XSvhd: Sequential VHDL 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 VHDL, for targeting to CPLDs, FPGAs, or ASICs. Some of these examples illustrate the decisions that are made when partitioning a design into multiple entities.

As we explained in Section 7.12, the basic VHDL language features that we introduced in Section 5.3, including processes, are just about all that's needed to model sequential-circuit behavior. Unlike ABEL, VHDL does not provide any special language elements for modeling state machines. Instead, most designers use a combination of existing "standard" features—most notably enumerated types and case statements—to write state-machine descriptions. We'll use this method in the examples in this section.

XSvhd.1 T-Bird Tail Lights

We described and designed a "T-bird tail-lights" state machine in Section 7.5. Just for fun, the T-bird's rear end is reproduced below, in Figure XSvhd-1.

Table XSvhd-1 is an equivalent VHDL program for the T-bird tail-lights machine. The state transitions in this machine are defined exactly the same as in the state diagram of Figure 7-58 on page 575. The machine uses an output-coded state assignment, taking advantage of the fact that the tail-light output values are different in each state. In this example, we have integrated the flip-flop and next-state behaviors into a single process.

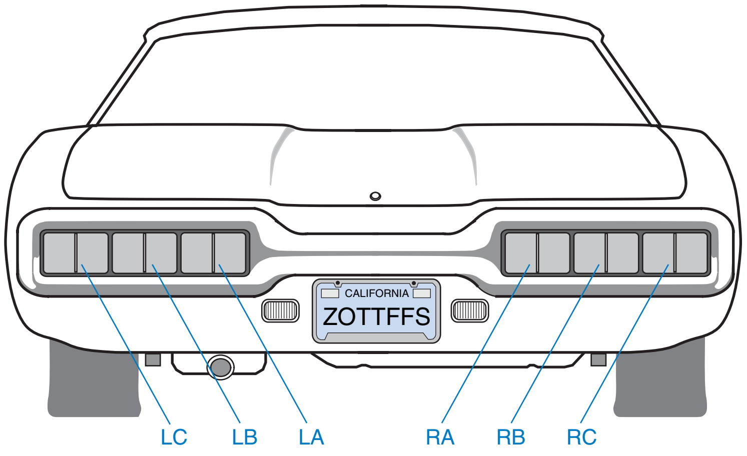

Figure XSvhd-1
T-bird tail lights.

Table XSvhd-1 VHDL program for the T-bird tail-lights machine.

```
entity Vtbird is
  port ( CLOCK, RESET, LEFT, RIGHT, HAZ: in STD_LOGIC;
         LIGHTS: buffer STD_LOGIC_VECTOR (1 to 6) );
end;

architecture Vtbird_arch of Vtbird is
constant IDLE: STD_LOGIC_VECTOR (1 to 6) := "000000";
constant L3  : STD_LOGIC_VECTOR (1 to 6) := "111000";
constant L2  : STD_LOGIC_VECTOR (1 to 6) := "110000";
constant L1  : STD_LOGIC_VECTOR (1 to 6) := "100000";
constant R1  : STD_LOGIC_VECTOR (1 to 6) := "000001";
constant R2  : STD_LOGIC_VECTOR (1 to 6) := "000011";
constant R3  : STD_LOGIC_VECTOR (1 to 6) := "000111";
constant LR3 : STD_LOGIC_VECTOR (1 to 6) := "111111";
begin
  process (CLOCK)
  begin
    if CLOCK'event and CLOCK = '1' then
      if RESET = '1' then LIGHTS <= IDLE; else
        case LIGHTS is
          when IDLE => if HAZ='1' or (LEFT='1' and RIGHT='1') then LIGHTS <= LR3;
                       elsif LEFT='1'                         then LIGHTS <= L1;
                       elsif RIGHT='1'                        then LIGHTS <= R1;
                       else                                        LIGHTS <= IDLE;
                       end if;
          when L1   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= L2; end if;
          when L2   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= L3; end if;
          when L3   => LIGHTS <= IDLE;
          when R1   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= R2; end if;
          when R2   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= R3; end if;
          when R3   => LIGHTS <= IDLE;
          when LR3  => LIGHTS <= IDLE;
          when others => null;
        end case;
      end if;
    end if;
  end process;
end Vtbird_arch;
```

IDLE MUSINGS

In VHDL state machines, it's not necessary to make an explicit assignment of a next state if it's the same state that the machine is already in. In the execution of a process, a VHDL signal keeps its value if no assignment is made to it. Thus, in Table XSvhd-1, the final "`else`" clause in the `IDLE` state could be omitted, with no effect on the machine's behavior.

Separately, the robustness of the state machine in Table XSvhd-1 could be improved by replacing the "`null`" statement in the "`when others`" case with a transition to the `IDLE` state.

