Source files for examples demonstrating the use of VHDL are in the /synopsys/syn/examples/vhdl directory. The examples are

- Moore Machine
- Mealy Machine
- Read-Only Memory (ROM)
- Waveform Generator
- Smart Waveform Generator
- Definable-Width Adder-Subtractor
- Count Zeros — Combinational Version
- Count Zeros — Sequential Version
- Soft Drink Machine — State Machine Version
- Soft Drink Machine — Count Nickels Version
- Carry-Lookahead Adder
- Serial-to-Parallel Converter — Counting Bits
- Serial-to-Parallel Converter — Shifting Bits
- Programmable Logic Array (PLA)
Moore Machine

Figure A–1 is a diagram of a simple Moore finite-state machine. It has one input (X), four internal states (S0 to S3), and one output (Z).

**Figure A–1 Moore Machine Specification**

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output (Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X=0</td>
<td>X=1</td>
<td>X=0</td>
</tr>
<tr>
<td>S0</td>
<td>S0</td>
<td>S2</td>
</tr>
<tr>
<td>S1</td>
<td>S0</td>
<td>S2</td>
</tr>
<tr>
<td>S2</td>
<td>S2</td>
<td>S3</td>
</tr>
<tr>
<td>S3</td>
<td>S3</td>
<td>S1</td>
</tr>
</tbody>
</table>

The VHDL code implementing this finite-state machine is shown in Example A–1, which includes a schematic of the synthesized circuit.

The machine is described with two processes. One process defines the synchronous elements of the design (state registers); the other process defines the combinational part of the design (state assignment case statement). See the discussion under “wait Statement” in Chapter 6 for more details on using the two processes.
Example A–1  Implementation of a Moore Machine

entity MOORE is               -- Moore machine
  port(X, CLOCK: in BIT;
       Z: out BIT);
end;

architecture BEHAVIOR of MOORE is
  type STATE_TYPE is (S0, S1, S2, S3);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
begin

  -- Process to hold combinational logic
  COMBIN: process(CURRENT_STATE, X)
  begin
    case CURRENT_STATE is
    when S0 =>
      Z <= '0';
      if X = '0' then
        NEXT_STATE <= S0;
      else
        NEXT_STATE <= S2;
      end if;
    when S1 =>
      Z <= '1';
      if X = '0' then
        NEXT_STATE <= S0;
      else
        NEXT_STATE <= S2;
      end if;
    when S2 =>
      Z <= '1';
      if X = '0' then
        NEXT_STATE <= S2;
      else
        NEXT_STATE <= S3;
      end if;
    when S3 =>
      Z <= '0';
      if X = '0' then
        NEXT_STATE <= S3;
      else
        NEXT_STATE <= S1;
      end if;
    end case;
  end process;
-- Process to hold synchronous elements (flip-flops)
SYNCH: process
begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
end process;
end BEHAVIOR;

Example A-1 (continued) Implementation of a Moore Machine
Mealy Machine

Figure A–2 is a diagram of a simple Mealy finite-state machine. The VHDL code to implement this finite-state machine is shown in Example A–2. The machine is described in two processes, like the previous Moore machine example.

**Figure A–2  Mealy Machine Specification**

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output (Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X=0 X=1</td>
<td>X=0 X=1</td>
</tr>
<tr>
<td>S0</td>
<td>S0 S2</td>
<td>0 1</td>
</tr>
<tr>
<td>S1</td>
<td>S0 S2</td>
<td>0 0</td>
</tr>
<tr>
<td>S2</td>
<td>S2 S3</td>
<td>1 0</td>
</tr>
<tr>
<td>S3</td>
<td>S3 S1</td>
<td>0 1</td>
</tr>
</tbody>
</table>
Example A–2  Implementation of a Mealy Machine

entity MEALY is  -- Mealy machine
  port(X, CLOCK: in BIT;
       Z: out BIT);
end;

architecture BEHAVIOR of MEALY is
  type STATE_TYPE is (S0, S1, S2, S3);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
begin
  -- Process to hold combinational logic.
  COMBIN: process(CURRENT_STATE, X)
  begin
    case CURRENT_STATE is
      when S0 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= S0;
        else
          Z <= '1';
          NEXT_STATE <= S2;
        end if;
      when S1 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= S0;
        else
          Z <= '0';
          NEXT_STATE <= S2;
        end if;
      when S2 =>
        if X = '0' then
          Z <= '1';
          NEXT_STATE <= S2;
        else
          Z <= '0';
          NEXT_STATE <= S3;
        end if;
      when S3 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= S3;
        else
          Z <= '1';
    end case;
  end process;
end;
NEXT_STATE <= S1;
end if;
end case;
end process;

-- Process to hold synchronous elements (flip-flops)
SYNCH: process
begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
end process;
end BEHAVIOR;

Example A-2 (continued) Implementation of a Mealy Machine
Read-Only Memory (ROM)

Example A–3 shows how a ROM can be defined in VHDL. The ROM is defined as an array constant, ROM. Each line of the constant array specification defines the contents of one ROM address. To read from the ROM, simply index into the array.

The ROM’s number of storage locations and bit width can be easily changed. The subtype ROM_RANGE specifies that the ROM contains storage locations 0 to 7. The constant ROM_WIDTH specifies that the ROM is five bits wide.

After you define a ROM constant, you can index into that constant many times to read many values from the ROM. If the ROM address is computable (see “Computable Operands” in Chapter 5), no logic is built. The appropriate data value is simply inserted. If the ROM address is not computable, logic is built for each index into the value. For this reason, you need to consider resource sharing when using a ROM (see Chapter 9, “Resource Sharing”). In the example, ADDR is not computable, so logic is synthesized to compute the value.

VHDL Compiler does not actually instantiate a typical array-logic ROM, such as those available from ASIC vendors. Instead, the ROM is created from random logic gates (AND, OR, NOT, and so on). This type of implementation is preferable for small ROMs, or for ROMs that are very regular. For very large ROMs, consider using an array-logic implementation supplied by your ASIC vendor.

Example A–3 shows the VHDL source code and the synthesized circuit schematic.
Example A–3  Implementation of a ROM in Random Logic

package ROMS is
  -- declare a 5x8 ROM called ROM
  constant ROM_WIDTH: INTEGER := 5;
  subtype ROM_WORD is BIT_VECTOR (1 to ROM_WIDTH);
  subtype ROM_RANGE is INTEGER range 0 to 7;
  type ROM_TABLE is array (0 to 7) of ROM_WORD;
  constant ROM: ROM_TABLE := ROM_TABLE'(
    ROM_WORD'("10101"),  -- ROM contents
    ROM_WORD'("10000"),
    ROM_WORD'("11111"),
    ROM_WORD'("11111"),
    ROM_WORD'("10000"),
    ROM_WORD'("10101"),
    ROM_WORD'("11111"),
    ROM_WORD'("11111"));
end ROMS;

use work.ROMS.all;  -- Entity that uses ROM

entity ROM_5x8 is
  port(ADDR: in ROM_RANGE;
       DATA: out ROM_WORD);
end;

architecture BEHAVIOR of ROM_5x8 is
begin
  DATA <= ROM(ADDR);  -- Read from the ROM
end BEHAVIOR;
Waveform Generator

This example shows how to use the previous ROM example to implement a waveform generator.

