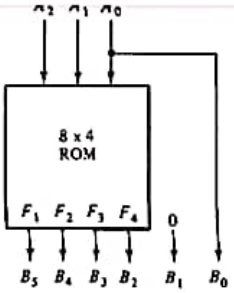


(a) Block diagram

A_2	A_1	A_0	F_1	F_2	F_3	F_4
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

(b) ROM truth table

Figure 5-24 ROM implementation of Example 5-5

EXAMPLE 5-5: Design a combinational circuit using a ROM. The circuit accepts a 3-bit number and generates an output binary number equal to the square of the input number.

The first step is to derive the truth table for the combinational circuit. In most cases this is all that is needed. In some cases we can fit a smaller truth table for the ROM by using certain properties in the truth table of the combinational circuit. Table 5-5 is the truth table for the combinational circuit. Three inputs and six outputs are needed to accommodate all possible numbers. We note that output B_0 is always equal to input A_0; so there is no need to generate B_0 with a ROM since it is equal to an input variable. Moreover, output B_1 is always 0, so this output is always known. We actually need to generate only four outputs with the ROM; the other two are easily obtained. The minimum size ROM needed must have three inputs and four outputs. Three inputs specify eight words, so the ROM size must be 8 × 4. The ROM implementation is shown in Fig. 5-24. The three inputs specify eight words of four bits each. The other two outputs of the combinational circuit are equal to 0 and A_0. The truth table in Fig. 5-24 specifies all the information needed for programming the ROM, and the block diagram shows the required connections.

202

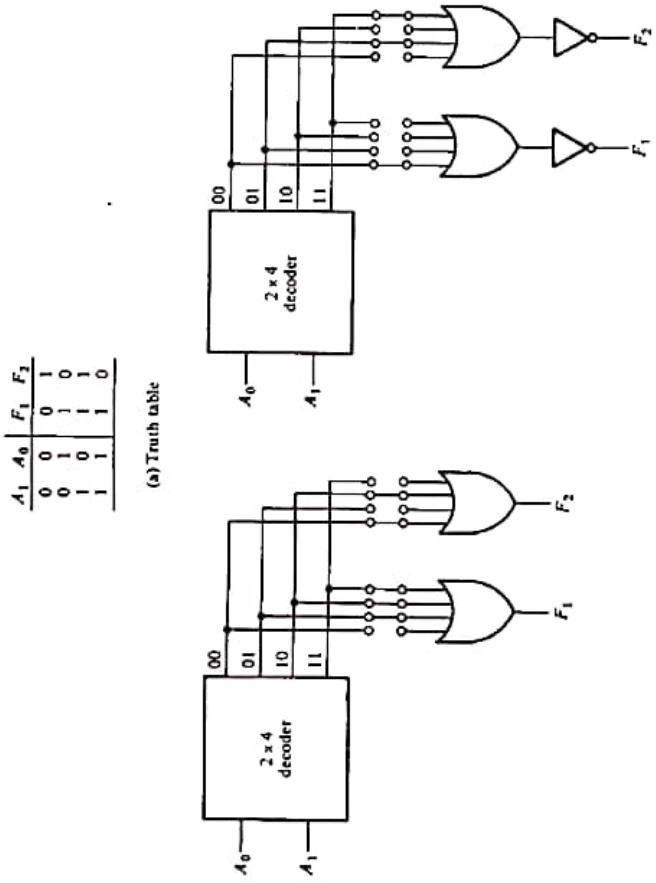
Types of ROMs

The required paths in a ROM may be programmed in two different ways. The first is called *mask programming* and is done by the manufacturer during the last fabrication process of the unit. The procedure for fabricating a ROM requires that

the customer fill out the truth table he wishes the ROM to satisfy. The truth table may be submitted on a special form provided by the manufacturer. More often, it is submitted on paper tape or punch cards in the format specified on the data sheet of the particular ROM. The manufacturer makes the corresponding mask for the paths to produce the 1's and 0's according to the customer's truth table. This procedure is costly because the vendor charges the customer a special fee for custom masking a ROM. For this reason, mask programming is economical only if large quantities of the same ROM configuration are to be manufactured.

