8 Digital circuits and devices

8.1 Introduction

In analog electronics, voltage is a continuous variable. This is useful because most physical quantities we encounter are continuous: sound levels, light intensity, temperature, pressure, etc.[1] Digital electronics, in contrast, is characterized by only two distinguishable voltages. These two states are called by various names: on/off, true/false, high/low, and 1/0. In practice, these two states are defined by the circuit voltage being above or below a certain value. For example, in TTL logic circuits, a high state corresponds to a voltage above 2.0 V, while a low state is defined as a voltage below 0.8 V.[2]

The virtue of this system is illustrated in Fig. 8.1. We plot the voltage level versus time for some electronic signal. If this was part of an analog circuit, we would say that the voltage was averaging about 3 V, but that it had, roughly, a 20% noise level, rather large for most applications and thus unacceptable. For a TTL digital circuit, however, this signal is always above 2.0 V and is thus always in the high state. There is no uncertainty about the digital state of this voltage, so the digital signal has zero noise. This is the primary advantage of digital electronics: it is relatively immune to the noise that is ubiquitous in electronic circuits. Of course, if the fluctuations in Fig. 8.1 became so large that the voltage dipped below 2.0 V, then even a digital circuit would have problems.

8.2 Binary numbers

Although digital circuits have excellent noise immunity, they also are limited to producing only two levels. This does not appear to be very helpful in representing the continuous signals we so frequently encounter. The solution starts with the

[1] This holds for most macroscopic quantities. On the atomic level, many physical quantities are quantized.
[2] If the voltage is between these thresholds, we say the state is undetermined, which means the circuit behavior cannot be insured.

Table 8.1 The first twelve counting numbers in binary

Base 10	Base 2	Base 10	Base 2
0	0	6	110
1	1	7	111
2	10	8	1000
3	11	9	1001
4	100	10	1010
5	101	11	1011

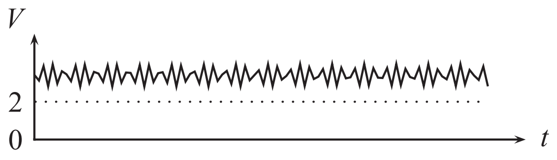


Figure 8.1 A noisy analog signal is noise-free in digital.

realization that we can represent a signal level by a number that only uses two digits. For these *binary numbers*, the two digits used are 0 and 1. Binary numbers are also call *base 2* numbers, and can be understood by abstracting the rules we all know for the numbers we commonly use (base 10 numbers). When we write down a base 10 number, each digit can have 10 possible values, 0 to 9, and each digit corresponds to 10 raised to a power. For example, when I write 1024_{10}, this is equal to

$$1024_{10} = 1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 \tag{8.1}$$

where we use the subscript 10 on 1024 to make explicit the base of the number.

Analogously, for binary numbers, each digit can have only 2 possible values, 0 or 1, and each digit of the number corresponds to 2 raised to a power. Thus

$$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}. \tag{8.2}$$

The first twelve base 10 numbers and their binary equivalents are given in Table 8.1.

In a similar manner, the rules for base 10 addition and subtraction can be mapped over to binary arithmetic. Some examples are shown in Table 8.2. In base 10, when we add the rightmost column, 9 plus 5 equals 14. Since this result cannot be expressed in a single digit with the ten available digits (0 to 9), we write down the 4 and carry the 1 to the next column. Similarly, when we add the 1 and 1 of the rightmost column of the binary number, we get 10_2. Since this cannot be expressed

Table 8.2 Adding and subtracting in binary

Addition		Subtraction	
Base 10	Base 2	Base 10	Base 2
9	1001	14	1110
+15	+1111	−9	−1001
24	11000	5	101

in a single binary digit, we write down the 0 and carry the 1 to the next column. We repeat the process as we work through the columns from right to left.

In the base 10 subtraction, we try to take 9 from 4, but need to borrow from the next column to the left. This gives us 14 which allows the subtraction to proceed, but we must remember to decrease the number in the 10^1 column by one. In like manner, for the binary number, we try to take one from zero in the first column. To progress, we need to borrow from the next column (the 2^1 column), carefully decreasing that column's digit by one. We then continue as with the base 10 number until we reach the leftmost column.

Note from Eq. (8.2) and Tables 8.1 and 8.2 that binary numbers tend to be long (i.e., have many digits) compared to base 10 numbers. As we will see, this has a direct impact on the complexity of digital circuits.

8.3 Representing binary numbers in a circuit

In the last section we saw that numbers can be expressed in base 2 just as they can in the more familiar base 10. Base 2 is particularly suitable for expressing a number digitally since digital electronics has only two levels, high and low, and these can be taken to represent the two digits (1 and 0) in a binary number. But a binary number typically consists of several digits.[3] How can we express all of these digits electronically? There are two basic methods, known as *parallel* and *serial* representation.

In parallel expression of a binary number, each digit or bit of the number is represented *simultaneously* by a voltage in the circuit. This is represented schematically in Fig. 8.2. Each output line has a high or low voltage relative to ground. These lines are assigned to represent a particular bit of the number. In this example,

[3] A *b*inary dig*it* is often referred to as a *bit*.

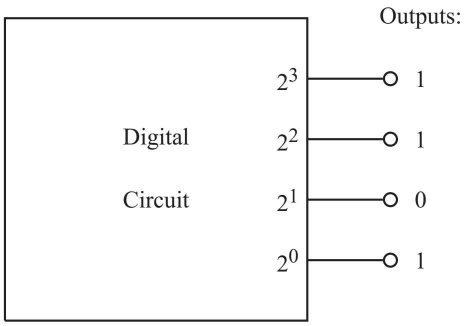


Figure 8.2 Parallel representation of a four-bit number 1101.

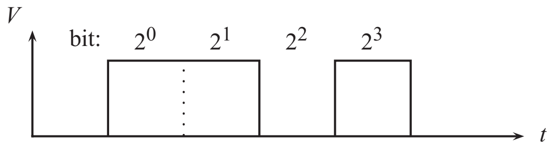


Figure 8.3 Serial representation of a four-bit number 1011.

the bottom line represents the 2^0 bit, the next line up represents the 2^1 bit, and so on. Because we have four independent lines, the entire four-bit number can be expressed at a point in time, so parallel communication of information is very fast. The price we pay for this speed is the increased number of lines in our circuit. The more precision we want in our number, the more significant figures we need, and the more lines are required.

An alternative way of expressing a binary number is by a serial representation. In this method, the various bits are communicated by sending a *time sequence* of high/low voltage levels on a single line. An example of this is shown in Fig. 8.3. The plot shows the voltage level on a serial line. The voltage switches between high and low levels, with each level lasting for a certain time interval. The first interval corresponds to the 2^0 bit of our number, the next interval represents the 2^1 bit, and so on. We are thus able to communicate the binary number on a single line, rather than the multiple lines required for parallel communications, but the communication is no longer instantaneous; we must wait for several intervals before we receive all the bits of our transmitted number.

In order for serial transmission of information to work, both the sender and receiver need to agree about several things. Some of these are: (1) how many bits of data are going to be sent, (2) what digital level (high or low) corresponds to the 1 bit, (3) what is the time interval between bits, and (4) how will the start of a number be recognized?

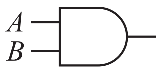
8.4 Logic gates

The basic circuit element for manipulating digital signals is the *logic gate*. There are several types of logic gate, and each performs a particular logical operation on the input signals. The logical operation of the gate is defined by its *truth table* which gives the output state for all possible combinations of the inputs.

The first logic gate we will consider is the AND gate. The output of an AND gate is high only when all of the inputs are high. Because this definition is clear for any number of inputs, this type of gate can, in principle, have as many inputs as you like. In Fig. 8.4 we show a two-input AND gate along with its truth table. As we will see below, there is a special algebra, called *Boolean algebra*, for logic operations. The symbolic representation of the AND operation for two inputs A and B is $A \cdot B$, pronounced "A and B" or "A ANDed with B." Note that it is *not* "A times B."

The output of an OR gate is high when any input is high. Again, this operation is defined for any number of inputs. A two-input OR gate is shown in Fig. 8.5 along with its truth table and logical expression $A + B$. $A + B$ is pronounced "A or B" or "A ORed with B." It is *not* "A plus B."

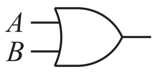
A third gate is called the exclusive-OR gate, or simply the XOR gate. The logic here is that the output will be high when either input is high, but not when both inputs are high. Note that this definition assumes there are only two inputs. This device is shown in Fig. 8.6. The circuit symbol is like the OR gate symbol; the curved line across the inputs denotes the exclusion described in the definition. The circle around the + in the Boolean expression $A \oplus B$ distinguishes this expression from the OR function. This is pronounced "A x or B."



Out $\equiv A \cdot B$

A	B	Out
0	0	0
1	0	0
0	1	0
1	1	1

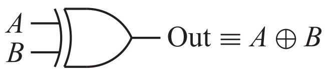
Figure 8.4 Circuit symbol, algebraic expression, and truth table for the AND gate.



Out $\equiv A + B$

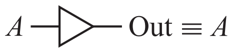
A	B	Out
0	0	0
1	0	1
0	1	1
1	1	1

Figure 8.5 Circuit symbol, algebraic expression, and truth table for the OR gate.



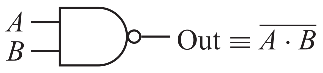
A	B	Out
0	0	0
1	0	1
0	1	1
1	1	0

Figure 8.6 Circuit symbol, algebraic expression, and truth table for the XOR gate.



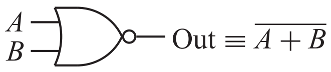
A	Out
0	0
1	1

Figure 8.7 Circuit symbol, algebraic expression, and truth table for the buffer gate.



