State Machine Design

Objectives

- Introduce the concepts of state machines.
- Discuss the different types of state machines.
- Build counters using D flip-flops and combinational logic.
- Show how a state machine is designed to solve a real problem.

Discussion

Simple State Machines

In the last chapter, you saw that you could build logic circuits that store previous inputs by altering their internal state. You also used these flip-flops to build a simple counter.

A simple state machine uses one or more flip-flops to store its internal state (Figure 7.1). The pattern of ones and zeros on the Q outputs of the flip-flops are the current state of the state machine. When a rising clock edge occurs, the current state is replaced by the next state. How is the next state determined? An easy way is to build a combinational logic circuit that accepts the current state as its input and computes the next state as its output. The output of the combinational logic is loaded into the flip-flops on the next rising clock edge. What is a good example of such a state machine? A counter! The current value of the count is held in the DFFs, and this value is passed to a combinational circuit that is just a simple incremener. The output of the incremener is clocked into the DFFs on the next rising clock edge.

So we already know everything we need to build state machines:

- We know how to build combinational logic circuits.
- We know how to store state information in D flip-flops.
- We know how to connect the combinational logic and the DFFs.

A state machine that only uses its current state to compute its next state is not very interesting (kind of like a person who only talks to himself or herself). A more useful variation allows inputs from the environment to affect the state transitions (see Figure 7.2). The external inputs interact inside the combina-
An Up/Down Counter

Since we have already done all the design work, we might as well build the 3-state up/down counter. The ABEL code for it is shown in Listing 7.1.

The 2 DFFs for storing the state information, Q1 and Q0, are declared on line 9. The clock input is attached to them on line 13.

Lines 14–31 hold the truth-table for the state machine. Line 15 lists the inputs and outputs for the truth-table logic. Note that Q1 and Q0 act both as inputs and outputs because the current state information is needed to determine the next state. Note also the use of two different types of output-list delimiters. We have already seen the -> delimiter used to denote the start of a list of combinatorial outputs. But now the :: delimiter is added to signal the start of a list of registered outputs. In this case, the registered outputs are the Q1 and Q0 flip-flop outputs. The -> delimiter indicates that the Q1 and Q0 outputs will only change to the values listed in the table when the flip-flops receive a rising clock edge (in this case). That is just what we want for a state machine.

Each of the remaining lines in the truth-table just replicates what we have already seen in Table 7.2.

The VHDL version of the 3-state up/down counter is given in Listing 7.2. On line 23, I define two 2-element signal arrays, one for holding the current state (curr_state) and another for holding the next state (next_state). Within the main body of the ARCHITECTURE section there are 2 processes that make use of these arrays. The first process takes the current state along with the reset and up/down control inputs and computes the value for the next state. For example, if the reset is active then the next state is set to all zeroes on line 37. In another case, if the reset is not active, the current state is 1 (curr_state=01) and the counter is counting up (up=1), then line 43 sets the next state to be 2 (next_state=10). Note that this process is entirely combinational: the next state is not latched by this process. If there were other combinational outputs that were a function of the current state, we could place their equations into the IF...THEN...ELSE statements of this process. (In this example the outputs are actually placed outside the process on lines 29 and 30 since they are simple copies of the current state array elements.)

The second process on lines 55 through 60 just copies the state value held in the next_state array into the curr_state array on a rising edge of the clock. Thus, this process moves the state machine from the current state to the next state. It is in this process where the actual state latches are implied. Also, the next state is affected by the reset input in the first process, but does not take effect until the second process updates the current state. This makes the reset synchronous.
Listing 7.1  ABEL code for a 3-state up/down counter.

```
001- MODULE CNT3
002- TITLE '3-state counter with reset'
003-
004- DECLARATIONS
005- C          PIN;        "clock input"
006- RESET      PIN;        "reset input"
007- UP         PIN;        "count direction input"
008- OUTP1..OUTP0 PIN ISTYPE 'COM'; "outputs"
009- Q1..Q0     PIN ISTYPE 'REG'; "two DFFs for counter"
010-
011- EQUATIONS
012-
013- [Q1..Q0].CLK = C;  'connect input clock to counter DFFs'
014- TRUTH_TARI F
015-  (((Q1,Q0,RESET,UP) => [Q1,Q0] => [OUTP1,OUTP0])
016-  [ 0, 0, 0, 0 ] => [ 1, 0 ] => [ 0, 0 ];  'down : 0->2
017-  [ 0, 0, 1, 1 ] => [ 0, 1 ] => [ 0, 0 ];  'up : 0->1
018-  [ 0, 0, 1, 0 ] => [ 0, 0 ] => [ 0, 0 ];  'reset: 0->0
019-  [ 0, 1, 0, 0 ] => [ 0, 0 ] => [ 0, 1 ];  'down : 1->0
020-  [ 0, 1, 0, 1 ] => [ 0, 0 ] => [ 0, 1 ];  'up : 1->2
021-  [ 0, 1, 1, 0 ] => [ 0, 0 ] => [ 0, 1 ];  'reset: 1->0
022-  [ 0, 1, 1, 1 ] => [ 0, 0 ] => [ 0, 1 ];  'reset: 1->0
023-  [ 1, 0, 0, 0 ] => [ 1, 0 ] => [ 1, 0 ];  'down : 2->1
024-  [ 1, 0, 0, 1 ] => [ 1, 0 ] => [ 1, 0 ];  'up : 2->0
025-  [ 1, 0, 1, 0 ] => [ 1, 0 ] => [ 1, 0 ];  'error->0
026-  [ 1, 0, 1, 1 ] => [ 1, 0 ] => [ 1, 0 ];  'error->0
027-  [ 1, 1, 0, 0 ] => [ 1, 0 ] => [ 0, 0 ];  'error->0
028-  [ 1, 1, 0, 1 ] => [ 1, 0 ] => [ 0, 0 ];  'error->0
029-  [ 1, 1, 1, 0 ] => [ 0, 0 ] => [ 0, 0 ];  'error->0
030-  [ 1, 1, 1, 1 ] => [ 0, 0 ] => [ 0, 0 ];  'error->0
031-
032-
033- END CNT3
```