For small quantities, it is more economical to use a second type of ROM called a *programmable read-only memory* or PROM. When ordered, PROM units contain all 0's (or all 1's) in every bit of the stored words. The links in the PROM are broken by application of current pulses through the output terminals. A broken link defines one binary state and an unbroken link represents the other state. This allows the user to program the unit in his own laboratory to achieve the desired relationship between input addresses and stored words. Special units called *PROM programmers* are available commercially to facilitate this procedure. I... case, all procedures for programming ROMs are *hardware* procedures even ... the word *programming* is used.

The hardware procedure for programming ROMs or PROMs is irre... and, once programmed the fixed pattern is permanent and cannot be altered. ... a bit pattern has been established, the unit must be discarded if the bit pattern is to be changed. A third type of unit available is called *erasable PROM* or EPROM.



A_1	A_0	F_1	F_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	0

(a) Truth table

(b) ROM with AND-OR gates

(c) ROM with AND-OR-INVERT gates

Figure 5-23 Combinational-circuit implementation with a 4 × 2 ROM

The ROM that implements the combinational circuit must have two inputs and two outputs; so its size must be 4 × 2. Figure 5-23(b) shows the internal construction of such a ROM. It is now necessary to determine which of the eight available links must be broken and which should be left in place. This can be easily done from the output functions listed in the truth table. Those minterms that specify an output of 0 should not have a path to the output through the OR
Thus, for this particular case the truth table shows three 0's, and their cor
ing links to the OR gates must be removed. It is obvious that we must ass
that an open input to an OR gate behaves as a 0 input.

Some ROM units come with an inverter after each of the OR gates
consequence, they are specified as having initially all 0's at their outputs.
programming procedure in such ROMs requires that we open the link paths of the

circuit shown in Fig. 5-10. Thus the don't-care terms cause a reduction in the number of inputs in most of the AND gates.

A careful designer should investigate the effect of the above minimization. Although it is true that under normal operating conditions the invalid six combinations will never occur, what if there is a malfunction and they do occur? An analysis of the circuit of Fig. 5-10 shows that the six invalid input combinations will produce outputs as listed in Table 5-3. The reader can look at the table and decide whether this is a good or bad design.

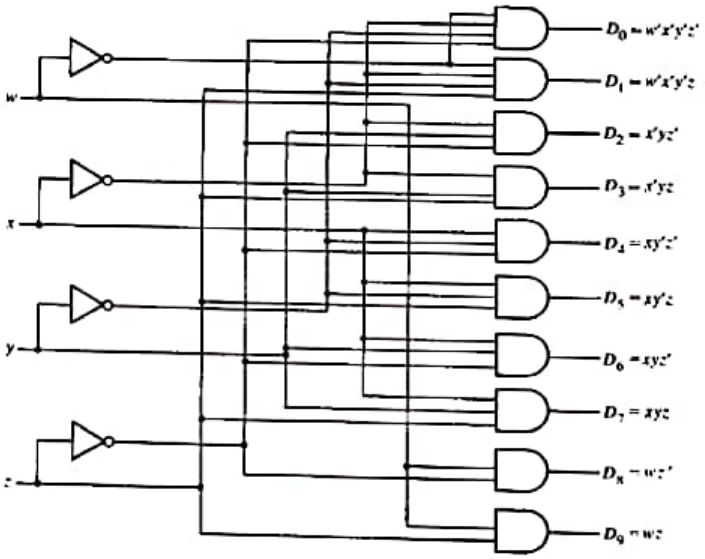


Figure 5-10 BCD-to-decimal decoder

TABLE 5-3 Partial truth table for the circuit of Fig. 5-10

Inputs				Outputs									
w	x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9
1	0	1	0	0	0	1	0	0	0	0	0	1	0
1	0	1	1	0	0	0	1	0	0	0	0	0	1
1	1	0	0	0	0	0	0	1	0	0	0	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1
1	1	1	0	0	0	0	0	0	0	1	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1	0	1

Another reasonable design decision would be to make all outputs equal to 0 when an invalid input combination occurs.* This would require ten 4-input AND gates. Other possibilities may be considered. In any case, one should not treat don't-care conditions indiscriminately but should try to investigate their effect once the circuit is in operation.