A	B	Out
0	0	1
1	0	1
0	1	1
1	1	0

Figure 8.8 Circuit symbol, algebraic expression, and truth table for the NAND gate.

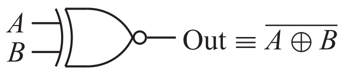


A	B	Out
0	0	1
1	0	0
0	1	0
1	1	0

Figure 8.9 Circuit symbol, algebraic expression, and truth table for the NOR gate.

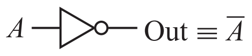
The buffer gate, shown in Fig. 8.7, seems to be superfluous. It has only one input, and the output is the same as the input. What good is this? This gate is used to regenerate logic signals. A logical high signal may start out at 5 V, say, but after being transmitted on conductors with non-zero resistance or after driving several other logic gates, the voltage level may fall and become perilously close to the defining threshold for a high logic level. The buffer is then used to boost the level up to a healthier level, thus maintaining the desirable noise immunity and extending the range for the transmission of the signal.

Each of the gates discussed so far has a corresponding negated version: the AND, OR, XOR, and buffer gates have the NAND, NOR, XNOR, and inverter gates as complements. The truth tables for these negated gates are the same as for the original gates except the output states are reversed. Thus the output states 0,0,0,1 for the AND gate become 1,1,1,0. The circuit symbol is the same except for a small circle on the output which indicates the inversion of the level. Finally, the Boolean symbol is changed by placing a bar over the original expression: thus $A \cdot B$ becomes $\overline{A \cdot B}$, and so on. These negated gates are shown in Figs. 8.8, 8.9, 8.10, and 8.11.



A	B	Out
0	0	1
1	0	0
0	1	0
1	1	1

Figure 8.10 Circuit symbol, algebraic expression, and truth table for the XNOR gate.



A	Out
0	1
1	0

Figure 8.11 Circuit symbol, algebraic expression, and truth table for the inverter gate.

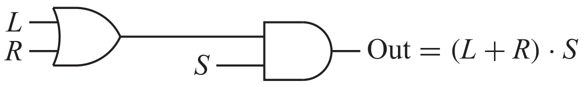


Figure 8.12 Solution of the car alarm problem.

8.5 Implementing logical functions

The implementation of simple logical functions can usually be determined after a little thought. For example, suppose you are designing a safety system for a two-door car. You want to sound an alarm (activated by a high level) when either door is ajar (this condition being indicated by a high logic level), but only if the driver is seated (again, indicated by a high level). Such a logic function is produced by the circuit in Fig. 8.12. The state of the left and right doors is represented by inputs L and R, while input S tells the circuit if the driver is seated. Thus if L or R is high (or both), and S is high, the output is high and the alarm sounds, as required.

With more complicated logic problems, the solution is less obvious. For such problems the *Karnaugh map* provides a method of solution. This method works for logic circuits having either three or four inputs. The first step in the method is to make a truth table for the problem. This follows from analyzing the requirements of our problem: under what conditions do we require a high output? As an example, suppose our analysis gives us the truth table shown in Fig. 8.13. For this example, we have three inputs, A, B, and C giving the output levels indicated.

The next step is to construct a Karnaugh map from the data in our truth table. This is illustrated in Fig. 8.14. The input states are listed along the top and left side of the map. For this example, with three inputs, we list the possible AB combinations along the top and the two C states along the left side.[4] When we

[4] For four inputs, the possible CD combinations would be listed along the left side as in Fig. 8.16.

A	B	C	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figure 8.13 Truth table for the Karnaugh map example.

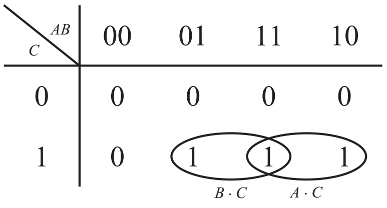


Figure 8.14 The Karnaugh map corresponding to Fig. 8.13.

list the AB combinations, we must follow a convention: only one digit at a time is changed as we write down the various combinations. In this example, we start (arbitrarily) with 00, and then change the second digit to get 01. To get another combination not yet listed, we change the first digit and get 11, and finally change the second digit obtaining 10. We then fill in the map with the data from the truth table.

The final steps are to identify groups of ones and then read the required logic from the map. The rule is to look for horizontal and/or vertical groups of 2, 4, 8, or 16. Diagonal groups are not allowed. In our example, there are two groups each containing two members. These are circled in Fig. 8.14. Now we identify the logic describing each group. To be a member of the group on the left, both B and C must be high, so the logic is $B \cdot C$. To be in the group on the right, both A and C must be high, so the logic is $A \cdot C$. Since a high output is obtained if we are a member of either group, the full logic describing our truth table is $(B \cdot C) + (A \cdot C)$. The implementation of this is shown in Fig. 8.15.

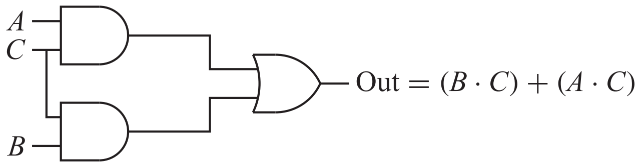


Figure 8.15 The logic circuit implementation of Fig. 8.13.

$CD \backslash AB$	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	0	0	1
10	1	0	0	1

Figure 8.16 A Karnaugh map example showing how edges connect.

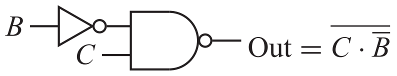


Figure 8.17 Circuit for the negated version of Fig. 8.16.

When looking for groups in the Karnaugh map, the edges of the map connect. This is illustrated in the map shown in Fig. 8.16. Because we can connect the right and left edges of the map, the ones in this map form a group of four, as indicated. To be a member of this group, C must be high and B must be low. Thus the logic for this group is $C \cdot \overline{B}$. This is much simpler logic than we would obtain if we instead identified two groups of two in our map.

Another simplification results in cases where the map has many ones and few zeros. In such cases, we can identify groups of zeros, find the logic for being a member of these groups, and apply an inversion to the result. For example, if the ones and zeros of the central portion of the map in Fig. 8.16 were reversed, we would find one group of four zeros. As we have seen, the logic for this group is $C \cdot \overline{B}$, but now we invert the result, obtaining $\overline{C \cdot \overline{B}}$. This final inversion could be done by using a NAND gate, as shown in Fig. 8.17

8.6 Boolean algebra

An *algebra* is a statement of rules for manipulating members of a set. You have, no doubt, learned in the past rules for doing mathematical manipulations with

Table 8.3 The Boolean algebra

Defining OR	$0 + A = A$
	$1 + A = 1$
	$A + A = A$
	$A + \overline{A} = 1$
Defining AND	$0 \cdot A = 0$
	$1 \cdot A = A$
	$A \cdot A = A$
	$A \cdot \overline{A} = 0$
Defining NOT	$\overline{\overline{A}} = A$
Commutation	$A + B = B + A$
	$A \cdot B = B \cdot A$
Association	$A + (B + C) = (A + B) + C$
	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
Distribution	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
	$A + (B \cdot C) = (A + B) \cdot (A + C)$
Absorption	$A + (A \cdot B) = A$
	$A \cdot (A + B) = A$
DeMorgan's 1	$\overline{A + B} = \overline{A} \cdot \overline{B}$
DeMorgan's 2	$\overline{A \cdot B} = \overline{A} + \overline{B}$

integers, real numbers, and complex numbers. There is also a special algebra for logical operations. It is called *Boolean algebra*.

The rules for Boolean algebra are shown in Table 8.3. They consist of definitions for the AND, OR, and NOT (or inversion) operations, and several theorems. In the table, A, B, and C are logical variables that can have values of 0 or 1. Once the definitions are accepted, the theorems can be proved by brute force by plugging in all the possible cases; since the variables have only two values, this is not too trying.

Boolean algebra can be used to find alternative ways of expressing a logical function. Consider the XOR function defined in Fig. 8.6. To get a high output, this function requires either A high while B is low, or B high while A is low. In algebraic terms,

$$A \oplus B = (A \cdot \overline{B}) + (B \cdot \overline{A}). \tag{8.3}$$

This equation shows us a way of producing the exclusive-OR function (other than buying an XOR gate). The resulting circuit is shown in Fig. 8.18. Note that in this figure (as in other figures in this chapter) we use the convention that crossing lines are not connected unless a dot is shown at the intersection point. This allows for more compact circuit drawings.

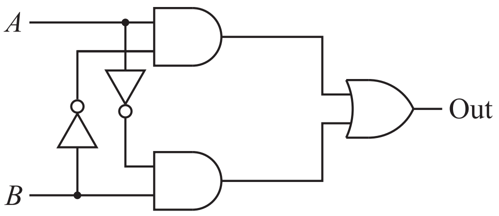


Figure 8.18 An alternative way of making an XOR gate.

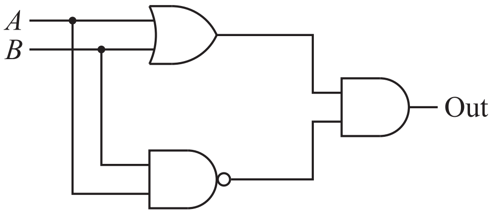


Figure 8.19 Another way of making an XOR gate.

