XSabl: Sequential ABEL Examples

Early digital designers and many designers through the 1980s wrote out state tables by hand and built corresponding circuits using the synthesis methods that we described in Section 7.4. However, hardly anyone does that anymore. Nowadays, most state tables are specified with a hardware description language (HDL) such as ABEL, VHDL, or Verilog. The HDL compiler then performs the equivalent of our synthesis methods and realizes the specified machine in a PLD, CPLD, FPGA, ASIC, or other target technology.

This section gives state-machine and other sequential-circuit design examples in ABEL, and we target the resulting machines to small PLDs. Some of these examples illustrate the partitioning decisions that are needed when an entire circuit does not fit into a single component.

In Section 7.4 we designed several simple state machines by translating their word descriptions into a state table, choosing a state assignment, and finally synthesizing a corresponding circuit. We repeated one of these examples using ABEL and a PLD in Table 7-24 on page 618 and another in Table 7-28 on page 621. These designs were much easier to do using ABEL and a PLD, for two reasons:

- You don't have to be too concerned about the complexity of the resulting excitation equations, as long as they fit in the PLD.
- You may be able to take advantage of ABEL language features to make the design easier to express and understand.

Before looking at more examples, let's examine the timing behavior and packaging considerations for state machines that are built from PLDs.

XSabl.1 Timing and Packaging of PLD-Based State Machines

Figure XSabl-1 on the next page shows how a generic PLD with both combinational and registered outputs might be used in a state-machine application. Timing parameters t_{CO} and t_{PD} were explained in Section 8.3.2; t_{CO} is the flip-flop clock-to-output delay and t_{PD} is the delay through the AND-OR array.

State variables are assigned to registered outputs, of course, and are stable at time t_{CO} after the rising edge of CLOCK. Mealy-type outputs are assigned to combinational outputs and are stable at time t_{PD} after any input change that affects them. Mealy-type outputs may also change after a state change, in which case they become stable at time $t_{CO} + t_{PD}$ after the rising edge of CLOCK.

A Moore-type output is, by definition, a combinational logic function of the current state, so it is also stable at time $t_{CO} + t_{PD}$ after the CLOCK. Thus, Moore outputs may not offer any speed advantage over Mealy outputs in a PLD realization. For faster propagation delay, we defined and used pipelined outputs in Sections 7.3.2 and 7.11.5. In a PLD realization, these outputs come directly from a flip-flop output and thus have a delay of only t_{CO} from the clock. Besides

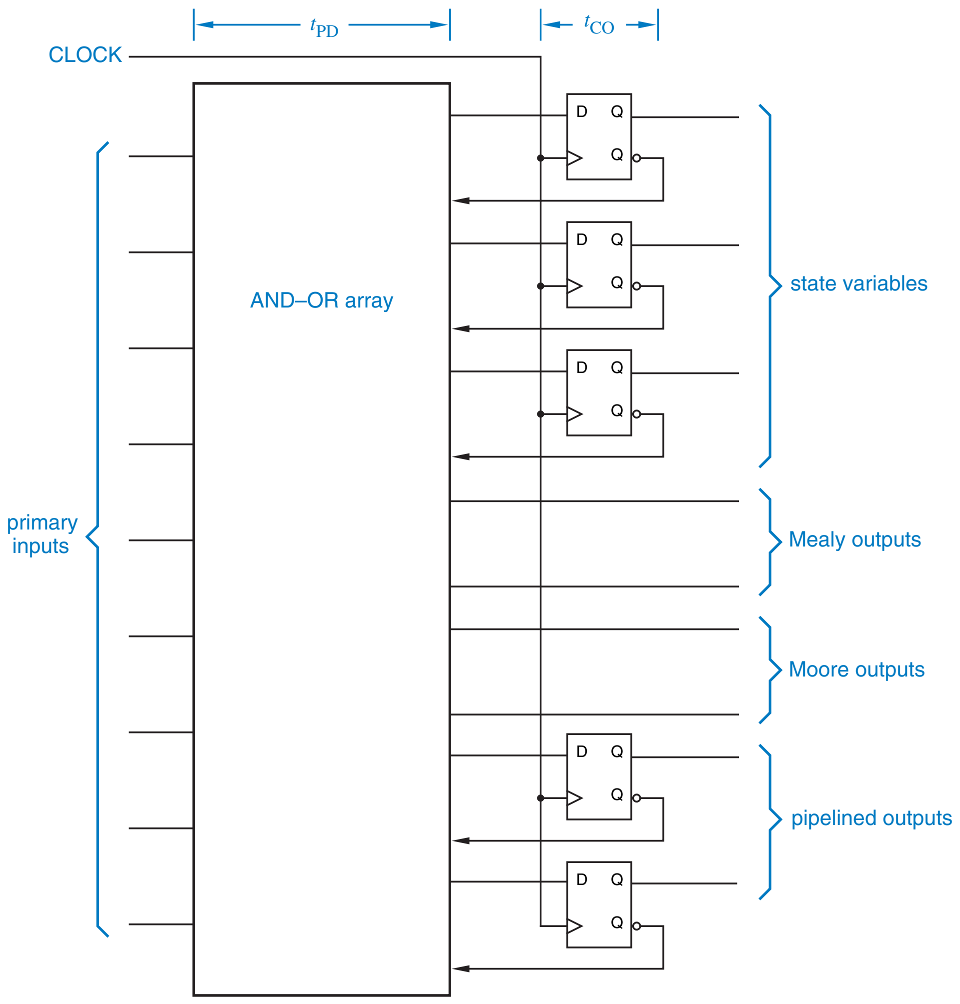


Figure XSabl-1 Structure and timing of a PLD used as a state machine.

having a shorter delay, they are also guaranteed to be glitch free, important in some applications.

PLD-based state-machine designs are often limited by the number of input and output pins available in a single PLD. According to the model of Figure XSabl-1, one PLD output is required for each state variable and for each Mealy- or Moore-type output. For example, the T-bird tail-lights machine of Section 7.5 starting on page 571 requires three registered outputs for the state variables and six combinational outputs for the lamp outputs, too many for most of the PLDs that we've described, except for the 22V10.

