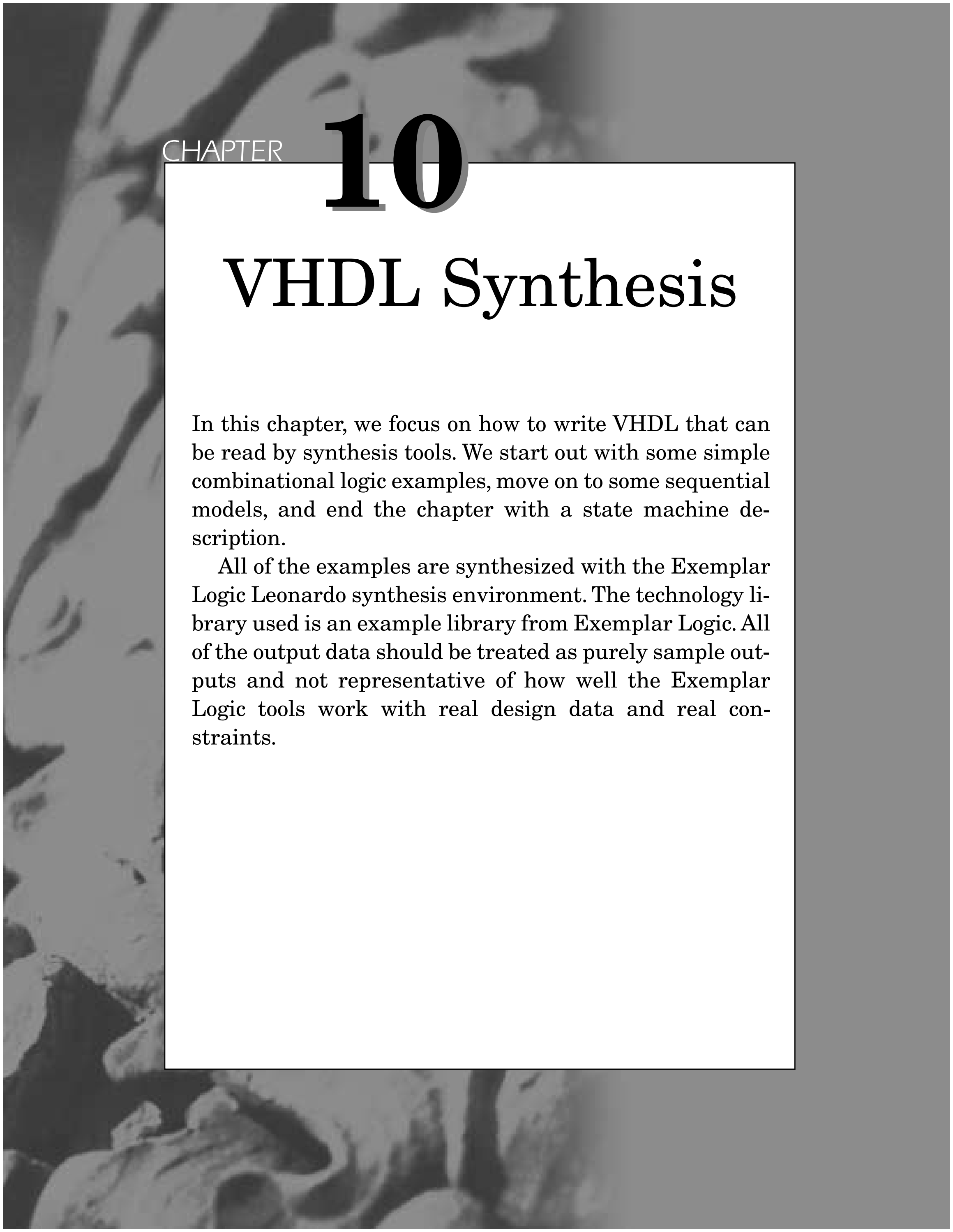


CHAPTER 10

VHDL Synthesis

In this chapter, we focus on how to write VHDL that can be read by synthesis tools. We start out with some simple combinational logic examples, move on to some sequential models, and end the chapter with a state machine description.

All of the examples are synthesized with the Exemplar Logic Leonardo synthesis environment. The technology library used is an example library from Exemplar Logic. All of the output data should be treated as purely sample outputs and not representative of how well the Exemplar Logic tools work with real design data and real constraints.

Simple Gate—Concurrent Assignment

The first example is a simple description for a 3-input **OR** gate:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY or3 IS
  PORT (a, b, c : IN std_logic;
         d : OUT std_logic);
END or3;

