Introduction

Computer hardware, in its short 50-year history, has experienced the most dramatic improvement in capabilities and costs ever experienced by humankind. Computing devices are now ubiquitous in our daily lives. They are no longer machines for a few scientists, but are now used by an ever-growing portion of the world’s population. From their birth as automatic machines for arithmetic calculations, computers are now part of everyday appliances such as microwave ovens, anti-lock brakes, and pocket calendars. We no longer think about computers specifically when we use the devices that they make possible. CD and DVD players let us listen to music and watch movies - but they are, in fact, "computing" devices in that they include several microprocessors as well as other digital hardware (see Figure 1.1). Just think how the lowly rotary telephone has become a highly integrated communication system with many components. A portable handset allows you to initiate and answer calls from just about anywhere in the world with newer models requiring ever less frequent recharging. A home answering machine can digitize and record your greeting and allow you to review, save, and erase your phone messages. And your home phone line now also lets your computers access an unprecedented wealth of information and services over global networks.
Logic design is one of the disciplines that have enabled the digital revolution that dramatically altered our economies, communication systems, and lives. Contemporary logic design is the particular approach that is dominant today in this discipline. It is quite different compared to the approaches to logic design common even a short 10 years ago. Any introduction to this subject must begin by defining these terms. We will then provide some historical perspective by quickly reviewing the evolution of the underlying technology that makes a digital world possible. We'll conclude the chapter with a couple of examples of logic design that will serve as a road map and preview of the remainder of the text.

1.1 Dissecting the Title

1.1.1 Design

Design is the process of devising a solution to a problem. This means that we not only need to understand precisely what the problem is, but also the constraints that are imposed on the solution. Constraints can arise from physical limitations or from aesthetic and subjective criteria. For example, if we are designing a building, the problem may be to create 50,000 square feet of space for office workers. Physical constraints on the building design could include limits on its height, how far it can go underground, the number of offices with windows, and all the service utilities the building will need. The building’s external design will also have to fit in to the surrounding structures to form a pleasing effect. These are, of course, only a small part of the constraints that architects must manage when designing a solution. The architect must also ensure that the building is completed within its cost budget.
Chapter 1 Introduction

We recognize building design as a very complex process. In fact, most of the objects we design are themselves going to consist of components. We do this so that we can divide our problem into smaller sub-problems. Human beings are only able to keep so many details and their interrelationships in their heads at one time. By dividing the building into its constituent parts, and using a team of designers each working on their own parts of the whole, we are better able to manage a large design task. Each component is now a design problem for its respective designer who may in turn choose to divide it down further. In our building example, these will include office layouts, window frames, elevators, etc. The elevator design may be further decomposed into the design of the shafts and the electronic controls. The building’s chief architect is responsible for bringing these pieces together to form the complete building. Although, the architect may have some influence over the design of these pieces, it is likely that, due to cost considerations, the choice may be limited to pre-designed solutions available in only specific configurations.

The design process is quite similar in what may at first appear to be quite different disciplines. For example, designing a new software application is not really all that different than our building example. Memory or performance limitations of the type of computer on which the software must run may impose constraints on how which algorithms may make sense. User interface concerns also have an influence on our program’s design. Our software designer will be trying to be efficient and will want to reuse utilities available from the computer’s operating system or parts of software packages that were previously written. Thus, the software will have be structured in such a way to take advantage of the pieces that may already be available. Finally, the software required may too large for one individual, group, or even company and pieces will have to be sub-contracted to others to complete.

Complex systems, such as our building or personal computer software, require us to adopt a design methodology if we are going to be able to manage the process efficiently and effectively. A design methodology is a domain-specific formalization of the design process into a set of well understood steps. Architects have developed a design methodology that involves contractors and builders, materials suppliers, community and municipal agencies, and the eventual occupants of the building. An automobile designer also has a methodology but it is likely to be quite different than that of the building architect and include much more concern with safety and with reusing parts from existing car models.

Design has three important facets. It is a creative process of coming up with a vision of the solution. It is an engineering process of evaluating tradeoffs and making decisions among many alternatives. It is an optimization process of choosing the best combination of components to realize the vision.

1.1.2 Logic Design

Digital hardware makes up the palette of components that the digital designer has at the ready. Combinations of switches, built from semiconductor transistors, form the basis of all of today’s digital hardware. We generally refer to some interconnected collection of switches as a circuit. Individual switches are not the only
building blocks available. There are also higher-level circuit modules such as logic gates and memories. The logic designer’s job is to choose the right components to solve a logic design problem. Constraints in logic design are often related to some combination of size, cost, performance, and power consumption. Cost and size are very closely related as a component’s complexity is highly correlated with the number of switches it contains. Thus, a component’s size is directly proportional to the cost of manufacturing the component. Performance and power consumption are determined by the arrangement of switches and the underlying materials from which they are constructed.

All digital components can be viewed as having a set of inputs and a set of outputs. Inputs and outputs both consist of wires that carry digital logic values. A digital value is either 0 or 1. It cannot take on any other values. Of course, in the real world, we have more continuous phenomena. In the case of digital electronic circuits, any voltage below some level, say 1 volt, is interpreted as a logical 0 any voltage above some other level, say 2 volts, is viewed as a logical 1. Arbitrary information can be represented using this digital abstraction. Binary notation is used to represent integers as well as floating-point or even complex numbers. More interestingly, the color values for each pixel of digital image or the volume of the sound to be created by a speaker can also be represented digitally. The digital abstraction is extremely powerful and has led to a powerful convergence of all our data by letting us represent information ranging from telephone calls to digitized maps to movies to e-mail messages using the same media, namely, simple voltage values on wires.

The transistors inside a digital component react to the voltage levels on the inputs by changing the voltage level on the outputs of the component. We often refer to some combination of inputs causing a value to be applied to the outputs. For example, we may have a circuit with two inputs and one output; the output is set to one if both inputs are also one. This is referred to as "AND logic gate" because both the first input and the second input must be one. We’ll later see a wide range of logic gates. These types of circuits, where inputs directly influence the value on outputs, are referred to as combinational logic circuits. They form the basis of all our computational elements including components that can add, subtract, or even multiply.