Assume you want to produce the waveform output shown in Figure A–3. First, declare a ROM wide enough to hold the output signals (four bits), and deep enough to hold all time steps (0 to 12, for a total of 13).

Next, define the ROM so that each time step is represented by an entry in the ROM.

Finally, create a counter that cycles through the time steps (ROM addresses), generating the waveform at each time step.

Figure A–3  Waveform Example

Example A–4 shows an implementation for the waveform generator. It consists of a ROM, a counter, and some simple reset logic.
Example A–4 Implementation of a Waveform Generator

package ROMS is
   -- a 4x13 ROM called ROM that contains the waveform
   constant ROM_WIDTH: INTEGER := 4;
   subtype ROM_WORD is BIT_VECTOR (1 to ROM_WIDTH);
   subtype ROM_RANGE is INTEGER range 0 to 12;
   type ROM_TABLE is array (0 to 12) of ROM_WORD;
   constant ROM: ROM_TABLE := ROM_TABLE'(
      "1100", -- time step 0
      "1100", -- time step 1
      "0100", -- time step 2
      "0000", -- time step 3
      "0110", -- time step 4
      "0101", -- time step 5
      "0111", -- time step 6
      "1100", -- time step 7
      "0100", -- time step 8
      "0000", -- time step 9
      "0110", -- time step 10
      "0101", -- time step 11
      "0111");  -- time step 12
end ROMS;

use work.ROMS.all;
entity WAVEFORM is            -- Waveform generator
   port(CLOCK: in BIT;
        RESET: in BOOLEAN;
        WAVES: out ROM_WORD);
end;

architecture BEHAVIOR of WAVEFORM is
   signal STEP: ROM_RANGE;
begin
   TIMESTEP_COUNTER: process   -- Time stepping process
      begin
         wait until CLOCK’event and CLOCK = ’1’;
         if RESET then               -- Detect reset
            STEP <= ROM_RANGE’low;  -- Restart
         elsif STEP = ROM_RANGE’high then   -- Finished?
            STEP <= ROM_RANGE’high; -- Hold at last value
         else
            STEP <= STEP + 1;        -- Continue stepping
         end if;
      end process;

end;
end process TIMESTEP_COUNTER;

    WAVES <= ROM(STEP);
end BEHAVIOR;

Example A-4 (continued)  Implementation of a Waveform Generator

Note that when the counter \texttt{STEP} reaches the end of the ROM, \texttt{STEP} stops, generates the last value, then waits until a reset. To make the sequence automatically repeat, remove the statement:

\texttt{STEP <= ROM\_RANGE'high;}  -- Hold at last value

and use the following statement instead (commented out in Example A–4):

\texttt{STEP <= ROM\_RANGE'low;}  -- Continuous wave
Smart Waveform Generator

This example is an extension of the waveform generator in the previous example. This smart waveform generator is capable of holding the waveform at any time step for several clock cycles.

Figure A–4 shows a waveform similar to the waveform of the previous example, where several of the time steps are held for multiple clock cycles.

![Waveform for Smart Waveform Generator Example](image)

The implementation of the smart waveform generator is shown in Example A–5. It is similar to the waveform generator of the previous example, but with two additions. A new ROM, \( D_{\text{ROM}} \), has been added to hold the length of each time step. A value of 1 specifies that the corresponding time step should be one clock cycle long; a value of 80 specifies that the time step should be 80 clock cycles long. The second addition to the previous waveform generator is a delay counter that counts out the clock cycles between time steps.

Note that in the architecture of this example, a selected signal assignment determines the value of the \( \text{NEXT\_STEP} \) counter.
Example A–5  Implementation of a Smart Waveform Generator

package ROMS is

-- a 4x13 ROM called W_ROM containing the waveform
constant W_ROM_WIDTH: INTEGER := 4;
subtype W_ROM_WORD is BIT_VECTOR (1 to W_ROM_WIDTH);
subtype W_ROM_RANGE is INTEGER range 0 to 12;
type W_ROM_TABLE is array (0 to 12) of W_ROM_WORD;
constant W_ROM: W_ROM_TABLE := W_ROM_TABLE'(
  "1100",   -- time step 0
  "1100",   -- time step 1
  "0100",   -- time step 2
  "0000",   -- time step 3
  "0110",   -- time step 4
  "0101",   -- time step 5
  "0111",   -- time step 6
  "1100",   -- time step 7
  "0100",   -- time step 8
  "0000",   -- time step 9
  "0110",   -- time step 10
  "0101",   -- time step 11
  "0111");  -- time step 12

-- a 7x13 ROM called D_ROM containing the delays
subtype D_ROM_WORD is INTEGER range 0 to 100;
subtype D_ROM_RANGE is INTEGER range 0 to 12;
type D_ROM_TABLE is array (0 to 12) of D_ROM_WORD;
constant D_ROM: D_ROM_TABLE := D_ROM_TABLE’(1,80,5,1,1,1,1,20,5,1,1,1);
end ROMS;

use work.ROMS.all;
entity WAVEFORM is        -- Smart Waveform Generator
port(CLOCK: in BIT;
    RESET: in BOOLEAN;
    WAVES: out W_ROM_WORD);
end;

architecture BEHAVIOR of WAVEFORM is
  signal STEP, NEXT_STEP: W_ROM_RANGE;
  signal DELAY: D_ROM_WORD;
begin

  -- Determine the value of the next time step
  NEXT_STEP <= W_ROM_RANGE’high when
STEP = W_ROM_RANGE’high
else
    \text{STEP + 1;}
-- Keep track of which time step we are in
TIMESTEP_COUNTER: process
begin
    wait until CLOCK’event and CLOCK = ’1’;
    if RESET then
        \text{-- Detect reset}
        \text{STEP <= 0; \quad -- Restart waveform}
    elsif DELAY = 1 then
        \text{STEP <= NEXT_STEP; \quad -- Continue stepping}
    else
        null; \quad -- Wait for DELAY to count down;
    end if;
end process;

-- Count the delay between time steps.
DELAY_COUNTER: process
begin
    wait until CLOCK’event and CLOCK = ’1’;
    if RESET then
        DELAY <= D_ROM(0); \quad -- Restart
    elsif DELAY = 1 then
        DELAY <= D_ROM(NEXT_STEP); \quad -- Next delay value
    else
        DELAY <= DELAY - 1; \quad -- decrement DELAY counter
    end if;
end process;

WAVES <= W_ROM(STEP); \quad -- Output waveform value
end BEHAVIOR;
Definable-Width Adder-Subtractor

VHDL lets you create functions for use with array operands of any size. This example shows an adder-subtractor circuit that, when called, is adjusted to fit the size of its operands.

Example A–6 shows an adder-subtractor defined for two unconstrained arrays of bits (type BIT_VECTOR), in a package named MATH. When an unconstrained array type is used for an argument to a subprogram, the actual constraints of the array are taken from the actual parameter values in a subprogram call.

Example A–7 shows how to use the adder-subtractor defined in the MATH package. In this example the vector arguments to functions ARG1 and ARG2 are declared as BIT_VECTOR(1 to 6). This declaration causes ADD_SUB to work with six-bit arrays. A schematic of the synthesized circuit follows.