ARCHITECTURE synth OF or3 IS
BEGIN
 d <= a OR b OR c;
END synth;
```

This model uses a simple concurrent assignment statement to describe the functionality of the **OR** gate. The model specifies the functionality required for this entity, but not the implementation. The synthesis tool can choose to implement this functionality in a number of ways, depending on the cells available in the technology library and the constraints on the model. For instance, the most obvious implementation is shown in Figure 10-1.

This implementation uses a 3-input **OR** gate to implement the functionality specified in the concurrent signal assignment statement contained in architecture **synth**.

What if the technology library did not contain a 3-input **OR** device? Two other possible implementations are shown in Figures 10-2 and 10-3.

The first implementation uses a 3-input **NOR** gate followed by an inverter. The synthesis tool may choose this implementation if there are no 3-input **OR** devices in the technology library. Alternatively, if there are no 3-input devices, or if the 3-input devices violate a speed constraint, the

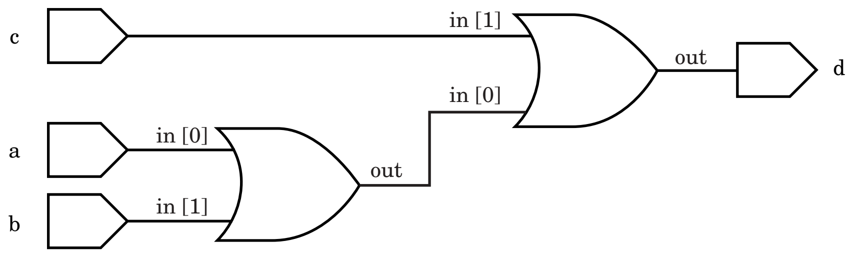


Figure 10-1 Model Implementation.

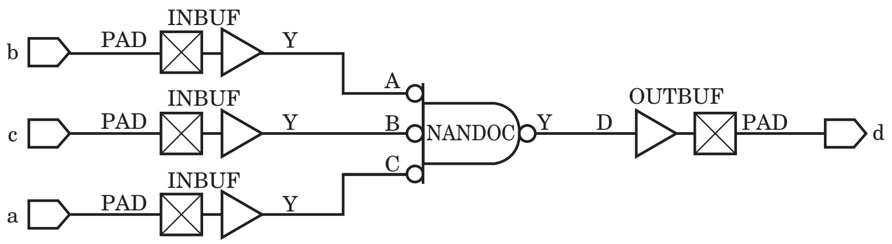


Figure 10-2
3-Input OR.

3-input OR function could be built from four devices, as shown in Figure 10-3. Given a technology library of parts, the functionality desired, and design constraints, the synthesis tool is free to choose among any of the implementations that satisfy all the requirements of a design, if such a design exists. There are lots of cases where the technology or constraints are such that no design can meet all of the design requirements.

IF Control Flow Statements

In the next example, control flow statements such as **IF THEN ELSE** are used to demonstrate how synthesis from a higher level description is accomplished. This example forms the control logic for a household alarm system. It uses sensor input from a number of sensors to determine whether or not to trigger different types of alarms. Following is the input description:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY alarm_cntrl IS
  PORT( smoke, front_door, back_door, side_door,
        alarm_disable, main_disable,
        water_detect : IN std_logic;
        fire_alarm, burg_alarm,
        water_alarm : OUT std_logic);
END alarm_cntrl;

ARCHITECTURE synth OF alarm_cntrl IS
BEGIN
  PROCESS(smoke, front_door, back_door, side_door,
          alarm_disable, main_disable,
          water_detect)
  BEGIN
```

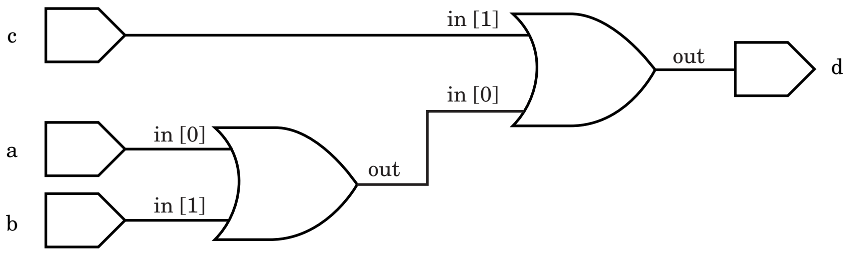


Figure 10-3 Another 3-Input OR Implementation.

```
     IF ((smoke = '1') AND (main_disable = '0')) THEN
       fire_alarm <= '1';
    ELSE
     fire_alarm <= '0';
   END IF;

        IF (((front_door = '1') OR (back_door = '1') OR
           (side_door = '1')) AND
             ((alarm_disable = '0') AND (main_disable =
               '0'))) THEN
          burg_alarm <= '1';
        ELSE
          burg_alarm <= '0';
        END IF;

        IF ((water_detect = '1') AND (main_disable = '0'))
           THEN
          water_alarm <= '1';
        ELSE
          water_alarm <= '0';
        END IF;
    END PROCESS;
  END synth;
```

The input description contains a number of sensor input ports such as a smoke detector input, a number of door switch inputs, a basement water detector, and two disable signals. The `main_disable` port is used to disable all alarms, while the `alarm_disable` port is used to disable only the burglar alarm system.

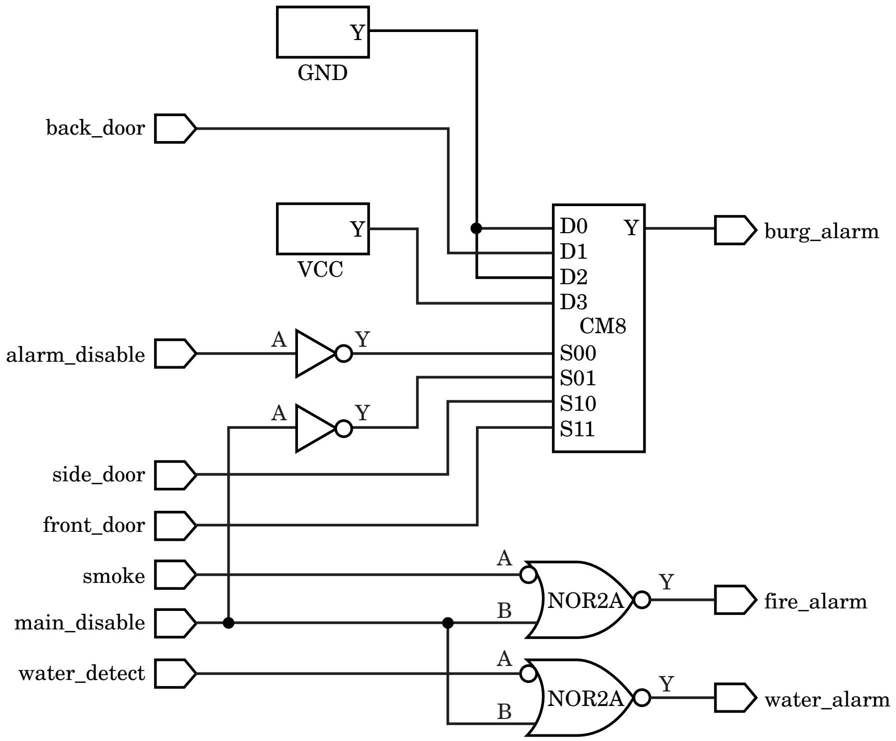
The functionality is described by three separate `IF` statements. Each `IF` statement describes the functionality of one or more output ports. Notice that the functionality could also be described very easily with equations, as in the first example. Sometimes, however, the `IF` statement style is more readable. For instance, the first `IF` statement can be described by the following equation:

```
fire_alarm <= smoke and not(main_disable);
```

Because the three **IF** statements are separate and they generate separate outputs, we can expect that the resulting logic would be three separate pieces of logic. However, the **main_disable** signal is shared between the three pieces of logic. Any operations that make use of this signal may be shared by the other logic pieces. How this sharing takes place is determined by the synthesis tool and is based on the logical functionality of the design and the constraints. Speed constraints may force the logical operations to be performed in parallel.