On the other hand, an output-coded state assignment (Section 7.3.2) usually requires a smaller total number of PLD outputs. Using an output-coded state assignment, the T-bird tail-lights machine can be built in a single 16V8, as we'll show in Section XSabl.3.

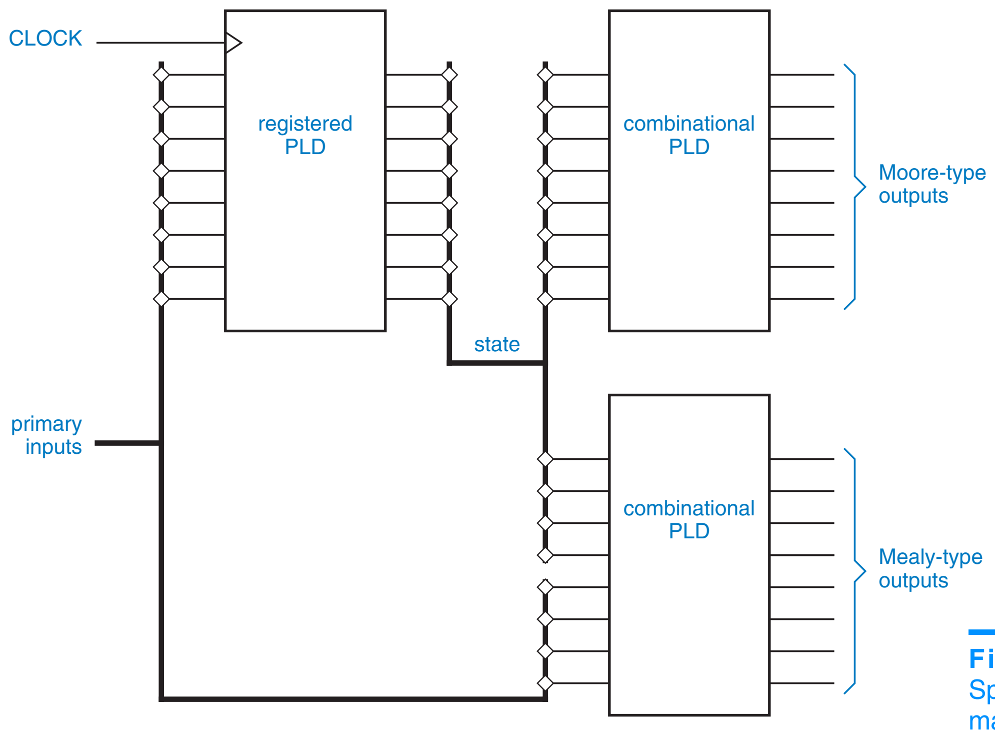


Figure XSabl-2 Splitting a state-machine design into three PLDs.

Like any state variable or Moore-type output, an output-coded state variable is stable at time t_{CO} after the clock edge. Thus, an output-coded state assignment improves speed as well as packaging efficiency. In T-bird tail lights, turning on the emergency flashers 10 ns sooner doesn't matter, but in a high-performance digital system, an extra 10 ns of delay could make the difference between a maximum system clock rate of 100 MHz and a maximum of only 50 MHz.

If a design *must* be split into two or more PLDs, the best split in terms of both design simplicity and packaging efficiency is often the one shown in Figure XSabl-2. A single sequential PLD is used to create the required next-state behavior, and combinational PLDs are used to create both Mealy- and Moore-type outputs from the state variables and inputs.

XSabl.2 A Few Simple Machines

In Section 7.4 we designed several simple state machines using traditional methods. We presented an ABEL- and PLD-based design for the first of these in Table 7-22 on page 615, and then we showed how we could make the machine's operation more understandable by making better use of ABEL features in Table 7-24 on page 618.

RELIEF FOR A SPLITTING HEADACHE

Modern software tools for PLD-based system design can eliminate some of the trial and error that might otherwise be associated with fitting a design into a set of PLDs. To use this capability, the designer enters the equations and state-machine descriptions for the design, without necessarily specifying the actual devices and pinouts that should be used to realize the design. A software tool called a *partitioner* attempts to split the design into the smallest possible number of devices from a given family, while minimizing the number of pins used for interdevice communication. Partitioning can be fully automatic, partially user controlled, or fully user controlled.

Larger devices—CPLDs and FPGAs—often have internal, architectural constraints that may create headaches in the absence of expert software assistance. It appears to the designer, based on input, output, and total combinational logic and flip-flop requirements, that a design will fit in a single CPLD or FPGA. However, the design must still be split among multiple PLDs or logic blocks inside the larger device, where each block has only limited functionality.

For example, an output that requires many product terms may have to steal some from physically adjacent outputs. This may in turn affect whether adjacent pins can be used as inputs or outputs, and how many product terms are available to them. It may also affect the ability to interconnect signals between nearby blocks and the worst-case delay of these signals. All of these constraints can be tracked by *fitter* software that uses a heuristic approach to find the best split of functions among the blocks within a single CPLD or FPGA.

In many design environments, the partitioner and fitter software work together and interactively with the designer to find an acceptable realization using a set of PLDs, CPLDs, and FPGAs.

Our second example was a “1s-counting machine” with the following specification:

> Design a clocked synchronous state machine with two inputs, X and Y, and one output, Z. The output should be 1 if the number of 1 inputs on X and Y since reset is a multiple of 4, and 0 otherwise.

We developed a state table for this machine in Table 7-9 on page 567. However, we can express the machine’s function much more easily in ABEL, as shown in Table XSabl-1. Notice that for this machine we were better off not using ABEL’s “state diagram” syntax. We could express the machine’s function more naturally using a nested when statement and the built-in addition operation. The first `when` clause forces the machine to its initial state and count of 0 upon reset, and the succeeding clauses increase the count by 2, 1, or 0 as required when the machine is not reset. Note that ABEL “throws away” the carry bit in addition, which is equivalent to performing addition modulo-4. The machine easily fits into a GAL16V8 device. It has four states, because there are two flip-flops in its realization.