Example A–6 MATH Package for Example A–7

package MATH is
  function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN) return BIT_VECTOR;
  -- Add or subtract two BIT_VECTORs of equal length
end MATH;

package body MATH is
  function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN) return BIT_VECTOR is
    variable CARRY: BIT;
    variable A, B, SUM:
      BIT_VECTOR(L'length–1 downto 0);
    begin
      if ADD then
        -- Prepare for an “add” operation
        A := L;
        B := R;
        CARRY := ’0’;
      else
        -- Prepare for a “subtract” operation
        A := R;
        B := L;
        CARRY := ’1’;
      end if;
      for K in 0 to L'length–1 loop
        SUM(K) := A(K) xor B(K) xor CARRY;
        if A(K) = B(K) then
          CARRY := A(K) and B(K);
        else
          CARRY := not A(K) and not B(K);
        end if;
      end loop;
      return SUM;
    end function;
end package body MATH;
-- Prepare for a "subtract" operation
A := L;
B := not R;
CARRY := '1';
end if;

-- Create a ripple-carry chain; sum up bits
for i in 0 to A'left loop
  SUM(i) := A(i) xor B(i) xor CARRY;
  CARRY := (A(i) and B(i)) or
           (A(i) and CARRY) or
           (CARRY and B(i));
end loop;
return SUM;  -- Result
end;
end MATH;

Within the function ADD_SUB, two temporary variables, A and B, are declared. These variables are declared to be the same length as L (and necessarily, R), but have their index constraints normalized to L’length–1 downto 0. After the arguments are normalized, you can create a ripple-carry adder by using a for loop.

Note that no explicit references to a fixed array length are in the function ADD_SUB. Instead, the VHDL array attributes ‘left and ’length are used. These attributes allow the function to work on arrays of any length.
Example A-7  Implementation of a Six-Bit Adder-Subtractor

use work.MATH.all;

entity EXAMPLE is
  port(ARG1, ARG2: in BIT_VECTOR(1 to 6);
     ADD: in BOOLEAN;
     RESULT : out BIT_VECTOR(1 to 6));
end EXAMPLE;

architecture BEHAVIOR of EXAMPLE is
begin
  RESULT <= ADD_SUB(ARG1, ARG2, ADD);
end BEHAVIOR;
Count Zeros—Combinational Version

This example illustrates a design problem where an eight-bit-wide value is given, and the circuit determines two things:

- That no more than one sequence of 0s is in the value.
- The number of 0s in that sequence (if any). This computation must be completed in a single clock cycle.

The circuit produces two outputs: the number of zeros found, and an error indication.

A legal input value can have at most one consecutive series of zeros. A value consisting entirely of ones is defined as a legal value. If a value is illegal, the zero counter resets to 0. For example, the value 00000000 is legal and has eight zeros; value 11000111 is legal and has three zeros; value 00111100 is not legal.

Example A–8 shows the VHDL description for the circuit. It consists of a single process with a for loop that iterates across each bit in the given value. At each iteration, a temporary INTEGER variable (TEMP_COUNT) counts the number of zeros encountered. Two temporary BOOLEAN variables (SEEN_ZERO and SEEN_TRAILING), initially FALSE, are set to TRUE when the beginning and end of the first sequence of zeros is detected.

If a zero is detected after the end of the first sequence of zeros (after SEEN_TRAILING is TRUE), the zero–count is reset (to 0), ERROR is set to TRUE, and the for loop is exited.

This example shows a combinational (parallel) approach to counting the zeros. The next example shows a sequential (serial) approach.
Example A–8  Count Zeros—Combinational

entity COUNT_COMB_VHDL is
  port(DATA:  in  BIT_VECTOR(7 downto 0);
       COUNT: out  INTEGER range 0 to 8;
       ERROR: out BOOLEAN);
end;

architecture BEHAVIOR of COUNT_COMB_VHDL is
begin
  process(DATA)
  variable TEMP_COUNT : INTEGER range 0 to 8;
  variable SEEN_ZERO, SEEN_TRAILING : BOOLEAN;
  begin
    ERROR <= FALSE;
    SEEN_ZERO := FALSE;
    SEEN_TRAILING := FALSE;
    TEMP_COUNT := 0;
    for I in 0 to 7 loop
      if (SEEN_TRAILING and DATA(I) = '0') then
        TEMP_COUNT := 0;
        ERROR <= TRUE;
        exit;
      elsif (SEEN_ZERO and DATA(I) = '1') then
        SEEN_TRAILING := TRUE;
      elsif (DATA(I) = '0') then
        SEEN_ZERO := TRUE;
        TEMP_COUNT := TEMP_COUNT + 1;
      end if;
    end loop;
    COUNT <= TEMP_COUNT;
  end process;
end BEHAVIOR;
Example A-8 (continued)  Count Zeros—Combinational
Count Zeros—Sequential Version

This example shows a sequential (clocked) variant of the preceding design (Count Zeros—Combinational Version).

The circuit now accepts the eight-bit data value serially, one bit per clock cycle, by using the **DATA** and **CLK** inputs. The other two inputs are

- **RESET**, which resets the circuit.
- **READ**, which causes the circuit to begin accepting data bits.

The circuit’s three outputs are

- **IS_LEGAL**, which is **TRUE** if the data was a legal value.
- **COUNT_READY**, which is **TRUE** at the first illegal bit or when all eight bits have been processed.
- **COUNT**, the number of zeros (if **IS_LEGAL** is **TRUE**).

Note that the output port **COUNT** is declared with mode **BUFFER** so that it can be read inside the process. **OUT** ports can only be written to, not read.

**Example A–9  Count Zeros—Sequential**

```vhdl
entity COUNT_SEQ_VHDL is
  port(DATA, CLK: in BIT;
       RESET, READ: in BOOLEAN;
       COUNT: buffer INTEGER range 0 to 8;
       IS_LEGAL: out BOOLEAN;
       COUNT_READY: out BOOLEAN);
  end;
```
architecture BEHAVIOR of COUNT_SEQ_VHDL is
begin
    process
        variable SEEN_ZERO, SEEN_TRAILING: BOOLEAN;
        variable BITS_SEEN: INTEGER range 0 to 7;
    begin
        wait until CLK'event and CLK = '1';

        if(RESET) then
            COUNT_READY <= FALSE;
            IS_LEGAL <= TRUE;
            SEEN_ZERO := FALSE;
            SEEN_TRAILING := FALSE;
            COUNT <= 0;
            BITS_SEEN := 0;
        else
            if (READ) then
                if (SEEN_TRAILING and DATA = '0') then
                    IS_LEGAL <= FALSE;
                    COUNT <= 0;
                    COUNT_READY <= TRUE;
                elsif (SEEN_ZERO and DATA = '1') then
                    SEEN_TRAILING := TRUE;
                elsif (DATA = '0') then
                    SEEN_ZERO := TRUE;
                    COUNT <= COUNT + 1;
                end if;

                if (BITS_SEEN = 7) then
                    COUNT_READY <= TRUE;
                else
                    BITS_SEEN := BITS_SEEN + 1;
                end if;

            end if;  -- if (READ)
        end if;  -- if (RESET)
    end process;
end BEHAVIOR;
Example A-9 (continued)  Count Zeros—Sequential
Soft Drink Machine—State Machine Version

This example is a control unit for a soft drink vending machine. The circuit reads signals from a coin input unit and sends outputs to a change dispensing unit and a drink dispensing unit. This example assumes that only one kind of soft drink is dispensed.

