Everybody knows that electricity makes things move. A brief glance around the average home reveals electric motors in appliances as diverse as clocks, fans, food processors, and compact disc players. Electricity also makes the cones in loudspeakers vibrate, bringing forth sounds, speech, and music from the stereo system and the television set. But perhaps the simplest and most elegant way that electricity makes things move is illustrated by a class of devices that are quickly disappearing as electronic counterparts replace them. I refer to the marvelously retro electric buzzers and bells.

Consider a relay wired this way with a switch and battery:

If this looks a little odd to you, you’re not imagining things. We haven’t seen a relay wired quite like this yet. Usually a relay is wired so that the input is
separate from the output. Here it’s all one big circle. If you close the switch, a circuit is completed:

The completed circuit causes the electromagnet to pull down the flexible contact:

But when the contact changes position, the circuit is no longer complete, so the electromagnet loses its magnetism and the flexible contact flips back up:

which, of course, completes the circuit again. What happens is this: As long as the switch is closed, the metal contact goes back and forth—alternately closing the circuit and opening it—most likely making a sound. If the contact makes a rasping sound, it’s a buzzer. If you attach a hammer to it and provide a metal gong, you’ll have the makings of an electric bell.
You can choose from a couple of ways to wire this relay to make a buzzer. Here's another way to do it using the conventional voltage and ground symbols:

![Diagram of a relay circuit]

You might recognize in this diagram the inverter from Chapter 11. The circuit can be drawn more simply this way:

![Simplified diagram]

As you'll recall, the output of an inverter is 1 if the input is 0, and 0 if the input is 1. Closing the switch on this circuit causes the relay in the inverter to alternately open and close. You can also wire the inverter without a switch to go continuously:

![Diagram of a relay circuit without a switch]

This drawing might seem to be illustrating a logical contradiction because the output of an inverter is supposed to be opposite the input, but here the output is the input! Keep in mind, however, that the inverter is actually just a relay, and the relay requires a little bit of time to change from one state to another. So even if the input is the same as the output, the output will soon change, becoming the inverse of the input (which, of course, changes the input, and so forth and so on).

What is the output of this circuit? Well, the output quickly alternates between providing a voltage and not providing a voltage. Or, we can say, *the output quickly alternates between 0 and 1*.

This circuit is called an oscillator. It's intrinsically different from everything else we've looked at so far. All the previous circuits have changed their state only with the intervention of a human being who closes or opens a switch. The oscillator, however, doesn't require a human being; it basically runs by itself.
Of course, the oscillator in isolation doesn't seem to be very useful. We'll see later in this chapter and in the next few chapters that such a circuit connected to other circuits is an essential part of automation. All computers have some kind of oscillator that makes everything else move in synchronicity. The output of the oscillator alternates between 0 and 1. A common way to symbolize that fact is with a diagram that looks like this:

```
\[\text{Diagram showing oscillations between 0 and 1.}\]
```

This is understood to be a type of graph. The horizontal axis represents time, and the vertical axis indicates whether the output is 0 or 1:

```
\[\text{Diagram with time on the x-axis and output on the y-axis.}\]
```

All this is really saying that as time passes, the output of the oscillator alternates between 0 and 1 on a regular basis. For that reason, an oscillator is sometimes often referred to as a *clock* because by counting the number of oscillations you can tell time (kind of).

How fast will the oscillator run? That is, how quickly will the metal contact of the relay vibrate back and forth? How many times a second? That obviously depends on how the relay is built. One can easily imagine a big, sturdy relay that clunks back and forth slowly and a small, light relay that buzzes rapidly.

A *cycle* of an oscillator is defined as the interval during which the output of the oscillator changes and then comes back again to where it started:

```
\[\text{Diagram showing one cycle of oscillation.}\]
```

The time required for one cycle is called the *period* of the oscillator. Let's assume that we're looking at a particular oscillator that has a period of 0.05 second. We can then label the horizontal axis in seconds beginning from some arbitrary time we denote as 0:
The frequency of the oscillator is 1 divided by the period. In this example, if the period of the oscillator is 0.05 second, the frequency of the oscillator is \(1 + 0.05\), or \(20 \text{ cycles per second}\). Twenty times per second, the output of the oscillator changes and changes back.

