### Solutions Manual (v4)

#### VHDL

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Exercises covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Introduction</td>
<td>---</td>
</tr>
<tr>
<td>2: Code Structure</td>
<td>1</td>
</tr>
<tr>
<td>3: Data Types</td>
<td>1, 9, 15, 20, 23, 30</td>
</tr>
<tr>
<td>4: Operators and Attributes</td>
<td>4, 6, 8, 9, 13</td>
</tr>
<tr>
<td>5: Concurrent Code</td>
<td>4, 7, 11, 20</td>
</tr>
<tr>
<td>6: Sequential Code</td>
<td>1, 4, 8, 9, 12, 13</td>
</tr>
<tr>
<td>7: SIGNAL and VARIABLE</td>
<td>3, 4, 7, 9</td>
</tr>
<tr>
<td>8: PACKAGE and COMPONENT</td>
<td>6, 8</td>
</tr>
<tr>
<td>9: FUNCTION and PROCEDURE</td>
<td>1, 5, 6</td>
</tr>
<tr>
<td>10: Simulation with VHDL Testbenches</td>
<td>1, 11, 12</td>
</tr>
</tbody>
</table>

#### Extended and Advanced Designs

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Exercises covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>11: VHDL Design of State Machines</td>
<td>4, 6, 11, 15</td>
</tr>
<tr>
<td>12: VHDL Design with Basic Displays</td>
<td>2, 13</td>
</tr>
<tr>
<td>13: VHDL Design of Memory Circuits</td>
<td>5, 9</td>
</tr>
<tr>
<td>14: VHDL Design of Serial Communications Circuits</td>
<td>10</td>
</tr>
<tr>
<td>15: VHDL Design of VGA Video Interfaces</td>
<td>1, 2, 3, 4, 6</td>
</tr>
<tr>
<td>16: VHDL Design of DVI Video Interfaces</td>
<td>2, 3, 4, 6</td>
</tr>
<tr>
<td>17: VHDL Design of FPD-Link Video Interfaces</td>
<td>2, 3, 4</td>
</tr>
</tbody>
</table>
Exercise 2.1: Multiplexer

a) Code

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux IS
  PORT (a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0));
  x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END mux;

ARCHITECTURE example OF mux IS
BEGIN
  PROCESS (a, b, sel)
  BEGIN
    IF (sel="00") THEN
      x <= "00000000";
    ELSIF (sel="01") THEN
      x <= a;
    ELSIF (sel="10") THEN
      x <= b;
    ELSE
      x <= "ZZZZZZZZ";
    END IF;
  END PROCESS;
END example;
```

b) Comments:
Lines 2–3: Library/package declarations. Because the type STD_LOGIC is employed, the package std_logic_1164 must be included. The other two indispensable libraries (std and work) are made visible by default.
Lines 5–10: ENTITY, here named `mux`.
Lines 7–9: Specifications of all input and output ports (all of type STD_LOGIC_VECTOR in this example).
Lines 12–27: ARCHITECTURE, here named `example`.
Lines 14–26: A PROCESS was employed to construct the circuit. Its sensitivity list (line 13) contains the signals `a`, `b`, and `sel`, so whenever one of these signals changes its value the process is run.
Lines 16–25: In the process, the IF statement was used to implement the multiplexer, according to the truth table of figure 2.9.
Lines 1, 4, 11, 28: Used just to improve code readability (these lines separate the three fundamental code sections).

Exercise 3.1: Possible data types #1

```vhdl
s1 <= '0';  --BIT, STD_LOGIC, CHARACTER
s2 <= 'Z';  --STD_LOGIC, CHARACTER
s3 <= TRUE; --BOOLEAN
s4 <= "01000";  --BIT_VECTOR, STD_LOGIC_VECTOR, SIGNED, UNSIGNED, STRING
```

Note: Above, STD_LOGIC means STD_LOGIC and STD_ULOGIC

Exercise 3.9: Possible packages

a) None, because arithmetic and comparison operators for INTEGER are already included in the definition package (standard), which is visible by default.
b) Either std_logic_unsigned or std_logic_signed. If VHDL 2008 has already been implemented, numeric_std_unsigned is another option.

Exercise 3.15: 1Dx1D array examples

```vhdl
TYPE type1 IS ARRAY (NATURAL RANGE <>) OF BIT_VECTOR(7 DOWNTO 0);
TYPE type2 IS ARRAY (1 TO M) OF BIT_VECTOR(N-1 DOWNTO 0);
TYPE type3 IS ARRAY (1 TO M) OF NATURAL RANGE 0 TO 2**N-1;
```
Exercise 3.20: Type conversion by specific functions

Figure 3.10 is very helpful to solve this exercise. Type casting was also included in some cases (see figure 3.10 of the book).

a) INTEGER to SLV: Function `conv_std_logic_vector(a, cs)` from the package `std_logic_arith`.
b) BV to SLV: Function `to_stdlogicvector(a, cs)` from the package `std_logic_1164`.
c) SLV to UNSIGNED: Function `unsigned(a)` from the package `numeric_std` or `std_logic_arith`.
d) SLV to INTEGER: Function `conv_integer(a, cs)` from the package `std_logic_signed` or `std_logic_unsigned`, or function `to_integer(a, cs)` from `numeric_std_unsigned`.
e) SIGNED to SLV: Function `std_logic_vector(a)` from the package `numeric_std` or `std_logic_arith`, or function `conv_std_logic_vector(a, cs)` from `std_logic_arith`.

Exercise 3.23: Array dimensionality

SIGNAL s1: BIT;         --scalar (a single bit)
SIGNAL s2: BIT_VECTOR(7 DOWNTO 0);   --1D (a vector of bits)
SIGNAL s3: STD_LOGIC;       --scalar (single bit)
SIGNAL s4: STD_LOGIC_VECTOR(7 DOWNTO 0); --1D (vector of bits)
VARIABLE v1: BIT_VECTOR(7 DOWNTO 0);  --1D (vector of bits)
VARIABLE v2: INTEGER RANGE -35 TO 35;  --1D (vector of bits)

Exercise 3.30: Illegal assignments #3

a) s4(0)(0) <= s7(1,1,1);    --cause 1 (BIT x STD_LOGIC)
b) s6(1) <= s4(1);       --cause 2 (pile of BIT_VECTOR x single BIT_VECTOR)
c) s1 <= "00000000";             --cause 3 (individual values must be TRUE or FALSE)
d) s7(0)(0)(0) <= 'Z';     --cause 4 (zero is invalid index) + cause 3 (wrong parenthesis)
e) s2(7 DOWNTO 5) <= s1(2 DOWNTO 0);--cause 1 (slice with BIT x slice with BOOLEAN)
f) s4(1) <= (OTHERS => 'Z'); --cause 3 ('Z' is an invalid value for BIT)
g) s6(1,1) <= s2;          --cause 4 (the correct index is s6(1)(1))
h) s2 <= s3(1) AND s4(1);  --invalid AND operator (for INT.) + cause 1 (type2 x INTEGER x BV)
i) s1(0 TO 1) <= (FALSE, FALSE);  --cause 4 (index of s1 must be downward)
j) s3(1) <= (3, 35, -8, 97);  --cause 2 (single INTEGER x pile of INTEGER)

Exercise 4.4: Logical operators

a) a(7 DOWNTO 4) NAND "0111" → "1100"
b) a(7 DOWNTO 4) XOR NOT b → "1100"
c) "1111" NOR b → "0000" → "0000"
d) b(2 DOWNTO 0) XNOR "101" → "101"

Exercise 4.6: Arithmetic operators #2

a) x REM y = 65 REM 7 = 65 - (65/7)*7 = 2 
b) x REM -y = 65 REM -7 = 65 - (65/-7)*-7 = 2 
c) (x + 2*y) REM y = 79 REM 7 = 79 - (79/7)*7 = 2 
d) (x + y) REM -x = 72 REM -7 = 72 - (72/-65)*-65 = 7 
e) x MOD y = 65 MOD 7 = 65 REM 7 + 0 = 2 
f) x MOD -y = 65 MOD -7 = (65 REM -7) + (-7) = 2 - 7 = -5 
g) -x MOD -y = -65 MOD -7 = (-65 REM -7) + 0 = -65 - (-65/-7)*-7 = -2 
h) ABS(-y) = ABS(-7) = 7
Exercise 4.8: Shift and concatenation operators

a) \( x \text{ SLL } 3 = "010000" \)  

b) \( x \text{ SLA } -2 = "111100" \)  

c) \( x \text{ SRA } 2 = "111100" \)  

d) \( x \text{ ROL } 1 = "100101" \)  

e) \( x \text{ ROR } -3 = "010110" \)

\[ a) x(2 \text{ DOWNTO } 0) \ & \ "000"; \]
\[ b) x(5) \ & \ x(5) \ & \ x(5 \text{ DOWNTO } 2); \]
\[ c) x(5) \ & \ x(5) \ & \ x(5 \text{ DOWNTO } 2); \]
\[ d) x(4 \text{ DOWNTO } 0) \ & \ x(5); \]
\[ e) x(2 \text{ DOWNTO } 0) \ & \ x(5 \text{ DOWNTO } 3); \]

Exercise 4.9: Arithmetic operators for signed types

a) \( x \leq a + b; \)  

b) \( x \leq b + c; \)  

c) \( x \leq 3*b; \)  

d) \( x \leq 3*a + b; \)  

e) \( x \leq a + c + "1111"; \)  

f) \( x \leq a + c + \text{SIGNED}'("1111"); \)  

Notes:
1) Output size (number of bits) must be equal to the largest input
2) Output size must be twice the input size
3) Output size must be equal to the sum of the two input sizes
4) Note that “1111” is signed, determined by context
5) Note the use of a qualified expression (see section 3.17)

Exercise 4.13: The \textit{enum_encoding} attribute

a) Sequential: \( a="000", b="001", c="010", d="011", e="100", f="101" \)  
b) Gray: \( a="000", b="001", c="011", d="010", e="110", f="111" \)  
c) Johnson: \( a="000", b="100", c="110", d="111", e="011", f="001" \)  
\quad or \( a="000", b="001", c="011", d="111", e="110", f="100" \)  
d) One-hot: \( a="00001", b="00010", c="00100", d="01000", e="01000", f="10000" \)

Exercise 5.4: Generic parity generator

Note in the code below the use of the recommendation introduced in section 7.7.