*IC type 7442 is a BCD-to-decimal decoder. The selected outputs are in the 0 state, and all the invalid combinations give an output of all 1's.

MSI function. Second, this is a good example for demonstrating the practical consquences of don't-care conditions.

*IC type 74138 is a 3-to-8 line decoder. It is constructed with NAND gates. The outputs are the complements of the values shown in Table 5-2.

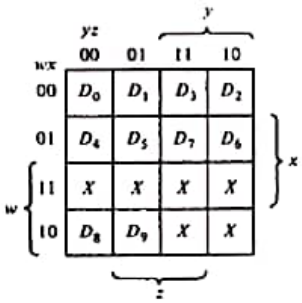


Figure 5-9 Map for simplifying a BCD-to-decimal decoder

Since the circuit has ten outputs, it would be necessary to draw ten maps to simplify each one of the output functions. There are six don't-care conditions here, and they must be taken into consideration when we simplify each of the output functions. Instead of drawing ten maps, we will draw only one map and write each of the output variables, D_0 to D_9, inside its corresponding minterm square as shown in Fig. 5-9. Six input combinations will never occur, so we mark their corresponding minterm squares with X's.

It is the designer's responsibility to decide on how to treat the don't-care conditions. Assume that it is decided to use them in such a way as to simplify the functions to the minimum number of literals. D_0 and D_1 cannot be combined with any don't-care minterms. D_2 can be combined with the don't care minterm m_{10} to give:

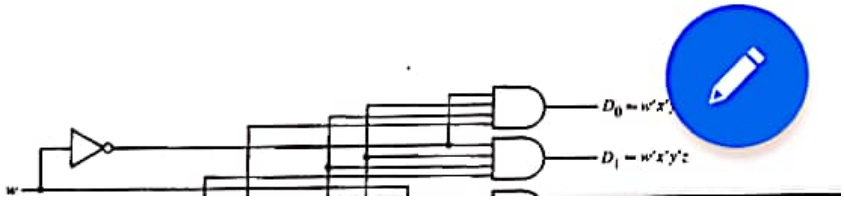
$$D_2 = x'yz'$$

The square with D_9 can be combined with three other don't-care squares to give:

$$D_9 \equiv wz$$

Using the don't-care terms for the other outputs, we obtain the circuit shown in Fig. 5-10. Thus the don't-care terms cause a reduction in the number of inputs in most of the AND gates.

A careful designer should investigate the effect of the above minimization. Although it is true that under normal operating conditions the invalid six combinations will never occur, what if there is a malfunction and they do occur? An analysis of the circuit of Fig. 5-10 shows that the six invalid input combinations will produce outputs as listed in Table 5-3. The reader can look at the table and decide whether this is a good or bad design.



name *decoder* is also used in conjunction with some code converters such as a BCD-to-seven-segment decoder (see Problem 4-14).

As an example, consider the 3-to-8 line decoder circuit of Fig. 5-8. The three inputs are decoded into eight outputs, each output representing one of the minterms of the 3-input variables. The three inverters provide the complement of the inputs, and each one of the eight AND gates generates one of the minterms. A particular application of this decoder would be a binary-to-octal conversion. The input variables may represent a binary number, and the outputs will then represent the eight digits in the octal number system. However, a 3-to-8 line decoder can be used for decoding any 3-bit code to provide eight outputs, one for each element of the code.

The operation of the decoder may be further clarified from its input-output relationships, listed in Table 5-2. Observe that the output variables are mutually exclusive because only one output can be equal to 1 at any one time. The output line whose value is equal to 1 represents the minterm equivalent of the binary number presently available in the input lines.*

182

TABLE 5-2 Truth table of a 3-to-8 line decoder

Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

EXAMPLE 5-2: Design a BCD-to-decimal decoder.

