

as binary information. In Chapter 10, Section 10.5, we discussed how transistors are used to perform logic operations much faster and less expensively than could be accomplished using electromechanical switches and relays.

From our discussions in the chapters mentioned above, we can state the following two underlying principles of computing:

### PHYSICAL PRINCIPLES OF COMPUTING (I, II)

---



---

- (i) Computers require that physical devices be used for representing and storing logic bit values. Each object can be in one of two states, or configurations, representing logic values 1 or 0.
  - (ii) Computers require physical switches (usually electronic) for performing logic operations.
- 
- 

These imply that the ultimate limits of computing machinery are subject to the laws of physics. Computing does not exist in an abstract, purely mathematical realm. It exists in the physical world.

In this chapter, we will see that in order to make electric circuits that can store bit values (i.e., act as memory) we need to consider logic circuits that are somewhat different from those we studied in Chapter 6. We will also see that the physical devices we use for short-term memory are different from those we use for long-term memory. We will discuss the hierarchy of memory types used in computers and see how this leads to efficient use of hardware. Finally, we will discuss how the components of a computer are organized, and how information is processed through these components.

## 11.2 SEQUENTIAL LOGIC FOR COMPUTER MEMORY

In this text we will not delve deeply into how logic instructions are carried out in a computer. Rather, we will focus on the physics underlying some of the important components. Especially important is implementation of computer memory. What is memory? In your brain, memory is a physical record of some earlier event. How can we build electronic circuits that have this property? We will see that the underlying physics of each type of memory technology determines whether it is best suited for high-speed, temporary memory or for low-speed, permanent memory, or for something in between.

Switches and logic gates are likely candidates for building memory, but there is a catch. The circuits that we studied in Chapter 6, called *combinational logic circuits*, cannot function as memory devices. This is because their output values are determined completely by the present values of their inputs. They are not influenced by the values that the inputs had previously. Recall the rules for combinational logic circuits:

- Rule 1. It is forbidden to combine two wires into one wire.
- Rule 2. It is allowed to split a wire into two or more wires.
- Rule 3. An output wire from one gate can be used as an input to a different gate.
- Rule 4. No output of a gate can eventually feed back into the input of the same gate.

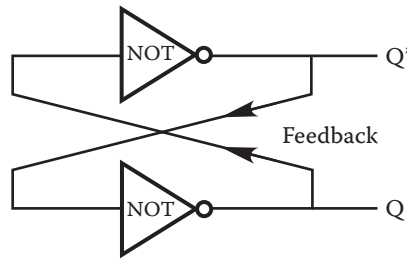
To make *memory circuits*, we must change rule 4; that is, we must allow the output of a gate to feed back into the input of the same gate. This creates a self-reinforcing situation, which continually reminds the gate of what its input was at an earlier time. It puts the gate into a kind of logic loop that it has a hard time getting out of. (It is a little like when you cannot get an irritating song out of your head.) Logic circuits that have feedback are called *sequential logic*, because their outputs depend on the sequence in which the inputs change from one value to another.

Rule 4' for sequential logic: By permitting a logic gate's output to feed back into its input, a logic circuit can be designed to store information. Consider some examples of sequential logic:

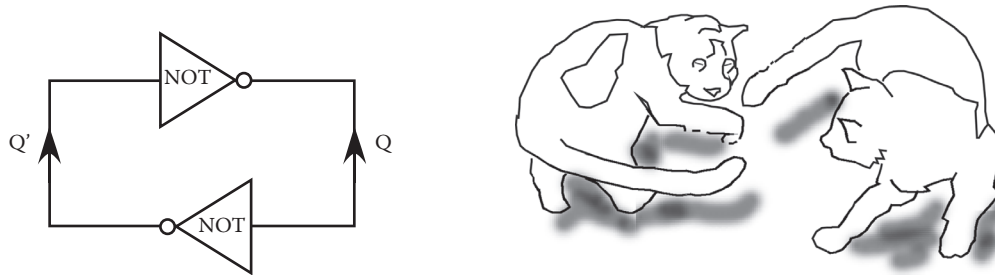
**Feedback Example #1: NOT Loop**

**Figure 11.1** shows two NOT gates wired into a feedback loop. The lower output  $Q$  is "fed back" to the input of the upper NOT gate, and vice versa. This *feedback* circuit has two stable states (configurations). It is stable if  $Q = 1$  and  $Q' = 0$ . It is also stable if  $Q = 0$  and  $Q' = 1$ . If we could find a way to cause the circuit to switch between these two stable configurations, we could use it as a memory to store a single bit value. The problem with this circuit is that it has no inputs, so it cannot be switched.

