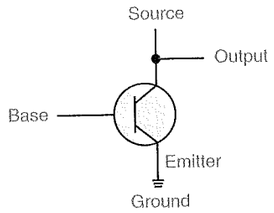



FIGURE 4.8 The connections of a transistor

Before tackling the details of transistors, let's discuss some basic principles of electricity. An electrical signal has a source, such as a battery or an outlet in your wall. If the electrical signal is *grounded*, it is allowed to flow through an alternative route to the ground (literally), where it can do no harm. A grounded electrical signal is pulled down, or reduced, to 0 volts.

A transistor has three terminals: a source, a base, and an emitter. The emitter is typically connected to a ground wire, as shown in Figure 4.8. For computers, the source produces a high voltage value, approximately 5 volts. The base value regulates a gate that determines whether the connection between the source and ground is made. If the source signal is grounded, it is pulled down to 0 volts. If the base does not ground the source signal, it stays high.

An output line is usually connected to the source line. If the source signal is pulled to the ground by the transistor, the output signal is low, representing a binary 0. If the source signal remains high, so does the output signal, representing a binary 1.

The transistor is either on, producing a high output signal, or off, producing a low output signal. This output is determined by the base electrical signal. If the base signal is high (close to a +5 voltage), the source signal is grounded, which turns the transistor off. If the base signal is low (close to a 0 voltage), the source signal stays high, and the transistor is on.

Now let's see how a transistor is used to create various types of gates. It turns out that, because of the way a transistor works, the easiest gates to create are the NOT, NAND, and NOR gates. Figure 4.9 shows how these gates can be constructed using transistors.

The diagram for the NOT gate is essentially the same as our original transistor diagram. It takes only one transistor to create a NOT gate. The signal $V_{in}$ represents the input signal to the NOT gate. If it is high, the source is grounded and the output signal $V_{out}$ is low. If $V_{in}$ is low, the source is not grounded and $V_{out}$ is high. Thus the input signal is inverted, which is exactly what a NOT gate does.

The NAND gate requires two transistors. The input signals $V_1$ and $V_2$ represent the input to the NAND gate. If both input signals are high, the source is grounded and the output $V_{out}$ is low. If either input signal is low,



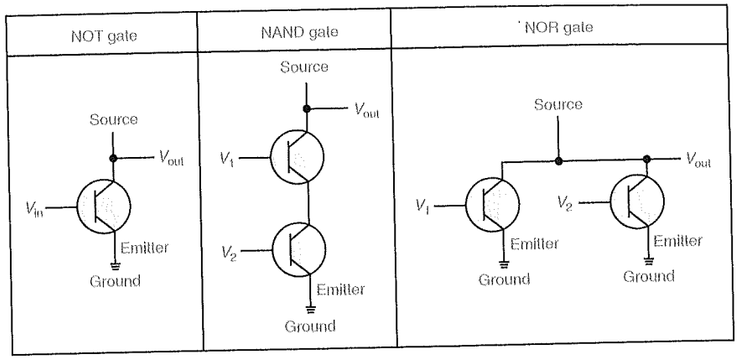



FIGURE 4.9 Constructing gates using transistors

however, one transistor or the other keeps the source signal from being grounded and the output is high. Therefore, if $V_1$ or $V_2$ or both carry a low signal (binary 0), the output is a 1. This is consistent with the processing of a NAND gate.

The construction of a NOR gate also requires two transistors. Once again, $V_1$ and $V_2$ represent the input to the gate. This time, however, the transistors are not connected in series. The source connects to each transistor separately. If either transistor allows the source signal to be grounded, the output is 0. Therefore, the output is high (binary 1) only when both $V_1$ and $V_2$ are low (binary 0), which is what we want for a NOR gate.

An AND gate produces output that is exactly opposite of the NAND output of a gate. Therefore, to construct an AND gate, we simply pass the output of a NAND gate through an inverter (a NOT gate). That's why AND gates are more complicated to construct than NAND gates: They require three transistors, two for the NAND and one for the NOT. The same reasoning can be applied to understand the relationship between NOR and OR gates.