```vhdl
1 ENTITY parity_generator IS
2 GENERIC (N: INTEGER := 8); -- number of bits
3 PORT (x: IN BIT_VECTOR(N-1 DOWNTO 0); y: OUT BIT_VECTOR(N DOWNTO 0));
4 END ENTITY;
5 ARCHITECTURE structural OF parity_generator IS
6 SIGNAL internal: BIT_VECTOR(N-1 DOWNTO 0);
7 BEGIN
8 internal(0) <= x(0);
9 gen: FOR i IN 1 TO N-1 GENERATE
10 internal(i) <= internal(i-1) XOR x(i);  
11 END GENERATE;
12 y <= internal(N-1) & x;
13 END structural;
```

Exercise 5.7: Hamming weight with \text{GENERATE}

Note in the code below again the use of the recommendation introduced in section 7.7.

```vhdl
1 ENTITY parity_generator IS
2 GENERIC (N: INTEGER := 8); -- number of bits
3 PORT (x: IN BIT_VECTOR(N-1 DOWNTO 0); y: OUT BIT_VECTOR(N DOWNTO 0));
4 END ENTITY;
5 ARCHITECTURE structural OF parity_generator IS
6 SIGNAL internal: BIT_VECTOR(N-1 DOWNTO 0);
7 BEGIN
8 internal(0) <= x(0);
9 gen: FOR i IN 1 TO N-1 GENERATE
10 internal(i) <= internal(i-1) XOR x(i);  
11 END GENERATE;
12 y <= internal(N-1) & x;
13 END structural;
```
--- ENTITY hamming_weight IS
  GENERIC (
    N: INTEGER := 8); --number of bits
  PORT ( 
    x: IN BIT_VECTOR(N-1 DOWNTO 0); 
    y: OUT NATURAL RANGE 0 TO N);
END ENTITY;
---
ARCHITECTURE hamming_weight OF hamming_weight IS
  TYPE natural_array IS ARRAY (0 TO N) OF NATURAL RANGE 0 TO N;
  SIGNAL internal: natural_array;
BEGIN
  internal(0) <= 0;
  gen: FOR i IN 1 TO N GENERATE
    internal(i) <= internal(i-1) + 1 WHEN x(i-1)='1' ELSE internal(i-1);
  END GENERATE;
  y <= internal(N);
END ARCHITECTURE;
---

Exercise 5.11: Arithmetic circuit with STD_LOGIC

Note in the solution below the use of the sign-extension recommendation seen in sections 3.18 and 5.7.
Exercise 5.20: Generic Multiplexer

a) Solution with a user-defined 2D type:
In this solution, a PACKAGE, called my_data_types, is employed to define a new data type, called matrix, which is then used in the ENTITY of the main code to specify the multiplexer’s inputs.

```vhdl
--Package:---------------------------------------------------------------
PACKAGE my_data_types IS
   TYPE matrix IS ARRAY (NATURAL RANGE <> , NATURAL RANGE <> ) OF BIT;
END PACKAGE my_data_types;
-------------------------------------------------------------------
--Main code:-------------------------------------------------------------
USE work.my_data_types.all;

ENTITY generic_mux IS
   GENERIC (
      M: POSITIVE := 8;  --number of inputs
      N: POSITIVE := 4);  --size of each input
   PORT (
      x: IN matrix (0 TO M-1, N-1 DOWNTO 0);
      sel: IN INTEGER RANGE 0 TO M-1;
      y: OUT BIT_VECTOR (N-1 DOWNTO 0));
END ENTITY;

ARCHITECTURE arch1 OF generic_mux IS
BEGIN
   gen: FOR i IN N-1 DOWNTO 0 GENERATE
      y(i) <= x(sel, i);
   END GENERATE gen;
END ARCHITECTURE;
-----------------------------------------------------
```

b) Solution with a predefined type:
In this solution, no package was employed. Notice, however, that x was not defined as a 1Dx1D or 2D structure; instead, it was specified simply as a long vector of length $M \times N$. Though this will not affect the result, such a “linearization” might be confusing sometimes, so it is usually not recommended.

```vhdl
ENTITY generic_mux IS
   GENERIC (
      M: POSITIVE := 8;  --number of inputs
      N: POSITIVE := 4);  --size of each input
   PORT (
      x: IN BIT_VECTOR (M*N-1 DOWNTO 0);
      sel: IN NATURAL RANGE 0 TO M-1;
      y: OUT BIT_VECTOR(N-1 DOWNTO 0));
END ENTITY;

ARCHITECTURE arch2 OF generic_mux IS
BEGIN
   gen: FOR i IN N-1 DOWNTO 0 GENERATE
      y(i) <= x(N*sel+i);
   END GENERATE gen;
END ARCHITECTURE;
```

Exercise 6.1: Latch and flip-flop

a) See figure below.
b) VHDL code and simulation:

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY example IS
PORT (d, clk: IN STD_LOGIC;
q1, q2: OUT STD_LOGIC);
END ENTITY;

ARCHITECTURE example OF example IS
BEGIN
PROCESS(clk, d)
BEGIN
---Latch:--------
IF clk='1' THEN
q1 <= d;
END IF;
---Flip-flop:----
IF clk'EVENT AND clk='1' THEN --or IF rising_edge(clk) THEN
q2 <= d;
END IF;
END PROCESS;
END ARCHITECTURE;
```

Simulation results:

<table>
<thead>
<tr>
<th>Name</th>
<th>0 ps</th>
<th>100 ns</th>
<th>160 ns</th>
<th>240 ns</th>
<th>320 ns</th>
<th>400 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Exercise 6.4: Generic registered multiplexer**

Just add a register ($N$ flip-flops) to the output of the multiplexer designed in exercise 5.20 (recall that now a process is needed).

**Exercise 6.8: Signal generator**

Two additional waveforms ($a$, $b$) were included in the figure. Note that there is a big difference between generating $a$, $b$ versus $x$, $y$, because each signal in the former pair changes its state always at the same clock edge, so its resolution is one clock period, while in the latter pair each signal can change at both clock transitions, so with a resolution equal to one-half of a clock period (which is the maximum resolution in digital systems). In the code below, first $a$ and $b$ are generated, then
trivial logic gates are employed to obtain \( x \) and \( y \). Note that this implementation is free from glitches because only two signals enter the gates, and such signals can never change at the same time (they operate at different clock edges).

```vhdl
ENTITY signal_generator IS
  PORT (clk: IN BIT;
       x, y: OUT BIT);
END ENTITY;

ARCHITECTURE arch OF signal_generator IS
BEGIN
  PROCESS(clk)
  VARIABLE a, b: BIT;
  BEGIN
    IF clk'EVENT AND clk='1' THEN –-or IF rising_edge(clk) THEN
      a := NOT a;
    ELSIF clk'EVENT AND clk='0' THEN –-or ELSIF falling_edge(clk) THEN
      b := NOT a;
    END IF;
    x <= a AND b;
    y <= a NOR b;
  END PROCESS;
END ARCHITECTURE;
```

Exercise 6.9: Switch debouncer

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY debouncer IS
  GENERIC(
    fclk: NATURAL := 50;    --clock freq in MHz
    twindow: NATURAL := 10); --time window in ms
  PORT (sw: IN STD_LOGIC;
        clk: IN STD_LOGIC;
        deb_sw: BUFFER STD_LOGIC);
END ENTITY;

ARCHITECTURE digital_debouncer OF debouncer IS
  CONSTANT max: NATURAL := 1000 * fclk * twindow;
BEGIN
  PROCESS (clk)
  VARIABLE count: NATURAL RANGE 0 TO max;
  BEGIN
    IF (clk'EVENT AND clk='1') THEN –-or IF rising_edge(clk) THEN
      IF (deb_sw /= sw) THEN
        count := count + 1;
        IF (count=max) THEN
          deb_sw <= sw;
          count := 0;
        END IF;
      ELSE
        count := 0;
      END IF;
    END IF;
  END PROCESS;
END ARCHITECTURE;
```

Exercise 6.12: Programmable signal generator

(Note: See a continuation for this design in exercise 6.13.)

a) The dividers must be even if we want to operate in only one (say, rising) clock edge (if the clock’s duty cycle is unknown that is the only option).

\( \frac{50 \text{MHz}}{10 \text{kHz}} = 5000 \text{ (error = 0 Hz)} \)
50MHz/9kHz = 5555.55 (use 5556 → 50M/5556 = 8999.28, so error = -0.72 Hz)
50MHz/8kHz = 6250 (error = 0 Hz)
50MHz/7kHz = 7142.86 (use 7142 → 50M/7142 = 7000.84, so error = 0.84 Hz)
50MHz/6kHz = 8333.33 (use 8334 → 50M/8334 = 5999.52, so error = -0.48 Hz)
50MHz/5kHz = 10000 (error = 0 Hz)
50MHz/4kHz = 12500 (error = 0 Hz)
50MHz/3kHz = 16666.66 (use 16666 → 50M/16666 = 3000.12, so error = 0.12 Hz)
50MHz/2kHz = 25000 (error = 0 Hz)
50MHz/1kHz = 50000 (error = 0 Hz)

b) VHDL code:
Note the use of “>=” rather than “=” in the “count>=divider/2” test. This is needed for the counter to immediately shift to the new mode in case the counter already is above “count=divider/2” when the test is done (otherwise, we would need to wait until the counter reached its full scale, after which it would automatically restart from zero).

