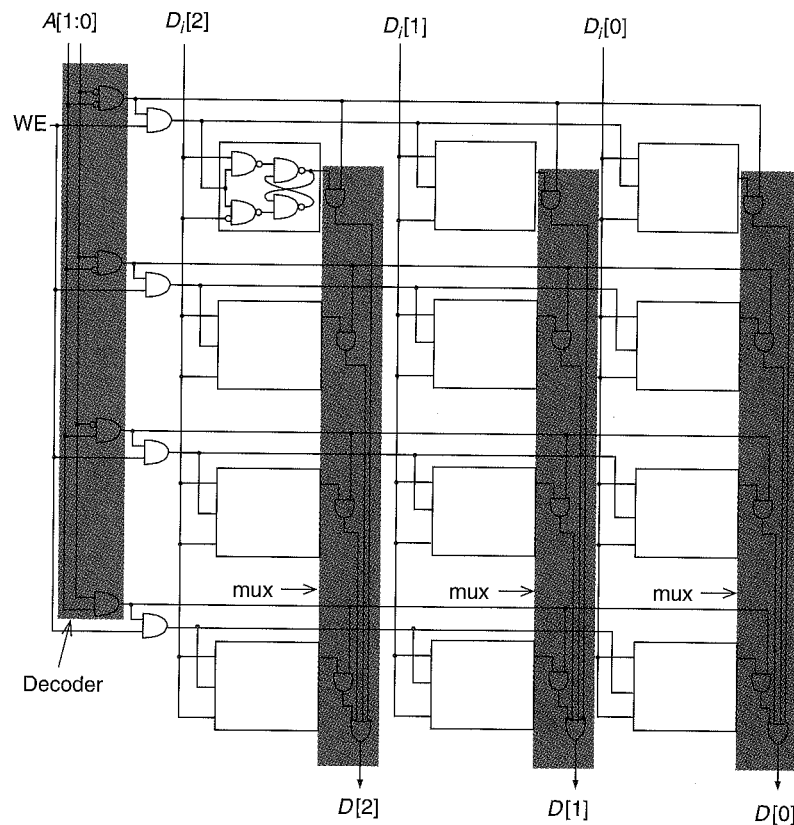
### 3.5.2 Addressability

The number of bits stored in each memory location is the memory's *addressability*. A 16 megabyte memory is a memory consisting of 16,777,216 memory locations, each containing 1 byte (i.e., 8 bits) of storage. Most memories are byte-addressable. The reason is historical; most computers got their start processing data, and one character stroke on the keyboard corresponds to one 8-bit ASCII character, as we learned in Chapter 2. If the memory is byte-addressable, then each ASCII code occupies one location in memory. Uniquely identifying each byte of memory allowed individual bytes of stored information to be changed easily.

Many computers that have been designed specifically to perform large scientific calculations are 64-bit addressable. This is due to the fact that numbers used in scientific calculations are often represented as 64-bit floating point quantities. Recall that we discussed the floating point data type in Chapter 2. Since scientific calculations are likely to use numbers that require 64 bits to represent them, it is reasonable to design a memory for such a computer that stores one such number in each uniquely identifiable memory location.

### 3.5.3 A $2^2$ -by-3-Bit Memory

Figure 3.21 illustrates a memory of size  $2^2$  by 3 bits. That is, the memory has an address space of four locations, and an addressability of 3 bits. A memory of size  $2^2$  requires 2 bits to specify the address. A memory of addressability 3 stores 3 bits

Figure 3.21 A  $2^2$ -by-3-bit memory

of information in each memory location. Accesses of memory require decoding the address bits. Note that the address decoder takes as input  $A[1:0]$  and asserts exactly one of its four outputs, corresponding to the *word line* being addressed. In Figure 3.21, each row of the memory corresponds to a unique three-bit word; thus the term *word line*. Memory can be read by applying the address  $A[1:0]$ , which asserts the word line to be read. Note that each bit of the memory is ANDed with its word line and then ORed with the corresponding bits of the other words. Since only one word line can be asserted at a time, this is effectively a mux with the output of the decoder providing the select function to each bit line. Thus, the appropriate word is read.