Other types of digital circuits are referred to as sequential logic circuits. Their outputs not only react to the current values on the input wires but also to the past history of values on those same inputs. Thus, sequential circuits have memory in that they will remember the past inputs and react to the current inputs taking some aspects of the past into account. A simple example of sequential logic circuit is a memory component that has two inputs and one output; the output is set to the same value as one of the input wires when the second input wire signals it to do so. The input value that is "sampled" in this manner is then held indefinitely until the circuit is signaled again to store a new value. These types of circuits form the basis of memory devices that allow our computers to store and recall data as well as keep track of which step of a sequence of instructions, embodied in a computer program, they are currently executing.
Logic design is a set of abstractions and methodologies that let us devise, understand, and manipulate large collections of digital circuits. We've seen one abstraction already, namely, using 0 and 1 to represent all sorts of data. We'll see several others in this chapter and many others in the remainder of the text. Design methodologies are important procedures for ensuring a principled and effective design process. We'll be developing several of these for different types of digital logic. Finally, we'll also develop ways of transforming and optimizing our circuits so that they can have a better chance of meeting design constraints and be efficiently implemented using the different sets of components we may have at our disposal.

1.1.3 Contemporary Logic Design

Logic design has been around for at least 150 years. Its contemporary incarnation is due to several crucial trends in the design of digital hardware. First, our systems are becoming ever more complex as we integrate more functions into a device and perform computations on ever larger quantities of data. Just compare the complexity of a turn-of-the-century phonograph and today's DVD players (see Figure 1.2).

![Figure 1.2 Photographs of a 19th century phonograph and a 20th century digital video disc (DVD) player.](image)

The phonograph was virtually all mechanical and consisted of a handful of moving parts. A DVD player includes several microprocessors operating on a digital representation of images and sound giving us access to any part of a film instantaneously with no degradation over time. Second, the design of today's digital systems is happening in a much faster time frame as the demands of the consumer market place inexorable pressure on products to have a wide range of features appropriate for different uses and situations. New models of phonographs were introduced every few years. Today's DVD players began with dozens of mod-
els with new ones introduced continuously so that within a year we may see hundreds of models. Finally, the cost of digital hardware has become so low and its performance so high that we no longer need to be concerned with engineering the absolutely lowest cost solution.

These three trends have led to a radical change in the methodologies of logic design over the past 10 to 15 years. First among these is automatic generation of logic circuits using software tools. We can now specify the functions we want our circuits to perform using a high-level specification language and have a "logic compiler" refine and transform that specification into a set of components. Second, we have created digital components that do not have fixed functionality but can be used to perform a wide range of functions based on a configuration performed after they are manufactured. This provides immense economies of scale, as we no longer need to carry large inventories of different digital components. Third, the emphasis has now shifted from the crafting of the implementation (i.e., the arrangement of switches into circuits) to the crafting of the specification. That is, designers focus more on getting the high-level specification right, to meet all the functional requirements of their product, rather than on arrangements of transistors. They rely on the compilation tools to determine the set of components they will need and the configurations that will be required.

Contemporary logic design now faces many of the same problems as software design. Designers want to work using specification languages at ever higher-levels of abstraction. They can be much more productive at higher levels, as there are fewer components to consider. However, they also want to ensure that the resulting design will meet the design constraints. Hardware designers are interested in ensuring that an appropriate collection of components is used that costs no more than necessary and that they will perform their computations quickly enough. Similarly, software designers are also concerned with performance and the memory requirements of their applications. Both designers also need to be able to visualize their specification and debug them when things are not as intended. In logic design, simulation tools, that mimic the behavior of the real physical components, are an essential part of the designer's arsenal serving similar functions as the debugger does for software designers. Another important similarity between hardware and software also exists in methodologies for design. Both hardware and software designers understand the need to re-use as many portions of designs as possible. It is still an engineering art to devise specification of components that can be used in many different contexts. This is crucial to making design more economically efficient.

1.2 A Brief History of Logic Design

We will begin the history of logic design in 1850 when George Boole invented an algebraic system for manipulating logical propositions. Boolean algebra is now the mathematical foundation of logic design. It forms the basis for the optimization of digital logic much in the same way we use arithmetic algebra to transform expressions on variables into equivalent ones that have fewer operations.
example, by using the distributive law to perform one less multiplication in an algebraic expression \( a \times x + b \times x = [ a + b ] \times x \).

Claude Shannon’s Masters thesis in 1938 established a link between Boolean algebra and the switches used in the relay circuit of the day. This was an important step in moving Boolean algebra from the realm of abstract prepositional logic to physical devices that actually computed a logical expression.

By 1945, John von Neumann developed the first digital computer using vacuum tubes as the switching elements. This was a big advance from relays because vacuum tubes were much smaller and could switch much more quickly than relays. The disadvantage was that vacuum tubes were highly unreliable and over the course of a day many had to be replaced. These machines could perform several hundred multiplications per minute with 18,000 vacuum tubes - a huge advance from the hand-operated calculating machines, which were then the mainstay of scientific computation.

The invention of the transistor in 1947 heralded the dawn of the integrated circuit age. Highly reliable semiconductor switches replaced vacuum tubes very quickly. Soon thereafter, scientists were able to manufacture multiple transistors simultaneously using the technique of photolithography, invented in 19xx, to create patterns of semiconductor materials using light-sensitive materials and chemical etchants. Thus began the era of integrated circuits, which led to the first microprocessor, the 4-bit Intel 4004, in 19xx.

By the end of the 1960s, logic designers had available a large catalog of logic components (such as those described in the then ubiquitous Texas Instruments TTL data book). Arbitrary logic circuits could be built from these basic primitives which were mass produced in great quantities and so beginning the trend toward cheaper and cheaper electronic circuits with high reliability that continues to this day.

Programmable logic arrays, collections of switches in regular arrangements that could be configured by the logic designer to implement any one of a huge quantity of possible functions, soon arose to increase levels of integration and to make it easier for designers to change the wiring pattern between logic functions. These devices started to see wide use with the introduction of the Monolithic Memories PALs in 19xx.