## 4.4 Circuits

Now that we know how individual gates work and how they are constructed, let's examine how we combine gates to form circuits. Circuits can be classified into two general categories. In a **combinational circuit**, the input values explicitly determine the output. In a **sequential circuit**, the output is a function of the input values as well as the existing state of the circuit. Thus sequential circuits usually involve the storage of information. Most of the circuits we examine in this chapter are combinational circuits, although we briefly mention sequential memory circuits.

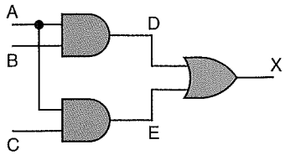
**Combinational circuit** A circuit whose output is solely determined by its input values

**Sequential circuit** A circuit whose output is a function of its input values and the current state of the circuit

As with gates, we can describe the operations of entire circuits using three notations: Boolean expressions, logic diagrams, and truth tables. These notations are different, but equally powerful, representation techniques.

## Combinational Circuits

Gates are combined into circuits by using the output of one gate as the input for another gate. For example, consider the following logic diagram of a circuit:




The output of the two AND gates is used as the input to the OR gate. The input value A is used as input to both AND gates. The dot indicates that two lines are connected. If the intersection of two crossing lines does not have a dot, you should think of one as "jumping over" the other without affecting each other.

What does this logic diagram mean? Well, let's work backward to see what it takes to get a particular result. For the final output X to be 1, either D must be 1 or E must be 1. For D to be 1, A and B must both be 1. For E to be 1, both A and C must be 1. Both E and D may be 1, but that isn't necessary. Examine this circuit diagram carefully; make sure that this reasoning is consistent with your understanding of the types of gates used.

Now let's represent the processing of this entire circuit using a truth table:

| A | B | C | D | E | X |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

Because there are three inputs to this circuit, eight rows are required to describe all possible input combinations. Intermediate columns show the intermediate values (D and E) in the circuit.



Finally, let's express this same circuit using Boolean algebra. A circuit is a collection of interacting gates, so a Boolean expression to represent a circuit is a combination of the appropriate Boolean operations. We just have to put the operations together in the proper form to create a valid Boolean algebra expression. In this circuit, there are two AND expressions. The output of each AND operation is input to the OR operation. Thus this circuit is represented by the following Boolean expression (in which the AND operator is assumed):

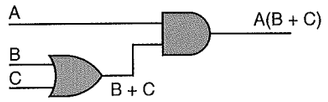
(AB + AC)

When we write truth tables, it is often better to label columns using these kinds of Boolean expressions rather than arbitrary variables such as D, E, and X. That makes it clear what each column represents. In fact, we can use Boolean expressions to label our logic diagrams as well, eliminating the need for intermediate variables altogether.

Now let's go the other way: Let's take a Boolean expression and draw the corresponding logic diagram and truth table. Consider the following Boolean expression:

A(B + C)

In this expression, the OR operation is applied to input values B and C. The result of that operation is used as input, along with A, to an AND operation, producing the final result. The corresponding circuit diagram is




Once again, we complete our series of representations by expressing this circuit as a truth table. As in the previous example, there are three input values, so there are eight rows in the truth table:

| A | B | C | B + C | A(B + C) |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Pick a row from this truth table and follow the logic of the circuit diagram to make sure the final results are consistent. Try it with a few rows to get comfortable with the process of tracing the logic of a circuit.

Now compare the final result column in this truth table to the truth table for the previous example. They are identical. We have just demonstrated circuit equivalence. That is, both circuits produce exactly the same output for each input value combination.