A sample synthesized output is shown in Figure 10-4. Notice that signal **main_disable** connects to all three output gates, while signal **alarm_disable** only connects to the alarm control logic. The logic for the water alarm and smoke detector turn out to be quite simple, but we could have guessed that because our equations were so simple. The next example is not so simple.

Figure 10-4
A sample synthesized output.



Case Control Flow Statements

The next example is an implementation of a comparator. There are two 8-bit inputs to be compared and a CTRL input that determines the type of comparison made. The possible comparison types are A > B, A < B, A = B, A ≠ B, A ≧ B, and A ≦ B. The design contains one output port for each of the comparison types. If the desired comparison output is true, then the output value on that output port is a `'1'`. If false, the output port value is a `'0'`. Following is a synthesizable VHDL description of the comparator:

```
PACKAGE comp_pack IS
  TYPE bit8 is range 0 TO 255;
  TYPE t_comp IS (greater_than, less_than, equal,
                  not_equal, grt_equal, less_equal);
END comp_pack;

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.comp_pack.ALL;
ENTITY compare IS
  PORT( a, b : IN bit8;
        ctrl : IN t_comp;
        gt, lt, eq, neq, gte, lte : OUT std_logic);
 END compare;

ARCHITECTURE synth OF compare IS
BEGIN

  PROCESS(a, b, ctrl)
  BEGIN
   gt <= '0'; lt <= '0'; eq <= '0'; neq <= '0'; gte <=
     '0'; lte <= '0';
   CASE ctrl IS
     WHEN greater_than =>
       IF (a > b) THEN
         gt <= '1';
       END IF;
     WHEN less_than =>
       IF (a < b) THEN
         lt <= '1';
       END IF;
     WHEN equal =>
       IF (a = b) THEN
         eq <= '1';
       END IF;
     WHEN not_equal =>
       IF (a /= b) THEN
         neq <= '1';
       END IF;
     WHEN grt_equal =>
       IF (a >= b) THEN
```

```
          gte <= '1';
        END IF;
      WHEN less_equal =>
        IF (a > b) THEN
          lte <= '1';
        END IF;
    END CASE;
   END PROCESS;
END synth;
```

Notice that, in this example, the equations of the inputs and outputs are harder to write because of the comparison operators. It is still possible to do, but is much less readable than the case statement shown earlier.

When synthesizing a design, the complexity of the design is related to the complexity of the equations that describe the design function. Typically, the more complex the equations, the more complex the design created. There are exceptions to this rule, especially when the equations reduce to nothing.

A sample synthesized output from the preceding description is shown in Figure 10-5. The inputs are shown on the left of the schematic diagram, and the outputs are shown in the lower right of the schematic. The equations for the comparison operators have all been shared and combined together to produce an optimal design. This design is a very small number of gates for the operation performed.

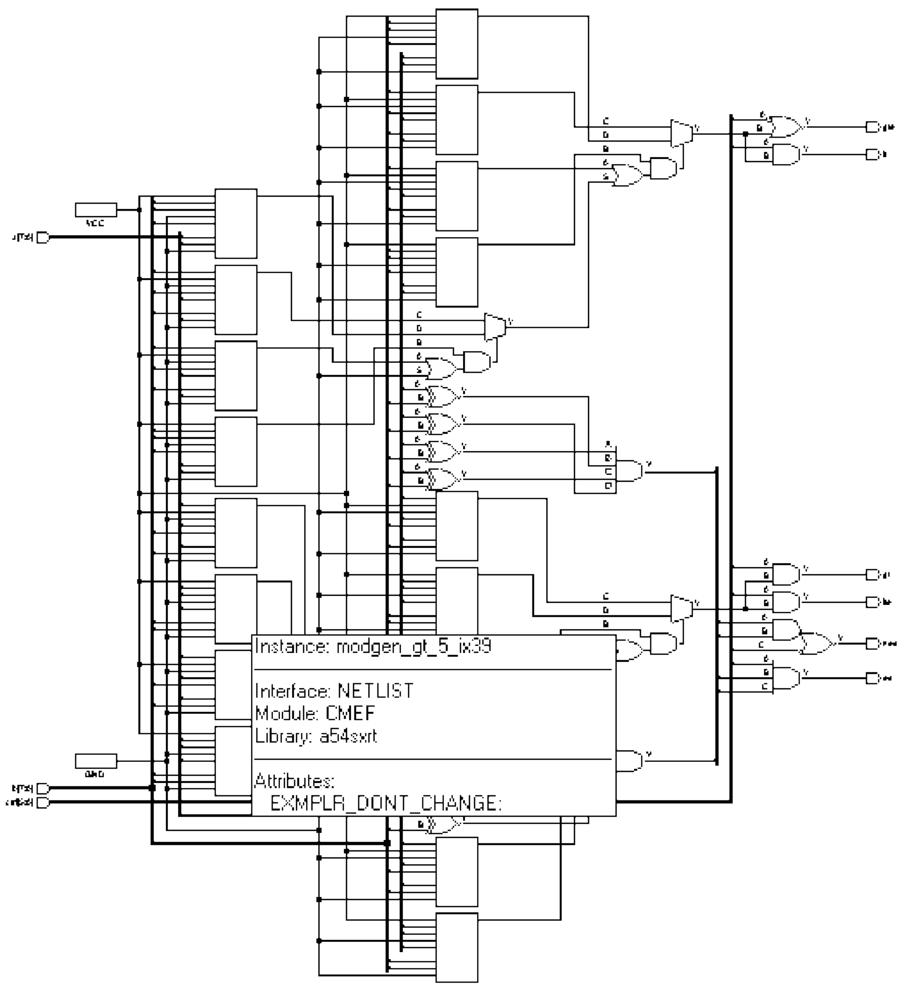
There are still a number of cases where hand design can create smaller designs, but in most cases today the results of synthesis are very good; and you get the added benefit of using a higher level design language for easier maintainability and a shorter design cycle.