With the increased levels of integration in digital circuits and the need for designers to program their reconfigurable logic components came the development of logic synthesis tools. Starting from Boolean algebra expressions, these software packages could determine the precise configuration of logic arrays to implement the specified function and therefore free designers from the drudgery of dealing with every single switch and allowing them to focus on higher level design. One can think of these early tools as the assemblers of their day, translating from assembly language to machine code. Their successors today are much closer to high-level language compilers and development environments.

Today, we see the continued development of programmable logic, in the form of field-programmable gate arrays, which can now be reconfigured over and over again. This makes possible logic circuits that can be altered over time, field upgraded after a product has been purchased, or even from one use of the product
to the next. Synthesis tools have followed closely with the appropriate compilation technology to configure these new types of components. Finally, level of integration have continued to increase so that we can now consider using many transistors just to give our circuits more flexibility rather than stingily allocating each individual transistor as in the early days of integrated circuits.

1.3 Computation

Up to now, for most readers, computation has been an abstract process. You may have performed computation or specified the steps for a computer to execute. The details of how that computation is actually accomplished have probably been somewhat of a mystery. You may know that digital computers operate on bits, 0s and 1s, and you may know how they represent integers and characters using strings of bits, usually 32 and 8 bits long, respectively. But how do these strings of bits lead to the execution of complex programs?

This text is about de-mystifying computation. It will guide you through the first steps in understanding how computers work. We’ll also generalize the concept of computation, from what you may have seen in your introductory programming classes, to include parallel computation. After all, we can have many circuits working in parallel on different inputs. At a small scale, arithmetic circuits provide an excellent motivation for parallelism since we can work on the different parts of the strings of bits in parallel. At a larger scale we’ll see how two simple computers, working independently, can go about communicating and coordinating their activities.

In the coming chapters, we’ll see how to implement all the common programming constructs, including variable assignment, arithmetic operations, conditional and iterative statements, and subprocedures. We will learn how to construct circuits that perform all these functions using a few simple primitive elements. In addition, we’ll also see that circuits can perform functions in parallel, not just sequentially, and operate on arbitrary data types, not just bits and integers.

1.3.1 Switches, relays, and circuits

Switches are the basic building blocks of digital computers. Like adenine, cytosine, guanine, and thyomine, the components of DNA, switches in the proper arrangement express the physical embodiment of a more abstract computation.

Let’s review how switches work. In Figure 1.3, we show a simple switching network that is found in every home, probably in every room. A switch is used to disconnect a light bulb from its power source. If the switch is open, current does not flow through the circuit and the light bulb is off. If the switch is closed, then the light bulb is turned on as the battery’s current can now travel through the light bulb and return to the battery - a completed circuit.

If we represent the state of the switch by using a Boolean variable, say A, that we set to 0 if the switch is open and 1 if it is closed, and represent the state of the
light bulb using another Boolean variable, say $Z$, that we set to 0 if the bulb is off and 1 if it is on, then we can write:

$$Z = A$$

as a Boolean expression that represents the functionality of this circuit. If $A$ is 0, then $Z$ is 0 (switch open, light off) and if $A$ is 1, then $Z$ is 1 (switch closed, light on). We can also use the symbol in Figure 1.4 to indicate this circuit in a schematic drawing. Note the arrow shape indicating that the output $Z$ gets the value of the input $A$.

We can make our simple circuit a bit more interesting if we now add a second switch in series with our original switch represented by $A$ (see Figure 1.5). We will represent this new switch with the Boolean variable $B$. Now the functionality of our circuit is such that both switches have to be closed for the bulb to turn on. We can write this expression as:

$$Z = A \text{ and } B$$

to indicate that both $A$ and $B$ have to be 1 (both switches have to be closed for the bulb to be on). A circuit like this is commonly found in automobiles where the key activates one switch and the windshield wiper wand activates another. Both have to be "on", that is, both switches have to be closed for the windshield wipers to work. The schematic symbol for AND is shown in Figure 1.6.

At this point, it is clear that we think of 1 as having a meaning of "true" and 0 as "false". This is, in fact, the common convention, and it makes the use of the word and makes sense in everyday English. However, the choice is really arbitrary. We'll see later on that it really doesn't matter what value represents true or false, just as long as we are consistent.

Another choice for the arrangement of two switches is a parallel arrangement (see Figure 1.7). In this case, either $A$ or $B$ being closed will turn on the bulb. Our expression for this circuit is

$$Z = A \text{ or } B$$

to indicate that either $A$ or $B$ have to be 1 for the bulb to be turned on. An example of this is also found in cars where the dome light will turn on whether the driver's or the passenger's door is opened. The schematic symbol for OR is shown in Figure 1.8.

Switch settings determine whether a complete circuit or conducting path exists to light the bulb. We've assumed so far that someone sets the switches to be open or closed. To build larger and more interesting circuits, we have to be able to combine the basic ones we've just discussed. To do this, we need a way to get the state of a light bulb to control a switch on another circuit. In the early days of digital circuits, this was accomplished using a special device called a relay (see

**Figure 1.4** Schematic symbol for the simple circuit of Figure 1.3.

**Figure 1.5** Two switches in series will close a circuit if both $A$ and $B$ are closed.

**Figure 1.6** The schematic symbol for the AND function.

**Figure 1.7** Two switches in parallel will close a circuit if either $A$ or $B$ is closed.

**Figure 1.8** The schematic symbol for the OR function.
Figure 1.9 A simple magnetic relay. The switch at the top is opened when a current passes through the electromagnetic directly below it.

Figure 1.10 The symbols for n-type and p-type CMOS transistors. The terminals are labelled G (gate), S (source), and D (drain).

Figure 1.11 A simple CMOS network utilizing two transistors to connect the output Y to either 3V or 0V based on the value of X.

Figure 1.12 The schematic symbol for the NOT circuit of Figure 1.11

Figure 1.9). Its name is derived from the fact that it serves as a connection point between two otherwise independent circuits.