**Circuit equivalence** The same output for each corresponding input-value combination for two circuits

In fact, this situation demonstrates an important property of Boolean algebra called the *distributive law:*

$$A(B + C) = AB + AC$$

That's the beauty of Boolean algebra: It allows us to apply provable mathematical principles to design logical circuits. The following chart shows a few of the properties of Boolean algebra:

| Property | AND | OR |
|---|---|---|
| Commutative | AB = BA | A + B = B + A |
| Associative | (AB)C = A(BC) | (A + B) + C = A + (B + C) |
| Distributive | A(B + C) = (AB) + (AC) | A + (BC) = (A + B)(A + C) |
| Identity | A1 = A | A + 0 = A |
| Complement | A(A') = 0 | A + (A') = 1 |
| DeMorgan's law | (AB)' = A' OR B' | (A + B)' = A'B' |

DeMorgan's Law, named for Augustus DeMorgan DeMorgan, a contemporary of George Boole, was the first professor of mathematics at the University of London in 1828, where he continued to teach for 30 years. He wrote elementary texts on arithmetic, algebra, trigonometry, and calculus as well as papers on the possibility of establishing a logical calculus and the fundamental problem of expressing thought by means of symbols. DeMorgan did not discover the law bearing his name, but he is credited with formally stating it as it is known today.[3]

These properties are consistent with our understanding of gate processing as well as with the truth table and logic diagram representations. For instance, the commutative property, in plain English, says that the order of the input signals doesn't matter, which is true. (Verify it using the truth tables of individual gates.) The complement property says that if we put a signal and its inverse through an AND gate, we are guaranteed to get 0, but if we put a signal and its inverse through an OR gate, we are guaranteed to get 1.

There is one very famous—and useful—theorem in Boolean algebra called *DeMorgan's law.* This law states that the NOT operator applied to the AND of two variables is equal to the NOT applied to each of the two variables with an OR between. That is, inverting the output of an AND gate is equivalent to inverting the individual signals first and then passing them through an OR gate:

$$(AB)' = A' \text{ OR } B'$$

The second part of the law states that the NOT operator applied to the OR of two variables is equal to the NOT applied to each of the two variables with an AND between. Expressed in circuit terms, this means that inverting the output of an OR gate is equivalent to inverting both signals first and then passing them through an AND gate:

$$(A + B)' = A'B'$$

DeMorgan's law and other Boolean algebra properties provide a formal mechanism for defining, managing, and evaluating logical circuit designs.



## Adders

Perhaps the most basic operation a computer can perform is to add two numbers together. At the digital logic level, this addition is performed in binary. Chapter 2 discusses this process in depth. These types of addition operations are carried out by special circuits called, appropriately, adders.

**Adder** An electronic circuit that performs an addition operation on binary values

**Half adder** A circuit that computes the sum of two bits and produces the appropriate carry bit

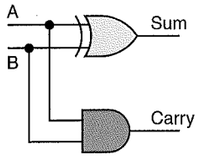
Like addition in any base, the result of adding two binary digits could potentially produce a *carry value.* Recall that 1 + 1 = 10 in base 2. A circuit that computes the sum of two bits and produces the correct carry bit is called a half adder.

Let's consider all possibilities when adding two binary digits A and B: If both A and B are 0, the sum is 0 and the carry is 0. If A is 0 and B is 1, the sum is 1 and the carry is 0. If A is 1 and B is 0, the sum is 1 and the carry is 0. If both A and B are 1, the sum is 0 and the carry is 1. This yields the following truth table:

| A | B | Sum | Carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

In this case, we are actually looking for two output results, the sum and the carry. As a consequence, our circuit has two output lines.

If you compare the sum and carry columns to the output of the various gates, you see that the sum corresponds to the XOR gate and the carry corresponds to the AND gate. Thus the following circuit diagram represents a half adder:



Test this diagram by assigning various combinations of input values and determining the two output values produced. Do the results follow the rules of binary arithmetic? They should. Now compare your results to the corresponding truth table. They should match the results there as well.

What about the Boolean expression for this circuit? Because the circuit produces two distinct output values, we represent it using two Boolean expressions:

$$\text{sum} = A \oplus B$$
$$\text{carry} = AB$$

**Full adder** A circuit that computes the sum of two bits, taking an input carry bit into account

Note that a half adder does not take into account a possible carry value *into* the calculation (carry-in). That is, a half adder is fine for adding two single digits, but it cannot be used as is to compute the sum of two binary values with multiple digits each. A circuit called a **full adder** takes the carry-in value into account.

