Inductors:

- Store energy in a magnetic field produced by current [DC-to-DC power supply]
- Generate magnetic fields [electromagnets]
- With a capacitor, make an LC resonator or filter [radio tuner, card key]
- Couple noninvasively to time-varying current [clip-on ammeter]
- Coupled to another inductor, change level of time-varying voltage [transformer]
- Inhibit flow of high-frequency current [RF choke in powerline communications]
- Model storage of kinetic energy in mechanical component [flywheel]

Transformers:

- Raise or lower time-varying voltages or currents [electric power delivery, automobile ignition coil, public address system speakers, switching power converters]
- Isolate AC circuits having different grounds [geographically spread-out measuring system]
- Interface between two circuits for optimum power transfer [coupling between TV antenna and coaxial cable]
- Model mechanical transmissions and electromechanical transducers

Diodes:

- Pass current preferentially in one direction [rectifier in AC-DC converter]
- Limit voltage to a certain range of values [clamp, regulator]
- Convert current flow to light [light-emitting diode]
- Convert light to flowing electric charge [solar cell, photodiode receiver for remote tuner]

Transistors:

- Amplify current or voltage [bipolar junction transistor, op-amp]
- Match an electrical source to an electrical load [microphone/speaker hookup]
- Convert a control voltage to a corresponding current flow [field-effect transistor]
- Convert light to flowing electric charge and amplify the current [phototransistor]
- Perform logic operations such as AND, OR, NOT, and NAND [NAND chip]
- Act as electrically controlled switch [infrared door alarm]

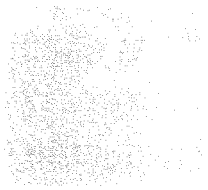
Op-amps:

- Amplify current or voltage [stereo preamplifier]
- Isolate circuits having different grounds [distributed measuring system]
- Process differential signals [reducing noise in communications gear]
- Analog computation [summing circuit]
- Active filter [hearing aid]
- Analog function inversion [logarithmic amplifier]

Other Components Whose Use is Obvious:

- Switch, lamp, microphone, speaker, socket, plug, and meter

Digital Logic Devices



Your digital wristwatch uses them to count seconds and display the time correctly. The telephone company uses them to route your call to the phone you dialed. Computers are full of them. What are they? *Digital logic devices* — electronic circuits that handle information encoded in binary form.

In the digital realm, the signals that are transmitted over wires consist of voltages or currents having only high or low values that represent a logical "**1**" or a logical "**0**". Special circuits have been developed for handling binary valued signals quickly and reliably.

22.1 Binary Number System

A student's question about multimeters: "What's better — analog or digital?"

This information may already be familiar to many readers. It is included to be sure that everyone is clear about the binary number system, which is the foundation upon which digital logic systems rest.

Our goal is to represent any number in terms of a sequence of just two symbols. We could use as the two symbols the words "true" and "false," a choice that relates what we're doing to human logic. Since we're going to deal with electronic circuits, we could choose two voltage levels — "high" and "low." But the conventional choice, which we adopt here, is to use the symbols for one and zero — **1** and **0**. We use boldface type to indicate that these are different from the ordinary numbers one and zero.

Before talking about binary numbers, let's consider what is really meant when we write a number in the familiar decimal system. A number such as 340, for example, really means that the value represented is the sum of "three one-hundreds" and "four tens." We can generalize this to say that when we write a decimal number JKLM it stands for the following expression written using successive powers of ten:

$$JKLM = J \times 10^3 + K \times 10^2 + L \times 10^1 + M \times 10^0 = J \times 1000 + K \times 100 + L \times 10 + M$$

× 1. In the decimal number system, the multipliers such as J, K, and M can have any one of 10 values, from 0 to 9.

Let's now consider the binary number system where the binary digits — the binary multipliers **J**, **K**, and so on, represented in boldface type — can only have one of two possible values, binary one (**1**) or binary zero (**0**). (Incidentally, binary digits are usually referred to by the contraction "bit.") The value of a binary number **JKLM** is the sum of successive powers of two: $\mathbf{JKLM} = \mathbf{J} \times 2^3 + \mathbf{K} \times 2^2 + \mathbf{L} \times 2^1 + \mathbf{M} \times 2^0 = \mathbf{J} \times 8 + \mathbf{K} \times 4 + \mathbf{L} \times 2 + \mathbf{M} \times 1$. Thus the binary number **101** can be written out more explicitly as $\mathbf{1} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0 = 4 + 0 + 1 = 5$. Similarly, the binary number **11100** is equivalent to decimal 28, and the binary number **11101** is equivalent to decimal 29.

To represent in the binary system numbers that have a fractional part, we can put in a period (called a binary point) and continue the sequence of **1**s and **0**s to represent negative powers of two. For example, the decimal number 3.5 would be **11.1** in binary, and the decimal number 5.25 would be **101.01** in binary.

22.2 Converting a Decimal Number to its Binary Equivalent

Table 22.1 Example of a decimal-to-binary conversion

Digits	Bits
340	**0**
170	**0**
85	**1**
42	**0**
21	**1**
10	**0**
5	**1**
2	**0**
1	**1**

A surefire procedure for converting a number from decimal to binary form is to divide the decimal number by the number 2 repeatedly, each time writing down the remainder (either a **0** or a **1**) in the next most significant position.

Exercise 22.1 Find the binary equivalent of the decimal number 340.