XSvhd.2 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. A VHDL program for the guessing game is shown in Table XSvhd-2 on the next page. This version also includes a `RESET` input that forces the game to a known starting state.

The program is pretty much a straightforward translation of the original state diagram in Figure 7-60 on page 581. Perhaps its only noteworthy feature is in the "`SOK | SERR`" case. Since the next-state transitions for these two states are identical (either go to `S1` or stay in the current state), they can be handled in one case. However, this tricky style of saving typing isn't particularly desirable from the point of view of state-machine documentation or maintainability. In the author's case, the trick's primary benefit was to help fit the program on one book page!

The program in Table XSvhd-2 does not specify a state assignment; a typical synthesis tool will use three bits for `Sreg` and assign the six states in order to binary combinations 000–101. For this state machine, it is also possible to use an output-coded state assignment, using just the lamp and error output signals that are already required. VHDL does not provide a convenient mechanism for grouping together the entity's existing output signals and using them for state, but we can still achieve the desired effect with the changes shown in Table XSvhd-3. Here we used a comment to document the correspondence between outputs and the bits of the new, 5-bit `Sreg`, and we changed each of the output assignment statements to pick off the appropriate bit instead of fully decoding the state.

Table XSvhd-2 VHDL program for the guessing-game machine.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vggame is
  port ( CLOCK, RESET, G1, G2, G3, G4: in  STD_LOGIC;
         L1, L2, L3, L4, ERR:         out STD_LOGIC );
end;

architecture Vggame_arch of Vggame is
type Sreg_type is (S1, S2, S3, S4, SOK, SERR);
signal Sreg, Snext: Sreg_type;
begin

  process (CLOCK) -- state memory
    begin
      if CLOCK'event and CLOCK = '1' then
      Sreg <= Snext; end if;
    end process;

  process (RESET, G1, G2, G3, G4, Sreg) -- next-state logic
  begin
    if RESET = '1' then Snext <= SOK; else
      case Sreg is
        when S1 => if    G2='1' or G3='1' or G4='1' then Snext <= SERR;
                   elsif G1='1'                      then Snext <= SOK;
                   else                                   Snext <= S2;
                   end if;
        when S2 => if    G1='1' or G3='1' or G4='1' then Snext <= SERR;
                   elsif G2='1'                      then Snext <= SOK;
                   else                                   Snext <= S3;
                   end if;
        when S3 => if    G1='1' or G2='1' or G4='1' then Snext <= SERR;
                   elsif G3='1'                      then Snext <= SOK;
                   else                                   Snext <= S4;
                   end if;
        when S4 => if    G1='1' or G2='1' or G3='1' then Snext <= SERR;
                   elsif G4='1'                      then Snext <= SOK;
                   else                                   Snext <= S1;
                   end if;
        when SOK | SERR => if G1='0' and G2='0' and G3='0' and G4='0'
                        then Snext <= S1; end if;
        when others => Snext <= S1;
      end case;
    end if;
  end process;

  L1  <= '1' when Sreg = S1   else '0';
  L2  <= '1' when Sreg = S2   else '0';
  L3  <= '1' when Sreg = S3   else '0';
  L4  <= '1' when Sreg = S4   else '0';
  ERR <= '1' when Sreg = SERR else '0';

end Vggame_arch;
```

```
architecture Vggameoc_arch of Vggame is
signal Sreg: STD_LOGIC_VECTOR (1 to 5);
-- bit positions of output-coded assignment: L1, L2, L3, L4, ERR
constant S1:   STD_LOGIC_VECTOR (1 to 5) := "10000";
constant S2:   STD_LOGIC_VECTOR (1 to 5) := "01000";
constant S3:   STD_LOGIC_VECTOR (1 to 5) := "00100";
constant S4:   STD_LOGIC_VECTOR (1 to 5) := "00010";
constant SERR: STD_LOGIC_VECTOR (1 to 5) := "00001";
constant SOK:  STD_LOGIC_VECTOR (1 to 5) := "00000";
begin
  ...                -- no change to memory or next-state processes
  L1  <= Sreg(1);
  L2  <= Sreg(2);
  L3  <= Sreg(3);
  L4  <= Sreg(4);
  ERR <= Sreg(5);

