An easy way to design complex program controllers

With less than a handful of functional integrated circuits, an engineer can use a general method to readily put together a logic program controller to direct even the most involved operations

by Charles L. Richards, *Seaco Computer Displays Inc., Garland, Texas**

☐ When an electronics engineer needs to design a complicated program controller, he may well experience a sinking feeling—it could mean a return to the textbooks to relearn the techniques of transfer tables, combinational and sequential logic, and component minimization. But a new general design method relieves the engineer of these burdens and allows him to configure and prototype even an extremely complex logic controller with a minimum of effort, time, and cost.

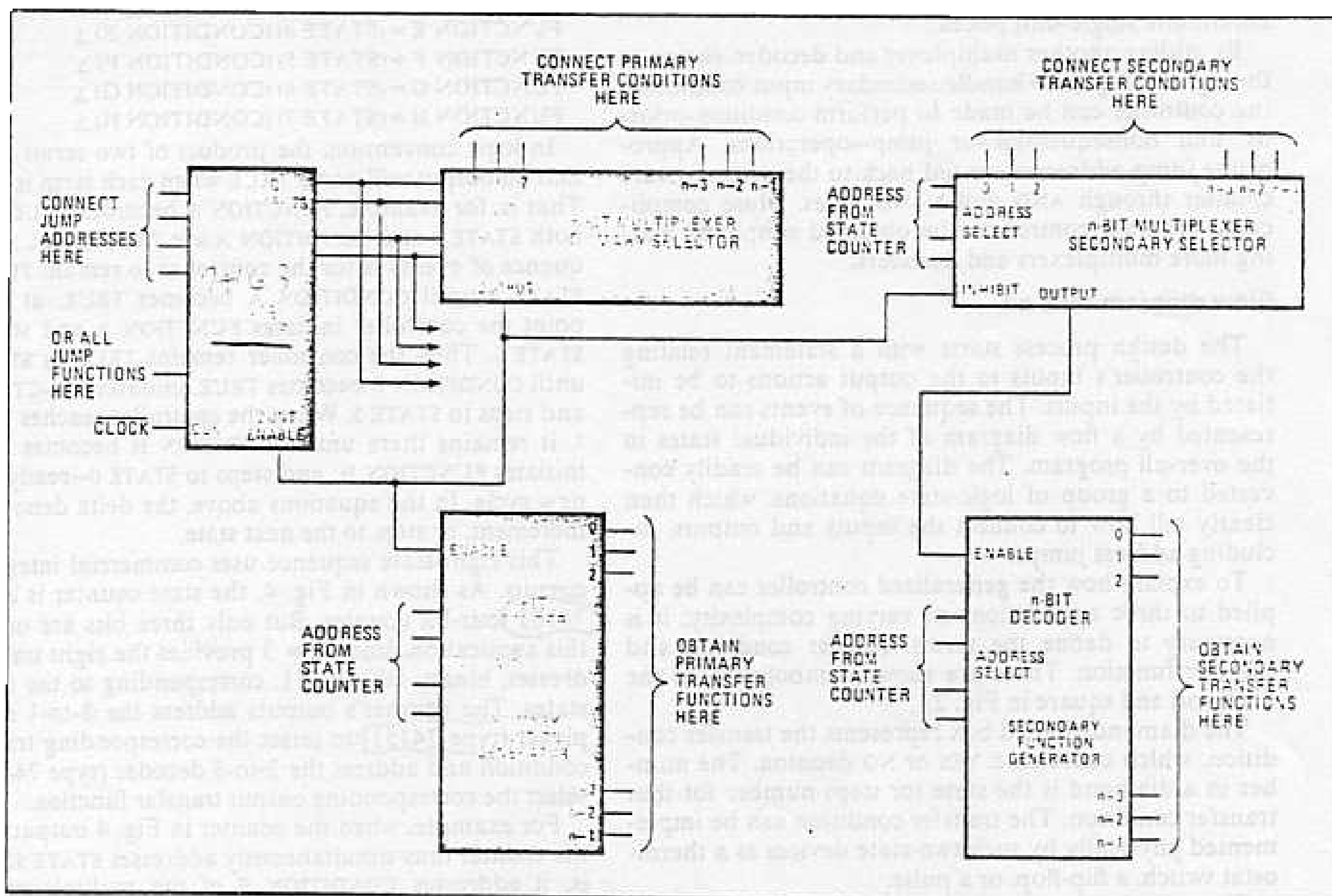
What's more, the generalized approach applies not only to straightforward sequential controllers, but also those that implement nonsequential YES-NO and multiple-choice decisions. That is, a controller can be made to index one state (or step) at a time, or to jump forward or backward to any predetermined state, or to choose which input condition out of many in the same state is to cause it to either index or jump.

In fact, the method is so easy to learn and apply that an engineer using it for the first time was able to design

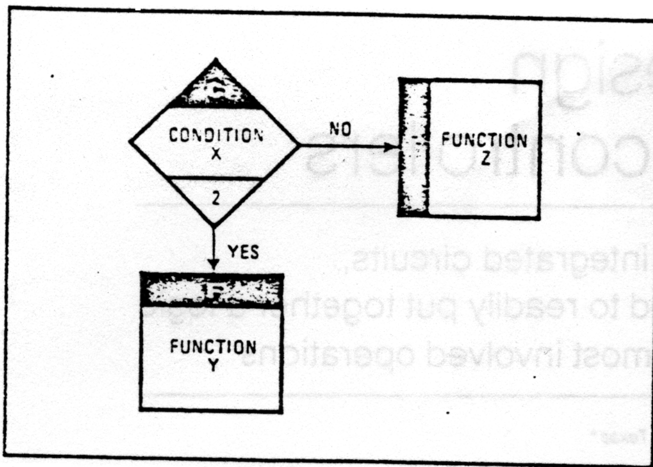
Closing the loop

Readers wanting to discuss this technique further with the author can call him on Feb. 12, 13, or 14, between 7 and 10 p.m. CST at (214) 272-9458.

