Basics Of Digital Logic And Data Representation

The Fundamentals From Which Computers Are Built
2

Fundamentals Of Digital Logic

2.1 Introduction

This chapter covers the basics of digital logic. The goal is straightforward — provide a background that is sufficient for a reader to understand remaining chapters. Thus, we will not need to delve into electrical details, discuss the underlying physics, or learn the design rules that engineers follow to interconnect devices. Instead, we will learn a few basics that will allow us to understand how complex digital systems work.

2.2 Electrical Terminology: Voltage And Current

Engineers use the terms voltage and current to refer to quantifiable properties of electricity: the voltage between two points (measured in volts) represents the potential force, and the current (measured in amperes or amps) represents the flow of electrons along a path (e.g., along a wire). A good analogy can be made with water: voltage corresponds to water pressure, and current corresponds to the amount of water flowing through a pipe at a given time. There is also an analogy in the relationship between water pressure and water flow and the relationship between voltage and current. If a hole appears that allows water to flow, water pressure drops; if current starts flowing through a wire, voltage drops.

The most important thing to know about electrical voltage is that voltage can only be measured as the difference between two points (i.e., the measurement is relative). Thus, a voltmeter, which is used to measure voltage, always has two probes; the meter
does not register a voltage until both probes have been connected. To simplify measurement, we assume one of the two points represents zero volts, and express the voltage of the second point relative to zero. Electrical engineers use the term ground to refer to the point that is assumed to be at zero volts. In all digital circuits shown in this text, for example, we will assume that electrical power is supplied by two wires: one wire is a ground wire, which is assumed to be at zero volts, and a second wire is at five volts.

Fortunately, we can understand the essentials of digital logic without knowing more about voltage and current. We only need to understand how electrical flow can be controlled and how electricity can be used to represent digital values.

### 2.3 The Transistor

The mechanism used to control flow of electrical current is a semiconductor device that is known as a transistor†. At the lowest level, all digital systems are composed of transistors.

Each individual transistor functions like a miniature switch that can be operated electrically. A transistor has three connections: two through which a large current can flow and one for a small current that controls the flow. When a small, positive current flows through the control connection, a large current can flow through the other two connections; when the small current stops flowing through the control connection, the large current also stops flowing. Figure 2.1 shows the diagram engineers use to denote a transistor‡:

![Figure 2.1 A transistor. When a small current flows between points B and E, a large current can flow between points C and E.](image)

It may seem that a single transistor — a device that controls electrical current — is irrelevant to digital circuits and computer architecture. We will see, however, how transistors can be used to build more complex components that are used to build digital systems.

---

†Although other semiconductors can be used, most transistors are made from silicon.
‡Technically, the diagram shows an NPN transistor because the emitter and collector (labeled E and C) are made from N-type silicon and the base (labeled B) is made from P-type silicon.
2.4 Logic Gates

How are digital circuits built? The answer lies in Boolean algebra. Programmers are familiar with the three basic Boolean functions: \( \textit{and} \), \( \textit{or} \), and \( \textit{not} \). Figure 2.2 lists the possible input values and the result of each function.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A and B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A or B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>not A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2.2 Boolean functions and the result for each possible set of inputs. A logical value of zero represents \textit{false}, and a logical value of one represents \textit{true}.

Boolean functions are used in building digital hardware. More important, it is possible to use transistors to construct circuits that implement each of the Boolean functions. Thus, Boolean functions can be translated directly into hardware.

To understand the relationship between Boolean functions and hardware, consider the Boolean \( \textit{not} \). If we use five volts to represent a Boolean \( 1 \) and zero volts to represent a Boolean \( 0 \), a single transistor plus a component known as a resistor can implement the Boolean \( \textit{not} \). That is, the output is the opposite of the input — when five volts is placed on the input, the output drops to zero volts; when zero volts is placed on the input, the output rises to five volts\(^\dagger\). Figure 2.3 illustrates a circuit that implements Boolean \( \textit{not} \).

\[\text{Figure 2.3} \text{ A transistor and a resistor used to implement the Boolean function } \textit{not}. \text{ When the input is zero volts, the output is five volts, and vice versa.}\]

\(\dagger\)In practice, the resistor limits the amount of current that can flow, which means the two values for voltage are not exactly zero and five. However, the voltages are close enough that we can think of them as exact.
To understand how the circuit operates, imagine the transistor to be a switch. When it is turned on, the transistor connects the output to zero volts; when it is turned off, the transistor disconnects the output from zero volts, and the output registers five volts. An input of five volts causes the transistor to turn on, and an input of zero volts causes the transistor to turn off. Thus, the output is always the opposite of the input.