Solution: See Table 22.1. Divide 340 by two to get 170 with remainder **0**. Divide 170 by two to get 85 with remainder **0**. Divide 85 by two to get 42 with remainder 1. Continuing the process, we finally obtain the result that 340 = **101010100**.

It is also possible to generate the bits in the opposite order by subtracting successive powers of two, as presented in Exercise 22.2, starting with the largest power of two less than or equal to the number. For each power of two that can be subtracted, a binary one is placed in the answer at the corresponding bit position.

Exercise 22.2 Find the binary equivalent of the decimal number 522 using the subtractive method. The powers of two are presented in Table 22.2.

Solution: Locate the largest power of two that is less than or equal to 522, that is, 512. Since $512 = 2^9$, or **100000000**, write down the binary result as **1**[ ][ ][ ][ ][ ][ ][ ][ ] (a one followed by 8 placeholders), since we don't yet know the values of the other bits. Subtract the 512 from the 522, leaving 10. Continue by finding a power of two that is less than or equal to 10, that is, 8. The binary result becomes **10001**[ ][ ]. Notice that we fill in the intervening placeholders with **0**s. When we subtract, we are left with 2. This being a power of two, we can add the appropriate bit and fill in the remaining position(s) with **0**(s). The binary result becomes **100001010**.

"How long have people been using binary?"

Table 22.2 Powers of 2

N	2^N	Comments
0	1	
1	2	
2	4	
3	8	
4	16	
5	32	
6	64	
7	128	
8	256	
9	512	
10	1,024	$2^{10} = 1024$ is the closest power of 2 to the decimal number 1000, so the prefix kilo has been used when labeling binary quantities such as 64 kilobytes (64kB) $= 64 \times 10^{24}$B = 65,536B. A new (1998) standard attempts to correct the inaccuracy of this by creating the prefix Kibi (Ki) (from kilo binary) For more on this standard see [a]
11	2,048	
12	4,096	
13	8,192	
14	16,384	
15	32,768	$2^{15} = 32768$ is often used as the frequency in Hz of a digital wristwatch's clock crystal — the master timing source for the watch. If you successively half the frequency of the electrical output from the clock crystal 15 times, you get an electrical pulse exactly once per second.
16	65,536	
17	131,072	
18	262,144	
19	524,288	
20	1,048,576	$2^{20} = 1,048,576$ is close to one million, so this number is similarly denoted as Mebi (Mi), when describing the size of a computer file or memory.
30	1,073,741,824	Similarly, denoted Gibi (Gi)
32	4,294,967,296	Size of the address space of most 32-bit computers = 4GiB

[a] http://physics.nist.gov/cuu/Units/binary.html

22.3 Logic Operations and Truth Tables

We can design circuits for doing arithmetic operations with binary digits by combining elementary elements called *logic gates*. Each gate performs a single logic operation, and each gate can be realized by combining electronic components such as transistors, resistors, and voltage sources. The operations required to perform a computation in a digital computer are carried out by circuits containing the logic gates that we will now describe. For example, a binary adder may be used in doing addi-

tion that is called for in arithmetic manipulations of data input to a computer. A different application of a binary adder is to increment a counter by one unit each time a programmed sequence of operations is carried out.

The seven basic logic gates are identified and characterized in Table 22.3. In each row, the truth table (which is a list of the output that the gate produces for every possible combination of its inputs) that characterizes the gate is on the left-hand side. The name of the logical operation appears in the leftmost column. Two schematic symbols for the gate are shown along with boolean algebra formulas for the gate function. In each figure, the schematic symbol on the left is the standard representation, and the one on the right is the so-called DeMorgan equivalent representation. (See Section 22.5.) Each DeMorgan equivalent functions logically and represents the same circuit as the gate to its left. The equivalents, which are a different way of thinking about these gates, may be helpful when finding the minimum number of transistors required to realize a logic function. (See Figures 27.3 through 27.6 and Problem 22.8).

While we show gates with no more than two inputs here, gates having more inputs are frequently used. Variables or formulas with an overline indicate inversion and are read as "bar"; for example, $\overline{A}$ is read as "A bar," which means "not-A," the complement of the binary variable A. For ease in writing with a word processor, the symbols !A, /A, A! or A′ may be used instead.

Table 22.3 Logic gates

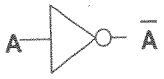
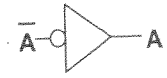
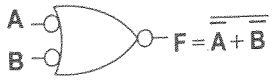
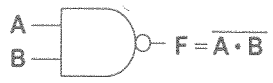
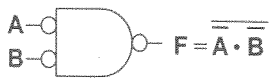
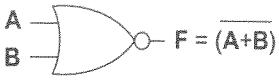
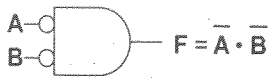
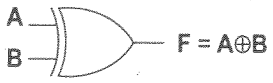
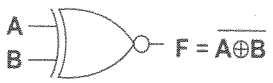
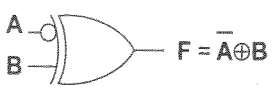
Logic Gate	Truth Tables			Schematic Symbols: Standard Symbol	Schematic Symbols: DeMorgan's Equivalent
	Inputs		Output		
NOT or Inverter	A		$\overline{A}$		
	0		1		
	1		0		
AND	A	B	A·B		
	0	0	0		
	0	1	0		
	1	0	0		
	1	1	1		