A relay operates much in the same way as a light bulb. The difference is that instead of creating a glowing light, the current that passes through the device creates an electromagnet. A special switch is constructed from a ferric material so that if the magnet is on, it opens the switch by attracting its flexible half of it away from the stationary half. If the magnet is off, the switch's flexible half snaps back into position and closes the switch. This is an example of a switch that is normally closed and opened when current flows through the nearby electromagnet. The magnet and switch, together, form the relay.

Relay circuits were large and slow and could never have been used to construct large-scale computing machines. Magnets take time to charge and mechanical switches are slow to move as they have inertia directly proportional to their mass. It took the invention of electronic devices such as vacuum tubes to make that possible. However, vacuum tubes were still large and very unreliable. The invention of the metal-oxide-semiconductor, or MOS transistor, was needed to open the doors to useful and affordable computing devices.

1.3.2 Transistors

In today’s integrated circuit technologies we have two types of switches or two transistor types. They are referred to as n-type and p-type transistors and are named after the type of semiconductor material from which they are constructed. They can be switched open or closed by applying a low or high voltage to one of their terminals. N-type devices are normally open switches just like the ones we discussed above. By normally open we mean that when a low voltage is applied to their "gate", or controlling terminal, they act as an open switch and when a high voltage is applied they act as closed switch completing the connection across their other two terminals. P-type devices are exactly the opposite in that they are normally closed and open when a high voltage is applied to their "gate". Hence, the name CMOS technology, which stands for complementary MOS technology referring to its two complementary types of switches (see Figure 1.10).

A simple CMOS network is shown in Figure 1.11. It consists of one switch of each type. The p-type device is used to connect the output, Y, to a high voltage while the n-type device is used to connect the output to a low one. Both transistors are controlled by the same voltage as their gates are connected to the same wire. If the input, X, is 0, then the p-type switch will be closed and the n-type switch will be open, causing the output to be connected to a high voltage or 1. The reverse is true if the input voltage is high, then the output is connected to a low voltage or 0. This is a very simple logic device called an inverter because its output is the opposite of its input. It is often referred to as a NOT device or gate (an historical term that, unfortunately, is the same word used for the controlling terminal of a transistor). The symbol for the NOT gate is shown in Figure 1.12. The bubble at the end of the arrow shape is used to indicate the inversion property of the circuit.

You should now be asking yourself: why this particular arrangement of switches? The motivation for it comes from the fact that our transistors are not the
idealized switches we have discussed so far. P-type devices do a good job of connecting high-voltages but are not very good at connecting low voltages. What happens is that the output voltage isn't quite low enough and could not be easily used to control a transistor in another circuit. Similarly again, the n-type device is good at connecting low voltages but not high ones. Fortunately, we have both types.

CMOS transistor networks are constructed by creating a p-type switch network for the cases when we want the output to be high and an n-type switch network for the remainder of the cases when we want the output to be low. The next most basic CMOS networks are shown in Figure 1.13 and represent serial and parallel arrangements of the transistors. You'll note that when the p-type devices are serially connected, the n-type devices are in parallel and vice-versa.

Let's step through the operation of the device on the left. When either input is low, the output will be connected to a high voltage. Only when both inputs are high, is the output connected, through the two serially connected n-type devices, to a low voltage. This is starting to sound like an AND arrangement except that the output is low when both inputs are high. Thus, we refer to this circuit as a NAND gate (for a NOT-AND gate in that it is an AND gate with its output inverted). Its dual is the switching network on the right which is referred to as a NOR gate for NOT-OR. The schematic symbols for NAND and NOR are shown in Figure 1.14. Note, again, the use of the bubbles at the output to indicate inversion.

Transistor networks are much faster than relays. The switching time is determined by the flow of electronics in the semiconductor materials that make up the transistors. A water faucet provides a useful analogy (this idea is not really that far-fetched, since electrical effects are ultimately due to the flow of electrons, which act in a way quite analogous to water). Theoretically, a water faucet is either on, with water flowing, or off, with no flow. However, if you observed the action of a faucet being shut off, you would see the stream of water change from a strong flow, to a dripping weak flow, to a few drips, and finally to no flow at all. The same thing happens in electrical devices. They start our by draining charge rather quickly, but eventually the discharge slows down to a trickle and finally

Figure 1.13 Two CMOS transistors networks that demonstrate the arrangement of switches for a NAND operation (left) and a NOR operation (right).

Figure 1.14 Schematic symbols for the NAND and NOR circuits of Figure 1.13, respectively.
stops. To complicate matters further, transistors are leaky faucets and never quite stop the flow of electrons completely.

As transistors have shrunk, and continue to shrink, in size, their switching speeds are increasing. There is less water in the pipes. This is what has fueled the rapid advances in computing technology of the past 40 years. Of course, we may soon see the day when we do reach fundamental limits and our circuits will depend on the movement of so few electrons that other physical effects will come into play.

1.3.3 Digital representations

Real world electronics are quite complex. The digital abstraction, interpreting all voltages as falling either into the "high" or "low" category, is a crucial step in being able to build digital circuits from imperfect switches such as transistors. Just imagine for a moment how difficult it might be to realize a circuit that depended upon precise and continuous voltages. A slight variation in a transistor, due to the process involved in its manufacture, could alter the operating voltages and render the circuit useless. Transistors, as all electronic components, exhibit aspects of continuous, or analog, behavior. They do so because output transitions are not instantaneous, they require electrons to flow and charge or discharge wires analogous to a water faucet or drain filling or emptying a tank.

Digital logic eliminates these problems by not taking on the too difficult task of recognizing a single voltage value as logic 1 or 0. Digital logic must be able to deal with degraded or imperfect signals. Since digital circuits recognize any analog value above a specified voltage as logic 1, they can use degraded inputs to still generate correct output voltage levels. As an example, assume that "on" or logic 1 is represented by +3 volts. "Off" or logic 0 is represented by 0 volts. But we will interpret any voltage above 2 volts as a 1 and any voltage below 1 volt as a 0. Figure 1.15 shows a plot, or waveform, of an output switching from on/1 to off/0. You might observe a waveform like this on an oscilloscope. The transition in the figure is certainly not instantaneous. Other values, between 0 and 3 volts, are visible even if only for a brief instant in time. These are interpreted by digital circuits as either 0 or 1.