Figure 3.22 shows the process of reading location 3. The code for 3 is 11. The address  $A[1:0] = 11$  is decoded, and the bottom word line is asserted. Note that the three other decoder outputs are not asserted. That is, they have the value 0. The value stored in location 3 is 101. These three bits are each ANDed with their word line producing the bits 101, which are supplied to the three output OR gates. Note that all other inputs to the OR gates are 0, since they have been produced by ANDing with unasserted word lines. The result is that  $D[2:0] = 101$ . That is, the value stored in location 3 is output by the OR gates. Memory can be written in a similar fashion. The address specified by  $A[1:0]$  is presented to

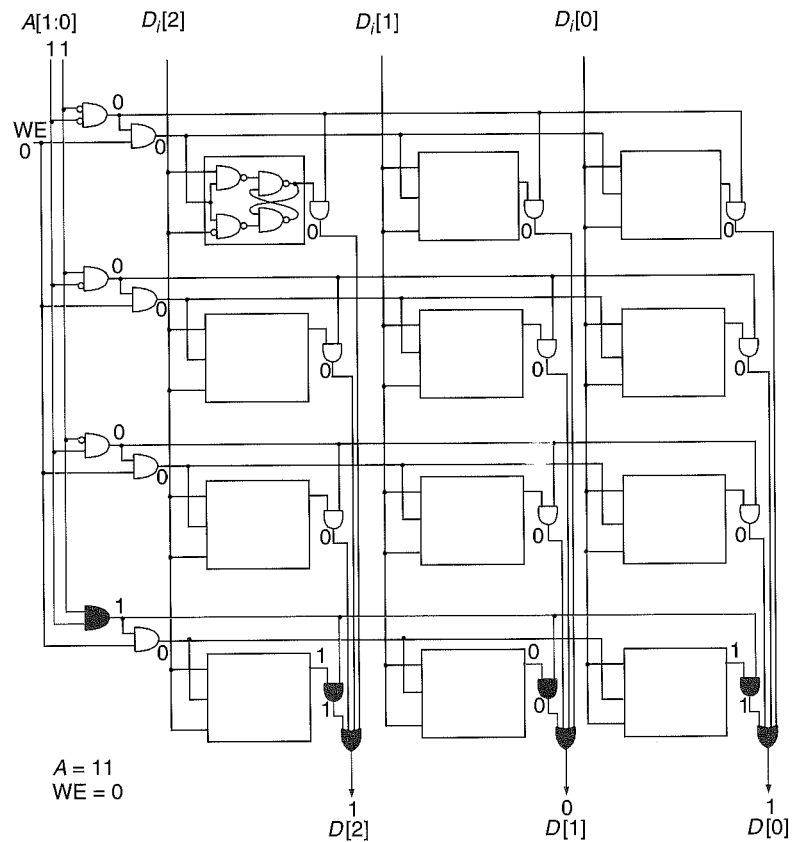


Figure 3.22 Reading location 3 in our 2<sup>2</sup>-by-3-bit memory

the address decoder, resulting in the correct word line being asserted. With WE asserted as well, the three bits  $D_i[2:0]$  can be written into the three gated latches corresponding to that word line.

### 3.6 Sequential Logic Circuits

In Section 3.3, we discussed digital logic structures that process information (decision structures, we call them) wherein the outputs depend solely on the values that are present on the inputs **now**. Examples are muxes, decoders, and full adders. We call these structures combinational logic circuits. In these circuits there is no sense of the past. Indeed, there is no capability for storing any information of anything that happened before the present time. In Sections 3.4 and 3.5, we described structures that do store information—in Section 3.4, some basic storage elements, and in Section 3.5, a simple 2<sup>2</sup>-by-3-bit memory.

In this section, we discuss digital logic structures that can **both** process information (i.e., make decisions) **and** store information. That is, these structures base their decisions not only on the input values now present, but also (and this is

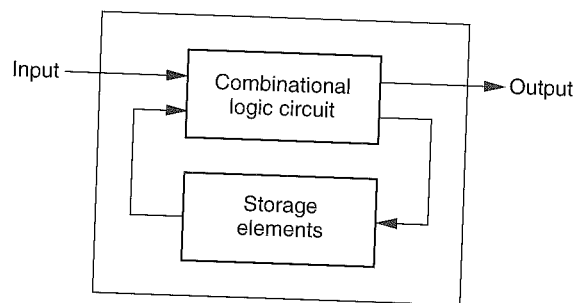


Figure 3.23 Sequential logic circuit block diagram

very important) on what has happened before. These structures are usually called **sequential logic circuits**. They are distinguishable from combinational logic circuits because, unlike combinational logic circuits, they contain storage elements that allow them to keep track of prior history information. Figure 3.23 shows a block diagram of a sequential logic circuit. Note the storage elements. Note, also, that the output can be dependent on both the inputs now and the values stored in the storage elements. The values stored in the storage elements reflect the history of what has happened before.