end Vggameoc_arch;
```

Table XSvhd-3 VHDL architecture for guessing game using output-coded state assignment.

XSvhd.3 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)

Figure XSvhd-2 Traffic sensors and signals at an intersection in Sunnyvale, California.

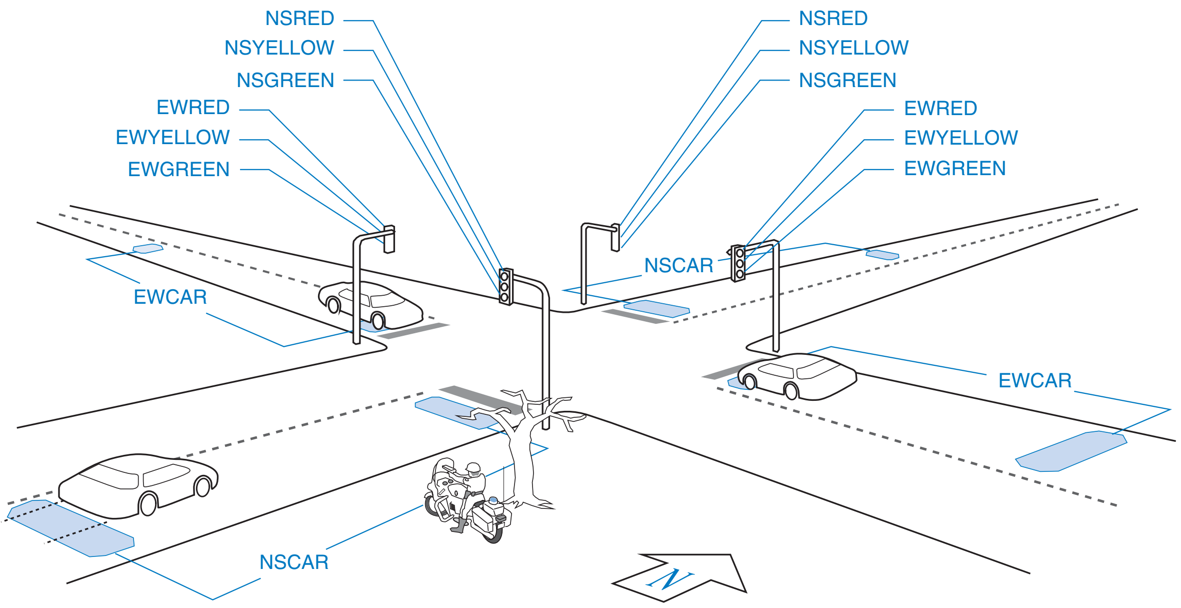

Table XSvhd-4 VHDL program for Sunnyvale traffic-light controller.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vsvale is
  port ( CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG: in  STD_LOGIC;
         OVERRIDE, FLASHCLK:                          in  STD_LOGIC;
         NSRED, NSYELLOW, NSGREEN:                    out STD_LOGIC;
         EWRED, EWYELLOW, EWGREEN, TMRESET:           out STD_LOGIC );
end;

architecture Vsvale_arch of Vsvale is
type Sreg_type is (NSGO, NSWAIT, NSWAIT2, NSDELAY,
                   EWGO, EWWAIT, EWWAIT2, EWDELAY);
signal Sreg, Snext: Sreg_type;

begin
  process (CLOCK) -- state memory
    begin
      if CLOCK'event and CLOCK = '1' then
      Sreg <= Snext; end if;
    end process;

process (RESET, NSCAR, EWCAR, TMSHORT, TMLONG, Sreg) -- next-state logic
begin
  if RESET = '1' then Snext <= NSDELAY; else
    case Sreg is
      when NSGO =>                                         -- North-south green.
        if    TMSHORT='0' then Snext <= NSGO;              -- Minimum 5 seconds.
        elsif TMLONG='1'  then Snext <= NSWAIT;            -- Maximum 5 minutes.
        elsif EWCAR='1' and NSCAR='0' then Snext <= NSGO;   -- Make EW car wait.
        elsif EWCAR='1' and NSCAR='1' then Snext <= NSWAIT; -- Thrash if cars both ways.
        elsif EWCAR='0' and NSCAR='1' then Snext <= NSWAIT; -- New NS car? Make it stop!
        else                               Snext <= NSGO;   -- No one coming, no change.
        end if;
      when NSWAIT  => Snext <= NSWAIT2;                    -- Yellow light,
      when NSWAIT2 => Snext <= NSDELAY;                    --   two ticks for safety.
      when NSDELAY => Snext <= EWGO;                       -- Red both ways for safety.
      when EWGO =>                                         -- East-west green.
        if    TMSHORT='0' then Snext <= EWGO;              -- Same behavior as above.
        elsif TMLONG='1'  then Snext <= EWWAIT;
        elsif NSCAR='1' and EWCAR='0' then Snext <= EWGO;
        elsif NSCAR='1' and EWCAR='1' then Snext <= EWWAIT;
        elsif NSCAR='0' and EWCAR='1' then Snext <= EWWAIT;
        else                               Snext <= EWGO;
        end if;
      when EWWAIT  => Snext <= EWWAIT2;
      when EWWAIT2 => Snext <= EWDELAY;
      when EWDELAY => Snext <= NSGO;
      when others  => Snext <= NSDELAY;                    -- "Reset" state.
    end case;
  end if;
end process;
```