Table XSabl-1
ABEL program for ones-counting state machine.

```
module onesctsm
title 'Ones-counting State Machine'
ONESCTSM device 'P16V8R';

" Inputs and outputs
CLOCK, RESET, X, Y     pin 1, 2, 3, 4;
Z                      pin 13 istype 'com';
COUNT1..COUNT0         pin 14, 15 istype 'reg';

" Sets
COUNT = [COUNT1..COUNT0];

equations
COUNT.CLK = CLOCK;
when RESET then COUNT := 0;
else when X & Y then COUNT := COUNT + 2;
else when X # Y then COUNT := COUNT + 1;
else COUNT := COUNT;

Z = (COUNT == 0);

end onesctsm
```

Another example from Section 7.4 is a combination-lock state machine (below we omit the HINT output in the original specification):

> Design a clocked synchronous state machine with one input, X, and one output, UNLK. The UNLK output should be 1 if and only if X is 0 and the sequence of inputs received on X at the preceding seven clock ticks was 0110111.

We developed a state table for this machine in Table 7-11 on page 568, and we wrote an equivalent ABEL program in Table 7-28 on page 621. However, once again we can take a different approach that is easier to understand. For this example, we note that the output of the machine at any time is completely determined by its inputs over the preceding eight clock ticks. Thus, we can use a so-called "finite-memory" approach to design this machine, where we explicitly keep track of the past seven inputs and then form the output as a combinational function of these inputs.

RESETTING BAD HABITS

In Chapter 7 we started the bad habit of designing state tables without including an explicit reset input. There was a reason for this—each additional input potentially doubles the amount of work we would have had to do to synthesize the state machine using manual methods, and there was little to be gained pedagogically.

Now that we're doing language-based state-machine design using automated tools, we should get into the habit of always providing an explicit reset input that sends the machine back to a known state. It's a requirement in real designs!

```
module comblckf
title 'Combination-Lock State Machine'
"COMBLCKF device 'P16V8R';

" Input and output pins
CLOCK, RESET, X             pin 1, 2, 3;
X1..X7                      pin 12..18 istype 'reg';
UNLK                        pin 19;

" Sets
XHISTORY = [X7..X1];
SHIFTX   =  [X6..X1, X];

equations
XHISTORY.CLK = CLOCK;
XHISTORY := !RESET & SHIFTX;
UNLK = !RESET & (X == 0) & (XHISTORY == [0,1,1,0,1,1,1]);
end comblckf
```

Table XSabl-2 Finite-memory program for a combination-lock state machine.

The ABEL program in Table XSabl-2 uses the finite-memory approach. It is written using sets to make modifications easy, for example, changing the combination. However, note that the HINT output would be just as difficult to provide in this version of the machine as in the original (see Exercise XSabl.4).

FINITE-MEMORY DESIGN

The finite-memory design approach to state-machine design can be generalized. Such a machine's outputs are completely determined by its current input and outputs during the previous *n* clock ticks, where *n* is a finite, bounded integer. Any machine that can be realized as shown in Figure XSabl-3 is a finite-memory machine. Note that a finite-*state* machine need not be a finite-*memory* machine. For example, the ones-counting machine in Table XSabl-1 has only four states but is not a finite-memory machine; its output depends on every value of X and Y since reset.

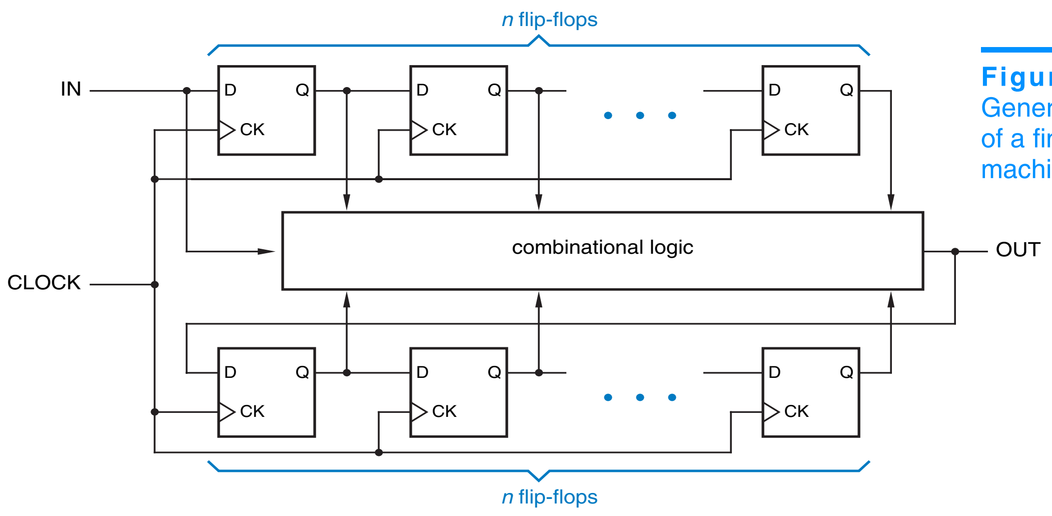


Figure XSabl-3 General structure of a finite-memory machine.

XSabl.3 T-Bird Tail Lights

We described and designed a "T-bird tail-lights" state machine in Section 7.5. Table XSabl-3 is an equivalent ABEL "state diagram" for the T-bird tail-lights machine. There is a close correspondence between this program and the state diagram of Figure 7-58 on page 575 using the state assignment of Table 7-13 on page 574. Except for the added RESET input, the program produces exactly the same reduced equations as the explicit transition equations resulting from the transition list, which we worked out by hand in Section 7.6 on page 577.