Digital circuits also do not output perfect voltages (0 and 3 volts). Their transistors, again due to imperfections in the manufacturing process, will most likely output voltages somewhat greater and less than the nominal 0 and 3 volts. Furthermore, ambient conditions such as humidity, temperature, and radio waves may also affect the behavior of devices. We can not count on the outputs of our circuits to be perfect. This variation is often referred to as "noise". The noise margin of our circuits is the tolerance in voltage values, the difference between a nominal output voltage and the range of input voltages that will be properly interpreted.

Let's look at the transfer characteristic of the inverter. Figure 1.16 is a plot of the inverter's output voltage given all possible values of its input voltage. You'll note as the input goes higher, the output goes lower. However, it doesn't do so linearly. For input voltages between 0 and 1 volts, the output is very close to 3 volts. Similarly, for input voltages from 2 to 3 volts the output is very close to 0 volts.
Intermediate values of input voltage, between 1 and 2 volts, cause the output to be further away from either of the two nominal values of 0 and 3 volts. This one volt range for logic 0 and logic 1 inputs is called the noise margin. Outputs can differ by as much as 1 volt from the nominal value when there is a 1 volt noise margin. Clearly, we should try to avoid voltages in that intermediate range between 1 and 2 volts.

### 1.3.4 Encoding

The manipulation of digital data is at the heart of computing devices. Digital representations exist for everything from numbers and characters to music and images. A digital representation is simply a string of 0s and 1s with an agreed upon interpretation. Of course, the more people, organizations, and software applications agree on the interpretation, the better.

A simple example of encoding is the audio stored on a CD. Small pits in the plastic of the disc cause a fine laser beam to reflect differently than if the surface patch the laser illuminates did not have a pit. Collections of these bits make up a sound sample, or the volume to be reproduced on a speaker at the particular rate of playback. If the sample is interpreted correctly, that is, the bits are in the right order to represent the volume level, and they are sent to the speaker’s digital-to-analog converter at the right speed, then the recorded sound will be faithfully reproduced.

Another example of encoding led us through the Y2K fiasco. Early programmers used only two digits to represent the year. This enabled them to encode the year of a database record in only two 8-bit bytes or sometimes just 8-bits altogether. Of course, when the year 2000 finally arrived, 1900 was indistinguishable from 2000 and records were in danger of being wrongly sorted and correlated.

We also use encoding when we write software programs. The variables we use are often encoded as 32 or 64-bit binary numbers. The characters we use to represent text are represented as 8 or 16-bit values. We’ll encounter encoding issues over and over again in this book. In each case, we’ll have some freedom to decide how many bits will be used and what the order of the bits will mean. For example, a traffic light could have its interface consist of three signals, one for each bulb, or only two, as there are only 3 possible settings for the light and these can be encoded in a 2-bit binary number, say 10 for green, 11 for yellow, and 01 for red. We must take care in deciding on our encoding that we’ve taken into account the possible values that we’ll need to represent so we do not repeat the Y2K problems.

### 1.4 Examples

It is now time to turn to two examples to illustrate many of the concepts discussed so far in this introduction. We’ll use the examples to bring up some more of the terminology of logic design and introduce the primitive logic elements we’ll be using in the rest of the text.
1.4.1 Calendar

Our first example is a very small circuit of a larger design we’ll build as we progress through the book. Its function is to decide, based on the month and whether a year is a leap year or not, how many days are in a month. This will be used as part of a calendar display that could be part of a wristwatch.

Let’s begin by thinking about the specification we’ve just been given. Our simple system has two inputs: a month (January to December) and leap year indicator. It also has to have some outputs that tell us whether the month has 28, 29, 30, or 31 days. We could write a simple procedure that performs the function of our circuit. We’ll use the C language in this book.

```c
integer number_of_days ( month, leap_year_flag) {
    switch (month) {
        case 1: return (31);
        case 2: if (leap_year_flag == 1) then return (29) else return (28);
        case 3: return (31);
        case 4: return (30);
        case 5: return (31);
        case 6: return (30);
        case 7: return (31);
        case 8: return (31);
        case 9: return (30);
        case 10: return (31);
        case 11: return (30);
        case 12: return (31);
        default: return (0);
    }
}
```

The procedure is quite simple. It has the two inputs as parameters and the output as the return value. The body of the procedure is a large switch statement that branches based on the value of the parameter for the month and returns the appropriate value. The only complication is February which has a conditional statement that checks the leap year flag.

In implementing this program in digital logic, we are already faced with some encoding problems. How do we represent the month? In software, it would be an integer but there are only 12 of them and we are unlikely to add any more. We do not need all the values possible with a 32-bit integer. To represent twelve possibilities, we need a minimum of 4 bits if we use a “binary” encoding. But 4 bits have 16 possible values and we’ll only need 12. Four of them will be unused. We term these “don’t care” combinations because they should never occur and we will not be concerned with how our circuit reacts. Of course, should they occur we might be in trouble. Later on in the text we’ll see how we can use don’t cares to help us make smaller and faster circuits. Using four bits is not our only option. We could use 12 wires, one for each month. This is called a “one-hot” encoding. In this type
Chapter 1 Introduction

of encoding, we have one and only one wire ever carry logic 1, the wire correspon-
ding to that month. This is a very simple encoding but requires many more
wires, in this case, 12 instead of 4.

The leap year flag is straightforward, as it is a simple Boolean value. We’ll
need only a single wire that will be a 1 if the year is leap and 0 if it is not. We
have to make sure to decide on that encoding and not its opposite. This decision
better make its way into our documentation so that whoever builds the rest of our
calendar system will be well informed of the choice we made.

Finally, we have to encode our output. The value will be as high as 31. This
would require 5 bits in a binary encoding. On the other hand, there are only 4 pos-
sible results. We could use 4 wires and use a one-hot encoding.

Let’s make some choices and get on to designing our logic. We’ll choose a 4-
bit binary encoding for the month, a single wire for the leap year flag, and a 4-bit
one-hot encoding for the result. Figure 1.17 shows the inputs and outputs of our
circuits schematically.