Boolean circuits are fundamental to digital systems, and are given the name logic gates. Engineers do not construct gates from individual transistors because manufacturers sell electronic parts (actually integrated circuits) that contain all the circuitry for a gate.

A detail adds a minor complication: because of the way electronic circuits work, it takes fewer transistors to provide the inverse of Boolean functions. Thus, electronic parts that implement logic gates provide the inverse of and or: \textit{nand} (which stands for \textit{not and}) and \textit{nor} (which stands for \textit{not or}). Figure 2.4 shows how truth tables can be used to list the functions that logic gates provide.

\begin{center}
\begin{tabular}{ccc}
\hline
A & B & \text{A nand B} \\
\hline
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ccc}
\hline
A & B & \text{A nor B} \\
\hline
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 0 \\
\hline
\end{tabular}
\end{center}

\textbf{Figure 2.4} The \textit{nand} and \textit{nor} functions implemented by logic gates. Using the inverse simplifies the circuitry required.

\section*{2.5 Symbols Used For Gates}

When they design circuits, engineers do not think about individual transistors. Instead, they represent each gate by a symbol, and draw circuits by interconnecting gates. Figure 2.5 shows the symbols used for the three basic Boolean functions that hardware provides. Engineers use the term \textit{inverter} for a gate that performs the Boolean \textit{not} operation.

\begin{center}
\textbf{nand gate} \hspace{1cm} \textbf{nor gate} \hspace{1cm} \textbf{inverter}
\end{center}

\textbf{Figure 2.5} The symbols for \textit{nand}, \textit{nor}, and \textit{inverter} gates. Inputs are shown on the left, and the output is shown on the right.
2.6 Construction Of Gates From Transistors

For our purposes, the internal details of gates are unimportant. All we need to understand is how gates are used. However, it is interesting to see that transistors can be used to create a gate. Figure 2.6 provides an example by showing the internal structure of a nor gate composed of transistors, resistors, and components known as diodes. The diagram reveals the underlying complexity: six transistors, five resistors, and three diodes are needed to form a single nor gate.

![Nor Gate Diagram](image)

**Figure 2.6** The internal structure of a nor gate formed from transistors and other components. A solid dot indicates an electrical connection between two wires. Resistors are labeled with a value in ohms, with $k$ indicating multiplication by 1000.

The drawing in the figure is known as a schematic diagram. Each line on a schematic corresponds to a wire that connects one component to another. In addition, the schematic shows wires that correspond to two inputs, an output, power (five volts), and ground (zero volts).

The diagram in Figure 2.6 uses a common convention: two lines that cross do not indicate an electrical connection unless a solid dot appears. That is, two lines that cross without a dot correspond to a situation in which there is no physical connection; we can imagine that the wires are positioned so an air gap exists between them (i.e., the wires do not touch).
Now that we have seen an example of how a gate can be created out of transistors, we do not need to consider individual transistors again. Throughout the rest of the chapter, we will discuss gates without referring to their internal mechanisms; later chapters discuss larger, more complex mechanisms that are composed of gates.

### 2.7 Example Interconnection Of Gates

The electronic parts that implement gates are classified as Transistor-Transistor Logic (TTL) because the output transistors in each gate are designed to connect directly to input transistors in other gates. In fact, an output can connect to several inputs†. For example, suppose a circuit is needed in which the output is true if a disk is spinning and the user presses a power-down button. Logically, the output is a Boolean and of two inputs, but none of the gates described above provides and. However, the and function can be created by directly connecting the output of a nand gate to the input of an inverter. Figure 2.7 illustrates the connection.

![Figure 2.7 Illustration of gate interconnection. The output from one logic gate can connect directly to the inputs of other gates.](image)

As another example, consider the circuit in Figure 2.8 that shows three inputs. What function does the circuit in the figure implement? There are two ways to answer the question: we can determine the Boolean formula to which the circuit corresponds, or we can enumerate the value that appears on each wire for all eight possible combinations of input values.