Now we employ some algebraic manipulations to find another (and simpler) way to express the XOR function. In the first line of Eq. (8.4), we use the fact that $A \cdot \overline{A} = 0$ and $0 + A = A$ (for any A) to rewrite Eq. (8.3). The next line uses the Distribution Theorem to group terms together. The third line uses the second DeMorgan Theorem and the last line again uses the Distribution Theorem. The resulting logic is implemented in Fig. 8.19. Note that this way of making an XOR gate is simpler than that in Fig. 8.18 because it uses fewer gates:

$$\begin{aligned} A \oplus B &= (A \cdot \overline{B}) + (B \cdot \overline{A}) + (A \cdot \overline{A}) + (B \cdot \overline{B}) \\ &= A \cdot (\overline{A} + \overline{B}) + B \cdot (\overline{A} + \overline{B}) \\ &= A \cdot \overline{(A \cdot B)} + B \cdot \overline{(A \cdot B)} \\ &= (A + B) \cdot \overline{(A \cdot B)}. \end{aligned} \tag{8.4}$$

We have seen that we can construct an XOR gate from combinations of other gates. There is an interesting theorem that states that *any* logic function can be constructed from NOR gates alone, or from NAND gates alone. For example, suppose we want to make an AND gate from NOR gates. Using Boolean algebra, we can find the way:

$$A \cdot B = \overline{(\overline{A} + \overline{B})} = \overline{\overline{(A+0)} + \overline{(B+0)}}. \tag{8.5}$$

In the first equality, we have used the second DeMorgan Theorem and in the second equality we have used the fact that anything OR'd with 0 remains the same. The point is that the final expression is all in terms of NOR functions. The resulting circuit is shown in Fig. 8.20.

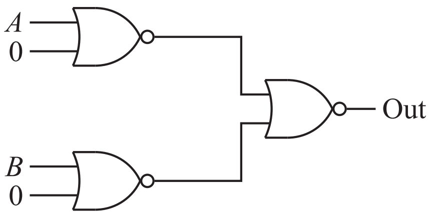


Figure 8.20 Making an AND gate from NORs.

It may seem that this is a silly thing to do. If you need an AND, why not just buy an AND instead of making it from NORs? There are two reasons. The first concerns the way logic gates are packaged. A typical integrated circuit (IC) chip will have four or six gates on a single chip, but all the gates are the same type (e.g., all NORs). Now if you are building a logic circuit that needs one NOR gate and one AND gate, you can buy two integrated circuits (one with NOR gates on it and one with AND gates on it) *or* you can use a single NOR gate IC containing at least four gates. In the latter case, one of the gates is used for the NOR function and the other three are used, as in Fig. 8.20, to produce the AND function. Thus you have saved money and circuit board space by using the NOR equivalent for the AND.

The second reason is, again, a practical one. If one is working on a logic circuit and runs out of one type of gate, is it useful to know that you can make do with a combination of NOR or NAND gates. Alternatively, if you are stocking an electronic workshop, you could just buy NOR or NAND gates instead of stocking all the different logic gates; you could always construct a needed function from the one type of gate you had on hand.

8.7 Making logic gates

Although we have discussed how logic gates function, we have not yet indicated how to make them. There are, in fact, many ways to make logic gates. A simple, low-tech way is to use an electromagnetic switch or *relay*, as shown in Fig. 8.21. The relay has a solenoid with a movable iron core that is mechanically attached to a switch. When a voltage is applied to the control input, the iron core is pulled into the solenoid and closes the switch. Without a control voltage, a spring (not shown) returns the switch to an open position.

Figures 8.22 and 8.23 show the use of relays to form an AND gate and an OR gate. The gate inputs *A* and *B* are connected to the relay controls and close the relevant switch when they are high. For the AND gate, two relays are connected in series, and for the OR gate, two relays are connected in parallel. When the switches

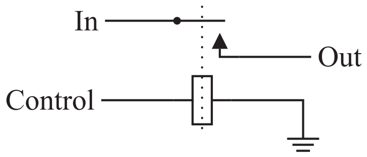


Figure 8.21 A basic relay.

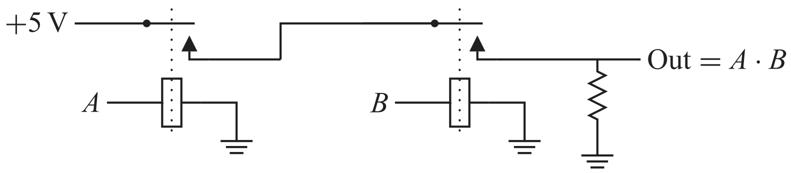


Figure 8.22 AND gate made from relays.

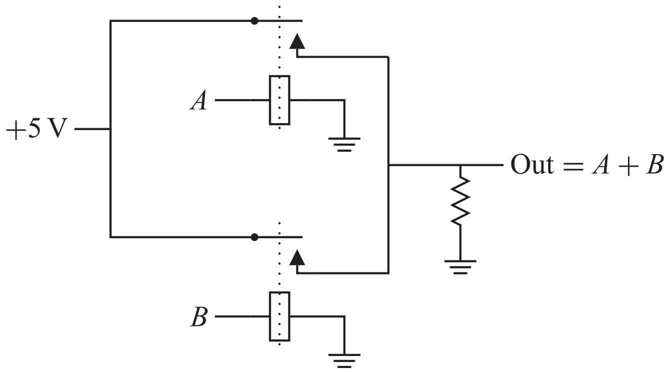


Figure 8.23 OR gate made from relays.

are open, the output is held at ground potential by the resistor *R*. When the logic is satisfied, the output is connected to the +5 V supply voltage and is thus high.

Semiconductor logic gates come in various types or families. One common type is the transistor-transistor logic (or *TTL*) family. Here bipolar transistors are used to create the logic gates. For example, the TTL NOR gate is shown in Fig. 8.24. If either *A* nor *B* is high, its transistor is driven into saturation, making the collector-emitter voltage small and the gate output low. If neither *A* nor *B* is high, both transistors are off, so there is no voltage drop across *R* and the output is high (+5 V).

Another important logic family is the complementary metal oxide semiconductor (or *CMOS*) family. These circuits employ field-effect transistors. For example, a CMOS NOT gate is shown in Fig. 8.25. When a high voltage is applied to the input *A*, the transistor turns on and the voltage across it becomes small, thus giving a low output. When a low voltage is applied, the transistor does not conduct, so the output remains at +5 V (i.e., high).

Table 8.4 Characteristics of some logic families

Family	Pros	Cons
TTL	Common, fast, cheap	High power consumption
CMOS	Low power consumption, suitable for large scale integration	Relatively slow
ECL	Fastest	High power consumption, low noise immunity

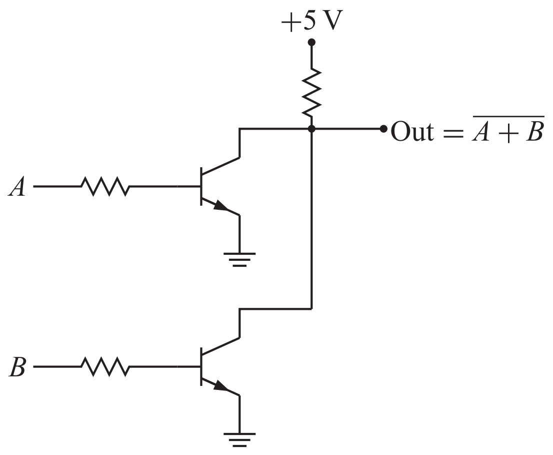


Figure 8.24 A TTL NOR gate made with bipolar transistors.

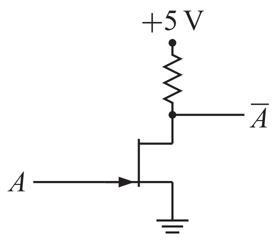


Figure 8.25 A CMOS NOT gate made with a field-effect transistor.

There are also special purpose logic families, like the fast-switching ECL (emitter coupled logic) family, but we will leave these for more advanced study. Some of the pros and cons of the families we have mentioned are shown in Table 8.4.

8.8 Adders

In addition to performing logic functions, gates can also be used to *add* binary numbers. To see this, consider the data in Table 8.5. We imagine that A and B are two binary numbers consisting of one bit, so each can only have the value 0 or 1.

Table 8.5 Adding two one-bit binary numbers

A	*B*	Sum	*C*	*S*
0	0	0	0	0
1	0	1	0	1
0	1	1	0	1
1	1	10	1	0

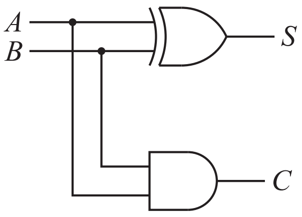


Figure 8.26 A half-adder made from an AND gate and an OR gate.

Our table shows the four possible combinations for A and B and their sum. Note that the last entry ($1 + 1 = 10$) requires two bits for the sum. We separate these two bits in the columns C and S.

If we think of columns C and S as outputs for a logic circuit having inputs A and B, we see that C is provided by the AND logic and S is given by XOR logic. We can thus perform this one-bit addition with the circuit of Fig. 8.26. This circuit is called a *half-adder*, which we will abbreviate HA. S stands for *sum* and C stands for *carry*.

Adding one-bit binary numbers is nice, but usually our arithmetic needs will involve longer binary numbers. Perhaps we can use a series of n half-adders to add an n-bit binary number. Unfortunately, this does not work. If we review the binary addition example in Table 8.2, we see that each column of the addition requires *three* possible inputs: one from each number and one to accommodate the possibility of a carry from the sum of the digits to the right. The half-adder *outputs* a sum and a carry but has no provision for a carry *input*.

The solution to this problem, shown in Fig. 8.27, uses two half-adders and an OR gate. This new circuit is called a *full adder* (FA). The full adder has three inputs, thus allowing for a carry input from the result of a previous addition. The output has a carry and a sum, just as for the half-adder. The reader can verify the proper operation of this device by working through the logic for all possible input combinations.

