How To Design Digital Circuits

Part 2—With digital circuitry becoming an increasingly important factor in our everyday lives, it's time that we learn how to design logic circuits. Here the author discusses digital logic design—including sequential circuits and multiple output functions.

JERRY WOOLSEY

LAST MONTH WE WENT THROUGH THE BAsics of digital circuit design, using Karnaugh maps and Quine-McCluskey tables. Now, we'll look at multiple-output functions and those where the output depends on sequential input events.

Multiple-output functions

It is often the case that we wish to design a circuit with not only multiple inputs, but also multiple outputs, all of which are dependent on the same inputs. In the truth table of Fig. 17-a, we show such an example, with three inputs, a, b and c, and three outputs, f_1, f_2 and f_3. Each of these functions could be treated separately, and designed using Karnaugh maps, as shown in Figs. 17-b and 17-c. However, this type of design does not lead to optimum gate use. Some gates are repeated, and combinations of gates to perform several functions cannot be taken into account. To resolve this, we resort to a modified Quine-McCluskey method.

The workings are similar to the method described for a single-output function, but all three functions are combined into one table, and each entry is subscripted with the functions (f_1, f_2 or f_3) that it covers. Refer to Figs. 17-a and 18-a. Since an input of all zeroes produces no 1-outputs, we have no 0-bit group in the input column of the Q-M table. An input of 1 (abc = 001) causes a 1-output for functions f_1 and f_2, so we enter a 1 subscripted with these functions in the 1-bit group. We continue in this manner, filling the input column as we did for a single-output function, subscripting each with the functions that produce a 1-output for the given input.

INPUTS a	b	c	OUTPUTS f_1	f_2	f_3
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	1	0	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	0	1	1
1	1	0	0	0	0
1	1	1	0	0	0

a

f_1:

a \ bc	00	01	11	10
0	0	1	1	1
1	1	0	0	0

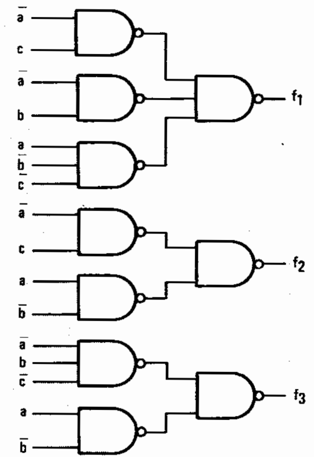
f_2:

a \ bc	00	01	11	10
0	0	1	1	0
1	1	1	0	0

f_3:

a \ bc	00	01	11	10
0	0	0	0	1
1	1	1	0	0

b



c

FIG. 17—MULTIPLE INPUTS AND OUTPUTS can also be handled. Truth table with three inputs and three outputs is shown in *a*. Resulting Karnaugh maps are shown in *b*; logic circuit is shown in *c*.

1 – f_1f_2
2 – f_1f_3
4 – $f_1f_2f_3$
---
3 – f_1f_2
5 – f_2f_3

a

1 – f_1f_2	1, 3(2) – f_1f_2
2 – f_1f_3	1, 5(4) – f_2
4 – $f_1f_2f_3$	2, 3(1) – f_1
3 – f_1f_2	4, 5(1) – f_2f_3
5 – f_2f_3	

b

FIG. 18—MODIFIED QUINE-McCLUSKEY method is used to simplify circuit shown in Fig. 17.

We now proceed to form 1-cubes as before, except now we must make sure that at least one subscript is common to each of the lower cubes being combined. (See Fig. 18-b.) Inputs 1 and 3 are adjacent, and also have the same subscripts, so we enter this in the next column as a 1-cube, also entering the subscripts. The 1 and 3 entries in the input column can be checked off, since the 1-cube just formed covers both of these inputs for all outputs. Inputs 1 and 5 are adjacent and have a common subscript, f_2, so we enter this as a 1-cube, but the subscript is only entered for f_2, since this is the only common subscript and hence the only function which contains this 1-cube.