Table 22.3 Logic gates (continued)

Logic Gate	Truth Tables			Schematic Symbols: Standard Symbol	Schematic Symbols: DeMorgan's Equivalent
	Inputs		Output		
NAND	A	B	$\overline{A \cdot B}$		
	0	0	1		
	0	1	1		
	1	0	1		
	1	1	0		
OR	A	B	A+B		
	0	0	0		
	0	1	1		
	1	0	1		
	1	1	1		
NOR	A	B	(A+B)		
	0	0	1		
	0	1	0		
	1	0	0		
	1	1	0		
XOR	A	B	A⊕B		
	0	0	0		
	0	1	1		
	1	0	1		
	1	1	0		
XNOR or NXOR or EQUAL	A	B	$\overline{A \oplus B}$		
	0	0	1		
	0	1	0		
	1	0	0		
	1	1	1		

The NOT gate is also known as an inverter, since its output is the opposite of its input: If the input is a **1**, the output is a **0**, and vice versa. The AND gate has the property that its output is a **1** if both of its inputs are **1**s. The NAND gate's output is the negation of the output of the AND gate; the small circle to the right of the gate symbol indicates that negation. The OR gate's output is a **1** if either of its inputs is a **1**. The XOR gate is more explicitly known as the "exclusive-OR" gate: Its output is a **1** if either, but not both, of its inputs is a **1**. We will see shortly how to use a simple network of these gates to construct a circuit that will add binary numbers, but first we will show a procedure for constructing a logic gate array that will produce an arbitrarily chosen set of outputs when presented with all possible binary inputs.

22.4 Logic Gate Array that Produces an Arbitrarily Chosen Output

Suppose that our task is to design a logic gate array that has three binary inputs — labelled **A**, **B** and **C** — and that produces some particular set of outputs, **F**. The truth table that we'll take as an example is shown in Table 22.4. (The output values were just picked out of thin air and don't represent anything in particular.)

Note that we've written the binary inputs on the left in a very regular way: if regarded as being three-digit binary numbers, their eight values would increase in steps of one from zero at the top to seven at the bottom. Writing them in this structured way helps us avoid leaving out any possible values.

Here's a set of steps that is guaranteed to produce a logic gate array that behaves the same as the truth table that you start with:

Table 22.4 What circuit of logic gates could produce these (arbitrarily chosen) outputs F in response to inputs A, B and C? See text for method and Figure 22.1 for result.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Step 1: Use an OR gate as the output gate. Provide this OR gate with as many inputs as there are **1**s in the output (the F column). In our example, the output contains four **1**s, and so the output OR gate must have four inputs, corresponding to the third, fourth, sixth, and eighth rows in the truth table above.

Step 2: Set up a general-purpose circuit on the input side, at the left, that will take in the inputs (**A**, **B**, and **C**) and that will provide internally both the inputs (**A**, **B**, and **C**) and the negations of the inputs ($\overline{\mathbf{A}}$, $\overline{\mathbf{B}}$, and $\overline{\mathbf{C}}$) in easily accessed columns. NOT gates are used to form the negations ($\overline{\mathbf{A}}$, $\overline{\mathbf{B}}$, and $\overline{\mathbf{C}}$) from the inputs (**A**, **B**, and **C**).

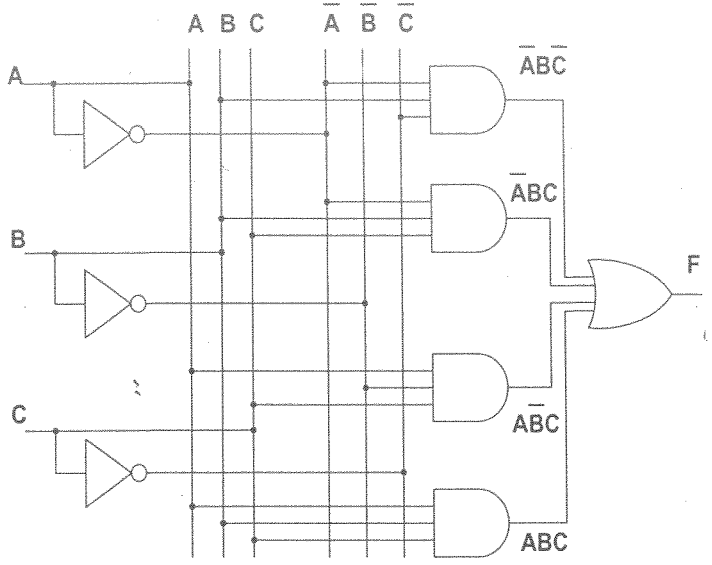


Figure 22.1 Example of general purpose circuit to implement the truth table of Table 22.4. (This solution is NOT minimized.)

Step 3: Use an AND gate with as many inputs as the circuit (three in this case) to compute each row of the truth table that produces a **1**. Connect the output of each AND gate to one of the inputs of the OR gate. For example, the first set of inputs that would produce a **1** output is the third row of the arbitrarily chosen truth table. To get a **1** output from the OR gate when the inputs A, B, and C have values **0**, **1**, and **0** (the third row of the truth table), we would connect $\overline{\text{A}}$, B, and $\overline{\text{C}}$ to a three-input AND gate as shown in Figure 22.1.