The elements of information in this case are the ten decimal digits represented by the BCD code. The code itself has four bits. Therefore, the decoder should have four inputs to accept the coded digit and ten outputs, one for each decimal digit. This will give a 4-line to 10-line BCD-to-decimal decoder.

There is really no need to design such a decoder because it can be found in IC form as an MSI function. We will design it anyway for two reasons. First, it gives insight on what to expect in such an MSI function. Second, this is a good example for demonstrating the practical consquences of don't-care conditions.

*IC type 74138 is a 3-to-8 line decoder. It is constructed with NAND gates. The outputs are the complements of the values shown in Table 5-2.

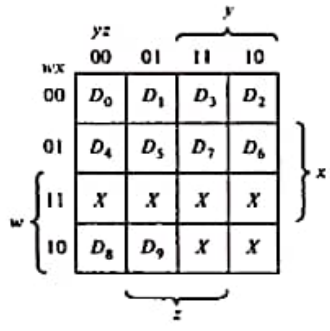


Figure 5-9 Map for simplifying a BCD-to-decimal decoder

outputs are generated with equivalence (exclusive-NOR) circuits and applied to an AND gate to give the output binary variable ($A = B$). The other two outputs use the x variables to generate the Boolean functions listed above. This is a multilevel implementation and, as clearly seen, it has a regular pattern. The procedure for obtaining magnitude comparator circuits for binary numbers with more than four bits should be obvious from this example. The same circuit can be used to compare the relative magnitudes of two BCD digits.

5-5 DECODERS

Discrete quantities of information are represented in digital systems with binary codes. A binary code of n bits is capable of representing up to 2^n distinct elements of the coded information. A *decoder* is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. If the n-bit decoded information has unused or don't-care combinations, the decoder output will have less than 2^n outputs.

The decoders presented here are called n-to-m line decoders where $m \leq 2^n$. Their purpose is to generate the 2^n (or less) minterms of n input variables. The

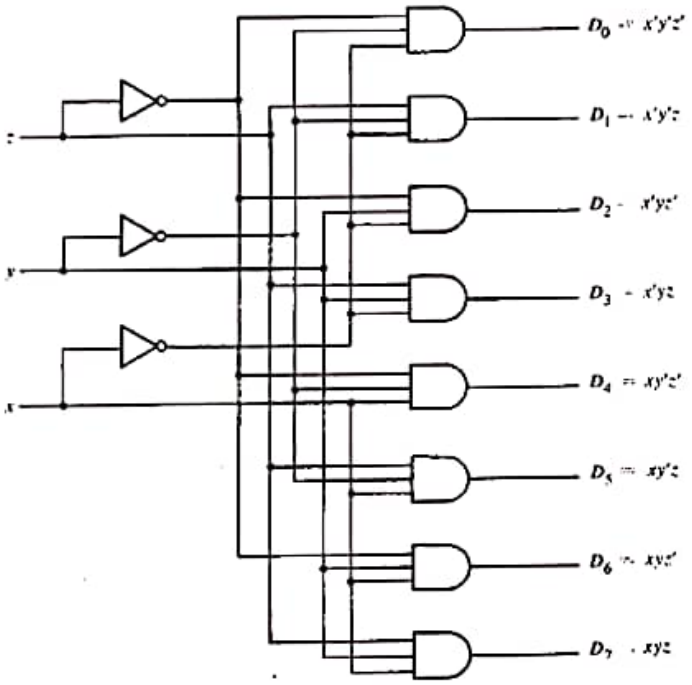


Figure 5-8 A 3-to-8 line decoder

name *decoder* is also used in conjunction with some code converters such as a BCD-to-seven-segment decoder (see Problem 4-14).