Table XSvhd-5 (continued) VHDL program for Sunnyvale traffic-light controller.

```
-- Output logic
TMRESET  <= '1' when Sreg=NSWAIT2 or Sreg=EWWAIT2 else '0';
NSRED    <= FLASHCLK when OVERRIDE='1' else
            '1' when Sreg/=NSGO and Sreg/=NSWAIT and Sreg/=NSWAIT2 else '0';
NSYELLOW <= '0' when OVERRIDE='1' else
            '1' when Sreg=NSWAIT or Sreg=NSWAIT2 else '0';
NSGREEN  <= '0' when OVERRIDE='1' else '1' when Sreg=NSGO else '0';
EWRED    <= FLASHCLK when OVERRIDE='1' else
            '1' when Sreg/=EWGO and Sreg/=EWWAIT and Sreg/=EWWAIT2 else '0';
EWYELLOW <= '0' when OVERRIDE='1' else
            '1' when Sreg=EWWAIT or Sreg=EWWAIT2 else '0';
EWGREEN  <= '0' when OVERRIDE='1' else '1' when Sreg=EWGO else '0';

end Vsvale_arch;
```

has the sensors and signals shown in Figure XSvhd-2 below. 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.

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 VHDL program of Table XSvhd-4. 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.

While writing the program, we took the opportunity to add two inputs that weren't in the original specification. The OVERRIDE 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 the FLASHCLK input. This allows them to manually clear up the traffic snarls created by this wonderful invention.

Like most of our other examples, Table XSvhd-4 does not give a specific state assignment. And like many of our other examples, this state machine works well with an output-coded state assignment. Many of the states can be identified

Table XSvhd-6 Definitions for Sunnyvale traffic-lights machine with output-coded state assignment.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vsvale is
  port ( CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG: in  STD_LOGIC;
         OVERRIDE, FLASHCLK:                          in  STD_LOGIC;
         NSRED, NSYELLOW, NSGREEN:                    out STD_LOGIC;
         EWRED, EWYELLOW, EWGREEN, TMRESET:           out STD_LOGIC );
end;

architecture Vsvaleoc_arch of Vsvale is
signal Sreg: STD_LOGIC_VECTOR (1 to 7);
-- bit positions of output-coded assignment:  (1) NSRED, (2) NSYELLOW, (3) NSGREEN,
--                                 (4) EWRED, (5) EWYELLOW, (6) EWGREEN, (7) EXTRA
constant NSGO:    STD_LOGIC_VECTOR (1 to 7) := "0011000";
constant NSWAIT:  STD_LOGIC_VECTOR (1 to 7) := "0101000";
constant NSWAIT2: STD_LOGIC_VECTOR (1 to 7) := "0101001";
constant NSDELAY: STD_LOGIC_VECTOR (1 to 7) := "1001000";
constant EWGO:    STD_LOGIC_VECTOR (1 to 7) := "1000010";
constant EWWAIT:  STD_LOGIC_VECTOR (1 to 7) := "1000100";
constant EWWAIT2: STD_LOGIC_VECTOR (1 to 7) := "1000101";
constant EWDELAY: STD_LOGIC_VECTOR (1 to 7) := "1001001";

begin
...             -- no change to memory or next-state processes
TMRESET  <= '1' when Sreg=NSWAIT2 or Sreg=EWWAIT2 else '0';
NSRED    <= Sreg(1);
NSYELLOW <= Sreg(2);
NSGREEN  <= Sreg(3);
EWRED    <= Sreg(4);
EWYELLOW <= Sreg(5);
EWGREEN  <= Sreg(6);

end Vsvaleoc_arch;
```

by a unique combination of light-output values. But there are three pairs of states that are not distinguishable by looking at the lights alone: (NSWAIT, NSWAIT2), (EWWAIT, EWWAIT2), and (NSDELAY, EWDELAY). We can handle these by adding one more state variable, "EXTRA", that has different values for the two states in each pair. This idea is realized in the modified program in Table XSvhd-5.

XSvhd.4 A Synchronous System Design Example

This subsection presents a representative example of a synchronous system design in VHDL. The example is a *shift-and-add multiplier* for unsigned integers using the algorithm of Section 2.8. Besides illustrating synchronous design (using a single clock), this example also shows the hierarchical possibilities of design with VHDL.

shift-and-add multiplier

As shown in Figure XSvhd-3, the multiplier design has five individual modules nested up to three levels deep. The top level is broken into a datapath and a control unit, and the control unit contains both a state machine and a counter. The design also creates and all of the entities use the MPYdefs package shown in Table XSvhd-6. By changing the constant MPYwidth in this package, the designer can create a shift-and-add multiplier of any desired width; we'll use a width of 8 in the rest of this discussion.