22.5 Boolean Algebra

The expression of logic formulas in a notation such as:

$$\mathbf{F} = \overline{\mathbf{A}}\mathbf{B}\overline{\mathbf{C}} + \overline{\mathbf{A}}\mathbf{B}\mathbf{C} + \mathbf{A}\overline{\mathbf{B}}\mathbf{C} + \mathbf{A}\mathbf{B}\mathbf{C}$$

allows the manipulation and simplification of such formulas using the rules of boolean algebra.

The rules of boolean algebra are listed in Table 22.5.

Table 22.5 Rules of boolean algebra. The two entries in the last row are used frequently and are known as DeMorgan's theorem.

AND Rules	OR Rules
$A \cdot A = A$	$A+A = A$
$A \cdot \overline{A} = 0$	$A+\overline{A} = 1$
$0 \cdot A = 0$	$0+A = A$
$1 \cdot A = A$	$1+A = 1$
$A \cdot B = B \cdot A$	$A+B = B+A$
$A(BC) = (AB)C$	$A+(B+C) = (A+B)+C$
$A(B+C) = AB+AC$	$A+BC = (A+B)(A+C)$
$\overline{A \cdot B} = \overline{A}+\overline{B}$	$\overline{A+B} = \overline{A} \cdot \overline{B}$

Example 22.3 Using these rules we can take a number of steps to successively simplify the expression as follows:

$$F = \overline{A}\overline{B}\overline{C} + \overline{A}BC + A\overline{B}\overline{C} + ABC$$

$$F = (\overline{A}\overline{B}\overline{C} + \overline{A}BC) + (A\overline{B}\overline{C} + ABC)$$

$$F = \overline{A}\overline{B}(\overline{C}+C) + AC(\overline{B}+B)$$

$$F = \overline{A}\overline{B}(1) + AC(1)$$

Finally,

$$F = \overline{A}\overline{B} + AC$$

The final expression is clearly simpler than our initial expression.

22.6 Adding Binary Numbers

We can develop a procedure for adding numbers written in binary form by observing the following basics:

Binary: **0 + 0 = 0** (Equivalent to Decimal: 0 + 0 = 0),

Binary: **1 + 0 = 1** (Equivalent to Decimal: 1 + 0 = 1),

Binary: **1 + 1 = 10** (Equivalent to Decimal: 1 + 1 = 2).

The last binary result could be written out more formally by using two binary digits to express the numbers being added and the result: **01 + 01 = 10**. In words, the sum of the two right-hand columns is two, which is represented in binary form as **10**. We can also say that the two right-hand columns when added generate a sum digit (**0**) and a carry digit (**1**).

Table 22.6 Truth table for a half adder

Inputs		Outputs	
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

We express this addition, called a "half-adder," as a truth table in Table 22.6. If we then observe that the carry column is equivalent to the AND operation and the sum is equivalent to the XOR operation, we could build the half-adder in Figure 22.2.

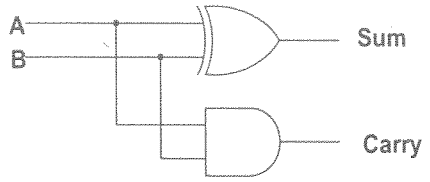


Figure 22.2 Half-adder circuit

Example 22.4 Determine the sum of the following binary numbers: **1101100** and **1011010** (decimal numbers 108 and 90, respectively).

Solution: Writing this out in traditional grade school form and indicating carry digits at the head of the column, we have

```
carry digits --->   11110000
                    1101100
                +   1011010
                    --------
                    11000110
```

Checking: The answer is 198, which is the sum of the numbers 108 and 90.

Table 22.7 Truth table for a full adder

Inputs			Outputs	
A	B	C	Carry	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder

We can see from the exercise that a practical adder circuit to implement the addition of one column must add not two but three bits. (This explains why the previous circuit is called a half-adder.) Let's write out the truth table for a full adder. (See Table 22.7). We write the eight possible variations of the input variables by counting in binary. Next, we figure out the sum and carry bits by simply looking at how many input bits are **1** in each row. If in a given row there's only one input that is a logical one, then for that row, we have a sum bit but no carry bit. A row containing two logical ones will have a carry bit, but no sum bit, and a row having three logical ones produces both a sum and a carry bit.

The construction of a full-adder logic circuit is left as a problem. (See Problem 22.6.) You can actually build the circuit in the lab and verify that it operates correctly. You can also simulate the adder with a computer program called LogicWorks™ to verify your design before actually building it.

We can abstract the actual circuit for the full adder into a schematic block with the appropriate inputs and outputs shown in Figure 22.3.

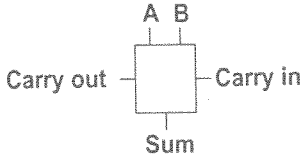


Figure 22.3 Schematic block representing a full adder

If we want to build an eight-bit adder circuit, we can connect full adders as shown in Figure 22.4. Note that the carry bits flow from right to left.