1. Key logic elements. The state counter, multiplexer, and decoder, in color, are the main devices needed to produce a sequential controller that indexes from one step to the next. Adding secondary devices permits both nonsequential and priority control actions.



*Now with Texas Instruments Incorporated, Dallas, Texas



2. Transfers defined. Diamond denotes transfer condition, while rectangle denotes transfer function. One function is the action initiated by a transfer condition of YES, while a transfer condition of NO can initiate the other transfer function.

and prototype a controller involving 54 different states, with many states having five decision levels. The controller required 178 integrated circuits, had no logic errors, and worked perfectly the first time power was applied.

The three integrated circuits shown in color in Fig. 1 form the kernel of the logic-program controller. These primary devices are a k-bit state (or step) counter, an n-bit multiplexer, and an n-bit decoder. Here, n, the number of controller states, equals 2^k. For an eight-state controller, the three IC devices in plastic dual in-line packages cost about $12, even when bought at their maximum, single-unit prices.

By adding another multiplexer and decoder, shown at the right of Fig. 1, to handle secondary input conditions, the controller can be made to perform condition-priority and nonsequential—or jump—operations. Appropriate jump addresses are fed back to the primary state counter through AND and NAND gates. More complicated program control can be obtained simply by adding more multiplexers and decoders.

Flow diagram tells all

The design process starts with a statement relating the controller's inputs to the output actions to be initiated by the inputs. The sequence of events can be represented by a flow diagram of the individual states in the over-all program. The diagram can be readily converted to a group of logic-state equations, which then clearly tell how to connect the inputs and outputs, including address jumps.

To explain how the generalized controller can be applied to three applications of varying complexity, it is necessary to define the terms transfer condition and transfer function. These are shown symbolically as the diamond and square in Fig. 2.

The diamond-shaped box represents the transfer condition, which concerns a YES or NO decision. The number in a diamond is the state (or step) number for that transfer condition. The transfer condition can be implemented physically by such two-state devices as a thermostat switch, a flip-flop, or a pulse.

The transfer function, denoted by the rectangles in Fig. 2, is an action that is started or stopped by the transfer condition. As examples, the transfer function can gate a digital counter or start a motor. As shown, a YES transfer condition initiates one transfer function and a NO another transfer function.

Furthermore, depending on the controller's application, the transfer conditions can be either independent of or dependent on the transfer functions. In a dependent case, for example, the transfer condition might trigger a transfer function that starts a count of 1,000 events. The occurrence of the 1,000th count then serves as the next transfer condition. In an independent case, the next transfer condition might be an input from a timer occurring 500 milliseconds after the count starts, whether or not the count has reached 1,000.

1. Designing an eight-state sequence controller

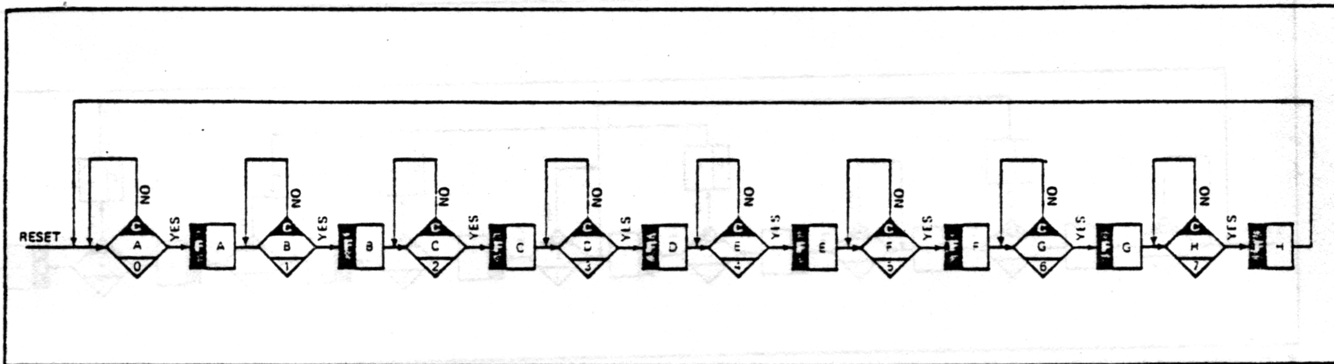
Probably the simplest program controller is one that sequences from one step to the next. Figure 3 contains the flow diagram for an eight-state sequence controller. Transfer functions are not required from any NO conditions, so NO-outputs are simply symbolically looped back as a condition input. The corresponding logic equations are:

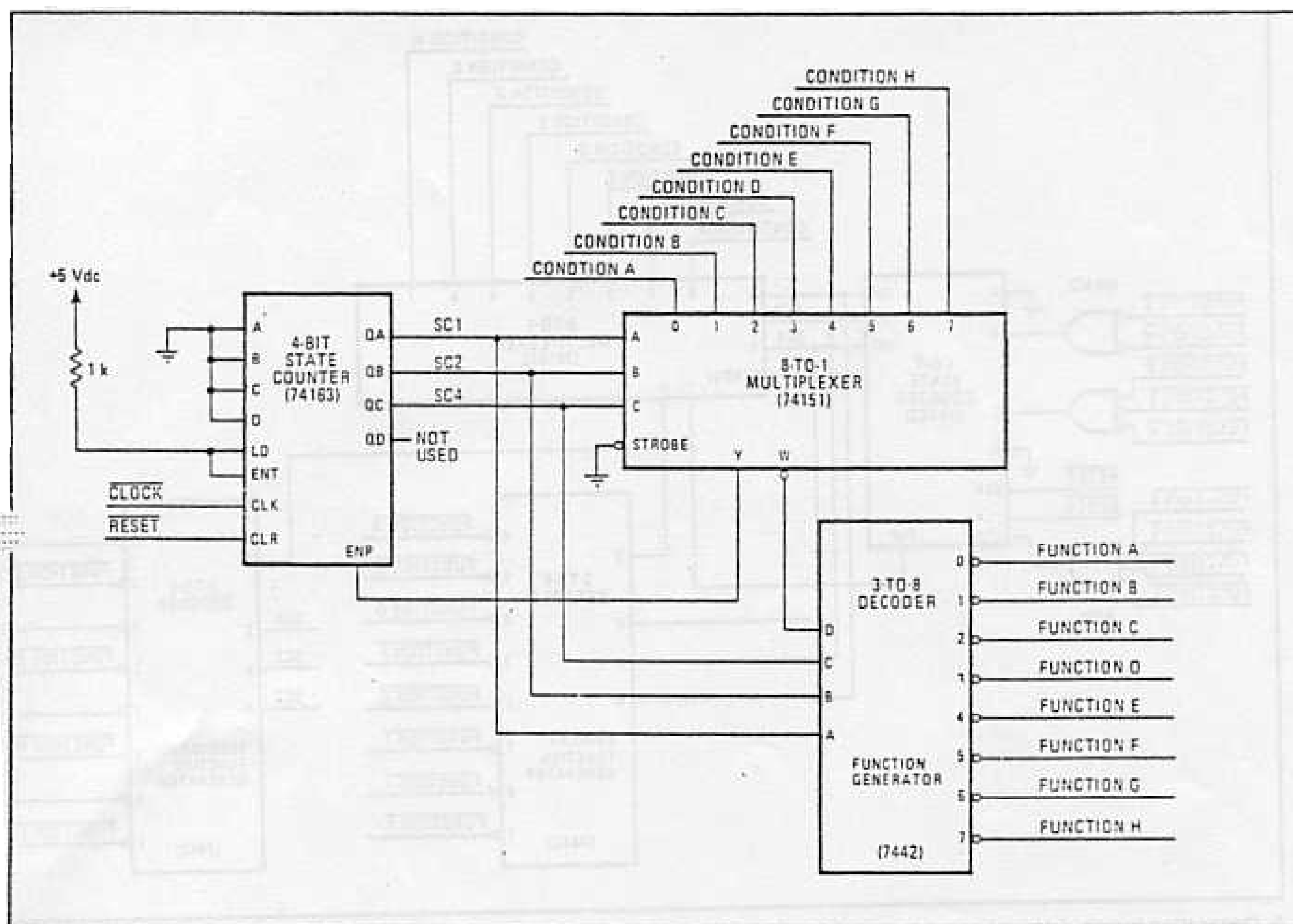
FUNCTION A = (STATE 0) (CONDITION A) Δ
FUNCTION B = (STATE 1) (CONDITION B) Δ
FUNCTION C = (STATE 2) (CONDITION C) Δ
FUNCTION D = (STATE 3) (CONDITION D) Δ
FUNCTION E = (STATE 4) (CONDITION E) Δ
FUNCTION F = (STATE 5) (CONDITION F) Δ
FUNCTION G = (STATE 6) (CONDITION G) Δ
FUNCTION H = (STATE 7) (CONDITION H) Δ

In logic convention, the product of two terms means that an output will occur TRUE when each term is TRUE. That is, for example, FUNCTION A becomes TRUE when both STATE 0 and CONDITION A are TRUE. Thus, the sequence of events is for the controller to remain TRUE in STATE 0 until CONDITION A becomes TRUE, at which point the controller initiates FUNCTION A and steps to STATE 1. Then the controller remains TRUE in STATE 1 until CONDITION B becomes TRUE, initiates FUNCTION B, and steps to STATE 3. When the controller reaches STATE 7, it remains there until CONDITION H becomes TRUE, initiates FUNCTION H, and steps to STATE 0—ready for a new cycle. In the equations above, the delta denotes on increment, or step, to the next state.

This eight-state sequence uses commercial integrated circuits. As shown in Fig. 4, the state counter is a type 74163 four-bit counter. But only three bits are used in this application, since k = 3 provides the eight state addresses, binary 000 to 111, corresponding to the 0 to 7 states. The counter's outputs address the 8-to-1 multiplexer (type 74151) to select the corresponding transfer condition and address the 3-to-8 decoder (type 7442) to select the corresponding output transfer function.

For example, when the counter in Fig. 4 outputs 101, the counter thus simultaneously addresses STATE 5. That is, it addresses CONDITION F of the multiplexer and FUNCTION F at the decoder.