```
package MPYdefs is

constant MPYwidth:  integer := 8;
constant MPYmsb:    integer := MPYwidth-1;
constant PRODmsb:   integer := 2*MPYwidth-1;
constant MaxCnt:    integer := MPYmsb;
subtype  CNTRrange is integer range 0 to MaxCnt;
type SMstate is (IDLE, INIT, RUN, WAIT);

end MPYdefs;
```

Table XSvhd-7 Common definitions for shift-and-add multiplier.

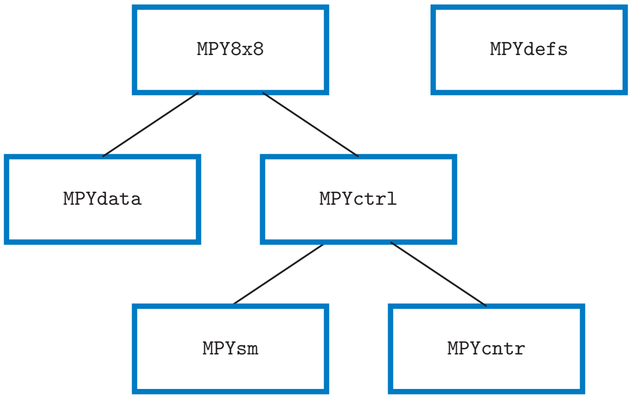

Figure XSvhd-3 VHDL entities and package used in the shift-and-add multiplier.

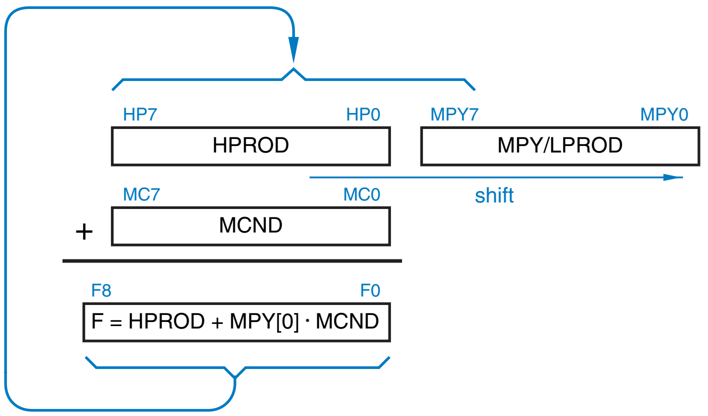

Figure XSvhd-4 Registers and functions used by the shift-and-add multiplication algorithm.

As we showed in Figure XSvhd-3, the multiplier is partitioned into two entities—a control unit `MPYctrl` and a data unit `MPYdata`. Before looking at details of either, it's important for you to understand the basic data-unit registers and functions that are used to perform an 8-bit multiplication, as shown in Figure XSvhd-4:

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 VHDL 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.
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.

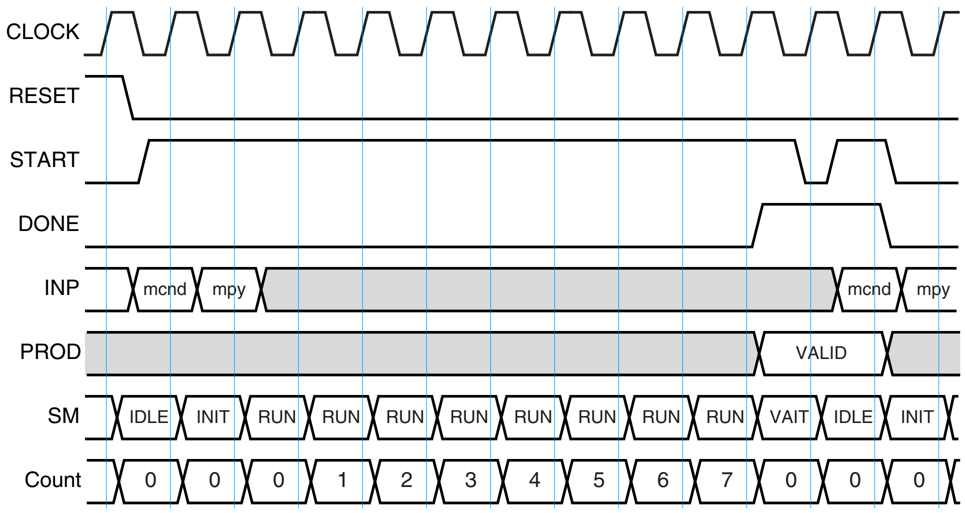

Figure XSvhd-5 Timing diagram for multiplier system.

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 XSvhd-5. 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 it *may* be started later.

Our control unit `MPYcntrl` for running the multiplication is based on a decomposed state machine, as introduced in Section 7.8. A top-level state machine `MPYsm` controls the overall operation, while a counter `MPYcntr` counts the eight shift-and-add steps. These three VHDL entities are shown in tables on the next three pages.

Table XSvhd-8 VHDL control-unit entity `MPYctrl`.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.MPYdefs.all;

entity MPYctrl is
  port (RESET, CLK: in STD_LOGIC;
        START:      in STD_LOGIC;
        DONE:       out STD_LOGIC;
        SM:         out SMstate   );
end;

architecture MPYctrl_arch of MPYctrl is

component MPYsm   port (RESET, CLK: in STD_LOGIC;
                        START:      in STD_LOGIC;
                        MAX:        in STD_LOGIC;
                        SM:         out SMstate   );
end component;

component MPYcntr port (RESET, CLK: in STD_LOGIC;
                        SM:         in SMstate;
                        MAX:        out STD_LOGIC );
end component;

signal MAX: STD_LOGIC;
signal SMi: SMstate;

begin
  U1: MPYsm   port map (RESET=>RESET, CLK=>CLK, START=>START, MAX=>MAX, SM=>SMi);
  U2: MPYcntr port map (RESET=>RESET, CLK=>CLK, SM=>SMi, MAX=>MAX);

process (CLK) -- implement DONE output function
  begin
    if CLK'event and CLK='1' then
      if RESET='1' then DONE <= '0';
      elsif (SMi=RUN and MAX='1') or (SMi=VAIT) then DONE <= '1';
      else DONE <= '0';
      end if;
    end if;
  end process;

SM <= SMi; -- Output copy of SM state, visible to other entities

end MPYctrl_arch;
```

