Chapter 4

Combinational Logic Design

The foundations for the design of digital logic circuits were established in the preceding chapters. The elements of Boolean algebra (two-element "switching algebra") and how the operations in Boolean algebra can be represented schematically by means of gates (primitive devices) were presented in Chapter 2. How switching expressions can be manipulated and represented in different ways was the subject of Chapter 3, which also presented various ways of implementing such representations in a variety of circuits using primitive gates.

With all of the tools for the purpose now in hand, we will be concerned in this chapter with the design of more complex logic circuits. Circuits in which all outputs at any given time depend only on the inputs at that time are called *combinational* logic circuits. The design procedures will be illustrated with important classes of circuits that are now universal in digital systems.

The approach taken is to examine the tasks that a combinational logic circuit is intented to perform and then identify one or more circuits that can perform the task. One circuit may have some specific advantages over others, but it may also have certain deficiencies. Often one factor can be improved, but only at the expense of others. Some important factors are speed of operation, complexity or cost of hardware, power dissipation, and availability in prefabricated units. We will take up a number of different operations that are useful in different contexts and show how appropriate circuits can be designed to carry out these operations.

1 BINARY ADDERS

One of the most important tasks performed by a digital computer is the operation of adding two binary numbers.[1] A useful measure of performance is speed. Of course, speed can be improved by using gate designs that favor speed at the

[1] As discussed in Chapter 1, subtraction of two numbers is included in the meaning of addition, since subtraction is performed first by carrying out some operation on the subtrahend and then adding the result. (What operation is first performed depends on the type of computer—either inverting the subtrahend or taking its two's complement, as discussed in Chapter 1.)

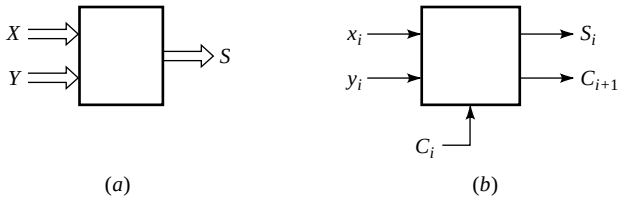


Figure 1 Binary addition. (*a*) General adder. (*b*) Full adder of two 1-bit words.

expense of other measures, such as power consumption (using advanced Schottky, for example, versus low-power Schottky designs). But for the *logic* designer, the important question is how to design an adder to increase the speed, regardless of the type of gate used. It may be that increased speed can be achieved at the expense of increased circuit complexity. That is, there might be several designs, each characterized by a certain speed and a certain circuit complexity. A judgment must be made as to the acceptable trade-offs between them.

A symbolic diagram representing a binary adder is shown in Figure 1*a*. Each open arrowhead represents multiple variables; in this case the inputs are two binary numbers. If each number has n digits, then each line shown really represents n lines. The sum of two n-bit numbers is an $(n + 1)$-bit number. Thus, S (sum) represents $n + 1$ output lines. If this circuit were designed by the methods of Chapter 3, we would require a circuit with $n + 1$ output functions, each one dependent on $2n$ variables. The truth table for each of the output functions would have 2^{2n} rows. Since n could easily be in the range 20–40, a different approach is obviously needed.

Full Adder

An alternative approach for the addition of two n-bit numbers is to use a separate circuit for each corresponding pair of bits. Such a circuit would accept the 2 bits to be added, together with the carry resulting from adding the less significant bits. It would yield as outputs the 1-bit sum and the 1-bit carry out to the more significant bit. Such a circuit is called a *full adder*. A schematic diagram is shown in Figure 1*b*. The 2 bits to be added are x_i and y_i, and the *carry in* is C_i. The outputs are the *sum* S_i and the *carry out* C_{i+1}. The truth table for the full adder and the logic maps for the two outputs are shown in Figure 2.

The minimal sum-of-products expressions for the two outputs obtained from the maps are

$$S_i = x_i'y_iC_i' + x_iy_i'C_i' + x_i'y_i'C_i + x_iy_iC_i \tag{1a}$$

$$\begin{aligned} C_{i+1} &= x_iy_i + x_iC_i + y_iC_i \\ &= x_iy_i + C_i(x_i + y_i) \end{aligned} \tag{1b}$$

(Make sure you verify these.) Each minterm in the map of S_i constitutes a prime implicant. Hence, a sum-of-products expression will require four 3-input AND gates and a 4-input OR gate. The carry out will require three AND gates and an

Short ___
Even ___

C_i	X_i	Y_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	0	1
1	1	0	1	1
1	1	1	0	1

(*a*)

yz \ x	0	1
00		1
01	1	
11		1
10	1	

(*b*)

yz \ x	0	1
00		
01		1
11	1	1
10		1

(*c*)

Figure 2 Truth table and logical maps of the full adder. (*a*) Truth table. (*b*) S_i map. (*c*) C_{i+1} map.

OR gate. If we assume that each gate has the same propagation delay t_p, then a two-level implementation will have a propagation delay of $2t_p$.

In the map of the carry out, minterm m_7 is covered by each of the three prime implicants. This is overkill; since m_7 is covered by prime implicant $x_i y_i$, there is no need to cover it again by using it to form prime implicants with m_5 and m_6. If there is some benefit to it, we might use the latter two minterms as implicants without forming prime implicants with m_7. The resulting expression for C_{i+1} becomes

$$C_{i+1} = x_i y_i + C_i(x_i' y_i + x_i y_i') = x_i y_i + C_i(x_i \oplus y_i) \tag{2}$$

(Confirm this result.) We already have an expression for S_i in (1*a*), but it is in canonic sum-of-products form. It would be useful to seek an alternative form for a more useful implementation.

Exercise 1 With the use of switching algebra, confirm that the expression for the sum in (1*a*) can be converted to

$$S_i = x_i \oplus y_i \oplus C_i \tag{3}$$ ◆

Using the expressions for S_i and C_{i+1} containing XORs, confirm that we can obtain the implementation of the full adder shown in Figure 3*a*. Notice that the circuit consists of two identical XOR-AND combinations and an additional OR gate. The circuit inside each dashed box is shown in Figure 3*b*; it is named a *half adder*. Its only inputs are the 2 bits to be added, without a carry in. The two outputs are (1) the sum of the 2 bits and (2) the carry out.

Assuming that an XOR gate (implemented in a two-level circuit) has a propagation delay of $2t_p$, the full adder in Figure 3*a* has a propagation delay of $4t_p$, both for the sum and for the carry. (Verify these claims.)

We will observe in the following section that the overall speed in the addition of two *n*-bit binary numbers depends mainly on the speed with which the carry propagates from the least significant bit to the most significant bit. Hence, reducing the delay experienced by the carry of a full adder is a significant improvement. This is an incentive in seeking other implementations of the full adder. In some of the cases in Problem 1 at the end of the chapter, additional implementations of the full adder are

___ Short
___ Even

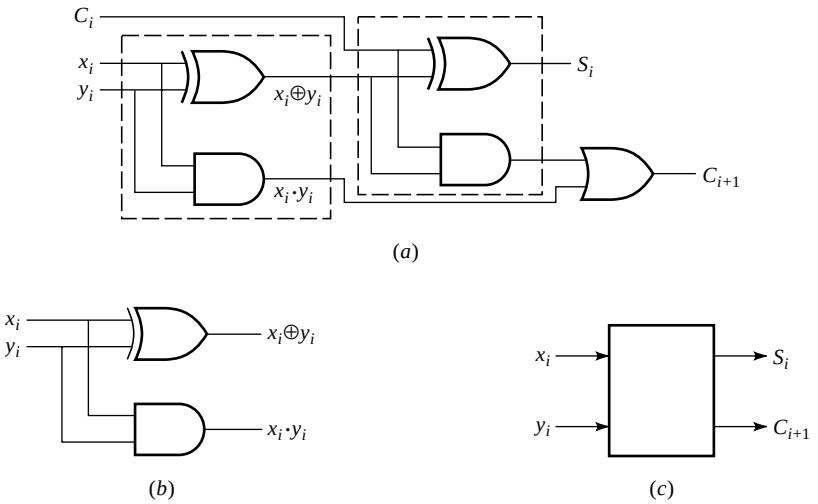


Figure 3 Full adder implemented with half adders. (*a*) Full adder. (*b*) Half adder. (*c*) Half adder schematic diagram.

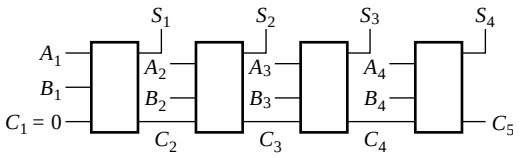


Figure 4 Four-bit ripple-carry adder.

proposed in which the propagation delay for the carry is $2t_p$ instead of $4t_p$. Henceforth, for a full adder, we will assume that the propagation delay of the carry is $2t_p$.

Ripple-Carry Adder

The problem of adding two multidigit binary numbers has the following form. Two *n*-bit binary numbers are available, with all digits being presented in parallel. The addition is performed by using a full adder to add each corresponding pair of digits, one from each number. The full adders are connected in tandem so that the carry out from one stage becomes the carry into the next stage, as illustrated for the case of four-digit numbers in Figure 4. Thus, the carry *ripples* through each stage. For binary addition, the carry into the first (least significant) stage is 0. The last carry out (the *overflow* carry) becomes the most significant bit of the $(n + 1)$-bit sum.

Since the carry of each full adder has a propagation delay of $2t_p$, the total delay in carrying out the sum of two *n*-bit numbers is $2nt_p$. Not every pair of two *n*-bit numbers will experience this much delay. Take the following two numbers as an example:

Short ___
Even ___

101010
010101

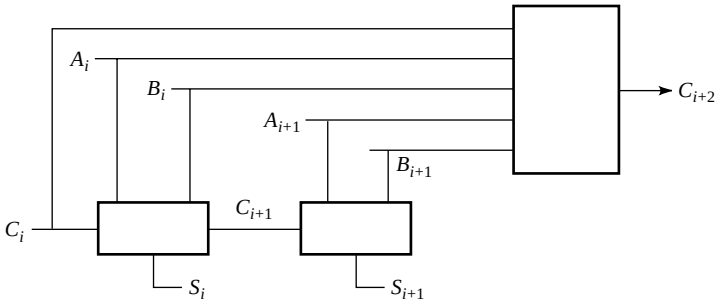


Figure 5 Carry-lookahead circuit schematic.

Assuming that the carry into the first stage is zero, no carries are generated at any stage in taking the sum. Hence, there will be no carry ripple, and so no propagation delay along the carry chain.

However, to handle the general case, provision must be made for the worst case; no new numbers should be presented for addition before the total delay represented by the worst case. The maximum addition speed, thus, is limited by the worst case of carry propagation delay.

Carry-Lookahead Adder

In contemplating the addition of two n-digit binary numbers, we were appalled by the thought of a single combinational circuit with all those inputs. So we considered the repeated use of a simpler circuit, a full adder, with the least possible number of inputs. But what is gained in circuit simplicity with this approach is lost in speed. Since the speed is limited by the delay in the carry function, some of the lost speed might be regained if we could design a circuit—just for the carry—with more inputs than 2 but not as many as $2n$. Suppose that several full-adder stages are treated as a unit. The inputs to the unit are the carry into the unit as well as the input digits to all the full adders in that unit. Then perhaps the carry out could be obtained faster than the ripple carry through the same number of full adders.

These concepts are illustrated in Figure 5 with a unit consisting of just two full adders and a carry-lookahead circuit. The four digits to be added, as well as the input carry C_i, are present simultaneously. It is possible to get an expression for the carry out, C_{i+2}, from the unit by using the expression for the carry of the full adder in (2).

For reasons which will become clear shortly, let's attach names to the two terms in the carry expression in (2), changing the names of the variables to A and B from x and y in accordance with Figure 5. Define the *generated carry* G_i and the *propagated carry* P_i for the ith full adder as follows:

$$G_i = A_i B_i \tag{4a}$$

$$P_i = A_i \oplus B_i \tag{4b}$$

Inserting these into the expression for the carry out in (2) gives

$$C_{i+1} = A_i B_i + C_i(A_i \oplus B_i) = G_i + P_i C_i \tag{5}$$

___ Short
___ Even

A carry will be *generated* in the *i*th full adder (that is, $G_i = 1$) if A_i and B_i both equal 1. But if only one of them is 1, a carry out will not be generated. In that case, however, P_i will be 1. (Confirm this.) Hence, the carry out will be $C_{i+1} = C_i$. We say that the carry will be *propagated* forward.