As an example, consider the 3-to-8 line decoder circuit of Fig. 5-8. The three inputs are decoded into eight outputs, each output representing one of the minterms of the 3-input variables. The three inverters provide the complement of the inputs, and each one of the eight AND gates generates one of the minterms. A particular application of this decoder would be a binary-to-octal conversion. The input variables may represent a binary number, and the outputs will then represent the eight digits in the octal number system. However, a 3-to-8 line decoder can be used for decoding any 3-bit code to provide eight outputs, one for each element of the code.

The operation of the decoder may be further clarified from its input-output relationships, listed in Table 5-2. Observe that the output variables are mutually exclusive because only one output can be equal to 1 at any one time. The output

(see Problem 5-4).

*A typical look-ahead carry generator is the IC type 74182. It is implemented with AND-OR-INVERT gates. It also has two outputs, G and P, to generate $C_3 = G + PC_1$.

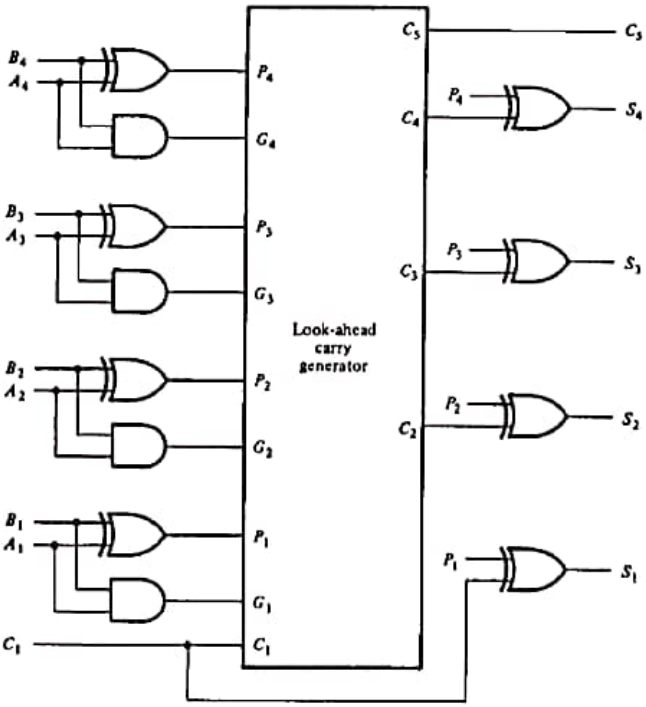


Figure 5-5 4-bit full-adders with look-ahead carry

5-3 DECIMAL ADDER

Computers or calculators that perform arithmetic operations directly in the decimal number system represent decimal numbers in binary-coded form. An adder for such a computer must employ arithmetic circuits that accept coded decimal numbers and present results in the accepted code. For binary addition, it was sufficient to consider a pair of significant bits at a time, together with a previous carry. A decimal adder requires a minimum of nine inputs and five outputs, since four bits are required to code each decimal digit and the circuit must have an input carry and output carry. Of course, there is a wide variety of possible decimal adder circuits, dependent upon the code used to represent the decimal digits.

The design of a nine-input, five-output combinational circuit by the c
method requires a truth table with $2^9 = 512$ entries. Many of the input c
tions are don't-care conditions, since each binary code input has six comb
that are invalid. The simplified Boolean functions for the circuit may be ob
by a computer-generated tabular method, and the result would probably be a
connection of gates forming an irregular pattern. An alternate procedure is to add

