Table 9-4 Output-coded state assignment for the T-bird tail-lights machine.

```
module tbirdsdo
title 'Output-Coded T-Bird Tail Lights State Machine'
TBIRDSDO device 'P16V8R';

" Input and output pins
CLOCK, LEFT, RIGHT, HAZ, RESET     pin 1, 2, 3, 4, 5;
L3Z, L2Z, L1Z, R1Z, R2Z, R3Z       pin 18..13 istype 'reg';

" Definitions
QSTATE = [L3Z,L2Z,L1Z,R1Z,R2Z,R3Z];  " State variables
IDLE   = [  0,  0,  0,  0,  0,  0];  " States
L3     = [  1,  1,  1,  0,  0,  0];
L2     = [  0,  1,  1,  0,  0,  0];
L1     = [  0,  0,  1,  0,  0,  0];
R1     = [  0,  0,  0,  1,  0,  0];
R2     = [  0,  0,  0,  1,  1,  0];
R3     = [  0,  0,  0,  1,  1,  1];
LR3    = [  1,  1,  1,  1,  1,  1];
```

9.1.4 The Guessing Game

A "guessing game" machine was defined in Section 7.7.1 starting on page 596, with the following description:

> Design a clocked synchronous state machine with four inputs, G1–G4, that are connected to pushbuttons. The machine has four outputs, L1–L4, connected to lamps or LEDs located near the like-numbered pushbuttons. There is also an ERR output connected to a red lamp. In normal operation, the L1–L4 outputs display a 1-out-of-4 pattern. At each clock tick, the pattern is rotated by one position; the clock frequency is about 4 Hz.
>
> Guesses are made by pressing a pushbutton, which asserts an input Gi. When any Gi input is asserted, the ERR output is asserted if the "wrong" pushbutton was pressed, that is, if the Gi input detected at the clock tick does not have the same number as the lamp output that was asserted before the clock tick. Once a guess has been made, play stops and the ERR output maintains the same value for one or more clock ticks until the Gi input is negated, then play resumes.

As we discussed in Section 7.7.1, the machine requires six states—four in which a corresponding lamp is on, and two for when play is stopped after either a good or a bad pushbutton push. An ABEL program for the guessing game is shown in Table 9-5. Two enhancements were made to improve the testability and robustness of the machine—a RESET input that forces the game to a known starting state, and the two unused states have explicit transitions to the starting state.

The guessing-game machine uses the same state assignments as the original version in Section 7.7.1. Using these assignments, the ABEL compiler

Table 9-5 ABEL program for the guessing-game machine.

```
module ggame
Title 'Guessing-Game State Machine'
GGAME device 'P16V8R';

" Inputs and outputs
CLOCK, RESET, G1..G4          pin 1, 2, 3..6;
L1..L4, ERR                   pin 12..15, 19 istype 'com';
Q2..Q0                        pin 16..18 istype 'reg';

" Sets
G = [G1..G4];
L = [L1..L4];

" States
QSTATE = [Q2,Q1,Q0];
S1     = [ 0, 0, 0];
S2     = [ 0, 0, 1];
S3     = [ 0, 1, 1];
S4     = [ 0, 1, 0];
SOK    = [ 1, 0, 0];
SERR   = [ 1, 0, 1];
EXTRA1 = [ 1, 1, 0];
EXTRA2 = [ 1, 1, 1];

state_diagram QSTATE

state S1:  IF RESET THEN SOK ELSE IF G2 # G3 # G4 THEN SERR
           ELSE IF G1 THEN SOK ELSE S2;

state S2:  IF RESET THEN SOK ELSE IF G1 # G3 # G4 THEN SERR
           ELSE IF G2 THEN SOK ELSE S3;

state S3:  IF RESET THEN SOK ELSE IF G1 # G2 # G4 THEN SERR
           ELSE IF G3 THEN SOK ELSE S4;

state S4:  IF RESET THEN SOK ELSE IF G1 # G2 # G3 THEN SERR
           ELSE IF G4 THEN SOK ELSE S1;

state SOK:  IF RESET THEN SOK
            ELSE IF G1 # G2 # G3 # G4 THEN SOK ELSE S1;

state SERR:  IF RESET THEN SOK
             ELSE IF G1 # G2 # G3 # G4 THEN SERR ELSE S1;

state EXTRA1:  GOTO SOK;
state EXTRA2:  GOTO SOK;

equations

QSTATE.CLK = CLOCK;

L1  = (QSTATE == S1);
L2  = (QSTATE == S2);
L3  = (QSTATE == S3);
L4  = (QSTATE == S4);
ERR = (QSTATE == SERR);

end ggame
```

Table 9-6 Product-term usage in the guessing-game state-machine PLD.

```
 P-Terms   Fan-in  Fan-out  Type  Name
---------  ------  -------  ----  --------
   1/3       3        1     Pin   L1
   1/3       3        1     Pin   L2
   1/3       3        1     Pin   L3
   1/3       3        1     Pin   L4
   1/3       3        1     Pin   ERR
   6/2       7        1     Pin   Q2.REG
   1/7       7        1     Pin   Q1.REG
  11/8       8        1     Pin   Q0.REG
=========
  23/32          Best P-Term Total: 16
                       Total Pins: 14
                Average P-Term/Output: 2
```

cranks out minimized equations with the number of product terms shown in Table 9-6. The Q0 output just barely fits in a GAL16V8 (eight product terms). If we needed to save terms, the way in which we've written the program allows us to try alternate state assignments (see Exercise 9.2).