Sequential logic circuits are used to implement a very important class of mechanisms called finite state machines. We use finite state machines in essentially all branches of engineering. For example, they are used as controllers of electrical systems, mechanical systems, aeronautical systems, and so forth. A traffic light controller that sets the traffic light to red, yellow, or green depends on the light that is currently on (history information) and input information from sensors such as trip wires on the road and optical devices that are monitoring traffic.

We will see in Chapter 4 when we introduce the von Neumann model of a computer that a finite state controller is at the heart of the computer. It controls the processing of information by the computer.

### 3.6.1 A Simple Example: The Combination Lock

A simple example shows the difference between combinational logic structures and sequential logic structures. Suppose one wishes to secure a bicycle with a lock, but does not want to carry a key. A common solution is the combination lock. The person memorizes a “combination” and uses this to open the lock. Two common types of locks are shown in Figure 3.24.

In Figure 3.24a, the lock consists of a dial, with the numbers from 0 to 30 equally spaced around its circumference. To open the lock, one needs to know the “combination.” One such combination could be: R13-L22-R3. If this were the case, one would open the lock by turning the dial two complete turns to the right, and then continuing until the dial points to 13, followed by one complete turn to the left, and then continuing until the dial points to 22, followed by turning the dial again to the right until it points to 3. At that point, the lock opens. What is important here is the *sequence* of the turns. The lock will not open, for example

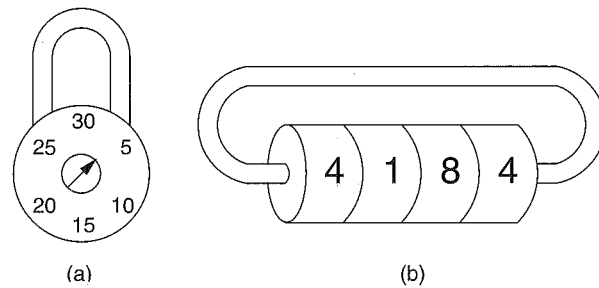


Figure 3.24 Combination locks

if one performed two turns to the right, and then stopped on 20, followed by one complete turn to the left, ending on 22, followed by one turn to the right, ending on 3. That is, even though the final position of the dial is 3, the lock would not open. Why? Because the lock stores the previous rotations and makes its decision (open or don't open) on the basis of the current input value (R3) *and* the history of the past operations. This mechanism is a simple example of a sequential structure.

Another type of lock is shown in Figure 3.24b. The mechanism consists of (usually) four wheels, each containing the digits 0 through 9. When the digits are lined up properly, the lock will open. In this case, the combination is the set of four digits. Whether or not this lock opens is totally independent of the past rotations of the four wheels. The lock does not care at all about past rotations. The only thing important is the current value of each of the four wheels. This is a simple example of a combinational structure.

It is curious that in our everyday speech, both mechanisms are referred to as "combination locks." In fact, only the lock of Figure 3.24b is a combinational lock. The lock of Figure 3.24a would be better called a sequential lock!

### 3.6.2 The Concept of State

For the mechanism of Figure 3.24a to work properly, it has to keep track of the sequence of rotations leading up to the opening of the lock. In particular, it has to differentiate the correct sequence R13-L22-R3 from all other sequences. For example, R13-L29-R3 must not be allowed to open the lock. Likewise, R10-L22-R3 must also not be allowed to open the lock. The problem is that, at any one time, the only external input to the lock is the current rotation.

For the lock of Figure 3.24a to work, it must identify several relevant situations, as follows:

- A. The lock is not open, and NO relevant operations have been performed.
- B. The lock is not open, but the user has just completed the R13 operation.
- C. The lock is not open, but the user has just completed R13, followed by L22.
- D. The lock is open.