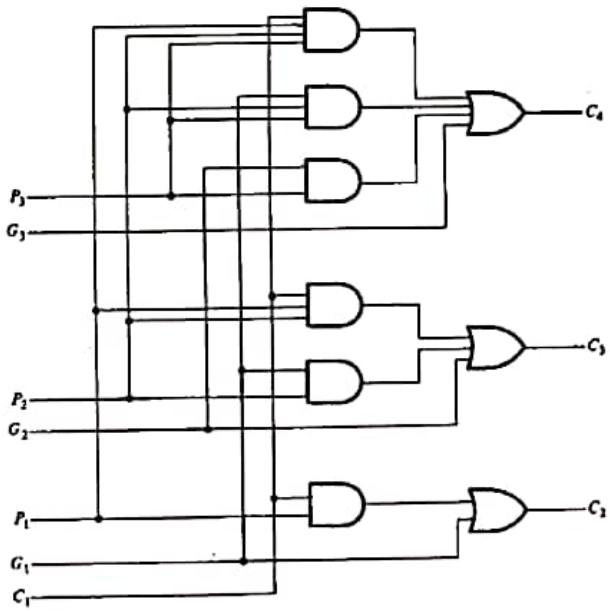
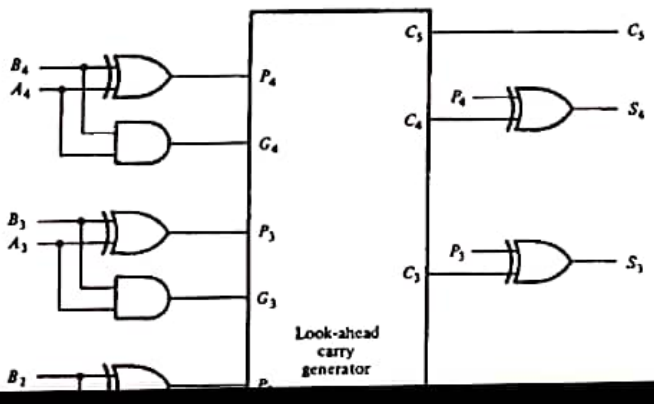


Figure 5-4 Logic diagram of a look-ahead carry generator

does not have to wait for C_3 and C_2 to propagate; in fact, C_4 is propagated at the same time as C_2 and C_3.*

The construction of a 4-bit parallel adder with a look-ahead carry scheme is shown in Fig. 5-5. Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generate the P_i variable, and the AND gate generates the G_i variable. All the P's and G's are generated in two gate levels. The carries are propagated through the look-ahead carry generator (similar to that in Fig. 5-4) and applied as inputs to the second exclusive-OR gate. After the P and G signals settle into their steady-state values, all output carries are generated after a delay of two levels of gates. Thus, outputs S_2 through S_4 have equal propagation delay times. The two-level circuit for the output carry C_5 is not shown in Fig. 5-4. This circuit can be easily derived by the equation-substitution method as done above (see Problem 5-4).

*A typical look-ahead carry generator is the IC type 74182. It is implemented with AND-OR-INVERT gates. It also has two outputs, G and P, to generate $C_3 = G + PC_1$.



If there are four full-adders in the parallel adder, the output carry C_5 would have $2 \times 4 = 8$ gate levels from C_1 to C_5. The total propagation time in the adder would be the propagation time in one half-adder plus eight gate levels. For an n-bit parallel adder, there are $2n$ gate levels for the carry to propagate through.

The carry propagation time is a limiting factor on the speed with which two numbers are added in parallel. Although a parallel adder, or any combinational circuit, will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs. Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is very critical. An obvious solution for reducing the carry propagation delay time is to employ faster gates with reduced delays. But physical circuits have a limit to their capability. Another solution is to increase the equipment complexity in such a way that the carry delay time is reduced. There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique employs the principle of *look-ahead* carry and is described below.

173

Consider the circuit of the full-adder shown in Fig. 5-3. If we define two new binary variables:

$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$

the output sum and carry can be expressed as:

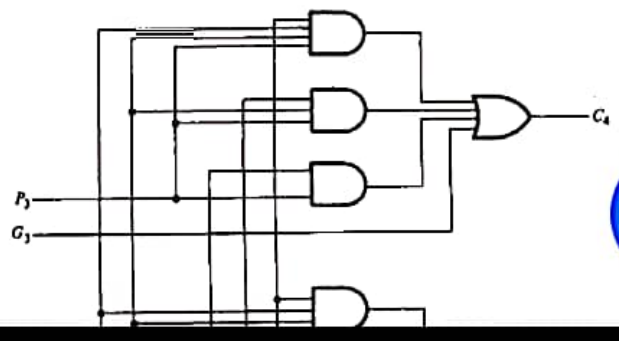
$$S_i = P_i \oplus C_i$$
$$C_{i+1} = G_i + P_i C_i$$

G_i is called a *carry generate* and it produces an output carry when both A_i and B_i are one, regardless of the input carry. P_i is called a *carry propagate* because it is the term associated with the propagation of the carry from C_i to C_{i+1}.