As shown in Table XSvhd-7, the control unit `MPYctrl` instantiates the state machine and the counter, and also has a small process to implement the `DONE` output function which requires a register. Notice how input signals `RESET`, `CLK`, and `START` simply “flow through” `MPYctrl` and become inputs of `MPYsm` and `MPYcntr`. Also notice how a local signal, `SMi`, is declared to receive the state from `MPYsm` and deliver it both to `SMcntr` and the output of `MPYctrl`.

Table XSvhd-9 VHDL state-machine entity `MPYsm`.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.MPYdefs.all;

entity MPYsm is
  port (RESET, CLK: in STD_LOGIC;
        START:      in STD_LOGIC;
        MAX:        in STD_LOGIC;
        SM:         out SMstate  );
end;

architecture MPYsm_arch of MPYsm is
signal Sreg, Snext: SMstate;
begin
  process (CLK) -- state memory
  begin
    if CLK'event and CLK = '1' then
      if RESET='1' then Sreg <= IDLE; -- synchronous reset
      else Sreg <= Snext; end if;
    end if;
  end process;

process (START, MAX, Sreg) -- next-state logic
  begin
    case Sreg is
      when IDLE => if START = '1'                  then Snext <= INIT;
                   else                                 Snext <= IDLE;
                   end if;
      when INIT =>                                      Snext <= RUN;
      when RUN  => if    MAX = '1' and START = '0' then Snext <= IDLE;
                   elsif MAX = '1' and START = '1' then Snext <= VAIT;
                   else                                 Snext <= RUN;
                   end if;
      when VAIT => if    START = '0'               then Snext <= IDLE;
                   else                                 Snext <= VAIT;
                   end if;
      when others  =>                                   Snext <= IDLE;
    end case;
  end process;

  SM <= Sreg; -- Output copy of SM state, visible to other entities

end MPYsm_arch;
```

The `MPYsm` state machine has four states for multiplier control. Multiplication begins when `START` is asserted. The machine goes to the `INIT` state and then the `RUN` state, and stays in the `RUN` state until the `MAX` input, produced by the `MPYcntr` entity, is asserted after eight clock ticks. Then it goes to the `IDLE` or the `VAIT` state, depending on whether or not `START` has been negated yet. (`VAIT` is named strangely to avoid conflict with the VHDL keyword `wait`.)

Table XSvhd-10
VHDL counter entity MPYcntr.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.MPYdefs.all;

entity MPYcntr is
  port (RESET, CLK: in STD_LOGIC;
        SM:         in SMState;
        MAX:        out STD_LOGIC );
end;

architecture MPYcntr_arch of MPYcntr is
signal Count: CNTRrange;
begin
  process (CLK)
  begin
    if CLK'event and CLK='1' then
      if RESET='1' then Count <= 0;
      elsif SM=RUN then Count <= (Count + 1) mod MPYwidth;
      else Count <= 0;
      end if;
    end if;
  end process;

  MAX <= '1' when Count = MaxCnt else '0';

end MPYcntr_arch;
```

The `MPYcntr` entity counts from 0 to `MaxCnt` (`MPYwidth-1`) when the state machine is in the `RUN` state. The state-machine states and counter values during an 8-bit multiplication sequence were shown in the last two waveforms in Figure XSvhd-5.

The multiplier data path logic is defined in the `MPYdata` entity, shown in Table XSvhd-10. This entity declares local registers `MPY`, `MCND`, and `HPROD`. Note that these all have type `UNSIGNED` from the `std_logic_arith` package, so that the addition operation is supported. Note the use of concatenation to pad the addends to nine bits in the addition operation that assigns a value to `F` near the end of the module.