The expression for the carry out in (5) can be updated by changing the index i to $i + 1$:

$$\begin{aligned} C_{i+2} &= G_{i+1} + P_{i+1}C_{i+1} = G_{i+1} + P_{i+1}(G_i + P_iC_i) \\ &= G_{i+1} + P_{i+1}G_i + P_{i+1}P_iC_i \end{aligned} \tag{6}$$

The last expression can be interpreted in the following way. A carry will appear at the output of the unit under three circumstances:

- It is generated in the last stage: $G_{i+1} = 1$.
- It is generated in the first stage, $G_i = 1$, and propagated forward: $P_{i+1} = 1$.
- The input carry C_i is propagated through both stages: $P_i = P_{i+1} = 1$.

Obviously, this result can be extended through any number of stages, but the circuit will become progressively more complicated.

Exercise 2 Extend the previous result by one more stage and write the expression for C_{i+3}. Then describe the ways in which this carry out can be 1. Confirm your result using the general result given next. ◆

Extending the design to j stages, the expression in (6) becomes

$$C_{i+j+1} = G_{i+j} + P_{i+j}G_{i+j-1} + P_{i+j}P_{i+j-1}G_{i+j-2} + \cdots + (P_{i+j}P_{i+j-1} \cdots P_i)C_i \tag{7}$$

This expression looks complicated, but it is easy to interpret. Since the carry out $C_{i+j+1} = 1$ if any one of the additive terms on the right is 1, the carry out from the unit will be 1 for several possibilities. Either it is generated in the last (jth) stage of the unit, or it is generated in an earlier stage and is propagated through all succeeding stages, or the carry into the unit is propagated through all the stages to the output.

The greater the number of full-adder stages included in a unit, the greater the improvement in speed—but also the greater the complexity of the carry-lookahead circuit. There is an obvious trade-off between the two. Consider a unit of four stages. This unit is to add two 4-bit words A and B. Each stage can be considered as having a sum circuit (S) and a separate carry circuit (C). The sum circuit of each stage has as inputs the carry from the preceding stage and the corresponding bits of the A and B words. The inputs to the carry network of each stage consist of *all* the bits of the A and B words up to that stage and the carry—not just from the preceding stage, but from the input to the whole unit. Thus, if the first stage is stage i, the inputs to the carry circuit of stage $i + 2$ are: A_i, A_{i+1}, A_{i+2}, B_i, B_{i+1}, B_{i+2}, and C_i.

Exercise 3 Draw a schematic diagram for a three-stage unit using rectangles to represent the sum and carry circuits of each stage. (Let the first stage be 1 instead of the general i.) ◆

Short ___
Even ___

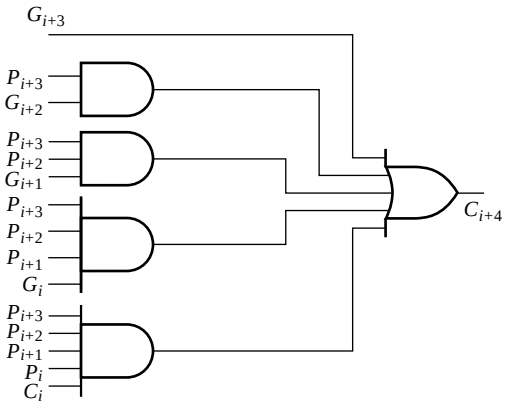


Figure 6 Four-stage carry-lookahead circuit.

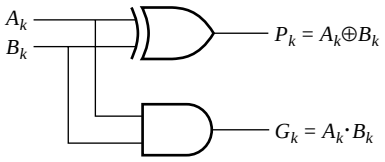


Figure 7 Half adder for generated and propagated carries.

A circuit implementation of the carry network of the last stage in a four-stage unit is shown in Figure 6. Except for C_i, the carry into the unit, all other inputs to the AND gates are generated carries and propagated carries from the various stages of the unit. These generated and propagated carries are produced by the half-adder circuits in Figure 7.

A semi-block diagram of the four-stage carry-lookahead adder is shown in Figure 8. (Note that pins that carry the same label in different subcircuits are assumed to be connected.) Since each propagated carry P_{i+j} is the output of an XOR gate, the overall propagation delay of the carry circuit having the design of Figure 7 is $4t_p$. However, all generated and propagated carries, G_{i+j} and P_{i+j}, of all units become available within $2t_p$ after the two words are first presented for addition, as evident from Figure 6. Hence, in all carry-lookahead units besides the first, the propagation delay of the carry network is only $2t_p$.

Exercise 4 Suppose that a carry-lookahead adder is to have k 4-bit units to carry out the addition of two $4k$-bit words. From the preceding discussion, from the diagram of Figure 8 implementing each unit, and from a consideration of the first and last units, determine the propagation delay of this adder in terms of t_p, the propagation delay through one gate. (Don't peek at the answer until you do the work.)

Answer [2]

[2]The sum of the delays through (a) the carry circuit of each unit ($2t_p$ each), (b) the sum circuit of the last unit ($2t_p$) since it depends on having the carry from the last unit, and (c) the extra delay in getting the carry from the first unit. Total delay = $(k + 1 + 1)2t_p = (2k + 4)t_p$ ◆

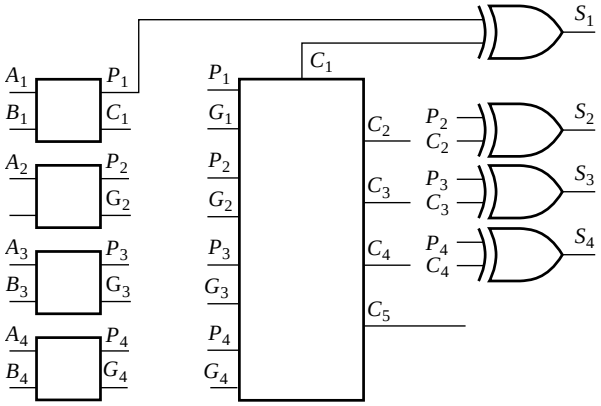


Figure 8 Schematic diagram of 4-bit carry-lookahead adder.

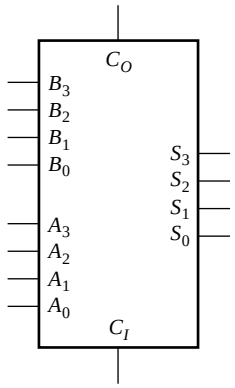


Figure 9 High-speed adder: 4-bit words.

If an adder has eight 4-bit units, the propagation delay through a carry-lookahead adder will be $20t_p$. The corresponding ripple-carry adder will have a propagation delay of $4 \times 8 \times 2t_p = 64t_p$. Thus, the carry-lookahead adder will have an advantage of 320 percent in speed over the ripple-carry adder. All is not gravy, however: the speed advantage has been paid for in the cost of the added hardware.

Exercise 5 From a count on the number of gates in each implementation, estimate the hardware disadvantage (in percent) of the carry-lookahead adder compared with the ripple-carry adder. Compare the disadvantage with the 320 percent speed advantage. ◆

The circuits described here are available in IC packages. A single full adder, for example, is available as a unit. A ripple-carry adder, as illustrated in Figure 4, and a carry-lookahead adder for 4-bit words, as shown in Figure 8, are available as MSI packages.

Short ___
Even ___

Externally, a package consisting of a ripple-carry adder of 4-bit words would look the same as a package consisting of a carry-lookahead adder of 4-bit words. The block diagram in Figure 9 illustrates such a package. There are

nine inputs: the carry in and four inputs per word. There are five outputs: the carry out and the 4 bits of the sum. (The carry out becomes the most significant bit of the sum if the circuit is used just to add 4-bit words, and not as part of an adder of longer words.)

Binary Subtractor

In Chapter 1 two representations of signed binary numbers were studied: one's complement and two's complement. Recall that when numbers are represented in one of the complement forms, the only special treatment needed in the addition of a negative number with another positive or negative number is in the final carry out. Thus, the adders studied in the previous section are suitable for the addition of complement numbers if some additional circuitry is used to process the final carry out. Also, binary subtraction can be performed using the same adder circuits by negating the subtrahend.

Two's-Complement Adder and Subtractor

Recall from Chapter 1 that when the addition of 2 two's complement binary numbers produces a final carry, it can be ignored. However, it is necessary to detect the overflow that can occur when the result of the addition is out of range.[3] In Chapter 1 it was concluded that an arithmetic overflow could be detected if the carry in and carry out of the most significant bit position are different. Thus, the overflow can be detected with one additional Exclusive-OR gate. The two's complement adder is not much different from the binary adder for unsigned numbers.

What about subtraction? We already suggested that subtraction should be carried out by complementing the subtrahend and adding. So the task is to design a circuit whose output is the two's complement of the input, and use its output as one input to an adder. Such a circuit can be designed easily, but why should a system contain some hardware dedicated to addition and other hardware dedicated to subtraction? If the only difference between these two circuits is a circuit that computes the two's complement, then why not design a circuit where either addition or subtraction can be selected with one additional input? When this additional input is, say, 0 the circuit performs addition, and when the input is 1 the circuit performs subtraction. It sounds easy; a representation of the circuit can be derived using the techniques of Chapter 3, but an elegant solution exists that we describe next.

Examine the truth table of the Exclusive-OR operation and notice that it can be viewed as a conditional inverter. If one input is 0, then the output is identical to the second input. If one input is 1, then the output is the complement of the second input. This is convenient for producing the complement of an input to our adder/subtractor circuit when we want to perform subtraction. However, to compute the two's complement of a binary number we have to add

[3]The range of binary numbers having n binary digits represented in two's complement form is $-2^{n-1} \le m \le 2^{n-1} - 1$.

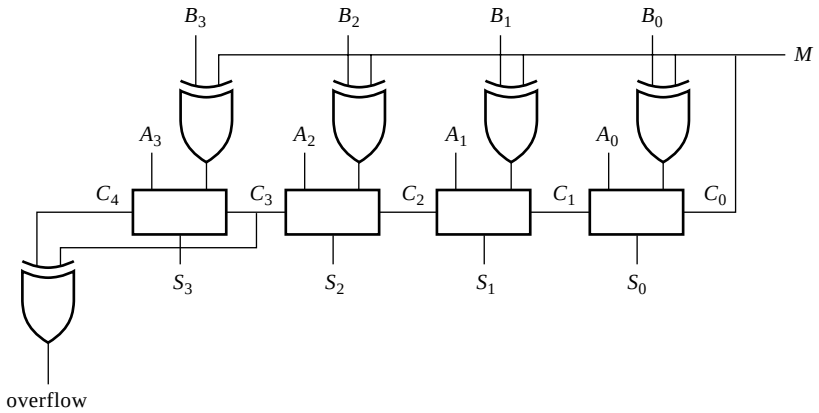


Figure 10 Two's complement adder/subtractor with overflow detection.

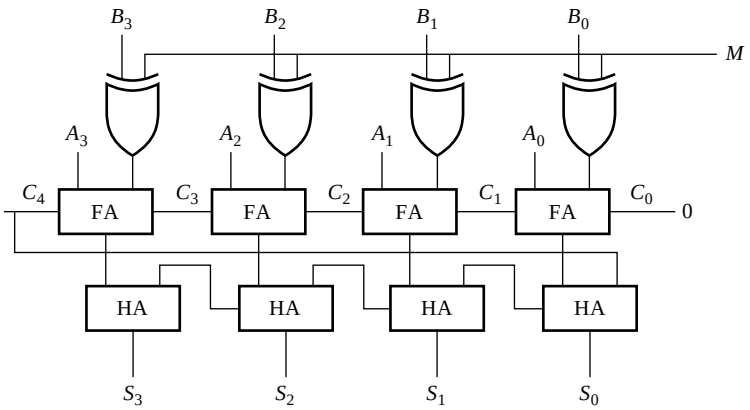


Figure 11 One's complement adder/subtractor.