We now write the Boolean function for the carry output of each stage and substitute for each C_i its value from the previous equations:

$$C_2 = G_1 + P_1 C_1$$
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1 C_1) = G_2 + P_2 G_1 + P_2 P_1 C_1$$
$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_1$$

Since the Boolean function for each output carry is expressed in sum of products, each function can be implemented with one level of AND gates followed by an OR gate (or by a two-level NAND). The three Boolean functions for C_2, C_3, and C_4 are implemented in the look-ahead carry generator shown in Fig. 5-4. Note that C_4



signals whose values depend on the IC logic family used. For TTL circuits, logic-1 is equivalent to 3.5 volts and logic-0 is equivalent to ground. The S outputs from the circuit give the excess-3 equivalent code of the input BCD digit. This implementation requires one IC package and five wire connections, not including input and output wiring.

Carry Propagation

The addition of two binary numbers in parallel implies that all the bits of the augend and the addend are available for computation at the same time. As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals. The total propagation time is equal to the propagation delay of a typical gate times the number of gate levels in the circuit. The longest propagation delay time in a parallel adder is the time it takes the carry to propagate through the full-adders. Since each bit of the sum output depends on the value of the input carry, the value of S_i in any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated. Consider output S_4 in Fig. 5-1. Inputs A_4 and B_4 reach a steady value as soon as input signals are applied to the adder. But input carry C does not settle to its final steady-state value until C_3 is available in its steady-state value. Similarly, C_3 has to wait for C_2, and so on down to C_1. Thus only after the carry propagates through all stages will the last output S_4 and carry C_5 settle to their final steady-state value.

The number of gate levels for the carry propagation can be found from the circuit of the full-adder. This circuit was derived in Fig. 4-5 and is redrawn in Fig. 5-3 for convenience. The input and output variables use the subscript i to denote a typical stage in the parallel adder. The signals at P_i and G_i settle to their steady-state value after the propagation through their respective gates. These two signals are common to all full-adders and depend only on the input augend and addend bits. The signal from the input carry, C_i, to the output carry, C_{i+1}, propagates through an AND gate and an OR gate, which constitute two gate levels.

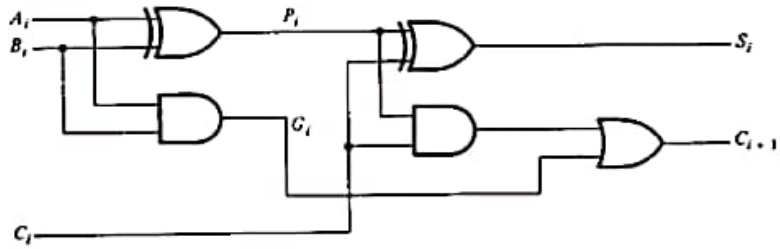


Figure 5-3 Full-adder circuit

If there are four full-adders in the parallel adder, the output carry C_5 would have $2 \times 4 = 8$ gate levels from C_1 to C_5. The total propagation time in the adder would be the propagation time in one half-adder plus eight gate levels. For an n-bit parallel adder, there are $2n$ gate levels for the carry to propagate through.

The carry propagation time is a limiting factor on the speed with which two numbers are added in parallel. Although a parallel adder, or any combinational circuit, will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs. Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is very critical. An obvious solution for reducing the carry propagation delay time is to employ faster gates with reduced delays. But physical circuits have a limit to their capability. Another solution is to increase the equipment complexity in such a way that the carry delay time is reduced. There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique employs the principle of *look-ahead* carry and is described below.

Consider the circuit of the full-adder shown in Fig. 5-3. If we define two new binary variables:

$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$