To derive a Boolean formula, observe that input $Y$ is connected directly to an inverter. Thus, the value on wire $A$ corresponds to the Boolean function not $Y$. The nor gate takes inputs not $Y$ (from the inverter) and $Z$, so the value on wire $B$ corresponds to the Boolean function:

$$Z \text{ nor } (\text{not } Y)$$

---

†The technology limits the number of inputs that can be supplied from a single output; we use the term fanout to specify the number of inputs to which an output connects.
Finally, from Figure 2.7, we know that the combination of a nand gate followed by an inverter produces the Boolean and of the two inputs. Thus, the output value corresponds to:

\[ X \text{ and } (Z \text{ nor } \neg Y) \]

which can also be expressed:

\[ X \text{ and } \neg (Z \text{ or } \neg Y) \]  \hspace{1cm} (2.1)

In practice, engineers spend more time constructing new circuits than analyzing existing circuits. The equivalence between Boolean expressions and digital logic circuits is also used in design. An engineer can start by finding a Boolean expression that solves the problem, and then translate the expression into equivalent hardware. More important, tools are available that optimize Boolean expressions. That is, an engineer can create a Boolean expression that specifies the behavior of a circuit, and then use a tool that automatically transforms the expression into an equivalent expression that requires fewer gates.

A second technique used to understand a logic circuit consists of enumerating all possible inputs, and then finding the corresponding values at each point in the circuit. For example, because the circuit in Figure 2.8 has three inputs, eight possible combinations of input exist. The table in Figure 2.9 lists the input combinations on wires X, Y, and Z along with the resulting values on the wires labeled A, B, C, and output.
A truth table that enumerates values for three inputs for the circuit in Figure 2.8, and the resulting values at various points in the circuit.

To make the output easier to understand, it can be converted to conventional Boolean operations. The resulting Boolean expression is equivalent to both the truth table in the figure and expression (2.1) above†:

\[ X \text{ and } Y \text{ and } (\neg Z) \]

2.8 Multiple Gates Per Integrated Circuit

The table in Figure 2.9 is generated by starting with the eight possible values in columns X, Y, and Z, and then filling in the remaining columns one at a time. For example, point A in the circuit represents the output from the first inverter, which is the inverse of input Y. Thus, column A can be filled in by reversing the values in column Y. Similarly, column B represents the nor of columns A and Z.

Because the logic gates described above do not require many transistors, multiple gates that use TTL can be manufactured on a single, inexpensive electronic component. One popular set of TTL components that implement logic gates is known as the 7400 family‡; each component in the family is assigned a part number that begins with 74. Physically, many of the parts in the 7400 family consist of a rectangular package approximately one-half inch long with fourteen copper wires (called pins) that are used to connect the part to a circuit. Part number 7400 contains four nand gates, part number 7402 contains four nor gates, and part number 7404 contains six inverters. Figure 2.10 illustrates how the inputs and outputs of individual logic gates connect to pins in each case.

†Both Boolean equations and truth tables are useful. Boolean equations tend to be used during design, and truth tables tend to be used when debugging circuits.

‡In addition to the logic gates described in this section, the 7400 family also includes more sophisticated mechanisms, such as flip-flops, counters, and demultiplexors, that are described later in the chapter.
2.9 The Need For More Than Combinatorial Circuits

The circuits described above are classified as *combinatorial* because the output is a Boolean combination of input values. In a combinatorial circuit, the output only changes when an input value changes. Although combinatorial circuits are essential, they are not sufficient — computation requires circuits that can take action without waiting for inputs to change. For example, when a user presses a button to power on a computer, hardware must perform a sequence of operations, and the sequence must proceed without further input from the user. The hardware does not require a user to hold the power button continuously — the startup sequence must continue even after the user releases the button. Furthermore, pressing the same button again causes the hardware to initiate a shutdown sequence.

How can digital logic perform a sequence of operations without requiring the input values to change? How can a digital circuit continue to operate after an input reverts to its initial condition? The answers involve additional mechanisms. The first case is handled by circuits that are more sophisticated than Boolean logic gates, and the second case is handled by a *clock*. The next sections present examples of sophisticated circuits, and later sections explain clocks.

2.10 Circuits That Maintain State

In addition to Boolean gates, electronic parts are available that maintain *state*. Such a part responds to the history of inputs, not just the current input values. The most trivial state maintaining mechanism is a *flip-flop*. One form of flip-flop acts exactly like the power switch on a computer: the first time its input becomes *I*, the flip-flop turns on the output, and the second time its input becomes *I*, the flip-flop turns off...
the output. That is, receiving an input of 1 causes the flip-flop to change the output from the current state to the opposite. Like a push-button switch used to control power, a flip-flop does not respond to a continuous input — the input must return to 0 before a value of 1 will cause the flip-flop to change state. Figure 2.11 shows a sequence of inputs and the resulting output.

![Flip-flop diagram](image)

**Figure 2.11** Illustration of how one type of flip-flop reacts to a sequence of inputs. The flip-flop output changes when the input transitions from 0 to 1 (i.e., from zero volts to five volts).