We have labeled these four situations A, B, C, and D. We refer to each of these situations as the *state* of the lock.

The notion of **state** is a very important concept in computer engineering, and actually, in just about all branches of engineering. The state of a mechanism—more generally, the state of a system—is a snapshot of that system in which all relevant items are explicitly expressed.

That is: *The state of a system is a snapshot of all the relevant elements of the system at the moment the snapshot is taken.*

In the case of the lock of Figure 3.24a, there are four states A, B, C, and D. Either the lock is open (State D), or if it is not open, we have already performed either zero (State A), one (State B), or two (State C) correct operations. This is the sum total of all possible states that can exist. Exercise: Why is that the case? That is, what would be the snapshot of a fifth state that describes a possible situation for the combination lock?

There are many common examples of systems that can be easily described by means of states.

The state of a game of basketball can be described by the scoreboard in the basketball arena. Figure 3.25 shows the state of the basketball game as Texas 73, Oklahoma 68, 7 minutes and 38 seconds left in the second half, 14 seconds left on the shot clock, Texas with the ball, and Texas and Oklahoma each with four team fouls. This is a snapshot of the basketball game. It describes the state of the basketball game right now. If, 12 seconds later, a Texas player were to score a two-point shot, the new state would be described by the updated scoreboard. That is, the score would then be Texas 75, Oklahoma 68, the time remaining in the game would be 7 minutes and 26 seconds, the shot clock would be back to 25 seconds, and Oklahoma would have the ball.

The game of tic-tac-toe can also be described in accordance with the notion of state. Recall that the game is played by two people (or, in our case, a person

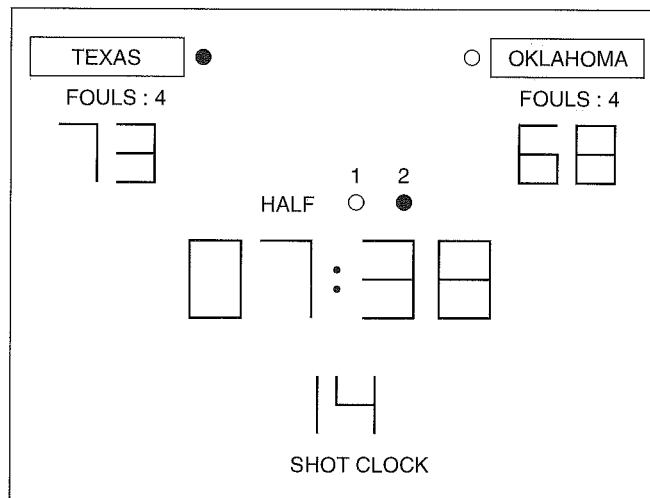


Figure 3.25 An example of a state

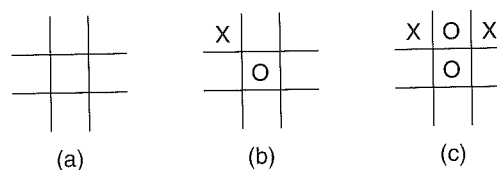


Figure 3.26 Three states in a tic-tac-toe machine

and the computer). The state is a snapshot of the game in progress each time the computer asks the person to make a move. The game is played as follows: There are nine locations on the diagram. The person and then the computer take turns placing an X (the person) and an O (the computer) in an empty location. The person goes first. The winner is the first to place three symbols (three Xs for the person, three Os for the computer) in a straight line, either vertically, horizontally, or diagonally.

The initial state, before either the person or computer has had a turn, is shown in Figure 3.26a. Figure 3.26b shows a possible state of the game when the person is prompted for a second move, if he/she put an X in the upper left corner as the first move. In the state shown, the computer put an O in the middle square as its first move. Figure 3.26c shows a possible state of the game when the person is being prompted for a third move if he/she put an X in the upper right corner on the second move (after putting the first X in the upper left corner). In the state shown, the computer put its second O in the upper middle location.

### 3.6.3 Finite State Machines

We have seen that a state is a snapshot of all relevant parts of a system at a particular point in time. At other times, that system can be in other states. The behavior of a system can often be best understood by describing it as a *finite state machine*.

A finite state machine consists of five elements:

1. a finite number of states
2. a finite number of external inputs
3. a finite number of external outputs
4. an explicit specification of all state transitions
5. an explicit specification of what determines each external output value.