Cycles per second is a fairly self-explanatory term, much like miles per hour or pounds per square inch or calories per serving. But cycles per second isn't used much any more. In commemoration of Heinrich Rudolph Hertz (1857–1894), who was the first person to transmit and receive radio waves, the word hertz is now used instead. This usage started first in Germany in the 1920s and then expanded into other countries over the decades.

Thus, we can say that our oscillator has a frequency of 20 hertz, or (to abbreviate) 20 Hz.

Of course, we just guessed at the actual speed of one particular oscillator. By the end of this chapter, we'll be able to build something that lets us actually measure the speed of an oscillator.

To begin this endeavor, let's look at a pair of NOR gates wired a particular way. You'll recall that the output of a NOR gate is a voltage only if both inputs aren't voltages:

\[
\begin{array}{c|cc}
\text{NOR} & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
\end{array}
\]

Here's a circuit with two NOR gates, two switches, and a lightbulb:

Notice the oddly contorted wiring: The output of the NOR gate on the left is an input to the NOR gate on the right, and the output of the right NOR gate is an input to the left NOR gate. This is a type of feedback. Indeed, just as in the oscillator, an output circles back to become an input. This idiosyncrasy will be a characteristic of most of the circuits in this chapter.
At the outset, the only current flowing in this circuit is from the output of the left NOR gate. That's because both inputs to that gate are 0. Now close the upper switch. The output from the left NOR gate becomes 0, which means the output from the right NOR gate becomes 1 and the lightbulb goes on:

![Diagram showing the circuit with the upper switch closed]

The magic occurs when you now open the upper switch. Because the output of a NOR gate is 0 if either input is 1, the output of the left NOR gate remains the same and the light remains lit:

![Diagram showing the circuit with the upper switch open]

Now this is odd, wouldn't you say? Both switches are open—the same as in the first drawing—yet now the lightbulb is on. This situation is certainly different from anything we've seen before. Usually the output of a circuit is dependent solely upon the inputs. That doesn't seem to be the case here. Moreover, at this point you can close and open that upper switch and the light remains lit. That switch has no further effect on the circuit because the output of the left NOR gate remains 0.

Now close the lower switch. Because one of the inputs to the right NOR gate is now 1, the output becomes 0 and the lightbulb goes out. The output of the left NOR gate becomes 1:
Now you can open the bottom switch and the lightbulb stays off:

We're back where we started. At this time, you can close and open the bottom switch with no further effect on the lightbulb. In summary

- Closing the top switch causes the lightbulb to go on, and it stays on when the top switch is opened.
- Closing the bottom switch causes the lightbulb to go off, and it stays off when the bottom switch is opened.

The strangeness of this circuit is that sometimes when both switches are open the light is on, and sometimes when both switches are open, the light is off. We can say that this circuit has two stable states when both switches are open. Such a circuit is called a flip-flop, a word also used for beach sandals and the tactics of politicians. The flip-flop dates from 1918 with the work of English radio physicist William Henry Eccles (1875–1966) and F.W. Jordan (about whom not much seems to be known).

A flip-flop circuit retains information. It “remembers.” In particular, the flip-flop shown previously remembers which switch was most recently closed. If you happen to come upon such a flip-flop in your travels and you see that the light is on, you can surmise that it was the upper switch that was most recently closed; if the light is off, the lower switch was most recently closed.

A flip-flop is very much like a seesaw. A seesaw has two stable states, never staying long in that precarious middle position. You can always tell from looking at a seesaw which side was pushed down most recently.

Although it might not be apparent yet, flip-flops are essential tools. They add memory to a circuit to give it a history of what's gone on before. Imagine trying to count if you couldn't remember anything. You wouldn't know what number you were up to and what number comes next! Similarly, a circuit that counts (which I'll show you later in this chapter) needs flip-flops.