The program in Table XSabl-3 handles only the state variables of the tail-lights machine. The output logic requires six combinational outputs, but only

Table XSabl-3 ABEL program for the T-bird tail-lights machine.

```
module tbirdsd
title 'State Machine for T-Bird Tail Lights'
TBIRDSD device 'P16V8R';

" Input and output pins
CLOCK, LEFT, RIGHT, HAZ, RESET   pin 1, 2, 3, 4, 5;
Q0, Q1, Q2                       pin 14, 15, 16 istype 'reg';

" Definitions
QSTATE = [Q2,Q1,Q0];            " State variables
IDLE   = [ 0, 0, 0];            " States
L1     = [ 0, 0, 1];
L2     = [ 0, 1, 1];
L3     = [ 0, 1, 0];
R1     = [ 1, 0, 1];
R2     = [ 1, 1, 1];
R3     = [ 1, 1, 0];
LR3    = [ 1, 0, 0];

equations
QSTATE.CLK = CLOCK;

state_diagram QSTATE
state IDLE: if RESET then IDLE
            else if (HAZ # LEFT & RIGHT) then LR3
            else if LEFT then L1 else if RIGHT then R1
            else IDLE;
state L1:   if RESET then IDLE else if HAZ then LR3 else L2;
state L2:   if RESET then IDLE else if HAZ then LR3 else L3;
state L3:   goto IDLE;
state R1:   if RESET then IDLE else if HAZ then LR3 else R2;
state R2:   if RESET then IDLE else if HAZ then LR3 else R3;
state R3:   goto IDLE;
state LR3:  goto IDLE;

end tbirdsd
```

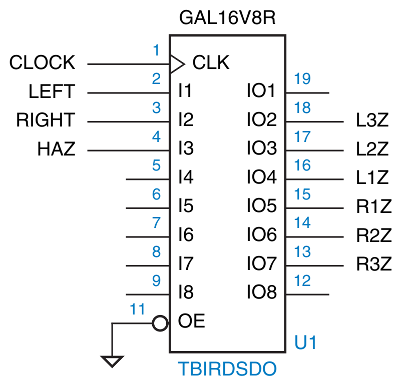


Figure XSabl-4 A single-PLD design for T-bird tail lights.

five more outputs are available in the 16V8 specified in the program. A second PLD could be used to decode the states, using the kind of partitioning that we showed in Figure XSabl-2. Alternatively, a larger PLD, such as the 22V10, could provide enough outputs for a single-PLD design.

An even better approach is to recognize that the output values of the tail-lights machine are different in each state, so we can also use an output-coded state assignment. This requires only six registered outputs and no combinational outputs of a 16V8, as shown in Figure XSabl-4. Only the device, pin, and state definitions in the previous ABEL program must be changed, as shown in Table XSabl-4. The six resulting excitation equations each use four product terms.

Table XSabl-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];
```

XSabl.4 The Guessing Game

A “guessing game” machine was defined in Section 7.7.1 starting on page 580, 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 XSabl-5 on the next page. 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 cranks out minimized equations with the number of product terms shown in Table XSabl-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 XSabl.6).

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 XSabl-5. The new assignment is shown in Table XSabl-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.

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 XSabl-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 XSabl-8.

Table XSabl-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
```

```
 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
```

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

```
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];
...
```

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

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

In the don't-care version, we are assuming that the state bits never take on any combination of values other than the ones we originally defined in Table XSabl-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

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 XSabl-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 XSabl-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.

can set these bits to "don't care" in `S1`'s definition in Table XSabl-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 XSabl-8 use three product terms for `L1`, one each for `L2..L4`, and only seven for `ERR`. So the change was worthwhile. However, we must remember that the new machine is different from the one in Table XSabl-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

RESETTING EXPECTATIONS

Reading the guessing-game program in Table XSabl-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 XSabl-7 or XSabl-8, don't take anything for granted.

Referring to the box at the top of this 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.

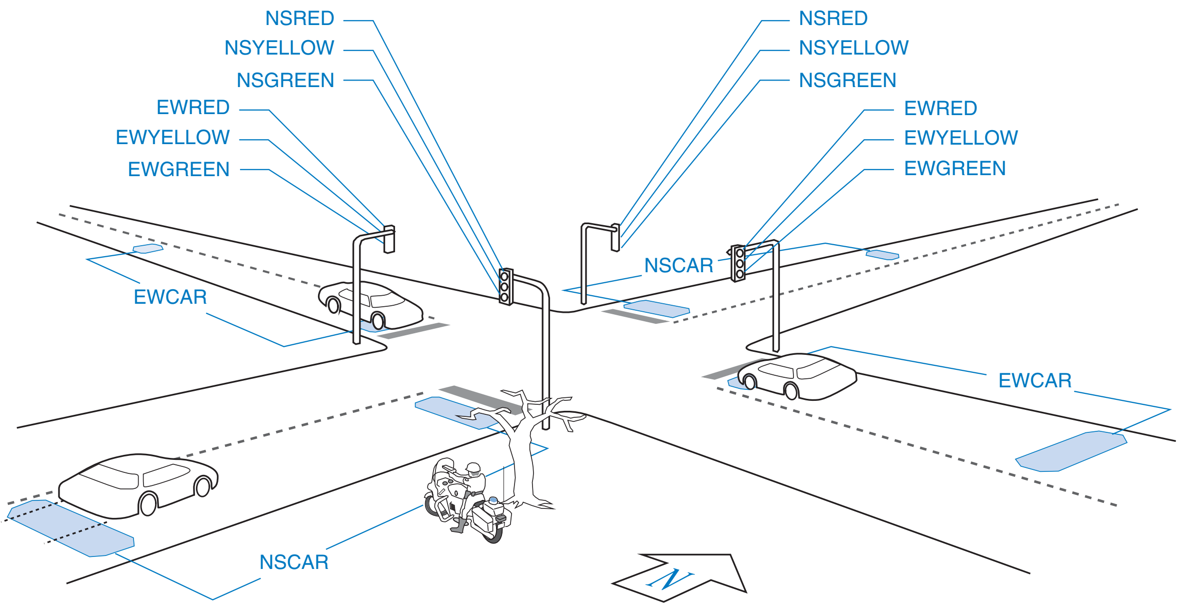
states, so the state machine will always go to the state coded 00000 (`S0K`) from unspecified states. In the new machine, “unspecified” states aren’t really unspecified; for example, the state coded 11111 actually matches five coded states, `S1`–`S4` and `SERR`. The next state will actually be the “OR” 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.

XSabl.5 Reinventing Traffic-Light Controllers

Our next 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 XSabl-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.

Figure XSabl-5 Traffic sensors and signals at an intersection in Sunnyvale, California.