1. Any ideas on how to do this without additional gates? (Think about it before you continue.)

The full adder for the least significant bit has a carry input signal that can be utilized to add the required 1. The design of our two's complement adder/subtractor circuit is complete; a version for adding 4-bit numbers is shown in Figure 10. If the control signal M is 0, then the circuit performs $A+B$; however, if M is 1, the circuit performs $A - B$.

One's-Complement Adder and Subtractor

To perform subtraction in one's complement we can use the Exclusive-OR circuit used in the two's complement adder/subtractor. The only difference is that we do not want to inject a carry into the least significant bit. One's complement addition requires the addition of 1 to the sum when a carry out from the most significant bit position occurs. This can be accomplished using multiple half adders as shown in Figure 11. Overflow detection for one's complement addition is left as a problem for you.

Short ____
Even ____

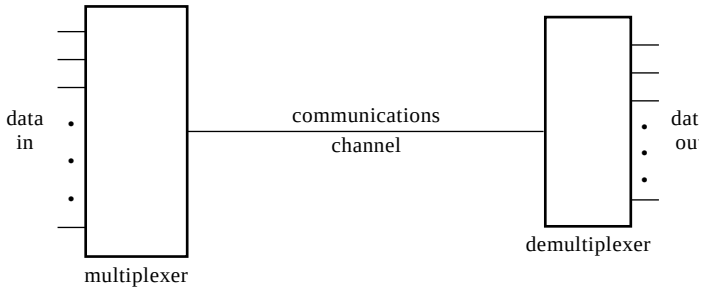


Figure 12 A data communication problem.

Two's complement addition is the most common method implemented in modern computers due to its reduced circuit complexity compared with one's complement.

This is as far as we will go with the addition of multibit words; other adder circuits are left for the problem set.

2 MULTIPLEXERS

Many tasks in communications, control, and computer systems can be performed by combinational logic circuits. When a circuit has been designed to perform some task in one application, it often finds use in a different application as well. In this way, it acquires different names from its various uses. In this and the following sections, we will describe a number of such circuits and their uses. We will discuss their principles of operation, specifying their MSI or LSI implementations.

One common task is illustrated in Figure 12. Data generated in one location is to be used in another location; A method is needed to transmit it from one location to another through some communications channel.

The data is available, in parallel, on many different lines but must be transmitted over a single communications link. A mechanism is needed to select which of the many data lines to activate sequentially at any one time so that the data this line carries can be transmitted at that time. This process is called *multiplexing*. An example is the multiplexing of conversations on the telephone system. A number of telephone conversations are alternately switched onto the telephone line many times per second. Because of the nature of the human auditory system, listeners cannot detect that what they are hearing is chopped up and that other people's conversations are interspersed with their own in the transmission process.

Needed at the other end of the communications link is a device that will undo the multiplexing: a *demultiplexer*. Such a device must accept the incoming serial data and direct it in parallel to one of many output lines. The interspersed snatches of telephone conversations, for example, must be sent to the correct listeners.

A digital multiplexer is a circuit with 2^n data input lines and one output line. It must also have a way of determining the specific data input line to be selected at any one time. This is done with n other input lines, called the *select* or *selector* inputs, whose function is to select one of the 2^n data inputs for connec-

___ Short
___ Even

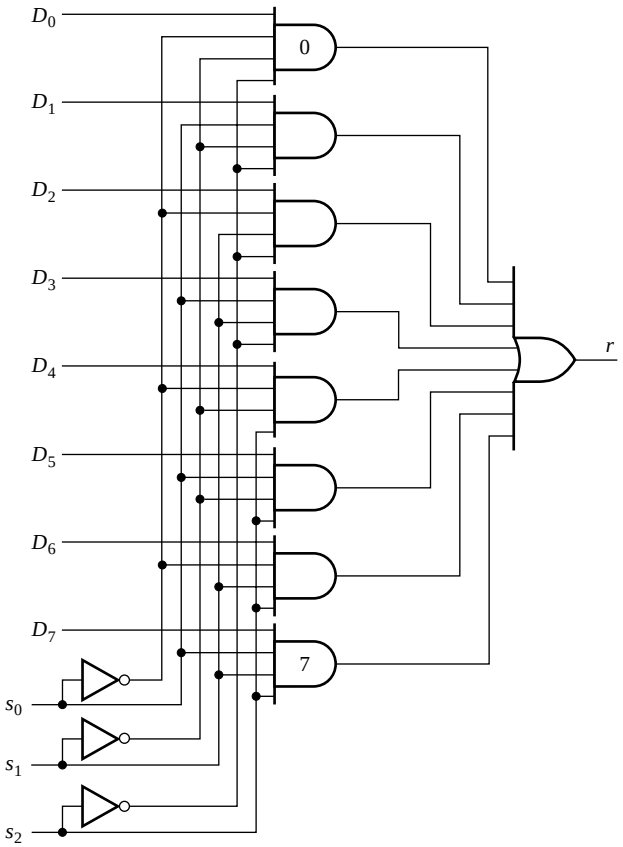


Figure 13 Multiplexer with eight data inputs.

tion to the output. A circuit for $n = 3$ is shown in Figure 13. The n selector lines have $2^n = 8$ combinations of values that constitute binary *select numbers*.

Exercise 6 Write expressions for each of the AND gate outputs in terms of the s_i and D_i inputs, confirming that the multiplier of D_k is the binary equivalent of k. ◆

When the selector inputs have the combination $s_2s_1s_0 = 011$, for example, the outputs of all AND gates except the one to which data line D_3 is connected will be 0. All other inputs to that AND gate besides D_3 will be 1. Hence, D_3 appears at the output of the circuit. In this way, the select inputs whose binary combination corresponds to decimal 3 have selected data input D_3 for transmittal to the output.

Standard MSI packages are available as multiplexers. Figure 14*a* shows the circuit for a package containing two separate multiplexers for $n = 2$. Practical considerations not included in Figure 13 account for some of the features of this circuit. The *enable* input E, for example, is used to control the period of time that the multiplexer is operative. Thus, when the value of E is 1, the output will be 0 no matter what the values of the select inputs. The circuit will be operative only when the corresponding enable input is 0. (In other circuits, the

Short ____
Even ____

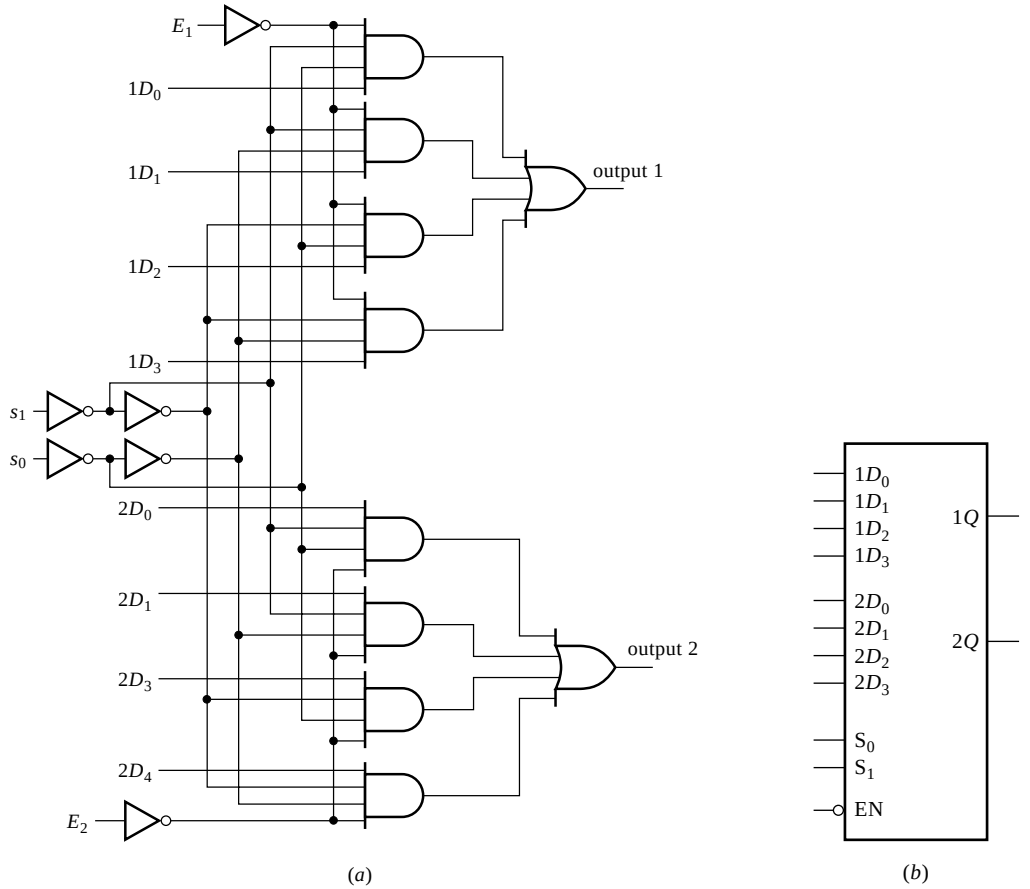


Figure 14 (*a*) Dual four-input multiplexer with enable. (*b*) Dual four-input multiplexer with single enable.

enable signal is not inverted; in such cases, the circuit is operative when $E = 1$, just the opposite of the case shown in Figure 14*a*.)

In addition, note from the figure that both the selector signals and their complements are inputs to AND gates. The signal inputs themselves are obtained after two inversions. This is especially useful if n is large. In this way, the circuit that produces the select inputs has as load only a single gate (the inverter) rather than several AND gates. In Figure 14*a* the select inputs are common to both multiplexers, but each has its own enable. In other designs, the enable can also be common. A schematic diagram of a dual four-input multiplexer (MUX) with a single enable is shown in Figure 14*b*.

The preferred gate form for many IC logic packages (for example, the 74LS00 and the 74LS10) is the NAND gate. Since the multiplexer design in either Figure 13 or 14 is a two-level AND-OR circuit, a direct replacement of all AND and OR gates by NAND gates will maintain the logic function, as discussed in the preceding chapter. In this way, the actual implementation of the multiplexer is carried out with NAND gates.

Multiplexers as General-Purpose Logic Circuits

It is clear from Figures 13 and 14 that the structure of a multiplexer is that of a two-level AND-OR logic circuit, with each AND gate having $n + 1$ inputs, where n is the number of select inputs. It appears that the multiplexer would constitute a canonic sum-of-products implementation of a switching function if all the data lines together represent just one switching variable (or its complement) and each of the select inputs represents a switching variable.

Let's work backward from a specified function of m switching variables for which we have written a canonic sum-of-products expression. The size of multiplexer needed (number of select inputs) is not evident. Suppose we choose a multiplexer that has $m - 1$ select inputs, leaving only one other variable to accommodate all the data inputs. We write an output function of these select inputs and the 2^{m-1} data inputs D_i. Now we plan to assign $m - 1$ of these variables to the select inputs; but how to make the assignment?[4] There are really no restrictions, so it can be done arbitrarily.

The next step is to write the multiplexer output after replacing the select inputs with $m - 1$ of the variables of the given function. By comparing the two expressions term by term, the D_i inputs can be determined in terms of the remaining variable.