### 2.11 Transition Diagrams

To understand how a flip-flop works, it is helpful to plot the input and output in graphical form as a function of time. Engineers use the term *transition diagram* for such a plot. Figure 2.12 illustrates a transition diagram for the flip-flop values from Figure 2.11.

![Transition diagram](image)

**Figure 2.12** Illustration of a transition diagram that shows how a flip-flop reacts to the series of inputs in Figure 2.11. Marks along the x-axis indicate times; each corresponds to one bit.
The transition diagram shows that for the example flip-flop, the output only changes when the input transitions from zero to one. Engineers say that the output transition occurs on the *leading edge* of the input change; circuits that transition when the input changes from one to zero are said to occur on the *falling edge*.

In practice, additional details complicate flip-flops. For example, most flip-flops include an additional input named *reset* that places the output in a 0 state. In addition, several variants of flip-flops exist. For example, some flip-flops provide a second output that is the inverse of the main output (in some circuits, having the inverse available results in fewer gates).

### 2.12 Binary Counters

A single flip-flop only offers two possible output values: 0 or 1. An alternative mechanism, called a *counter* accumulates a numeric total. Like a flip-flop, a counter's output changes whenever the input transitions from 0 to 1. Unlike a flip-flop, however, a counter has multiple outputs that represent the total transition count in binary. Figure 2.13 illustrates how a counter with three outputs responds to input changes.

Like our description of a flip-flop, our description of a binary counter lacks several details. For example, counters have an additional input used to reset the count to zero.

---

†The next chapter considers data representation in more detail; for now it is sufficient to understand that the outputs represent a number.
Because it has a fixed number of output pins, a counter has a maximum value it can represent. When the accumulated count exceeds the maximum value, the counter resets the output to zero, and has an additional output that is used to indicate that an overflow occurred.

### 2.13 Clocks And Sequences

We said that a mechanism known as a *clock* allows hardware to operate without requiring the input to change. In fact, most digital logic circuits are said to be *clocked*, which means that a clock, rather than a set of inputs, controls and synchronizes the operation of individual components and subassemblies to ensure that they work together as intended.

What is a clock? In terms of digital circuits, we can imagine that a clock is a mechanism that emits an alternating sequence of 0 and 1 values at a regular rate. The speed of a clock is measured in *Hertz* (Hz) (the number of times per second the clock cycles through a 1 followed by a 0). Most clocks in high-speed digital computers operate at a speed of one hundred megahertz (100 MHz) or several gigahertz (GHz). For example, at present, the clock in a high-speed PC operates at 3 GHz.

It is difficult for a human to imagine clocks that operate at such high rates. To make the concept clear, let’s assume a clock is available that operates at an extremely slow rate of 1 Hz. Such a clock might be used to control an interface for a human. For example, if a computer contains an LED that flashes on and off to indicate that the computer is active, a slow clock is needed to control the LED. Note that a clock rate of 1 Hz means the clock completes an entire cycle in one second. That is, the clock emits a logical 1 for one-half cycle followed by a logical zero for one-half cycle. If a circuit arranges to turn on an LED whenever the clock emits a logical 1, the LED will remain on for one-half second, and then will be off for one-half second.

How does an alternating sequence of 0 and 1 make digital circuits more powerful? To understand, we will consider a simple clocked circuit. Suppose that during startup, a computer must perform the following sequence of steps:

- Test the battery
- Power on and test the memory
- Start the disk spinning
- Power up the CRT
- Read the boot sector from disk into memory
- Start the CPU
Furthermore, to simplify the explanation, we will assume that one second must pass after starting a step before the circuit starts the next step. Thus, we desire a circuit that, once it has been started, will perform the six steps in sequence, at one-second intervals with no further changes in input.

For now, we will focus on the essence of the circuit, and consider how it can be started later. A circuit to handle the task of performing six steps in sequence can be built from three building blocks: a clock, a binary counter, and a device known as a demultiplexor, often abbreviated demux. We have already considered a counter, and we will assume that a clock is available that generates digital output at a rate of exactly one cycle per second. The last component, a demultiplexor, is a single integrated circuit that maps between a binary value and a set of outputs. That is, a demultiplexor takes a binary value as input, and uses the value to choose an output. Only one output of a demultiplexor is on at any time; all others are off — when the input lines represent the value \(i\) in binary, the demultiplexor selects the \(i^{th}\) output. Figure 2.14 illustrates the concept.