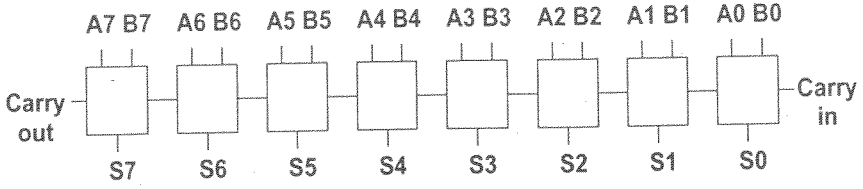


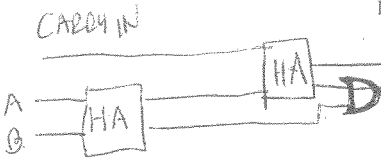
Figure 22.4 Circuit for adding two eight-bit binary numbers

Negative Numbers and Subtraction

How does one represent negative numbers in a binary system? They are usually represented in twos-complement notation. To form a negative number in twos-complement notation, first complement the bits (change all the zeros to ones and the ones to zeros) and then add one. Try this on numbers such as 0, 1, and 2 to see how the procedure works. To make the distinction between the positive and negative numbers, the most significant bit can be reserved to indicate the sign. This means that if an 8-bit quantity is being treated as a signed number, then the range becomes -128 to 127 instead of 0 to 255.

Exercise 22.5 Add a negative and a positive number together. Let's use negative five and positive three.

Solution: First, to form the negative number, take five (**00000101**), complement it to **11111010**, and add one to get **11111011**. Next, let's add the three (**00000011**):



carry digits —> 00000110
11111011
\+ 00000011
11111110

Finally, we see that the most significant bit of the answer (the leftmost bit) is a one, so the answer must be negative. To find its value, we complement the answer to get **00000001** and add one to get **00000010**, or 2. The answer is -2.

We can also use the circuit shown in Figure 22.4 to subtract two binary numbers, if we set the carry-in to a **1** and complement the subtrahend.

22.7 Memory Elements

So far, we have been discussing only combinatorial logic circuits, whose outputs are strictly a function of the current inputs. We can combine two logic gates as shown in Figure 22.5 using feedback and create an element with *memory* (i.e., whose outputs depend not only of the current inputs, but also on some previous input.) The cross-

$\bar{S}$	$\bar{R}$	Q	$\bar{Q}$	Comment
0	0	1	1	Weird State
0	1	1	0	Set
1	0	0	1	Reset
1	1	Q	$\bar{Q}$	Hold

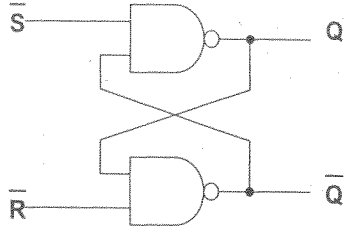


Figure 22.5 R–S latch (right) and its state table (left).

coupled NAND gates shown form an R–S latch (Reset–Set). The inputs of this particular version are marked with bars above the name to indicate a logical inversion of the input.

Exercise 22.6 Verify the operation of the R–S latch shown in Figure 22.5.

Solution: Assume that $\bar{S}$ is **0** and that $\bar{R}$ is **1**. The output of the upper NAND gate with a zero input is **1**. Then, both inputs to the lower NAND gate are **1**, resulting in a **0** output. Now, let **S** become **1**. Since the **Q** output is **0**, the **Q** output remains at **1**, holding the state. Now assume that **S** is **1** and that **R** is **0**. The output of the lower NAND gate with a zero input is **1**. Now, both inputs to the upper NAND gate are **1**,

resulting in a 0 output. Let $\overline{R}$ become 1. Since the Q output is 0, the $\overline{Q}$ output remains at 1, again holding the state. Finally, we can check the undesired condition of having both set and reset signals applied simultaneously, $S = R = 0$. In this case, both NAND gates find a 0 at one input and the outputs will both be 1.

R–S latches can also be built by cross-coupling NOR gates, in which case the inputs are non-inverted (normal).

Another type of memory device, a transparent latch, can be built with a few more gates, as shown in Figure 22.6. When the enable input is true, the value on the D input is reflected to the Q output (the device is transparent), and when the enable is false, the values on the outputs are held.

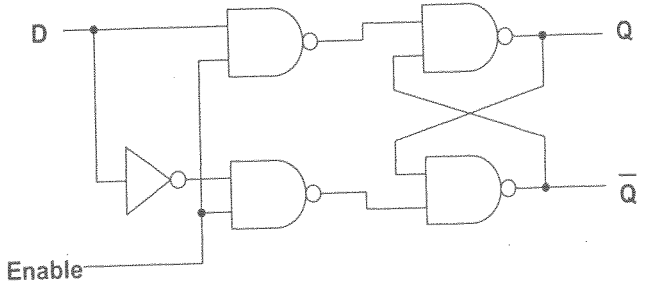


Figure 22.6 Transparent latch circuit

Two of these devices can be combined to form a *master–slave flip-flop*. (See Figure 22.7).) The left half (the master) passes data through to the second stage