We can draw the NOT feedback loop in a more suggestive way, as in **Figure 11.2**. This might remind you of two cats chasing each others' tail.



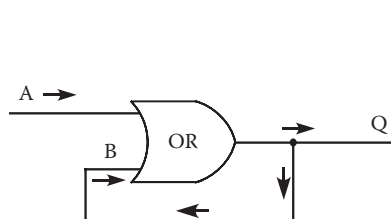
**FIGURE 11.1** NOT loop feedback circuit.



**FIGURE 11.2** NOT loop feedback circuit as mutual tail chasing.

**Feedback Example #2: One-Time Latch**

Next, consider the circuit in **Figure 11.3**. The output  $Q$  is fed back to the input  $B$ , with the arrows showing the direction of travel of the bit values. The other input,  $A$ , can equal 0 or 1. What are the possible stable configurations of inputs and outputs? Suppose



Sequential Logic History			
Event Order	A	B	Q
1	0	0	0
2	1	1	1
3	0	1	1
4	1	1	1

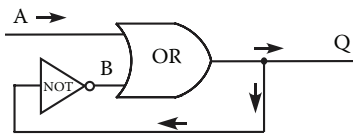
**FIGURE 11.3** Feedback circuit, forming a one-time latch. After the input  $A$  is set to 1, flipping  $Q$  to 1, it cannot be reset to  $Q = 0$ .

that  $A = 0$  and  $B = 0$ . Then the output  $Q$  is 0, and that is consistent with  $B = 0$ . So, this is a possible stable configuration.

Assume that initially the circuit is in this configuration:  $A = B = 0$ , with  $Q = 0$ . In the sequential logic history shown, this top row is called “event 1.” The sequential logic table is similar to a story (a history). It should be read starting at the top row, and progressing down one row at a time (as frames on a movie film strip). If at some later time (event 2) the value of  $A$  is changed to 1, then  $Q$  goes to 1, causing  $B$  to equal 1 as well. We say the  $Q$  value has been set. This is also a stable configuration. In fact, it is so stable that even if, at a later time (event 3),  $A$  is changed back to 0, the  $Q$  value remains at 1 (and so does  $B$ ). The value of  $Q$  is locked and cannot be changed after this. This latch is not resettable. It is called a *one-time latch* because it can operate only once. Notice that  $A = 1, B = 0$  is a configuration that can never be reached. This circuit certainly has memory. It “knows” whether or not it has been set. It would be nice, though, to have a circuit that can be set to  $Q = 1$ , and then reset back to  $Q = 0$ .

### QUICK QUESTION 11.1

For the circuit below, the  $A$  value initially equals 1, giving  $Q = 1$ , in a stable configuration. Then you change the input value to  $A = 0$ . Explain why this does not lead to a stable configuration.



Sequential Logic History

Event Order	A	B	Q
1	1	0	1
2	0	?	?

### THINK AGAIN

Not all input values will lead to stable configurations. Consider the example in the following Quick Question.

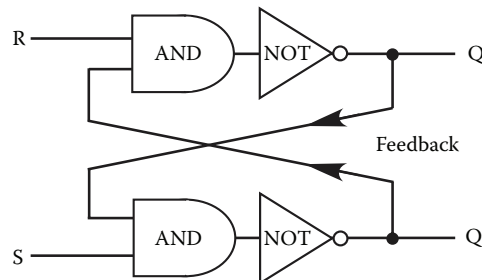
#### 11.2.1 The Set-Reset Latch

Let us design a latch that can be reset; that is, one that can be set to  $Q = 1$ , then later set back to 0, and so on. This type of latch is called a *set-reset latch*, or *S-R latch*. It can be constructed by combining two feedback circuits, as in **Figure 11.4**. The logic value  $S$  is the *set* input, the logic value  $R$  is the *reset* input, and  $Q$  and  $Q'$  are the outputs.

In such a circuit, the output values are not given uniquely by the present values of the inputs, but depend on their past history. The behavior of a S-R latch is summarized as follows. First, consider two possible cases:

- If  $R = 0$  and  $S = 1$ , then  $Q = 1$ , and  $Q' = 0$ . That is,  $Q = S$  (and  $Q' = \text{NOT } S$ ).
- If  $R = 1$  and  $S = 0$ , then  $Q = 0$ , and  $Q' = 1$ . That is  $Q = \text{NOT } S$  (and  $Q' = S$ ).