There are a couple of different types of flip-flops. What I've just shown is the simplest and is called an R-S (or Reset-Set) flip-flop. The two NOR gates are more commonly drawn and labeled as in the diagram at the top of the next page to give it a symmetrical look.
Chapter Fourteen

The output that we used for the lightbulb is traditionally called Q. In addition, there's a second output called \( \overline{Q} \) (pronounced \( Q \) bar) that's the opposite of Q. If Q is 0, then \( \overline{Q} \) is 1, and vice versa. The two inputs are called S for set and R for reset. You can think of these verbs as meaning "set Q to 1" and "reset Q to 0." When S is 1 (which corresponds to closing the top switch in the earlier diagram), Q becomes 1 and \( \overline{Q} \) becomes 0. When R is 1 (corresponding to closing the bottom switch in the earlier diagram), Q becomes 0 and \( \overline{Q} \) becomes 1. When both inputs are 0, the output indicates whether Q was last set or reset. These results are summed up in the following table:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>S R</td>
<td>Q ( \overline{Q} )</td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0</td>
<td>Q ( \overline{Q} )</td>
</tr>
<tr>
<td>1 1</td>
<td>Disallowed</td>
</tr>
</tbody>
</table>

This is called a function table or a logic table or a truth table. It shows the outputs that result from particular combinations of inputs. Because there are only two inputs to the R-S flip-flop, the number of combinations of inputs is four. These correspond to the four rows of the table under the headings.

Notice the row second from the bottom when S and R are both 0: The outputs are indicated as Q and \( \overline{Q} \). This means that the Q and \( \overline{Q} \) outputs remain what they were before both the S and R inputs became 0. The final row of the table indicates that a situation in which the S and R inputs are both 1 is disallowed or illegal. This doesn't mean you'll get arrested for doing it, but if both inputs are 1 in this circuit, both outputs are 0, which violates the notion of \( \overline{Q} \) being the opposite of Q. So when you're designing circuitry that uses the R-S flip-flop, avoid situations in which the S and R inputs are both 1.

The R-S flip-flop is often drawn as a little box with the two inputs and two outputs labeled like this:
The R-S flip-flop is certainly interesting as a first example of a circuit that seems to "remember" which of two inputs was last a voltage. What turns out to be much more useful, however, is a circuit that remembers whether a particular signal was 0 or 1 at a particular point in time.

Let's think about how such a circuit should behave before we actually try to build it. It would have two inputs. Let's call one of them Data. Like all digital signals, the Data input can be 0 or 1. Let's call the other one Hold That Bit, which is the digital equivalent of a person saying "Hold that thought." Normally the Hold That Bit signal is 0, in which case the Data signal has no effect on the circuit. When Hold That Bit is 1, the circuit reflects the value of the Data signal. The Hold That Bit signal can then go back to being 0, at which time the circuit remembers the last value of the Data signal. Any changes in the Data signal have no further effect.

In other words, we want something that has the following function table:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data, Hold That Bit</td>
<td>Q</td>
</tr>
<tr>
<td>0, 1</td>
<td>0</td>
</tr>
<tr>
<td>1, 1</td>
<td>1</td>
</tr>
<tr>
<td>0, 0</td>
<td>Q</td>
</tr>
<tr>
<td>1, 0</td>
<td>Q</td>
</tr>
</tbody>
</table>

In the first two cases, when the Hold That Bit signal is 1, the output Q is the same as the Data input. In the second two cases, when the Hold That Bit signal is 0, the Q output is the same as it was before. Notice in the second two cases that when Hold That Bit is 0, the Q output is the same regardless of what the Data input is. The function table can be simplified a little, like this:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data, Hold That Bit</td>
<td>Q</td>
</tr>
<tr>
<td>0, 1</td>
<td>0</td>
</tr>
<tr>
<td>1, 1</td>
<td>1</td>
</tr>
<tr>
<td>X, 0</td>
<td>Q</td>
</tr>
</tbody>
</table>

The X means "don't care." It doesn't matter what the Data input is because if the Hold That Bit input is 0, the output Q is the same as it was before.

Implementing a Hold That Bit signal based on our existing R-S flip-flop requires that we add two AND gates at the input end, as in the diagram at the top of the following page.
Recall that the output of an AND gate is 1 only if both inputs are 1. In this diagram, the Q output is 0 and the $\overline{Q}$ output is 1.