We do not yet check the 1 or 5 in the input column, since the higher cube does not cover either input for all functions. The input 5 is checked off when we combine it with input 4, since the cube formed has the same subscripts as 5. We continue as in the case of single-output functions, until there are no more cubes that can be formed. The completed table appears in Fig. 18-b.

	F_1				F_2				F_3		
	✓ 1	✓ 2	✓ 3	✓ 4	✓ 1	✓ 3	✓ 4	✓ 5	✓ 2	✓ 4	✓ 5
* 2—$f_1 f_3$		✓							✓		
* 4—$f_1 f_2 f_3$				✓			✓			✓	
*1,3—$f_1 f_2$	✓		✓		✓	✓					
1,5—f_2					✓			✓			
2,3—f_1		✓	✓								
*4,5—$f_2 f_3$							✓	✓		✓	✓

FIG. 19—COVER MAP is generated from table shown in Fig. 18-b.

A cover map is now made as in Fig. 19, which includes the inputs that will produce a 1-output for each separate function as column headers and the unchecked entries of Fig. 18-b as row headers. Since the row labeled 2 is subscripted with f_1 and f_3, we check the columns labeled 2 under f_1 and f_3, and so on for all the rows. Following the covering procedure outlined previously, we find that the rows marked with an asterisk are essential to cover all columns. The circuit can now be drawn.

A gate is drawn for each row with an asterisk, again with the inputs to the gates corresponding to the nonchanging coordinates of the cube formed by the row header. We then draw three output gates with no input connections, and the result is as in Fig. 20.

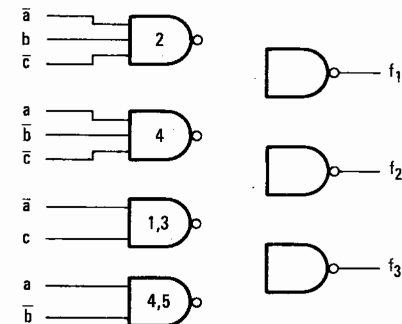


FIG. 20—PARTIAL CIRCUIT is drawn showing the outputs and inputs.

Returning to Fig. 19, we now take a minimum cover for each separate function. For f_1, we see we need the rows labeled 2, 4 and (1,3) to cover the columns under that function. The gates corresponding to these rows are thus fed to gate f_1. For f_2, we need only (1,3) and (4,5) to cover all 1-outputs, so we feed these gates to gate f_2. Note that the row labeled 4 is not needed for f_2 even though it is checked in f_2, because this is covered by the 1-cube (4,5). Similarly, f_3 requires rows 2 and (4,5). The completed circuit now appears as in Fig. 21, and is a substantial savings over the circuit shown in Fig. 17-c.

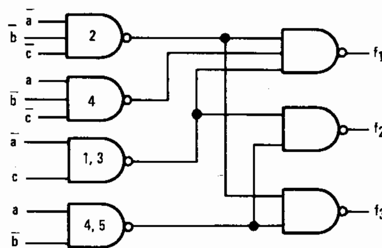


FIG. 21—SIMPLIFIED LOGIC CIRCUIT for three inputs and three outputs requires less gates than circuit shown in Fig. 17-c.

Sequential circuits

Up to this point, we have concerned ourselves only with circuits whose output depends solely on its input at a given time. However, it is often the case that a circuit must produce an output that depends not only on the present inputs to the circuit, but also on previous inputs (or outputs). To perform this, we must make use of a "memory" circuit to hold the previous information. For the experimenter, the simplest type of memory circuit is the flip-flop. When fed a clock pulse, it will store information according to its input, and hold it until the next clock pulse. This implies we must have a clock running the circuit, which we will consider later.

We can thus hold information from one clock pulse to the next. But suppose we need to know not only what happened on the previous clock pulse, but a string of several before that. We could store the entire string in a series of flip-flops, i.e., a shift register, but this could be costly for long strings and wasteful of gates, since we do not really need to look at every bit in the string as it comes in.