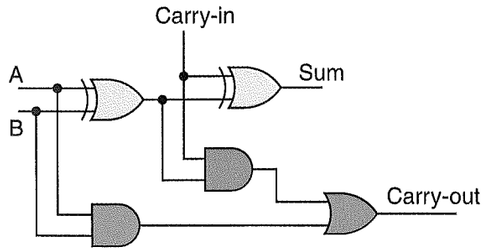
We can use two half adders to make a full adder. How? Well, the input to the sum must be the carry-in and the sum from adding the two input values. That is, we add the sum from the half adder to the carry-in. Both of these additions have a carry-out. Could both of these carry-outs be 1, yielding yet another carry? Fortunately, no. Look at the truth table for the half adder. There is no case where the sum and the carry are both 1.

Figure 4.10 shows the logic diagram and the truth table for the full adder. This circuit has three inputs: the original two digits (A and B) and the carry-in value. Thus the truth table has eight rows. We leave the corresponding Boolean expression as an exercise.

To add two 8-bit values, we can duplicate a full-adder circuit eight times. The carry-out from one place value is used as the carry-in to the next highest place value. The value of the carry-in for the rightmost bit position is assumed to be zero, and the carry-out of the leftmost bit position is discarded (potentially creating an overflow error).

There are various ways to improve on the design of these adder circuits, but we do not explore them in any more detail in this text.

**Logic Diagram**




**Truth Table**

| A | B | Carry-in | Sum | Carry-out |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

FIGURE 4.10 A full adder



## Multiplexers

A **multiplexer** (often referred to as a *mux*) is a general circuit that produces a single output signal. This output is equal to one of several input signals to the circuit. The multiplexer selects which input signal to use as an output signal based on the value represented by a few more input signals, called *select signals* or *select control lines*.

**Multiplexer** A circuit that uses a few input control signals to determine which of several input data lines is routed to its output

Let's look at an example to clarify how a multiplexer works. Figure 4.11 shows a block diagram of a mux. The control lines S0, S1, and S2 determine which of eight other input lines (D0 through D7) are routed to the output (F).

The values of the three control lines, taken together, are interpreted as a binary number, which determines which input line to route to the output. Recall from Chapter 2 that three binary digits can represent eight different values: 000, 001, 010, 011, 100, 101, 110, and 111. These values, which simply count in binary from 0 to 7, correspond to our output values D0 through D7. Thus, if S0, S1, and S2 are all 0, the input line D0 would be the output from the mux. If S0 is 1, S1 is 0, and S2 is 1, then D5 would be output from the mux.

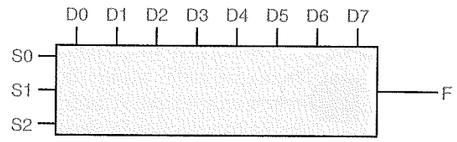
The following truth table shows how the input control lines determine the output for this multiplexer:

| S0 | S1 | S2 | F |
|---|---|---|---|
| 0 | 0 | 0 | D0 |
| 0 | 0 | 1 | D1 |
| 0 | 1 | 0 | D2 |
| 0 | 1 | 1 | D3 |
| 1 | 0 | 0 | D4 |
| 1 | 0 | 1 | D5 |
| 1 | 1 | 0 | D6 |
| 1 | 1 | 1 | D7 |

**The digital archeologist**

The work of modern archeology has been greatly assisted by digital technologies. For example, GIS (Geographic Information Systems) is used to georeference the vertical layers of an archeological site and can be used to produce a three-dimensional map. GPS (Global Positioning System)—currently a system of 24 orbiting satellites—is used to fix the geographical location of the site on the Earth. GPS data can be transferred to GIS, which makes the GIS representation of a site more powerful and precise. Archeologists, who were once reliant upon geographical survey maps, now use GPS to fix geographical points of their sites.