As long as the Hold That Bit signal is 0, the Set signal has no effect on the outputs:

Similarly, the Reset signal has no effect:

Only when the Hold That Bit signal is 1 will this circuit function the same way as the normal R-S flip-flop shown earlier:
It behaves like a normal R-S flip-flop because now the output of the upper AND gate is the same as the Reset signal, and the output of the lower AND gate is the same as the Set signal.

But we haven’t yet achieved our goal. We want only two inputs, not three. How is this done? If you recall the original function table of the R-S flip-flop, the case in which Set and Reset were both 1 was disallowed, so we want to avoid that. And it doesn’t make much sense for the Set and Reset signals to now both be 0 because that’s simply the case in which the output didn’t change. We can accomplish the same thing in this circuit by setting Hold That Bit to 0.

So it makes sense that if Set is 1, Reset is 0; and if Set is 0, Reset is 1. A signal called Data can be equivalent to a Set, and the Data signal inverted can be the Reset signal:

![Diagram of flip-flop circuit with Hold That Bit set to 0.](image)

In this case, both inputs are 0 and the output $Q$ is 0 (which means that $\overline{Q}$ is 1). As long as Hold That Bit is 0, the Data input has no effect on the circuit:

![Diagram of flip-flop circuit with Hold That Bit set to 0 and Data input.](image)

But when Hold That Bit is 1, the circuit reflects the value of the Data input:

![Diagram of flip-flop circuit with Hold That Bit set to 1 and Data input.](image)
The Q output is now the same as the Data input, and \( \overline{Q} \) is the opposite. Now Hold That Bit can go back to being 0:

![Circuit Diagram 1]

The circuit now remembers the value of Data when Hold That Bit was last 1, regardless of how Data changes. The Data signal could, for example, go back to 0 with no effect on the output:

![Circuit Diagram 2]

This circuit is called a *level-triggered D-type flip-flop*. The D stands for Data. Level-triggered means that the flip-flop saves the value of the Data input when the Hold That Bit input is at a particular level, in this case 1. (We'll look at an alternative to level-triggered flip-flops shortly.)

Usually when a circuit like this appears in a book, the input isn't labeled Hold That Bit. It's usually labeled Clock. Of course, this signal isn't a real clock, but it might sometimes have clocklike attributes, which means that it might tick back and forth between 0 and 1 on a regular basis. But for now, the Clock input simply indicates when the Data input is to be saved:

![Circuit Diagram 3]
And usually when the function table is shown, Data is abbreviated as $D$ and Clock is abbreviated as $Clk$:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Q</td>
</tr>
<tr>
<td>Clk</td>
<td>Q-bar</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>Q</td>
</tr>
</tbody>
</table>

This circuit is also known as a level-triggered D-type latch, and that term simply means that the circuit latches onto one bit of data and keeps it around for further use. The circuit can also be referred to as a 1-bit memory. I’ll demonstrate in Chapter 16 how very many of these flip-flops can be wired together to provide many bits of memory.

Saving a multibit value in latches is often useful. Suppose you want to use the adding machine in Chapter 12 to add three 8-bit numbers together. You’d key in the first number on the first set of switches and the second number on the second set of switches as usual, but then you’d have to write down the result. You’d then have to key in that result on one set of switches and key in the third number on the other set of switches. You really shouldn’t have to key in an intermediate result. You should be able to use it directly from the first calculation.

Let’s solve this problem using latches. Let’s assemble eight latches in a box. Each of the eight latches uses two NOR gates and two AND gates and one inverter, as shown previously. The Clock inputs are all connected. Here’s the resultant package:

This latch is capable of saving 8 bits at once. The eight inputs on the top are labeled $D_0$ through $D_7$, and the eight outputs on the bottom are labeled $Q_0$ through $Q_7$. The input at the left is the Clock. The Clock signal is normally 0. When the Clock signal is 1, the 8-bit value on the D inputs is transferred to the Q outputs. When the Clock signal goes back to 0, that 8-bit value stays there until the next time the Clock signal is 1.

The 8-Bit Latch can also be drawn with the eight Data inputs and eight Q outputs grouped together as you see on the following page.