Thus, the essential hazard in the pulse catcher is detected by the arrows in Figure 7-102, starting in internal state 10 with P R = 10.

A fundamental-mode circuit must have at least three states to have an essential hazard, so latches don't have them. On the other hand, all flip-flops (circuits that sample inputs on a clock edge) do.

*7.10.7 Summary

In summary, you use the following steps to design a feedback sequential circuit:

1. Construct a primitive flow table from the circuit's word description.
2. Minimize the number of states in the flow table.
3. Find a race-free assignment of coded states to named states, adding auxiliary states or splitting states as required.
4. Construct the transition table.
5. Construct excitation maps and find a hazard-free realization of the excitation equations.
6. Check for essential hazards. Modify the circuit if necessary to ensure that minimum excitation and feedback delays are greater than maximum inverter or other input-logic delays.
7. Draw the logic diagram.

Also note that some circuits routinely violate the basic fundamental-mode assumption that inputs change one at a time. For example, in a positive-edge-triggered D flip-flop, the D input may change at the same time that CLK changes from 1 to 0, and the flip-flop still operates properly. The same thing certainly cannot be said at the 0-to-1 transition of CLK. Such situations require analysis of the transition table and circuit on a case-by-case basis if proper operation in "special cases" is to be guaranteed.

A FINAL QUESTION Given the difficulty of designing fundamental-mode circuits that work properly, let alone ones that are fast or compact, how did anyone ever come up with the 6-gate, 8-state, commercial D flip-flop design in Figure 7-20? Don't ask me, I don't know!

7.11 ABEL Sequential-Circuit Design Features

7.11.1 Registered Outputs

ABEL has several features that support the design of sequential circuits. As we'll show in \secref{SeqPLDs}, most PLD outputs can be configured by the user to be *registered outputs* that provide a D flip-flop following the AND-OR logic, as in Figure 7-103. To configure one or more outputs to be registered, an

registered output

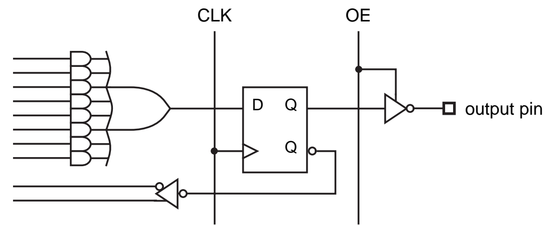

Figure 7-103
PLD registered output.

reg

ABEL program's pin declarations normally must contain an `istype` clause using the keyword "`reg`" (rather than "`com`") for each registered output. Table 7-22 is an example program that has three registered outputs and two combinational outputs.

As suggested by Figure 7-103, a registered output has at least two other attributes associated with it. The three-state buffer driving the output pin has an output-enable input OE, and the flip-flop itself has a clock input CLK. As shown in Table 7-22, the signals that drive these inputs are specified in the equations section of the program. Each input signal is specified as the corresponding main output signal name followed by an attribute suffix, *.CLK* or *.OE*. Some PLDs have flip-flops with additional controllable inputs; for example, preset and clear inputs have attribute suffixes `.PR` and `.RE` (reset). And some PLDs provide flip-flop types other than D; their inputs are specified with suffixes like `.J` and `.K`.

.CLK
.OE
.PR
.RE

Within the `equations` section of the ABEL program, the logic values for registered outputs are established using the *clocked assignment operator, :=*. When the PLD is compiled, the expression on the right-hand-side will be applied to the D input of the output flip-flop. All of the same rules as for combinational outputs apply to output polarity control, don't-cares, and so on. In Table 7-22, the state bits Q1–Q3 are registered outputs, so they use clocked assignment, "`:=`". The UNLK and HINT signals are Mealy outputs, combinational functions of current state and input, so they use unclocked assignment, "=". A machine with pipelined outputs (Figure 7-37 on page 454), would instead use clocked assignment for such outputs.

clocked assignment operator, :=

ABEL's truth-table syntax (Table 4-16 on page 255) can also be used with registered outputs. The only difference is that "->" operator between input and output items is changed to "`:>`".

clocked truth-table operator, :>

IS `istype` ESSENTIAL?