```vhdl
ENTITY sig_generator IS
  GENERIC (
fclk: POSITIVE := 50_000_000);
  PORT (
    clk, freq: IN BIT;
    sig_out: BUFFER BIT);
END ENTITY;

ARCHITECTURE sig_generator OF sig_generator IS
  TYPE ROM IS ARRAY (0 TO 3) OF POSITIVE RANGE 1 TO 8;
  CONSTANT coefficients: ROM := (8, 6, 4, 2);
  SIGNAL divider: POSITIVE RANGE 1 TO 8;
BEGIN
  PROCESS (freq)
  VARIABLE i: NATURAL RANGE 0 TO 4;
  BEGIN
    IF (freq'EVENT AND freq='1') THEN
      i := i + 1;
      IF (i=4) THEN
        i := 0;
      END IF;
    END IF;
    divider <= coefficients(i);
  END PROCESS;

  PROCESS (clk)
  VARIABLE count: NATURAL RANGE 0 TO 4;
  BEGIN
    IF (clk'EVENT AND clk='1') THEN
      count := count + 1;
      IF (count>=divider/2) THEN
        count := 0;
        sig_out <= NOT sig_out;
      END IF;
    END IF;
  END PROCESS;
END ARCHITECTURE;
```

c) Simulation results:

<table>
<thead>
<tr>
<th>Name</th>
<th>Dps</th>
<th>160.0 ns</th>
<th>320.0 ns</th>
<th>480.0 ns</th>
<th>640.0 ns</th>
<th>800.0 ns</th>
<th>960.0 ns</th>
<th>1.12 us</th>
<th>1.38 us</th>
<th>1.44 us</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>freq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sig_out</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exercise 6.13: Programmable Signal Generator with Frequency Meter
a) See circuits and code in section 22.5 of [Pedroni 2008]. Adopt the implementation in Fig. 22.9(a).

b) Put the code of part (a) above together with the code of exercise 6.12 to attain the complete system. Another option (not discussed yet) it to use the COMPONENT construct (described in chapter 8), in which case the main code can be constructed with just two lines (two component instantiations).

c) This implementation in the FPGA board is a very interesting and stimulating design demonstration.

Exercise 7.3: Latches and flip-flops

(a) Latch with reset  (b) Flip-flop with clear  (c) Latch with clear

Code 1: This is a D-type latch with asynchronous reset (figure (a) above). Note that, contrary to flip-flop implementations, d needs to be in the sensitivity list because the circuit is transparent during a whole semi-period of the clock, not just during the clock transition. (Even though the compiler might understand from the context that a latch is wanted even if d is not in the sensitivity list, that is not a good practice.)

Code 2: Now we have a D-type flip-flop instead of a D-type latch (note the 'EVENT attribute). Because reset is only activated when a (positive) clock edge occurs, it is synchronous; since in our context an asynchronous reset is an actual reset, while a synchronous reset is indeed a clear signal, the proper circuit is that of figure (b).

Code 3: This is an example of bad design. In principle, it seems that a latch with clear is wanted, like that in figure (c). However, because only clk is in the sensitivity list, the process will only be run when clk changes, which emulates the behavior of a flip-flop. In summary, the flip-flop of figure (b) might be inferred.

Exercise 7.4: Combinational versus sequential circuits #1

a) It is combinational because the output depends solely on the current input.

b) According to Rule 6 in figure 7.1, registers are inferred when a value is assigned to a signal at the transition of another signal (normally using the 'EVENT attribute; or, equivalently, the rising_edge or falling_edge function), which does not happen in this code.

c) Same as (a) above.

d) Same as (b) above.

Exercise 7.7: Registered circuits

a) Code analysis:
Code 1 is a 8x1 multiplexer (8 inputs, 1 bit each), implemented by line 14, followed by a one-bit register. Hence only one DFF is needed. The compiled circuit is shown in figure (a) below.

Code 2 is the opposite, that is, it contains a register (8 bits, so 8 DFFs are needed, inferred by lines 14-16) followed by an 8x1 mux (line 17). The compiled circuit is in figure (b).

Code 3 is similar to code 1. The only difference is that no internal signal was used to implement the multiplexer (it was implemented directly in line 14). Hence the circuit is that of figure (a), with one DFF.

b) Compilation: The inferred circuits are in the figure below.
c) Codes 1 and 3 implement the same circuit. The hardwares inferred from these two codes are equal, so from the implementation point-of-view there are no differences between them. However, code 3 is more compact, while code 1 makes it clearer that a mux followed by a register is the wanted circuit.

Exercise 7.9: Frequency divider with VARIABLE

a) Number of flip-flops:
An $M$-state counter (from 0 to $M$–1, for example) can be used to solve this problem, so the number of DFFs is $\left\lceil \log_2 M \right\rceil$ for the counter, plus one to store $clk\_out$. For example, for $M=4$ or $M=5$, 3 or 4 DFFs are needed, respectively.

b) VHDL code (see simulation results in the figure that follows):

```vhdl
  ENTITY clock_divider IS
  GENERIC (M: NATURAL := 5);
  PORT (clk_in: IN BIT; clk_out: OUT BIT);
  END ENTITY;

  ARCHITECTURE circuit OF clock_divider IS
  BEGIN
    PROCESS (clk_in)
    VARIABLE counter: NATURAL RANGE 0 TO M;
    BEGIN
      IF clk_in'EVENT AND clk_in='1' THEN –-or IF rising_edge(clk_in) THEN
        counter := counter + 1;
        IF counter=M/2 THEN
          clk_out <= '1';
        ELSIF counter=M THEN
          clk_out <= '0';
        END IF;
      END IF;
    END PROCESS;
  END ARCHITECTURE;
```

Simulation results for $M=5$:

<table>
<thead>
<tr>
<th>Time (ns)</th>
<th>clk_in</th>
<th>clk_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>150</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>200</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>250</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>300</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>350</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>400</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>450</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>550</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>600</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>650</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

c) Because a DFF is used to store $clk\_out$ (so this signal comes directly from a flip-flop), it is automatically free from glitches.

Exercise 8.6: Synchronous counter with COMPONENT
a) The standard cell is illustrated on the left-hand side of the figure above. A corresponding VHDL code follows, using method 1.

```vhdl
--The component:--------------------------
ENTITY counter_cell IS
  PORT ( 
    clk, a, b: IN BIT;
    c, q: BUFFER BIT);
END ENTITY;
--------------------------------------------
ARCHITECTURE counter_cell OF counter_cell IS
  SIGNAL d: BIT;
BEGIN
  PROCESS (clk, a, b)
  BEGIN
    c <= a AND b;
    d <= c XOR q;
    IF clk'EVENT AND clk='1' THEN
      q <= d;
    END IF;
  END PROCESS;
END ARCHITECTURE;
--------------------------------------------
--Main code:---------------------------------------------------------
ENTITY N_bit_counter IS
  GENERIC (
    N: NATURAL := 4); --number of bits
  PORT ( 
    clk: IN BIT;
    q: OUT BIT_VECTOR(0 TO N-1));
END ENTITY;
-----------------------------------------------------------------------
ARCHITECTURE structural OF N_bit_counter IS
  SIGNAL a, b: BIT_VECTOR(0 TO N);
  --component declaration:--
  COMPONENT counter_cell IS
    PORT ( 
      clk, a, b: IN BIT;
      c, q: BUFFER BIT);
  END COMPONENT;
BEGIN
  a(0)<='1';
  b(0)<='1';
  gen: FOR i IN 0 TO N-1 GENERATE
    counter: counter_cell PORT MAP (clk, a(i), b(i), a(i+1), b(i+1));
  END GENERATE;
  q <= b(1 TO N);
END ARCHITECTURE;
-----------------------------------------------------------------------
```

b) Having seen the code for method 1, doing it for method 3 is straightforward (see example 8.4).

Exercise 8.8: Tapped delay line with COMPONENT and GENERIC MAP

The “standard” cells can be seen in the figure below. Each cell contains an $M$-stage (so the cell is not truly standard) shift register followed by a multiplexer. The latter selects either the cell input (if sel='0'), hence without delay, or the cell output (when sel='1'), hence with a delay of $M$ clock cycles. The figure also shows the signal names, as used in the VHDL code that follows (int=interface, sel=select, etc.). The circuit produced by the compiler is depicted in the figure after the code.

```vhdl
--The component:--------------------------
ENTITY delay_cell IS
  sel2
  d
  clk
  rst
  sel
  int2
  int1
  int0
  q
END ENTITY;
```

```vhdl
--Main code:---------------------------------------------------------
ENTITY delay_cell IS
  sel0
  int0
  sel1
  int1
  sel2
  int2
  4-stage
  2-stage
  1-stage
  SR
  SR
  SR
BEGIN
  --component declaration:--
  COMPONENT delay_cell IS
    PORT ( 
      sel: IN BIT;
      int, q: OUT BIT);
  END COMPONENT;
BEGIN
  int2 <= '1';
  int1 <= '1';
  int0 <= '1';
  gen: FOR i IN 0 TO N-1 GENERATE
    counter: delay_cell PORT MAP (sel, int(i), q(i+1));
  END GENERATE;
END ARCHITECTURE;
-----------------------------------------------------------------------
```

```vhdl
--Example 8.4: Tapped delay line with COMPONENT and GENERIC MAP

The “standard” cells can be seen in the figure below. Each cell contains an M-stage shift register followed by a multiplexer. The latter selects either the cell input (if sel='0'), hence without delay, or the cell output (when sel='1'), hence with a delay of M clock cycles. The figure also shows the signal names, as used in the VHDL code that follows (int=interface, sel=select, etc.). The circuit produced by the compiler is depicted in the figure after the code.

```
Exercise 9.1: ASSERT statement #1

Just include the code below between lines 15-16 of the package.

```vhdl
ASSERT (a'LENGTH=b'LENGTH)
REPORT "Signals a and b do not have the same length!"
SEVERITY FAILURE;
```
Note: There is a typo in example 9.5; remove the semi-colon at the end of line 18.

Exercise 9.5: Function my_not

A code for this problem is shown below (note that it is similar to that in example 9.6). The function my_not was created in a package, then used in the main code to invert a vector x, passing the result to y. An alias was employed for a in order to make the code independent from the range indexes employed to specify the function input. For instance, note the unusual ranges used for x and y in the main code, with only are fine because the alias aa was adopted.