The VHDL code in Listing 7.2 also has special modifications so it will run on the XS400 Board. When targeting an XC4000 FPGA, the Foundation Series synthesis software tries to use special clock buffers to drive clock signals to the flip-flops. These buffers minimize the skew between the times at which the clock edges reach the individual flip-flops. The Foundation Series Implementation tools also like to have the clock input come into the XC4000 FPGA through special pins which connect directly to these clock buffers. But for our example designs we use XSPORT to drive clock signals through the parallel port interface, and none of these pins directly connects to these special clock input pins.
So we have to explicitly describe the connection of the clock input to one of the clock buffers. This is done on lines 26 and 27. On line 26, the main clock input is connected to an input buffer (IBUF) module. Then the output of the IBUF module is connected to the input of a general-purpose clock buffer (BUFG) on line 27. The output of the BUFG, buf_c, becomes the clock that drives the state updating process. (The IBUF and BUFG interfaces are described by the COMPONENT statements on lines 18–21.) You must use this clock input buffer circuitry when you are targeting the XC4000 FPGAs. For the XS95 Board and the XC9500 CPLDs, you can remove these buffers and use the clock input directly.

The simulated waveforms for the up/down counter are shown in Figure 7.10. The RESET input is activated from 0–10 ns and resets the [Q1..Q0] state vector to 0. When RESET is released, the counter begins to count down since UP=0. The counter progresses in the sequence 0, 2, 1, 0, and rolls-over back to 2 over the period 10–50 ns. Then we raise the direction control so UP=1 at 50 ns. This makes the counter increment over the 50–100 ns period. Finally, we raise RESET again at 100 ns. Since it is a synchronous reset, the counter moves from state 1 to 0 only at the next rising clock edge at 106 ns. The state machine seems to be doing what we want.

A constraint file for the CNT3 design should assign the inputs and outputs like this for the XS40 Board:

```plaintext
NET c LOC=P44;  # B0 argument of XSPORT
NET reset LOC=P45;  # B1 argument of XSPORT
NET up LOC=P46;  # B2 argument of XSPORT
NET q0 LOC=P25;  # S0 LED segment
NET q1 LOC=P26;  # S1 LED segment
NET outp0 LOC=P24;  # S2 LED segment
NET outp1 LOC=P20;  # S3 LED segment
```

Or use the following constraints for the XS95 Board:

```plaintext
NET c LOC=P44;  # R0 argument of XSPORT
NET reset LOC=P47;  # B1 argument of XSPORT
NET up LOC=P48;  # B2 argument of XSPORT
NET q0 LOC=P21;  # S0 LED segment
NET q1 LOC=P23;  # S1 LED segment
NET outp0 LOC=P19;  # S2 LED segment
NET outp1 LOC=P17;  # S3 LED segment
```

The clock input must be assigned to either B0 or B1. These inputs pass through a Schmitt-trigger buffer on the XS40 or XS95 Board that removes any spurious glitches caused by the slow transitions of the PC parallel port signals. Using any of the B2 B7 bits may cause the counter to decrement erratically. This will be true for any sequential design we do.
Listing 7.2  VHDL code for a 3-state up/down counter.