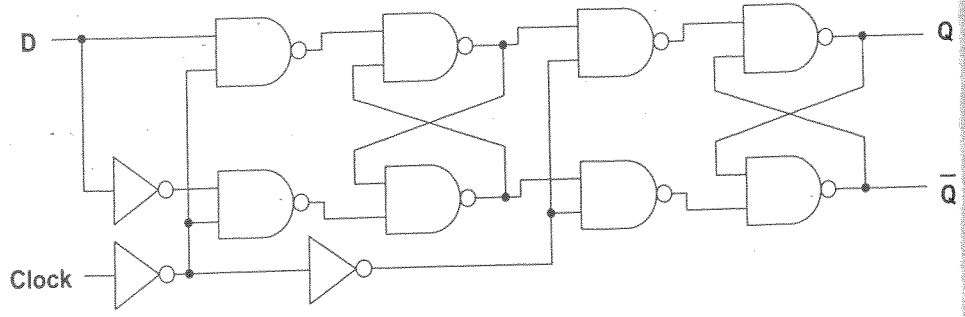


Figure 22.7 A master–slave flip-flop

while the clock is low and holds the value while the clock is high. The right half (the slave) passes the data from the first stage when the clock goes high and continues to hold the data while the clock is low. This behavior transfers the data from the D input to the Q outputs at the rising edge of the clock signal. This is one way to build devices known as *D flip-flops*, which we use in sequential logic. A schematic symbol for a D flip-flop is shown in Figure 22.8. An input marked with a triangle is an edge

sensitive input, meaning that it will activate the flip-flop on the rising edge of a signal that is connected to it.

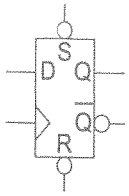


Figure 22.8 Schematic symbol for a D flip-flop with set (S) and reset (R) inputs

22.8 Sequential Logic

Circuits that employ memory, in the form of latches, flip-flops, or RAM, in addition to combinatorial logic, are called *sequential* logic. There are many kinds, such as self-timed, asynchronous, and synchronous. We will discuss a style of synchronous logic known as the *state machine*.

In a state machine (see Figure 22.9), the "state" information is contained in a memory device, such as a set of flip-flops. A clock provides periodic pulses to the memory to store a new state value. Combinatorial logic is used to combine the external inputs and the current state to form the new state and any outputs that the system needs.

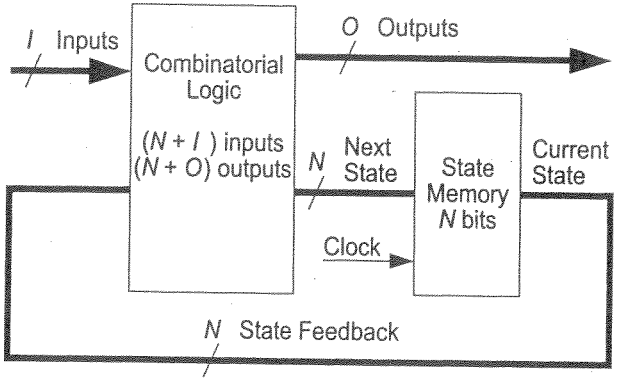


Figure 22.9 Schematic of a generalized state machine. Each bold line is a collection of wires known as a bus. The slash across the bus is labeled with the number of wires in the bus. Since this is a generalized schematic, these are labelled with variable names — *N*, *I*, and *O*.

The behavior of a state machine may be expressed in a *state diagram*, which uses circles to represent states and directed arrows to show the allowed transitions between states, the stimulus that would cause the transition, and any actions that

would result from the transition. States may have descriptive names or just representative letters.

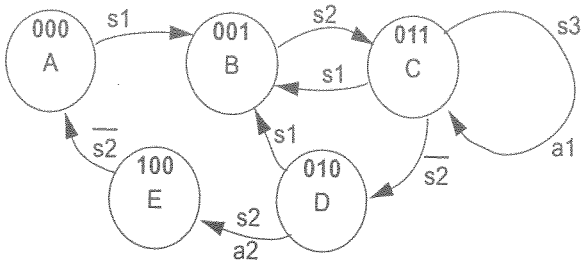


Figure 22.10 A simple state diagram with five states (A–E), three stimuli (s1–s3), and two actions (a1 and a2).

Each stimulus may correspond with any combination of external inputs, and each action may specify some set of outputs. The state is used in combination with the stimulus to form the new state. In the process of converting this sort of free-form state diagram into a logic schematic, you may want to organize the information into a state table that would have one row for each transition. The state will be represented by a set of N bits such that the number of states is less than 2^N. For the simple state diagram of Figure 22.10, the number of states is five, requiring three bits for the state memory.

We could number the states in an arbitrary fashion, but that might result in an unnecessarily complicated design. Instead we will assign binary numbers to the states so as to minimize the number of bit transitions between states in the entire graph; this will have the effect of minimizing the number of required logic gates. We can take a clue from the existence of the so-called *gray code*, in which only one bit changes on going from one state to the next. An example of a three-bit binary gray code is **000, 001, 011, 010, 110, 111, 101, 100**. We have used this concept in numbering the states in Fig. 22.10. (For more on the gray code, see Problem 22.10.)

Elevator Control Circuit

We'll design part of a simple elevator control circuit for a four-floor, two-shaft elevator. The top and bottom floors will have one call button and the other floors will have two directional call buttons in the hallway. Each car will have a button for each floor and an emergency (E-stop) button. We will design the circuits in small pieces using the idea of coupling several small state machines together.

First, we'll draw the state diagrams of the cars and the call buttons. The diagram for the door's state machine is left to the reader.

Next we'll number the states for the lift state diagram in the center of Figure 22.11. For the lift state diagram, there are five transitions. We can make four of them be one-bit transitions by labeling them as shown in Table 22.8. We will typically number as zero the state that we want the machine to "wake up" in. For the Lift State, that would be "Stop." For the Button States, that would be "Not Lit," and for the Car-Direction State, it doesn't matter, but we'll choose "Up."