Simple Sequential Statements

Let's take a closer look at an example that we already discussed in the last chapter. This is the inferred D flip-flop. Inferred flip-flops are created by **WAIT** statements or **IF THEN ELSE** statements, which are surrounded by sensitivities to a clock. By detecting clock edges, the synthesis tool can locate where to insert flip-flops so that the design that is ultimately built behaves as the simulation predicts.

Following is an example of a simple sequential design using a **WAIT** statement:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY dff IS
```



Instance: modgen_gt_5_ix39

Interface: NETLIST
Module: CMEF
Library: a54sxrt

Attributes:
  EXMPLR_DONT_CHANGE:

Figure 10-5
A Sample Synthesized Output.

```
  PORT( clock, din : IN std_logic;
        dout : OUT std_logic);
END dff;

ARCHITECTURE synth OF dff IS
BEGIN
  PROCESS
  BEGIN
   WAIT UNTIL ((clock'EVENT) AND (clock = '1'));

   dout <= din;

  END PROCESS;
END synth;
```

The description contains a synthesizable entity and architecture representing a D flip-flop. The entity contains the **clock**, **din**, and **dout** ports needed for a D flip-flop, while the architecture contains a single process statement with a single **WAIT** statement. When the clock signal has a rising edge occur, the contents of **din** are assigned to **dout**. Effectively, this is how a D flip-flop operates.

The synthesized output of this design matches the functionality of the RTL description. It is very important for the synthesis and simulation results to agree. Otherwise, the resulting synthesized design may not work as planned. Part of the synthesis methodology should require that a final gate level simulation of the design is executed to verify that the gate level functionality is correct. (We perform this step in an example later on.)

The output of the Leonardo synthesis tool is shown in Figure 10-6. As expected, the output of the synthesis tool produced a single D flip-flop. The synthesis tool connected the ports of the entity to the proper ports of actual FPGA library macro so that the device works as expected in the design.

Asynchronous Reset

In a number of instances, D flip-flops are required to have an asynchronous reset capability. The previous D flip-flop did not have this capability. How would we generate a D flip-flop with an asynchronous reset? Remember the simulation and synthesis results must agree. Following is one way to accomplish this:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY dff_asynch IS
```

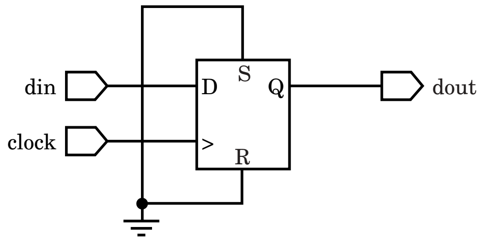


Figure 10-6
The Output of the Leonardo Synthesis Tool.

```
  PORT( clock, reset, din : IN std_logic;
        dout : OUT std_logic);
END dff_asynch;

ARCHITECTURE synth OF dff_asynch IS
BEGIN
  PROCESS(reset, clock)
  BEGIN
   IF (reset = '1') THEN
     dout <= '0';
   ELSEIF (clock'EVENT) AND (clock = '1') THEN
     dout <= din;
   END IF;
  END PROCESS;
END synth;
```

The **ENTITY** statement now has an extra input, the **reset** port, which is used to asynchronously reset the D flip-flop. Notice that **reset** and **clock** are in the process sensitivity list and cause the process to be evaluated. If an event occurs on signals **clock** or **reset**, the statements inside the process are executed.

First, signal **reset** is tested to see if it has an active value (**'1'**). If active, the output of the flip-flop is reset to **'0'**. If **reset** is not active (**'0'**), then the **clock** signal is tested for a rising edge. If signal **clock** has a rising edge, then input **din** is assigned as the new flip-flop output.

The fact that the **reset** signal is tested first in the **IF** statement gives the **reset** signal a higher priority than the **clock** signal. Also, because the **reset** signal is tested outside of the test for a clock edge, the **reset** signal is asynchronous to the clock.

The Leonardo synthesis tool produces a D flip-flop with an asynchronous **reset** input, as shown in Figure 10-7. The resulting design has an extra inverter (**IVP** component) in the circuit because the only flip-flop macro that would match the functionality required had a **reset** input that was active low.

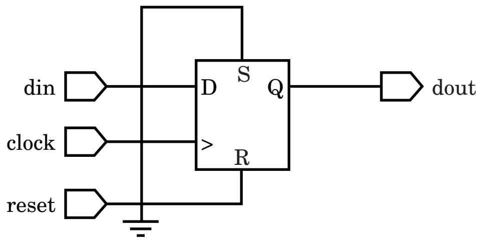


Figure 10-7 The Leonardo Synthesis Tool Produces a D Flip-Flop.