![Diagram of a demultiplexor](image)

**Figure 2.14** Illustration of a demultiplexor with three input lines and eight output lines. When inputs \(x\), \(y\), and \(z\) have the values 0, 1, and 1, the fourth output from the top is selected.

A demultiplexor provides the last piece needed for our simplistic sequencing mechanism — if we combine a clock, counter, and demultiplexor, the resulting circuit can execute a series of steps. For example, Figure 2.15 shows the interconnection in which the output of a clock is used as input to a binary counter, and the output of a binary counter is used as input to a demultiplexor.
Figure 2.15 An illustration of how a clock can be used to create a circuit that performs a sequence of six steps. Output lines from the counter connect directly to input lines of the demultiplexor.

To understand how the circuit operates, assume that the counter has been reset to zero. Because the counter output is 000, the demultiplexor selects the topmost output, which is not used (i.e., not connected). Operation starts when the clock changes from logical 0 to logical 1. The counter accumulates the count, which changes its output to 001. When its input changes, the demultiplexor selects the second output, which is labeled test battery. Presumably, the output wire connects to a circuit that performs the necessary test. The second output remains selected for one second. During the second, the clock output remains at logical 1 for one-half second, and then reverts to logical 0 for one-half second. When the clock output changes back to logical 1, the counter output lines change to 010, and the demultiplexor selects the third output, which is connected to circuitry that tests memory.

Of course, details are important. For example, to be compatible with other devices, the clock must use five volts for logical 1, and zero volts for logical 0. Furthermore, to be directly connected, the output lines of the binary counter must use the same binary representation as the input lines of the demultiplexor. The next chapter considers representation in more detail; for now, we assume they are compatible.

2.14 The Important Concept Of Feedback

The simplistic circuit in Figure 2.15 lacks an important feature: there is no way to control operation (i.e., to start or stop the sequence). Because a clock runs forever, the counter in the figure counts from zero through its maximum value, and then starts again at zero. As a result, the demultiplexor will repeatedly cycle through its outputs, with each output being held for one second before moving on to the next.
Sec. 2.14 The Important Concept Of Feedback

Few digital circuits perform the same series of steps repeatedly. How can we arrange to stop the sequence after the six steps have been executed? The solution lies in a fundamental concept: feedback. Feedback lies at the heart of complex digital circuits because it allows the results of processing to affect the way a circuit behaves. In the computer startup sequence, feedback is needed for each of the steps. If the disk cannot be started, for example, the boot sector cannot be read from the disk.

To see a trivial example of feedback, consider how we might use the final output, call it $F$, from the demultiplexer to stop the process. An easy way consists of using the value of $F$ to prevent clock pulses from reaching the counter. That is, instead of connecting the clock output directly to the counter input, we insert logic gates that only allow the counter pulses to continue when $F$ has the value 0. In terms of Boolean algebra, the counter input should be:

$$\text{CLOCK and (not F)}$$

That is, as long as $F$ is false, the counter input should be equal to the clock; when $F$ is true, however, the counter input changes to (and remains) zero; Figure 2.16 shows how two inverters and a nand gate can be used to implement the necessary function.

![Figure 2.16 A modification of the circuit in Figure 2.15 that includes feedback to stop processing after one pass through each output.](image)

The feedback in Figure 2.16 is fairly obvious because there is an explicit physical connection between the last output and the combinatorial circuit on the input side. The figure also makes it easy to see why feedback mechanisms are sometimes called feedback loops†.

---

†A feedback loop is also present among the gates used to construct a flip-flop.
2.15 Starting A Sequence

Figure 2.16 shows that it is possible to use feedback to terminate a process. However, the circuit is still incomplete because it does not contain a mechanism that allows the sequence to start. Fortunately, adding a starting condition is trivial. To understand why, recall that a counter contains a separate input line that resets the count to zero. All that is needed to make our circuit start running is another input (e.g., from a button that a user pushes) connected to the counter reset.

When the user pushes the button, the counter resets to zero, which causes the counter’s output to become 000. When it receives an input of all zeros, the demultiplexer turns on the first output, and turns off the last output. When the last output turns off, the nand gate allows the clock pulses through, and the counter begins to run.

Although it does indeed start the sequence, allowing a user to reset the counter can cause problems. For example, consider what happens if a user becomes impatient during the startup sequence and presses the button a second time. Once the counter resets, the sequence starts again from the beginning. In some cases, performing an operation twice simply wastes time. In other cases, however, repeating an operation causes problems (e.g., some disk drives require that only one command be issued at a time). Thus, a production system uses complex combinatorial logic to prevent a sequence from being interrupted or restarted before it completes.