001- -- 3-state up/down counter
002-
003- LIBRARY IEEE;
004- USE IEEE.std_logic_1164.ALL;
005-
006- ENTITY cnt3 IS
007- PORT
008- ( c: IN STD_LOGIC; -- clock input
009- reset: IN STD_LOGIC; -- active-high reset
010- up: IN STD_LOGIC; -- up/down control input
011- outp0, outp1: OUT STD_LOGIC; -- outputs
012- q0,q1: OUT STD_LOGIC -- state outputs
013- );
014- END cnt3;
015-
016- ARCHITECTURE cnt3_arch OF cnt3 IS
017- COMPONENT IBUF PORT(I: IN STD_LOGIC;
018- 0: OUT STD_LOGIC); END COMPONENT;
019- COMPONENT BUFG PORT(I: IN STD_LOGIC;
020- 0: OUT STD_LOGIC); END COMPONENT;
021- SIGNAL in_c, buf_c: STD_LOGIC;
022- SIGNAL curr_state, next_state: STD_LOGIC_VECTOR(1 DOWNTO 0);
023- BEGIN
024- BEGIN
025- u0: IBUF PORT MAP(I=>c, 0=>in_c);-- connect clock to
026- u1: BUFG PORT MAP(I=>in_c, 0=>buf_c);-- clock buffer
027- begin
028- outp0 <= curr_state(0); -- connect outputs to current state
029- outp1 <= curr_state(1);
030- q0 <= curr_state(0); -- connect state outputs to current state
031- q1 <= curr_state(1);
032- PROCESS(reset,up,curr_state) -- compute next state from current
033- BEGIN
034- IF reset='1' THEN
035- next_state <= "00";
036- ELSEIF curr_state='00' THEN
037- IF up='1' THEN next_state <= "01";
038- ELSE next_state <= "10";
039- END IF;
040- ELSIF curr_state='01' THEN
041- END PROCESS;
Listing 7.2  VHDL code for a 3-state up/down counter. (Cont’d.)

043- IF up='1' THEN next_state <= "10";
044- ELSE next_state <= "00";
045- END IF;
046- ELSEIF curr_state="10" THEN
047- IF up='1' THEN next_state <= "00";
048- ELSE next_state <= "01";
049- END IF;
050- END ELSE
051- next_state <= "00";
052- END IF;
053- END PROCESS;
054
055- PROCESS(buf_c,next_state)-- update the current state vector
056- BEGIN
057- IF (buf_c'event AND buf_c='1') THEN
058- curr_state <= next_state;
059- END IF;
060- END PROCESS;
061-
062- END until done;

Figure 7.10  Simulated waveforms for the 3-state up/down counter.

After downloading your circuit to the XS40 or XS95 Board, you can use the following command sequences to move the state machine between its various states. Each set of commands activates a particular input and then sends a
rising clock edge to the state machine. You can read off the current state by observing the S0 and S1 LED segments.

<table>
<thead>
<tr>
<th>Action</th>
<th>XSPORT Command Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset the counter</td>
<td>XSPORT 010</td>
</tr>
<tr>
<td></td>
<td>XSPORT 011</td>
</tr>
<tr>
<td></td>
<td>XSPORT 000</td>
</tr>
<tr>
<td>Increment by one</td>
<td>XSPORT 100</td>
</tr>
<tr>
<td></td>
<td>XSPORT 101</td>
</tr>
<tr>
<td></td>
<td>XSPORT 000</td>
</tr>
<tr>
<td>Decrement by one</td>
<td>XSPORT 000</td>
</tr>
<tr>
<td></td>
<td>XSPORT 001</td>
</tr>
<tr>
<td></td>
<td>XSPORT 000</td>
</tr>
</tbody>
</table>

**A Drink Dispenser Controller**

This example will show you how to design a state machine to solve an actual problem: dispensing soft-drinks.

You have been contracted to design the logic for a soft-drink machine that dispenses two brands of cola, Swill and Krunk. The dispenser should accept nickels and dimes and should allow the buyer to select one of the soft-drinks after he or she has put in 25¢. That is the extent of the requirements.

The first step is to get some idea of the inputs and outputs of the logic circuitry. From the requirements, you know that the logic must receive inputs that indicate whether the buyer has put in a nickel or a dime. You also know that there have to be two inputs for indicating whether the buyer wants to enjoy a can of refreshing Swill or Krunk. Finally, what if the buyer changes his or her mind in the middle of a transaction? There should be an input to return the buyer's money.

What about the outputs from the logic? There should be outputs to control the individual dispensers for Swill and Krunk so that a can is dispensed whenever a high logic level is present. There should also be two outputs to control the return of change: one that causes a nickel to be dropped into the change return slot, and another that drops a dime.

OK, you seem to have an initial idea of what the I/O looks like. The next step is to determine the states of your state machine (i.e., what your logic has to remember). The obvious thing to remember is how much money the buyer has put into the dispenser. This machine dispenses a soft-drink after it receives 25¢ in either nickels or dimes, so the possible states for the machine are that it has received 0¢, 5¢, 10¢, 15¢, 20¢, or 25¢. That is a total of 6 states. At this point, the drink dispenser controller looks like Figure 7.11.