The set of states represents all possible situations (or snapshots) that the system can be in. Each state transition describes what it takes to get from one state to another.

#### The State Diagram

A finite state machine can be conveniently represented by means of a *state diagram*. Figure 3.27 is an example of a state diagram. A state diagram is drawn as a set of circles, where each circle corresponds to one state, and a set of connections



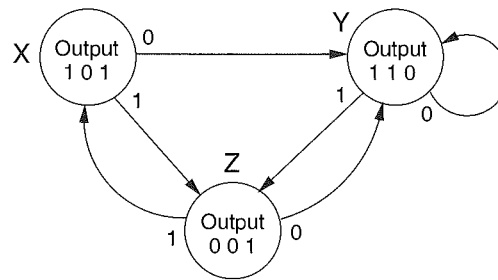


Figure 3.27 A state diagram

between some of the states, where each connection is drawn as an arrow. The more sophisticated term for “connection” is *arc*. Each arc identifies the transition from one state to another. The arrowhead on each arc specifies which state the system is coming from, and which state it is going to. We refer to the state the system is coming from as the *current state*, and the state it is going to as the *next state*. The finite state machine represented by the state diagram of Figure 3.27 consists of three states, with six state transitions. Note that there is no state transition from state Y to state X.

It is often the case that from a current state there are multiple transitions to next states. The state transition that occurs depends on the values of the external inputs. In Figure 3.27, if the current state is state X and the external input has value 0, the next state is state Y. If the current state is state X and the external input has the value 1, the next state is state Z. In short, the next state is determined by the combination of the current state and the current external input.

The output values of a system can be determined just by the current state of the system, or they can be determined by the combination of the current state and the values of the current external inputs. In all the cases we will study, the output values are specified by the current state of the system. In Figure 3.27, the output is 101 when the system is in state X, the output is 110 when the system is in state Y, and 001 when the system is in state Z.

Figure 3.28 is a state diagram of the combination lock of Figure 3.24a, for which the correct combination is R13, L22, R3. Note the four states, labeled A, B, C, D, identifying whether the lock is open, or, in the cases where it is not open, the number of correct rotations performed up to now. The external inputs are the possible rotation operations. The output is the condition “open” or “do not open.” The output is explicitly associated with each state. That is, in states A, B, and C, the output is “do not open.” In state D, the output is “open.” Note further that the “arcs” out of each state comprise all possible operations that one could perform when the mechanism is in that state. For example, when in state B, all possible rotations can be described as (1) L22 and (2) everything except L22. Note that there are two arrows emanating from state B in Figure 3.28, corresponding to these two cases.

We could similarly draw a state diagram for the basketball game we described earlier, where each state would be one possible configuration of the scoreboard. A transition would occur if either the referee blew a whistle or the other team got the

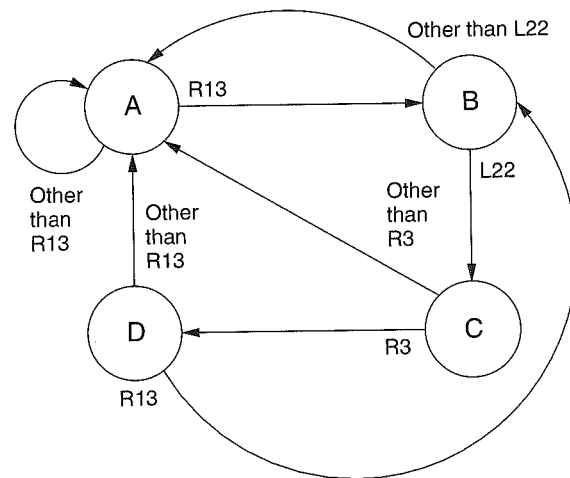


Figure 3.28 State diagram of the combination lock of Figure 3.24

ball. We showed earlier the transition that would be caused by Texas scoring a two-point shot. Clearly, the number of states in the finite state machine describing a basketball game would be huge. Also clearly, the number of legitimate transitions from one state to another is small, compared to the number of arcs one could draw connecting arbitrary pairs of states. The input is the activity that occurred on the basketball court since the last transition. Some input values are: Texas scored two points, Oklahoma scored three points, Texas stole the ball, Oklahoma successfully rebounded a Texas shot, and so forth. The output is the final result of the game. The output has three values: Game still in progress, Texas wins, Oklahoma wins.