Instead, we can assign to each unique string of bits that may appear at the input a *state number* that corresponds to that string. We know what the string was if we know what the state number is. Thus, the input string 0000 could be assigned a state number of 0, the string 0001 a state number of 1, etc. At first glance, this does not seem to help matters much, since a 4-bit input string can have 16 possible states, which requires 4 bits for saving the state number, which is the same number required to hold the input string. But this is not necessarily so, depending on the function, and if it is so, methods have been devised for reducing the number of states. What we need to do, then, is store the state number, and update it as each bit enters.

In implementing sequential functions, we make use of two tools known as the state diagram and state table. These merely show us the possible states that our function may assume. We start first with the state diagram.

As an example, let us assume that we have a string of bits entering our circuit, and we want to know when the pattern 1101 enters. It may come at any bit time,

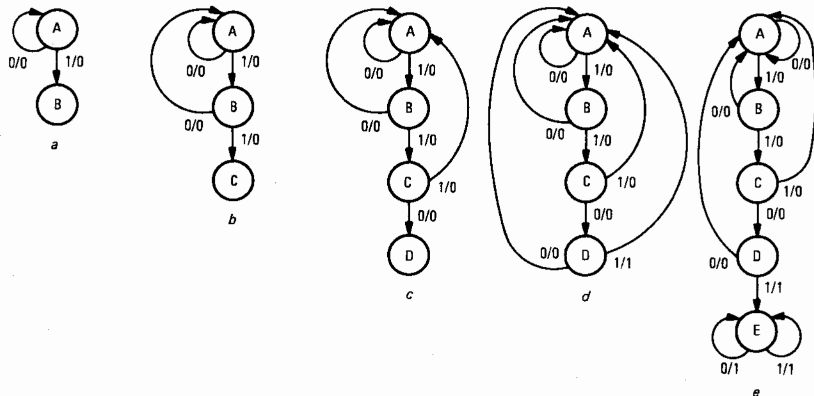


FIG. 22—STATE DIAGRAM is used in designing sequential logic circuits.

i.e., it may start at the first bit entered, or the third, etc. We start the state diagram by assuming an initial state which we call state A, and write this down in a circle. See Fig. 22-a. There are two possible occurrences at state A; we may receive either a 0 or a 1. If we receive a 0, we have not detected the start of the string 1101, so we draw an arrow from A back to itself and label it 0/0 (applied input/generated output). This means we follow this arrow if we are at state A and receive a 0-input, and the output of the circuit is to be 0. The arrow, of course, brings us back to state A to look for the first bit of the string.

This loop will continue until a 1-bit is received. At this point, we must "remember" that we have found the first bit of the string, so we draw an arrow to a new state which we name B. The arrow is labeled 1/0, and indicates that if we are at state A and a 1 is received, we are to go to state B and output a 0. Since we have covered both input conditions for state A, we move to state B. If we are at state B, we have received the first 1 of the string. If we now receive a 0, we must go back to state A, and start searching for the beginning of the string again.

If a 1 is received, we have received the first two bits of the desired string, so we

Present State (PS)	Next State (NS)		Output	
	x=0	x=1	x=0	x=1
A	A	B	0	0
B	A	C	0	0
C	D	A	0	0
D	A	A	0	1

FIG. 23—STATE TABLE listing present state, next state and output is generated from state diagram.

PS	NS		Output	
	x=0	x=1	x=0	x=1
00	00	01	0	0
01	00	10	0	0
10	11	00	0	0
11	00	00	0	1

FIG. 24—BINARY NUMBERS are assigned to present states and next states in state table.

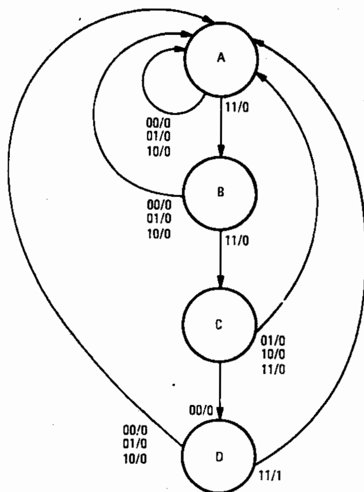


FIG. 25—STATE DIAGRAM of circuit with multiple inputs.

go to a new state, called C, which tells us that we have received a 11 so far, and a 0 is to be output. See Fig. 22-b. We follow the same procedure with state C. At state D, if we receive a 0, we have received the string of 1100 instead of 1101, so we return to state A and output a 0. If we receive a 1, however, we have received the desired 1101 string. We now have two alternatives. If we wish to continue checking for the string, we can output a 1 and return to state A, as in Fig. 22-d, or we could go to a new state, E, which simply ignores the remainder of the incoming data and outputs a constant 1 (or it could output a constant 0 or follow the incoming data). See Fig. 22-e.