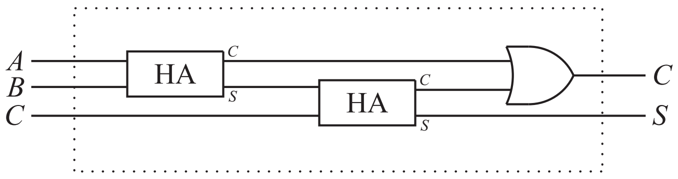


Figure 8.27 A full-adder made from two half-adders and an OR gate.

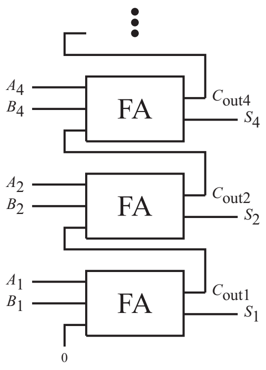


Figure 8.28 Several full-adders are used to sum the digits of two binary numbers.

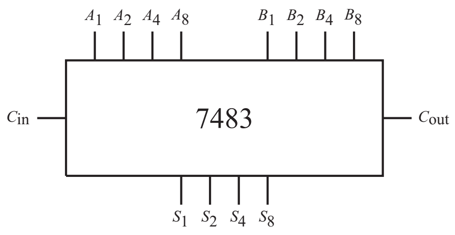


Figure 8.29 The four-bit full-adder IC.

The use of full adders to add two four-bit binary numbers is illustrated in Fig. 8.28. We imagine that our first number has digits A_1, A_2, A_4, and A_8, and that our second number has digits B_1, B_2, B_4, and B_8 (the subscripts are 2^n). In our circuit, each of these digits is represented by a logic level on a separate line. These are connected to a series of full adders as shown, with the carry out of one full adder connected to the carry in of the next. The carry input of the first full adder is not used and is thus connected to a low level. The sum output of each adder becomes a digit (S_1, S_2, S_4, and S_8) of the resulting sum, with the carry out from the last full adder giving S_{16}.

A four-bit full adder is available as an integrated circuit, shown schematically in Fig. 8.29. It has inputs for the four bits of A and B and outputs the four bits of S as well as a possible additional bit (the carry out). The carry in connection allows for the connection of multiple units when adding a number with more than four bits.

It is worth noting at this point how the complexity of our circuit has multiplied. Each half-adder uses two gates, so a full-adder has five gates, and the four-bit adder has twenty gates. Yet the usage rules for the four-bit adder are relatively simple. This is characteristic of digital circuitry: we build functionality by combining smaller

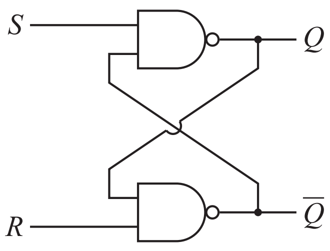


Figure 8.30 The basic flip-flop.

units. In the end, even the most complex digital circuit is built from humble logic gates.

8.9 Information registers

The output of the digital devices we have studied so far always reflects the current state of the inputs: change the inputs and the output changes. There is no memory of former input states. For many applications we would like to retain information about previous input states. This is done with *information registers*.

8.9.1 The basic or R-S flip-flop

The simplest information register is shown in Fig. 8.30. It is called by various names: the *basic flip-flop, binary*, or the *R-S flip-flop* (RSFF). The circuit has inputs S and R (which stand for Set and Reset) and outputs labeled Q and $\overline{Q}$. We emphasize that Q and $\overline{Q}$ are just labels. In most cases these will be in opposite logic states, but not always.

To see how this circuit functions, suppose that both S and R are high. What will be the state of Q and $\overline{Q}$? Because of the feedback between the outputs and the NAND gate inputs, we have to look for states that are self-consistent (i.e., that satisfy the logic of the entire circuit). Suppose, for example, that we assume $Q = 1$ and $\overline{Q} = 0$. If we follow the circuit, putting $Q = 1$ into the bottom NAND gate along with $R = 1$, we get a zero output, which is consistent with our assumption of $\overline{Q} = 0$. Putting $\overline{Q} = 0$ into the top gate along with $S = 1$ gives a output of 1, which is consistent with our assumption for Q. Thus the output state $Q, \overline{Q} = 1, 0$ satisfies the logic of the circuit. The reader can verify that the state $Q, \overline{Q} = 0, 1$ also satisfies the logic of the circuit, while $Q, \overline{Q} = 0, 0$ and 1,1 do not. Thus this circuit has two stable output states, or is *bistable*.

Table 8.6 Response of the basic flip-flop to changes in inputs

Case	Time	R	S	Q	$\overline{Q}$
1	Start	1	1	0	1
	End	1	0	1	0
2	Start	1	1	1	0
	End	1	0	1	0
3	Start	1	1	1	0
	End	0	1	0	1
4	Start	1	1	0	1
	End	0	1	0	1

The point of the circuit is that we can choose which of these two states the circuit will be in by making either R or S momentarily zero. The four possible cases are summarized in Table 8.6. For each case we start with both R and S high and Q and $\overline{Q}$ in one of the two stable states. Then we examine what happens when we make R or S low. In case 1, we see that if $Q, \overline{Q} = 0, 1$ initially, then making S low changes the outputs to $Q, \overline{Q} = 1, 0$. Case 2 shows that if $Q, \overline{Q} = 1, 0$ initially, then making S low has no effect. Similarly, cases 3 and 4 show that making R low leaves the outputs in the state $Q, \overline{Q} = 0, 1$ regardless of the initial state.

We can summarize these results by saying that the current output state tells us which input was low last. If S was low last, then the output state will be $Q, \overline{Q} = 1, 0$. If R was low last, the output state will be $Q, \overline{Q} = 0, 1$. In shorthand, changes are governed by

$$(R, S) \rightarrow (Q, \overline{Q}) \tag{8.6}$$

assuming only one of the inputs is low.[5]

8.9.2 The clocked flip-flop

While the basic flip-flop remembers which input was low last, this memory is limited; the next time R or S changes, the former result is lost. This is the motivation for our next circuit, shown in Fig. 8.31. This circuit is called the *clocked* or *gated R-S flip-flop*. The right part of the circuit is the basic flip-flop we studied in the

[5] If both R and S are low, the outputs become $Q, \overline{Q} = 1, 1$. But when R and S are returned to their normal high state, this output cannot remain since it is not a stable state. The output falls into one of the stable states, but we cannot predict which one.

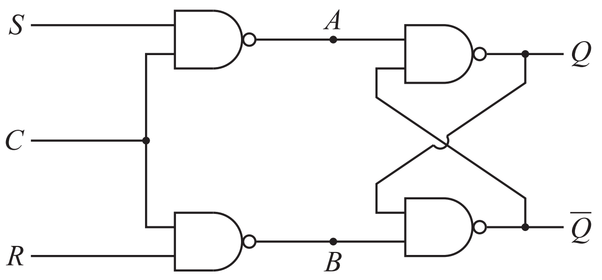


Figure 8.31 The clocked flip-flop.

last section; recall that this requires a low input to change states. The inputs to this basic flip-flop come from the NAND gates on the left.

The truth table for a NAND gate tells us that it will only have a low output if both inputs are high. Our clocked flip-flop thus works as follows: we imagine that R and S are normally low and that occasionally one of them becomes high. If the clock input C is also high, the corresponding NAND gate can output a low and thus change Q and $\overline{Q}$. If C is low, however, changes to R and S have no effect on Q and $\overline{Q}$. In shorthand, changes are governed by

$$(S, R) \rightarrow (Q, \overline{Q}) \tag{8.7}$$

but only if C is high and assuming only one of the inputs S, R is high.

Our "clock" input C thus provides a degree of isolation between the inputs S and R and the outputs Q and $\overline{Q}$. The input information is only "read" and "saved" in the output state when C is high. Our memory is now more selective and permanent: we only store input information when C is high and we can keep that information expressed on the outputs as long as C is low.

8.9.3 The M-S flip-flop

The clocked flip-flop of the last section isolates changes in the inputs from changes in the outputs through the use of the clock input. The output can change only when C is high. If, however, there are *multiple changes* in the levels of the inputs S and R during the time C is high, the outputs will also change multiple times. We would like to take the process of isolating input changes from output changes one step further so that the state of the inputs S and R at one instant of time will be read and saved.

We can accomplish this using the circuit shown in Fig. 8.32. Two clocked flip-flops (CFF) are used along with a NOT gate to form the *master-slave or M-S flip-flop* (MSFF). When C is low, changes in the inputs S and R will not affect the

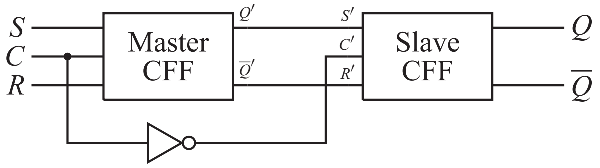


Figure 8.32 The M-S flip-flop.

outputs of the master flip-flop Q' and $\overline{Q}'$. The clock input C' of the slave flip-flop, however, is high, and its inputs S' and R' are directly connected to Q' and $\overline{Q}'$. Thus the current state of Q' and $\overline{Q}'$ is reflected in the final outputs Q and $\overline{Q}$.

When C is high, changes in S and R are reflected in the master outputs Q' and $\overline{Q}'$, but since C' is low, these do not change the final outputs Q and $\overline{Q}$. When C changes from high to low, the state of S and R at that instant will also be present at the input of the second flip-flop, and this input will be read as C' changes from low to high. We have thus created a circuit that reads and stores the input states S and R only at the high-low transition of the clock input C. This is called *negative transition edge clocking*. Again, in shorthand, changes are governed by

$$(S,R) \rightarrow (Q,\overline{Q}) \tag{8.8}$$

but only at the high-low transition of C and assuming only one of the inputs S,R is high.