The state machine has seven outputs:

NSRED, NSYELLOW, NSGREEN	Control the north-south lights.
EWRED, EWYELLOW, EWGREEN	Control the east-west lights.
TMRESET	When asserted, resets the timer and negates TMSHORT and TMLONG. The timer starts timing when TMRESET is negated.

A typical, municipally approved algorithm for controlling the traffic lights is embedded in the ABEL program of Table XSabl-9. This algorithm produces two frequently seen behaviors of "smart" traffic lights. At night, when traffic is light, it holds a car stopped at the light for up to five minutes, unless a car approaches on the cross street, in which case it stops the cross traffic and lets the waiting car go. (The "early warning" sensor is far enough back to change the lights before the approaching car reaches the intersection.) During the day, when traffic is heavy and there are always cars waiting in both directions, it cycles the lights every five seconds, thus minimizing the utilization of the intersection and maximizing everyone's waiting time, thereby creating a public outcry for more taxes to fix the problem.

The equations for the TMRESET output are worth noting. This output is asserted during the "double-red" states, NSDELAY and EWDELAY, to reset the timer in preparation for the next green cycle. The desired output signal could be generated on a combinational output pin by decoding these two states, but we have chosen instead to generate it on a registered output pin by decoding the *predecessors* of these two states.

The ABEL program in Table XSabl-9 defines only the state variables and one registered Moore output for the traffic controller. Six more Moore outputs are needed for the lights, more than remain on the 16V8. Therefore, a separate

Figure XSabl-6 Sunnyvale traffic-light controller using two PLDs.

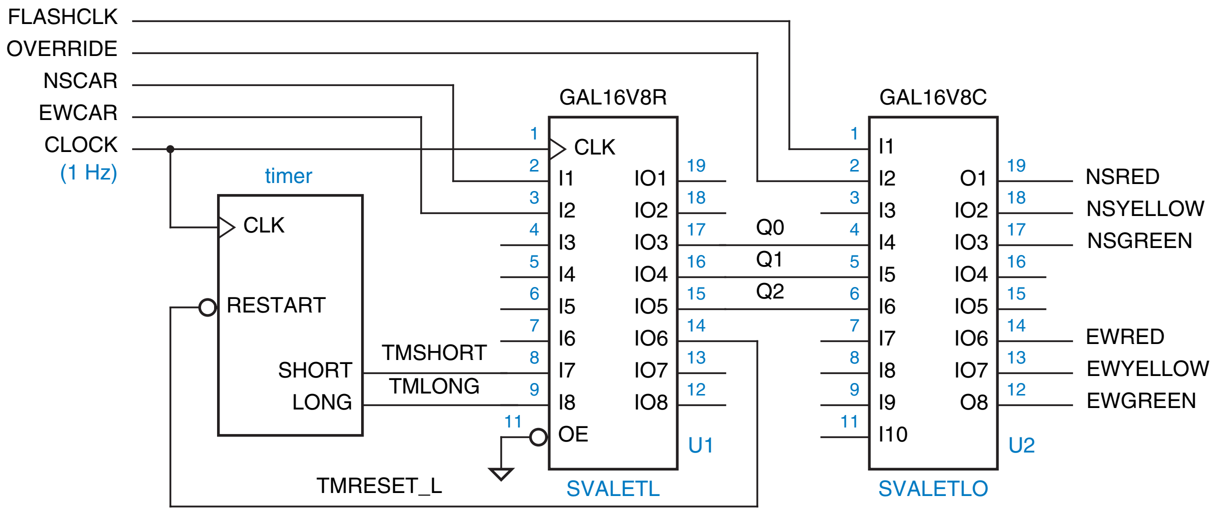


Table XSabl-9 Sunnyvale traffic-lights program.

```
module svaletl
title 'State Machine for Sunnyvale, CA, Traffic Lights'
SVALETL device 'P16V8R';

" Input and output pins
CLOCK, !OE                        pin 1, 11;
NSCAR, EWCAR, TMSHORT, TMLONG     pin 2, 3, 8, 9;
Q0, Q1, Q2, TMRESET_L             pin 17, 16, 15, 14 istype 'reg';

" Definitions
LSTATE  = [Q2,Q1,Q0];            " State variables
NSGO    = [ 0, 0, 0];            " States
NSWAIT  = [ 0, 0, 1];
NSWAIT2 = [ 0, 1, 1];
NSDELAY = [ 0, 1, 0];
EWGO    = [ 1, 1, 0];
EWWAIT  = [ 1, 1, 1];
EWWAIT2 = [ 1, 0, 1];
EWDELAY = [ 1, 0, 0];

state_diagram LSTATE
state NSGO:                      " North-south green
   if (!TMSHORT) then NSGO       " Minimum green is 5 seconds.
   else if (TMLONG) then NSWAIT  " Maximum green is 5 minutes.
   else if (EWCAR & !NSCAR)      " If E-W car is waiting and no one
          then NSGO              "  is coming N-S, make E-W wait!
   else if (EWCAR & NSCAR)       " Cars coming in both directions?
          then NSWAIT            " Thrash!
   else if (!NSCAR)              " Nobody coming N-S and not timed out?
          then NSGO              " Keep N-S green.
   else NSWAIT;                  " Else let E-W have it.

state NSWAIT:  goto NSWAIT2;     " Yellow light is on for two ticks for safety.
state NSWAIT2: goto NSDELAY;     " (Drivers go 70 mph to catch this turkey green!)
state NSDELAY: goto EWGO;        " Red in both directions for added safety.

state EWGO: " East-west green; states defined analogous to N-S
   if (!TMSHORT) then EWGO
   else if (TMLONG) then EWWAIT
   else if (NSCAR & !EWCAR) then EWGO
   else if (NSCAR & EWCAR) then EWWAIT
   else if (!EWCAR) then EWGO else EWWAIT;

state EWWAIT: goto EWWAIT2;
state EWWAIT2: goto EWDELAY;
state EWDELAY: goto NSGO;

equations

LSTATE.CLK = CLOCK;  TMRESET_L.CLK = CLOCK;
!TMRESET_L := (LSTATE == NSWAIT2)  " Reset the timer when going into
            + (LSTATE == EWWAIT2); "   state NSDELAY or state EWDELAY.
end svaletl
```