This is a clocked design with \texttt{CLK} and \texttt{RESET} input signals.

The price of the drink is 35 cents. Input signals from the coin input unit are \texttt{NICKEL\_IN} (nickel deposited), \texttt{DIME\_IN} (dime deposited), and \texttt{QUARTER\_IN} (quarter deposited).

Output signals to the change dispensing unit are \texttt{NICKEL\_OUT} and \texttt{DIME\_OUT}.

The output signal to the drink dispensing unit is \texttt{DISPENSE} (dispense drink).

The first VHDL description for this design uses a state machine description style. The second VHDL description is in the next example section.

Example A–10  Soft Drink Machine—State Machine

```vhdl
library synopsys; use synopsys.attributes.all;

entity DRINK\_STATE\_VHDL is
  port(NICKEL\_IN, DIME\_IN, QUARTER\_IN, RESET: BOOLEAN;
    CLK: BIT;
    NICKEL\_OUT, DIME\_OUT, DISPENSE: out BOOLEAN);
end;

architecture BEHAVIOR of DRINK\_STATE\_VHDL is
  type STATE\_TYPE is (IDLE, FIVE, TEN, FIFTEEN,
    TWENTY, TWENTY\_FIVE, THIRTY, OWE\_DIME);
  signal CURRENT\_STATE, NEXT\_STATE: STATE\_TYPE;
  attribute STATE\_VECTOR : STRING;
  attribute STATE\_VECTOR of BEHAVIOR : architecture is "CURRENT\_STATE";
```

For further assistance, email support\_center@synopsys.com or call your local support center.
attribute sync_sync_reset of reset : signal is "true";
begin

process(NICKEL_IN, DIME_IN, QUARTER_IN,
        CURRENT_STATE, RESET, CLK)
begin
    -- Default assignments
    NEXT_STATE <= CURRENT_STATE;
    NICKEL_OUT <= FALSE;
    DIME_OUT <= FALSE;
    DISPENSE <= FALSE;

    -- Synchronous reset
    if(RESET) then
        NEXT_STATE <= IDLE;
    else

    -- State transitions and output logic
    case CURRENT_STATE is
    when IDLE =>
        if(NICKEL_IN) then
            NEXT_STATE <= FIVE;
        elsif(DIME_IN) then
            NEXT_STATE <= TEN;
        elsif(QUARTER_IN) then
            NEXT_STATE <= TWENTY_FIVE;
        end if;
    when FIVE =>
        if(NICKEL_IN) then
            NEXT_STATE <= TEN;
        elsif(DIME_IN) then
            NEXT_STATE <= FIFTEEN;
        elsif(QUARTER_IN) then
            NEXT_STATE <= THIRTY;
        end if;
    when TEN =>
        if(NICKEL_IN) then
            NEXT_STATE <= FIFTEEN;
        elsif(DIME_IN) then
            NEXT_STATE <= TWENTY;
        elsif(QUARTER_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
        end if;
    when FIFTEEN =>

end case;
end if;
end process;

if(NICKEL_IN) then
    NEXT_STATE <= TWENTY;
elsif(DIME_IN) then
    NEXT_STATE <= TWENTY_FIVE;
elsif(QUARTER_IN) then
    NEXT_STATE <= IDLE;
    DISPENSE <= TRUE;
    NICKEL_OUT <= TRUE;
end if;

when TWENTY =>
    if(NICKEL_IN) then
        NEXT_STATE <= TWENTY_FIVE;
    elsif(DIME_IN) then
        NEXT_STATE <= THIRTY;
    elsif(QUARTER_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
        DIME_OUT <= TRUE;
    end if;

when TWENTY_FIVE =>
    if(NICKEL_IN) then
        NEXT_STATE <= THIRTY;
    elsif(DIME_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
    elsif(QUARTER_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
        DIME_OUT <= TRUE;
        NICKEL_OUT <= TRUE;
    end if;

when THIRTY =>
    if(NICKEL_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
    elsif(DIME_IN) then
        NEXT_STATE <= IDLE;
        DISPENSE <= TRUE;
        NICKEL_OUT <= TRUE;
    elsif(QUARTER_IN) then
        NEXT_STATE <= OWE_DIME;
        DISPENSE <= TRUE;
        DIME_OUT <= TRUE;
end if;

when OWE_DIME =>
  NEXT_STATE <= IDLE;
  DIME_OUT <= TRUE;

  end case;
  end if;
end process;

-- Synchronize state value with clock.
-- This causes it to be stored in flip flops
process
begin
  wait until CLK'event and CLK = '1';
  CURRENT_STATE <= NEXT_STATE;
end process;

end BEHAVIOR;
Soft Drink Machine—Count Nickels Version

This example uses the same design parameters as the preceding example (Soft Drink Machine — State Machine Version) with the same input and output signals.

In this version, a counter counts the number of nickels deposited. This counter is incremented by 1 if the deposit is a nickel, by 2 if it is a dime, and by 5 if it is a quarter.

Example A–11  Soft Drink Machine—Count Nickels

entity DRINK_COUNT_VHDL is
  port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET: BOOLEAN;
    CLK: BIT;
    NICKEL_OUT, DIME_OUT, DISPENSE: out BOOLEAN);
end;

architecture BEHAVIOR of DRINK_COUNT_VHDL is
  signal CURRENT_NICKEL_COUNT,
    NEXT_NICKEL_COUNT: INTEGER range 0 to 7;
  signal CURRENT_RETURN_CHANGE, NEXT_RETURN_CHANGE : BOOLEAN;
begin

  process(NICKEL_IN, DIME_IN, QUARTER_IN, RESET, CLK,
    CURRENT_NICKEL_COUNT, CURRENT_RETURN_CHANGE)
  variable TEMP_NICKEL_COUNT: INTEGER range 0 to 12;
begin

    -- Default assignments
    NICKEL_OUT <= FALSE;
    DIME_OUT <= FALSE;
    DISPENSE <= FALSE;
    NEXT_NICKEL_COUNT <= 0;
    NEXT_RETURN_CHANGE <= FALSE;

    -- Synchronous reset
    if (not RESET) then
      TEMP_NICKEL_COUNT := CURRENT_NICKEL_COUNT;
    end if;

    -- Check whether money has come in
    if (NICKEL_IN) then
      -- NOTE:  This design will be flattened, so
      -- these multiple adders will be optimized
      TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT + 1;
elsif(DIME_IN) then
    TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT + 2;
elsif(QUARTER_IN) then
    TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT + 5;
end if;

-- Enough deposited so far?
if(TEMP_NICKEL_COUNT >= 7) then
    TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT – 7;
    DISPENSE <= TRUE;
end if;

-- Return change
if(TEMP_NICKEL_COUNT >= 1 or 
    CURRENT_RETURN_CHANGE) then
    if(TEMP_NICKEL_COUNT >= 2) then
        DIME_OUT <= TRUE;
        TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT – 2;
        NEXT_RETURN_CHANGE <= TRUE;
    end if;
    if(TEMP_NICKEL_COUNT = 1) then
        NICKEL_OUT <= TRUE;
        TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT – 1;
    end if;
end if;
Example A-11 (continued) Soft Drink Machine—Count Nickels

NEXT_NICKEL_COUNT <= TEMP_NICKEL_COUNT;
end if;
end process;

-- Remember the return-change flag and
-- the nickel count for the next cycle
process
begin
  wait until CLK'event and CLK = '1';
  CURRENT_RETURN_CHANGE <= NEXT_RETURN_CHANGE;
  CURRENT_NICKEL_COUNT <= NEXT_NICKEL_COUNT;
end process;

end BEHAVIOR;
Carry-Lookahead Adder

This example uses concurrent procedure calls to build a 32-bit carry-lookahead adder. The adder is built by partitioning the 32-bit input into eight slices of four bits each. Each of the eight slices computes propagate and generate values by using the PG procedure. Figure A–5 shows the overall structure.