Now, using Fig. 22-d, we put the diagram down in a state table, as shown in Fig. 23. The "Present State" (PS) column lists all the states that appear on the state diagram. The "Next State" (NS) column lists the next state to go to when the input is 0 (x = 0) or 1 (x = 1). For example, if we are at state A and receive an input of x = 0, the next state is A. If we receive an input of x = 1, the next state is B. The output column specifies the output to be produced when at the present state and an input of x = 0 or x = 1 is received. For example, the only time a 1 is output is when we are at state D and the input x = 1 is received.

We can now assign numbers to the state, letting A = 0, B = 1, C = 2 and D = 3, and obtain the Transition Table shown in Fig. 24. Note that with only four possible states, we need only two flip-flops to "remember" the 4-bit sequence. This table will be used later to construct the actual circuit.

Multiple input circuits can also be designed using this method. For example, Fig. 25 shows the state diagram for a circuit which is to produce a 1-output only when two input lines simultaneously input the string 1101. The NS and OUTPUT columns of the state table would then have four sub-columns, for inputs x = 00, x = 01, x = 10 and x = 11.

As another example, suppose we wished to design a circuit that would compute odd parity for a 3-bit data word, and set a flag when the parity bit was ready, after which it would compute parity on the next three bits, etc. Figure 26 shows the state diagram for the circuit. The first bit of the output is the parity bit, and the second is a flag indicating when the parity bit is ready to sample. The state table is shown in Fig. 27. Looking at the state table, we see that both states D and G advance to the same state (A) when x = 0 is input, and advance to the same state (A) when x = 1 is input. Also, the outputs of the two states are the same when x = 0 is input and when x = 1 is input.

Since the entire row D (except, of course, the PS column) is identical to G, the two states are equivalent, and we can strike out state D and replace all references to it with state G. States E and F are also equivalent, so we can eliminate state E and replace references to it with state F. Our reduced state table now appears as in Fig. 28, and we number the states to obtain the transition table shown in Fig. 29.

We are now ready to design the actual circuitry, using the table of Fig. 29. We will use D-type flip-flops as memory elements, since these have only one input, as opposed to two for the J-K flip-flop. When a clock pulse occurs on a D-type flip-flop, it merely stores the value present at its input at the time of the pulse, and makes this available at the Q-output, while the inverse is available at the $\overline{Q}$-output. Three flip-flops are needed to hold the current state numbers.

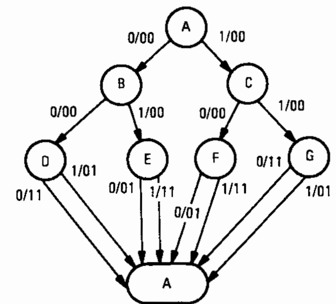


FIG. 26—STATE DIAGRAM for a circuit that derives odd parity for a 3-bit data word.

PS	NS		Output	
	x=0	x=1	x=0	x=1
A	B	C	00	00
B	D	E	00	00
C	F	G	00	00
D	A	A	11	01
E	A	A	01	11
F	A	A	01	11
G	A	A	11	01

FIG. 27—STATE TABLE derived from state diagram shown in Fig. 26.

PS	NS		Output	
	x=0	x=1	x=0	x=1
A	B	C	00	00
B	G	F	00	00
C	F	G	00	00
F	A	A	01	11
G	A	A	11	01

FIG. 28—REDUCED STATE TABLE is obtained by eliminating redundant states.