A more productive alternative might be to try an output-coded state assignment. We can use one state/output bit per lamp (L1..L4), and use one more bit (ERR) to distinguish between the SOK and SERR states when the lamps are all off. This allows us to drop the equations for L1..L4 and ERR from Table 9-5. The new assignment is shown in Table 9-7. With this assignment, L1 uses two product terms and L2..L4 use only one product term each. Unfortunately, the ERR output blows up into 16 product terms.

Table 9-7 ABEL definitions for the guessing-game machine with an output-coded state assignment.

```
module ggameoc
Title 'Guessing-Game State Machine'
"GGAMEOC device 'P16V8R';

" Inputs and outputs
CLOCK, RESET, G1..G4          pin 1, 2, 3..6;
L1..L4, ERR                   pin 12..15, 18 istype 'reg';

" States
QSTATE = [L1,L2,L3,L4,ERR];
S1     = [ 1, 0, 0, 0,  0];
S2     = [ 0, 1, 0, 0,  0];
S3     = [ 0, 0, 1, 0,  0];
S4     = [ 0, 0, 0, 1,  0];
SOK    = [ 0, 0, 0, 0,  0];
SERR   = [ 0, 0, 0, 0,  1];

...
```

Part of our problem with this particular output-coded assignment is that we're not taking full advantage of its properties. Notice that it is basically a "one-hot" encoding, but the state definitions in Table 9-7 require all five state bits to be decoded for each state. An alternate version of the coding using "don't-cares" is shown in Table 9-8.

In the new version, we are assuming that the state bits never take on any combination of values other than the ones we originally defined in Table 9-7. Thus, for example, if we see that state bit L1 is 1, the machine must be in state S1 regardless of the values of any other state bits. Therefore, we can set these bits to "don't care" in S1's definition in Table 9-8. ABEL will set each X to 0 when encoding a next state, but will treat each X as a "don't-care" when decoding the current state. Thus, we must take *extreme* care to ensure that decoded states are in fact mutually exclusive, that is, that no legitimate next state matches two or more different state definitions. Otherwise, the compiled results will not have the expected behavior.

The reduced equations that result from the output coding in Table 9-8 use three product terms for L1, one each for L2..L4, and only seven for ERR. So the

```
X = .X.;
QSTATE = [L1,L2,L3,L4,ERR];
S1     = [ 1, X, X, X,  X];
S2     = [ X, 1, X, X,  X];
S3     = [ X, X, 1, X,  X];
S4     = [ X, X, X, 1,  X];
SOK    = [ 0, 0, 0, 0,  0];
SERR   = [ X, X, X, X,  1];
```

Table 9-8 Output coding for the guessing-game machine using "don't cares."

DON'T-CARE, HOW IT WORKS

To understand how the don't-cares work in a state encoding, you must first understand how ABEL creates equations internally from state diagrams. Within a given state S, each transition statement (IF-THEN-ELSE or GOTO) causes the on-sets of certain state variables to be augmented according to the transition condition. The transition condition is an expression that must be true to "go to" that target, including being in state S. For example, all of the conditions specified in a state such as S1 in Table 9-5 are implicitly ANDed with the expression "QSTATE==S1". Because of the way S1 is defined using don't-cares, this equality check generates only a single literal (L1) instead of an AND term, leading to further simplification later.

For each target state in a transition statement, the on-sets of only the state variables that are 1 in that state are augmented according to the transition condition. Thus, when a coded state such as S1 in Table 9-8 appears as a target in any transition statement, only the on-set of L1 is augmented. This explains why the actual coding of state S1 as a target is 100000.

change was worthwhile. However, we must remember that the new machine is different from the one in Table 9-7. Consider what happens if the machine ever gets into an unspecified state. In the original machine with fully specified output coding, there are no next-states for the $2^5 - 6 = 26$ unspecified states, so the state machine will always go to the state coded 00000 (SOK) from unspecified states. In the new machine, "unspecified" states aren't really unspecified; for example, the state coded 11111 actually matches five coded states, S1–S4 and SERR. The next state will actually be the "OR" of next-states for the matching coded states. (Read the box on the previous page to understand why these outcomes occur.) Again, you need to be careful.

RESETTING EXPECTATIONS

Reading the guessing-game program in Table 9-5, you would expect that the RESET input would force the machine to the SOK state, and it does. However, the moment that you have unspecified or partially coded states as in Tables 9-7 or 9-8, don't take anything for granted.

Referring to the box on the previous page, remember that transition statements in ABEL state machines augment the on-sets of state variables. If a particular, unused state combination does not match any of the states for which transition statements were written, then no on-sets will be augmented. Thus, the only transition from that state will be to the state with the all-0s coding.

For this reason, it is useful to code the reset state or a "safe" state as all 0s. If this is not possible, but the all-0s state is still unused, you can explicitly provide a transition from the all-0s state to a desired safe state.

9.1.5 Reinventing Traffic-Light Controllers

Our final example is from the world of cars and traffic. Traffic-light controllers in California, especially in the fair city of Sunnyvale, are carefully designed to *maximize* the waiting time of cars at intersections. An infrequently used intersection (one that would have no more than a "yield" sign if it were in Chicago) has the sensors and signals shown in Figure 9-5. The state machine that controls the traffic signals uses a 1 Hz clock and a timer and has four inputs:

NSCAR Asserted when a car on the north-south road is over either sensor on either side of the intersection.

EWCAR Asserted when a car on the east-west road is over either sensor on either side of the intersection.

TMLONG Asserted if more than five minutes has elapsed since the timer started; remains asserted until the timer is reset.

TMSHORT Asserted if more than five seconds has elapsed since the timer started; remains asserted until the timer is reset.