```vhdl
1  ----Package:-------------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----------------------------------------------------------------
5  PACKAGE my_package IS
6  FUNCTION my_not (a: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
7  END PACKAGE;
8  -----------------------------------------------------------------
9  PACKAGE BODY my_package IS
10  TYPE stdlogic_1d IS ARRAY (STD_ULOGIC) OF STD_ULOGIC;
11  CONSTANT not_table: stdlogic_1d :=
12  -----------------------------------------------
13  -- U    X    0    1    Z    W    L    H    -
14  -----------------------------------------------
15  ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X');
16  FUNCTION my_not (a: STD_LOGIC_VECTOR) RETURN STD_LOGIC_
17  ALIAS aa: STD_LOGIC_VECTOR(1 TO a'LENGTH) IS a;
18  VARIABLE result: STD_LOGIC_VECTOR(1 TO a'LENGTH);
19  BEGIN
20    FOR i IN result'RANGE LOOP
21      result(i) := not_table (aa(i));
22    END LOOP;
23    RETURN result;
24  END FUNCTION;
25  END PACKAGE BODY;
26  -----------------------------------------------------------------
27  ----Main code:----------------------------
28  LIBRARY ieee;
29  USE ieee.std_logic_1164.all;
30  USE work.my_package.all;
31  --------------------------------------------
32  ENTITY test IS
33    PORT (x: IN STD_LOGIC_VECTOR(8 DOWNTO 1);
34      y: OUT STD_LOGIC_VECTOR(2 TO 9));
35  END ENTITY;
36  --------------------------------------------
37  ARCHITECTURE example OF test IS
38    BEGIN
39      y <= my_not(x);
40    END ARCHITECTURE;
41  --------------------------------------------
```

Exercise 9.6: Function bcd_to_ssd

See the function integer_to_ssd in section 2.5. The ASSERT statement can be based on the solution for exercise 9.1.

Exercise 10.1: Generation of periodic stimuli

All three signals are depicted in the figure below, followed by a VHDL code that produces all three. Note that the WAIT FOR statement was used in this solution. The reader is invited to redo it using AFTER. Which is simpler in this case?
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY stimuli_generator IS
END ENTITY;

ARCHITECTURE testbench OF stimuli_generator IS
SIGNAL sig1, sig2, y: STD_LOGIC;
BEGIN
PROCESS
BEGIN
--signal sig1:--
sig1 <= '1';
WAIT FOR 25ns;
sig1 <= '0';
WAIT FOR 50ns;
--signal sig2:--
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
sig2 <= '1';
WAIT FOR 25ns;
sig2 <= '0';
WAIT FOR 50ns;
--signal y:--
y <= '1';
WAIT FOR 20ns;
y <= '0';
WAIT FOR 10ns;
y <= '1';
WAIT FOR 10ns;
y <= '0';
WAIT FOR 40ns;
END PROCESS;
END ARCHITECTURE;

Exercise 10.11: Type I testbench for an address decoder

The simulation stimuli (ena and address) to be used in this exercise are those of example 2.4 (figure 2.7), repeated in the figure below. The expected functional response (wordline) derived from them is also included in the figure.

A corresponding design file (address_decoder.vhd) is presented next. Notice that only STD_LOGIC data (lines 8-10) was employed, which is the industry standard. Observe also that this code is totally generic; to implement an address decoder of any size, the only change needed is in line 7.

The circuit can be designed in several ways. Here, concurrent code (with GENERATE and WHEN) was employed. Another option can be seen in the solutions for the 1st edition of the book.

-----Design file (address_decoder.vhd):-----------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
------------------------------------------------------------------------
ENTITY address_decoder IS
  GENERIC (N: NATURAL := 3); --number of input bits
  PORT (ena: STD_LOGIC;
         address: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
         wordline: OUT STD_LOGIC_VECTOR(2**N-1 DOWNTO 0));
END ENTITY;
------------------------------------------------------------------------
ARCHITECTURE decoder OF address_decoder IS
  SIGNAL addr: NATURAL RANGE 0 TO 2**N-1;
  BEGIN
    addr <= CONV_INTEGER(address);
    gen: FOR i IN 0 TO 2**N-1 GENERATE
      wordline(i) <= '1' WHEN ena='0' ELSE
                     '0' WHEN i=addr ELSE
                     '1';
    END GENERATE;
END ARCHITECTURE;
------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
------------------------------------------------------------------------
ENTITY address_decoder_tb IS
  GENERIC (N: NATURAL := 3;          --# of bits at input
            T: TIME := 80 ns;         --stimulus period
            Tfinal: TIME := 800 ns);  --end of simulation time
END ENTITY;
------------------------------------------------------------------------
ARCHITECTURE testbench OF address_decoder_tb IS
  --Signal declarations:-----
  SIGNAL ena_tb: STD_LOGIC := '0';
  SIGNAL address_tb: STD_LOGIC_VECTOR(N-1 DOWNTO 0) := (OTHERS => '0');
  SIGNAL wordline_tb: STD_LOGIC_VECTOR(2**N-1 DOWNTO 0); --from circuit
  --DUT declaration:--------
  COMPONENT address_decoder IS
    PORT (ena: STD_LOGIC;
           address: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
           wordline: OUT STD_LOGIC_VECTOR(2**N-1 DOWNTO 0));
  END COMPONENT;
  BEGIN
  --DUT instantiation:-------
  dut: address_decoder PORT MAP (ena_tb, address_tb, wordline_tb);
  --Generate enable:
  ena_tb <= '1' AFTER T;
  --Generate address:--------
  PROCESS
    BEGIN
      --DUT instantiation:------
      dut: address_decoder PORT MAP (ena_tb, address_tb, wordline_tb);
      --Generate enable:
      ena_tb <= '1' AFTER T;
      --Generate address:-------
      WAIT FOR T;
    WHILE NOW<Tfinal LOOP --this loop could be unconditional
      WAIT FOR T;
      IF (address_tb < 2**N-1) THEN
        address_tb <= address_tb + 1;
      ELSE
        address_tb <= (OTHERS => '0');
      END IF;
      END LOOP;
    END PROCESS;
  END ARCHITECTURE;
------------------------------------------------------------------------

VHDL code for a type I testbench (file address_decoder_tb.vhd) is shown next. The input waveforms are those in the previous figure, with \( T = 80 \text{ ns} \) (line 9). The code construction is similar to that seen in section 10.8 (example 10.4). The statement in line 34 guarantees that after a certain time (800 ns in this example – see line 10) the simulation will end.
Simulation results (from ModelSim) are shown in the next figure. Note that, being it a type I (therefore functional) simulation, there are no propagation delays between the transitions of address_tb and wordline_tb. Observe also that the result produced by the circuit (wordline_tb) does match the expected result (wordline) shown in the previous figure.

Exercise 10.12: Type IV testbench for an address decoder

Note: Before examining this solution, please read the item “Additional Details on Type IV Simulation” in the “Extra Material” link of the book website.

The simulation stimuli (ena and address) to be used in this exercise are those of example 2.4 (figure 2.7), repeated in the figure below. From them, the expected functional response (ideal wordline, \( w_{\text{ideal}} \)) is that also included in the figure, already seen in the solution to exercise 10.11. The last waveform (worst-case wordline, \( w_{\text{worst}} \)) is the signal expected to be produced by the circuit, which will be used as a reference to define the comparison points and stability test intervals.

The design file (address_decoder.vhd) is the same seen in the previous exercise. A corresponding testbench file (address_decoder_tb.vhd) for type IV simulation is presented below. The maximum propagation delay for the device used in this project (a Cyclone II FPGA, from Altera, synthesized with Quartus II; for Xilinx devices, ISE would be the natural choice) was approximately 13 ns, so \( t_{\text{pmax}}=15 \text{ ns} \) was adopted (line 10). The stimulus (address) period is 80 ns (line 9), and both time margins are 1 ns (lines 11-12). The total simulation time is 800 ns (line 13).

The simulation procedure is that described in chapter 10 and appendix D (must include the SDF and delay-annotated files). See section 10.11 and example 10.6 for additional details.

```
1  --Testbench file (address_decoder_tb.vhd)-----------------------------------
2  --Note 1: Because GENERIC parameters cannot be passed to the .vho file, if the value of
3  --N below must be changed, then the design must be resynthesized to get a new .vho file.
4  LIBRARY ieee;
5  USE ieee.std_logic_1164.all;
6  USE ieee.std_logic_arith.all;
7  ENTITY address_decoder_tb IS
8    GENERIC (N: NATURAL := 3; --# of bits at input (see Note 1 above)
9      T: TIME := 80 ns; --stimulus period
10     tmax: TIME := 15 ns; --maximum propagation delay
11     Tbefore: TIME := 1 ns; --time margin before transition
12     Tafter: TIME := 1 ns; --time margin after transition
13     Tfinal: TIME := 800 ns); --end of simulation
14  END ENTITY;
15  ARCHITECTURE testbench OF address_decoder_tb IS
16    --Constant and signal declarations:---------
17    CONSTANT T1: TIME := Tmax + Tafter; --16ns
18    CONSTANT T2: TIME := T - T1 - Tbefore; --63ns
19    SIGNAL ena_tb: STD_LOGIC := '0';
20    SIGNAL wordline_lb: STD_LOGIC_VECTOR(N-1 DOWNTO 0) := (OTHERS => '0');
21    SIGNAL wordline_glb: STD_LOGIC_VECTOR(2**N-1 DOWNTO 0) := (OTHERS => '1'); --ideal
22    SIGNAL wordline_ideal: STD_LOGIC_VECTOR(2**N-1 DOWNTO 0) := (OTHERS => '1'); --ideal
23    SIGNAL wordline_ideal: STD_LOGIC_VECTOR(2**N-1 DOWNTO 0); --actual circuit output
```
--DUT declaration:-----------------------------
COMPONENT address_decoder IS --see Note 1 above
  PORT (
    ena: STD_LOGIC;
    address: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
    wordline: OUT STD_LOGIC_VECTOR(2**N-1 DOWNTO 0));
END COMPONENT;
BEGIN
  --DUT instantiation (see Note 1 above):------
  dut: address_decoder PORT MAP (ena_tb, address_tb, wordline_tb);
  --Generate enable:---------------------------
  ena_tb <= '1' AFTER T;
  --Generate address + expected ideal output:--
  PROCESS
    VARIABLE count: NATURAL RANGE 0 TO 2**N-1 := 0;
    BEGIN
      WAIT FOR T;
      WHILE NOW<Tfinal LOOP --this loop could be unconditional
        address_tb <= conv_std_logic_vector(count, N);
        wordline_ideal <= (count=>'0', OTHERS=>'1');
        WAIT FOR T;
        IF (count < 2**N-1) THEN
          count := count + 1;
        ELSE
          count := 0;
        END IF;
      END LOOP;
    END PROCESS;
  --Comparison + stability test:---------------
  PROCESS
    BEGIN
      IF (NOW<Tfinal) THEN
        WAIT FOR T1;
        ASSERT (wordline_tb=wordline_ideal)
        REPORT "Error: Signal values are not equal!"
        SEVERITY FAILURE;
        WAIT FOR T2;
        ASSERT wordline_tb'STABLE(T2)
        REPORT "Error: Signal is unstable!"
        SEVERITY FAILURE;
        WAIT FOR Tbefore;
      ELSE
        ASSERT FALSE
        REPORT "END OF SIMULATION: No error found!"
        SEVERITY FAILURE;
      END IF;
    END PROCESS;
END ARCHITECTURE;
-----------------------------------------------------------------
Simulation results are depicted in the figure below. Observe that the actual output (wordline_tb) does exhibit propagation delays (of the order of 13 ns for the chosen device) after corresponding address (address_tb) transitions.

The following (very important) challenges are left to the reader:
1) To play with the testbench file. For example, what happens if the value of t_{max} is reduced to a value below the actual propagation delay (say, 8 ns)? And what happens if line 60 is changed to “ASSERT wordline_tb'STABLE(T2 + 5 ns) . . .”?
2) To include in the ASSERT statements the code needed for the simulator to display the time value and the signal values in case an error is found (as in example 10.6).

Exercise 11.4: Zero-to-nine counter
The adjusted state transition diagram is shown above. The only difference is that $T (=1s)$ is now a condition for the system to progress from one state to another (see timed machines in section 11.6). A corresponding VHDL code is shown below. $T$ was set to 50M, which was assumed to be the clock frequency, so the system changes its state after every second. The SSD was considered to be a common-anode display (so a ‘0’ lights a segment, while a ‘1’ turns it off). See additional comments after the code.

---

