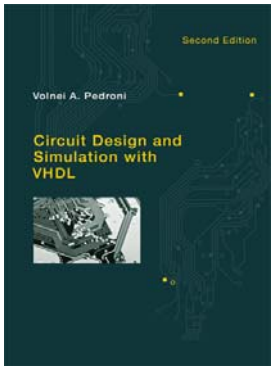


Circuit Design and Simulation with VHDL
2nd edition
Volnei A. Pedroni
MIT Press, 2010
Book web: www.vhdl.us

Solutions Manual
(v4)

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

Extended and Advanced Designs	Exercises covered
Chapter 11: VHDL Design of State Machines	4, 6, 11, 15
Chapter 12: VHDL Design with Basic Displays	2, 13
Chapter 13: VHDL Design of Memory Circuits	5, 9
Chapter 14: VHDL Design of Serial Communications Circuits	10
Chapter 15: VHDL Design of VGA Video Interfaces	1, 2, 3, 4, 6
Chapter 16: VHDL Design of DVI Video Interfaces	2, 3, 4, 6
Chapter 17: VHDL Design of FPD-Link Video Interfaces	2, 3, 4

Exercise 2.1: Multiplexer

a) Code

```
 1  ------------------------------------------------
 2  LIBRARY ieee;
 3  USE ieee.std_logic_1164.all;
 4  ------------------------------------------------
 5  ENTITY mux IS
 6     PORT (
 7        a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
 8        sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0));
 9        x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
10  END mux; --or END ENTITY;
11  ---------------------------------------
12  ARCHITECTURE example OF mux IS
13  BEGIN
14    PROCESS (a, b, sel)
15    BEGIN
16      IF (sel="00") THEN
17        x <= "00000000";
19      ELSIF (sel="01") THEN
20        x <= a;
21      ELSIF (sel="10") THEN
22        x <= b;
23      ELSE
24        x <= "ZZZZZZZZ";
25      END IF;
26    END PROCESS;
27  END example; --or END ARCHITECTURE;
28  ------------------------------------------------
```

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

```
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

```
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)
SIGNAL s5: INTEGER RANGE -35 TO 35;      --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 SLL 3 = "010000"`
b) `x SLA -2 = "111100"`
c) `x SRA 2 = "111100"`
d) `x ROL 1 = "100101"`
e) `x ROR -3 = "010110"`

a) `x(2 DOWNTO 0) & "000";`
b) `x(5) & x(5) & x(5 DOWNTO 2);`
c) `x(5) & x(5) & x(5 DOWNTO 2);`
d) `x(4 DOWNTO 0) & x(5);`
e) `x(2 DOWNTO 0) & x(5 DOWNTO 3);`

Exercise 4.9: Arithmetic operators for signed types

```
a) x <= a + b;                    --either numeric_std or std_logic_arith is fine (see note 1)
b) x <= b + c;                    --either numeric_std or std_logic_arith is fine (see note 1)
c) x <= 3*b;                      --only numeric_std is fine (see note 2)
d) x <= 3*a + b;                  --only numeric_std is fine (see note 3)
e) x <= a + c + "1111";           --only numeric_std is fine (see notes 1 and 4)
f) x <= a + c + SIGNED'("1111"); --either numeric_std or std_logic_arith is fine (see note 5)
```

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 *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"
or *a*="000", *b*="001", *c*="011", *d*="111", *e*="110", *f*="100"
d) One-hot: *a*="000001", *b*="000010", *c*="000100", *d*="001000", *e*="010000", *f*="100000"

Exercise 5.4: Generic parity generator

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

```
1  ------------------------------------------------
2  ENTITY parity_generator IS
3     GENERIC (N: INTEGER := 8); --number of bits
4  PORT (
5     x: IN BIT_VECTOR(N-1 DOWNTO 0);
6     y: OUT BIT_VECTOR(N DOWNTO 0));
7  END ENTITY;
8  ------------------------------------------------
9  ARCHITECTURE structural OF parity_generator IS
10    SIGNAL internal: BIT_VECTOR(N-1 DOWNTO 0);
11 BEGIN
12    internal(0) <= x(0);
13    gen: FOR i IN 1 TO N-1 GENERATE
14       internal(i) <= internaL(i-1) XOR x(i);
15    END GENERATE;
16    y <= internal(N-1) & x;
17 END structural;
18 ------------------------------------------------
```

Exercise 5.7: Hamming weight with GENERATE

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

```
----------------------------------------------------------------------
ENTITY hamming_weigth 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_weigth OF hamming_weigth 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.