Our next step is to figure out what the output should be for each combination
of inputs. For example, if the month is February or 0010, and the leap year flag
is 1, then the result is 29 which would be encoded as 0100 if the 4 bits are 28, 29,
30, and 31 days, from left to right. We’ve already done something very similar to
this in our program code. The completed table is shown in Figure 1.18. It has col-
umns for all the inputs on the left side of the vertical line and a column for each
output to the right. There is one row for each possible input combination. This
type of table is referred to as a “truth-table” because it specifies when a particular
output should be true (1, high voltage). It could have just as easily been called a
“falsity-table” but truth table sounds a whole lot better.

You should notice that the table is not complete in that every single input com-
bination is not shown explicitly. We took some shortcuts. For example, for March
(0011) we do not care what leap year flag is, March always has 31 days. Rather
than having two rows as for February, we only have one with a dash in the column
for the leap year flag indicating an "input don’t care". Don’t cares also appear in
the output columns. This occurs because of those 4 unused values of our 4 bit
month number (0, 13, 14, 15). Note also the use of the don’t care to merge the
rows for 14 and 15. Truth tables should be complete. This means that we should
make sure to specify what we want the output to be for every single input combi-
nation. If we had left any out, then we would have their outputs open to interpreta-
tion and a likely error when we start combining our piece with other components.
Output don’t cares let us say explicitly that a designer can choose either a 0 or 1
without worry.

The truth table can now be used to derive Boolean logic expressions for each
of our outputs. For example in the case of the output for 28 days, we have a very
simple expression that says the month must be February and the leap year flag
must be 0. Let’s give names to the four bits of the month and the four bits of the
number of days. We’ll use m8, m4, m2, m1 for the month, chosen so that the
names correspond to the weight of the bit in the binary representation, and we’ll
use \(d_{28}, d_{29}, d_{30}, \) and \(d_{31}\) for the number of days. Our Boolean expression for \(d_{28}\) can then be refined as follows:

\[
d_{28} = "February" \text{ AND } (\text{leap} == 0)
\]

\[
d_{28} = (m_8 == 0) \text{ AND } (m_4 == 0) \text{ AND } (m_2 == 1) \text{ AND } (m_1 == 0) \text{ AND } (\text{leap} == 0)
\]

\[
d_{28} = m_8' \text{ AND } m_4' \text{ AND } m_2 \text{ AND } m_1' \text{ AND } \text{leap'}
\]

The last line used a symbol for negation. \(m_2\) stands for \(m_2\) being a 1 while \(m_1'\) stands for \(m_1\) being a 0. We can implement the quote or negation symbol (\('\)) using an inverter or NOT gate.

We can use AND and NOT circuits to implement our logic for computing \(d_{28}\). On the left side of Figure 1.19 we used 4 inverters (for \(m_8, m_4, m_1, \) and leap) and one 5-input AND gate. These gates are only logical constructs that help us draw a diagram for the Boolean expression above. Our next step is to find physical gates that implement these same functions. Now, we know how to implement a NOT gate, but how does one build an AND gate? In CMOS technology, we've seen that it's easy to get a NAND function. Thus, we'll use a 5-input NAND gate and then invert its output with a fifth inverter as shown on the right side of Figure 1.19. We can increase the number of inputs by putting more switches in parallel (and in series on the complementary side of the gate).

\[d_{31} = "January" \text{ or } "March" \text{ or } "May" \text{ or } "July" \text{ or } "August" \text{ or } "October" \text{ or } "December"\]

\[d_{31} = (m_8' \text{ AND } m_4' \text{ AND } m_2' \text{ AND } m_1) \text{ OR } (m_8' \text{ AND } m_4' \text{ AND } m_2 \text{ AND } m_1) \]

\[\text{OR } \ldots \text{ OR } (m_8 \text{ AND } m_4 \text{ AND } m_2' \text{ AND } m_1')\]

![Figure 1.19 Logic gate diagrams for the circuit of the d28 output of the calendar.](image-url)
Chapter 1 Introduction

This output will require the use of a seven input OR gate as shown on the top in Figure 1.20. Does such a gate exist? Can we increase the number of inputs arbitrarily? We know we can do the same for the NOR as we did for the NAND gate and invert the output of the NOR to get an OR function using a NOR gate followed by a NOT gate. But what will we do about the number of inputs or "fan-in" to the gate? Recall that the transistors we use for building our switching circuits are not perfect. Those imperfections also make putting too many transistors in series impractical because the circuit get too slow and may cease to work entirely. We’ll limit our design to 4-input gates and replace the 7-input OR gate we need for the d31 output with the circuit shown on the bottom in Figure 1.20. We’ll see later how to convert this circuit to NOR and NAND gates.

Note that for this example, when we completed our truth table, we simply looked at each possible combination of input values in isolation from the others. This is why we refer to this type of circuit as combinational logic. Our next example introduces sequential logic, where the sequencing of different input values is important as well.

1.4.2 Combination lock

A simple door combination lock might consist of punching in a sequence of 3 specific keys on a small keyboard. The lock would then open if the sequence was correct and not open if it is was incorrect. A reset button may also be provided so that the user can start a sequence from scratch if they make an error. Our next design task is to implement such a lock.

Again, let’s begin by writing a software program that has the functionality of our lock. The procedure has no explicit inputs and assumes there is another procedure that can tell when a new key has been pressed (new_value) and another procedure that can read the value of the key that was pressed (read_value). You should note that the program has the combination for opening the door explicitly encoded in some static variables arranged into an array of three elements. It then uses conditional expressions to check the key pressed against the stored combination. A while statement is used so that the procedure can keep checking if a new key press has been made. After it detects a key press and there is a match it continues on to wait for the next key press. If there is a mismatch, it also continues...
on, but first sets an error flag. Finally, after three keys are pressed, it either opens the lock or not depending on the error flag.

```c
integer combination_lock () {
    integer v1, v2, v3;
    integer error = 0;
    static integer c[3] = 3, 4, 2;

    while (!new_value( ));
    v1 = read_value( );
    if (v1 != c[1]) then error = 1;

    while (!new_value( ));
    v2 = read_value( );
    if (v2 != c[2]) then error = 1;

    while (!new_value( ));
    v3 = read_value( );
    if (v2 != c[3]) then error = 1;

    if (error == 1) then return(0); else return (1);
}
```

Our logic design can now begin by determining the precise nature of the inputs and outputs of our system. First, we have a reset button that is not in the code above. The reset button is a simple Boolean value. Second, we need to decide how many keys to have and how they should be encoded. We'll choose 4-bit values in a binary encoding giving us 16 keys on the keypad and a total of 16*16*16 different combinations for the lock. The output is a simple Boolean value that either opens the lock or keeps it closed (as does the return value above). But where in the program are the inputs to our system? They are encapsulated in the calls to new_value and read_value. Those procedures are actually concerned with the keypad. Before we can tackle how to design logic for new_value and read_value there are a few other issues we need to resolve.