```vhdl
ENTITY slow_counter IS
  GENERIC (T: NATURAL := 50_000_000); --T=fclk -> 1s
  PORT (clk, rst: IN BIT;
         output: OUT BIT_VECTOR(6 DOWNTO 0));
END ENTITY;
```

```vhdl
ARCHITECTURE fsm OF slow_counter IS
  TYPE state IS (zero, one, two, three, four, five,
                  six, seven, eight, nine);
  SIGNAL pr_state, nx_state: state;
  SIGNAL counter_output: NATURAL RANGE 0 TO 9;
BEGIN

  ----Lower section of FSM:-----------
  PROCESS (rst, clk)
  VARIABLE count: NATURAL RANGE 0 TO T;
  BEGIN
    IF (rst='1') THEN
      pr_state <= zero;
    ELSIF (clk'EVENT AND clk='1') THEN
      count := count + 1;
      IF (count=T) THEN
        count := 0;
        pr_state <= nx_state;
      END IF;
    END IF;
  END PROCESS;

  ----Upper section of FSM:-----------
  PROCESS (pr_state)
  BEGIN
    CASE pr_state IS
      WHEN zero =>
        counter_output <= 0;
        nx_state <= one;
      WHEN one =>
        counter_output <= 1;
        nx_state <= two;
      WHEN two =>
        counter_output <= 2;
        nx_state <= three;
      WHEN three =>
        counter_output <= 3;
        nx_state <= four;
      WHEN four =>
        counter_output <= 4;
        nx_state <= five;
      WHEN five =>
        counter_output <= 5;
        nx_state <= six;
      WHEN six =>
        counter_output <= 6;
        nx_state <= seven;
      WHEN seven =>
        counter_output <= 7;
        nx_state <= eight;
      WHEN eight =>
        counter_output <= 8;
        nx_state <= nine;
      zero =>
        "0000"
      one =>
        "0001"
      two =>
        "0010"
      three =>
        "0011"
      four =>
        "0100"
      five =>
        "0101"
      six =>
        "0110"
      seven =>
        "0111"
      eight =>
        "1000"
      nine =>
        "1001"
      END CASE;
  END PROCESS;
END;
```

---
WHEN nine =>
counter_output <= 9;
end CASE;
END PROCESS;
---BCD-to-SSD converter:------
BEGIN
CASE counter_output IS
WHEN 0 => output<="0000001";  --"0" on SSD
WHEN 1 => output<="1001111";  --"1" on SSD
WHEN 2 => output<="0010010";  --"2" on SSD
WHEN 3 => output<="0000110";  --"3" on SSD
WHEN 4 => output<="1001100";  --"4" on SSD
WHEN 5 => output<="0100100";  --"5" on SSD
WHEN 6 => output<="0100000";  --"6" on SSD
WHEN 7 => output<="0001111";  --"7" on SSD
WHEN 8 => output<="0000000";  --"8" on SSD
WHEN 9 => output<="0000100";  --"9" on SSD
WHEN OTHERS => output<="0110000"; --"E"rror
END CASE;
END PROCESS;
END ARCHITECTURE;
------------------------------------------------------
To make the implementation clearer, the BCD-to-SSD converter (SSD driver) was implemented in a separate process, but it could be nested inside the code for the upper section of the FSM, as shown below:

---------------
CASE pr_state IS
  WHEN zero =>
    output <= "0000001";
    nx_state <= one;
  WHEN one =>
    output <= "1001111";
    nx_state <= two;
  ...
---------------

Exercise 11.6: Signal generator #2

a) The state transition diagram is included in the figure above. A 0-to-39 counter was chosen to control the state transitions (it counts from 0-to-19 in one state, hence 20 clock periods, then 0-to-39 in the other state, thus 40 clock cycles), but recall that any counter with 40 states would do [Pedroni 2008]. Another option would be to use a 60-state counter, which would only be reset after reaching 59. A VHDL code for the state machine shown in the figure is presented below. Because the output coincides with the state representation (that is, output=pr_state), the output comes directly from a flip-flop, so the circuit is automatically glitch-free. (Suggestion: to ease the inspection of the simulation results use 2 and 4 instead of 20 and 40 for the generic parameters PulsesLow and PulsesHigh, respectively.)
Exercise 11.11: Preventing state-bypass with a flag #2

This exercise deals with a crucial aspect of state machines: state bypass. This problem arises when we want the machine to stay in a certain state until a predefined condition “re-occurs”.

In figure (a) above, if the machine is in state A and a long x='1' occurs, it goes to B, then moves immediately to C at the next clock edge (that is, it stays in B during only one clock period). Sometimes this is exactly what we want, so there is no actual state bypass. However, in many cases (like the car alarm seen in section 11.5), we want the machine to proceed to the next state only if a certain condition happens anew. For example, in figure (b), the machine moves from A to B if x='1'; after arriving in B, only if a new x='1' occurs it should move to C, thus requiring x to return to '0' before a state transition is again considered. A VHDL code for that FSM (with a flag) is presented below. Simulation results are shown subsequently.

Note: In some applications the machine is only required to stay in certain (or all) states for a certain amount of time (that is, a certain number of clock cycles) before the input conditions are again evaluated. This must not be confused with the state-bypass problem, because this is simply a timed machine (described in section 11.6).
The view produced by the state machine viewer (with the `enum_encoding` attribute commented out) is shown below.

Simulation results:

Exercise 11.15: FSM with embedded timer #2

First we need to clarify the diagram above. There are two options for the timed transitions (B-C and C-A). Taking the B-C transition, for example, the specifications can be the as follows.
i) The machine must stay \( \text{time1} \) seconds in B and then evaluate \( x \); if \( x=2 \) after that time interval, then the machine must proceed to state C.

ii) The value of \( x \) must be \( x=2 \) for at least \( \text{time1} \) seconds; therefore, if \( x \) changes its value before \( \text{time1} \) has been completed, the time counting must restart.

Both options can occur, depending on the application. In the present solution, we will consider that case (ii) is wanted. A corresponding VHDL code follows. Note that because the \( \text{pr}_\text{state} <= \text{nx}_\text{state} \) assignment only occurs when all conditions are met, there is no need for specifying other next states besides the actual next state in each state specifications (that is, in state A, the only possible next state is B; in B, it is C; and in C, it is A).

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fsm IS
  GENERIC (
    time1: NATURAL := 3; -- # of clock periods in B
    time2: NATURAL := 4); -- # of clock periods in C
  PORT (
    clk, rst: IN STD_LOGIC;
    x: IN NATURAL RANGE 0 TO 3;
    y: OUT STD_LOGIC);
END ENTITY;

ARCHITECTURE fsm OF fsm IS
  TYPE state IS (A, B, C);
  SIGNAL pr_state, nx_state: state;
  ATTRIBUTE enum_encoding: STRING;
  ATTRIBUTE enum_encoding OF state: TYPE IS "sequential";
  SIGNAL T: NATURAL RANGE 0 TO time2;
  SIGNAL input1, input2: NATURAL RANGE x'RANGE;
BEGIN
  ----Lower section of FSM:---------
  PROCESS (rst, clk)
  VARIABLE count: NATURAL RANGE 0 TO time2;
  BEGIN
    IF (rst='1') THEN
      pr_state <= A;
    ELSIF (clk'EVENT AND clk='1') THEN
      IF (x=input1 OR x=input2) THEN
        count := count + 1;
        IF (count=T) THEN
          count := 0;
          pr_state <= nx_state;
        ELSE
          count := 0;
          END IF;
        END IF;
      END IF;
  END PROCESS;
  ----Upper section of FSM:---------
  PROCESS (pr_state, x)
  BEGIN
    CASE pr_state IS
      WHEN A =>
        y <= '1';
        T <= 1;
        input1<=0; input2<=1;
        nx_state <= B;
      WHEN B =>
        y <= '0';
        T <= time1;
        input1<=2; input2<=2;
        nx_state <= C;
      WHEN C =>
        y <= '1';
        T <= time2;
        input1<=3; input2<=3;
        nx_state <= A;
      END CASE;
    END PROCESS;
  END ARCHITECTURE;
```

The view produced by the state machine viewer (with the \text{enum_encoding} \ attribute commented out) is shown below.
Simulation results:

Exercise 12.2: Counter with LCD display

A VHDL code for this problem is presented below. The code for the up/down counter, with 0.5 seconds in each state, is shown explicitly, while the parts already seen in examples in the book are just indicated. The clock frequency (entered as a generic parameter) was assumed to be 50 MHz. (Suggestion: See first the design in section 12.2.)