Propagate (output P from PG) is ‘1’ for a bit position if that position propagates a carry from the next lower position to the next higher position. Generate (output G) is ‘1’ for a bit position if that position generates a carry to the next higher position, regardless of the carry-in from the next lower position.

The carry-lookahead logic reads the carry-in, propagate, and generate information computed from the inputs. It computes the carry value for each bit position. This logic makes the addition operation just an XOR of the inputs and the carry values.

Carry Value Computations

The carry values are computed by a three-level tree of four-bit carry-lookahead blocks.

1. The first level of the tree computes the 32 carry values and the eight group-propagate and generate values. Each of the first-level group-propagate and generate values tells if that four-bit slice propagates and generates carry values from the next lower group to the next higher. The first-level lookahead blocks read the group carry computed at the second level.
2. The second-level lookahead blocks read the group-propagate and generate information from the four first-level blocks, then compute their own group-propagate and generate information. They also read group carry information computed at the third level to compute the carries for each of the third-level blocks.

3. The third-level block reads the propagate and generate information of the second level to compute a propagate and generate value for the entire adder. It also reads the external carry to compute each second-level carry. The carry-out for the adder is ‘1’ if the third-level generate is ‘1’, or if the third-level propagate is ‘1’ and the external carry is ‘1’.

The third-level carry-lookahead block is capable of processing four second-level blocks. Since there are only two, the high-order two bits of the computed carry are ignored, the high-order two bits of the generate input to the third-level are set to zero 00, and the propagate high-order bits are set to 11. These settings cause the unused portion to propagate carries, but not to generate them.
Figure A–5  Carry-Lookahead Adder Block Diagram (shown on next page)
The VHDL implementation of the design in Figure A–5 is done with four procedures:

**CLA**  a four-bit carry-lookahead block.

**PG**  computes first-level propagate and generate information.

**SUM**  computes the sum by XORing the inputs with the carry values computed by CLA.

**BITSLICE**

- collects the first-level CLA blocks, the PG computations, and the SUM. This procedure performs all the work for a four-bit value except for the second- and third-level lookaheads.

Example A–12 shows a VHDL description of the adder.

**Example A–12  Carry-Lookahead Adder**

```vhdl
package LOCAL is
  constant N: INTEGER := 4;

procedure BITSLICE(
  A, B: in BIT_VECTOR(3 downto 0);
  CIN: in BIT;
  signal S: out BIT_VECTOR(3 downto 0);
  signal GP, GG: out BIT);

procedure PG(
  A, B: in BIT_VECTOR(3 downto 0);
  P, G: out BIT_VECTOR(3 downto 0));

function SUM(A, B, C: BIT_VECTOR(3 downto 0))
  return BIT_VECTOR;

procedure CLA(
  P, G: in BIT_VECTOR(3 downto 0);
  CIN: in BIT;
  C: out BIT_VECTOR(3 downto 0);
  signal GP, GG: out BIT);
end LOCAL;

package body LOCAL is
```

For further assistance, email support_center@synopsys.com or call your local support center.
-- Compute sum and group outputs from a, b, cin

procedure BITSLICE(
    A, B: in BIT_VECTOR(3 downto 0);
    CIN: in BIT;
    signal S: out BIT_VECTOR(3 downto 0);
    signal GP, GG: out BIT) is

    variable P, G, C: BIT_VECTOR(3 downto 0);

begin
    PG(A, B, P, G);
    CLA(P, G, CIN, C, GP, GG);
    S <= SUM(A, B, C);
end;

-- Compute propagate and generate from input bits

procedure PG(A, B: in BIT_VECTOR(3 downto 0);
    P, G: out BIT_VECTOR(3 downto 0)) is

begin
    P := A or B;
    G := A and B;
end;

-- Compute sum from the input bits and the carries

function SUM(A, B, C: BIT_VECTOR(3 downto 0)) return BIT_VECTOR is

begin
    return(A xor B xor C);
end;

-- 4-bit carry-lookahead block

procedure CLA(
    P, G: in BIT_VECTOR(3 downto 0);
CIN: in BIT;
C: out BIT_VECTOR(3 downto 0);
signal GP, GG: out BIT is

variable TEMP_GP, TEMP_GG, LAST_C: BIT;
begin
TEMP_GP := P(0);
TEMP_GG := G(0);
LAST_C := CIN;
C(0) := CIN;

for I in 1 to N–1 loop
  TEMGP := TEMGP and P(I);
  TEMGG := (TEMGG and P(I)) or G(I);
  LASTC := (LASTC and P(I–1)) or G(I–1);
  C(I) := LASTC;
end loop;

GP <= TEMP_GP;
GG <= TEMP_GG;
end; end LOCAL;

use WORK.LOCAL.ALL;

///////////////////////////////
-- A 32-bit carry-lookahead adder
///////////////////////////////

entity ADDER is
  port(A, B: in BIT_VECTOR(31 downto 0);
         CIN: in BIT;
         S: out BIT_VECTOR(31 downto 0);
         COUT: out BIT);
end ADDER;
architecture BEHAVIOR of ADDER is

  signal GG, GP, GC: BIT_VECTOR(7 downto 0);
  -- First-level generate, propagate, carry
  signal GGG, GGP, GCC: BIT_VECTOR(3 downto 0);
  -- Second-level gen, prop, carry
  signal GGGG, GGGP: BIT;
  -- Third-level gen, prop

  begin
    -- Compute Sum and 1st-level Generate and Propagate

    HOME    CONTENTS    INDEX

    For further assistance, email support_center@synopsys.com or call your local support center.
-- Use input data and the 1st-level Carries computed
-- later.
BITSLICE(A(3 downto 0), B(3 downto 0), GC(0),
S(3 downto 0), GP(0), GG(0));
BITSLICE(A(7 downto 4), B(7 downto 4), GC(1),
S(7 downto 4), GP(1), GG(1));
BITSLICE(A(11 downto 8), B(11 downto 8), GC(2),
S(11 downto 8), GP(2), GG(2));
BITSLICE(A(15 downto 12), B(15 downto 12), GC(3),
S(15 downto 12), GP(3), GG(3));
BITSLICE(A(19 downto 16), B(19 downto 16), GC(4),
S(19 downto 16), GP(4), GG(4));
BITSLICE(A(23 downto 20), B(23 downto 20), GC(5),
S(23 downto 20), GP(5), GG(5));
BITSLICE(A(27 downto 24), B(27 downto 24), GC(6),
S(27 downto 24), GP(6), GG(6));
BITSLICE(A(31 downto 28), B(31 downto 28), GC(7),
S(31 downto 28), GP(7), GG(7));