8.9.4 Other flip-flop variations

There are other variations in flip-flop construction and operation that should be noted. The clocked flip-flop and M-S flip-flop can also be constructed from NOR gates rather than NANDs. The result is a CFF that changes when the clock is low rather than high, and an MSFF that changes on the low-high transition of the clock rather than the high-low transition.

The simple summaries of our flip-flop operations given by Eqs. (8.6), (8.7), and (8.8) are accompanied by assumptions about the logic levels on the S and R inputs. If both inputs are high for the CFF or MSFF (or both low for the RSFF), the output state becomes unpredictable when the inputs return to their normal state. One way to deal with this problem is to add feedback lines from the outputs Q and $\overline{Q}$ to the inputs. A clocked flip-flop with this modification is shown in Fig. 8.33. Following common usage, the S and R inputs are renamed J and K, respectively, and the circuit is called a *clocked J-K flip-flop*. The reader can verify that this circuit follows the rules of the clocked RSFF with the following change. If both J and K are high when the clock goes high, the outputs switch (or toggle) from whichever

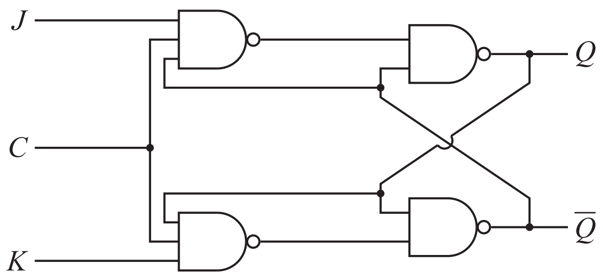


Figure 8.33 The *J-K* modification of the clocked flip-flop.

state they were in to the opposite state. Thus, $Q, \overline{Q} = (1, 0)$ becomes $(0, 1)$ and vice versa. Similarly, a *J-K* version of the MSFF can be constructed.

Finally, we note that direct set and reset (or clear) inputs are often added to flip-flops. These provide an additional way to change the outputs that usually overrides the other rules of operation. Thus, a prescribed logic level (this might be high or low depending on the device details) applied to the direct set input will immediately produce $Q, \overline{Q} = (1, 0)$ regardless of what the clock and other inputs are doing. Conversely, when the prescribed level is applied to the direct reset input, $Q, \overline{Q} = (0, 1)$ is immediately produced.

8.10 Counters

We have seen that flip-flops can be used to store information about the logic levels of their inputs. They are also used to produce other useful functions. One example of this is shown in Fig. 8.34 where the outputs of an MS flip-flop are connected to the inputs. The result is a *toggle flip-flop* (TFF). To see how this works, we imagine that the clock input C is regularly switching states, as shown in Fig. 8.35. The outputs Q and $\overline{Q}$ will be in opposite states, and we suppose they start out as $Q, \overline{Q} = (0, 1)$. The outputs will change to reflect the input states at the high-low transition of the clock C, in accordance to Eq. (8.8). Since $\overline{Q}$ is connected to S in our circuit, the state of $\overline{Q}$ at the transition will end up at Q. Similarly, the state of Q at the transition will end up at $\overline{Q}$. The $(Q, \overline{Q})$ states thus switch or *toggle* at the negative transition edge of C. If the clock waveform is a square wave as in Fig. 8.35, then Q and $\overline{Q}$ are also square waves, but with twice the period of C, or half the frequency. Because of this latter interpretation, the toggle flip-flop is sometimes called a *divide-by-two*.

If we connect several toggle flip-flops as shown in Fig. 8.36, we obtain a *binary counter*. The output of the leftmost TFF is used as the clock for the next TFF, and this connection strategy is repeated as we move to the right. The $\overline{Q}$ outputs are

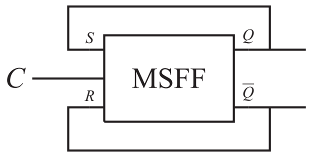


Figure 8.34 The toggle flip-flop.

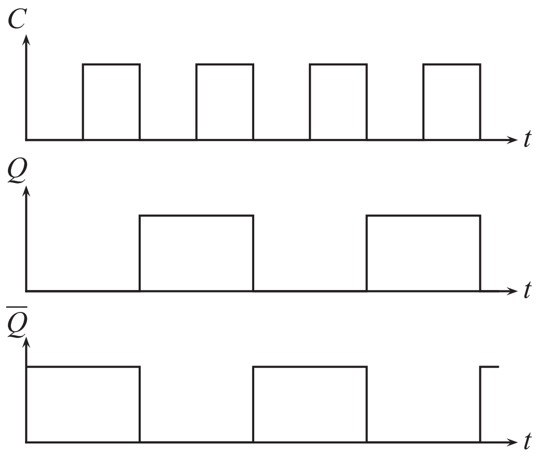


Figure 8.35 Typical waveforms for the toggle flip-flop.

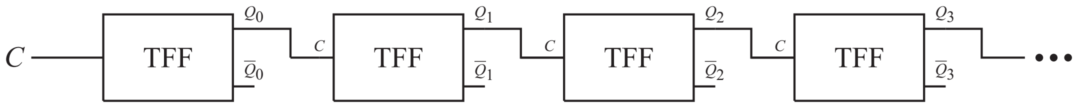


Figure 8.36 Toggle flip-flops joined to form a binary counter.

not used. The output of each TFF changes state as its particular clock waveform undergoes a high-low transition. The output waveforms for the first four TFFs are shown in Fig. 8.37.

To see why this circuit acts as a counter, think of the outputs Q_0, Q_1, Q_2, and Q_3 as the digits of a binary number $Q_3Q_2Q_1Q_0$ that gives the number of high-low transitions in C. We start with all the outputs low, so the count is 0000_2. After the first high-low transition, Q_0 is high, so we have 0001_2, as shown by the leftmost dotted line in Fig. 8.37. After seven high-low transitions, Q_0, Q_1, and Q_2 are high and our number is 0111_2, as shown by the next dotted line. One more transition gives 1000_2, matching the eight transitions that have occurred (see third dotted line). This process continues until the count reaches fifteen (1111_2). The next transition gives 0000_2 and our counter has reset to its initial state (last dotted line). Clearly, this can be extended to use any number n of toggle flip-flops, in which case the counter will reset to zero after 2^n clock transitions.

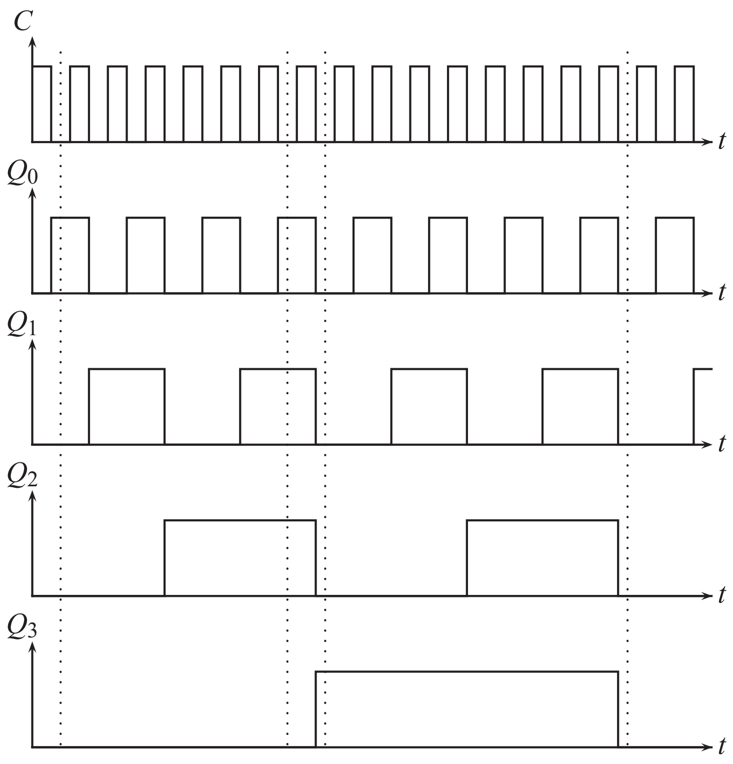


Figure 8.37 Typical waveforms for a binary counter.

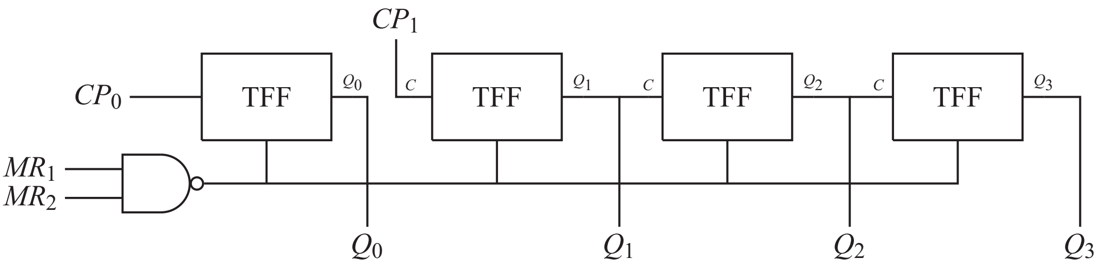


Figure 8.38 Schematic for the 7493 binary counter.

Counting circuits are available as integrated circuits. The functional schematic for one such IC (the 7493) is shown in Fig. 8.38. For flexibility, one TFF is independent, with its own clock input CP_0. The other three TFFs are connected internally as needed for a binary counter and are driven by CP_1. To make a four-bit counter, we simply connect Q_0 to CP_1.