Asynchronous *Preset* and *Clear*

Is it possible to describe a flip-flop with an asynchronous `preset` and `clear`? As an attempt, we can use the same technique as in the asynchronous `reset` example. The following example illustrates an attempt to describe a flip-flop with an asynchronous `preset` and `clear` inputs:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY dff_pc IS
  PORT( preset, clear, clock, din : IN std_logic;
        dout : OUT std_logic);
END dff_pc;

ARCHITECTURE synth OF dff_pc IS
BEGIN
PROCESS(preset, clear, clock)
  BEGIN
   IF (preset = '1') THEN
     dout <= '1';

   ELSEIF (clear = '1') THEN
     dout <= '0';

   ELSEIF (clock'EVENT) AND (clock = '1') THEN
     dout <= din;

   END IF;
  END PROCESS;
END synth;
```

The entity contains a `preset` signal that sets the value of the flip-flop to a `'1'`, a clear signal that sets the value of the flip-flop to a `'0'`, and the normal `clock` and `din` ports used for the clocked D flip-flop operation. The architecture contains a single process statement with a single `IF` statement to describe the flip-flop behavior. The `IF` statement assigns a `'1'` to the output for a `'1'` value on the `preset` input and a `'0'` to the output for a `'1'` on the `clear` input. Otherwise, the `clock` input is checked for a rising edge, and the `din` value is clocked to the output `dout`.

What does the output of the synthesis process produce for this VHDL input? The output is shown in Figure 10-8. We were expecting the output of the synthesis tool in which the design `preset` input was connected to the `preset` input of the flip-flop, and the design `clear` input was connected to the `clear` input of the flip-flop. The output from the synthesis tool is a design in which the design `preset` and `clear` inputs are separated from the flip-flop `preset` and `clear` inputs by some logic.

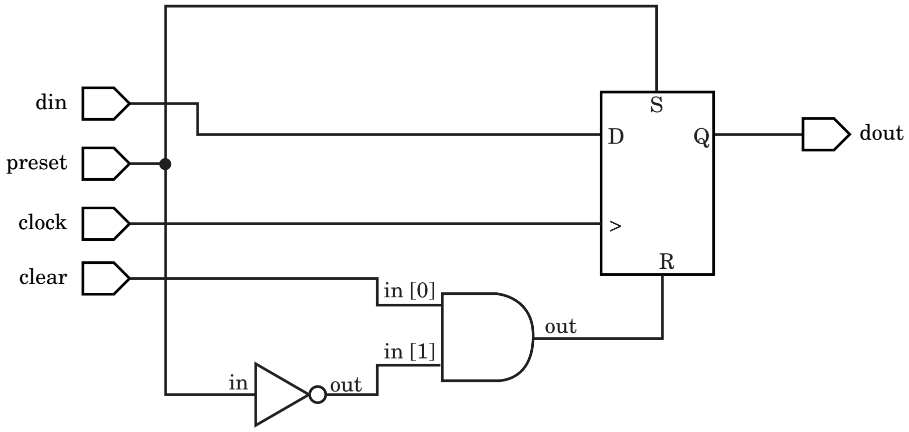


Figure 10-8
Output of synthesis process.

This logic circuitry performs a prioritization of the **`preset`** signal with respect to the **`clear`** signal. Because the **`preset`** signal occurs before the **`clear`** signal in the **`IF`** statement, the **`preset`** signal is tested before the **`clear`** signal. If the **`preset`** signal is active, the flip-flop presets regardless of the state of the **`clear`** input. Effectively, the **`preset`** signal has a higher priority than the **`clear`** signal. There is currently no way to write a VHDL description to generate a design in which the **`preset`** and **`clear`** inputs have the same priority.

More Complex Sequential Statements

The next example is a more complex sequential design of a 4-bit counter. This example makes use of a two-process description style. This style works very well for some synthesis tools, producing very good synthesis results.

Each process has a particular function. One process is clocked and the other is not. The clocked process is used to maintain the present state of the counter, while the unclocked process calculates the next state of the counter.

Following is an example of a counter written in this way:

```
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;
PACKAGE count_types IS
  SUBTYPE bit4 IS std_logic_vector(3 DOWNTO 0);
```

```
END count_types;

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE WORK.count_types.ALL;
ENTITY count IS
  PORT(clock, load, clear : IN std_logic;
       din : IN bit4;
       dout : INOUT bit4);
END count;

ARCHITECTURE synth OF count IS
  SIGNAL count_val : bit4;
BEGIN
  PROCESS(load, clear, din, dout)
  BEGIN
   IF (load = '1') THEN
     count_val <= din;
   ELSEIF (clear = '1') THEN
     count_val <= "0000";
   ELSE
     count_val <= dout + "0001";
   END IF;
  END PROCESS;

  PROCESS
  BEGIN
   WAIT UNTIL clock'EVENT and clock = '1';

   dout <= count_val;
  END PROCESS;
END synth;
```

The description contains a package that defines a 4-bit range that causes the synthesis tools to generate a 4-bit counter. Changing the size of the range causes the synthesis tools to generate different-sized counters. By using a constrained universal integer range, the model can take advantage of the built-in arithmetic operators for type universal integer. The other alternative is to define a type that is 4 bits wide and then create a package that overloads the arithmetic operators for the 4-bit type.

The entity contains a **`clock`** input port to clock the counter, a **`load`** input port that allows the counter to be synchronously loaded, a **`clear`** input port that synchronously clears the counter, a **`din`** input port that allows values to be loaded into the counter, and an output port **`dout`** that presents the current value of the counter to the outside world.

The architecture for the counter contains two processes. The process labeled **`synch`** is the process that maintains the current state of the counter. It is the process that is clocked by the clock and transfers the new calculated output **count_val** to the current output **dout**.

The other process contains a single **IF** statement that determines whether the counter is being loaded, cleared, or is counting up.

A sample synthesized output is shown in Figure 10-9. In this example, the generated results are as expected. The left side of the schematic shows the inputs to the counter; the right side of the schematic has the counter output. Notice that the design contains four flip-flops (**FDSR1**), exactly as specified. Also, notice that the logic generated for the counter is very small. This design was optimized for area; thus, the number of levels of logic are probably higher than a design optimized for speed.

Four-Bit Shifter

Another sequential example is a 4-bit shifter. This shifter can be loaded with a value and can be shifted left or right one bit at a time. Following is the model for the shifter:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
PACKAGE shift_types IS
   SUBTYPE bit4 IS std_logic_vector(3 downto 0);
END shift_types;


USE WORK.shift_types.ALL;
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY shifter IS
  PORT( din : IN bit4;
        clk, load, left_right : IN std_logic;
        dout : INOUT bit4);
END shifter;

ARCHITECTURE synth OF shifter IS
  SIGNAL shift_val : bit4;
BEGIN
  nxt: PROCESS(load, left_right, din, dout)
  BEGIN
   IF (load = '1') THEN
     shift_val <= din;
   ELSEIF (left_right = '0') THEN
     shift_val(2 downto 0) <= dout(3 downto 1);
     shift_val(3) <= '0';
   ELSE
     shift_val(3 downto 1) <= dout(2 downto 0);
```