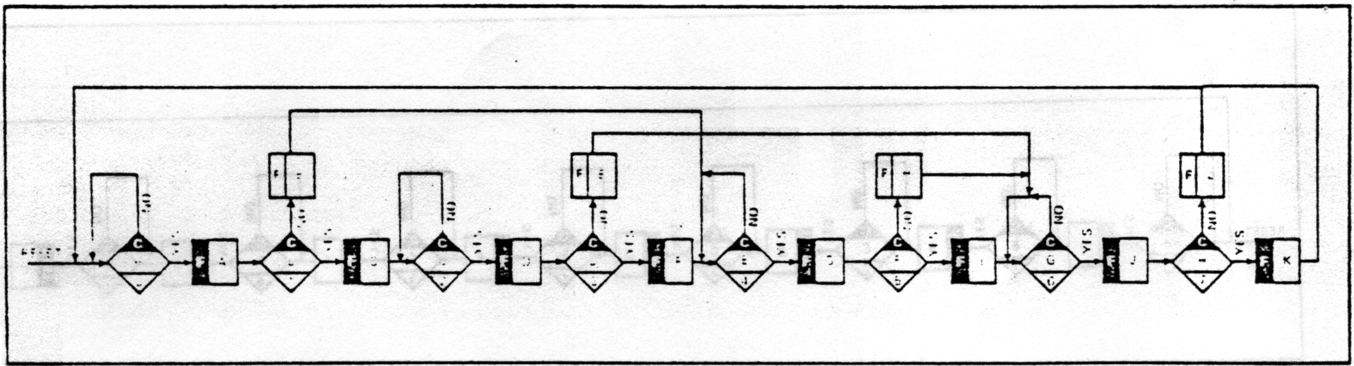


4. Sequence controller. In a step-by-step sequence controller, which can be implemented with as few as three IC packages, the multiplexer's Y-output enables the counter to increment the state address for the multiplexer and decoder to yield the required function.

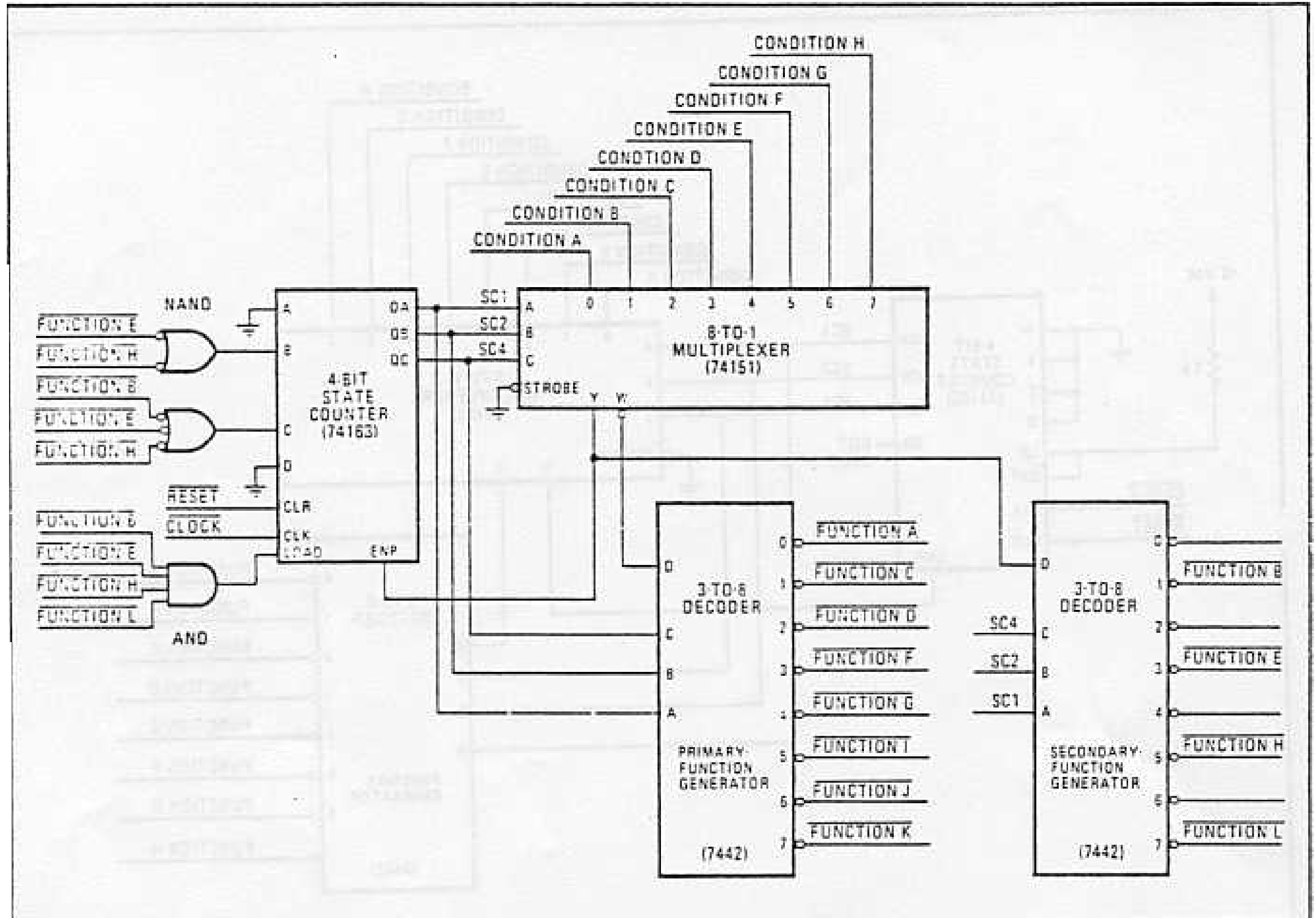
Assume the counter has been RESET to binary 000, corresponding to STATE 0 in the flow chart. This count on the multiplexer's address inputs gates the status of CONDITION A from the multiplexer's input to its complementary Y and W outputs. As long as CONDITION A is NO, the Y output is low and the W output is high. The low Y signal inhibits the counter's ENABLE-P INPUT, so the counter cannot increment even when a CLOCK pulse is present. The W output connects to the decoder's most-significant-bit output (D) which, if high, inhibits the decoder's 0 to 7 outputs. But when multiplexer output W goes low it enables the decoder output addressed by the state counter.