Older devices, such as the PAL16Rx family, contain a fixed, preassigned mix of combinational and registered outputs and are not configurable. With these devices, the compiler can deduce each output's type from its pin number and the `istype` statement is not necessary. Even with configurable outputs, some compilers can correctly the output type from the equations. Still, it's a good idea to include the `istype` information anyway, both as a double check and to enhance design portability.

Table 7-22
ABEL program using registered outputs.

```
module CombLock
Title 'Combination-Lock State Machine'

" Input and Outputs
X, CLOCK        pin;
UNLK, HINT      pin istype 'com';
Q1, Q2, Q3      pin istype 'reg';

Q = [Q1..Q3];

Equations

Q.CLK = CLOCK; Q.OE = 1;

" State variables
Q1 := Q1 & !Q2 & X  #  !Q1 & Q2 & Q3 & !X  #  Q1 & Q2 & !Q3;
Q2 := !Q2 & Q3 & X  #  Q2 & !Q3 & X;
Q3 := Q1 & !Q2 & !Q3  #  Q1 & Q3 & !X  #  !Q2 & !X
      #  !Q1 & !Q3 & !X  #  Q2 & !Q3 & X;

" Mealy outputs
UNLK = Q1 & Q2 & Q3 & !X;
HINT = !Q1 & !Q2 & !Q3 & !X  #  Q1 & !Q2 & X  #  !Q2 & Q3 & X
       #  Q2 & Q3 & !X  #  Q2 & !Q3 & X;

end CombLock
```

You can also design feedback sequential circuits in ABEL, without using any of the language's sequential-circuit features. For example, in \secref{ABEL-latch} we show how to specify latches using ABEL.

7.11.2 State Diagrams

The state-machine example in the previous subsection is just a transcription of the combination-lock machine that we synthesized by hand in Section 7.4.6 beginning on page 484. However, most PLD programming languages have a notation for defining, documenting, and synthesizing state machines directly, without ever writing a state, transition, or excitation table or deriving excitation equations by hand. Such a notation is called a *state-machine description language*. In ABEL, this notation is called a "state diagram," and the ABEL compiler does all the work of generating excitation equations that realize the specified machine.

state-machine description language

In ABEL, the keyword *state_diagram* indicates the beginning of a state-machine definition. Table 7-23 shows the textual structure of an ABEL "state diagram." Here *state-variables* is an ABEL set that lists the state variables of the machine. If there are n variables in the set, then the machine has 2^n possible states corresponding to the 2^n different assignments of constant values to

state_diagram

state-variables

Table 7-23
Structure of a "state diagram" in ABEL.

```
state_diagram state-variables
state state-value 1 : transition statement;
state state-value 2 : transition statement;
...
state state-value 2^n : transition statement;
```

variables in the set. States are usually given symbolic names in an ABEL program; this makes it easy to try different assignments simply by changing the constant definitions.

state
state-value
equation

An equation for each state variable is developed according to the information in the "state diagram." The keyword `state` indicates that the next states and current outputs for a particular current state are about to be defined; a *state-value* is a constant that defines state-variable values for the current state. A *transition statement* defines the possible next states for the current state.

GOTO statement
IF statement

ABEL has two commonly used transition statements. The *GOTO statement* unconditionally specifies the next state, for example "`GOTO INIT`". The *IF statement* defines the possible next states as a function of an arbitrary logic expressions. (There's also a seldom-used `CASE` statement which we don't cover.)

Table 7-24 shows the syntax of the ABEL `IF` statement. Here *TrueState* and *FalseState* are state values that the machine will go to if *LogicExpression* is true or false, respectively. Like `WHEN` statements, `IF` statements can be nested; that is, *FalseState* can itself be another `IF` statement. Also, *TrueState* can be another `IF` statement if it is enclosed in braces.