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
PACKAGE my_functions IS
  FUNCTION integer_to_lcd (SIGNAL input: NATURAL) RETURN STD_LOGIC_VECTOR;
  ... (insert function integer_to_lcd seen in sec. 12.2)
END my_functions;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_functions.all;

ENTITY counter_with_LCD IS
  GENERIC (fclk: POSITIVE := 50_000_000; --clock frequency (50MHz)
            clk_divider: POSITIVE := 50_000);  --for LCD clock (500Hz)
  PORT (clk, rst, up: IN STD_LOGIC;
        RS, RW, LCD_ON, BKL_ON: OUT STD_LOGIC;
        E: BUFFER STD_LOGIC;
        DB: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END ENTITY;

ARCHITECTURE counter_with_LCD OF counter_with_LCD IS
  TYPE state IS (FunctionSet1, FunctionSet2, FunctionSet3, FunctionSet4,
                 ClearDisplay, DisplayControl, EntryMode, WriteData, ReturnHome);
  SIGNAL pr_state, nx_state: state;
  SIGNAL counter_output: NATURAL RANGE 0 TO 15;
  SIGNAL lcd_input: STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN
  --Counter (up/down, 0.5s per state):-----
  PROCESS (clk, rst, up)
  VARIABLE counter1: INTEGER RANGE 0 TO fclk/2;
  VARIABLE counter2: INTEGER RANGE 0 TO 15;
  BEGIN
    IF (rst='1') THEN
      counter1 := 0;
      IF (up='1') THEN counter2 := 0;
      ELSE counter2 := 15;
      END IF;
    ELSIF (clk'EVENT AND clk='1') THEN
      IF (up='1') THEN
        IF (counter2 /= 15) THEN
          counter1 := counter1 + 1;
        ELSE
          counter1 := 0;
        END IF;
      ELSE
        IF (counter2 /= 15) THEN
          counter1 := counter1 - 1;
        ELSE
          counter1 := 15;
        END IF;
      END IF;
    END PROCESS;

    --LCD display
    IF (counter_output = 0) THEN
      lcd_input := "00000000";
    ELSE
      IF (counter_output = 15) THEN
        lcd_input := "11111111";  --Display 9
      ELSE
        lcd_input := tile(counter_output, 8);
      END IF;
    END IF;

    --State transitions
    --... (insert state transition logic)
END BEGIN;
```
Exercise 12.13: Frequency meter with Gray counter

The subject of this exercise is the very interesting and important issue of clock domain crossing. Only a brief visit to the subject is presented here, with the help of the figure below.

Figure (a) shows two interconnected domains. They operate with distinct (asynchronous with respect to each other) clocks, called clk1 and clk2. The problem is that a positive edge in clk2 can occur while the output of DFF1 (q1) is changing its value; in such a case, because the setup/hold times of DFF2 are not respected, a metastable state (a state between '0' and '1', which normally takes a relatively long time to settle) can occur at the output of DFF2.

The most popular synchronizer for clock domain crossing is depicted in figure (b), consisting simply of two-stage shift register, which causes the data signal (data) to be received without metastable states downstream. Another alternative is depicted in (c), which is adequate for the case when data is the output of a counter. Because in a gray counter [Pedroni 2008] only one bit changes at a time, if clk2 activates the register while it is changing its value (that is, when this change occurs during the setup + hold times of the register), at most one bit will be stored with incorrect value, so the output cannot be more than one unit off the actual value. This, however, does not prevent the register from producing a metastable state, which can be cleaned using the same arrangement of figure (b), resulting the combined implementation of figure (d).

In terms of hardware usage, the solution with a synchronizer (figure (b)) requires two extra registers, with N flip-flops each (N is the number of bits in data), while the solution with just a gray counter (figure (c)) does not require additional flip-
flops, but does require additional combinational logic, consisting of a layer with \(N-1\) XOR gates. Obviously, the solution in figure (d) requires the sum of both.

A straightforward implementation for a gray counter is to implement a regular sequential counter then convert its output to gray code using the equations below, where \(b\) is the output from the binary counter and \(g\) is the corresponding gray representation:

\[
g(N-1) = b(N-1) \\
g(i) = b(i) \oplus b(i+1) \text{ for } i = 0, 1, ..., N-2
\]

A corresponding circuit is shown in the figure below, for \(N=4\).

---

**Exercise 13.5: ROM implemented with a HEX file #1**

The only changes needed in the code of section 13.4, which employed CONSTANT to implement the ROM, are in the architecture declarations, replacing the original text with that below:

```vhdl
ARCHITECTURE rom OF rom_with_hex_file IS
  SIGNAL reg_address: INTEGER RANGE 0 TO 15;
  TYPE memory IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
  SIGNAL myrom: memory;
  ATTRIBUTE ram_init_file: STRING;
  ATTRIBUTE ram_init_file OF myrom: SIGNAL IS "rom_contents.hex";
BEGIN

Remember to create a plain text file with the contents below (see the construction of MIF files in section 13.3), saved with the name `rom_contents.hex` in the work directory:

```
: 10 0000 00 00 FF 1A 05 50 B0 00 00 00 00 00 00 00 11 C1
: 00 0000 01 FF
```

---

**Exercise 13.9: Synchronous RAM**

See code for `memory3` in section 20.6 of [Pedroni 2008].

---

**Exercise 14.10: I^2C interface for an RTC**

*Note: Before examining this solution, please read the topic “Additional Details on the I^2C Interface” in the “Extra Material” part of the book website.*

PCF8563, PCF8573, PCF8583, and PCF8593 are part of the NXP family of RTC (real time clock) devices. For availability reasons, the RTC employed in this solution was PFC8593, which is an RTC with clock/calendar plus timer/alarm options. The clock/calendar features were explored in this design, which are set using the first eight memory bytes, shown in figure 1.
Circuit Design and Simulation with VHDL, 2nd edition, Volnei A. Pedroni, MIT Press

Figure 1. First eight bytes of the PCF8593 register.

In the VHDL code, the eight registers above (addresses from 0 to 7) were filled with the following values:
Control: “00000000”
Subseconds: “00000000” (0.00 seconds)
Seconds: “00000000” (00 seconds)
Minutes: “00110000” (30 minutes)
Hours: “00010100” (14 hours, 24-hour option chosen)
Year/date: “00000001” (day 01)
Weekday/month: “00010001” (month 11 = November)
Timer: “00000000”

The setup for the experiments is shown in figure 2.

VHDL code for this exercise is presented next. Note that data was written to all 8 RTC registers of figure 1, but only information regarding seconds, minutes, and hours was retrieved, which illustrates the use of different starting addresses for writing and reading. Note also that the initial time (14:30:00 HH/MM/SS) was entered using GENERIC, but several other possibilities exist. Finally, observe that the clocking scheme is similar to that in section 14.4 of the book, but in “Additional Details on the I2C Interface” in the “Extra Material” part of the book website would also do.

1234567890123456789012345678901234567890
---Package with a function:-------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
PACKAGE my_functions IS
FUNCTION bin_to_ssd (CONSTANT input: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
END my_functions;
-----------------------------------------------------------------------------------
PACKAGE BODY my_functions IS
FUNCTION bin_to_ssd (CONSTANT input: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR IS
VARIABLE output: STD_LOGIC_VECTOR(6 DOWNTO 0);
BEGIN
CASE input IS
WHEN "0000" => output:="0000001";   --"0" on SSD
WHEN "0001" => output:="1001111";   --"1" on SSD
END CASE;
END;
END;

Figure 2. Setup for the experiments.