2.16 Iteration In Software Vs. Replication In Hardware

One of the fundamental differences between software and hardware arises from the way software and hardware handle operations that must be applied to a set of items. In software, the fundamental paradigm for handling multiple items consists of iteration — a programmer writes code that repeatedly finds the next item in a set and applies the operation to the item. That is, because the underlying system can only apply the operation to one item at a time, a programmer must explicitly specify the number of items in the set and the order in which they are to be processed. Iteration is so essential to programming that most programming languages provide a compact syntax that allows the programmer to express the iteration clearly (e.g., a for loop).

Although hardware can be built to perform iteration, doing so is difficult, and the resulting hardware is clumsy. Instead, the fundamental hardware paradigm for handling multiple items consists of replication — a hardware engineer creates multiple copies of the underlying gates, and allows each copy to act on one item. For example, suppose we need to compute a Boolean operation on a set of thirty-two Boolean values. The ideal hardware solution consists of replicating the necessary gate thirty-two times, and allowing each instance to operate on one of the thirty-two items. For example, to compute the Boolean not of thirty-two values, a hardware designer might use thirty-two inverters.
Replication is difficult for programmers to understand and appreciate because replication is antithetical to good programming — a programmer is taught to avoid duplicating code. In the hardware world, however, replication has two distinct advantages. First, as we mentioned above, replication often makes the resulting hardware more elegant than hardware that uses iteration because replication avoids the extra hardware needed to select an individual item, move it into place, and move the result back. Second, and more important, replication increases performance dramatically. In addition to avoiding the overhead of selecting and moving items, replication allows multiple operations to be performed simultaneously. For example, thirty-two inverters working at the same time can invert thirty-two bits in exactly the same amount of time that it takes one inverter to invert a single bit. To put it another way: thirty-two inverters working simultaneously are more than thirty-two times faster than a single inverter iteratively solving the same problem. The notion of hardware replication and parallel operation appears throughout the text; a later chapter explains how parallelism applies on a larger scale.

2.17 Gate And Chip Minimization

We have glossed over many of the underlying engineering details. For example, once they choose a general design and the amount of replication that will be used, engineers seek ways to minimize the amount of hardware needed. There are two issues: minimizing gates and minimizing integrated circuits. The first issue involves general rules of Boolean algebra. For example, consider the Boolean expression:

\[ \text{not} \left( \text{not} \ z \right) \]

A circuit to implement the expression consists of two inverters connected together. Of course, we know that two \text{not} operations are the identity function, so the expression can be replaced by \( z \). That is, a pair of directly connected inverters can be removed from a circuit without affecting the result.

As another example of Boolean expression optimization, consider the expression:

\[ x \ \text{nor} \ (\text{not} \ x) \]

Either \( x \) will have the value 1, or \( \text{not} \ x \) will have the value 1, which means the \text{nor} function will always produce the same value, a logical 0. Therefore, the entire expression can be replaced by the value 0. In terms of a circuit, it would be foolish to use a \text{nor} gate and an inverter to compute the expression because the two gates will always yield the same output. Thus, once an engineer writes a Boolean expression formula, the formula can be analyzed to look for instances of subexpressions that can be reduced or eliminated without changing the result.

Although Boolean formulas can be optimized, further optimization is needed because the overall goal is minimization of integrated circuits. To understand the complexity, recall that some integrated circuits contain multiple copies of a given type of gate. Thus, minimizing the number of Boolean operations may not help if the optimiza-
tion increases the types of gates required. For example, suppose a Boolean expression requires four \textit{nand} gates, and consider an optimization that reduces the requirements to three gates: two \textit{nand} gates and a \textit{nor} gate. Unfortunately, although the total number of gates is lower, the optimization increases the number of integrated circuits required (because a single integrated circuit contains four \textit{nand} gates, but two circuits are required for a \textit{nand} and a \textit{nor} gate).

2.18 Using Spare Gates

Consider the circuit in Figure 2.16 carefully. Assuming the clock, counter, and demultiplexor each require one integrated circuit, how many additional integrated circuits are required? The obvious answer is two: one is needed for the \textit{nand} gate (e.g., a 7400) and another for the two inverters (e.g., a 7404). However, a hardware engineer can implement the circuit with only one additional integrated circuit. To see how, observe that although the 7400 contains four \textit{nand} gates, only one is needed. How can the spare gates be used? The trick lies in observing that \textit{nand} of 1 and 0 is 1, and \textit{nand} of 1 and 1 is 0. That is,

\[ \text{not } x \]

So, to use a \textit{nand} gate as an inverter, an engineer simply connects one of the two inputs to five volts.