```
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
--------------------------------------------------------------------------------
ENTITY adder IS
   GENERIC (
      N: NATURAL := 4); --number of bits
   PORT (
      a, b: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
      cin: IN STD_LOGIC;
      opcode: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
      y: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
      cout: OUT STD_LOGIC);
END ENTITY;
--------------------------------------------------------------------------------
ARCHITECTURE adder OF adder IS
   --define internal unsigned signals:
   SIGNAL a_unsig, b_unsig: UNSIGNED(N-1 DOWNTO 0);
   SIGNAL y_unsig: UNSIGNED(N DOWNTO 0);
   --define internal signed signals:
   SIGNAL a_sig, b_sig: SIGNED(N-1 DOWNTO 0);
   SIGNAL y_sig: SIGNED(N DOWNTO 0);
BEGIN
   --convert inputs to unsigned:
   a_unsig <= unsigned(a);
   b_unsig <= unsigned(b);
   --convert inputs to signed:
   a_sig <= signed(a);
   b_sig <= signed(b);
   --implement circuit and convert to std_logic:
   WITH opcode(1 DOWNTO 0) SELECT
      y_unsig <=  ('0' & a_unsig) + ('0' & b_unsig) WHEN "00",
                  ('0' & a_unsig) - ('0' & b_unsig) WHEN "01",
                  ('0' & b_unsig) - ('0' & a_unsig) WHEN "10",
                  ('0' & a_unsig) + ('0' & b_unsig) + ('0' & cin) WHEN OTHERS;
   WITH opcode(1 DOWNTO 0) SELECT
      y_sig <= (a_sig(N-1) & a_sig) + (b_sig(N-1) & b_sig) WHEN "00",
               (a_sig(N-1) & a_sig) - (b_sig(N-1) & b_sig) WHEN "01",
                - (a_sig(N-1) & a_sig) + (b_sig(N-1) & b_sig) WHEN "10",
               (a_sig(N-1) & a_sig) + (b_sig(N-1) & b_sig) + ('0' & cin) WHEN OTHERS;
   WITH opcode(2) SELECT
      y <=  std_logic_vector(y_unsig(N-1 DOWNTO 0)) WHEN '0',
            std_logic_vector(y_sig(N-1 DOWNTO 0)) WHEN OTHERS;
   WITH opcode(2) SELECT
      cout <=  std_logic(y_unsig(N)) WHEN '0',
               std_logic(y_sig(N)) WHEN OTHERS;
END ARCHITECTURE;
--------------------------------------------------------------------------------
```

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.

```
---Package:-----------------------------------------------------
PACKAGE my_data_types IS
   TYPE matrix IS ARRAY (NATURAL RANGE <>, NATURAL RANGE <>) OF BIT;
END PACKAGE my_data_types;
-----------------------------------------------------------------
```

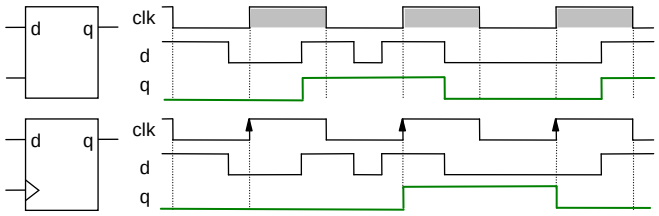
```
---Main code:----------------------------------
USE work.my_data_types.all;
-----------------------------------------------
ENTITY generic_mux IS
   GENERIC (
      M: POSITIVE := 8;    --number of inputs
      N: POSITIVE := 4);   --size of each input
   PORT (
      x: IN MATRIX (0 TO M-1, N-1 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*×*N*. Though this will not affect the result, such a "linearization" might be confusing sometimes, so it is usually not recommended.

```
-------------------------------------------------------
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

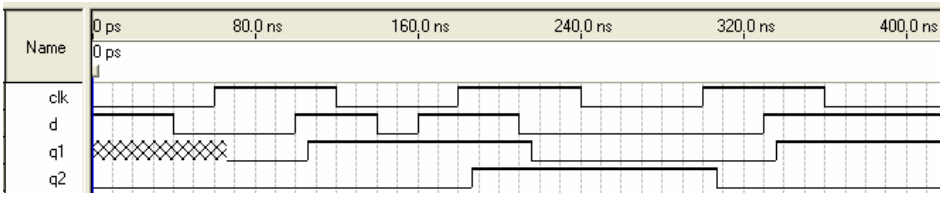
a) See figure below.



b) VHDL code and simulation:

```
1  ----------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------
5  ENTITY example IS
6     PORT (
7        d, clk: IN STD_LOGIC;
8        q1, q2: OUT STD_LOGIC);
9  END ENTITY;
10 ----------------------------------
11 ARCHITECTURE example OF example IS
12 BEGIN
13    PROCESS(clk, d)
14    BEGIN
15       ---Latch:--------
16       IF clk='1' THEN
17          q1 <= d;
18       END IF;
19       ---Flip-flop:----
20       IF clk'EVENT AND clk='1' THEN --or IF rising_edge(clk) THEN
21          q2 <= d;
22       END IF;
23    END PROCESS;
24 END ARCHITECTURE;
25 ----------------------------------
```

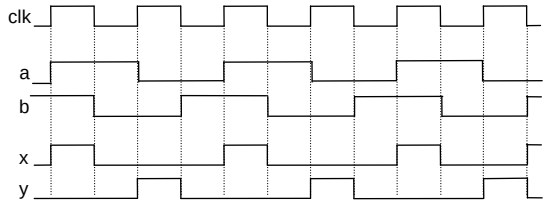
Simulation results:



Exercise 6.4: Generic registered multiplexer

Just add a register (*N* flip-flops) to the output of the mux 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).

```
---------------------------------------
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

```
-----------------------------------------------------
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.)

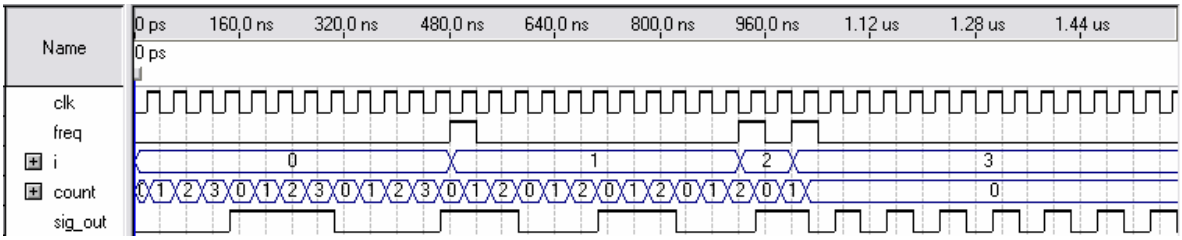
a) The dividers must be even if we want to operate in only one (say, rising) clock edge (if the clock's duty cycle is unknown that is the only option).
50MHz/10kHz = 5000 (error = 0 Hz)

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).