<table>
<thead>
<tr>
<th>Register name</th>
<th>Address</th>
<th>Bit7</th>
<th>Bit6</th>
<th>Bit5</th>
<th>Bit4</th>
<th>Bit3</th>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subseconds</td>
<td>1</td>
<td>1/10sec</td>
<td></td>
<td>1/100sec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seconds</td>
<td>2</td>
<td>10sec</td>
<td></td>
<td>sec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minutes</td>
<td>3</td>
<td>10min</td>
<td></td>
<td>min</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hours</td>
<td>4</td>
<td>AM/PM</td>
<td>12/24</td>
<td>10hour</td>
<td>hour</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Year / Date</td>
<td>5</td>
<td>year</td>
<td>10date</td>
<td>date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weekday/Month</td>
<td>6</td>
<td>weekday</td>
<td>10mo</td>
<td>month</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timer</td>
<td>7</td>
<td>Timer setup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
WHEN "0010" => output:="0010010";   --"2" on SSD
WHEN "0011" => output:="0000110";   --"3" on SSD
WHEN "0100" => output:="1001100";   --"4" on SSD
WHEN "0101" => output:="0100100";   --"5" on SSD
WHEN "0110" => output:="0100000";  --"6" on SSD
WHEN "0111" => output:="0001111";  --"7" on SSD
WHEN "1000" => output:="0000000";  --"8" on SSD
WHEN "1001" => output:="0000100";  --"9" on SSD
WHEN "1010" => output:="0001000";  --"A" on SSD
WHEN "1011" => output:="1100000";  --"b" on SSD
WHEN "1100" => output:="0110001";  --"C" on SSD
WHEN "1101" => output:="1000010";  --"d" on SSD
WHEN "1110" => output:="0110000";  --"E" on SSD
WHEN OTHERS => output:="0111000";  --"F" on SSD
END CASE;
RETURN output;
END PACKAGE BODY;
-----------------------------------------------------------------------------------
---Main code:------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_functions.all; --package with "integer_to_ssd" function.
---------------------------------------------------------------------------
ENTITY rtc_i2c IS
GENERIC (
--System paramenters:
fclk: POSITIVE := 50_000; --Freq. of system clock (in kHz)
data_rate: POSITIVE := 100; --Desired I2C bus speed (in kbps)
slave_addr_for_write: STD_LOGIC_VECTOR(7 DOWNTO 0):= "10100010";
slave_addr_for_read: STD_LOGIC_VECTOR(7 DOWNTO 0) := "10100011";
initial_address_wr: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00000000";
initial_address_rd: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00000010";
--Values to store in the RTC memory (clock/calendar settings):
set_control: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00000000";
set_subsec: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00000000"; --0.00 sec
set_sec: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00000000";   --00 sec
set_min: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00110000";   --30 min
set_hour: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00010100";   --14 h
set_date: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00000001";   --01 day
set_month: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00010001";  --11 November
set_timer: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00000000");
PORT (
--Clock and control signals:
clk, rst, wr, rd: IN STD_LOGIC;
--I2C signals:
SCL: OUT STD_LOGIC;
SDA: INOUT STD_LOGIC;
--System outputs (to SSD displays):
chip_rst: OUT STD_LOGIC;
ssd_sec: OUT STD_LOGIC_VECTOR(6 DOWNTO 0);   --units of seconds
ssd_10sec: OUT STD_LOGIC_VECTOR(6 DOWNTO 0);  --tens of seconds
ssd_min: OUT STD_LOGIC_VECTOR(6 DOWNTO 0);   --units of minutes
ssd_10min: OUT STD_LOGIC_VECTOR(6 DOWNTO 0);  --tens of minutes
ssd_hour: OUT STD_LOGIC_VECTOR(6 DOWNTO 0);   --units of hours
ssd_10hour: OUT STD_LOGIC_VECTOR(6 DOWNTO 0); --tens of hours
ack_error: OUT STD_LOGIC); --to an LED
END ENTITY;
-----------------------------------------------------------------------------------
ARCHITECTURE fsm OF rtc_i2c IS
--General signals:
CONSTANT divider: INTEGER := (fclk/8)/data_rate;
SIGNAL timer: NATURAL RANGE 0 TO 8;
SIGNAL aux_clk, bus_clk, data_clk: STD_LOGIC;
SIGNAL wr_flag, rd_flag: STD_LOGIC;
SIGNAL sec: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL min: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL hour: STD_LOGIC_VECTOR(7 DOWNTO 0);
SHARED VARIABLE i: NATURAL RANGE 0 TO 7 := 0;
--State-machine signals:
TYPE state IS (
idle, start_wr, slave_addr_wr, ack1, stop,
--Write-only states:
initial_addr_wr, ack2, wr_control, ack3, wr_subsec, ack4, wr_sec, ack5, wr_min, ack6,
wr_hour, ack7, wr_date, ack8, wr_month, ack9, wr_timer, ack10,
--Read-only states:
initial_addr_rd, ack11, start_rd, slave_addr_rd, ack12, rd_sec, ack13, rd_min, ack14,
rd_hour, no_ack);
SIGNAL pr_state, nx_state: state;
BEGIN
chip_rst <= NOT rst; --to reset the chip (pin 3)
-------Display signals:--------------------
ssd_sec <= bin_to_ssd(sec(3 DOWNTO 0));
ssd_10sec <= bin_to_ssd(sec(7 DOWNTO 4));
ssd_min <= bin_to_ssd(min(3 DOWNTO 0));
ssd_10min <= bin_to_ssd(min(7 DOWNTO 4));
ssd_hour <= bin_to_ssd(hour(3 DOWNTO 0));
ssd_10hour <= bin_to_ssd("00" & hour(5 DOWNTO 4));
-------Auxiliary Clock (400kHz):-----------
PROCESS (clk)
VARIABLE count1: INTEGER RANGE 0 TO divider;
BEGIN
IF (clk'EVENT AND clk='1') THEN
  count1 := count1 + 1;
  IF (count1=divider) THEN
    aux_clk <= NOT aux_clk;
    count1 := 0;
  END IF;
END IF;
END PROCESS;
-------Bus & data clocks (100kHz):--------
PROCESS (aux_clk)
VARIABLE count2: NATURAL RANGE 0 TO 3;
BEGIN
IF (aux_clk'EVENT AND aux_clk='1') THEN
  count2 := count2 + 1;
  IF (count2=0) THEN
    bus_clk <= '0';
  ELSIF (count2=1) THEN
    data_clk <= '1';
  ELSIF (count2=2) THEN
    bus_clk <= '1';
  ELSE
    data_clk <= '0';
  END IF;
END IF;
END PROCESS;
-------Lower section of FSM:---------------
PROCESS (rst, data_clk)
VARIABLE error_flag: STD_LOGIC;
BEGIN
IF (rst='1') THEN
  pr_state <= idle;
  wr_flag <= '0';
  rd_flag <= '0';
  error_flag := '0';
  i := 0;
  --Enter data for I2C bus:
ELSIF (data_clk'EVENT AND data_clk='1') THEN
  IF (i=timer-1) THEN
    pr_state <= nx_state;
    i := 0;
  ELSIF (data_clk'EVENT AND data_clk='0') THEN
    --Store data read from RTC memory:
    IF (pr_state=rd_sec) THEN
      sec(7-i) <= SDA;
    ELSIF (pr_state=rd_min) THEN
      min(7-i) <= SDA;
    ELSIF (pr_state=rd_hour) THEN
      hour(7-i) <= SDA;
    END IF;
    --Store write/read/error flags:
    IF (pr_state=idle) THEN
      wr_flag <= wr;
      rd_flag <= rd;
    ELSIF (pr_state=stop) THEN
      wr_flag <= '0';
      rd_flag <= '0';
    ELSIF (pr_state=ack1 OR pr_state=ack2 OR pr_state=ack3 OR pr_state=ack4 OR
      pr_state=ack5 OR pr_state=ack6 OR pr_state=ack7 OR pr_state=ack8 OR pr_state=ack9
    THEN
      wr_flag <= wr;
      rd_flag <= rd;
OR pr_state=ack10) THEN
error_flag := error_flag OR SDA;
END IF;
END PROCESS;

CASE pr_state IS
--common write-read states:
WHEN idle =>
SCL <= '1';
SDA <= '1';
timer <= 1;
IF (wr_flag='1' OR rd_flag='1') THEN
nx_state <= start_wr;
ELSE
nx_state <= idle;
END IF;

WHEN start_wr =>
SCL <= '1';
SDA <= data_clk;--or '0'
timer <= 1;
nx_state <= slave_addr_wr;

WHEN slave_addr_wr =>
SCL <= bus_clk;
SDA <= slave_addr_for_write(7-i);
timer <= 8;
nx_state <= ack1;

WHEN ack1 =>
SCL <= bus_clk;
SDA <= '2';
timer <= 1;
IF (wr_flag='1') THEN
nx_state <= initial_addr_wr;
ELSE
nx_state <= initial_addr_rd;
END IF;

--data-write states:
WHEN initial_addr_wr =>
SCL <= bus_clk;
SDA <= initial_address_wr(7-i);
timer <= 8;
nx_state <= ack2;

WHEN ack2 =>
SCL <= bus_clk;
SDA <= '2';
timer <= 1;
nx_state <= wr_control;

WHEN wr_control =>
SCL <= bus_clk;
SDA <= set_control(7-i);
timer <= 8;
nx_state <= ack3;

WHEN ack3 =>
SCL <= bus_clk;
SDA <= '2';
timer <= 1;
nx_state <= wr_subsec;

WHEN wr_subsec =>
SCL <= bus_clk;
SDA <= set_subsec(7-i);
timer <= 8;
nx_state <= ack4;

WHEN ack4 =>
SCL <= bus_clk;
SDA <= '2';
timer <= 1;
nx_state <= wr_sec;

WHEN wr_sec =>
SCL <= bus_clk;
SDA <= set_sec(7-i);
timer <= 8;
nx_state <= ack5;

WHEN ack5 =>
SCL <= bus_clk;
SDA <= '2';
timer <= 1;

WHEN wr_min =>
  SCL <= bus_clk;
  SDA <= set_min(7-i);
  timer <= 8;
  nx_state <= < ack6;

WHEN ack6 =>
  SCL <= bus_clk;
  SDA <= 'Z';
  timer <= 1;
  nx_state <= < wr_hour;

WHEN wr_hour =>
  SCL <= bus_clk;
  SDA <= set_hour(7-i);  
  timer <= 8;
  nx_state <= < ack7;

WHEN ack7 =>
  SCL <= bus_clk;
  SDA <= 'Z';
  timer <= 1;
  nx_state <= < wr_date;

WHEN wr_date =>
  SCL <= bus_clk;
  SDA <= set_date(7-i);
  timer <= 8;
  nx_state <= < ack8;

WHEN ack8 =>
  SCL <= bus_clk;
  SDA <= 'Z';
  timer <= 1;
  nx_state <= < wr_month;

WHEN wr_month =>
  SCL <= bus_clk;
  SDA <= set_month(7-i);
  timer <= 8;
  nx_state <= < ack9;

WHEN ack9 =>
  SCL <= bus_clk;
  SDA <= 'Z';
  timer <= 1;
  nx_state <= < wr_timer;

WHEN wr_timer =>
  SCL <= bus_clk;
  SDA <= set_timer(7-i);
  timer <= 8;
  nx_state <= < ack10;

WHEN ack10 =>
  SCL <= bus_clk;
  SDA <= 'Z';
  timer <= 1;
  nx_state <= < stop;

-- data-read states:

WHEN initial_addr_rd =>
  SCL <= bus_clk;
  SDA <= initial_address_rd(7-i);
  timer <= 8;
  nx_state <= < ack11;

WHEN ack11 =>
  SCL <= bus_clk;
  SDA <= 'Z';
  timer <= 1;
  nx_state <= < start_rd;

WHEN start_rd =>
  SCL <= '1'; -- or bus_clk;
  SDA <= data_clk;
  timer <= 1;
  nx_state <= < slave_addr_rd;