In each of these two cases, we say that  $Q$  follows  $S$ . Next, start from either of the above cases, then change  $S$  or  $R$  so that both  $R = 1$  and  $S = 1$ . Then  $Q$  holds the previous value of  $S$ . The input  $R = S = 0$  is to be avoided, because this leads, in some cases, to unstable values. (If we have  $R = S = 0$ , and then we simultaneously change both  $R$  and  $S$  to 1, then the  $Q$  and  $Q'$  values switch forever between 1 and 0.)



**FIGURE 11.4** Set-reset (S-R) latch.  $S$  is the *set* input,  $R$  is the *reset* input, and  $Q$  and  $Q'$  are the outputs.

Let us see how this behavior comes about. There are four useful logic states:

$R = 0, S = 1$	with	$Q = 1, Q' = 0$
$R = 1, S = 0$	with	$Q = 0, Q' = 1$
$R = 1, S = 1$	with	$Q = 0, Q' = 1$
$R = 1, S = 1$	with	$Q = 1, Q' = 0$

With the first row of values, the upper AND gate sees  $R = 0$  as an input, so it puts out a 0, regardless of the  $Q'$  value. The output of this AND gate is changed to 1 by the NOT gate, producing  $Q = 1$ . The  $Q$  value feeds back to the input of the lower AND gate, making its output 1, which is changed to a 0, making  $Q' = 0$ . Note that in this case  $Q = S$ . A similar argument holds for the second row of values in the list, again leading to  $Q = S$ . In these two cases,  $Q$  follows  $S$ .

With the third row of values, the upper AND gate sees two 1s as its inputs (assuming  $Q' = 1$ ), so it puts out a 1, which gets inverted to a 0 by the NOT gate, producing  $Q = 0$ . The  $Q$  value feeds back to the input of the lower AND gate, making its output 0, which gets changed to 1, making  $Q' = 1$ . The  $Q'$  value feeds back to the input of the upper AND gate, and we see that all is consistent. A similar argument holds for the fourth row of values in the list.

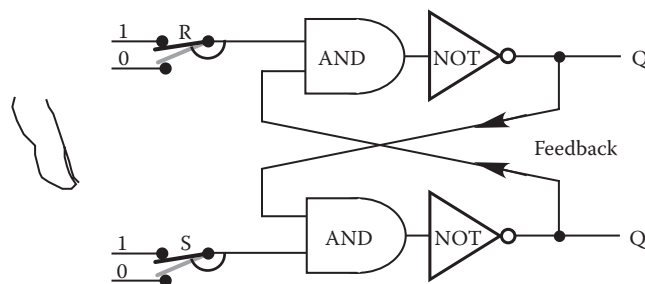
There is an interesting property of the third and fourth rows. They have the same inputs, but different outputs! This is clearly not a combinational logic circuit, whose outputs depend only on the present values of the inputs. The S-R latch circuit depends on prior history. For example, say the configuration is presently as in the first row, with  $R = 0, S = 1, Q = 1$ . Now change  $R$  to 1, so the configuration is as in the fourth row (not the third row!).  $Q$  has not changed. Its value is held to its previous value.

This circuit works as memory, but there is a potential problem. If both  $R$  and  $S$  equal 0, then the circuit would give  $Q = Q' = 0$ , and erase the earlier history. Worse yet, if we have  $R = S = 0$ , and then we simultaneously change both  $R$  and  $S$  to 1, then the  $Q$  and  $Q'$  values switch erratically. We need a way to avoid the  $R = S = 0$  state.

The  $R = S = 0$  configuration can be avoided by placing a push-off switch at each input, with a logical value 1 to the left of each switch, as in **Figure 11.5**. Only one switch can be pushed at a time (as indicated by the single finger) to create a 0 input, and when let go it pops back to the 1 value. This type of switch acts as a NOT gate. Next, we will consider how to implement this idea by using only logic gates, rather than switches and a finger.

### 11.2.2 The Enabled Data Latch, or D-Latch

The circuit in **Figure 11.6** is an *enabled data latch* (or *D-latch*, or a *D flip-flop*), which solves the problem in the S-R latch by automatically preventing the  $R = 0, S = 0$  input from occurring. The right-hand side of the circuit is again the S-R latch. But now we



**FIGURE 11.5** Set-reset (S-R) latch.  $S$  is the *set* input,  $R$  is the *reset* input, and  $Q$  and  $Q'$  are the outputs. Only one switch can be pushed open at a time.