```
1  ------------------------------------------------------
2  ENTITY sig_generator IS
3     GENERIC (
4        fclk: POSITIVE := 50_000_000);
5     PORT (
6        clk, freq: IN BIT;
7        sig_out: BUFFER BIT);
8  END ENTITY;
9  ------------------------------------------------------
10 ARCHITECTURE sig_generator OF sig_generator IS
11    TYPE ROM IS ARRAY (0 TO 3) OF POSITIVE RANGE 1 TO 8;
12    CONSTANT coefficients: ROM := (8, 6, 4, 2);
13    SIGNAL divider: POSITIVE RANGE 1 TO 8;
14 BEGIN
15    PROCESS (freq)
16       VARIABLE i: NATURAL RANGE 0 TO 4;
17    BEGIN
18       IF (freq'EVENT AND freq='1') THEN --or IF rising_edge(freq) THEN
19          i := i + 1;
20          IF (i=4) THEN
21             i := 0;
22          END IF;
23       END IF;
24       divider <= coefficients(i);
25    END PROCESS;
26    ----------------------
27    PROCESS (clk)
28       VARIABLE count: NATURAL RANGE 0 TO 4;
29    BEGIN
30       IF (clk'EVENT AND clk='1') THEN --or IF rising_edge(clk) THEN
31          count := count + 1;
32          IF (count>=divider/2) THEN --It is ">="
33             count := 0;
34             sig_out <= NOT sig_out;
35          END IF;
36       END IF;
37    END PROCESS;
38 END ARCHITECTURE;
39 ------------------------------------------------------
```

c) Simulation results:



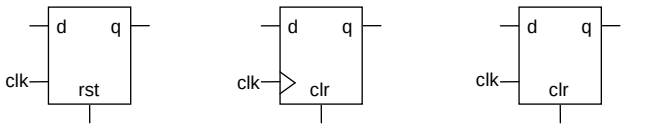
Exercise 6.13: Programmable Signal Generator with Frequency Meter

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 fuction), which does not happen in this code.

c) Same as (a) above.

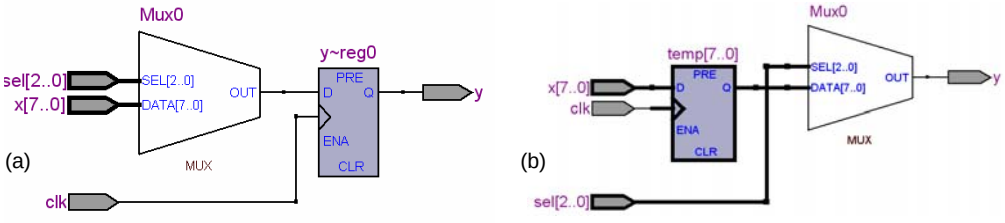
d) Same as (b) above.

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.



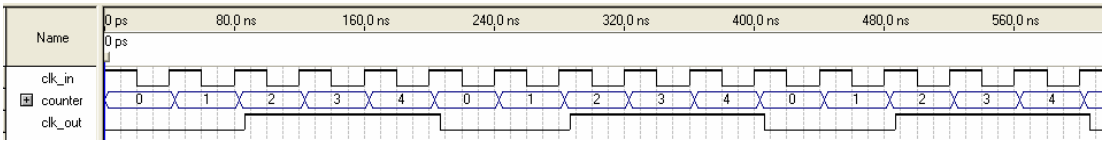
Exercise 7.9: Frequency divider with VARIABLE

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 $\lceil \log_2 M \rceil$ 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):

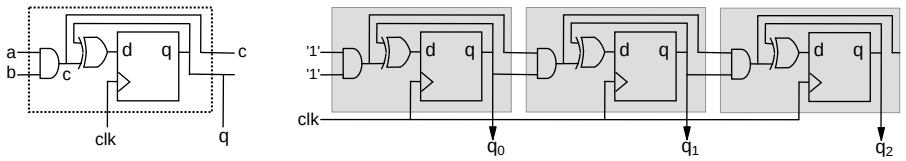
```
1  ------------------------------------------------
2  ENTITY clock_divider IS
3     GENERIC (
4        M: NATURAL := 5;
5     PORT (
6        clk_in: IN BIT;
7        clk_out: OUT BIT);
8  END ENTITY;
9  ------------------------------------------------
10 ARCHITECTURE circuit OF clock_divider IS
11 BEGIN
12    PROCESS (clk_in)
13       VARIABLE counter: NATURAL RANGE 0 TO M;
14    BEGIN
15       IF clk_in'EVENT AND clk_in='1' THEN --or IF rising_edge(clk_in) THEN
16          counter := counter + 1;
17          IF counter=M/2 THEN
18             clk_out <= '1';
19          ELSIF counter=M THEN
20             clk_out <= '0';
21             counter := 0;
22          END IF;
23       END IF;
24    END PROCESS;
25 END ARCHITECTURE;
26 ------------------------------------------------
```

Simulation results for M=5:



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*.

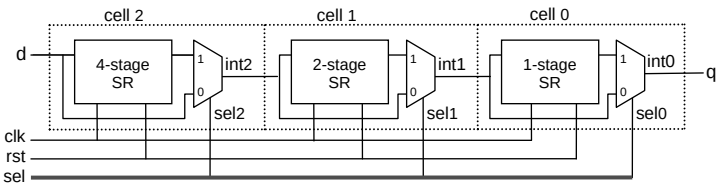
```
----The component:-------------------------
ENTITY counter_cell IS
   PORT (
      clk, a, b: IN BIT;
      c, q: BUFFER BIT);
END ENTITY;
-----------------------------------------
ARCHITECTURE counter_cell OF counter_cell IS
   SIGNAL d: BIT;
BEGIN
   PROCESS (clk, a, b)
   BEGIN
      c <= a AND 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.