-- Compute first-level Carries and second-level
-- generate and propagate.
-- Use first-level Generate, Propagate, and
-- second-level carry.
process(GP, GG, GGC)
variable TEMP: BIT_VECTOR(3 downto 0);
begin
CLA(GP(3 downto 0), GG(3 downto 0), GGC(0), TEMP,
GGP(0), GGG(0));
GC(3 downto 0) <= TEMP;
end process;

process(GP, GG, GGC)
variable TEMP: BIT_VECTOR(3 downto 0);
begin
CLA(GP(7 downto 4), GG(7 downto 4), GC(1), TEMP,
GGP(1), GGG(1));
GC(7 downto 4) <= TEMP;
end process;

-- Compute second-level Carry and third-level
-- Generate and Propagate
-- Use second-level Generate, Propagate and Carry-in
-- (CIN)
process(GGP, GGG, CIN)
variable TEMP: BIT_VECTOR(3 downto 0);
begin

CLA(GGP, GGG, CIN, TEMP, GGGP, GGGG);
GGC <= TEMP;
end process;

-- Assign unused bits of second-level Generate and Propagate
GGP(3 downto 2) <= "11";
GGG(3 downto 2) <= "00";

-- Compute Carry-out (COUT)
-- Use third-level Generate and Propagate and Carry-in (CIN).
COUT <= GGGG or (GGGP and CIN);
end BEHAVIOR;

Implementation

In the carry-lookahead adder implementation, procedures are used to perform the computation of the design. The procedures can also be written as separate entities and used by component instantiation, producing a hierarchical design. VHDL Compiler does not collapse a hierarchy of entities, but it does collapse the procedure call hierarchy into one design.

Note that the keyword signal is included before some of the interface parameter declarations. This keyword is required for out formal parameters when the actual parameters must be signals.

The output parameter C from the CLA procedure is not declared as a signal; thus it is not allowed in a concurrent procedure call; only signals can be used in such calls. To overcome this problem, subprocesses are used, declaring a temporary variable TEMP. TEMP receives the value of the C parameter and assigns it to the appropriate signal (a generally useful technique).
Serial-to-Parallel Converter—Counting Bits

The example below shows the design of a serial-to-parallel converter that reads a serial, bit-stream input and produces an eight-bit output.

The design reads the following inputs:

SERIAL_IN
  Serial input data.

RESET
  When '1', causes the converter to reset. All outputs are set to 0, and the converter is prepared to read the next serial word.

CLOCK
  The value of the RESET and SERIAL_IN is read on the positive transition of this clock. Outputs of the converter are also valid only on positive transitions.

The design produces the following outputs:

PARALLEL_OUT
  Eight-bit value read from the SERIAL_IN port.

READ_ENABLE
  When this output is '1' on the positive transition of CLOCK, the data on PARALLEL_OUT can be read.

PARITY_ERROR
  When this output is '1' on the positive transition of CLOCK, a parity error has been detected on the SERIAL_IN port. When a parity error is detected, the converter halts until restarted by the RESET port.
Input Format

When no data is being transmitted to the serial port, keep it at a value of ‘0’. Each eight-bit value requires 10 clock cycles to read. On the 11th clock cycle, the parallel output value can be read.

In the first cycle, a ‘1’ is placed on the serial input. This assignment indicates that an eight-bit value follows. The next eight cycles are used to transmit each bit of the value. The most significant bit is transmitted first. The 10th and final cycle transmits the parity of the eight-bit value. It must be ‘0’ if an even number of ‘1’s are in the eight-bit data, and ‘1’ otherwise. If the converter detects a parity error, it sets the PARITY_ERROR output to ‘1’ and waits until it is reset.

On the 11th cycle, the READ_ENABLE output is set to ‘1’ and the eight-bit value can be read from the PARALLEL_OUT port. If the SERIAL_IN port has a ‘1’ on the 11th cycle, another eight-bit value is read immediately; otherwise, the converter waits until SERIAL_IN goes to ‘1’.

Figure A–6 shows the timing of this design.
**Implementation Details**

The converter is implemented as a four-state finite-state machine with synchronous reset. When a reset is detected, the `WAIT_FOR_START` state is entered. The description of each state is:

**WAIT_FOR_START**

Stay in this state until a ‘1’ is detected on the serial input. When a ‘1’ is detected, clear the `PARALLEL_OUT` registers and go to the `READ_BITS` state.

**READ_BITS**

If the value of the `current_bit_position` counter is 8, all eight bits have been read. Check the computed parity with the transmitted parity; if it is correct, go to the `ALLOW_READ` state, otherwise go to the `PARITY_ERROR` state.
If all eight bits have not yet been read, set the appropriate bit in the `PARALLEL_OUT` buffer to the `SERIAL_IN` value, compute the parity of the bits read so far, and increment the `current_bit_position`.

**ALLOW_READ**
- This is the state where the outside world reads the `PARALLEL_OUT` value. When that value is read, the design returns to the `WAIT_FOR_START` state.

**PARITY_ERROR_DETECTED**
- In this state the `PARITY_ERROR` output is set to `’1’` and nothing else is done.

This design has four values stored in registers:

**CURRENT_STATE**
- remembers the state as of the last clock edge.

**CURRENT_BIT_POSITION**
- remembers how many bits have been read so far.

**CURRENT_PARITY**
- keeps a running XOR of the bits read.

**CURRENT_PARALLEL_OUT**
- stores each parallel bit as it is found.

The design is divided between two processes: the combinational `NEXT_ST` containing the combinational logic, and the sequential `SYNCH` that is clocked.

`NEXT_ST` performs all the computations and state assignments.
The **NEXT_ST process** starts by first assigning default values to all the signals it drives. This assignment guarantees that all signals are driven under all conditions. Next, the **RESET** input is processed. If **RESET** is not active, a **case** statement determines the current state and its computations. State transitions are performed by assigning the next state’s value you want to the **NEXT_STATE** signal.

The serial-to-parallel conversion itself is performed by these two statements in the **NEXT_ST process**:

```vhdl
NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <= SERIAL_IN;
NEXT_BIT_POSITION <= CURRENT_BIT_POSITION + 1;
```

The first statement assigns the current serial input bit to a particular bit of the parallel output. The second statement increments the next bit position to be assigned.

**SYNCH** registers and updates the stored values described above. Each registered signal has two parts, **NEXT_**... and **CURRENT_**...:

**NEXT_**...
- signals hold values computed by the **NEXT_ST** process.

**CURRENT_**...
- signals hold the values driven by the **SYNCH** process. The **CURRENT_**... signals hold the values of the **NEXT_**... signals as of the last clock edge.