When CONDITION A becomes YES, two things happen: the multiplexer's Y output goes high and allows the state counter to increment on the next CLOCK pulse; and the W output goes low and enables the decoder, addressed to 000, to produce a low output on line 0, thus yielding a signal to initiate FUNCTION A. (Here, a low-voltage output is defined as a TRUE FUNCTION A.)

When the next CLOCK pulse occurs, the state counter increments to 001 (or STATE 1). FUNCTION A goes back high, and the multiplexer's 001-address then gates CONDITION B through the multiplexer, but FUNCTION B from the decoder appears only when CONDITION B becomes YES and the counter increments to the next state. In this



5. Decide and jump. Controller executes steps in sequence unless a condition is NO, in which case—as shown in color—the controller initiates a secondary function and jumps to a new state. Inputs to state counter establish address for multiplexer and decoder.



6. Generating jumps. Adding a secondary decoder (function generator) provides the outputs for the secondary conditions, shown in Fig. 5, which are also fed back to the state-counter's inputs through gates to produce the new jump address for the multiplexer and decoders.

manner, the controller steps through to STATE 7 (111), and when CONDITION H becomes YES, FUNCTION H is generated, the state counter steps to STATE 0 (000), and the controller is ready for the next cycle of operation.

Note in Fig. 4 that the address inputs for the state counter are grounded. The reason is that in this application the required state-by-state indexing is carried out by a CLOCK pulse each time a selected YES condition drives the multiplexer's Y output high to ENABLE the counter. (In more complex controllers, the counter's inputs are addressed according to program requirements, as will shortly be explained.) Simple as it is, however, the sequence controller can prove useful, for example, where eight conditions must be performed in prescribed order to insure safe and proper operation of a production machine.

2. Designing a nonsequential alternate-function controller

More complex, and certainly more realistic, is a program controller that must trigger one transfer function when a condition is YES and another function if the condition is NO. Also required is that the controller se-

quence to the next state if the condition is YES or jump to a nonsequential state if NO.

Figure 5 contains the flow diagram for a controller that can perform these YES-NO decisions and nonsequential jumps. Here, for example, when it is in STATE 1 and CONDITION B is YES, it will initiate FUNCTION C; but when CONDITION B is NO, it will initiate FUNCTION B and jump to STATE 4. The logic equations, developed from inspection of the flow diagram (Fig. 5), are:

$$\overline{\text{FUNCTION A}} = (\text{STATE 0})(\text{CONDITION A})\Delta$$
$$\overline{\text{FUNCTION B}} = (\text{STATE 1})\overline{(\text{CONDITION B})} \rightarrow 4$$
$$\overline{\text{FUNCTION C}} = (\text{STATE 1})(\text{CONDITION B})\Delta$$
$$\overline{\text{FUNCTION D}} = (\text{STATE 2})(\text{CONDITION C})\Delta$$
$$\overline{\text{FUNCTION E}} = (\text{STATE 3})\overline{(\text{CONDITION D})} \rightarrow 6$$
$$\overline{\text{FUNCTION F}} = (\text{STATE 3})(\text{CONDITION D})\Delta$$
$$\overline{\text{FUNCTION G}} = (\text{STATE 4})(\text{CONDITION E})\Delta$$
$$\overline{\text{FUNCTION H}} = (\text{STATE 5})\overline{(\text{CONDITION F})} \rightarrow 6$$
$$\overline{\text{FUNCTION I}} = (\text{STATE 5})(\text{CONDITION F})\Delta$$
$$\overline{\text{FUNCTION J}} = (\text{STATE 6})(\text{CONDITION G})\Delta$$
$$\overline{\text{FUNCTION K}} = (\text{STATE 7})(\text{CONDITION H})\Delta$$
$$\overline{\text{FUNCTION L}} = (\text{STATE 7})\overline{(\text{CONDITION H})} \rightarrow 0$$

The horizontal arrows in the equation point to the required jump state, as determined from the application flow diagram.

Here, the complement (FALSE) of a function—denoted by the ban over, for example, $\overline{\text{FUNCTION A}}$—must actually be interpreted as the initiation of the required function so as to be internally consistent with the voltage-level convention of the devices in this particular controller. In these devices, a TRUE logic level means a high voltage level; a FALSE logic level means a low voltage level. Thus, the equations above are logically consistent with their electrical circuit (Fig. 6).

This implementation is substantially similar to that of the simple sequence controller, except for the addition of the secondary decoder to develop the nonsequential addresses for those transfer functions generated by the four NO conditions. Also required are NAND gates to drive the state counter to the correct state address and an AND gate to LOAD that address into the counter. If, in Fig. 5, all conditions go YES in sequence, then the operation is the same as that for the previous sequence controller.

Suppose, though, the controller has sequenced through to STATE 3, CONDITION D, which if YES initiates $\overline{\text{FUNCTION F}}$. However, if CONDITION D is NO, the flow diagram indicates the controller should jump to STATE 6, CONDITION G. Referring to Fig. 6, all transfer conditions are inputted through the 8-to-1 multiplexer, with the particular condition gated through the multiplexer (transfer-condition selector) depending on the address produced by the state counter. Also, depending on the counter's state address, the primary decoder will produce one primary function, or the secondary decoder one secondary function. Here, secondary function B occurs at STATE 1, E at STATE 3, H at STATE 5, and L at STATE 7. Thus, the controller uses the secondary decoder's 1, 3, 5, and 7 outputs.