EXAMPLE 1

A switching function to be implemented with a multiplexer is

$$f(x, y, z) = \Sigma(1, 2, 4, 7) = x'y'z + x'yz' + xy'z' + xyz$$

Since the function has three variables, the desired multiplexer will have $3 - 1 = 2$ select inputs; half of the dual four-input MUX of Figure 14 will do. The expression for the multiplexer output is

$$f = s_1's_0'D_0 + s_1's_0D_1 + s_1s_0'D_2 + s_1s_0D_3$$

There are no restrictions on how to assign the selector inputs to the variables of the given function; let $s_1 = x$ and $s_0 = y$ arbitrarily. Then

$$f = x'y'D_0 + x'yD_1 + xy'D_2 + xyD_3$$

Comparing this with the original expression for the given function leads to

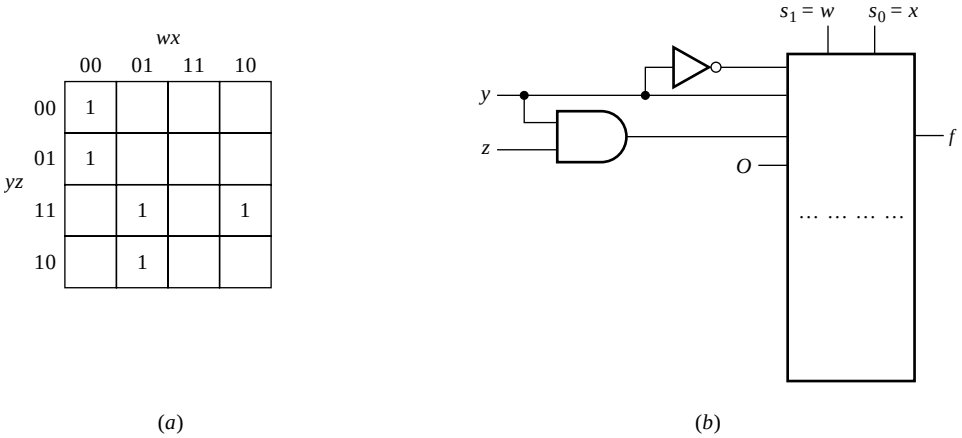
$$D_0 = D_3 = z$$
$$D_1 = D_2 = z'$$

The original function is thus implemented with a four-input multiplexer. ■

There are five other ways that the two select inputs could have been assigned to two of the three switching variables. No conditions need to be satisfied by the choice, so it is arbitrary. However, the specific outcome obtained for the D_i inputs depends on that initial choice.

Short ___
Even ___

[4]For a set of $m - 1$ variables, there are $m!$ ways of assigning $m - 1$ quantities to specific variables.



(*a*) (*b*)

Figure 15 Multiplexer implementation of $f = \Sigma(0, 1, 6, 7, 11)$.

Exercise 7 In the problem of Example 1, choose $s_1 = z$ and $s_0 = x$. Determine the D_i.
Answer[5]

Exercise 8 For practice, choose each of the remaining possible ways of assigning select inputs to the switching variables, and then determine the required D_i; specify the external gates needed. ◆

To implement a switching function of m variables, we have seen that a multiplexer of $m - 1$ select inputs will work. It might be possible in some cases that even a smaller multiplexer can be used. It should be expected that, when possible, this savings in MUX complexity must come at some other cost.

EXAMPLE 2

The function of four variables whose map is shown in Figure 15 is to be implemented by a multiplexer. One with 4 – 1 = 3 select variables is always possible. However, let's explore the possibility of using a multiplexer with only two select variables to implement this function.

Arbitrarily assign the two select inputs s_1 and s_0 to w and x. The expression for the output of the multiplexer is the same one given in Example 1, since this one has the same dimensions. For $wx = s_1s_0 = 00$, that expression reduces to D_0. But for the values $wx = 00$, the expression that covers the 1's in the map is $y'z' + y'z = y'$. Hence, $D_0 = y'$. Similarly, in the 01 column of the map, the expression reduces to D_1 and the map gives $yz + yz' = y$; hence, $D_1 = y$. In the same way, from the 11 column we find $D_3 = 0$ and from the 10 column $D_2 = yz$. (Confirm these.) The rather simple circuit is shown in Figure 15*b*. We find that to imple-

[5] $D_0 = D_3 = y$, $D_1 = D_2 = y'$ ◆

ment a certain specific function of four variables, a multiplexer of order lower than 3 can be used, at the cost of an additional AND gate. (The inverter would be necessary even with a higher-order multiplexer, so it does not count as additional cost.) ■

Exercise 9 In the preceding example, suppose that s_1 and s_0 are identified as y and z instead of w and x. Determine expressions for the data inputs in terms of w and x, and specify the external hardware that will be needed besides the multiplexer. Note the difference in complexity for the two choices of select inputs.
Answer[6]

In the implementation of an arbitrary switching function, different choices for the select inputs lead to different amounts of external hardware for a smaller-than-normal multiplexer. Unfortunately, short of trying them, there is no way to determine which choice will be most economical.

3 DECODERS AND ENCODERS

The previous section began by discussing an application: Given 2^n data signals, the problem is to select, under the control of n select inputs, sequences of these 2^n data signals to send out serially on a communications link. The reverse operation on the receiving end of the communications link is to receive data serially on a single line and to convey it to one of 2^n output lines. This again is controlled by a set of control inputs. It is this application that needs only one input line; other applications may require more than one. We will now investigate such a generalized circuit.

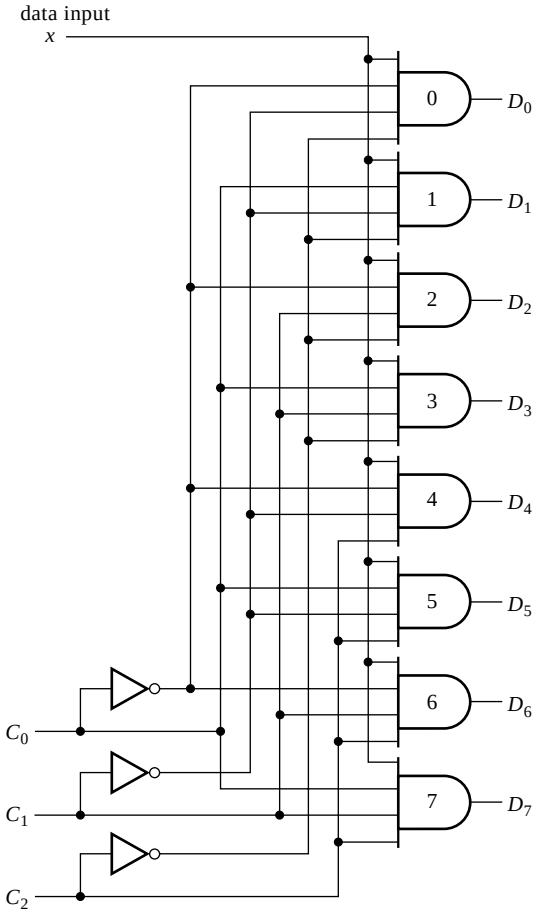
Conceivably, there might be a combinational circuit that accepts n inputs (not necessarily 1, but a small number) and causes data to be routed to one of many, say up to 2^n, outputs. Such circuits have the generic name *decoder*. Semantically, at least, if something is to be decoded, it must have previously been *encoded*, the reverse operation from decoding. Like a multiplexer, an encoding circuit must accept data from a large number of input lines and convert it to data on a smaller number of output lines (not necessarily just one). This section will discuss a number of implementations of decoders and encoders.

Demultiplexers

Refer back to the diagram in Figure 12. The demultiplexer shown there is a single-input, multiple-output circuit. However, in addition to the data input, there must be other inputs to control the transmission of the data to the appropriate data output line at any given time. Such a demultiplexer circuit

Short ____
Even ____

[6] $D_0 = D_1 = w'x'$, $D_2 = w'x$, $D_3 = w \oplus x$; three AND gates and one XOR gate, in addition to a four-input MUX. ◆



(*a*)

Control Inputs			Data Outputs							
C_2	C_1	C_0	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	*x*	0	0	0	0	0	0	0
0	0	1	0	*x*	0	0	0	0	0	0
0	1	0	0	0	*x*	0	0	0	0	0
0	1	1	0	0	0	*x*	0	0	0	0
1	0	0	0	0	0	0	*x*	0	0	0
1	0	1	0	0	0	0	0	*x*	0	0
1	1	0	0	0	0	0	0	0	*x*	0
1	1	1	0	0	0	0	0	0	0	*x*

(*b*)

Figure 16 A demultiplexer circuit (*a*) and its truth table (*b*).

having eight output lines is shown in Figure 16*a*. It is instructive to compare this demultiplexer circuit with the multiplexer circuit in Figure 13. For the same number of control (select) inputs, there are the same number of AND gates. But now each AND gate output is a circuit output. Rather than each gate having its own separate data input, the single data line now forms one of the inputs to each AND gate, the other AND inputs being control inputs.

When the word formed by the control inputs $C_2C_1C_0$ is the binary equivalent of decimal k, then the data input x is routed to output D_k. Viewed in another way, for a demultiplexer with n control inputs, each AND gate output corresponds to a minterm of n variables. For a given combination of control inputs, only one minterm can take on the value 1; the data input is routed to the AND gate corresponding to this minterm. For example, the logical expression for the output D_3 is $xC_2'C_1C_0$. Hence, when $C_2C_1C_0 = 011$, then $D_3 = x$ and all other D_i are 0. The complete truth table for the eight-output demultiplexer is shown in Figure 16*b*.

___ Short
___ Even

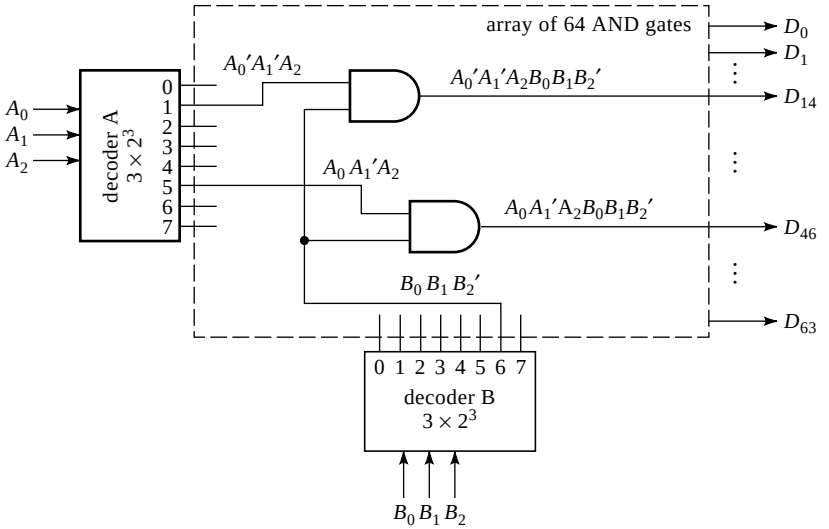


Figure 17 Design of a 6-to-2^6-line decoder from two 3-to-2^3-line decoders with an interconnection matrix of 64 AND gates.

n-to-2^n-Line Decoder

In the demultiplexer circuit in Figure 16, suppose the data input line is removed. (Draw the circuit for yourself.) Each AND gate now has only n (in this case three) inputs, and there are 2^n (in this case eight) outputs. Since there isn't a data input line to control, what used to be control inputs no longer serve that function. Instead, they are the data inputs to be decoded. This circuit is an example of what is called an *n-to-2^n-line decoder*. Each output represents a minterm. Output k is 1 whenever the combination of the input variable values is the binary equivalent of decimal k.

Now suppose that the data input line from the demultiplexer in Figure 16 is not removed but retained and viewed as an enable input. The decoder now operates only when the enable x is 1. Viewed conversely, an n-to-2^n-line decoder with an enable input can also be used as a demultiplexer, where the enable becomes the serial data input and the data inputs of the decoder become the control inputs of the demultiplexer.[7]

Decoders of the type just described are available as integrated circuits (MSI); $n = 3$ and $n = 4$ are quite common. There is no theoretical reason why n can't be increased to higher values. Since, however, there will always be practical limitations on the fan-in (the number of inputs that a physical gate can support), decoders of higher order are often designed using lower-order decoders interconnected with a network of other gates.

An illustration is given in Figure 17 for the design of a 6-to-2^6-line decoder constructed from two 3-to-2^3-line decoders. Each of the component decoders

[7]In practice, the physical implementation of the decoder with enable is carried out with NAND gates. In that case, it is the complements of the outputs in the circuit under discussion that are obtained, and the enable input is inverted before it is applied to the NAND gates. These are practical details that do not change the principles described here.

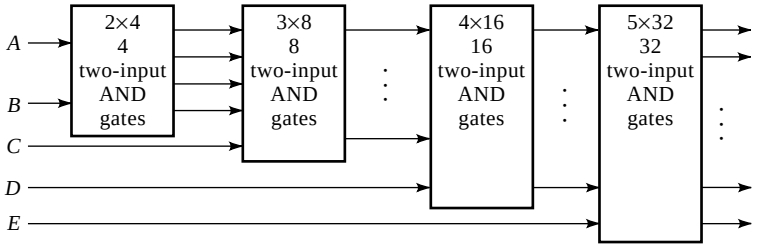


Figure 18 Design of tree decoder.

has eight outputs. Each of the outputs from the A decoder must be ANDed with each of the outputs from the B decoder to yield one of the 64 outputs from the complete decoder. Thus, in addition to the 8 three-input AND gates in each component decoder, there are 64 two-input AND gates in the interconnection network. Only two of these are shown explicitly in Figure 17.