Our first example using ABEL's "state diagram" capability is based on our first state-machine design example from Section 7.4.1 on page 466. A state table for this machine was developed in Figure 7-49 on page 469. This state diagram is adapted to ABEL in Table 7-25. Several characteristics of this program should be noted:

- The definition of `QSTATE` uses three variables to encode state.
- The definitions of `INIT`–`XTRA3` determine the individual state encodings.
- `IF-THEN-ELSE` statements are nested. A particular next state may appear in multiple places in one set of nested `IF-THEN-ELSE` clauses (e.g., see states `OK0` and `OK1`).
- Expressions like "`(B==1)*(A==0)`" were used instead of equivalents like "`B*!A`" only because the former are a bit more readable.

Table 7-24
Structure of an ABEL `IF` statement.

```
IF LogicExpression THEN
   TrueState;
ELSE
   FalseState;
```

Table 7-25
An example of ABEL's state-diagram notation.

```
module SMEX1
title 'PLD Version of Example State Machine'

" Input and output pins
CLOCK, RESET_L, A, B           pin;
Q1..Q3                         pin istype 'reg';
Z                              pin istype 'com';

" Definitions
QSTATE = [Q1,Q2,Q3];          " State variables
INIT   = [ 0, 0, 0];
A0     = [ 0, 0, 1];
A1     = [ 0, 1, 0];
OK0    = [ 0, 1, 1];
OK1    = [ 1, 0, 0];
XTRA1  = [ 1, 0, 1];
XTRA2  = [ 1, 1, 0];
XTRA3  = [ 1, 1, 1];
RESET = !RESET_L;

state_diagram QSTATE

state INIT: IF RESET THEN INIT
            ELSE IF (A==0) THEN A0
            ELSE A1;

state A0:   IF RESET THEN INIT
            ELSE IF (A==0) THEN OK0
            ELSE A1;

state A1:   IF RESET THEN INIT
            ELSE IF (A==0) THEN A0
            ELSE OK1;

state OK0:  IF RESET THEN INIT
            ELSE IF (B==1)&(A==0) THEN OK0
            ELSE IF (B==1)&(A==1) THEN OK1
            ELSE IF (A==0) THEN OK0
            ELSE IF (A==1) THEN A1;

state OK1:  IF RESET THEN INIT
            ELSE IF (B==1)&(A==0) THEN OK0
            ELSE IF (B==1)&(A==1) THEN OK1
            ELSE IF (A==0) THEN A0
            ELSE IF (A==1) THEN OK1;

state XTRA1: GOTO INIT;
state XTRA2: GOTO INIT;
state XTRA3: GOTO INIT;

equations

QSTATE.CLK = CLOCK; QSTATE.OE = 1;
Z = (QSTATE == OK0) # (QSTATE == OK1);

END SMEX1
```

Table 7-26
Reduced equations for SMEX1 PLD.

```
Q1 := (!Q2.FB & !Q3.FB & RESET_L
     # Q1.FB & RESET_L);
Q1.C = (CLOCK);
Q1.OE = (1);

Q2 := (Q1.FB & !Q3.FB & RESET_L & !A
     # Q1.FB & Q3.FB & RESET_L & A
     # Q1.FB & Q2.FB & RESET_L & B);
Q2.C = (CLOCK);
Q2.OE = (1);

Q3 := (!Q2.FB & !Q3.FB & RESET_L & A
     # Q1.FB & RESET_L & A);
Q3.C = (CLOCK);
Q3.OE = (1);

Z = (Q2 & Q1);
```

- The first IF statement in each of states INIT–OK1 ensures that the machine goes to the INIT state if RESET is asserted.
- Next-state equations are given for XTRA1–XTRA3 to ensure that the machine goes to a "safe" state if it somehow gets into an unused state.
- The single equation in the "equations" section of the program determines the behavior of the Moore-type output.

Table 7-26 shows the resulting excitation and output equations produced by ABEL compiler (the reverse-polarity equations are not shown). Notice the use of variable names like "Q1.FB" in the right-hand sides of the equations. Here, the ".FB" attribute suffix refers to the "feedback" signal into the AND-OR array coming from the flip-flop's Q output. This is done to make it clear that the signal is coming from the flip-flop, not from the corresponding PLD output pin, which can be selected in some complex PLDs. As shown in Figure 7-104, ABEL actually allows you to select among three possible values on the right-hand side of an equation using an attribute suffix on the signal name:

USE IT OR ELSE

The ELSE clause of an IF statement is optional. If it is omitted, however, the next state for some input combinations will be unspecified. Usually this is not the designer's intention.