The primary and secondary transfer functions initiate the desired external actions mandated by the particular application. A YES primary condition will cause the controller to index to the next state. But the secondary functions are fed back as inputs to the state counter to generate a jump address and to load the state counter with that address.

Connecting jump addresses

As shown in Fig. 5 and by the logic equations, the required address jumps are:

Function B $\rightarrow$ 4; E $\rightarrow$ 6; H $\rightarrow$ 6; L $\rightarrow$ 0

These state numbers are obtained by addressing the state counter's binary-weighted inputs. The counter's highest-ordered input (D) is permanently set to low level, or binary 0, by grounding, since the A, B, and C inputs can yield the required eight state addresses.

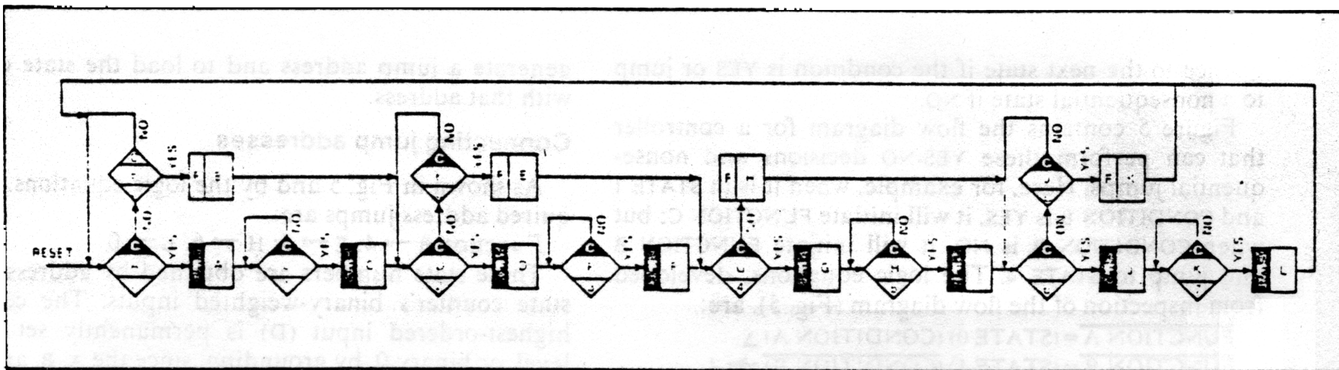
In Fig. 6, these addresses are developed through two NAND gates. $\overline{\text{FUNCTION B}}$ inputted to one NAND gate puts a high-level signal on the counter's C input and generates the 100 which is the jump-to-STATE 4 address applied to the multiplexer and decoders. And $\overline{\text{FUNCTION E}}$ is fed through both NAND gates to activate the B and C inputs to generate 110, the STATE 6 address. The 0 jump address occurs simply when there are no input signals on the NAND gates. Note that since only even-numbered jump addresses are used, the state counter's A input is permanently grounded. In applications requiring odd-numbered addresses, the A input would also be accessed through a NAND gate by the odd-numbered functions.

All secondary-decoder jump outputs serve as inputs to an AND gate that in turn connects to the state counter LOAD input. Because of the voltage-level convention, the AND gate actually performs an OR logic function. Therefore, whenever any jump function appears at the AND gate inputs, the counter's A, B, or C inputs LOAD the counter to set up the jump address at its output.