Exercise 10 A 6-to-2^6-line decoder is to be designed using the structure of Figure 16. Specify the number of AND gates and the total number of input lines to all gates. Compare this with the design in Figure 17. ◆

Tree Decoder

When higher-order decoders are designed in a hierarchy of several stages of lower-order ones, a practical difficulty with fan-out (number of gates driven by one terminal) results. (By a hierarchy of stages we mean, for example, two 3×8 stages to form a 6×64 decoder, as in Figure 17; then two 6×64 stages to form a 12×2^{12} decoder; and so on.) Even in Figure 17, each gate in the component decoders drives eight other gates. In the next level of the hierarchy, each of the outputs from the gates in the next-to-last level will have to drive 64 other gates.

This problem is overcome, but only partially, by the decoder design illustrated in Figure 18, called a *tree decoder*. The first stage is a 2-to-4-line decoder. A new variable is introduced in each successive stage; it or its inverse becomes one input to each of the two-input AND gates in this stage. The second input to each AND gate comes from the preceding stage. For example, one of the outputs of the second stage will be $AB'C$. This will result in two outputs from the next stage, $AB'CD$ and $AB'CD'$. This design does avoid the fan-out problem in the early stages but not in the later stages. Nevertheless, the problem exists only for the variables introduced in those stages. Any remedies required will have to be used for relatively few variables, as opposed to the large number needed by the design of Figure 17.

Decoders as General-Purpose Logic Circuits: Code Conversion

Since each output from an n-to-2^n-line decoder is a canonic product of literals, simply ORing all the outputs produces a canonic sum of products. And since every switching function can be expressed as a canonic sum of products, it fol-

___ Short
___ Even

Decimal Digit	Inputs: Excess-3 w	x	y	z	Outputs: Seven-Segment S_1	S_2	S_3	S_4	S_5	S_6	S_7
0	0	0	1	1	1	1	1	1	1	1	0
1	0	1	0	0	0	0	0	1	1	0	0
2	0	1	0	1	1	0	1	1	0	1	1
3	0	1	1	0	0	0	1	1	1	1	1
4	0	1	1	1	0	1	0	1	1	0	1
5	1	0	0	0	0	1	1	0	1	1	1
6	1	0	0	1	1	1	0	0	1	1	1
7	1	0	1	0	0	0	1	1	1	0	0
8	1	0	1	1	1	1	1	1	1	1	1
9	1	1	0	0	0	1	1	1	1	0	1

Figure 19 Excess-3 to seven-segment code conversion.

lows that every switching function can be implemented by an n-to-2^n-line decoder followed by an OR gate. (If 2^n exceeds the fan-in limitation of the OR gate, additional levels of OR gates will be needed.) Indeed, if more than one function of the same variables is to be implemented, the same decoder can be used, with each function having its own set of OR gates.

One major class of logic circuits is known as a *code converter*. This is a circuit that accepts as inputs the digits of a word that expresses some information in a particular code and that yields as outputs the digits of a word in a different code. (See Chapter 1 for an introduction to codes.) We will illustrate the use of a decoder as a code converter by designing a circuit to convert from excess-3 code to seven-segment code. (These codes were given in Figure 4 and Exercise 12 in Chapter 1; they are repeated here in Figure 19.)

Assume that a 4-to-16-line decoder is available. Since there are only 10 valid excess-3 code words, only 10 of the 16 AND gate outputs ever become 1. So only those 10 outputs from a 4-to-16-line decoder will be used. They are indicated in Figure 19 by their decimal equivalents.

Figure 19 is the truth table for each of seven output functions (the S_i) in terms of the four input variables. The circuit external to the decoder will consist of seven OR gates, one for each segment. Only one decision needs to be made: Which outputs from the decoder should become inputs to each OR gate? This is answered for each segment by listing the minterm numbers corresponding to each code word for which that segment output has the value 1. The minterm lists for the outputs corresponding to some of the segments are as follows:

$$\begin{aligned} S_3 &= \Sigma(3, 5, 6, 8, 10, 11, 12) \\ S_4 &= \Sigma(3, 4, 5, 6, 7, 10, 11, 12) \\ S_5 &= \Sigma(3, 4, 6, 7, 8, 9, 10, 11, 12) \\ S_6 &= \Sigma(3, 5, 6, 8, 9, 11) \end{aligned} \tag{8}$$

Short ___
Even ___

Only one of the OR gates (the one for S_6) is shown in Figure 20; there should be six others. Then, when an excess-3 code word corresponding to a decimal

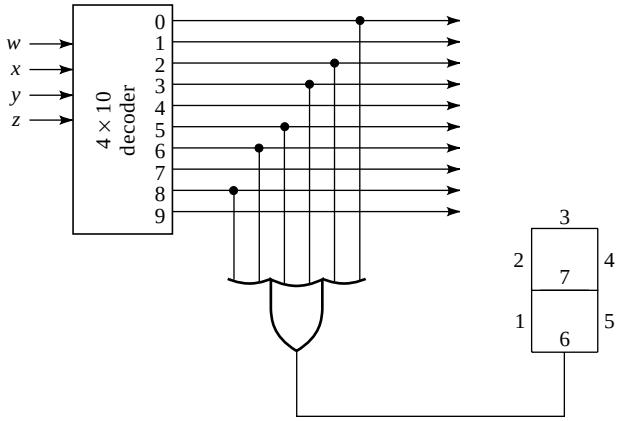


Figure 20 Excess-3 to seven-segment code converter.

digit appears at the input, the appropriate segments will light up, displaying the digit.

Exercise 11 Write the minterm lists for the three segments whose minterm lists were not given in (8). Confirm the inputs to the OR gate in Figure 20. ◆

4 READ-ONLY MEMORY (ROM)

A circuit for implementing one or more switching functions of several variables was described in the preceding section and illustrated in Figure 20. The components of the circuit are

- An $n \times 2^n$ decoder, with n input lines and 2^n output lines
- One or more OR gates, whose outputs are the circuit outputs
- An interconnection network between decoder outputs and OR gate inputs

The decoder is an MSI circuit, consisting of 2^n n-input AND gates, that produces all the minterms of n variables. It achieves some economy of implementation, because the same decoder can be used for any application involving the same number of variables. What is special to any application is the number of OR gates and the specific outputs of the decoder that become inputs to those OR gates. Whatever else can be done to result in a general-purpose circuit would be most welcome.

The most general-purpose approach is to include the maximum number of OR gates, with provision to interconnect all 2^n outputs of the decoder with the inputs to every one of the OR gates. Then, for any given application, two things would have to be done:

- The number of OR gates used would be fewer than the maximum number, the others remaining unused.
- Not every decoder output would be connected to all OR gate inputs.

This scheme would be terribly wasteful and doesn't sound like a good idea.

Instead, suppose a smaller number, m, is selected for the number of OR gates to be included, and an interconnection network is set up to interconnect

___ Short
___ Even

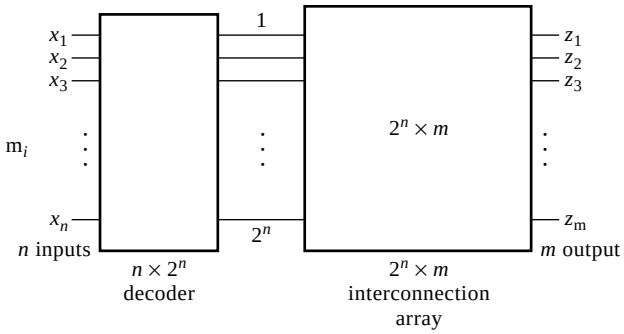


Figure 21 Basic structure of a ROM.

the 2^n decoder outputs to the m OR gate inputs. Such a structure is illustrated in Figure 21. It is an LSI combinational circuit with n inputs and m outputs that, for reasons that will become clear shortly, is called a *read-only memory* (ROM). A ROM consists of two parts:

- An $n \times 2^n$ decoder
- A $2^n \times m$ array of switching devices that form interconnections between the 2^n lines from the decoder and the m output lines

The 2^n output lines from the decoder are called the *word* lines. Each of the 2^n combinations that constitute the inputs to the interconnection array corresponds to a minterm and specifies an *address*. The *memory* consists of those connections that are actually made in the connection matrix between the word lines and the output lines.

Once made, the connections in the memory array are permanent.[8] So this memory is not one whose contents can be changed readily from time to time; we "write" into this memory but once. However, it is possible to "read" the information already stored (the connections actually made) as often as desired, by applying input words and observing the output words. That's why the circuit is called *read-only* memory.[9]

Before you continue reading, think of two possible ways in which to fabricate a ROM so that one set of connections can be made and another set left unconnected. Continue reading after you have thought about it.

The one-time "writing" into memory can be done as follows:

- A ROM can be almost completely fabricated except that none of the connections are made. Such a ROM is said to be *blank*. Forming the connections for a particular application is called *programming* the ROM. In the process of programming the ROM, a *mask* is produced to cover those connections that are not to be made. For this reason, the blank form of the ROM is called *mask programmable.*[10]

[8]In certain designs, it is possible for the connections to be *erasable;* this will be described shortly.

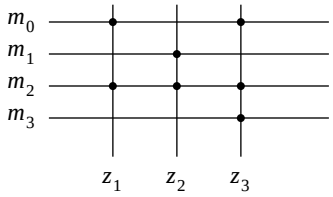
[9]Although "memory" appears in its name, a ROM does not have memory in the usual sense. As will be described in Chapters 5 and 6, memory is a characteristic of sequential, but not combinational, circuits.

[10]The mask, requiring minute attention, is expensive to produce. Hence, mask-programmable ROMs are used only when the cost is justified by very large production runs.

Short ____
Even ____

x_1	x_2	z_1	z_2	z_3
0	0	1	0	1
0	1	0	1	0
1	0	1	1	1
1	1	0	0	1

(*a*)



(*b*)

Figure 22 A ROM truth table and its program.

- A ROM can be completely fabricated such that *all* potential connections have been made. Such a ROM is also said to be blank. Programming the ROM for a specific application in this case consists of *opening* those connections that are unwanted. In this case, the blank ROM is said to be *field programmable* (designated PROM). The connections are made by placing a *fuse* or *link* at every connection point. In any specific application, the unwanted connections are opened or "blown out" by passing pulses of current through them. A measure of PROM cost is the number of fusible links, $2^n \times m$.[11]

Once a ROM has been programmed, an input word $x_1x_2 \ldots x_n$ activates a specific word line corresponding to the minterm formed by the specific values of the x_i. The connections in the output matrix result in the desired output word.

EXAMPLE 3

Figure 22*a* gives the truth table for the interconnection matrix of a $2^2 \times 3$ ROM. The truth table leads to the ROM program represented by the solid dots at the intersections of the input and output word lines in Figure 22*b*. Each input word defines an output word, as required by the truth table. If the input word is 01 (corresponding to minterm m_1), for example, only output line z_2 will be activated because that is the only connection with m_1 in the connection matrix. Hence, the output word will be 010, as confirmed also from the truth table. (Confirm from the truth table that the rest of the program is correct.) ■

Exercise 12 A ROM is to be programmed to implement the conversion from excess-3 to seven-segment code whose table was given in Figure 19. ROMs come in standard sizes, and $m = 7$ is not one of them. The next larger standard size is $m = 8$. Hence, the truth table will have six more rows and one more column than shown in Figure 19. (Specify what the entries in the truth table will be for these extra rows and column.) Draw the appropriate number of crossing lines for the input and output words. Using the truth table, program the ROM by putting dots at the appropriate intersections of the two words. ◆

[11]Some PROMs are fabricated so that it is possible to restore them to their blank condition after they have been programmed for a specific application; these are *erasable* PROMs, or EPROMs. They have some clear advantages over the nonerasable kind, but their cost is correspondingly higher.