WHEN slave_addr_rd =>
  SCL <= bus_clk;
  SDA <= slave_addr_for_read(7-i);
  timer <= 8;
  nx_state <= < ack12;

WHEN ack12 =>
  SCL <= bus_clk;
  SDA <= 'Z';
  timer <= 1;
  nx_state <= < rd_sec;
WHEN rd_sec =>
SCL <= bus_clk;
SDA <= 'Z';
timer <= 8;
nx_state <= ack13;
WHEN ack13 =>
SCL <= bus_clk;
SDA <= '0';
timer <= 1;
nx_state <= rd_min;
WHEN rd_min =>
SCL <= bus_clk;
SDA <= 'Z';
timer <= 8;
nx_state <= ack14;
WHEN ack14 =>
SCL <= bus_clk;
SDA <= '0';
timer <= 1;
nx_state <= rd_hour;
WHEN rd_hour =>
SCL <= bus_clk;
SDA <= 'Z';
timer <= 8;
nx_state <= no_ack;
WHEN no_ack =>
SCL <= bus_clk;
SDA <= '1';
timer <= 1;
nx_state <= stop;
--Common write-read state:
WHEN stop =>
SCL <= NOT data_clk; --or '0'
timer <= 1;
nx_state <= idle;
END CASE;
END PROCESS;
END ARCHITECTURE;

Exercise 15.1: 800x600x75Hz SVGA interface

All that is needed is to replace the values of \(H_{\text{pulser}}, H_{\text{BP}}, H_{\text{active}}, \text{VF}_{\text{pulser}}, \text{VF}_{\text{BP}}, \text{VF}_{\text{active}}\), and \(V_{\text{Fp}}\) with those for the SVGA standard. Such values are listed in section 17.3 and are used in the design of section 17.4.

Exercise 15.2: Image generation with hardware #1 (banner)

A major advantage of breaking the video interfaces into standard and non-standard parts, as done in chapters 15-17, is that most of the design stays always the same, so once it has been understood, doing other designs becomes relatively simple. To solve the present exercise, we only need to replace the code for Part 2 (image generator, lines 81-115) in section 15.9 with a new PROCESS, for the new image, which can be easily written based on the solution to exercise 17.2 shown ahead.

Exercise 15.3: Image generation with hardware #2 (sun in the sky)

As mentioned above, a major advantage of breaking the video interfaces into standard and non-standard parts, as done in chapters 15-17, is that most of the design stays always the same. To solve the present exercise, we only need to replace the code for Part 2 (image generator) in section 15.9 with a new code, for the new image, because the rest (control signals) remain the same. A code that solves this exercise is shown below (just replace lines 78-115 of the code in section 15.9 with this code).

```vhdl
--Part 2: IMAGE GENERATOR
---------------------------------------------
PROCESS (pixel_clk, Hsync, dens)
```
Exercise 15.4: Image generation with hardware #3 (filling with green)

As mentioned above, only the non-standard part of the VGA interface needs to be replaced. Take the code of section 15.9 and replace the code for Part 2 (image generator, lines 81-115) with a new PROCESS, for the new image, which can be easily written based on the solution to exercise 17.4, shown ahead.

Exercise 15.6: Image generation with hardware #5 (digital clock)

Again, the same code used in section 15.9 can be employed. Just replace Part 2 (image generator, lines 81-115) with a new code, for the new image, which can be based directly on the design presented in section 17.5. Note, however, that in the latter the image generator was constructed in two parts to ease its comprehension. Proper horizontal and vertical parameter adjustments might be needed, depending on the chosen display resolution.

Exercise 16.2: Image generation with hardware #1 (banner)

The same code used in section 16.7 can be employed here because it was already separated into standard and non-standard parts. The non-standard part (image generator) must be replaced with a code for the new image, while the other sections (control, serializer, etc.) remain the same. In summary, take the code of section 16.7 and replace its Part 1 (image generator) with a code similar to that shown in the solution to exercise 17.2, shown ahead. Make proper horizontal and vertical parameter adjustments according with the display resolution that you have chosen. Do not forget to create a file for the PLL (see sections 14.2 and 16.7).

Exercise 16.3: Image generation with hardware #2 (sun in the sky)

As mentioned above, the same code used in section 16.7 can be employed here because it was already separated into standard and non-standard parts. Take the code of section 16.7 and replace its Part 1 (image generator) with a code for the new image, which can be easily written based on the solution to exercise 17.3, shown ahead. Recall that the circle’s...
center must be at the center of the screen, so its coordinates must be adjusted according with the resolution chosen for the display. Do not forget to create a file for the PLL (see sections 14.2 and 16.7).

Exercise 16.4: Image generation with hardware #3 (filling with green)

Again, the same code used in section 16.7 can be employed. Just replace Part 1 (image generator) with a code for the new image, which can be easily written based on the solution to exercise 17.4, shown ahead. Make proper horizontal and vertical parameter adjustments according with the display resolution that you have chosen. Do not forget to create a file for the PLL (see sections 14.2 and 16.7).

Exercise 16.6: Image generation with hardware #5 (wall clock)

Again, the same code used in section 16.7 can be employed here because it was already separated into standard and non-standard parts. Only Part 1 (image generator) needs to be replaced. Since a wall clock was designed in section 17.5, that code can be used here. In summary, take the code of section 16.7 and replace its Part 1 with Part 1 and Part 2 of the wall clock (in section 17.5, the image generator was constructed in two parts to ease its comprehension). The only changes needed in the code brought from section 17.5 are adjustments in the horizontal and vertical parameters. Do not forget to create a file for the PLL (see sections 14.2 and 16.7).

Exercise 17.2: Image generation with hardware C1 (banner)

As seen in the examples of chapters 15-17, a major advantage of separating the video interfaces in standard and non-standard parts is that most of them stay always the same. Thus the code of section 17.4 can be used here, replacing only Part 1 (image generator) with the code below. Obviously, because the image generator is instantiated in the main code, some adjustments might be needed in the declaration and instantiation of the image generator there. Recall also that the screen here is 600×800 pixels. Do not forget to create a file for the PLL (see sections 14.2 and 17.4).

```vhdl
1 ----Image generator:-----------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------------------------------
5 ENTITY image_generator IS
6 PORT ( red_switch, green_switch, blue_switch: IN STD_LOGIC;
7    clk50, Hsync, Vsync, Hactive, Vactive, dena: IN STD_LOGIC;
8    R, G, B: OUT STD_LOGIC_VECTOR(5 DOWNTO 0));
9 END image_generator;
10 ---------------------------------------------------------------
11 ARCHITECTURE image_generator OF image_generator IS
12 BEGIN
13     PROCESS (clk50, dena)
14     BEGIN
15         VARIABLE x: INTEGER RANGE 0 TO 800;
16         VARIABLE y: INTEGER RANGE 0 TO 600;
17         BEGIN
18             ----Count columns:-----------------------------
19             IF (clk50 EVENT AND clk50='1') THEN
20                 IF (Hactive='1') THEN x := x + 1;
21                 ELSE x := 0;
22                 END IF;
23             END IF;
24             ----Count lines:-----------------------------
25             IF (Hsync EVENT AND Hsync='1') THEN
26                 IF (Vactive='1') THEN y := y + 1;
```
ELSE y := 0;
END IF;
END IF;

--Generate the image:---------------------
IF (dena='1') THEN
  IF (x<700-y) THEN
    R <= (OTHERS => red_switch);
    G <= (OTHERS => green_switch);
    B <= (OTHERS => blue_switch);
  ELSE
    R <= (OTHERS => NOT red_switch);
    G <= (OTHERS => NOT green_switch);
    B <= (OTHERS => NOT blue_switch);
  END IF;
ELSE
  R <= (OTHERS => '0');
  G <= (OTHERS => '0');
  B <= (OTHERS => '0');
END IF;
END PROCESS;
END image_generator;
----------------------------------------------------------

Exercise 17.3: Image generation with hardware #2 (sun in the sky)

As seen in the examples of chapters 15-17, a major advantage of separating the video interface in standard and non-standard parts is that most of them stay always the same. Thus the code of section 17.4 can be used here, replacing only Part 1 (image generator) with the code below. Obviously, because the image generator is instantiated in the main code, some adjustments might be needed in the declaration and instantiation of the image generator there. Recall that the screen size is 600×800 pixels and that the sun’s radius must be 150 pixels (see the figure below). Do not forget to create a file for the PLL (see sections 14.2 and 17.4).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY image_generator IS
  PORT (red_switch, green_switch, blue_switch: IN STD_LOGIC;
        clk50, Hsync, Vsync, Hactive, Vactive, dena: IN STD_LOGIC;
        R, G, B: OUT STD_LOGIC_VECTOR(5 DOWNTO 0));
END image_generator;
-----------------------------------------------
ARCHITECTURE image_generator OF image_generator IS
BEGIN
  PROCESS (clk50, Hsync, dena)
  VARIABLE x: INTEGER RANGE 0 TO 800;
  VARIABLE y: INTEGER RANGE 0 TO 600;
  BEGIN
    --Count columns:------------------------
    IF (clk50'EVENT AND clk50='1') THEN
      IF (Hactive='1') THEN x := x + 1;
      ELSE x := 0;
      END IF;
    END IF;
    --Count lines:------------------------
    IF (Hsync'EVENT AND Hsync='1') THEN
      IF (Vactive='1') THEN y := y + 1;
      ELSE y := 0;
      END IF;
    END IF;
    --Generate the image:-------------------
    IF (dena='1') THEN
```
Exercise 17.4: Image generation with hardware #3 (filling with green)

As mentioned above, a major advantage of separating the video interface in standard and non-standard parts is that most of them stay always the same. Thus the code of section 17.4 can be used here, replacing only Part 1 (image generator) with the code below. Do not forget to create a file for the PLL (see sections 14.2 and 17.4).
B <= (OTHERS => '0');
ELSE
R <= (OTHERS => red_switch);
G <= (OTHERS => green_switch);
B <= (OTHERS => blue_switch);
END IF;
END IF;
ELSE
R <= (OTHERS => '0');
G <= (OTHERS => '0');
B <= (OTHERS => '0');
END IF;
END PROCESS;
END image_generator;
----------------------------------------------------------