Can one have an arc from a state where the score is Texas 30, Oklahoma 28 to a state where the score is tied, Texas 30, Oklahoma 30? See Exercise 3.38.

Is it possible to have two states, one where Texas is ahead 30-28 and the other where the score is tied 30-30, but no arc between the two? See Exercise 3.39.

### The Clock

There is still one important property of the behavior of finite state machines that we have not discussed—the mechanism that triggers the transition from one state to the next. In the case of the “sequential” combination lock, the mechanism is the completion of rotating the dial in one direction, and the start of rotating the dial in the opposite direction. In the case of the basketball game, the mechanism is triggered by the referee blowing a whistle, or someone scoring or the other team otherwise getting the ball.

Frequently, the mechanism that triggers the transition from one state to the next is a clock circuit. A clock circuit, or, more commonly, a *clock*, is a signal whose value alternates between 0 volts and some specified fixed voltage. In digital logic terms, a clock is a signal whose value alternates between 0 and 1. Figure 3.29 illustrates the value of the clock signal as a function of time. A *clock cycle* is one interval of the repeated sequence of intervals shown in Figure 3.29.

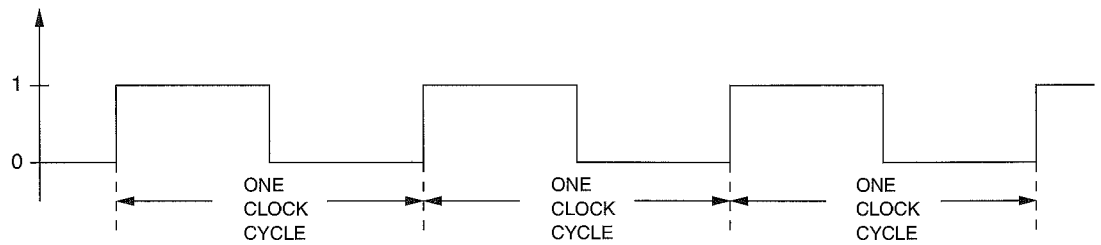


Figure 3.29 A clock signal

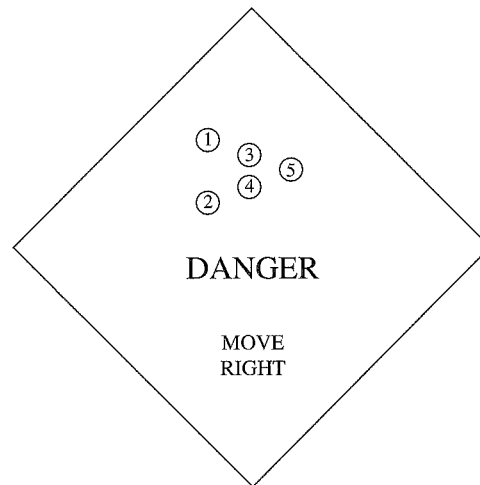


Figure 3.30 A traffic danger sign

In electronic circuit implementations of a finite state machine, the transition from one state to another occurs at the start of each clock cycle.

### 3.6.4 An Example: The Complete Implementation of a Finite State Machine

We conclude this section with the logic specification of a sequential logic circuit that implements a finite state machine. Our example is a controller for a traffic danger sign, as shown in Figure 3.30. Note the sign says, "Danger, Move Right." The sign also contains five lights (labeled 1 through 5 in the figure).

Like many sequential logic circuits, the purpose of our controller is to direct the behavior of a system. In our case, the system is the set of lights on the traffic danger sign. The controller's job is to have the five lights flash on and off as follows: During one cycle, all lights will be off. The next cycle, lights 1 and 2 will be on. The next cycle, lights 1, 2, 3, and 4 will be on. The next cycle, all five lights will be on. Then the sequence repeats: next cycle, no lights on, followed by 1 and 2 on, followed by 1, 2, 3, and 4 on, and so forth. Each cycle is to last  $\frac{1}{2}$  second.