**Example A–13 Serial-to-Parallel Converter—Counting Bits**

-- Serial-to-Parallel Converter, counting bits

package TYPES is
  -- Declares types used in the rest of the design
  type STATE_TYPE is (WAIT_FOR_START, READ_BITS, PARITY_ERROR_DETECTED, ALLOW_READ);
  constant PARALLEL_BIT_COUNT: INTEGER := 8;
subtype PARALLEL_RANGE is INTEGER
  range 0 to (PARALLEL_BIT_COUNT-1);
subtype PARALLEL_TYPE is BIT_VECTOR(PARALLEL_RANGE);
end TYPES;

use WORK.TYPES.ALL; -- Use the TYPES package

entity SER_PAR is -- Declare the interface
  port(SERIAL_IN, CLOCK, RESET: in BIT;
       PARALLEL_OUT: out PARALLEL_TYPE;
       PARITY_ERROR, READ_ENABLE: out BIT);
end;

architecture BEHAVIOR of SER_PAR is
  -- Signals for stored values
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
  signal CURRENT_PARITY, NEXT_PARITY: BIT;
  signal CURRENT_BIT_POSITION, NEXT_BIT_POSITION:
    INTEGER range PARALLEL_BIT_COUNT downto 0;
  signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
    PARALLEL_TYPE;
begin
  NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
                    CURRENT_BIT_POSITION, CURRENT_PARITY,
                    CURRENT_PARALLEL_OUT)
  -- This process computes all outputs, the next
  -- state, and the next value of all stored values
  begin
    PARITY_ERROR <= '0'; -- Default values for all
    READ_ENABLE <= '0'; -- outputs and stored values
    NEXT_STATE <= CURRENT_STATE;
    NEXT_BIT_POSITION <= 0;
    NEXT_PARITY <= '0';
    NEXT_PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

    if (RESET = '1') then -- Synchronous reset
      NEXT_STATE <= WAIT_FOR_START;
    else
      case CURRENT_STATE is -- State processing
        when WAIT_FOR_START =>
          if (SERIAL_IN = '1') then
            NEXT_STATE <= READ_BITS;
            NEXT_PARALLEL_OUT <= PARALLEL_TYPE'(others=>'0');
          end if;
        when READ_BITS =>
          -- Further processing
    end case;
  end process;

  -- Other processes...

END;
Example A–13 (continued)  Serial-to-Parallel Converter—Counting Bits

if (CURRENT_BIT_POSITION = PARALLEL_BIT_COUNT) then
    if (CURRENT_PARITY = SERIAL_IN) then
        NEXT_STATE <= ALLOW_READ;
        READ_ENABLE <= '1';
    else
        NEXT_STATE <= PARITY_ERROR_DETECTED;
    end if;
else
    NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <= SERIAL_IN;
    NEXT_BIT_POSITION <= CURRENT_BIT_POSITION + 1;
    NEXT_PARITY <= CURRENT_PARITY xor SERIAL_IN;
end if;
when PARITY_ERROR_DETECTED =>
    PARITY_ERROR <= '1';
when ALLOW_READ =>
    NEXT_STATE <= WAIT_FOR_START;
end case;
end if;
end process;

SYNCH: process
    -- This process remembers the stored values
    -- across clock cycles
begin
    wait until CLOCK’event and CLOCK = ’1’;
    CURRENT_STATE <= NEXT_STATE;
    CURRENT_BIT_POSITION <= NEXT_BIT_POSITION;
    CURRENT_PARITY <= NEXT_PARITY;
    CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
end process;

PARALLEL_OUT <= CURRENT_PARALLEL_OUT;
end BEHAVIOR;
Example A–13 (continued) Serial-to-Parallel Converter—Counting Bits
Serial-to-Parallel Converter—Shifting Bits

This example describes another implementation of the serial-to-parallel converter in the last example. This design performs the same function as the previous one, but uses a different algorithm to do the conversion.

In the previous implementation, a counter was used to indicate the bit of the output that was set when a new serial bit was read. In this implementation, the serial bits are shifted into place. Before the conversion takes place, a ‘1’ is placed in the least-significant bit position. When that ‘1’ is shifted out of the most significant position (position 0), the signal NEXT_HIGH_BIT is set to ‘1’ and the conversion is complete.

The listing of the second implementation follows. The differences are highlighted in bold. The differences relate to the removal of the ___BIT_POSITION signals, the addition of ___HIGH_BIT signals, and the change in the way NEXT_PARALLEL_OUT is computed.

Example A–14  Serial-to-Parallel Converter—Shifting Bits

package TYPES is
  -- Declares types used in the rest of the design
  type STATE_TYPE is (WAIT_FOR_START,
    READ_BITS,
    PARITY_ERROR_DETECTED,
    ALLOW_READ);
  constant PARALLEL_BIT_COUNT: INTEGER := 8;
  subtype PARALLEL_RANGE is INTEGER range 0 to (PARALLEL_BIT_COUNT–1);
  subtype PARALLEL_TYPE is BIT_VECTOR(PARALLEL_RANGE);
end TYPES;

use WORK.TYPES.ALL;      -- Use the TYPES package

entity SER_PAR is        -- Declare the interface
  port(SERIAL_IN, CLOCK, RESET: in BIT;
    PARALLEL_OUT: out PARALLEL_TYPE;
    PARITY_ERROR, READ_ENABLE: out BIT);
architecture BEHAVIOR of SER_PAR is
   -- Signals for stored values
   signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;

   signal CURRENT_PARITY, NEXT_PARITY: BIT;
   signal CURRENT_HIGH_BIT, NEXT_HIGH_BIT: BIT;
   signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
      PARALLEL_TYPE;
begin

   NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
   CURRENT_HIGH_BIT, CURRENT_PARITY,
   CURRENT_PARALLEL_OUT)
   -- This process computes all outputs, the next
   -- state, and the next value of all stored values
   begin
      PARITY_ERROR <= '0'; -- Default values for all
      READ_ENABLE <= '0';  -- outputs and stored values
      NEXT_STATE <= CURRENT_STATE;
      NEXT_HIGH_BIT <= '0';
      NEXT_PARITY <= '0';
      NEXT_PARALLEL_OUT <= PARALLEL_TYPE'(others=>'0');
      if(RESET = '1') then      -- Synchronous reset
         NEXT_STATE <= WAIT_FOR_START;
      else
         case CURRENT_STATE is   -- State processing
            when WAIT_FOR_START =>
               if (SERIAL_IN = '1') then
                  NEXT_STATE <= READ_BITS;
                  NEXT_PARALLEL_OUT <=
                     PARALLEL_TYPE'(others=>'0');
               end if;
            when READ_BITS =>
               if (CURRENT_HIGH_BIT = '1') then
                  if (CURRENT_PARITY = SERIAL_IN) then
                     NEXT_STATE <= ALLOW_READ;
                     READ_ENABLE <= '1';
                  else
                     NEXT_STATE <= PARITY_ERROR_DETECTED;
                  end if;
               else
               NEXT_HIGH_BIT <= CURRENT_PARALLEL_OUT(0);
               NEXT_PARALLEL_OUT <=
                  CURRENT_PARALLEL_OUT(""
1 to PARALLEL_BIT_COUNT-1) &
   SERIAL_IN;
   NEXT_PARITY <= CURRENT_PARITY xor
   SERIAL_IN;
   end if;
when PARITY_ERROR_DETECTED =>
   PARITY_ERROR <= '1';
when ALLOW_READ =>
   NEXT_STATE <= WAIT_FOR_START;
   end case;
   end if;
end process;