Nevertheless, if you can absolutely guarantee that the unspecified input combinations will never occur, you may be able to reduce the size of the transition logic. The ABEL compiler will treat the transition outputs for the unspecified input combinations as "don't-cares" if the @DCSET directive is given.

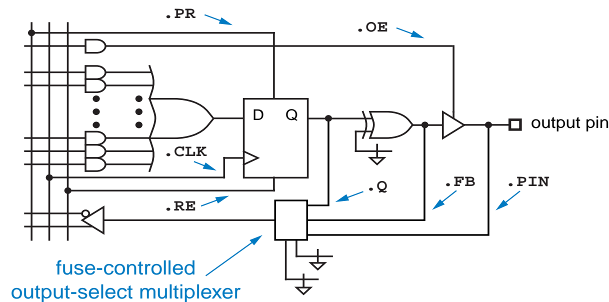

Figure 7-104
Output selection capability in a complex PLD.

`.Q` The actual flip-flop output pin before any programmable inversion. *.Q*

`.FB` A value equal to the value that the output pin would have if enabled. *.FB*

`.PIN` The actual signal at the PLD output pin. This signal is floating or driven by another device if the three-state driver is not enabled. *.PIN*

Obviously, the `.PIN` value should not be used in a state-machine excitation equation since it is not guaranteed always to equal the state variable.

Despite the use of "high-level language," the program's author still had to refer to the original, hand-constructed state table in Figure 7-49 to come up with ABEL version in Table 7-18. A different approach is shown in Table 7-27. This program was developed directly from the word description of the state machine, which is repeated below:

> Design a clocked synchronous state machine with two inputs, A and B, and a single output Z that is 1 if:
>
> – A had the same value at each of the two previous clock ticks, *or*
>
> – B has been 1 since the last time that the first condition was true.
>
> Otherwise, the output should be 0.

A key idea in the new approach is to remove the last value of `A` from the state definitions, and instead to have a separate flip-flop that keeps track of it (`LASTA`). Then only two non-`INIT` states must be defined: `LOOKING` ("still looking for a match") and `OK` ("got a match or `B` has been 1 since last match"). The `Z` output is a simple combinational decode of the `OK` state.

PHANTOM (OF THE) OPERAND

Real CPLDs typically have only a two-input output-select multiplexer and omit the `.FB` input shown in Figure 7-104. When an equation calls for a signal with the `.FB` attribute, the ABEL compiler uses the corresponding `.Q` signal and simply adjusts it with the appropriate inversion (or not).

Table 7-27
A more "natural" ABEL program for the example state machine.

```
module SMEX2
title 'Alternate Version of Example State Machine'

" Input and output pins
CLOCK, RESET_L, A, B            pin;
LASTA, Q1, Q2                   pin istype 'reg';
Z                               pin istype 'com';

" Definitions
QSTATE  = [Q1,Q2];               " State variables
INIT    = [ 0, 0];               " State encodings
LOOKING = [ 0, 1];
OK      = [ 1, 0];
XTRA    = [ 1, 1];
RESET = !RESET_L;

state_diagram QSTATE

state INIT:    IF RESET THEN INIT ELSE LOOKING;

state LOOKING: IF RESET THEN INIT
               ELSE IF (A == LASTA) THEN OK
               ELSE LOOKING;

state OK:      IF RESET THEN INIT
               ELSE IF B THEN OK
               ELSE IF (A == LASTA) THEN OK
               ELSE LOOKING;

state XTRA:    GOTO INIT;

equations
LASTA.CLK = CLOCK; QSTATE.CLK = CLOCK; QSTATE.OE = 1;

LASTA := A;
Z = (QSTATE == OK);

END SMEX2
```

*7.11.3 External State Memory

In some situations, the state memory of a PLD-based state machine may be kept in flip-flops external to the PLD. ABEL provides a special version of the `state_diagram` statement to handle this situation:

`state_diagram` *current-state-variables* -> *next-state variables*

current-state-variables
next-state-variables

Here *current-state-variables* is an ABEL set that lists the input signals which represent the current state of the machine, and *next-state-variables* is a set that lists the corresponding output signals which are the excitation for external D flip-flops holding the state of the machine, for example,

```
state_diagram [CURQ1, CURQ2] -> [NEXTQ1, NEXTQ2]
```

```
state_diagram state-variables

state state-value 1 :
        optional equation ;
        optional equation ;
        . . .
        transition statement ;

state state-value 2 :
        optional equation ;
        optional equation ;
        . . .
        transition statement ;

. . .

state state-value 2^n :
        optional equation ;
        optional equation ;
        . . .
        transition statement ;
```

Table 7-28
Structure of an ABEL state diagram with Moore outputs defined.

*7.11.4 Specifying Moore Outputs

The output `Z` in our example state machine is a Moore output, a function of state only, and we defined this output in Tables 7-25 and 7-27 using an appropriate equation in the `equations` section of the program. Alternatively, ABEL allows Moore outputs to be specified along with the state definitions themselves. The transition statement in a `state` definition may be preceded by one or more optional equations, as shown in Table 7-28. To use this capability with the machine in Table 7-27, for example, we would eliminate the `Z` equation in the `equations` section, and rewrite the state diagram as shown in Table 7-29.

As in other ABEL equations, when a variable such as `Z` appears on the left-hand side of multiple equations, the right-hand sides are OR'ed together to form the final result (as discussed in Section 4.6.3). Also notice that `Z` is still specified

```
state_diagram QSTATE

state INIT:     Z = 0;
                IF RESET THEN INIT ELSE LOOKING;

state LOOKING:  Z = 0;
                IF RESET THEN INIT
                ELSE IF (A == LASTA) THEN OK
                ELSE LOOKING;

state OK:       Z = 1;
                IF RESET THEN INIT
                ELSE IF B THEN OK
                ELSE IF (A == LASTA) THEN OK
                ELSE LOOKING;

state XTRA:     Z = 0;
                GOTO INIT;
```

Table 7-29
State machine with embedded Moore output definitions.

as a combinational, not registered, output. If Z were a registered output, the desired output value would occur one clock tick after the machine visited the corresponding state.

*7.11.5 Specifying Mealy and Pipelined Outputs

Some state-machine outputs are functions of the inputs as well as state. In Section 7.3.2, we called them Mealy outputs or pipelined outputs, depending on whether they occurred immediately upon an input change or only after a clock edge. ABEL's WITH statement provides a way to specify these outputs side-by-side with the next states, rather than separately in the equations section of the program.

As shown in Table 7-30, the syntax of the WITH statement is very simple. Any next-state value which is part of a transition statement can be followed by the keyword WITH and a bracketed list of equations that are "executed" for the specified transition. Formally, let "E" an excitation expression that is true only when the specified transition is to be taken. Then for each equation in the WITH's bracketed list, the right-hand side is AND'ed with E and assigned to the left-hand side. The equations can use either unclocked or clocked assignment to create Mealy or pipelined outputs, respectively.

Table 7-30 Structure of ABEL WITH statement.

```
next-state WITH {
        equation;
        equation;
        ...
}
```

We developed an example "combination lock" state machine with Mealy outputs in Table 7-14 on page 484. The same state machine is specified by the ABEL program in Table 7-31, using WITH statements for the Mealy outputs. Note that closing brackets take the place of the semicolons that normally end the transition statements for the states.

Based on the combination lock's word description, it is not possible to realize UNLK and HINT as pipelined outputs, since they depend on the current value of X. However, if we redefine UNLK to be asserted for the entire "unlocked" state, and HINT to be the actual recommended next value of X, we can create a new machine with pipelined outputs, as shown in Table 7-32. Notice that we used the clocked assignment operator for the outputs. More importantly, notice that the values of UNLK and HINT are different than in the Mealy example, since they have to "look ahead" one clock tick.