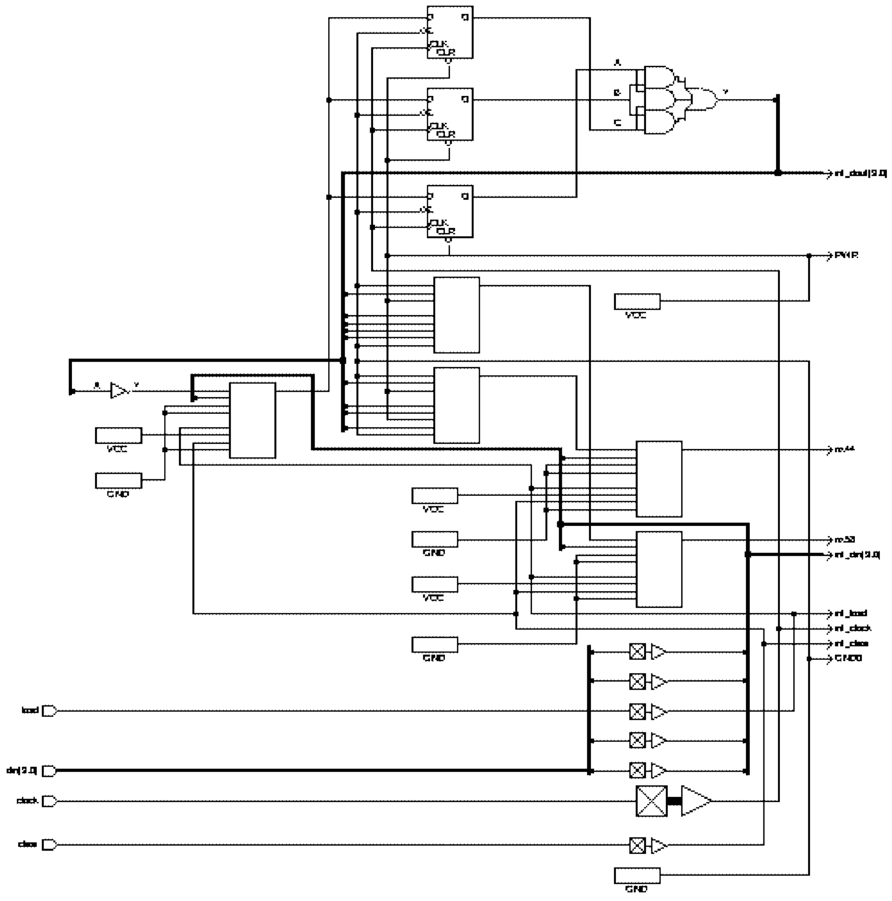


Figure 10-9
A Sample Synthesized Output.

```
      shift_val(0) <= '0';
    END IF;
  END PROCESS;

  current: PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1';

    dout <= shift_val;
  END PROCESS;
END synth;
```

The 4-bit type used for the input and output of the shifter is declared in package **`shift_types`**. This package is used by entity **`shifter`** to declare ports **`din`** and **`dout`**. Ports **`clk`**, **`load`**, and **`left_right`** are **`std_logic`** signals used to control the functions of the shifter.

The architecture is organized similarly to the last example, with two processes used to describe the functionality of the architecture. One process keeps track of the current value of the shifter, and the other calculates the next value based on the last value and the control inputs.

Process **`current`** is used to keep track of the current value of the shifter. It is a process that has a single **`WAIT`** statement and a single signal assignment statement. When the **`clk`** signal has a rising edge occur, the signal assignment statement is activated and the next calculated value of the shifter (**`shift_val`**) is written to the signal that holds the current state of the shifter (**`dout`**).

Process **`nxt`** is used to calculate the next value of **`shift_val`** to be written into **`dout`**. **`Load`** is the highest priority input and, if equal to **`'1'`**, causes **`shift_val`** to receive the value of **`din`**. Otherwise, signal **`left_right`** is tested to see if the shifter is shifting left or right. Because this shifter does not contain a **`carryin`** or **`carryout`**, **`'0'`** values are written into the bits whose value has been shifted over. (A good exercise is to write a shifter that contains a **`carryin`** and **`carryout`**.)

The synthesis tool produces a schematic for this input description as shown in Figure 10-10. By counting the flip-flops (**`FDSR1`**) on the page, it can be seen that this is indeed a 4-bit shifter.