SYNCH: process
   -- This process remembers the stored values
   -- across clock cycles
begin
   wait until CLOCK'event and CLOCK = '1';
   CURRENT_STATE <= NEXT_STATE;
   CURRENT_HIGH_BIT <= NEXT_HIGH_BIT;
   CURRENT_PARITY <= NEXT_PARITY;
   CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
end process;

PARALLEL_OUT <= CURRENT_PARALLEL_OUT;
end BEHAVIOR;

Note that the synthesized schematic for the shifter implementation is much simpler than the first (Example A–13). It is simpler because the shifter algorithm is inherently easier to implement.
Example A–14 (continued)  Serial-to-Parallel Converter—Shifting Bits

With the count algorithm, each of the flip-flops holding the **PARALLEL_OUT** bits needed logic that decoded the value stored in the **BIT_POSITION** flip-flops to see when to route in the value of **SERIAL_IN**. Also, the **BIT_POSITION** flip-flops needed an incrementer to compute their next value.

In contrast, the shifter algorithm requires no incrementer, and no flip-flops to hold **BIT_POSITION**. Additionally, the logic in front of most **PARALLEL_OUT** bits needs to read only the value of the previous flip-flop, or '0'. The value depends on whether bits are currently being read. In the shifter algorithm, the **SERIAL_IN** port needs to be connected only to the least significant bit (number 7) of the **PARALLEL_OUT** flip-flops.

These two implementations illustrate the importance of designing efficient algorithms. Both work properly, but the shifter algorithm produces a faster, more area-efficient design.
Programmable Logic Array (PLA)

This example shows a way to build PLAs in VHDL. The PLA function uses an input lookup vector as an index into a constant PLA table, then returns the output vector specified by the PLA.

The PLA table is an array of PLA_ROWS, where each row is an array of PLA_ELEMENTS. Each element is either a 1, a 0, a minus, or a space (‘1’, ‘0’, ‘–’, or ‘ ’). The table is split between an input plane and an output plane. The input plane is specified by 0s, 1s, and minuses. The output plane is specified by 0s and 1s. The two planes’ values are separated by a space.

In the PLA function, the output vector is first initialized to be all ‘0’s. When the input vector matches an input plane in a row of the PLA table, the ‘1’s in the output plane are assigned to the corresponding bits in the output vector. A match is determined as follows:

- If a ‘0’ or ‘1’ is in the input plane, the input vector must have the same value in the same position.
- If a ‘–’ is in the input plane, it matches any input vector value at that position.

The generic PLA table types and the PLA function are defined in a package named LOCAL. An entity PLA_VHDL that uses LOCAL needs only to specify its PLA table as a constant, then call the PLA function.
Note that the \texttt{PLA} function does not explicitly depend on the size of the PLA. To change the size of the PLA, change the initialization of the \texttt{TABLE} constant and the initialization of the constants \texttt{INPUT\_COUNT}, \texttt{OUTPUT\_COUNT} and \texttt{ROW\_COUNT}. In Example A–15, these constants are initialized to a PLA equivalent to the ROM shown previously (Example A–3). Accordingly, the synthesized schematic is the same as that of the ROM, with one difference: in Example A–3, the \texttt{DATA} output port range is 1 to 5; in Example A–15, the \texttt{OUT\_VECTOR} output port range is 4 downto 0.

This example is included mainly to show the capabilities of VHDL. It is more efficient to define the PLA directly, by using the PLA input format. See the \textit{Design Compiler Family Reference Manual} for more information about the PLA input format.

\textbf{Example A–15}  \textit{Programmable Logic Array}

package LOCAL is
  constant INPUT\_COUNT: INTEGER := 3;
  constant OUTPUT\_COUNT: INTEGER := 5;
  constant ROW\_COUNT: INTEGER := 6;
  constant ROW\_SIZE: INTEGER := INPUT\_COUNT +
                   OUTPUT\_COUNT + 1;
  type PLA\_ELEMENT is ('1', '0', '–', ' ');
  type PLA\_VECTOR is
    array (INTEGER range <>) of PLA\_ELEMENT;
  subtype PLA\_ROW is
    PLA\_VECTOR(ROW\_SIZE – 1 downto 0);
  subtype PLA\_OUTPUT is
    PLA\_VECTOR(OUTPUT\_COUNT – 1 downto 0);
  type PLA\_TABLE is
    array(ROW\_COUNT – 1 downto 0) of PLA\_ROW;

  function PLA(IN\_VECTOR: BIT\_VECTOR;
              TABLE: PLA\_TABLE) return BIT\_VECTOR;
end LOCAL;
package body LOCAL is

function PLA(IN_VECTOR: BIT_VECTOR;
            TABLE: PLA_TABLE) return BIT_VECTOR is
    subtype RESULT_TYPE is
        BIT_VECTOR(OUTPUT_COUNT – 1 downto 0);
    variable RESULT: RESULT_TYPE;
    variable ROW: PLA_ROW;
    variable MATCH: BOOLEAN;
    variable IN_POS: INTEGER;

    begin
        RESULT := RESULT_TYPE'(others => BIT'( '0' ));

        for I in TABLE’range loop
            ROW := TABLE(I);
            MATCH := TRUE;
            IN_POS := IN_VECTOR’left;

            -- Check for match in input plane
            for J in ROW_SIZE – 1 downto OUTPUT_COUNT loop
                if(ROW(J) = PLA_ELEMENT'( '1' )) then
                    MATCH := MATCH and
                    (IN_VECTOR(IN_POS) = BIT'( '1' ));
                elsif(ROW(J) = PLA_ELEMENT'( '0' )) then
                    MATCH := MATCH and
                    (IN_VECTOR(IN_POS) = BIT'( '0' ));
                else
                    null; -- Must be minus ("don’t care")
                end if;
                IN_POS := IN_POS – 1;
            end loop;

            -- Set output plane
            if(MATCH) then
                for J in RESULT’range loop
                    if(ROW(J) = PLA_ELEMENT'( '1' )) then
                        RESULT(J) := BIT'( '1' );
                    end if;
                end loop;
            end if;
        end loop;

        return(RESULT);
    end loop;
    end if;
end if;
end loop;

return(RESULT);
end;
end LOCAL;

use WORK.LOCAL.all;

entity PLA_VHDL is
  port(IN_VECTOR: BIT_VECTOR(2 downto 0);
       OUT_VECTOR: out BIT_VECTOR(4 downto 0));
end;

architecture BEHAVIOR of PLA_VHDL is
  constant TABLE: PLA_TABLE := PLA_TABLE'(
    PLA_ROW'("––– 10000"),
    PLA_ROW'("–1– 01000"),
    PLA_ROW'("0–0 00101"),
    PLA_ROW'("–1– 00101"),
    PLA_ROW'("1–1 00101"),
    PLA_ROW'("–1– 00010"));

begin
  OUT_VECTOR <= PLA(IN_VECTOR, TABLE);
end BEHAVIOR;