Because of "lookahead," pipelined outputs can be more difficult than Mealy outputs to design and understand. In the example above, we even had to modify the problem statement to accommodate them. The advantage of

Table 7-31
State machine with embedded Mealy output definitions.

```
module SMEX4
title 'Combination-Lock State Machine'

" Input and output pins
CLOCK, X                        pin;
Q1..Q3                          pin istype 'reg';
UNLK, HINT                      pin istype 'com';

" Definitions
S        = [Q1,Q2,Q3];          " State variables
ZIP      = [ 0, 0, 0];          " State encodings
X0       = [ 0, 0, 1];
X01      = [ 0, 1, 0];
X011     = [ 0, 1, 1];
X0110    = [ 1, 0, 0];
X01101   = [ 1, 0, 1];
X011011  = [ 1, 1, 0];
X0110111 = [ 1, 1, 1];

state_diagram S

state ZIP:      IF X==0 THEN X0    WITH {UNLK = 0; HINT = 1}
                ELSE ZIP           WITH {UNLK = 0; HINT = 0}

state X0:       IF X==0 THEN X0    WITH {UNLK = 0; HINT = 0}
                ELSE X01           WITH {UNLK = 0; HINT = 1}

state X01:      IF X==0 THEN X0    WITH {UNLK = 0; HINT = 0}
                ELSE X011          WITH {UNLK = 0; HINT = 1}

state X011:     IF X==0 THEN X0110 WITH {UNLK = 0; HINT = 1}
                ELSE ZIP           WITH {UNLK = 0; HINT = 0}

state X0110:    IF X==0 THEN X0    WITH {UNLK = 0; HINT = 0}
                ELSE X01101        WITH {UNLK = 0; HINT = 1}

state X01101:   IF X==0 THEN X0    WITH {UNLK = 0; HINT = 0}
                ELSE X011011       WITH {UNLK = 0; HINT = 1}

state X011011:  IF X==0 THEN X0110 WITH {UNLK = 0; HINT = 0}
                ELSE X0110111      WITH {UNLK = 0; HINT = 1}

state X0110111: IF X==0 THEN X0    WITH {UNLK = 1; HINT = 1}
                ELSE ZIP           WITH {UNLK = 0; HINT = 0}

equations
S.CLK = CLOCK;

END SMEX4
```

pipelined outputs is that, since they are connected directly to register outputs, they are valid a few gate-delays sooner after a state change than Moore or Mealy outputs, which normally include additional combinational logic. In the combination-lock example, it's probably not that important to open your lock or see your hint a few nanoseconds earlier. However, shaving off a few gate delays can be quite important in high-speed applications.

Table 7-32
State machine with embedded pipelined output definitions.

```
module SMEX5
title 'Combination-Lock State Machine'

" Input and output pins
CLOCK, X                        pin;
Q1..Q3                          pin istype 'reg';
UNLK, HINT                      pin istype 'reg';

" Definitions
S        = [Q1,Q2,Q3];          " State variables
ZIP      = [ 0, 0, 0];          " State encodings
X0       = [ 0, 0, 1];
X01      = [ 0, 1, 0];
X011     = [ 0, 1, 1];
X0110    = [ 1, 0, 0];
X01101   = [ 1, 0, 1];
X011011  = [ 1, 1, 0];
X0110111 = [ 1, 1, 1];

state_diagram S

state ZIP:      IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE ZIP           WITH {UNLK := 0; HINT := 0}

state X0:       IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE X01           WITH {UNLK := 0; HINT := 1}

state X01:      IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE X011          WITH {UNLK := 0; HINT := 0}

state X011:     IF X==0 THEN X0110 WITH {UNLK := 0; HINT := 1}
                ELSE ZIP           WITH {UNLK := 0; HINT := 0}

state X0110:    IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE X01101        WITH {UNLK := 0; HINT := 1}

state X01101:   IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE X011011       WITH {UNLK := 0; HINT := 1}

state X011011:  IF X==0 THEN X0110 WITH {UNLK := 0; HINT := 1}
                ELSE X0110111      WITH {UNLK := 1; HINT := 0}

state X0110111: IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE ZIP           WITH {UNLK := 0; HINT := 0}

equations
S.CLK = CLOCK; UNLK.CLK = CLOCK; HINT.CLK = CLOCK;

END SMEX5
```