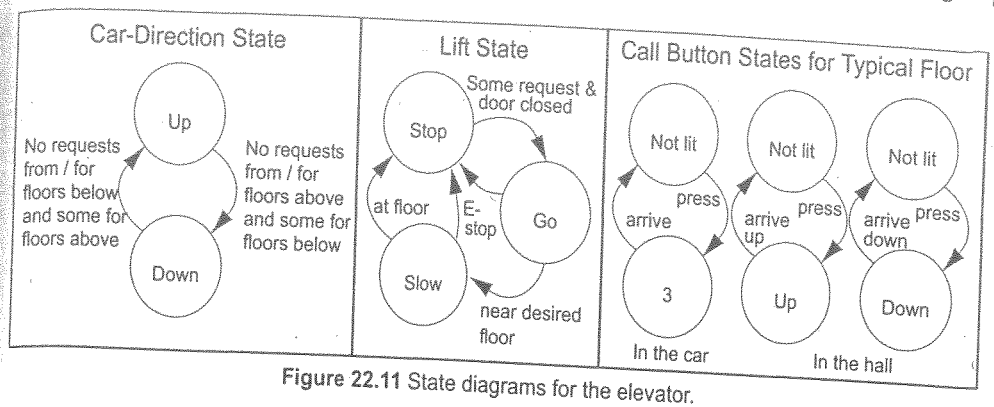


Figure 22.11 State diagrams for the elevator.

We create the state table of Table 22.8 by listing each transition as a row in the table. The columns on the left side are the inputs to our combinatorial logic, made up of state and input signals. The columns on the right are the resulting new states and the outputs. Note that outputs can be transient, occurring only during the transition or continuous, tied to one or more states. In this example, the motor and slow outputs can be tied to the states, while the door opening signal is transient. We simplify the table by listing only the transient outputs.

After we have put all the transitions from the state diagram into our table, we

Table 22.8 Lift State table for elevator

State		Inputs					Next State		Outputs
Name	Code Q_1Q_0	Door Closed C	Request R	Near Floor N	At floor A	E-Stop E	Code D_1D_0	XOR X_1X_0	Open Door D
Stop	00	1	1	x	1	0	00	00	1
	00	1	1	x	0	0	01	01	0
	00	x	x	x	1	1	00	00	1
Go	01	1	1	x	x	1	00	01	0
	01	x	x	1	0	0	11	10	0
Slow	11	x	x	x	0	1	00	11	0
	11	x	x	x	1	x	00	11	1

will study the inputs in each state to see if safety or other design concerns dictate that we add any more rows to the table. For instance, the first row was added to the table to handle the situation that arises when a button is pressed and the car is already at the desired floor. The **x**s in the table indicate "don't care" conditions. These mean that the input value doesn't matter and result in simplifications of the logic. In our table, we have five input variables, which can have 32 possible combined values. A full expression of all possible inputs would result in a table with 32 rows for each state.

Since we are basically recomputing the state with each tick of our system clock, we must either have one or more entries in our table to maintain each current state or adopt a style of logic in which the state is automatically maintained until we

specifically change it. While the latter approach initially looks more complicated, it can result in simpler designs. To implement this style, we'll connect the output of an XOR gate to the input to each D-flip-flop, with one input of the XOR connected to the Q-output of the flip-flop. This leaves us with one input to the XOR gate. That input will cause the flip-flop to remain in its current state as long as it is **0** and will cause the state to change if it is a **1**. As a result of this, we need to add a column to our state table that is the changes in the state bits. We can then extract the logic equations that we'll implement with gates in Figure 22.12:

$$X_1 = \overline{Q}_1 Q_0 N \overline{A}\,\overline{E} + Q_1 Q_0 \overline{A} E + Q_1 Q_0 A;$$

$$X_0 = \overline{Q}_1 \overline{Q}_0 C R \overline{A}\,\overline{E} + \overline{Q}_1 Q_0 C R E + Q_1 Q_0 \overline{A} E + Q_1 Q_0 A;$$

$$D = \overline{Q}_1 \overline{Q}_0 C R A \overline{E} + \overline{Q}_1 \overline{Q}_0 A E + Q_1 Q_0 A.$$

The realization of these functions appears in the figure. Note that only one of the OR gates has four inputs and that the other two have just three.