This circuit has an additional useful feature. There are two *master reset* inputs MR_1 and MR_2. When both of these inputs are high, the counter immediately resets all outputs to zero. By connecting one or two of the outputs to these master

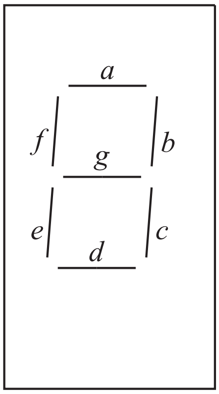


Figure 8.39 A seven-segment display showing the segment labels *a–g*.

resets, we can force the counter to return to zero after reaching a certain count. For example, if we connect Q_2 and Q_0 to MR_1 and MR_2, the first time these two outputs are high (i.e, 0101_2), the counter will immediately return to zero. Thus the count will go like this: 0000, 0001, 0010, 0011, 0100, 0000, etc. A counter that returns to zero after *n* counts is called a *modulo-n counter*. Using this terminology, in this example we have taken a modulo-16 counter and made it into a modulo-5 counter by employing the master resets. Since our usual base 10 number system resets after 10 counts (0,1,2,...9,0,...), a modulo-10 counter is given a special name: a *binary-coded decimal* or *BCD* counter.

8.11 Displays and decoders

The usual number displays in our world are decimal, using the digits zero through nine. A common device used for such numbers is the *seven-segment display*, shown in Fig. 8.39. The device has seven independent line segments (*a* through *g*) which can be lit by applying a high level to the appropriate input. By lighting the appropriate segments, a boxy version of each of the ten decimal digits can be produced.

As we have seen, our counter circuits produce binary representations of numbers. If we want to display these numbers using a seven-segment display, we need a circuit that will take the four outputs of our BCD counter and light up the appropriate segments of the display. Such a device is called a *decoder*, and a typical IC is shown in Fig. 8.40. It has inputs for the BCD digits Q_0, Q_1, Q_2, and Q_3 and outputs for each of the segments of the seven-segment display.

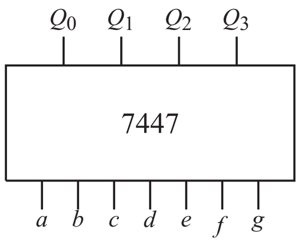


Figure 8.40 The 7447 decoder IC.

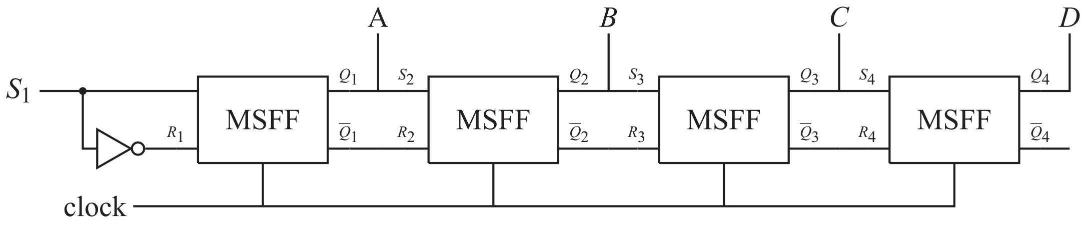


Figure 8.41 Using MSFFs to make a shift register.

8.12 Shift registers

Another way to employ MS flip-flops is to join them together to form a *shift register*, as shown in Fig. 8.41. In this case, all the flip-flops have a common clock input which is typically a square wave as shown in Fig. 8.42. A series of high and low levels is applied to S_1 synchronized with the clock. Since the MSFF operation follows Eq. (8.8), the level present at S_1 at the high-to-low transition of the clock will be transfered to Q_1. At the next transition, this will be transfered to Q_2, and so on. The net effect is that the waveform sequence applied at S_1 is transferred down the line of flip-flops, one step for each negative transition edge of the clock. If we wish to repeat the pattern, the output of the last flip-flop can be returned to the input of the first.

8.12.1 Shift register applications

Shift registers have a number of applications and we mention two of them here. The first is the production of a scrolling message sign of the type often seen in public places. If we connect each of the outputs *A, B, C*..., of Fig. 8.41 to an LED in a long row of LEDs, we would see our input pattern move down the row at a rate set by the clock. If we do the same thing with a number of shift register/LED rows and arrange the rows one under the other, we obtain a rectangular array of LEDs controlled by the various flip-flops in the shift registers. We can now load any pattern we choose into the array and propagate it down the line. Often this pattern

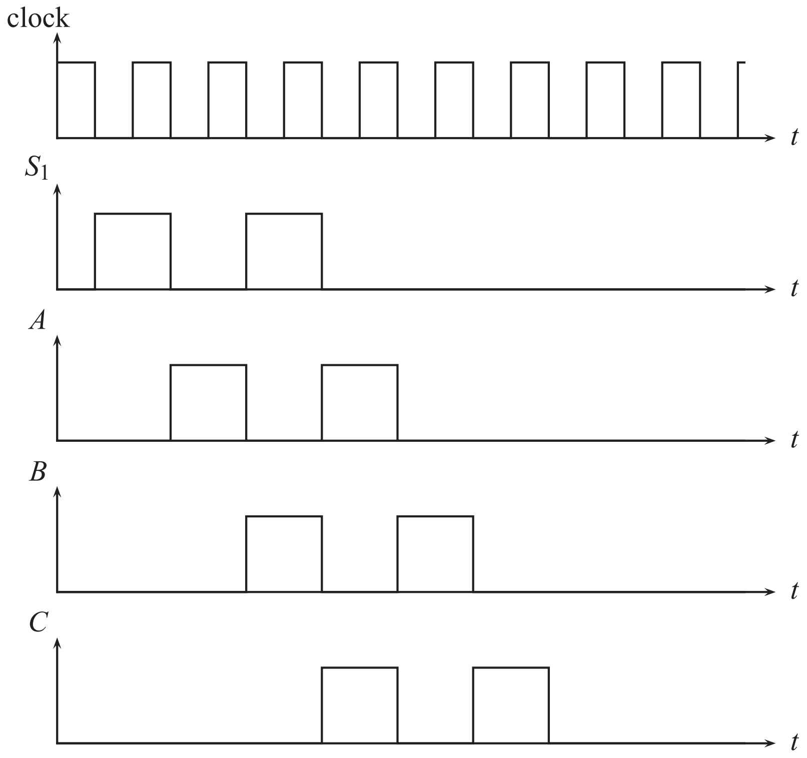


Figure 8.42 Example waveforms for the shift register.

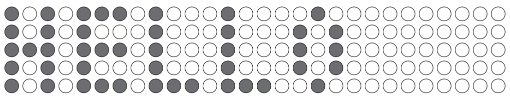


Figure 8.43 Scrolling message display.

is chosen to form the letters of a message as in Fig. 8.43. The word "HELLO" would move one column to the right with each clock pulse.

We can also use multiple shift registers for *digital waveform synthesis*. For this application we imagine we have m shift registers, each of which has n flip-flops and thus can hold n logic levels. We use only the output of the last flip-flop of each shift register. At any given time the logic levels of these m outputs form an m-bit binary number B. Each time the clock pulses, a different number B appears on the outputs, and, since each shift register has n flip-flops, we can produce n different m-bit numbers. The output of the last flip-flop of each shift register is connected to the input of the first flip-flop so that the pattern repeats. This setup is shown in Fig. 8.44.

Now suppose we wish to create a periodic waveform. We divide the period of the waveform into n equal divisions as shown in Fig. 8.45. An m-bit binary number

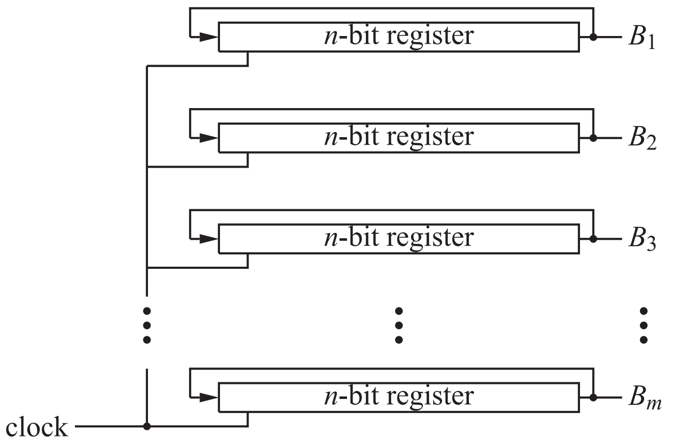


Figure 8.44 A set of shift registers used to make a waveform synthesizer.

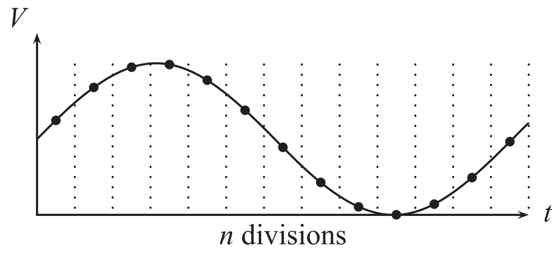


Figure 8.45 Waveform synthesizer wave.