2.19 Power Distribution And Heat Dissipation

In addition to planning digital circuits that correctly perform the intended function and minimizing the number of components used, engineers must contend with the underlying power and cooling requirements. For example, although the diagrams in this chapter only depict the logical inputs and outputs of gates, every gate consumes power. By everyday standards, the amount of power used by a single integrated circuit is insignificant. However, because hardware designers tend to use replication instead of iteration, complex digital systems contain many circuits. An engineer must calculate the total power required, construct the appropriate power supplies, and plan additional wiring that carries power to each chip.

The laws of physics dictate that any device that consumes power will generate heat. The amount of heat generated is proportional to the amount of power consumed, so a small integrated circuit generates a small amount of heat. Because a digital system uses hundreds of circuits that operate in a small, enclosed space, the total heat generated can be significant. Unless engineers plan a mechanism to dissipate heat, high tempera-
Sec. 2.19 Power Distribution And Heat Dissipation

tures will cause the circuits to fail. For small systems, engineers add holes to the chassis that allow hot air to escape and be replaced by cooler air from the surrounding room. For intermediate systems, such as personal computers, fans are added to move air through the system more quickly. For the largest digital systems, cool air is insufficient — a refrigeration system with liquid coolant must be used.

2.20 Timing

Our quick tour of digital logic omits another important aspect that engineers must consider: timing. A gate does not act instantly. Instead a gate takes time to settle (i.e., to change the output once the input changes). In our examples, timing is irrelevant because the clock runs at the incredibly slow rate of 1Hz and all gates settle in less than a microsecond. Thus, the gates settle long before the clock pulses.

In practice, timing is an essential aspect of engineering because digital circuits are designed to operate at high speed. To ensure that a circuit will operate correctly, an engineer must calculate the time required for all gates to settle.

Engineers must also calculate the time required to propagate signals throughout an entire system, and must ensure that the system does not fail because of clock skew. To understand clock skew, consider Figure 2.17 that illustrates a circuit board with a clock that controls three of the integrated circuits in the system.

![Figure 2.17](image.png)

**Figure 2.17** Illustration of three integrated circuits in a digital system that are controlled by a single clock. The length of wire between the clock and an integrated circuit determines when a clock signal arrives.

In the figure, the three integrated circuits are physically distributed (presumably, other integrated circuits occupy the remaining space). Unfortunately, a finite time is required for a signal from the clock to reach each of the circuits, and the time is proportional to the length of wire between the clock and a given circuit. As a result, the clock
signal will arrive at some of the integrated circuits sooner than it arrives at others. As a rule of thumb, a signal requires one nanosecond to propagate across one foot of wire. Thus, for a system that measures eighteen inches across, the clock signal can reach locations near the clock a nanosecond before the signal reaches the farthest location. Obviously, clock skew can cause a problem if parts of the system must operate before other parts. An engineer needs to calculate the length of each path and design a layout that avoids the problem of clock skew.

As a consequence of clock skew, engineers seldom use a single global clock to control a large system. Instead, several clocks are used, with each clock controlling one part of the system. In particular, clocks that run at the highest rates are used in small physical areas. Although using multiple clocks avoids the problems of clock skew, multiple clocks introduce another problem, clock synchronization: all clocks must be engineered to coordinate.

### 2.21 Physical Size And Process Technologies

Most digital circuits are built from integrated circuits (ICs), a technology that permits many transistors to be placed on a single silicon chip along with wiring that interconnects them. The idea is that the components on an IC form a useful circuit.

ICs are often created by using Complementary Metal Oxide Semiconductor (CMOS) technology. Silicon is doped with impurities to give it negative or positive ionization. The resulting substances are known as N-type silicon or P-type silicon. When arranged in layers, N-type and P-type silicon form transistors.

IC manufacturers do not create a single IC at a time. Instead, a manufacturer creates a round wafer that is a few inches in diameter and contains many copies of a given IC design. Once the wafer has been created, the vendor cuts out the individual chips, and packages each chip in a plastic case along with pins that connect to the chip.

ICs come in a variety of shapes and sizes; some have only eight external connections (i.e., pins), and others have over four hundred†. Some ICs contain dozens of transistors, others contain millions.