Table XSabl-10 Output logic for Sunnyvale traffic lights.

```
module svaletlo
title 'Output logic for Sunnyvale, CA, Traffic Lights'
SVALETLO device 'P16V8C';

" Input pins
FLASHCLK, OVERRIDE, Q0, Q1, Q2   pin 1, 2, 4, 5, 6;

" Output pins
NSRED, NSYELLOW, NSGREEN         pin 19, 18, 17 istype 'com';
EWRED, EWYELLOW, EWGREEN         pin 14, 13, 12 istype 'com';

" Definitions (same as in state machine SVALETL)
...

equations

NSRED = !OVERRIDE & (LSTATE != NSGO) & (LSTATE != NSWAIT) & (LSTATE != NSWAIT2)
      # OVERRIDE & FLASHCLK;
NSYELLOW = !OVERRIDE & ((LSTATE == NSWAIT) # (LSTATE == NSWAIT2));
NSGREEN  = !OVERRIDE & (LSTATE == NSGO);

EWRED = !OVERRIDE & (LSTATE != EWGO) & (LSTATE != EWWAIT) & (LSTATE != EWWAIT2)
      # OVERRIDE & FLASHCLK;
EWYELLOW = !OVERRIDE & ((LSTATE == EWWAIT) # (LSTATE == EWWAIT2));
EWGREEN  = !OVERRIDE & (LSTATE == EWGO);

end svaletlo
```

combinational PLD is used for these outputs, yielding the complete design shown in Figure XSabl-6 on the preceding page. An ABEL program for the output PLD is given in Table XSabl-10. We've taken this opportunity to add an OVERRIDE input to the controller. This input may be asserted by the police to disable the controller and put the signals into a flashing-red mode (at a rate determined by FLASHCLK), allowing them to manually clear up the traffic snarls created by this wonderful invention.

A traffic-light state machine including output logic can be built in a single 16V8, shown in Figure XSabl-7, if we choose an output-coded state assignment. Only the definitions in the original program of Table XSabl-9 must be changed, as shown in Table XSabl-11. This PLD does not include the OVERRIDE input and mode, which is left as an exercise (XSabl.9).

Table XSabl-11 Definitions for Sunnyvale traffic-lights machine with output-coded state assignment.

```
module svaletlb
title 'Output-Coded State Machine for Sunnyvale Traffic Lights'
SVALETLB device 'P16V8R';

" Input and output pins
CLOCK, !OE                           pin 1, 11;
NSCAR, EWCAR, TMSHORT, TMLONG        pin 2, 3, 8, 9;
NSRED, NSYELLOW, NSGREEN             pin 19, 18, 17 istype 'reg';
EWRED, EWYELLOW, EWGREEN             pin 16, 15, 14 istype 'reg';
TMRESET_L, XTRA                      pin 13, 12 istype 'reg';

" Definitions
LSTATE   = [NSRED,NSYELLOW,NSGREEN,EWRED,EWYELLOW,EWGREEN,XTRA]; " State vars
NSGO     = [    0,        0,       1,     1,        0,       0,    0]; " States
NSWAIT   = [    0,        1,       0,     1,        0,       0,    0];
NSWAIT2  = [    0,        1,       0,     1,        0,       0,    1];
NSDELAY  = [    1,        0,       0,     1,        0,       0,    0];
EWGO     = [    1,        0,       0,     0,        0,       1,    0];
EWWAIT   = [    1,        0,       0,     0,        1,       0,    0];
EWWAIT2  = [    1,        0,       0,     0,        1,       0,    1];
EWDELAY  = [    1,        0,       0,     1,        0,       0,    1];
```

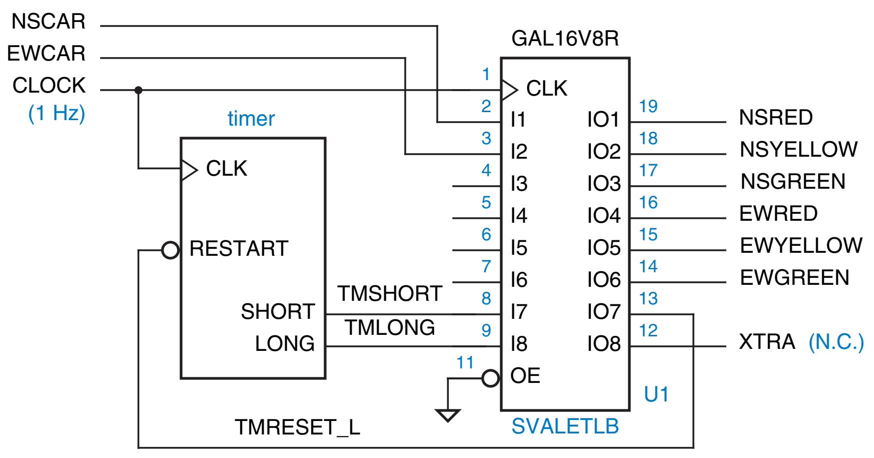


Figure XSabl-7 Traffic-light state machine using output-coded state assignment in a single PLD.

XSabl.6 A Synchronous System Design Example

This subsection presents a representative example of a synchronous system design in ABEL. The example is a *shift-and-add multiplier* for unsigned integers using the algorithm of Section 2.8. The complete design of its data and control units can be expressed using equations and a "state diagram."

shift-and-add multiplier

Figure XSabl-8 illustrates data-unit registers and functions that are used to perform an 8-bit multiplication:

MPY/LPROD	A shift register that initially stores the multiplier, and accumulates the low-order bits of the product as the algorithm is executed.
HPROD	A register that is initially cleared, and accumulates the high-order bits of the product as the algorithm is executed.
MCND	A register that stores the multiplicand throughout the algorithm.
F	A combinational function equal to the 9-bit sum of HPROD and MCND if the low-order bit of MPY/LPROD is 1, and equal to HPROD (extended to 9 bits) otherwise.

The MPY/LPROD shift register serves a dual purpose, holding both yet-to-be-tested multiplier bits (on the right) and unchanging product bits (on the left) as the algorithm is executed. At each step it shifts right one bit, discarding the multiplier bit that was just tested, moving the next multiplier bit to be tested to the rightmost position, and loading into the leftmost position one more product bit that will not change for the rest of the algorithm.