Short
Even

In Exercise 12 the number of entries in the truth table (which corresponds to the number of links between the input and output words) is $2^n \times m = 16 \times 8 = 128$. Of these, fully half represent don't-cares. There are cases far worse than this; sometimes as few as 1 percent of the links are used, resulting in considerable "waste" in such ROM implementations. Another implementation that avoids this waste would be most welcome. That's the subject of the next section.

5 OTHER LSI PROGRAMMABLE LOGIC DEVICES

One way of looking at the ROM discussed in the previous section is as a device with a specific structure (a set of AND gates and a set of OR gates) that a designer can use to achieve desired outputs by making a few modifications. We might say that the ROM has been "programmed" to produce its specific outputs. There are other structures that have this property, namely, programmability. A generic name for them is *programmable* (or programmed) *logic device* (PLD).

The ROM implements logic functions as sums of minterms. For n input variables there are 2^n minterms and, hence, 2^n AND gates, each one with n inputs. As just discussed, in a number of important logic functions, many of the AND gates and the links connecting them to the output OR gates are unused. We will now discuss two implementations in which some of this "waste" is avoided.

Programmed Logic Array (PLA)

The canonic sum-of-products implementation of a logic function is wasteful in two ways: in the number of AND gates used (as many as there are minterms, 2^n) and in the number of inputs to each AND gate (n). Suppose we contemplate a reduced (possibly minimal) sum-of-products implementation. Given a logic function of n variables, the largest number of terms in a minimal sum-of-products expression representing this function is 2^{n-1}—just half the number of minterms. (See Problem 36 in Chapter 3.) That means a savings of 50 percent in AND gates for the worst single-output case. Since there will be a reduced set of inputs to the AND gates, this saving in gates is paid for by the need to program not only the outputs of the AND gates but their inputs as well. The structure of the circuit that results is called a *programmable* (or *programmed*) *logic array* (PLA). It is illustrated in Figure 23 for the case $n = 3$ input variables, $m = 4$ output functions, and four AND gates.

The diagram in Figure 23 is not a circuit diagram but a schematic diagram. A single line is shown to represent all inputs to each AND and OR gate. The number of input lines to each AND gate should be $2n$, twice the number of inputs, to accommodate the possibility of connecting each variable or its complement to each AND gate. The number of input lines to each OR gate should equal the number of AND gates, say p. (For simplicity and without fear of confusion, even the gate symbols can be omitted.) The programmed connections between the inputs and the AND gates, and between the AND-gate outputs and the OR gates for a specific set of output functions are shown by the heavy dots at the intersections.

Short ___
Even ___

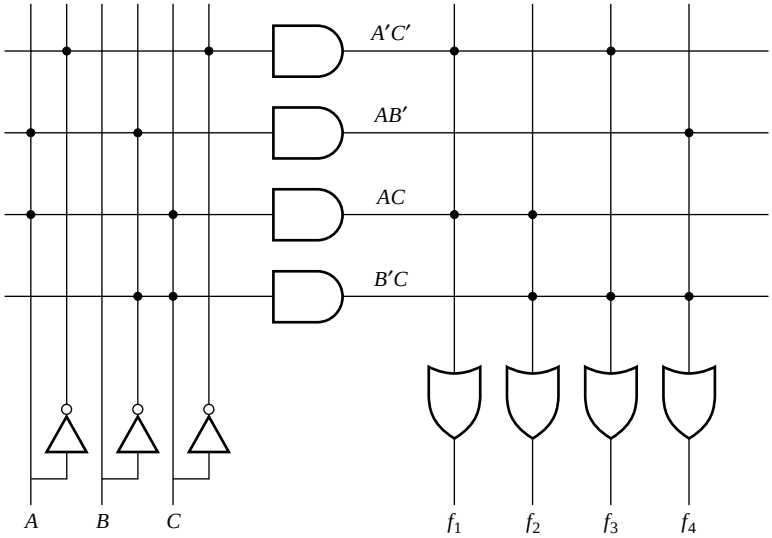


Figure 23 Structure of a PLA.

Maps of the four output functions and minimal sum-of-products expressions are shown in Figure 24. In this example, a total of only four product terms covers all functions, so only four AND gates are needed in the implementation. Two sets of lines must be programmed: the input lines and the output lines. To do this, we construct a *programming table* as follows:

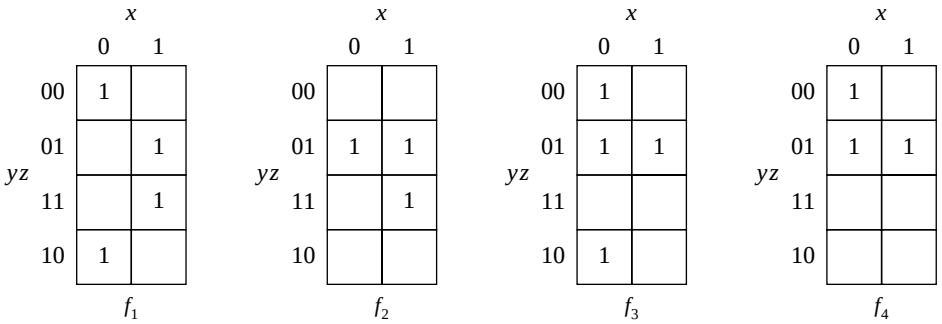
- The implicants (product terms) are listed as row headings.
- In one set of columns, the headings are the input variables; this part of the table must provide the information that tells which variables (or their complements) are factors in each implicant.
- In a second set of columns, the headings are the output functions; this part of the table must provide the information that indicates the output gate to which each implicant (AND-gate output) is directed.

In the first set of columns, if a variable (uncomplemented) is present in a particular row, the corresponding entry is 1; if its complement is present, the entry is 0. If neither is present, the entry can be left blank, but it is preferable to show some symbol instead; a dash is often used.

In the second set of columns, corresponding to the output functions, if a particular function covers a particular implicant, then the corresponding entry is 1; otherwise it could be left blank, but it is customary to enter a dot. To illustrate, consider row 4. Since the implicant is $y'z$, the entry in column z is 1, that in column y is 0, and that in x is a dash. In the output columns, only f_1 does not cover implicant $y'z$; hence, the entry will be 1 in every column in row 4 except the f_1 column, where the entry is •. Confirm the remaining rows.

Once the programming is done, fabricating the links (connection points) in a PLA is carried out in a similar manner as for the ROM. The PLA is either mask programmable or field programmable (FPLA). In the case of the FPLA, with p = the number of AND gates, there will be $2np$ links at the inputs and mp

___ Short
___ Even



Product Term	Inputs x	y	z	Outputs f_1	f_2	f_3	f_4	
1: $x'z'$	0	–	0	1	•	1	•	$f_1 = x'z' + xz$
2: xy'	1	0	–	•	•	•	1	$f_2 = xz + y'z$
3: xz	1	–	1	1	1	•	•	$f_3 = x'z' + y'z$
4: $y'z$	–	0	1	•	1	1	1	$f_4 = xy' + y'z$

Figure 24 Programming the PLA.

links at the outputs. For the example in Figure 23, the number of links is 4(6 + 4) = 40. Only 16 of these are to be kept, meaning that, during field programming, 24 links are to be blown out. Typical PLAs have many more inputs, outputs, and AND gates than are shown in the example in Figure 23. (IC type 82S100, for example, has $n = 16$, $m = 8$, and $p = 48$.)

When a set of switching functions is presented for implementation with a PLA, a design goal would be reduction in p (the number of AND gates). The economy achieved is not derived from a reduction in the production cost of gates. (The production cost of an IC is practically the same for one with 40 gates as it is for one with 50 gates.) Rather, the removal of one AND gate eliminates $2n + m$ links; the main source of savings is the elimination of a substantial number of links due to the elimination of each AND gate. On the other hand, reduction of the number of AND gates to a minimum does not mean that each function should be minimized or that all implicants should be *prime* implicants. The implicants should be chosen so that as many as possible of them are common to many of the output functions.

Programmed Array Logic (PAL)

A ROM has a large number of fusible links ($m \times 2^n$) because of the large number (2^n) of AND gates. Programming of links is performed only on the outputs from the AND gates. In a PLA, the number of links is drastically reduced by reducing the number of AND gates. The latter is done by changing the expression representing the switching function from a canonic sum-of-products form to a sum of products with fewer terms. The price paid is the need to program not only the outputs from the AND gates, but also the inputs to the AND gates. What other possibility for programming is there beyond the two cases of (a) programming the outputs of the AND gates and (b) programming both the inputs

Short ___
Even ___

Product Term		Inputs												Outputs					
Number	Function	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6
1														•	•	•	•	•	1
2														•	•	•	•	•	1
3														•	•	•	•	•	1
4														•	•	•	•	•	1
5														•	•	•	•	1	•
6														•	•	•	•	1	•
7	$x_1x_2'x_5x_7x_{11}'x_{12}$	1	0	–	–	1	–	1	–	–	–	0	1	•	•	•	1	•	•
8														•	•	•	1	•	•
9														•	•	1	•	•	•
10														•	•	1	•	•	•
11														•	1	•	•	•	•
12														•	1	•	•	•	•
13														1	•	•	•	•	•
14														1	•	•	•	•	•
15														1	•	•	•	•	•
16														1	•	•	•	•	•

Figure 25 Programming table for a PAL example.

and the outputs? We're sure you answered, "programming only the inputs." This is a possibility, but is it worthwhile?

In the case of the ROM, there is no need to program the inputs because, for *any* function of n variables, there will be the same (large) number of AND gates. In the same way, if the number of OR gates at the output could be fixed, then programming the outputs of the AND gates could be avoided.

In many circuits with multiple outputs, even though the outputs are functions of a large number of input variables, the number of product terms in each output is small. Hence the number of AND gates that drive each OR gate is small. In such cases, permanently fixing the number of OR gates and leaving only the programming of the AND gate inputs for individual design might make economic sense. The resulting circuit is called *programmed array logic* (PAL).[12] The number of fusible links in a PAL is only $2np$. Standard PALs for a number of low values of p exist. For example, the PAL16L8 has a maximum of 16 inputs and 8 outputs.

A programming table for a PAL is similar to the one for a PLA. A case with six outputs is illustrated in Figure 25. A ROM with 12 input variables would require $2^{12} = 4096$ AND gates. However, let's assume that for some possible cases, the canonic sum-of-products expression can be reduced to 16 implicants, only one of which is shown in Figure 25. The entries in the table would have the same meanings as those for the PLA. However, for the PAL, the output columns would be fixed by the manufacturer on the basis of the number of AND gates already connected to each OR gate.

In the present case, two of the output OR gates are each driven by four AND gates; the remaining four OR gates are each driven by two AND gates. For any

[12]PAL is a registered trademark of Advanced Micro Devices.

___ Short
___ Even

given design problem, the first step is to obtain an appropriate sum-of-products expression, just as in the case of a PLA implementation. The input connections are indicated in the table as in the case of the PAL: an entry is a 1 if a variable appears uncomplemented in an implicant, a 0 if it appears complemented, and a dash if it does not appear at all. This is illustrated for one row in Figure 25. The number of fusible links in this example is $2 \times 12 \times 16 = 384$. This is 20 percent fewer than the number of links of a PLA having the same dimensions. Typically, however, PLAs have many more AND gates and so, for a PAL, the number of links would typically be many times more than the number for a comparable PLA.

Exercise 13 Suppose two of the rows of inputs in Figure 25 are as follows:

0 1 0 – 0 – – 1 – – – –

1 0 1 – – 0 – – 1 1 – –

What are the corresponding product terms? ◆

Further attention will be devoted to PLDs in Chapter 8. Attention will also be given there to the use of hardware description languages in designs using PLDs.