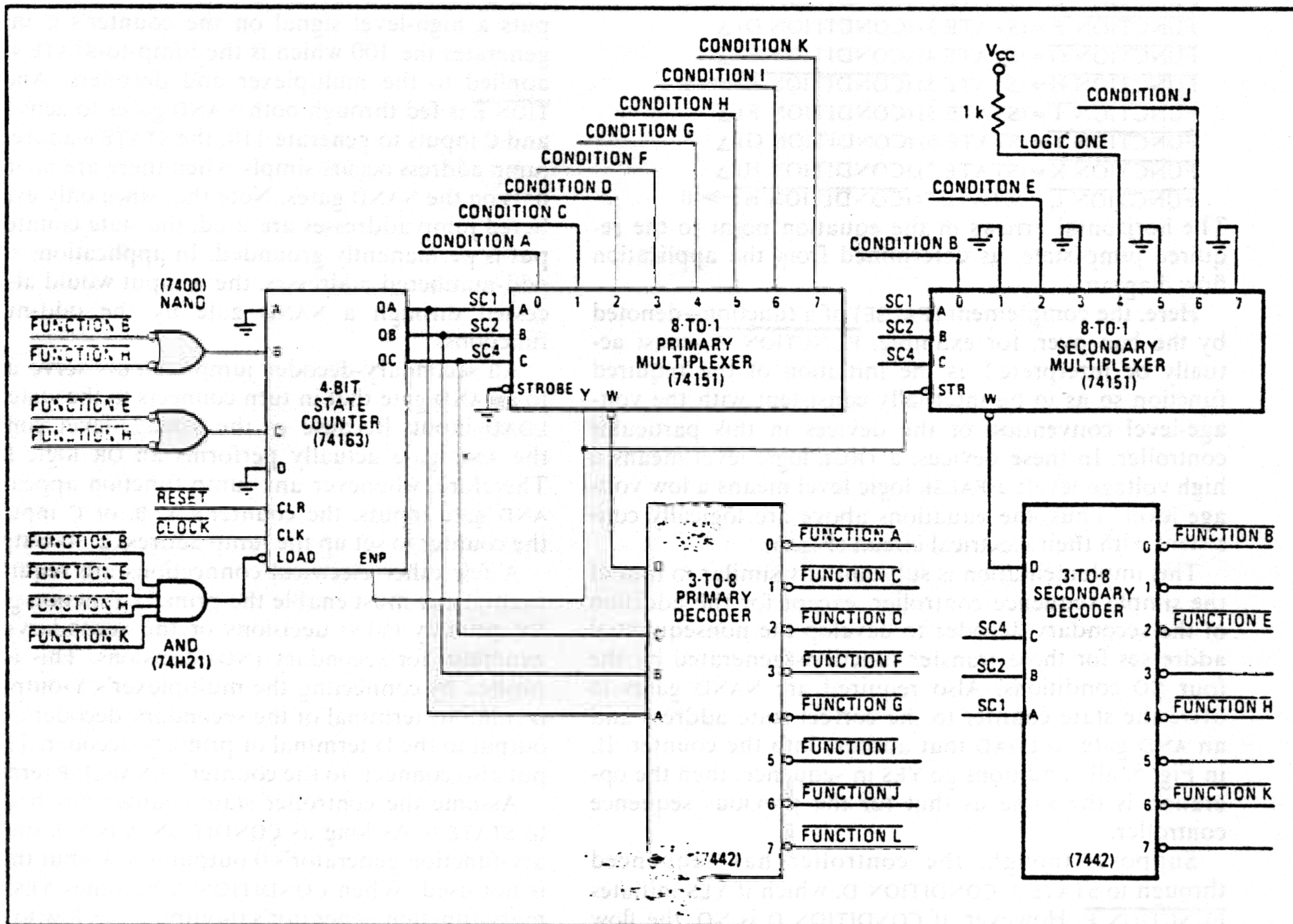
A few other electrical connections are required. The multiplexer must enable the primary function generator for primary (YES) decisions or the secondary-function generator for secondary (NO) decisions. This is accomplished by connecting the multiplexer's Y-output to the D (inhibit) terminal of the secondary decoder and the W output to the D terminal of primary decoder. The Y output also connects to the counter's ENABLE-P terminal.

Assume the controller state counter has been RESET to STATE 0. As long as CONDITION A is NO, the secondary-function generator's 0 output is low—but this output is not used. When CONDITION A becomes YES, the primary-function generator's 0-output goes low to generate $\overline{\text{FUNCTION A}}$. At the same time the multiplexer's Y output goes high to drive the state counter's ENABLE-P input and, on the next CLOCK pulse, the counter increments to STATE 1. Here, as shown in Figs. 5 and 6, if CONDITION B is YES, the primary-function generator is enabled, because W is low, to produce $\overline{\text{FUNCTION C}}$, and, because Y is high, the state counter increments to STATE 2 on the next CLOCK. However, if CONDITION B is NO the low Y signal on the secondary-function generator's D terminal enables that decoder to yield $\overline{\text{FUNCTION B}}$. And the counter must jump to STATE 4. Therefore, $\overline{\text{FUNCTION B}}$ gets fed to the counter's C input through the NAND gate, and to the LOAD input through the AND gate. Thus, the next CLOCK pulse loads the counter to a count of 100, or STATE 4.

In this manner, the controller will either index to the next state or jump to a prescribed state. As shown in



7. Priority control. Flow diagram indicates controller must give first priority, at any state, to primary conditions, at left, but if a primary condition is NO and secondary condition—in color—is YES, then controller initiates secondary function and jumps.



8. Dual decision. Adding a secondary multiplexer, upper right, provides gating of secondary, or low-priority, inputs, with the primary multiplexer's enable and inhibit outputs choosing whether to give priority to primary or secondary transfer conditions.

Fig. 5. initiation of FUNCTION L will bypass $\overline{\text{FUNCTION K}}$ and reset the controller to STATE O. but if CONDITION H is YES. the controller will first initiate FUNCTION K and then increment to STATE 0.

3. Designing a nonsequential priority-condition controller

Consider now any application in which. at one or more states. two input conditions exist and the program controller has to choose which condition will initiate the next function. Thus. the controller must follow a set of priority rules. This controller is slightly more complex. electrically. than the previous two examples. but is still easily put together with standard ICs.

In STATE 0 of Fig. 7. for instance CONDITION A could represent a thermostat switch which. if closed (YES) initiates FUNCTION A and indexes the controller to STATE 1. But if the thermostat is open (NO). then CONDITION B should be implemented. Here CONDITION B could be a timer input. In STATE 0 the controller is to give first priority to the temperature input. but if the temperature does not close the thermostat. then after some elapsed time the controller will operate through CONDITION B

and jump to STATE 2. And if the temperature and time are both YES, then the controller is to obey the move dictated by the priority assignment, CONDITION A. Figure 7 includes eight high-priority conditions—A, C, D, F, G, H, I, and K—and three low-priority conditions—B, E, and J—at STATE 0, 2, and 6 with jumps to, respectively, STATES 2, 4, and 0. Also a jump is needed to STATE 6 when CONDITION G, at STATE 4, is NO.

Inspection of the flow diagram (Fig. 7) leads to the following logic equations, which indicate the connections between the devices making up the controller (Fig. 8). Again the delta means index to next state, and the horizontal arrow means jump to the indicated state.