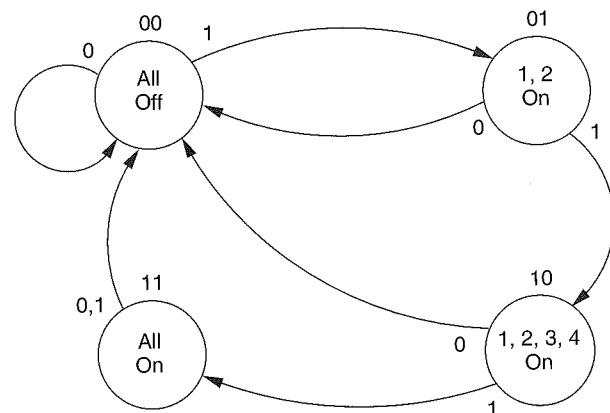


Figure 3.31 State diagram for the traffic danger sign controller

Figure 3.31 is a finite state machine that describes the behavior of the traffic danger sign. Note that there are four states, one for each of the four relevant situations. Note the transitions from each state to the next state. If the switch is on (input = 1), the lights flash in the sequence described. If the switch is turned off, the state always transfers immediately to the “all off” state.

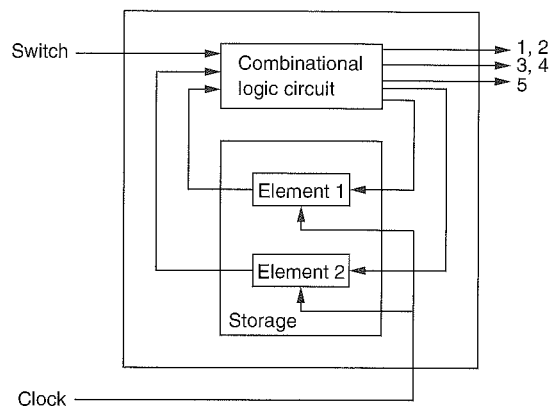
Figure 3.32 shows the implementation of a sequential logic circuit that implements the finite state machine of Figure 3.31. Figure 3.32a is a block diagram, similar to Figure 3.23. Note that there is one external input, a switch that determines whether or not the lights should flash. There are three external outputs, one to control when lights 1 and 2 are on, one to control when lights 3 and 4 are on, and one to control when light 5 is on. Note that there are two internal storage elements that are needed to keep track of which state the controller is in, which is determined by the past behavior of the traffic danger sign. Note finally that there is a clock signal that must have a cycle time of  $\frac{1}{2}$  second in order for the state transitions to occur every  $\frac{1}{2}$  second.

The only relevant history that must be retained is the state that we are transitioning from. Since there are only four states, we can uniquely identify them with two bits. Therefore, only two storage elements are needed. Figure 3.31 shows the two-bit code used to identify each of the four states.

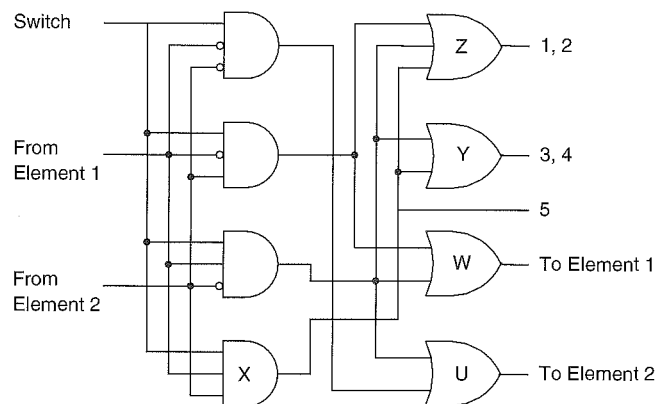
### Combinational Logic

Figure 3.32b shows the combinational logic circuit required to complete the implementation of the controller for the traffic danger sign. Two sets of outputs of the combinational logic circuit are required for the controller to work properly: a set of external outputs for the lights and a set of internal outputs to determine the inputs to the two storage elements that keep track of the state.