CHAPTER SUMMARY AND REVIEW

In Chapter 3, designs were carried out with primitive gates in SSI circuits. This chapter advanced the design process to more complex circuits implemented in MSI units. The topics included were

- Binary adder
- Full adder
- Ripple-carry adder
- Carry-lookahead adder
- Binary subtractor
- Two's complement adder and subtractor
- One's complement adder and subtractor
- Multiplexer
- Data input
- Select input
- Implementation of general-purpose logic circuits with multiplexers
- Demultiplexer
- Data input lines
- Control input lines
- Decoder
- $n \times 2^n$-line decoder
- Tree decoder
- Implementation of general-purpose logic circuits with decoders
- Code conversion
- Read-only memory (ROM)
- $n \times 2^n$ decoder
- $2^n \times m$ interconnection array
- Programming a ROM

Short ___
Even ___

- Mask-programmable ROM
- Field-programmable ROM
- Programmable logic device (PLD)
- Programmed logic array (PLA)
- Programmed array logic (PAL)

PROBLEMS

1 **a.** Analyze each of the full adder circuits shown in Figure P1 and write expressions for the output of each intermediate gate.
b. Obtain logic expressions for the sum and carry circuit outputs.
c. Verify that these expressions are equivalent to the sum and carry functions in equations (1) in the text.

2 **a.** A 4-bit carry-lookahead adder is to be designed. In equation (7) in the text for the carry function, let $i = 0$ and let j range from 0 to 4. Write the resulting expressions for C_1, C_2, C_3, and C_4.
b. Construct the logic diagram for the 4-bit carry-lookahead whose schematic diagram is given in Figure 8.

3 A 4-bit binary number $Y = y_3y_2y_1y_0$ is to be multiplied by a 3-bit binary number $X = x_2x_1x_0$. Use two 4-bit adders and other gates that you might need to implement this operation, and draw the corresponding diagram.

4 Prove formally that if the propagate variable P_i for a carry-lookahead adder is defined as $A_i + B_i$ instead of $A_i \oplus B_i$, the sum and carry outputs of the adder will still be computed correctly. (Give an informal proof also.) Which definition is better for implementation purposes?

5 Design a circuit for overflow detection in the one's complement adder/subtractor shown in Figure 11.

6 **a.** Show the connections on a schematic diagram of a dual four-input multiplexer for implementing the sum and carry functions of a full adder.
b. Repeat using a 3-to-2^3-line decoder.

7 Realize each of the following functions using an 8×1 multiplexer.

a. $f = \Sigma(0, 1, 10, 11, 12, 13, 14, 15)$
b. $f = \Sigma(0, 3, 4, 7, 10)$
c. $f = \Sigma(0, 3, 4, 6, 7, 8, 12)$
d. $f = \Sigma(1, 2, 5, 8, 11, 12, 14)$

8 Realize each of the functions in Problem 7 using half of a dual 4×1 multiplexer and the minimum number of external gates.

9 Repeat Problem 7 using a 3-to-2^3-line decoder.

10 Use a dual four-input multiplexer to implement each of the following pairs of functions with the fewest external gates.

a. $f_1 = \Sigma(0, 4, 5, 7, 9, 11)$, $f_2 = \Sigma(2, 3, 5, 6, 10, 13)$
b. $f_1 = \Sigma(0, 4, 7, 10, 12, 14, 15)$, $f_2 = \Sigma(2, 7, 8, 9, 12, 13, 14, 15)$

11 **a.** Show how to connect a 4-bit MSI adder to serve as a BCD-to-excess-3 code converter.
b. Repeat using a 4-to-10-line (BCD-to-decimal) decoder and four AND gates.

12 Design a BCD-to-decimal decoder using two 2-to-4-line decoders and a minimum of interconnecting AND gates.

___ Short
___ Even

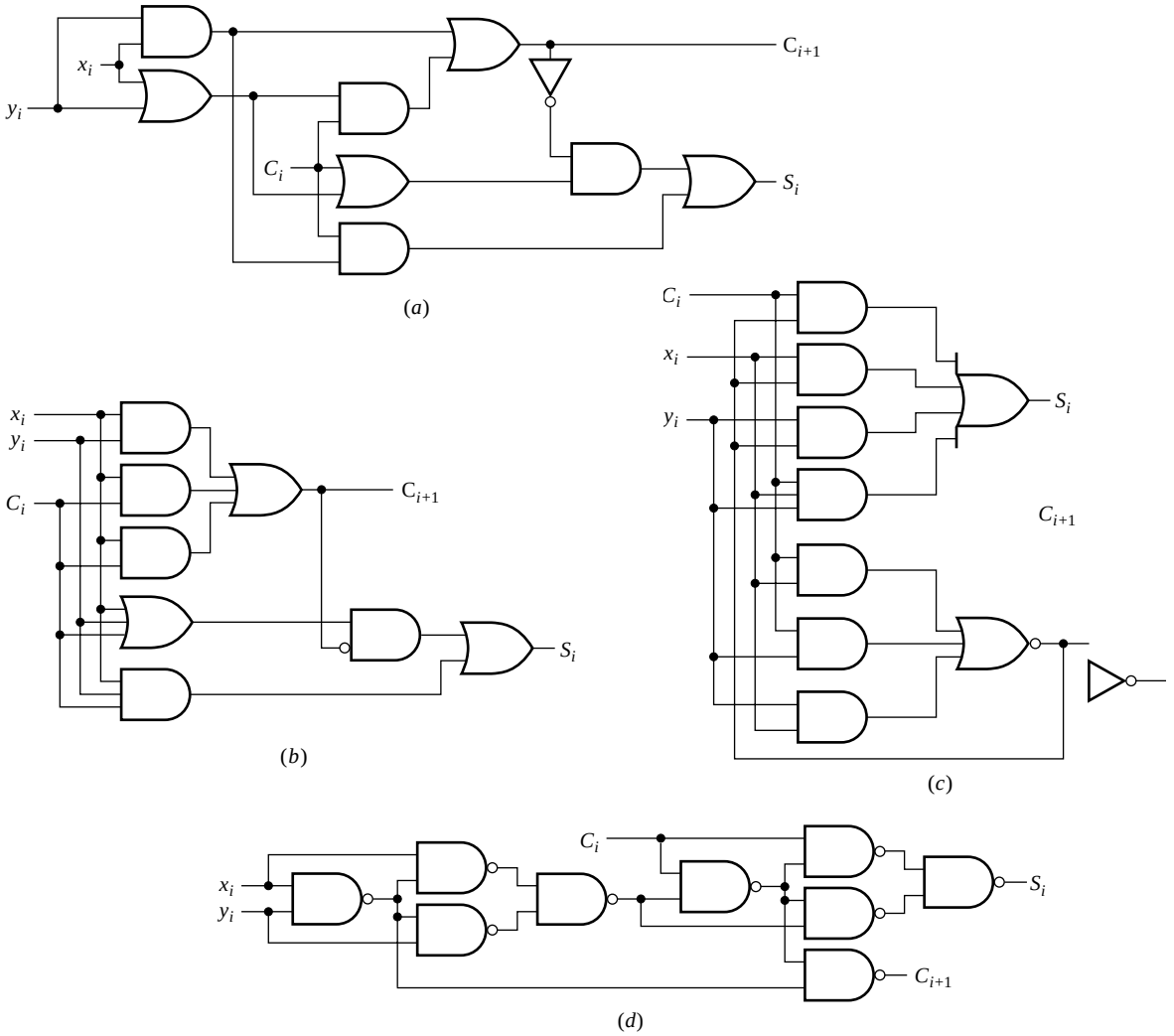


Figure P1

13 A circuit is to accept two 2-bit binary numbers x_1x_0 and y_1y_0 and emit the product as a 4-bit binary number $z_3z_2z_1z_0$. (Review binary multiplication in Chapter 1 if you need to.)

- **a.** The result is to be achieved by a (possibly) multilevel circuit with two-input gates. Determine appropriate expressions for each output. How many levels of gates does each output have?
- **b.** Design a circuit using a 4-to-2^4-line decoder with external OR gates.

14 Examine late editions of manufacturers' data books.

- **a.** What is n for the largest n-to-2^n-line decoders?
- **b.** Note what the standard sizes of ROMs are.
- **c.** What are some representative dimensions of a PLA chip?
- **d.** What are some representative dimensions of a PAL?
- **e.** Is there a BCD adder in a single MSI package?

Short ____
Even ____

15 A switching function of n variables is to be implemented by an n-to-2^n-line decoder followed by an external OR gate. The physical gate available for this purpose has both an OR and a NOR output. (It is an ECL gate.) For practical reasons (to avoid fan-in problems), it would be best to try to reduce the number of inputs to an external gate.

- **a.** Describe how to implement the function using the available physical gate if the number of minterms contained in the function is more than $2^{n-1} = 2^n/2$.
- **b.** Illustrate with the following function:

$$f = \Sigma(0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 13, 14, 15)$$

16 **a.** Design a BCD-to-decimal decoder using the minimal number of two-input AND gates.
b. Repeat, using two 2-to-4-line decoders and a few interconnecting AND gates.

17 **a.** Use two identical n-to-2^n-line decoders with enable inputs to construct an $(n+1)$-to-2^n-line decoder without enable. Show how the outputs are obtained.
b. Illustrate with two 2-to-4-line decoders.

18 Design an octal to binary encoder. This is a circuit with 8 inputs, x_i, and 3 outputs, z_i. Only one of the outputs is 1 at any one time. Octal digit k is represented by $x_k = 1$.

19 A decimal-digit code converter from 2-out-of-5 to seven-segment code is to be designed. A number of different possibilities are to be explored, assuming that only valid code words will occur as inputs.

- **a.** Draw a circuit diagram using a complete 5×2^5 decoder design.
- **b.** Assuming a design using discrete gates:
  - **i.** Draw a circuit for a sum-of-minterms design. (This would constitute a partial decoder.)
  - **ii.** The AND gates in the preceding design are five-input gates. Is it possible to use the same structure but with two-input gates? Justify your answer.
  - **iii.** Carry out a minimal sum-of-products design that uses 11 AND gates and 7 OR gates, each with no more than three inputs.
  - **iv.** Consider a minimal product-of-sums design. Is this more economical than the minimal sum-of-products design?
  - **v.** Now suppose that, in addition to valid code words, invalid ones can also occur. Modify the best of the preceding designs so that, whenever there is an invalid code word, the symbol E (for error) is displayed.

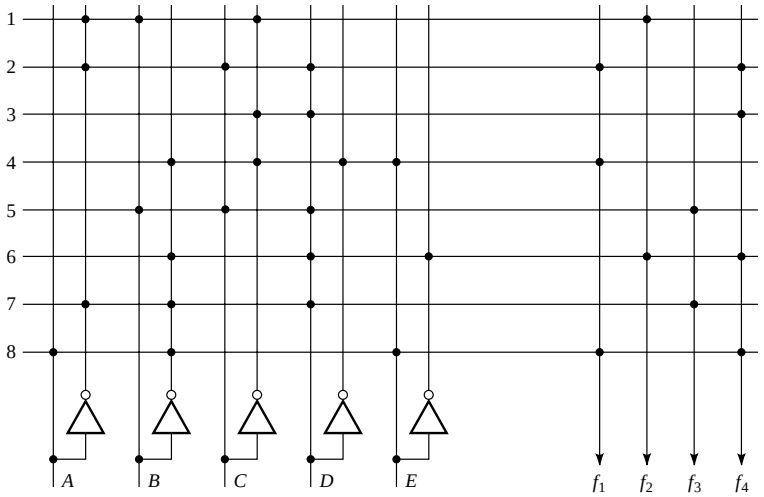
20 The code converter in Problem 19 is to be designed with a ROM. The closest-size ROM available is a $2^5 \times 8$. Construct the required programming table. Specify the number of links.

21 The code converter in Problem 19 is to be implemented with a PLA. A 5×8 PLA with 12 AND gates is available. Draw a programming diagram for implementing the desired code converter. Specify the number of links.