7.11.6 Test Vectors

.C., clock edge

Test vectors for sequential circuits in ABEL have the same uses and limitations as test vectors for combinational circuits, as described in Section 4.6.7. One important addition to their syntax is the use of the constant “.C.” to denote a clock edge, 0→1→0. Thus, Table 7-33 is an ABEL program, with test vectors, for a simple 8-bit register with a clock-enable input. A variety of vectors are used to test loading and holding different input values.

```
module REG8EN
title '8-bit register with clock enable'

" Input and output pins
CLK, EN, D1..D8        pin;
Q1..Q8                 pin istype 'reg';

" Sets
D = [D1..D8];
Q = [Q1..Q8];

equations

Q.CLK = CLK;

WHEN EN == 1 THEN Q := D ELSE Q := Q;

test_vectors ([CLK, EN,  D  ] -> [ Q  ])
              [.C.,  1, ^h00] -> [^h00]; " 0s in every bit
              [.C.,  0, ^hFF] -> [^h00]; " Hold capability, EN=0
              [.C.,  1, ^hFF] -> [^hFF]; " 1s in every bit
              [.C.,  0, ^h00] -> [^hFF]; " Hold capability
              [.C.,  1, ^h55] -> [^h55]; " Adjacent bits shorted
              [.C.,  0, ^hAA] -> [^h55]; " Hold capability
              [.C.,  1, ^hAA] -> [^hAA]; " Adjacent bits shorted
              [.C.,  1, ^h55] -> [^h55]; " Load with quick setup
              [.C.,  1, ^hAA] -> [^hAA]; " Again
END REG8EN
```

Table 7-33 ABEL program with test vectors for a simple 8-bit register.

A typical approach to testing state machines is to write vectors that not only cause the machine to visit every state, but also to exercise every transition from every state. A key difference and challenge compared to combinational-circuit test vectors is that the vectors must first drive the machine into the desired state before testing a transition, and then come back again for each different transition from that state.

Thus, Table 7-34 shows test vectors for the state machine in Table 7-27. It's important to understand that, unlike combinational vectors, these vectors work only if applied in exactly the order they are written. Notice that the vectors were written to be independent of the state encoding. As a result, they don't have to be modified if the state encoding is changed.

We encounter another challenge if we attempt to create test vectors for the combination-lock state machine of Table 7-31 on page 539. This machine has a major problem when it comes to testing—it has no reset input. Its starting state at power-up may be different in PLD devices and technologies—the individual flip-flops may be all set, all reset, or all in random states. In the machine's actual application, we didn't necessarily need a reset input, but for testing purposes we somehow have to get to a known starting state.

synchronizing sequence

Luckily, the combination-lock machine has a *synchronizing sequence*—a fixed sequence of one or more input values that will always drive it to a certain known state. In particular, starting from any state, if we apply X=1 to the

Table 7-34 Test vectors for the state machine in Table 7-27.

```
test_vectors
([RESET_L, CLOCK, A, B] -> [QSTATE , LASTA, Z])
 [   0    ,  .C. , 0, 0] -> [INIT    ,   0   , 0]; " Check -->INIT (RESET)
 [   0    ,  .C. , 1, 0] -> [INIT    ,   1   , 0]; "  and LASTA flip-flop
 [   1    ,  .C. , 0, 0] -> [LOOKING,   0   , 0]; " Come out of initialization
 [   0    ,  .C. , 0, 0] -> [INIT    ,   0   , 0]; " Check LOOKING-->INIT (RESET)
 [   1    ,  .C. , 0, 0] -> [LOOKING,   0   , 0]; " Come out of initialization
 [   1    ,  .C. , 1, 0] -> [LOOKING,   1   , 0]; " --> LOOKING since 0!=1
 [   1    ,  .C. , 1, 0] -> [OK      ,   1   , 1]; " --> OK since 1==1
 [   0    ,  .C. , 0, 0] -> [INIT    ,   0   , 0]; " Check OK-->INIT (RESET)
 [   1    ,  .C. , 0, 0] -> [LOOKING,   0   , 0]; " Go back towards OK ...
 [   1    ,  .C. , 0, 0] -> [OK      ,   0   , 1]; " --> OK since 0==0
 [   1    ,  .C. , 1, 1] -> [OK      ,   1   , 1]; " --> OK since B, even though 1!=0
 [   1    ,  .C. , 1, 0] -> [OK      ,   1   , 1]; " --> OK since 1==1
 [   1    ,  .C. , 0, 0] -> [LOOKING,   0   , 0]; " --> LOOKING since 0!=1
```