Depending on the number of transistors on the chip, ICs can be divided into four broad categories:

<table>
<thead>
<tr>
<th>Name</th>
<th>Example Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Scale Integration (SSI)</td>
<td>Basic Boolean gates</td>
</tr>
<tr>
<td>Medium Scale Integration (MSI)</td>
<td>Intermediate logic, such as counters</td>
</tr>
<tr>
<td>Large Scale Integration (LSI)</td>
<td>Small, embedded processors</td>
</tr>
<tr>
<td>Very Large Scale Integration (VLSI)</td>
<td>Complex processors</td>
</tr>
</tbody>
</table>

†Engineers use the term pinout to describe the purpose of each pin on a chip.
For example, integrated 7400, 7402, and 7404 circuits described in this chapter are classified as SSI. A binary counter or flip-flop is classified as MSI.

The definition of VLSI keeps changing as manufacturers devise new ways to increase the density of transistors per square area. Gordon Moore, a cofounder of Intel Corporation, is attributed with having observed that the density of silicon circuits, measured in the number of transistors per square inch, would double every year. The observation, known as Moore’s Law, was revised in the 1970s, when the rate slowed to doubling every eighteen months.

In addition to general-purpose ICs that are designed and sold by vendors, it has become possible to build special-purpose ICs. Known as Application Specific Integrated Circuits (ASICs), the ICs are designed by a private company, and then the designs are sent to a vendor to be manufactured. Although designing an ASIC is expensive and time-consuming — approximately a million dollars and nearly two years — once the design is completed, copies of the ASIC are inexpensive to produce. Thus, high-end digital systems often use ASIC chips.

2.22 Circuit Boards And Layers

Most digital systems are built using a printed circuit board that consists of a fiberglass board with thin metal strips attached to the surface and holes for mounting integrated circuits and other components. In essence, the metal strips on the circuit board form the wiring that interconnects components.

Can a circuit board be used for complex interconnections that require wires to cross? Interestingly, engineers have developed multilayer circuit boards that solve the problem. In essence, a multilayer circuit board allows wiring in three dimensions — when a wire must cross another, the designer can arrange to pass the wire up to a higher layer, make the crossing, and then pass the wire back down.

It may seem that a few layers will suffice for any circuit. However, large complex circuits with thousands of interconnections may need additional layers. It is not uncommon for engineers to design circuit boards that have eighteen layers; the most advanced boards can have twenty-four layers.

2.23 Levels Of Abstraction

As this chapter illustrates, it is possible to view digital logic at various levels of abstraction. At the lowest level, a transistor is created from silicon. At the next level, multiple transistors are used along with components, such as resistors and diodes, to form gates. At the next level, multiple gates are combined to form intermediate scale units, such as flip flops. In later chapters, we will discuss more complex mechanisms, such as processors, memory systems, and I/O devices, that are each constructed from multiple intermediate scale units. Figure 2.18 summarizes the levels of abstraction.
Abstraction Implemented With

- Computer
- Circuit board(s)
- Circuit board Components such as processor and memory
- Processor
- VLSI chip
- VLSI chip Many gates
- Gate
- Many transistors
- Transistor
- Semiconductor implemented in silicon

Figure 2.18 An example of levels of abstraction in digital logic. An item at one level is implemented using items at the next lower level.

The important point is that moving up the levels of abstraction allows us to hide more details and talk about larger and larger pieces without giving internal details. When we describe processors, for example, we can consider how a processor works without examining the internal structure at the level of gates or transistors.

An important consequence of abstraction arises in the diagrams architects and engineers use to describe digital systems. As we have seen, schematic diagrams can represent the interconnection of transistors, resistors, and diodes. Diagrams can also be used to represent an interconnection among gates. In later chapters, we will use high-level diagrams that represent the interconnection of processors and memory systems. In such diagrams, a small rectangular box will represent a processor or a memory without showing the interconnection of gates. When looking at an architectural diagram, it will be important to understand the level of abstraction, and to remember that a single item in a high-level diagram can correspond to an arbitrarily large number of gates.

### 2.24 Summary

Digital logic refers to the pieces of hardware used to construct digital systems such as computers. As we have seen, Boolean algebra is an important tool in digital circuit design; there is a direct relationship between Boolean functions and the gates used to implement combinatorial digital circuits. We have also seen that Boolean logic values can be described using truth tables.

A clock is a straightforward mechanism that emits pulses at regular intervals. A clock allows a digital circuit to change without requiring inputs to change, and can be used to provide synchronization among multiple parts of a circuit.

Although we think of digital logic from a mathematical point of view, building practical circuits involves understanding the underlying hardware details. In particular, besides basic correctness, engineers must contend with problems of power distribution, heat dissipation, and clock skew.