PS	NS		Output	
	x=0	x=1	x=0	x=1
000	001	010	00	00
001	100	011	00	00
010	011	100	00	00
011	000	000	01	11
100	000	000	11	01

FIG. 29—BINARY NUMBERS are assigned to the present states and next states.

Suppose we have the PS = 000 stored in the Q-outputs of flip-flop 1 (FF1), FF2 and FF3, and at the next bit time the input is x = 0. We then wish to set the flip-flops so that the Q-output of FF1 is 0, FF2 is 0, and FF3 is 1, so we know we are now at state 001. From state 001, if x = 1 is applied, we want to set FF1 to 0, FF2 to 1 and FF3 to 1 to indicate the new state, 011, etc.

We need three combinational circuits for this, one for each flip-flop, to place a 0 or a 1 at the input of each flip-flop. The input to the combinational circuits will be

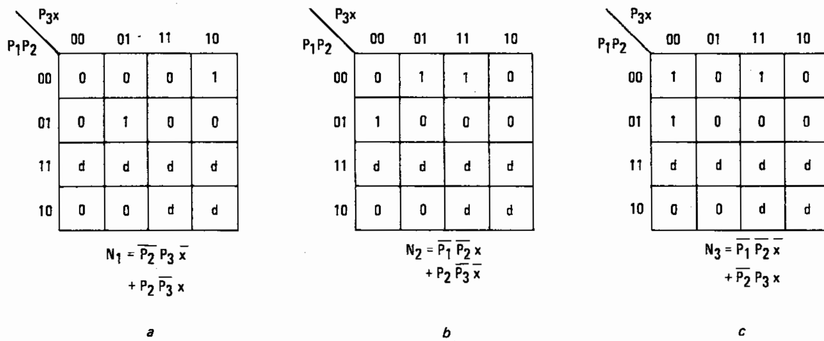


FIG. 30—A KARNAUGH MAP is drawn for each flip-flop.

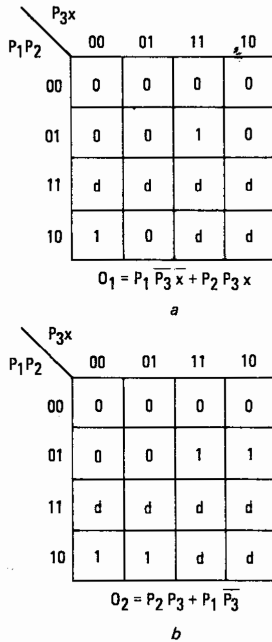


FIG. 31—OUTPUT FUNCTIONS are derived from the Karnaugh maps.

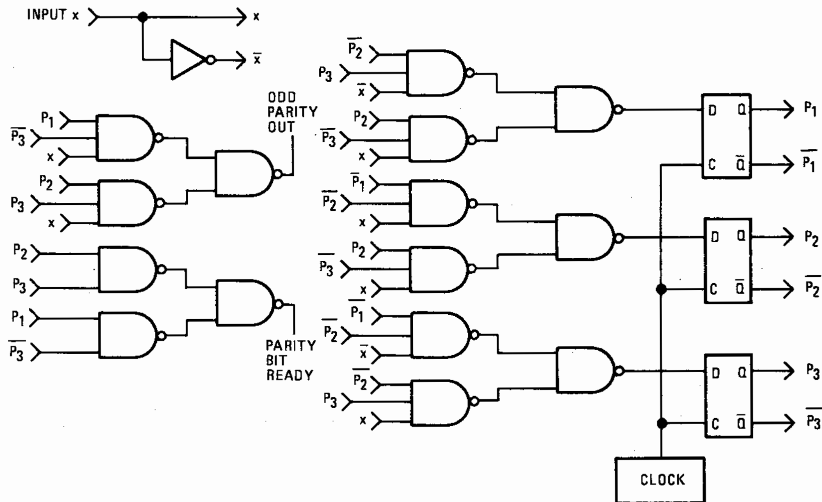


FIG. 32—COMPLETE LOGIC CIRCUIT for deriving odd parity using D-type flip-flops.

the outputs of the flip-flops, i.e., the PS, and the input x. We can label the PS-bits as p_1, p_2 and p_3, so a PS of 011 indicates $p_1 = 0$, $p_2 = 1$ and $p_3 = 1$, where p_i is the Q-output of FF_i. Now we can see that our combinational circuits have four inputs, p_1, p_2, p_3, and x, and one output, which we can label n_i to correspond to the bits of the number of the next state.