FIGURE 4.11 A block diagram of a multiplexer with three select control lines



The block diagram in Figure 4.11 hides a fairly complicated circuit that carries out the logic of a multiplexer. Such a circuit could be shown using eight three-input AND gates and one eight-input OR gate. We won't get into the details of this circuit in this book.

A multiplexer can be designed with various numbers of input lines and corresponding control lines. In general, the binary values on $n$ input control lines are used to determine which of $2^n$ other data lines are selected for output.

A circuit called a *demultiplexer* (*demux*) performs the opposite operation. That is, it takes a single input and routes it to one of $2^n$ outputs, depending on the values of the $n$ control lines.

## 4.5 Circuits as Memory

Digital circuits play another important role: They can store information. These circuits form a sequential circuit, because the output of the circuit also serves as input to the circuit. That is, the existing state of the circuit is used in part to determine the next state.

Many types of memory circuits have been designed. We examine only one type in this book: the *S-R latch*. An S-R latch stores a single binary digit (1 or 0). An S-R latch circuit could be designed using a variety of gates. One such circuit, using NAND gates, is pictured in Figure 4.12.

The design of this circuit guarantees that the two outputs X and Y are always complements of each other. That is, when X is 0, Y is 1, and vice versa. The value of X at any point in time is considered to be the current state of the circuit. Therefore, if X is 1, the circuit is storing a 1; if X is 0, the circuit is storing a 0.

Recall that a NAND gate produces an output of 1 unless both of its input values are 1. Each gate in this circuit has one external input (S or R) and one input coming from the output of the other gate. Suppose the current state of the circuit is storing a 1 (that is, X is 1), and suppose both S and R are 1. Then Y remains 0 and X remains 1. Now suppose that the circuit is currently storing a 0 (X is 0) and that R and S are again 1. Then Y remains 1 and X remains 0. No matter which value is currently being

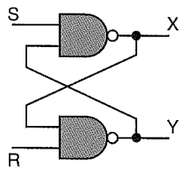



FIGURE 4.12 An S-R latch



stored, if both input values S and R are 1, the circuit keeps its existing state.

This explanation demonstrates that the S-R latch maintains its value as long as S and R are 1. But how does a value get stored in the first place? We set the S-R latch to 1 by momentarily setting S to 0 while keeping R at 1. If S is 0, X becomes 1. As long as S is returned to 1 immediately, the S-R latch remains in a state of 1. We set the latch to 0 by momentarily setting R to 0 while keeping S at 1. If R is 0, Y becomes 0, and thus X becomes 0. As long as R is immediately reset to 1, the circuit state remains 0.

By carefully controlling the values of S and R, the circuit can be made to store either value. By scaling this idea to larger circuits, we can design memory devices with larger capacities.

## 4.6 Integrated Circuits

An **integrated circuit** (also called a chip) is a piece of silicon on which multiple gates have been embedded. These silicon pieces are mounted on a plastic or ceramic package with pins along the edges that can be soldered onto circuit boards or inserted into appropriate sockets. Each pin connects to the input or output of a gate, or to power or ground.

**Integrated circuit (chip)** A piece of silicon on which multiple gates have been embedded

Integrated circuits (IC) are classified by the number of gates contained in them. These classifications also reflect the historical development of IC technology:

| Abbreviation | Name | Number of Gates |
|---|---|---|
| SSI | Small-Scale Integration | 1 to 10 |
| MSI | Medium-Scale Integration | 10 to 100 |
| LSI | Large-Scale Integration | 100 to 100,000 |
| VLSI | Very-Large-Scale Integration | more than 100,000 |

An SSI chip has a few independent gates, such as the one shown in Figure 4.13. This chip has 14 pins: eight for inputs to gates, four for output of the gates, one for ground, and one for power. Similar chips can be made with different gates.

How can a chip have more than 100,000 gates on it? That would imply the need for 300,000 pins! The key is that the gates on a VLSI chip are not independent, as they are in small-scale integration. VLSI chips embed circuits with a high gate-to-pin ratio. That is, many gates are combined to create complex circuits that require only a few input and output values. Multiplexers are an example of this type of circuit.