The ABEL realization of the multiplier system will have the following inputs and outputs:

CLOCK	A single clock signal for the state machine and registers.
RESET	A reset signal to clear the registers and put the state machine into its starting state before the system begins operation.

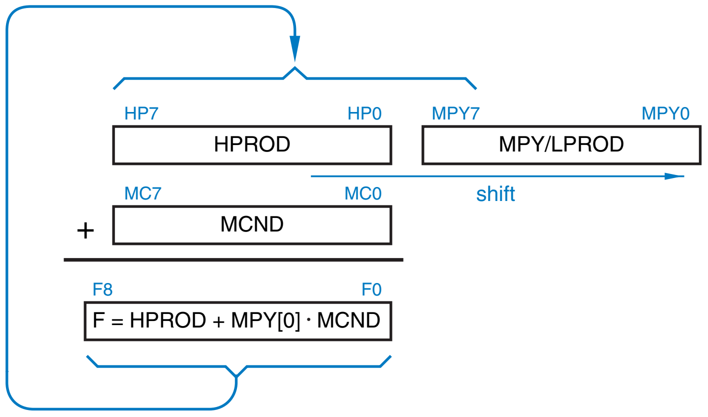


Figure XSabl-8 Registers and functions used by the shift-and-add multiplication algorithm.

INP[7..0]	An 8-bit input bus for the multiplicand and multiplier to be loaded into registers in two clock ticks at the beginning of a multiplication.
PROD[15..0]	A 16-bit output bus that will contain the product at the end of a multiplication.
START	An input that is asserted prior to a rising clock edge to begin a multiplication. START must be negated before asserting it again will start a new multiplication.
DONE	An output that is asserted when the multiplication is done and PROD[15..0] is valid.

A timing diagram for the multiplier system is shown in Figure XSabl-9. The first six waveforms show the input/output behavior and how a multiplication takes place in 10 or more clock periods as described below:

1. START is asserted. The multiplicand is placed on the INP bus and is loaded into the MCND register at the end of this clock period.
2. The multiplier is placed on the INP bus and is loaded into the MPY register at the end of the clock period.

3–10. One shift-and-add step is performed at each of the next eight clock ticks. Immediately following the eighth clock tick, DONE should be asserted and the 16-bit product should be available on PROD[15..0]. A new multiplication can also be started during this clock tick, but *may* start later.

Our strategy for controlling the multiplication is based on a decomposed state machine, as introduced in Section 7.8. A top-level state machine controls the overall operation, while a counter counts the eight shift-and-add steps.

Figure XSabl-9 Timing diagram for multiplier system.

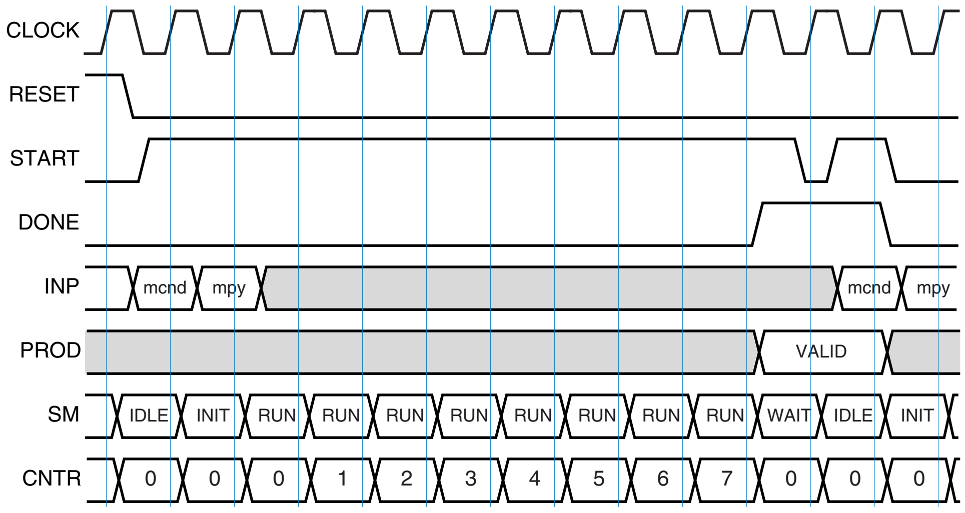


Table XSabl-12 ABEL module for an 8x8 shift-and-add multiplier.

```
module MPY8X8
title '8x8 shift-and-add multiplier'

" Inputs
CLOCK, RESET, START, INP7..INP0  pin;

" Outputs
PROD15..PROD0             pin istype 'com';
DONE                      pin istype 'reg';

" Locals
CNTR2..CNTR0              pin istype 'reg'; " Step counter
SM1..SM0                  pin istype 'reg'; " State machine
MPY7..MPY0, MCND7..MCND0  pin istype 'reg'; " MPY/LPROD and MCND registers
HPROD7..HPROD0            pin istype 'reg'; " HPROD register
F8..F0                    pin istype 'com'; " Output of mux/adder (incl. carry)

" Constant expressions
INP   = [INP7..INP0];
PROD  = [PROD15..PROD0];
CNTR  = [CNTR2..CNTR0];
MPY   = [MPY7..MPY0];
MCND  = [MCND7..MCND0];
HPROD = [HPROD7..HPROD0];
F     = [F8..F0];

" State-machine states
SM   = [SM1, SM0];
IDLE = [  0,   0];
INIT = [  0,   1];
RUN  = [  1,   1];
WAIT = [  1,   0];

state_diagram SM

state IDLE: if RESET then IDLE
            else if START then INIT
            else IDLE;

state INIT: if RESET then IDLE
            else RUN;

state RUN:  if RESET then IDLE
            else if ((CNTR==7) & START) then WAIT
            else if ((CNTR==7) & !START) then IDLE
            else RUN;

state WAIT: if RESET then IDLE
            else if !START then IDLE
            else WAIT;
```