machine for four ticks, we will always be in state ZIP by the fourth tick. This is the approach taken by the first four vectors in Table 7-35. Until we get to the known state, we indicate the next-state on the right-hand side of the vector as being "don't care," so the simulator or the physical device tester will not flag a random state as an error.

Once we get going, we encounter something else that's new—Mealy outputs that must be tested. As shown by the fourth and fifth vectors, we don't have to transition the clock in every test vector. Instead, we can keep the clock fixed at 0, where the last transition left it, and observe the Mealy output values produced by the two input values of X. Then we can test the next state transition.

For the state transitions, we list the expected next state but we show the output values as don't-cares. For a correct test vector, the outputs must show the values attained *after* the transition, a function of the *next* state. Although it's possible to figure them out and include them, the complexity is enough to give you a headache, and they will be tested by the next CLOCK=0 vectors anyway.

Creating test vectors for a state machine by hand is a painstaking process, and no matter how careful you are, there's no guarantee that you've tested all its functions and potential hardware faults. For example, the vectors in Table 7-34 do not test (A LASTA) = 10 in state LOOKING, or (A B LASTA) = 100 in state OK. Thus, generating a complete set of test vectors for fault-detection purposes is a process best left to an automatic test-generation program. In Table 7-35, we

SYNCHRONIZING SEQUENCES AND RESET INPUTS

We lucked out with the combination-lock; not all state machines have synchronizing sequences. This is why most state machines are designed with a reset input, which in effect allows a synchronizing sequence of length one.

Table 7-35 Test vectors for the combination-lock state machine of Table 7-31.

```
test_vectors
([CLOCK, X] -> [   S    , UNLK, HINT])
 [ .C. , 1] -> [.X.     , .X. , .X. ]; " Since no reset input, apply
 [ .C. , 1] -> [.X.     , .X. , .X. ]; "   a 'synchronizing sequence'
 [ .C. , 1] -> [.X.     , .X. , .X. ]; "   to reach a known starting
 [ .C. , 1] -> [ZIP     , .X. , .X. ]; "   state
 [  0  , 0] -> [ZIP     ,  0  ,  1  ]; " Test Mealy outputs for both
 [  0  , 1] -> [ZIP     ,  0  ,  0  ]; "   values of X
 [ .C. , 1] -> [ZIP     , .X. , .X. ]; " Test ZIP-->ZIP (X==1)
 [ .C. , 0] -> [X0      , .X. , .X. ]; "   and ZIP-->X0 (X==0)
 [  0  , 0] -> [X0      ,  0  ,  0  ]; " Test Mealy outputs for both
 [  0  , 1] -> [X0      ,  0  ,  1  ]; "   values of X
 [ .C. , 0] -> [X0      , .X. , .X. ]; " Test X0-->X0 (X==0)
 [ .C. , 1] -> [X01     , .X. , .X. ]; "   and X0-->X01 (X==1)
 [  0  , 0] -> [X01     ,  0  ,  0  ]; " Test Mealy outputs for both
 [  0  , 1] -> [X01     ,  0  ,  1  ]; "   values of X
 [ .C. , 0] -> [X0      , .X. , .X. ]; " Test X01-->X0 (X==0)
 [ .C. , 1] -> [X01     , .X. , .X. ]; " Get back to X01
 [ .C. , 1] -> [X011    , .X. , .X. ]; " Test X01-->X011 (X==1)
```

petered out after writing vectors for the first few states; completing the test vectors is left as an exercise (7.74). Still, on the functional testing side, writing a few vectors to exercise the machine's most basic functions can weed out obvious design errors early in the process. More subtle design errors are best detected by a thorough system-level simulation.