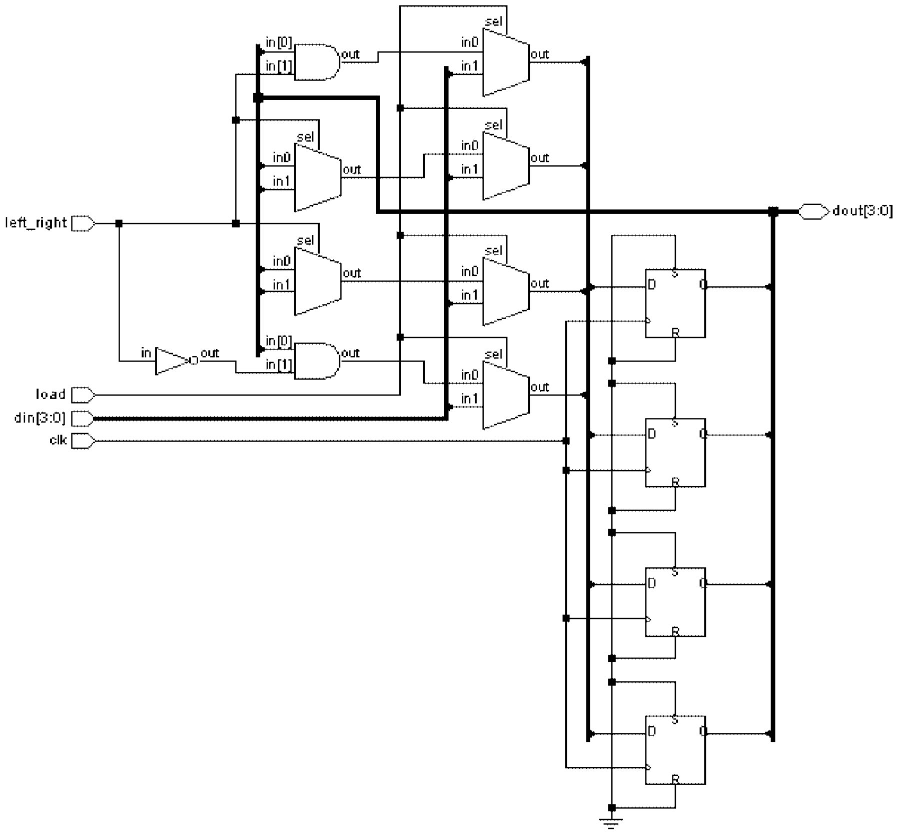
State Machine Example

The next example is a simple state machine used to control a voicemail system. (This example does not represent any real system in use and is necessarily simple to make it easier to fit in the book.) The voicemail controller allows the user to send messages, review messages, save messages, and erase messages. A state diagram showing the possible state transitions is shown in Figure 10-11.

The normal starting state is state **`main`**. From **`main`**, the user can select whether to review messages or send messages. To get to the Review menu, the user presses the 1 key on the touch-tone phone. To select the Send Message menu, the user presses the 2 key on the touch-tone phone. After the user has selected either of these options, further menu options allow the user to perform other functions such as Save and Erase. For instance, if the user first selects the Review menu by pressing key 1, then pressing key 2 allows the user to save a reviewed message when reviewing is complete.

Following is the VHDL description for the voicemail controller:

Figure 10-10
The Synthesis Tool Produces a Schematic.

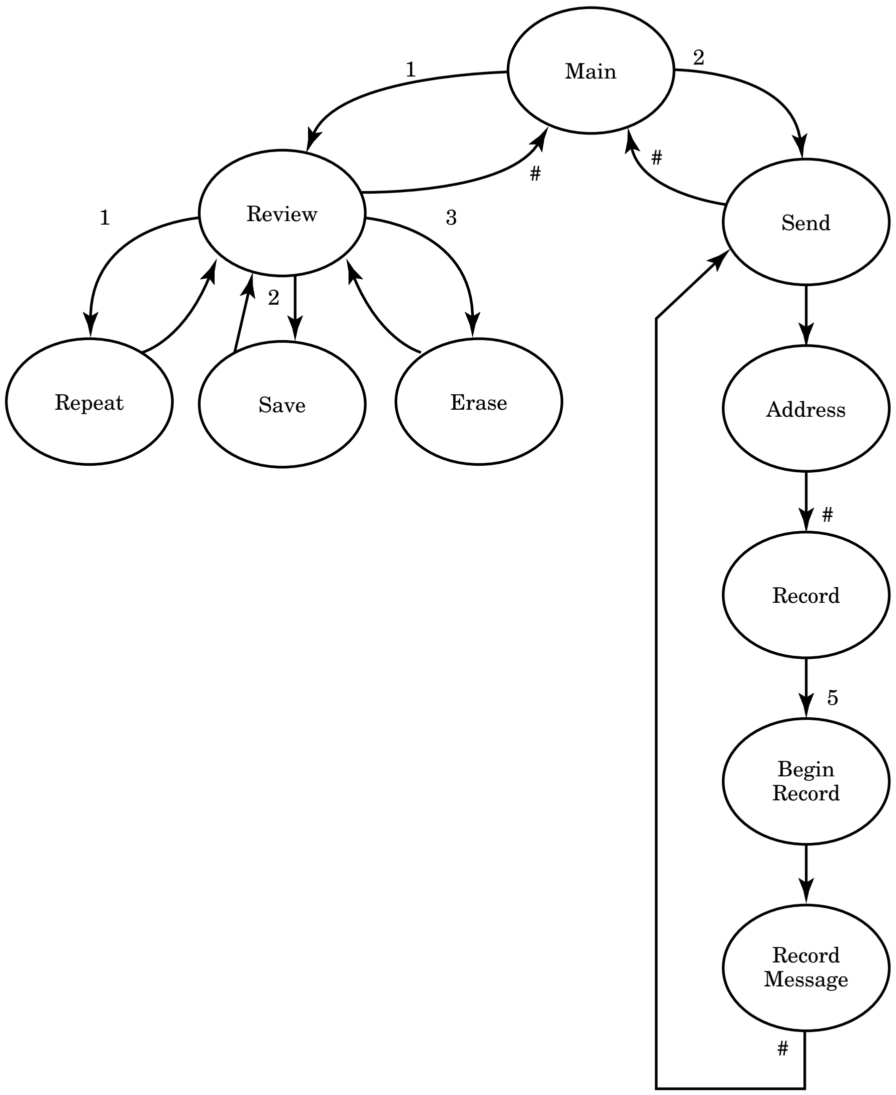


```
PACKAGE vm_pack IS
  TYPE t_vm_state IS (main_st, review_st, repeat_st,
                     save_st,
                     erase_st, send_st,
                     address_st, record_st,
                     begin_rec_st, message_st);
  TYPE t_key IS ('0','1','2','3','4','5','6','7','8','9',
                '*','#');

END vm_pack;

USE WORK.vm_pack.ALL;
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY control IS
  PORT( clk : in std_logic;
        key : in t_key;
```