Table XSabl-12 (continued)

```
equations
" Clocks
CNTR.CLK = CLOCK; SM.CLK = CLOCK; MPY.CLK = CLOCK;
MCND.CLK = CLOCK; HPROD.CLK = CLOCK; DONE.CLK = CLOCK;

when (RESET) then CNTR := 0;
else when (SM==RUN) then CNTR := CNTR + 1;
else CNTR := 0;

when (RESET) then MCND := 0;
else when ((SM==IDLE) & START) then MCND := INP;
else MCND := MCND;

when (RESET) then MPY := 0;
else when (SM==INIT) then MPY := INP;
else when (SM==RUN) then MPY := [F0, MPY7..MPY1];
else MPY := MPY;

when (RESET) then HPROD := 0;
else when (SM==IDLE) then HPROD := 0;
else when (SM==RUN) then HPROD := [F8..F1];
else HPROD := HPROD;

@CARRY 2;
when (MPY0) then F = [0,HPROD7..HPROD0] + [0,MCND7..MCND0];
else F = [0,HPROD7..HPROD0];

PROD = [HPROD7..HPROD0,MPY7..MPY0];

DONE := !RESET & ( ((SM==RUN) & (CNTR==7)) # (SM==WAIT) );

end MPY8X8
```

Table XSabl-12 is an ABEL program for the complete multiplier system. A state machine SM with four states is used for the top-level control. Individual equations are written for 3-bit counter CNTR, which allows the state machine to stay in the RUN state for eight clock ticks. The state-machine and counter behaviors are shown in the last two waveforms in Figure XSabl-9.

Individual equations are also written for internal registers MPY, MCND, and HPROD, for the internal adder/multiplexer function F (combinational), and for outputs PROD (combinational) and DONE (registered). You should study the code and the timing diagram to get a feel for how this all works together.

The code for the adder/multiplexer function F has two noteworthy aspects. The @CARRY compiler directive is used to limit the size of the adder equations, forcing a ripple carry per 2-bit group. And the addends are explicitly padded so that F is a 9-bit result.

Exercises

XSabl.1 Design a clocked synchronous state machine that checks parity on a serial byte-data line with timing similar to Figure XSbb-3 in Section XSbb.1. The circuit should have three inputs, RESET, SYNC, and DATA, in addition to CLOCK, and one Moore-type output, ERROR. The ERROR output should be asserted if any DATA byte received since reset had odd parity. Using ABEL, devise a state machine that does the job using no more than four states. Include comments to describe each state's meaning and use. Target your design to a GAL16V8 PLD. Write ABEL test vectors that test your design for proper operation by applying three bytes in succession with even, odd, and even parity.

XSabl.2 Enhance the state machine in the preceding exercise by also asserting ERROR if SYNC was not asserted within eight clock ticks after the machine starts up after reset, or if any SYNC pulses fail to be exactly eight clock ticks apart. Try to fit the enhanced design in a GAL16V8 PLD, and add test vectors to check for proper machine operation in the newly defined conditions.

XSabl.3 Using ABEL, design a clocked synchronous state machine with two inputs, INIT and X, and one Moore-type output Z. As long as INIT is asserted, Z is continuously 0. Once INIT is negated, Z should remain 0 until X has been 0 for two successive ticks and 1 for two successive ticks, regardless of the order of occurrence. Then Z should go to 1 and remain 1 until INIT is asserted again. Target your design to a GAL16V8 PLD and write ABEL test vectors that test your design for proper operation. (*Hint:* No more than ten states are required.)

XSabl.4 Modify the ABEL program of Table XSabl-2 to include the HINT output from the original state-machine specification in Section 7.4.

XSabl.5 Redesign the T-bird tail-lights machine of Section XSabl.3 to include parking-light and brake-light functions. When the BRAKE input is asserted, all of the lights should go on immediately, and stay on until BRAKE is negated, independent of any other function. When the PARK input is asserted, each lamp is turned on at 50% brightness at all times when it would otherwise be off. This is achieved by driving the lamp with a 100-Hz signal DIMCLK with a 50% duty cycle. Draw a logic diagram for the circuit using one or two PLDs, write an ABEL program for each PLD, and write a short description of how your system works.

XSabl.6 Find a 3-bit state assignment for the guessing-game state machine in Table XSabl-5 that reduces the maximum number of product terms per output to 7. Can you do even better?

XSabl.7 The operation of the guessing game in Section XSabl.4 is very predictable; it's easy for a player to learn the rate at which the lights change and always hit the button at the right time. The game is more fun if the rate of change is more variable.

Modify the ABEL state machine in Table XSabl-5 so that in states `S1`–`S4`, the machine advances only if a new input, `SEN`, is asserted. (`SEN` is intended to be hooked up to a pseudorandom bit-stream generator.) Both correct and incorrect button pushes should be recognized whether or not SEN is asserted. Determine whether your modified design still fits in a 16V8.

XSabl.8 In connection with the preceding exercise, write an ABEL program for an 8-bit LFSR using a single 16V8, such that one of its outputs can be used as a pseudorandom bit-stream generator. After how many clock ticks does the bit sequence repeat? What is the maximum number of 0s that occur in a row? What is the maximum number of 1s?

XSabl.9 Add an OVERRIDE input to the traffic-lights state machine of Figure XSabl-7, still using just a single 16V8. When OVERRIDE is asserted, the red lights should flash on and off, changing once per second. Write a complete ABEL program for your machine.

XSabl.10 Modify the behavior of the ABEL traffic-light-controller machine in Table XSabl-9 to have more reasonable behavior, the kind you'd like to see for traffic lights in your own home town.

XSabl.11 Simulate the multiplier system of Table XSabl-12, and verify that the product is available on PROD[15..0] and DONE is asserted for exactly one clock period when back-to-back multiplications are performed (one multiplication every 10 clock ticks). Then modify the program so that PROD[15..0] is valid and DONE is asserted for two clock periods in this situation, and one period longer for each tick that the state machine spends in the WAIT state.

XSabl.12 Using ABEL, design a data unit and a control-unit state machine for multiplying 8-bit two's-complement numbers using the algorithm discussed in Section 2.8.

XSabl.13 Using ABEL, design a data unit and control-unit state machine for dividing 8-bit unsigned numbers using the shift-and-subtract algorithm discussed in Section 2.9.