STOP COMPLAINING!

In Table XSvhd-9, notice the `mod` operation that forces the counter state back to 0 after state `MaxCnt` (`MPYwidth-1`). This adds a little bit of logic to the counter's realization if `MPYwidth` is not a power of two, and it's not really needed for proper system operation. However, during simulation, the VHDL simulator will complain if `Count` takes on a value outside `CNTRrange`. So this extra code and logic were added simply to make the simulator stop complaining.

Table XSvhd-11
VHDL data-path entity MPYdata.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.MPYdefs.all;

entity MPYdata is
  port (RESET, CLK, START: in STD_LOGIC;
        INP:               in UNSIGNED (MPYmsb downto 0);
        SM:                in SMstate;
        PROD:              out UNSIGNED (PRODmsb downto 0) );
end;

architecture MPYdata_arch of MPYdata is

signal MPY, MCND, HPROD: UNSIGNED (MPYmsb downto 0);
signal F: UNSIGNED (MPYmsb+1 downto 0);

begin

  process (CLK) -- implement registers
  begin
    if CLK'event and CLK='1' then
      if RESET='1' then
        MPY  <= (others=>'0'); MCND <= (others=>'0');
        HPROD <= (others=>'0'); -- clear registers on reset
      elsif (SM=IDLE and START='1') then  -- load MCND, clear HPROD
        MCND <= INP; HPROD <= (others => '0');
      elsif SM=INIT                 then MPY <= INP;  -- load MPY
      elsif SM=RUN then
        MPY <= F(0) & MPY(MPYmsb downto 1);
        HPROD <= F(MPYmsb+1 downto 1);
      end if;
    end if;
  end process;

  F <= ('0' & HPROD) + ('0' & MCND) when MPY(0)='1'
       else ('0' & HPROD);

  PROD <= HPROD & MPY;

end MPYdata_arch;
```

Besides the RESET, CLK, and INP inputs and the PROD output, which you would naturally need for the data path, this entity also has START and the state-machine state SM as inputs. These are needed to determine when to load the MPY and MCND registers, and when to update the partial product (in the RUN state).

The last statement in this module produces the PROD output as a combinational concatenation of the HPROD and MPY registers.

Table XSvhd-12 VHDL top-level entity MPY8x8.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.MPYdefs.all;

entity MPY8x8 is
  port (RESET, CLK: in STD_LOGIC;
        START:      in STD_LOGIC;
        INP:        in UNSIGNED (MPYmsb downto 0);
        DONE:       out STD_LOGIC;
        PROD:       out UNSIGNED (PRODmsb downto 0) );
end;

architecture MPY8x8_arch of MPY8x8 is

component MPYdata port (RESET, CLK, START: in STD_LOGIC;
                        INP:               in UNSIGNED (MPYmsb downto 0);
                        SM:                in SMstate;
                        PROD:              out UNSIGNED (PRODmsb downto 0) );
end component;

component MPYctrl port (RESET, CLK: in STD_LOGIC;
                        START:      in STD_LOGIC;
                        DONE:       out STD_LOGIC;
                        SM:         out SMstate   );
end component;

signal SM: SMstate;

begin
  U1: MPYdata port map (RESET=>RESET, CLK=>CLK, START=>START, INP=>INP,
                        SM=>SM, PROD=>PROD);
  U2: MPYctrl port map (RESET=>RESET, CLK=>CLK, START=>START, DONE=>DONE, SM=>SM);
end MPY8x8_arch;
```

VERY VERBOSE VHDL

Comparing this example's roughly five pages of VHDL in six individual files with the ABEL description of the same function in less than two pages in Section XSabl.6, you probably get the sense that hierarchy might be nice but it has a price. However, this verboseness is due much more to the nature of VHDL than to hierarchy.

For example, the component port definitions had to be replicated in every entity that instantiates another, the library definitions were repeated everywhere, and VHDL is just downright verbose even for the simplest things (e.g., compare VHDL's "`MPYmsb downto 0`" with ABEL's "`MPYmsb..0`" or Verilog's "`MPYmsb:0`"). The same design in Verilog is much more compact, as shown in Section XSver.4.