You may have noticed that we need to be able to tell how far along we are in entering a combination into the lock. Are we looking for the first key press, the second, or the third? To do this, we need to introduce the notion of sequence and with it a time component to the behavior of our circuit. In the program, this was done using the sequential nature of our computers (in that they execute one step at a time) and the fact that our computers have what is called a program counter that points to the part of the program to be executed next. Conditional statements, such as the while statements in our code, alter the program counter so it can execute instructions that do not follow each other linearly in the text of the program.

To accomplish the same thing in our digital circuit, we'll use what a computer uses: a special signal called a "clock". Its name is quite descriptive indicating that it alternates between logic 1 and logic 0 at a regular rate. The period of this oscil-
lation (the time it takes to complete an entire cycle of being set to 0 and 1) is the inverse of the frequency which is the parameter commonly used to discuss the performance of computers. The clock's beat lets us advance from one step to the next. We'll also need some memory to keep track of where we are at any given time in completing a combination sequence. This is called the "state" of our system.

We can now begin describing how the state of our system changes over time. This is accomplished through a "state diagram" (see Figure 1.21). Each possible state is represented as a bubble. Arcs drawn from one state to another indicate under what conditions the system's state will change as indicated. The change will not occur until there is another clock tick. That way there is time for our logic circuitry to make a decision as to which state to go to next. If the clock is too fast, the decision may not be complete and our system will go to the wrong state. If its too slow, then we'll waste time waiting for the clock tick with the decision already made.

![State diagram for the combination lock.](Diagram)

Figure 1.21 State diagram for the combination lock.

Clearly, there should be a state for each step of the sequence. We'll also have two states to represent whether we open the lock or if there has been an error. These five states are shown in the state diagram below. The conditions on the arcs coming out of a state represent the decisions that need to be made when the system is in that state. For example, when waiting for the first key press in state S1, the system is waiting for a new key press (the signal new indicates this) and whether this new key matches the first element of the combination (\(C_1 = \text{value}\)) or not. If there is no new key pressed, then the system's state does not change (the arc coming back to the same state S1). If there is a new key press, then depending on the match, the system either enters the error state, ERR, or moves on to state S2 to wait for the second key press.

How do we know when a key is pressed? Key presses may last a while. We only want to get a signal that there was a new key pressed once per key entered. We wouldn't want to view a single long key press as two or three separate key
presses. We'll defer the details of this to a later chapter. For now, lets assume that every time a new key is pressed the new signal is set to 1 for exactly 1 clock period and no more. This is another example of a constraint now imposed on whoever will design the keypad circuitry and it certainly belongs in our documentation.

Next on our list is the reset button. What does it do exactly? The state diagram provides a clear explanation. If the reset signal is ever 1 for a clock period then that means that no matter what state the system is presently in, it will transition to state S1 at the next clock tick. This is shown in the diagram of Figure 1.21 in shorthand by using an arrow that just points into state S1. This implies an arc from every other state to S1 that is traversed whenever reset is true. Of course, this also implies that the condition on every other arc also includes reset being false.

Finally, we have our system's inputs and outputs. There is a signal wire for when a new key is pressed (new), there are 4 wires for the value of the key (which only make sense if new is true), and there is a reset wire. In addition, there is the clock signal that can also be viewed as another input. The output is a single wire that controls whether the lock is open or not. Figure 1.22 shows the inputs and outputs of our system and the fact that it has internal state.

We can now turn to the internal structure of our lock system. We'll begin by separating the portions of the circuit that will operate on the key values from those that will concern themselves with the proper sequence of comparisons and their result. These are referred to as the "data-path" and the "controller" of the circuit. Elements of the data path operate on their inputs the same way no matter what the values of those inputs. For example, it always compares the new key pressed with an element of the combination no matter what their values are. The result is forwarded to the controller that is concerned with whether it matched or not because that will determine the next step to take.

The internal structure of our circuit is shown in Figure 1.23. The datapath consists of three memory components that store the combination (we'll see in later chapter how these values are set in memory), a multiplexer, and a comparator. All of these circuits operate on 4 bit quantities. A multiplexer is simply a switch. Its output is set to the same value as one of its multiple inputs. Which input is determined by the values on some other input signals referred to as control wires. We'll see later how to build a multiplexer from combinational logic. The comparator is also combinational logic that outputs a single bit that indicates if the values it compared were equal or not. Data path elements are either combinational logic.
Chapter 1 Introduction

such as multiplexers and comparators or they are memory elements, usually called registers.

The controller is a "finite state machine", a concept that we will refine quite a bit further in later chapters. It will have five possible internal states (corresponding to S1, S2, S3, ERR, and OPEN in the state diagram) and inputs for reset and new as well as equal, the result of the comparator. It, of course, also requires a clock input to be used to advance from one state to the next. Its outputs are the controlling wire for the lock as well as the control for the multiplexer in the data path so that the right element of the combination gets to the comparator in its corresponding step in the computation.

To implement our finite state machine we need to revisit the state diagram and turn it into a state table. A state table is very much like a truth table (see ). A row corresponds to every combination of inputs and states. The outputs on the row are only the output signals but also what the next state of the machine should be. There is no difference in the information content of a state diagram and a state table. Let’s take a look at one of the rows. The second row of the table states that in state S1, if there is not a new key press and reset is also 0, then it doesn’t matter what the value of equal is, the next state will be S1, the multiplexer will be set to channel the first combination element to the comparator, and the lock will be closed.