$\overline{\text{FUNCTION A}}$ = (STATE 0) (CONDITION A) Δ
$\overline{\text{FUNCTION B}}$ =
(STATE 0) $\overline{\text{(CONDITION A)}}$ (CONDITION B) $\rightarrow$ 2
$\overline{\text{FUNCTION C}}$ = (STATE 1) (CONDITION C) Δ
$\overline{\text{FUNCTION D}}$ = (STATE 2) (CONDITION D) Δ
$\overline{\text{FUNCTION E}}$ =
(STATE 2) $\overline{\text{(CONDITION D)}}$ (CONDITON E) $\rightarrow$ 4
$\overline{\text{FUNCTION F}}$ = (STATE 3) (CONDITION F) Δ
$\overline{\text{FUNCTION G}}$ = (STATE 4) (CONDITION G) Δ
$\overline{\text{FUNCTION H}}$ = (STATE 4) $\overline{\text{(CONDITION G)}}$ $\rightarrow$ 6
$\overline{\text{FUNCTION I}}$ = (STATE 5) (CONDITION H) Δ
$\overline{\text{FUNCTION J}}$ = (STATE 6) (CONDITION I) Δ
$\overline{\text{FUNCTION K}}$ =
(STATE 6) $\overline{\text{(CONDITION I)}}$ (CONDITION J) $\rightarrow$ 0
$\overline{\text{FUNCTION L}}$ = (STATE 7) (CONDITION K) Δ

Here again the logic equations show that the required function results when the corresponding decoder output goes low, to be consistent with device electrical levels.

Generating priorities

In Fig. 8, the high-priority conditions are the same as the primary conditions used in the previous examples, and they are gated through the multiplexer generating the high-priority transfer condition. Another multiplexer generates the low-priority transfer conditions. Again two decoders are used, one to output the high-priority functions, the other the low-priority functions. Since this application also requires nonsequential jumps, the jump addresses are obtained by the same procedure of feeding back appropriate secondary (or low-priority) output functions through NAND gates to the state counter. And the presence of any one of these jump functions and the AND gate (operating in an OR mode) loads the address into the state counter. As in the preceding example, the addresses developed by the state counter drive the multiplexers and decoders.

Of particular interest in this example is how the devices are connected so that they properly assess the required priority (if any) in a given state, and enable or inhibit the associated integrated circuits. The Y output of the primary multiplexer connects to the state counter's ENABLE-P terminal to provide sequential indexing when needed. This Y output also goes to the STROBE terminal of the low-priority multiplexer, which inhibits the low-priority transfer-function selector (multiplexer) any time the selected high-priority transfer condition is TRUE. As in the preceding example, when a multiplexer's W output, the complement of Y, is low, it inhibits the function output of the related decoder.

Consider now some of the alternative actions provided by this program controller that choose and implement a function depending on the priorities assigned to two conditions at a given state. Assume the controller has been RESET to STATE 0. Here, the high-priority (primary) multiplexer is addressed to select CONDITION A and the low-priority (secondary) multiplexer to select CONDITION B.

If CONDITION A is YES (or TRUE), three things happen: the Y output of the primary multiplexer ENABLES the state counter to index to the next step, the W output of the same multiplexer removes the inhibit on the D terminal of the primary decoder and thus generates the addressed $\overline{\text{FUNCTION A}}$: the Y output, connected to the STROBE terminal of the secondary multiplexer, inhibits—through that multiplexer's W output—the secondary transfer-function generator (decoder). As required, the controller generates $\overline{\text{FUNCTION A}}$ and steps to STATE 1.

However, suppose the controller is in STATE 0 and that CONDITION A is NO and CONDITION B is YES. As shown in Fig. 7, the controller in this situation is to initiate $\overline{\text{FUNCTION B}}$ and jump to STATE 2. Since CONDITION A is NO, the low-priority transfer-function generator is enabled, resulting in $\overline{\text{FUNCTION B}}$ appearing on output line 0, as required. Furthermore, this function is fed back to the state counter's NAND gate which enables input-terminal B to a 100 address so the controller jumps to STATE 4, as required.

For the case where CONDITION A and B are both YES, the controller is to give priority to, and react to, CONDITION A only. This action is the result of the high Y output of the primary multiplexer inhibiting the secondary multiplexer and thus preventing CONDITION B from being gated through to the decoder. Therefore, the controller ignores CONDITION B and the primary Y output enables the state counter to increment to STATE 1. Of course, if CONDITIONS A and B are both NO, the controller stays in STATE 0.

In some states, as for example STATE 3, the controller is required to step to STATE 4 only when a condition (here CONDITION F) becomes YES. Because the secondary multiplexer and decoder are inhibited, the controller indexes in the same manner as in the sequence controller in the first example.

Even without having to make a priority decision, this controller can also perform YES-NO nonsequential jumps, as is required at STATE 4. Here, the controller is to generate $\overline{\text{FUNCTION G}}$ and step to STATE 5 if CONDITION G is YES, or generate $\overline{\text{FUNCTION H}}$ if CONDITION G is NO. In the YES, or primary condition, the primary multiplexer inhibits the secondary multiplexer, so the controller simply generates $\overline{\text{FUNCTION G}}$ and goes to STATE 5. If CONDITION G is NO, the controller must initiate $\overline{\text{FUNCTION H}}$ and jump to STATE 6. To accomplish this, a YES condition is permanently connected to input 4 of the secondary controller, shown as the logic one connection in Fig. 8. Being in STATE 4, with the secondary multiplexer enabled through its STROBE connection and the secondary decoder enabled through its D input, the controller can then "gate" this permanent YES condition through the multiplexer and decoder to generate $\overline{\text{FUNCTION H}}$, as required. Furthermore, this output is fed to the B and C NAND gates to produce the 110 corresponding to the required jump address of 6. □