Table XSvhd-13 VHDL test bench for shift-and-add multiplier.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.MPYdefs.all;

entity MPY8x8_tb is
end;

architecture MPY8x8_tb of MPY8x8_tb is
signal Tclk, RST, START, DONE: STD_LOGIC;
signal INP:  UNSIGNED (MPYmsb downto 0);
signal PROD: UNSIGNED (PRODmsb downto 0);

component MPY8x8 -- The UUT
  port (RESET, CLK, START: in STD_LOGIC;
        INP:               in UNSIGNED (MPYmsb downto 0);
        DONE:              out STD_LOGIC;
        PROD:              out UNSIGNED (PRODmsb downto 0) );
end component;

begin
  UUT: MPY8x8 port map( CLK => Tclk, RESET => RST, START => START, INP => INP,
                        DONE => DONE, PROD => PROD);   -- instantiate UUT

process -- create free-running test clock
  begin
    Tclk <= '1'; wait for 5 ns;     -- clock cycle 10 ns
    Tclk <= '0'; wait for 5 ns;
  end process;

process -- apply tests
  variable ii, jj, cnt: integer;
  begin
    RST <= '1'; START <= '0'; INP <= (others => '0');
    wait for 15 ns;    -- reset for 15 ns
    RST <= '0';        -- then apply inputs and check outputs
    for ii in 0 to 2**MPYwidth-1 loop
      for jj in 0 to 2**MPYwidth-1 loop
        START <= '1'; INP <= CONV_UNSIGNED(ii, MPYwidth);
        wait for 10 ns;
        START <= '0'; INP <= CONV_UNSIGNED(jj, MPYwidth);
        wait for 10 ns;
        for cnt in 0 to MPYwidth-1 loop wait for 10 ns; end loop;
        assert (CONV_INTEGER(PROD)=ii*jj) report "Bad product" severity ERROR;
      end loop;
    end loop;
    report "Ending test bench for MPY8x8" severity NOTE;
    wait; -- end simulation
  end process;

end MPY8x8_tb;
```

Finally, the `MPY8x8` entity in Table XSvhd-11 on the preceding page instantiates the data-path and control-unit entities to create the multiplier system. Besides the main system inputs and outputs, it declares only one local signal `SM` to convey the state-machine state from the control unit to the data path.

A test bench can be written for the multiplier as shown in Table XSvhd-12. This test bench has two processes. The first creates a free-running clock with a 10-ns period. The second has a nested `for` loop that performs multiplication of all possible pairs of numbers, taking ten clock ticks for each pair.

After multiplying each pair of numbers, the test bench compares the circuit's result (`PROD`) with a result calculated by the simulator, and prints an error message if there is a mismatch. Of course, the error message would be a lot more useful if it also included the values of `ii`, `jj`, `PROD`, and the expected result. Unfortunately, VHDL does not have any nice built-in functions for input and output of signal and variable values. Some VHDL suppliers have produced decent text and file input/output packages to perform these functions, but their definitions and use are beyond the scope of this book.

Exercises

XSvhd.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 VHDL, devise a state machine that does the job using no more than four states. Include comments to describe each state's meaning and use. Write a VHDL test bench that checks your machine for proper operation by applying three bytes in succession with even, odd, and even parity.

XSvhd.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. Augment the test bench to check for proper machine operation in the newly defined conditions.

XSvhd.3 Using VHDL, 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. Write a VHDL test bench that checks your design for proper operation. (*Hint:* No more than ten states are required.)

XSvhd.4 Using VHDL, design a parallel-to-serial conversion circuit with eight 2.048-Mbps, 32-channel serial links and a single 2.048-MHz, 8-bit, parallel data bus that carries 256 bytes per frame. Each serial link should have the frame format defined in Figure XSbb-3 in Section XSbb.1. Each serial data line

SDATAi should have its own sync signal SYNCi; the sync pulses should be staggered so that SYNCi + 1 has a pulse one tick after SYNCi. Show the timing of the parallel bus and the serial links, and write a table or formula that shows which parallel-bus timeslots are transmitted on which serial links and timeslots. Create a test bench that checks your design by applying at least one frame's worth of sequential parallel data (256 bytes, 0 through 255) to your circuit, and checking for the corresponding data on the serial data lines.

XSvhd.5 In the same environment as the preceding exercise, design a serial-to-parallel conversion circuit that converts eight serial data lines into a parallel 8-bit data bus. Create a test bench that connects the SDATAi outputs of the preceding exercise with the SDATAi inputs of this one. It should check the two circuits together by applying random parallel data bytes to the inputs of the first and looking for them at the output of the second, a certain number of clock ticks later.

XSvhd.6 Redesign the T-bird tail-lights machine of Section XSvhd.1 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. Partition the VHDL design into as many entities as you feel are appropriate, but target the design to a single CPLD or FPGA. Also, write a short description of how your system works.

XSvhd.7 The operation of the guessing game in Section XSvhd.2 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 VHDL guessing-game program of Table XSvhd-2 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.) Button pushes should be recognized whether or not `SEN` is asserted.

Also, add another process to the program to provide a pseudorandom bit-stream generator using an 8-bit LFSR. 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? How can you double these numbers?

XSvhd.8 Modify the behavior of the VHDL traffic-light-controller machine in Table XSvhd-4 to have more reasonable behavior, the kind you'd like to see for traffic lights in your own home town.

XSvhd.9 Using VHDL, 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.

XSvhd.10 Using VHDL, 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.