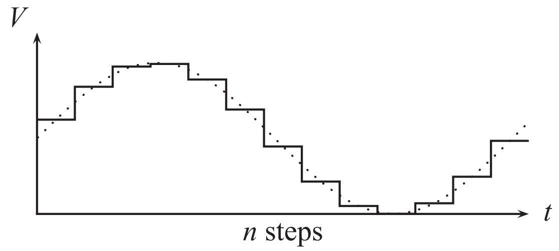


Figure 8.46 Synthesized wave (solid line). The dotted line shows the waveform after low-pass filter.

proportional to the amplitude of the waveform at each time is then loaded into the shift register array. After all this information is loaded, we run the clock at its normal rate. The circuit now gives a series of binary numbers B at the output that represent the amplitude of the waveform at subsequent time steps. If we can translate these numbers into an analog voltage (see Section 8.13), we can produce a stepped version of the original waveform as in Fig. 8.46. This can then be passed through a low-pass filter which will smooth it into a sine wave.

Although this method of producing waveforms seems more complicated than using an analog circuit, it has certain advantages. The most striking is that we can reproduce *any periodic waveform* with this method. Our analog circuits can produce certain waveforms (sine, sawtooth, square waves), but would be hard pressed to make a more irregular pattern. Second, the frequency of our synthesized waveform is easily changed by changing the clock rate. Lastly, the circuit design is easily extended; we can improve the precision (in time or amplitude) of our synthesized waveform by increasing the number of shift registers or the number of flip-flops in each shift register.

8.13 Digital to analog converters

The last step in our digital waveform synthesizer is to take the series of binary numbers B presented at the output of our array of shift registers and convert them to analog voltages. This requires a device called a *digital to analog converter* (also written as *D/A converter* or *DAC*). It has an input for each bit of the binary number and outputs a voltage proportional to that number. There are various ways to do this, but a simple one which uses our previous knowledge is shown in Fig. 8.47. It uses the op-amp adder circuit developed in Section 6.3. Recall that this circuit produces a *weighted sum*, with the weights set by the resistor values:

$$V_{\text{out}} = -\left(\frac{R_\text{f}}{R_1}V_1 + \frac{R_\text{f}}{R_2}V_2 + \frac{R_\text{f}}{R_3}V_3 + \frac{R_\text{f}}{R_4}V_4\right). \tag{8.9}$$

Since our input voltages come from a digital circuit, all high levels will be at the same voltage and similarly for the low levels. If these represent the bits of a binary number, however, we want the more significant bits to count more than those of lesser significance. This can be achieved by weighting the sum. If we choose the resistor values shown in Fig. 8.47, Eq. (8.9) reduces to

$$V_{\text{out}} = -(V_1 + 2V_2 + 4V_3 + 8V_4). \tag{8.10}$$

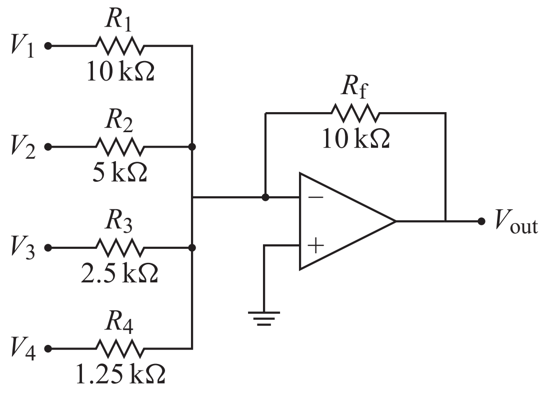


Figure 8.47 A four-bit digital to analog converter.

Thus, if we connect the 2^0 bit of our binary number to V_1, the 2^1 bit to V_2, the 2^2 bit to V_3, and the the 2^4 bit to V_4 we will get an output voltage that is proportional to the binary input number. Clearly, this scheme can be extended to any number of bits.

In most cases, a user will simply buy a D/A converter on a chip rather than build one from components. In addition to our scaled resistor method, there are several other ways to achieve digital to analog conversion, and the interested reader should consult the end-of-chapter references for details. Here we briefly discuss some of the issues common to all DACs.

The *resolution* of a D/A converter refers to the smallest change step at the output. This is set by the number of input bits. For an n-bit input, the resolution is one part in 2^n. *Linearity* is a measure of how much the output varies from a perfect proportionality to the input binary number. The *accuracy* tells you how well the proportionality matches a specified value. For our op-amp example, linearity and accuracy would depend on how well the resistors matched the desired values. Finally, the *settling time* is the time required for the output to get within some specified amount of its final value. For the op-amp DAC, the settling time would depend on the slew rate of the op-amp.

8.14 Analog to digital converters

While most computation and data analysis is done on digital computers, most laboratory signals are analog. We therefore need a device that will change our lab signals into binary numbers, and this device is called an *analog to digital converter* (also written as *A/D converter* or *ADC*). Again, there are several different methods to achieve this. Here we discuss a simple method that uses some of our previously studied devices. The schematic for a *staircase A/D* is shown in Fig. 8.48 with associated waveforms in Fig. 8.49.

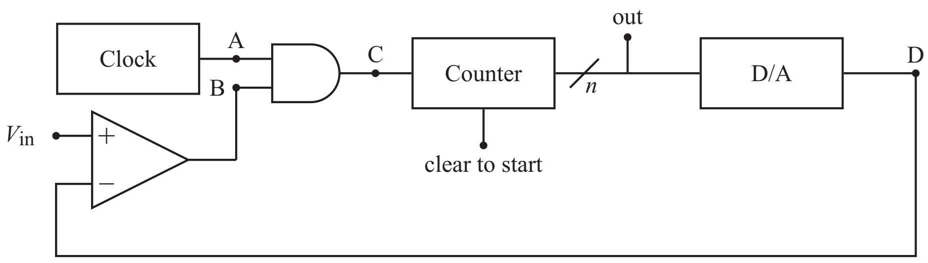


Figure 8.48 A staircase analog to digital converter.

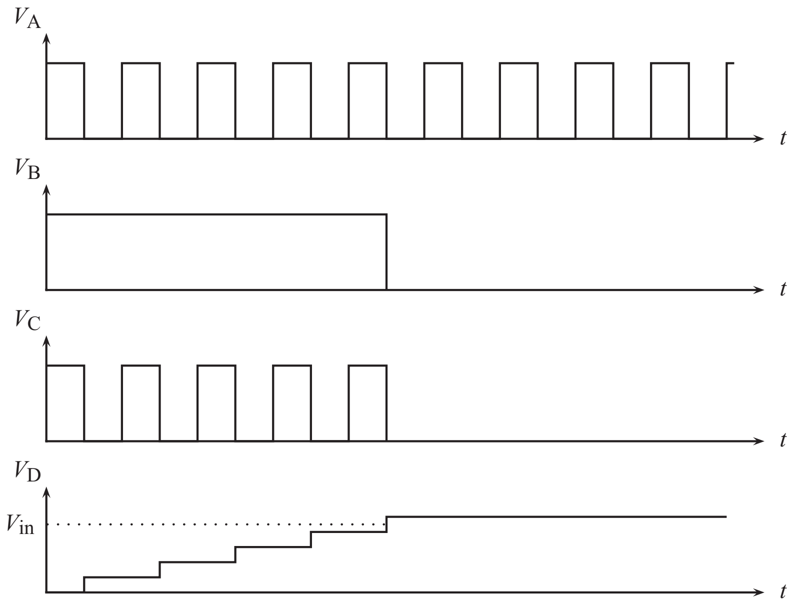


Figure 8.49 Waveforms for the staircase analog to digital converter.

The voltage to be converted, V_{in}, which we take to be positive, is applied to the (+) input of a comparator. To start the conversion, the binary counter is cleared (set to zero). All *n* output lines of the counter (represented by a single slashed line in Fig. 8.48) are thus low. The D/A converter then converts this zero binary number to zero voltage, which is fed into the (−) input of the comparator. Since $V_{in} > 0$, the comparator output is high. This allows the clock signal to pass through the AND gate and be counted by the counter. As the count increases, the output voltage, V_D, of the D/A converter increases in a staircase fashion. Eventually, V_D becomes greater than V_{in} and this causes the comparator output to go low. This in turn prevents the clock signal from passing through the AND gate so the counter stops. At this point the binary number at the output of the counter is proportional to the input voltage, and the conversion is complete.

8.15 Multiplexers and demultiplexers

Multiplexers and demultiplexers are devices that route signals. A *multiplexer* takes one of several inputs and connects it to a single output. The input line that is connected is determined by the binary number applied to the address lines. A schematic of a multiplexer with four input lines is shown in Fig. 8.50. The two

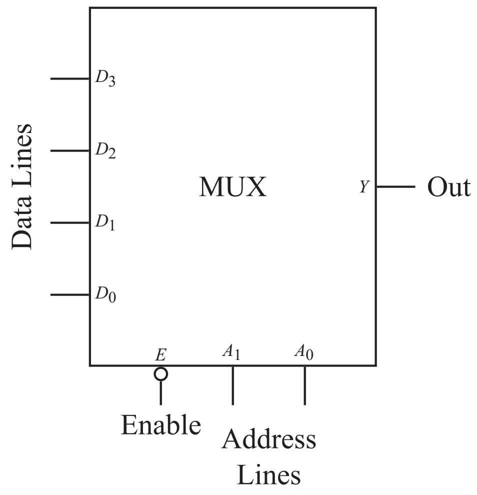


Figure 8.50 A simple four-to-one digital multiplexer.

address lines can express numbers zero to three corresponding to the four inputs. For example, if $A_1A_0 = 10$, then the logic level at D_2 appears at the output pin Y.

Several general comments about multiplexers can be made. The number of inputs on a multiplexer is not limited to four and is related to the number of address lines. It takes N address lines to support 2^N data lines. A digital multiplexer is unidirectional: the digital level at the selected data line is transferred to the output Y, but a digital level applied at Y will *not* be transfered to the selected data line. Finally, most multiplexers have an Enable input, as shown in Fig. 8.50. If this input is low, the multiplexer is enabled and the operation described above occurs. If Enable is high, the output is not connected to any data line and has a high impedance to ground. This allows one essentially to disconnect Y from whatever follows it. The small circle on the Enable input reflects the fact that the logic is reversed (a high level would normally be associated with turning a device on, not off).