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>next state</th>
<th>mux</th>
<th>open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>S1</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S1</td>
<td>ERR</td>
<td>-</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S1</td>
<td>S2</td>
<td>C2</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>S2</td>
<td>S2</td>
<td>C2</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S2</td>
<td>ERR</td>
<td>-</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S2</td>
<td>S3</td>
<td>C3</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>S3</td>
<td>S3</td>
<td>C3</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S3</td>
<td>ERR</td>
<td>-</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S3</td>
<td>OPEN</td>
<td>-</td>
<td>open</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>OPEN</td>
<td>OPEN</td>
<td>-</td>
<td>open</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>ERR</td>
<td>ERR</td>
<td>-</td>
<td>closed</td>
</tr>
</tbody>
</table>

Figure 1.23 Internal structure of the combination lock.

Figure 1.24 State table for combination lock.
The one thing that is different about our state table than a truth table is that we have symbolic names for the states rather than 0s and 1s. Our next step is to derive an encoded state table where we assign a unique code to each and every state so that the circuit can tell them apart. We also have to assign codes to the multiplexer control wires and the lock output. We have many choices in this encoding. For the five states we can use anywhere from three to five bits (binary to one-hot encoding). Conceptually, it doesn’t matter which we use as long as each state has a unique identifying code. Practically, one code may lead to a much smaller circuit than another one would. We’ll revisit this topic in later chapters as well.

The encoding of the multiplexer control signals depends on how we designed the multiplexer or the particular multiplexer we chose from a catalog. This is an example of inter-related component design. One can’t be designed completely without the other being completed - a classic chicken-and-egg problem. The solution is to not charge ahead on one component’s design but to iterate and refine both in parallel. In our case, we could use 2 or 3 wires to indicate which combination element to compare. Finally, the lock output is simple enough that we can use a single wire that is 0 when the door is locked and 1 when it is open.

An encoded state table is shown below in Figure 1.25. Note that the multiplexer uses a one-hot encoding for its control signals while the state code was chosen to be 4 bits with 11 of the 16 combinations going unused. You’ll note however, that the first three bits of the next state are identical to the multiplexer control outputs and the fourth bit of the next state is identical to the lock output. Because we had lots of flexibility to choose 5 codes out of the 16 available, we were able to choose wisely so that we only need to implement 4 output circuits rather than 8.

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>next state</th>
<th>mux</th>
<th>open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>0001</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>−</td>
<td>0001</td>
<td>0001</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0001</td>
<td>0000</td>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0001</td>
<td>0010</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>−</td>
<td>0010</td>
<td>0010</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0010</td>
<td>0000</td>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0010</td>
<td>0100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>−</td>
<td>0100</td>
<td>0100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0100</td>
<td>0000</td>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0100</td>
<td>1000</td>
<td>−</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>−</td>
<td>−</td>
<td>1000</td>
<td>1000</td>
<td>−</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>−</td>
<td>−</td>
<td>0000</td>
<td>0000</td>
<td>−</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1.25 Encoded state table for the combination lock.

The controller’s internal structure is shown in figure XXX. It consists of two parts: a memory module that keeps track of the state and is updated every time the clock ticks and a combinational logic module that given the current state (the output of the state memory or register) and the inputs to the controller determines the
outputs and the next state. The next state will become the current state at the next
clock tick.

This example was a whirlwind tour of sequential circuit design and some of
the issues designers face. These issues are the subject of several chapters later in
the text. We highlighted the difference between combinational and sequential logic
by demonstrating through the example, not only the need for memory, but also for
sequencing a computation through its steps.

Chapter review

This first chapter has necessarily introduced what may very well be an over-
whelming number of definitions and concepts. The following chapters will hope-
fully make these crystal clear by covering them more slowly, in much more depth,
and with practice through examples and problem sets.

The chapter introduced many of the layers of abstraction that are crucial to
making the logic design of today’s complex digital systems possible. We intro-
duced transistors and relays as underlying technologies but quickly abstracted their
details into switches. Switching networks were abstracted into truth tables. Boolean
algebra was introduced as the mathematical foundation for manipulating the logic
expressions implemented by the switching networks. Logic gates abstracted away
the implementation of the switching networks themselves by leveraging Boolean
operators. We then talked about time and sequencing culminating in the design of
a sequential circuit that included a finite-state-machine and a data-path. These are
the same conceptual components of all digital computing devices. We will revisit
all of these abstractions in the chapters ahead.

We also briefly discussed the similarity between hardware and software and
the parallelism that we can achieve in logic circuits that we can’t achieve on our
general-purpose computing platforms. Finally, through two examples demonstrated
how starting from simple primitives such as logic gates and memory registers we
can construct such devices as our days-in-a-month calculator and a combination
lock. Hierarchy will be an important concept in design that enables us to tackle
larger and larger problems (we’ll see several large examples in chapters xxx). As
shown in Figure 1.27, everything we’ll build will be constructed from simple switching elements. Switches are used to construct our logic gates and registers. These, in turn, are used to construct combinational and sequential logic. Our last example used sequential and combinational logic in both its data-path and controller. Combinational logic appeared as a multiplexer, comparator, and the next-state computation of a finite-state-machine. Sequential logic appeared in the combination element memory elements and in the state registers of the finite-state-machine. The next layer in the hierarchy divided the logic according to function, whether it was a component of the data-path or the controller. Our combination lock system isn’t the end of the hierarchy chain, however. Systems like this one can be further composed into ever-larger ones.

**Summary**

Our hope is that this book will take you on an interesting and intellectually stimulating journey through the landscape of digital design. Before embarking on any journey, however, we usually collect guidebooks and maps that help us plan our route and read about the places we will visit. This chapter should serve as that guidebook to the rest of the text. We have used it to preview all the topics you will encounter in the following chapters. Necessarily, it only gave you a broad and superficial view. The journey itself, the visits to the later chapters and the direct experience of their material through exercises and laboratory assignments, will provide the depth and details.

**Further Reading**

**Exercises**