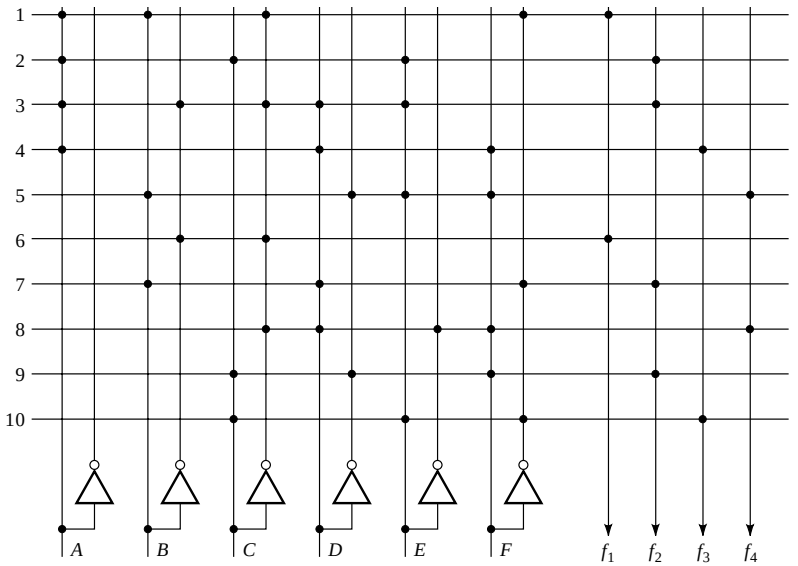
22 **a.** Suppose the circuit in Problem 13 is to be implemented with a $2^4 \times 4$ PROM. Show the programming table and draw an appropriate diagram.
b. Suppose instead that the circuit is to be implemented by a 4×4 PLA with 10 AND gates. Show the programming diagram (in the form of Figure 23 in the text). Compare the number of links with those of the PROM implementation. Construct the programming table in the form of Figure 25 in the text.
c. Now suppose that the circuit is to be implemented by a PAL. Construct the programming table in the form of Figure 25 in the text.

23 A combinational circuit having three inputs and six outputs is to be designed. The output word is to be the square of the input word.

- **a.** Design the circuit using a ROM that has the smallest possible dimensions. Construct the truth table and specify the number of links.



(*a*)



(*b*)

Figure P24

b. Design the circuit using a PLA with the fewest number of product terms. Construct the programming diagram and specify the number of links.

24 The programming diagrams for two PLAs are shown in Figure P24.

a. Write the equations of the outputs realized by each PLA. Specify the number of links.

b. The same functions are to be implemented with a ROM. Specify the dimensions of the ROM and the number of links. Set up its programming table.

c. The same functions are to be implemented with a PAL. Is it possible to do so? If so, set up the programming table and specify the number of links. If it is not possible, explain why not.

Short ____
Even ____

25 (Review Chapter 1 on Hamming codes if you need to.) Using an n-to-2^n-line decoder (for an appropriate n) and any additional logic:

a. Design the error-correcting logic for a single-error-correcting Hamming code assuming 3 *message* bits in each code word. The outputs of the circuit shoud be

- E, indicating that an error has been detected
- IV, indicating that the MSG output is invalid (obviously, IV is 0 when no error, or only a single error, has occurred)
- MSG, a 3-bit output that contains the corrected transmitted message in the cases of zero and one error

b. Design the single-error-correcting and *double-error-detecting* (SEC-DED) logic for an error-correcting Hamming code extended by the addition of a parity bit over all (that is, message and parity) positions. Assume 3 *message* bits in each code word. The output signals and their meanings are to be the same as in part *a*.

26 Explain in words the behavior of the diagram in Figure P26. (The open-headed arrows represent multiple-bit inputs and outputs.)

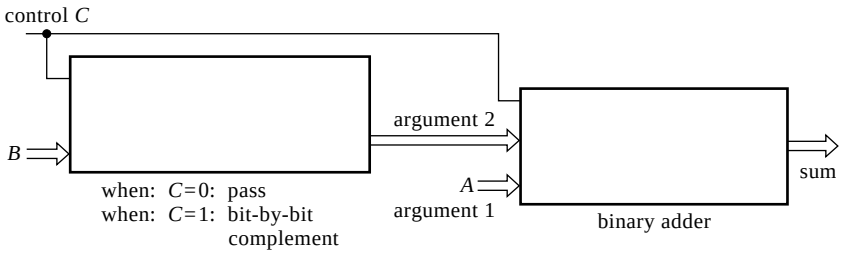


Figure P26

27 A microprocessor (μp) outputs three control signals that have the meanings given in the following table. (No knowledge of μp is necessary to solve this problem.)

R'	W'	M/I'O'	
0	1	1	μp wants to read memory
1	0	1	μp wants to write to memory
0	1	0	μp wants to read an input/output device
1	0	0	μp wants to write to an input/output device
1	1	×	μp wants none of the preceding operations

a. Design a logic circuit using a suitable multiplexer and minimal additional logic to transform these three signals into the following four signals, each representing an operation:

(M R)', (M W)', (IO R)', (IO W)'

When any of the operations is desired (not desired), the value of the corresponding signal is to be 0 (1).

b. Design a multiplexer implementation to perform the inverse transformation.

28 The 4-bit lookahead unit shown in Figure P28*a* receives generate and propagate variables from units 0 through 3 comprising a similar group. It also receives *C*, the carry input to unit

___ Short
___ Even

0 of the group. It computes C_0, C_1, and C_2, which are the carry outputs from units 0, 1, and 2, respectively. It also computes the generate and propagate variables, G and P, for the whole group. The carry outputs are generated in parallel, not in ripple fashion.

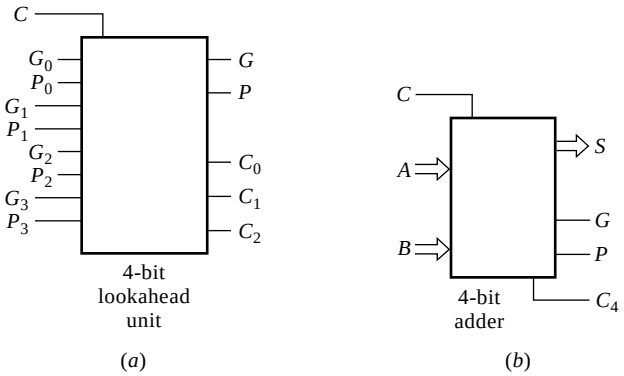


Figure P28

- **a.** Derive equations for all the outputs, and show the implementation.
- **b.** Using 4-bit lookahead units of the above type and 4-bit adders of the type shown in Figure P28*b*, draw the logic diagram for a 48-bit adder using a single-level lookahead. (The open arrows represent multiple-bit inputs and outputs—in this case, 4 bits. *A*, for example, stands for a vector of 4 bits: A_0, A_1, A_2, A_3.)
- **c.** Repeat part *b* using two levels of lookahead, in which the G and P outputs of the first-level lookahead units feed the G_i and P_i inputs of the second-level lookahead units. Compare with respect to speed with the design of part *b*.

29 This problem concerns the design of a 4-bit lookahead subtractor (Figure P29). The 4-bit vector B ($B_3B_2B_1B_0$) is to be subtracted from 4-bit vector A. The borrow input C_0 is 1 if and only if the next lower unit is borrowing a 1 from this unit. The 4-bit vector D is the difference output, and C_4 is the borrow output. G and P are generate and propagate variables from the whole unit.

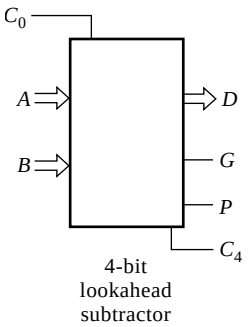


Figure P29

- **a.** Give an expression for each output and show the implementation.
- **b.** As in Problem 28, there will be more than one way to define the propagate variable. Give these definitions and compare the differences in their implementation.
- **c.** Suppose that multiple-bit subtraction is to be carried out. For this purpose, can 4-bit lookahead units, of the type described in Problem 28 in the context of addition, be used with 4-bit lookahead subtractors of the type defined here? Justify your answer.
- **d.** Using 4-bit subtractors of the type described in this problem, and also suitable 4-bit lookahead units, design a 24-bit lookahead subtractor.

Short ____
Even ____

30 An 8-input priority encoder (Figure P30) has eight request inputs: $I(7\ldots0)$. A logic 1 on any of these lines denotes the presence of a request from the corresponding source for some service. The priority varies from the highest for 7 to the lowest for 0. Output LR (Local Request) is 1 if and only if there is at least one request among the eight *I* inputs. If EI (Enable Input) is 1, the encoder identifies the request having the highest priority and outputs its 3-bit address on $A(0\ldots2)$. If no request is active, it outputs a zero address. If the encoder is not enabled (EI = 0), it outputs zeros on *A*. EO (Enable Output) is 1 if and only if the encoder is enabled (EI = 1) *and* there is no request among the eight *I* inputs.

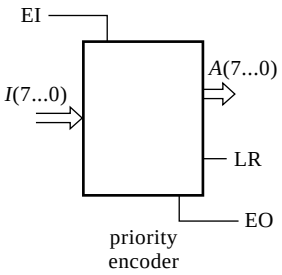


Figure P30

- **a.** Derive expressions for each output and simplify.
- **b.** Design a 48-input priority encoder using 8-bit priority encoders of the type described in this problem and minimal additional logic. Use a ripple configuration.
- **c.** Considering the enable signals, EI and EO, as the equivalent of carry signals, derive expressions for the generate and propagate variables for the eight-input priority encoder. As in Problem 29, give two expressions for the propagate variable and pick the "better" one. Does it require extra logic to compute the generate and propogate variables, or are they available from the outputs of the eight-input priority encoder described here?
- **d.** Using suitable 4-bit lookahead units, design a lookahead implementation for a 48-input priority encoder and compare its speed with the design in part *b*.
- **e.** Suppose that the eight-input priority encoder has *disable* signals, DI and DO, instead of enable signals EI and EO. Repeat parts *c* and *d* considering the disable signals as the equivalent of carry signals.

31 A BCD-to-seven-segment decoder has "blank" signals, BI and BO, to help suppress leading 0's in integer displays and trailing 0's in fraction displays. When BI is 1, if the input digit is 0, all outputs should be 0; that is, the digit will be blanked. When BI is 0, there is no blanking, but then BO is a blank signal to the next digit. A diagram is shown in Figure P31*a*.

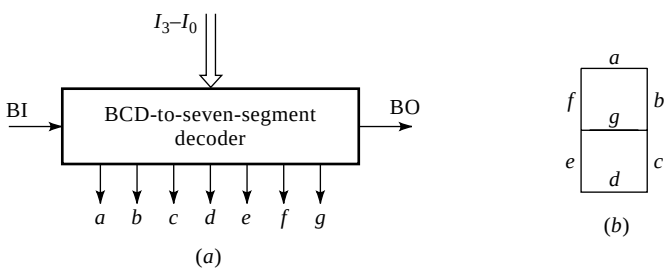


Figure P31

a. Give expressions for the outputs BO, *a*, and *f*.
b. Design an 8-bit display with four digits each for the integer and fractional parts. The least significant integer digit should never be blanked, even if the integer part of the number is 0.
c. Considering sluggish human response times, the ripple implementation in part *b* should be adequate. However for pedagogical purposes, suppose you wanted to design a lookahead implementation of the display, so that each digit would settle into the blanked or unblanked state faster. Treating BI and BO as carry signals, give expressions for the generate and propagate variables for this decoder.
d. Suppose that, instead of the BI and BO pins, the decoder has DBI ("don't blank input") and DBO ("don't blank output") pins. Treat these as the carry signals this time, and repeat part *c*.

32 Prove formally that if the propagate variable P_i for a lookahead adder is defined as the Boolean sum of A_i and B_i instead of their Exclusive-OR, the sum and carry outputs of the adder will still be computed correctly. Give an informal proof also. Which definition is better for implementation purposes?

33 A 4-bit data selector has four data inputs, $D_3 \ldots D_0$, and two select inputs, $s_1 s_0$. The output z is one of the data inputs as selected by the select inputs. Thus, $z = D_2$ when $s_1 s_0 = 10$.

a. Draw an AND-OR diagram of the data selector.
b. Another circuit consists of two XOR gates. The inputs to XOR1 are two signals *A* and *B*. The inputs to XOR2 are the output of XOR1 and a third signal *C*. Draw this circuit and write its output in terms of *A*, *B*, and *C*.
c. Choose the select inputs and the data inputs in part *a* in terms of *A*, *B*, and *C* so that the circuits in parts *a* and *b* will have the same outputs. If there is more than one choice, show all of them.

34 **a.** Design a BCD adder using a ROM (and any other logic needed), assuming only legal BCD words are used as input. Specify the dimensions of the ROM and show a schematic diagram.
b. Describe the programming table and illustrate it (at least partially).
c. Specify the number of links.

Short ____
Even ____