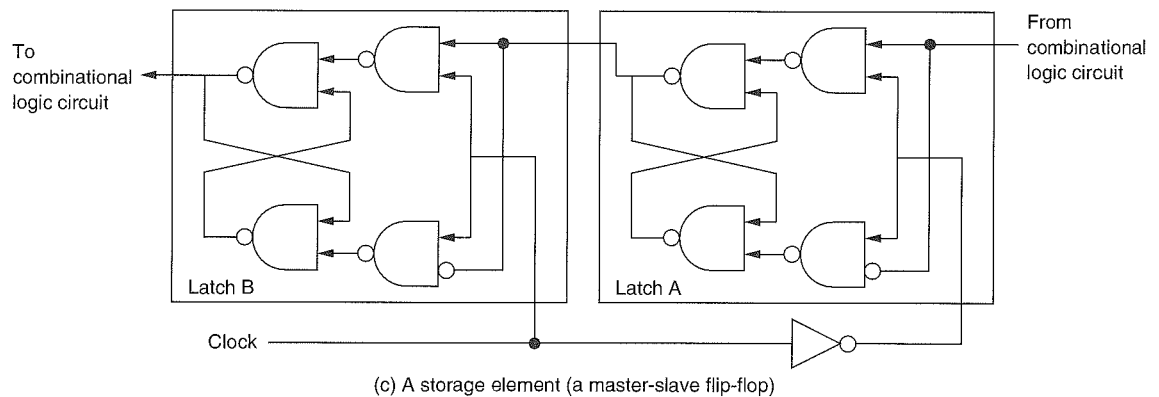
First, let us look at the outputs that control the lights. As we have said, there are only three outputs necessary to control the lights. Light 5 is controlled by the output of the AND gate labeled X, since the only time light 5 is on is if the switch is on, and the controller is in state 11. Lights 3 and 4 are controlled by the output of the OR gate labeled Y, since there are two states in which those lights are on,



(a) Block diagram



(b) The combinational logic circuit



(c) A storage element (a master-slave flip-flop)

Figure 3.32 Sequential logic circuit implementation of Figure 3.30

those labeled 10 and 11. Why are lights 1 and 2 controlled by the output of the OR gate labeled Z? See Exercise 3.42.

Next, let us look at the internal outputs that control the storage elements. Storage element 1 should be set to 1 for the next clock cycle if the next state is to be 10 or 11. This is true only if the switch is on and the current state is either 01 or 10. Therefore the output signal that will make storage element 1 be 1 in the next clock cycle is the output of the OR gate labeled W. Why is the next state of storage element 2 controlled by the output of the OR gate labeled U? See Exercise 3.42.

### Storage Elements

The last piece of logic needed for the traffic danger sign controller is the logic circuit for the two storage elements shown in Figure 3.32a. Why can't we use the gated *D* latch discussed in Section 3.4, one might ask? The reason is as follows: During the current clock cycle the output of the storage element is an internal input to the combinational logic circuit, and the output of the combinational logic circuit is an input to the storage element that must not take effect until the *start* of the next clock cycle. If we used a gated *D* latch, the input would take effect immediately and overwrite the value in the storage element, instead of waiting for the start of the next cycle.

To prevent that from happening, a simple logic circuit for implementing the storage element is the *master-slave flip-flop*. A master-slave flip-flop can be constructed out of two gated *D* latches, as shown in Figure 3.32c. During the first half of the clock cycle, it is not possible to change the value stored in latch A. Thus, whatever is in latch A is passed to latch B, which is an internal input to the combinational logic circuit. During the second half of the clock cycle, it is not possible to change the value stored in latch B, so the value present during the first half of the clock cycle remains in latch B as the input to the combinational logic circuit for the entire cycle. However, during the second half of the clock cycle, it is possible to change the value stored in latch A. Thus the master-slave flip-flop allows the current state to remain intact for the entire cycle, while the next state is produced by the combinational logic to change latch A during the second half of the cycle so as to be ready to change latch B at the start of the next cycle.

## 3.7 The Data Path of the LC-3

In Chapter 5, we will specify a computer, which we call the LC-3, and you will have the opportunity to write computer programs to execute on it. We close out this chapter with Figure 3.33, which shows a block diagram of what we call the *data path* of the LC-3 and the finite state machine that controls all the LC-3 actions. The data path consists of all the logic structures that combine to process information in the core of the computer. Right now, Figure 3.33 is undoubtedly more than a little intimidating, and you should not be concerned by that. You are not ready to analyze it yet. That will come in Chapter 5. We have included it here, however, only to show you that you are already familiar with many of the basic structures that make up a computer. That is, you already know how most of the