As an application of the multiplexer, consider Fig. 8.51 which shows a parallel-to-serial converter. The parallel data presented at the four input lines are sequentially presented at the output as the counter driven by a clock oscillator steps through its four binary numbers: $A_1A_0 = 00$, 01, 10, and 11. After this the counter resets to zero and the process repeats. Although not shown, the Enable line could be employed to prevent the repetition.

A multiplexer can also be used to implement a truth table. A simple example of this is shown in Fig. 8.52. The truth table inputs A and B are connected to the address lines and the correct output level for the truth table is achieved by connecting the data input lines either high or low. Although this example would be trivial to

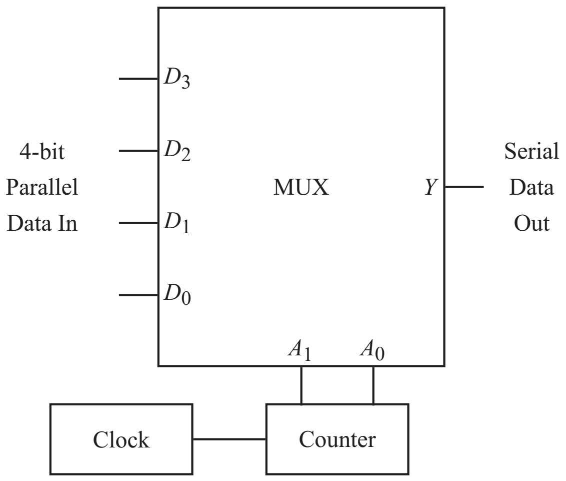
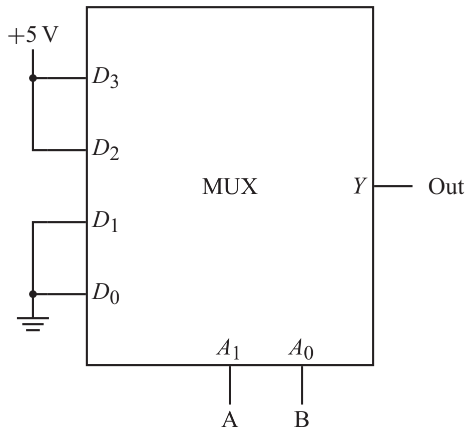


Figure 8.51 Using a multiplexer to perform parallel to serial conversion.



A	*B*	Out
0	0	0
0	1	0
1	0	1
1	1	1

Figure 8.52 Using a multiplexer to implement a truth table.

implement without a multiplexer (Output $= A$), this method of implementing truth tables becomes more attractive (i.e., simpler and more compact) when the number of inputs increases.

The opposite of a multiplexer is a *demultiplexer*, shown in Fig. 8.53. In this case, the logic level at the single input D is routed to one of several output lines Y in accordance with the binary number applied to the address lines. Unused output lines have a high impedance to ground, as does the selected output if the Enable input is high. As with the multiplexer, digital demultiplexers are unidirectional.

Finally, we note that there exist analog versions of the multiplexer that will pass along an analog signal (i.e., one that can vary continuously) and not just high/low logic levels. These devices are typically bidirectional so that signals can pass in

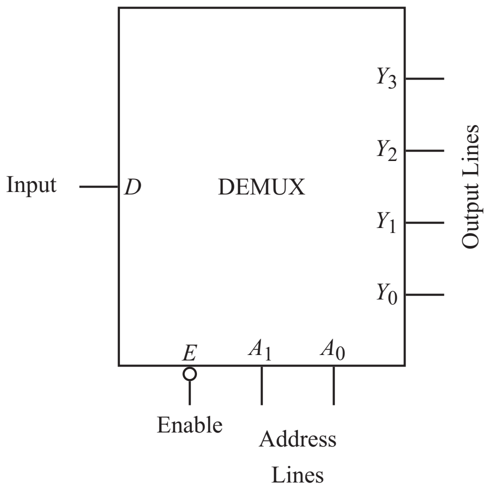


Figure 8.53 A simple one-to-four digital demultiplexer.

either direction as they would with a mechanical switch, hence the alternative name *analog switch*. Also note that, since they are bidirectional, the same device can be used either as a multiplexer or as a demultiplexer.

8.16 Memory chips

The information registers discussed earlier stored one bit of information, a single high/low level. With the advent of large scale integration technologies it has become possible to multiply this process thousands or millions of times on a single chip. A representative device is shown in Fig. 8.54. We first note that the bits are stored in groups called a data *word*. For our example, the word size is four bits, as seen by the four data lines. The number of words in the memory is limited to and typically set by the number of address lines. In our case, there are eight address lines and so $2^8 = 256$ words of memory on the chip. The Enable pin must be low to either read or write data. When it is high the data lines are essentially disconnected (have a high impedance to ground). When the Write pin is low, logic levels on the data lines are stored (written or input) to the memory word specified by the binary number on the address lines. When the Write pin is high, logic levels stored at the memory location specified by the address lines are placed on the data lines where they can be read by or output to other devices.

Memory chips come in a huge variety of sizes and types and carry their own set of acronyms and terminology. *Volatile* memory devices lose all stored information if the electrical power supply is removed, while *non-volatile* memory does not. *Static*

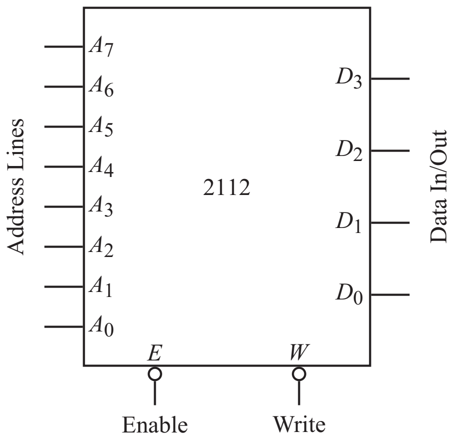


Figure 8.54 A 4-bit × 256 RAM.

memory stores data as long as the chip has power, while *dynamic* memory needs to have the stored information periodically refreshed or re-written to maintain storage. The example given above is an example of *RAM* or *random access memory* for which any data word can be accessed via the address lines in an order chosen by the user. In contrast, the shift register might be termed a sequential access memory because the bits have to be accessed in order of their position within the register.

The information in a *ROM* or *read only memory* cannot be routinely changed (written) but is intended as permanently stored information that will be read as needed. The stored information might be set at the time of manufacture, or, with a *PROM* or *programmable ROM*, set by the user using a special apparatus. With some PROMs, this step can occur only once, while an *EPROM* can be Erased and re-programmed.

EXERCISES

1. Develop the truth table for the circuit shown in Fig. 8.55.

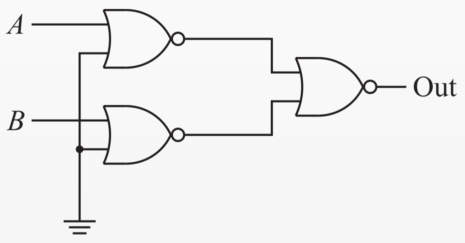


Figure 8.55 Circuit for Problem 1.

2. Using only NAND gates, construct a circuit that will implement the following logical expressions. Use Boolean algebra to simplify the expressions as much as possible before you begin.

(a) $(A \cdot B) + (\overline{A} \cdot B) + (A \cdot \overline{B}) + (\overline{A} \cdot \overline{B})$

(b) $[(A \cdot B) + C] \cdot [(A \cdot B) + D]$

(c) $[(\overline{A} \cdot B) \cdot (A \cdot B)] + (A \cdot B)$

(d) $(1 + B) \cdot (A \cdot B \cdot C)$

3. Using only NOR gates, give circuits that are equivalent to each of the following: AND, OR, NAND, and XOR.

4. Produce the truth table for the AND-OR-INVERT (AOI) gate shown in Fig. 8.56.

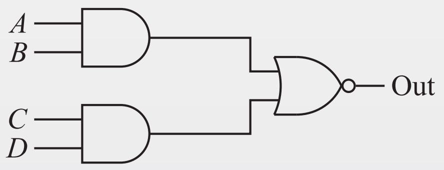


Figure 8.56 Circuit for Problem 4.

5. Without a calculator, find the binary equivalents of the following base 10 numbers: 92, 66, 120, 511, 37, 255.

6. Using two-input logic gates, design an alarm system which lights an LED when any of five doors is open. Assume that an open door gives a high logic level.

7. Using only NOR gates, produce a clocked flip-flop with the same functionality as the one from Section 8.9.2. Use no more than five gates. Hint: try using the same configuration with the NAND gates replaced with NORs.

8. Suppose we want a four-person vote counter that will output a high level when three or four persons vote YES (indicated by a high input level). Produce a Karnaugh map for this problem and give the resulting logic circuit. You may use logic gates with any legal number of inputs.

FURTHER READING

Dennis Barnaal, *Digital and Microprocessor Electronics for Scientific Application* (Prospect Heights, IL: Waveland Press, 1982).
Don Lancaster, *TTL Cookbook* (Indianapolis, IN: Sams, 1983).
Don Lancaster, *CMOS Cookbook*, 2nd edition (Boston, MA: Newnes, 1997).
Sol Libes, *Small Computer Systems Handbook*, (Rochelle Park, NJ: Hayden, 1978).
Niklaus Wirth, *Digital Circuit Design for Computer Science Students: An Introductory Textbook* (New York: Springer, 1995).