It is thus an easy matter to draw a Karnaugh map for each flip-flop input. Figure 30-a shows the map for FF1. If the PS is 000 and x = 0 is applied, then n_1, the first bit of the NS, is to be a 0, so in the box with coordinates $p_1p_2p_3x = 0000$, we place a 0. Similarly, for a PS of 000 and x = 1 ($p_1p_2p_3x = 0001$), we must have $n_1 = 0$, so a 0 is placed in box 0001. When the PS is 001 and x = 0 is applied, n_1, the first bit of the NS, is to be a 1, so a 1 is placed in box 0010. This procedure is repeated up to $p_1p_2p_3x = 1001$. Since there is no state 101, we can enter a "d" (don't-care) in boxes 1010 through 1111. Using the d-labeled boxes, we get the resultant equation for n_1, which is also shown in Fig. 30-a. The same procedure is repeated for bits n_2 and n_3 of NS, as shown in Figs. 30-b and 30-c. With these outputs applied to the inputs of the flip-flops, they will assume the correct next state after the next clock pulse.

The output functions are also designed in this way, since they depend on only the PS = $p_1p_2p_3$ and the input x. Labeling the first output bit o_1 and the second o_2, the equations are written from the Karnaugh maps as shown in Figs. 31-a and 31-b.

Each of the five functions may now be easily implemented, as shown in Fig. 32. The outputs p_i of the flip-flops are fed back to the NAND gates as shown. In actual operation, the circuit would be set to the initial state before use by toggling the CLEAR inputs on the flip-flops by a computer command or a manual switch.

This circuit could also have been realized using J-K flip-flops, using two input circuits to each flip-flop instead of one. Thus, we would need eight Karnaugh maps, one for each J-input, one for each K-input, and one for each output. These would be derived from the truth table of a J-K flip-flop, shown in Fig. 33.

As an example, referring to Fig. 29, if we wish to find the J-input of FF1 to obtain the next state (call this J_{n1}), we draw the Karnaugh map as in Fig. 34. For $p_1p_2p_3x = 0000$, we change the first state bit from 0 to 0, which requires a J-input of 0, so we enter a 0 in box 0000. With the PS = 001 and an input of x = 0, we must change the first bit of the state from a 1 to a 0, which requires a J-input of 1. If PS = 100 and x = 0 or x = 1, we n... change the first NS bit from 1 to 0, so J = d. Due to the increased number of don't-cares, the circuits feeding the J- and K-inputs are often simpler than those feeding D-inputs, though there are twice as many. (For example, the map for the K-input of FF1 from Fig. 29 will show K is merely equal to 1, or always high.)

One item essentially ignored here has been the clock pulse. In actual circuits, the clock pulse is very important.

The frequency of the clock depends on the data transmission (baud) rate of the data line, and must be synchronized so that the clock pulse occurs as close to the middle of the bit time as possible. The clock pulse must not begin until all gates have had time to settle after the new input bit has arrived and must end before the next data bit arrives.

We have now gone through the basics of logic design, and you should be able to design most common types of circuits using methods that will produce a more efficient circuit. **R-E**

To Change		Input	
From	To	J	K
0	0	0	d
0	1	1	d
1	0	d	1
1	1	d	0

FIG. 33—TRUTH TABLE for parity circuit using J-K flip-flops.

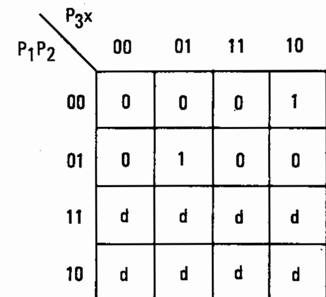


FIG. 34—KARNAUGH MAP for J-input of flip-flop FF1.