Figure 10-11 State Transition Diagram for Voicemail Controller.



```
        play, recrd, erase, save, address
      : out std_logic);
END control;

ARCHITECTURE synth OF control IS
  SIGNAL next_state, current_state :
      t_vm_state;
BEGIN
```

```
 PROCESS(current_state, key)
 BEGIN
  play <= '0';
  save <= '0';
  erase <= '0';
  recrd <= '0';
  address <= '0';

CASE current_state IS
  WHEN main_st =>
    IF (key = '1') THEN
      next_state <= review_st;
    ELSEIF (key = '2') THEN
      next_state <= send_st;
    ELSE
      next_state <= main_st;
    END IF;

  WHEN review_st =>
    IF (key = '1') THEN
      next_state <= repeat_st;
    ELSEIF (key = '2') THEN
      next_state <= save_st;
    ELSEIF (key = '3') THEN
      next_state <= erase_st;
    ELSEIF (key = '#') THEN
      next_state <= main_st;
    ELSE
      next_state <= review_st;
    END IF;

  WHEN repeat_st =>
    play <= '1';
    next_state <= review_st;

  WHEN save_st =>
    save <= '1';
    next_state <= review_st;

  WHEN erase_st =>
    erase <= '1';
    next_state <= review_st;

  WHEN send_st =>
    next_state <= address_st;

  WHEN address_st =>
    address <= '1';
    IF (key = '#') THEN
      next_state <= record_st;
    ELSE
      next_state <= address_st;
```

```
        END IF;

      WHEN record_st =>
        IF (key = '5') THEN
          next_state <= begin_rec_st;
        ELSE
          next_state <= record_st;
        END IF;

      WHEN begin_rec_st =>
        recrd <= '1';
        next_state <= message_st;

      WHEN message_st =>
          recrd <= '1';
          IF (key = '#') THEN
            next_state <= send_st;
          ELSE
            next_state <= message_st;
          END IF;
      END CASE;
    END PROCESS;

    PROCESS
    BEGIN
      WAIT UNTIL clk = '1' AND clk'EVENT;

      current_state <= next_state;
    END PROCESS;
  END synth;
```

Package `vm_types` contains the type declarations for the state values and keys allowed by the voicemail controller. Notice that the states are all named something meaningful as opposed to S1, S2, S3, and so on. This makes the model much more readable.

This package is used by the entity to declare local signals and the `key` input port. The entity only has one input, the `key` input, which represents the possible key values from a touch-tone phone keypad. All of the other ports of the entity are output ports (except `clk`) and are used to control the voicemail system operations.

This model uses the two-process style to describe the operation of the state machine. This style is very useful for describing state machines as one process represents the current state register, and the other process represents the next state logic.

The next state process starts by initializing all of the output signals to `'0'`. The reason for this is to provide the synthesis tool with a default value to assign the signal if the signal was not assigned in the `CASE` statement.

The rest of the next state process consists of one **CASE** statement. This **CASE** statement describes the action to occur based on the current state of the state machine and any inputs that affect the state machine. The condition that the **CASE** statement keys from is the current state. The state machine can be placed in a different state depending on the inputs that are being tested by the current state. For instance, if the current state is **main_st**, when the **key** input is **'1'**, the next state is **review_st**; when the **key** input is **'2'**, the next state is **send_st**.

When this description is synthesized using the Leonardo synthesis tool, the schematic shown in Figure 10-12 is generated. The **key** and **clk** inputs are shown coming into the left side of the schematic and outputs **save**, **recrd**, **address**, **erase**, and **play** are shown coming out of the right side of the schematic. Intermixed in the design are the state flip-flops that are used to hold the current state of the voicemail controller and the logic used to generate the next state of the controller. This type of output is indicative of state machine descriptions.

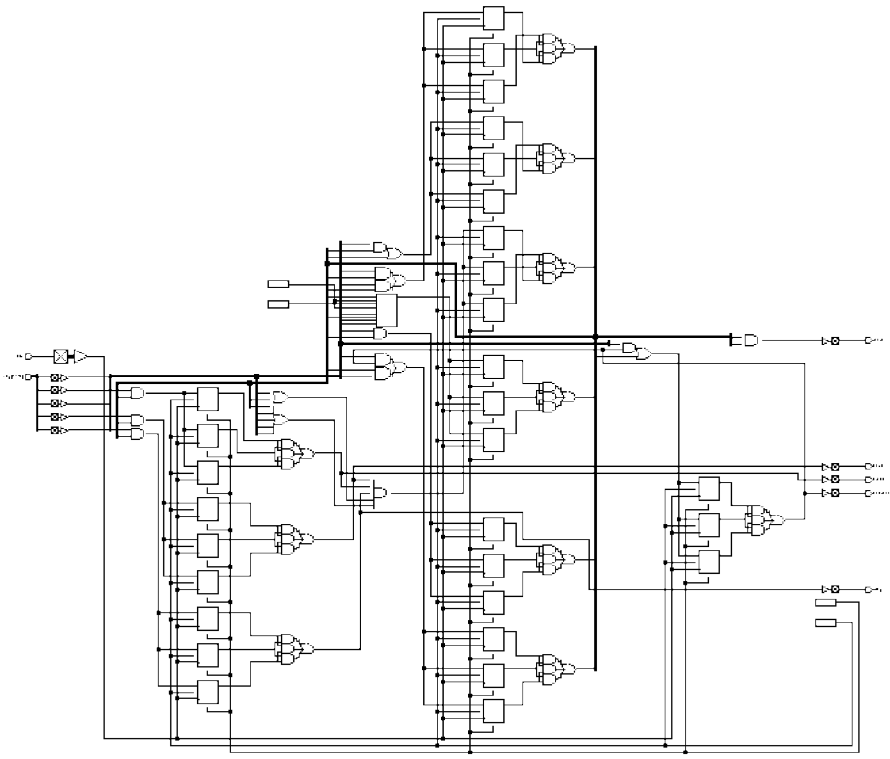


Figure 10-12
Generated Using the Leonardo Synthesis Tool.

SUMMARY

In this chapter, we looked at a number of different VHDL synthesis examples. They ranged from simple gate level descriptions to more complex examples that contained state machines. In the next few chapters, we look at a more complex example that requires a number of state machines, and we follow the process from start to finish.