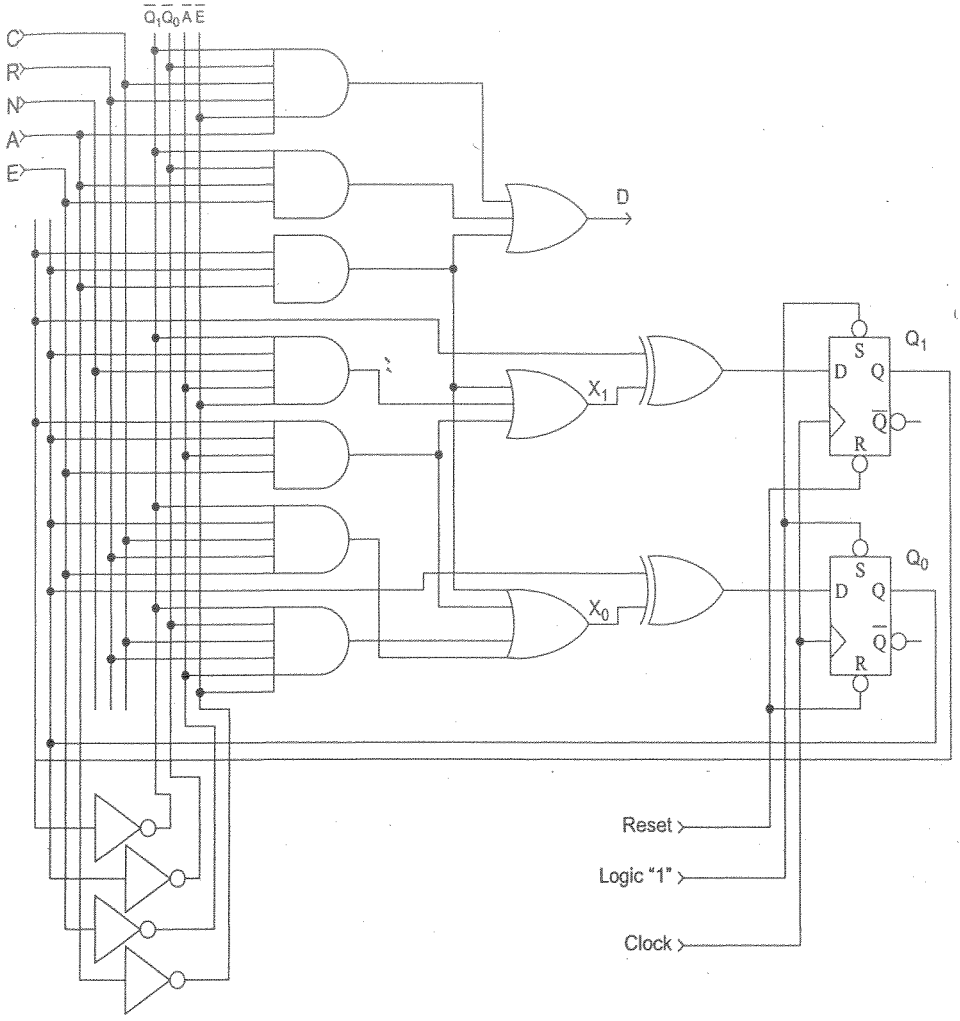


Figure 22.12 Elevator Lift State Machine schematic

22.9 Problems

Note: In the problems below, the asterisk (*) identifies particularly challenging problems.

22.1 To what base-10 number is each of the following binary numbers equivalent?
(a) **1100100**
(b) **1111.111**

22.2 Convert the three following base-10 numbers to binary numbers: 5, 87, and 20.125.

22.3 Find the sums of the following pairs of binary numbers, and check your answers in the base-10 system:
(a) **01111** and **00001**
(b) **10001** and **01010**

22.4 Find the sum of the following binary numbers, and check your answer in the base-10 system: **11101**, **11011**, and **11111**.

22.5 Multiply the following binary numbers, and check your answers in the base-10 system:
(a) **10101** and **00001**
(b) **10101** and **00010**
(c) **11111** and **00101**

22.6 Using a program such as LogicWorks™, simulate a full adder composed of only AND, OR, and NOT gates. Turn in a printout of the network of gates you used and the timing diagram of the circuit's response at significant points in the circuit as you step through the values of the two input variables.

22.7 Using a program such as LogicWorks™, simulate a synchronous counter, made of interconnected gates and flip-flops, that will count from 0 to 15 clock pulses.

22.8 Consider the full adder for three-level logic that you designed and simulated in Problem 22.6. This adder employed three inverters, eight AND gates, and two OR gates.
(a) Convert the design so that it uses only NAND gates (instead of AND and OR), which are simpler to fabricate in integrated circuit form. You will need to keep the three inverters.
(b) Use the rules of boolean algebra (Table 22.5) to optimize the NAND-gate design of the carry circuit so that it uses a minimum number of transistors. Helpful information: the optimized carry design will contain 4 NAND gates, having different numbers of inputs.
(c) Determine the number of field-effect transistors required in the optimized CMOS full-adder design.

22.9 *Multiplier. Given a full adder (FA) block (Figure 22.3), design a multiplier to multiply 8 bits by 8 bits, yielding a 16-bit product using FAs and AND gates.

22.10 Design a three-bit gray-code counter. A "gray" code is one in which only one bit changes with each count. An example of this is: **000 001 011 010 110 111 101 100 000**. Such sequences are useful for shaft encoders where a number of bits must be reported out, and any misalignment between bits might result in an erroneous reading if a normal code were used.
(a) Write state table and corresponding boolean expressions for D_2, D_1, and D_0 in terms of Q_0, Q_1, and Q_2 and their inverses.
(b) Reduce the expression for D_0 using boolean algebra to show that:

$$D_0 = \overline{Q}_2\overline{Q}_1 + Q_2Q_1.$$

22.11 *Continuing from Problem 22.10, reduce the expression for D_1 using boolean algebra, to show that:

$$D_1 = \overline{Q}_2(\overline{\overline{Q}_0\overline{Q}_1}) + Q_1\overline{Q}_0.$$

(Hint: use $A = A + A$.)

22.12 Draw the schematic for the gray-code state machine, given that

$$D_2 = Q_1\overline{Q}_0 + Q_2(\overline{\overline{Q}_0\overline{Q}_1}).$$

You may use LogicWorks™ or a similar program to produce a neat schematic and to allow you to verify its operation.