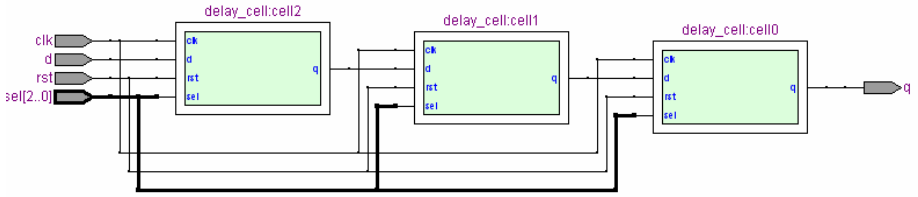



```
----The component:----------------------------
ENTITY delay_cell IS
```

```
   GENERIC (
      M: NATURAL);
   PORT (
      d, clk, rst, sel: IN BIT;
      q: OUT BIT);
END ENTITY;
-----------------------------------------------
ARCHITECTURE delay_cell OF delay_cell IS
   SIGNAL internal: BIT_VECTOR(0 TO M-1);
BEGIN
   PROCESS (clk, rst)
   BEGIN
      IF (rst='1') THEN
         internal <= (OTHERS => '0');
      ELSIF (clk'EVENT AND clk='1') THEN
         IF (M=1) THEN
            internal(0) <= d;
         ELSE
            internal <= d & internal(0 TO M-2);
         END IF;
      END IF;
   END PROCESS;
   q <= d WHEN sel='0' ELSE internal(M-1);
END ARCHITECTURE;
-----------------------------------------------
```

```
----Main code:-------------------------------------------------------------------------
ENTITY tapped_delay_line IS
   GENERIC (
      M: NATURAL := 4);
   PORT (
      d, clk, rst: IN BIT;
      sel: IN BIT_VECTOR(2 DOWNTO 0);
      q: OUT BIT);
END ENTITY;
-----------------------------------------------------------------------------------------
ARCHITECTURE structural OF tapped_delay_line IS
   SIGNAL int: BIT_VECTOR(2 DOWNTO 0);
   --component declaration:
   COMPONENT delay_cell IS
      GENERIC (
         M: NATURAL);
      PORT (
         d, clk, rst, sel: IN BIT;
         q: OUT BIT);
   END COMPONENT;
BEGIN
   cell2: delay_cell GENERIC MAP (M=>4) PORT MAP (d, clk, rst, sel(2), int(2));
   cell1: delay_cell GENERIC MAP (M=>2) PORT MAP (int(2), clk, rst, sel(1), int(1));
   cell0: delay_cell GENERIC MAP (M=>1) PORT MAP (int(1), clk, rst, sel(0), int(0));
   q <= int(0);
END ARCHITECTURE;
-----------------------------------------------------------------------------------------
```

Result shown by the RTL Viewer:



Exercise 9.1: ASSERT statement #1

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

```
   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.

```
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       ------------------------------------------------
17    FUNCTION my_not (a: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR IS
18       ALIAS aa: STD_LOGIC_VECTOR(1 TO a'LENGTH) IS a;
19       VARIABLE result: STD_LOGIC_VECTOR(1 TO a'LENGTH);
20    BEGIN
21       FOR i IN result'RANGE LOOP
22          result(i) := not_table (aa(i));
23       END LOOP;
24       RETURN result;
25    END FUNCTION;
26 END PACKAGE BODY;
27 ------------------------------------------------------------------
```

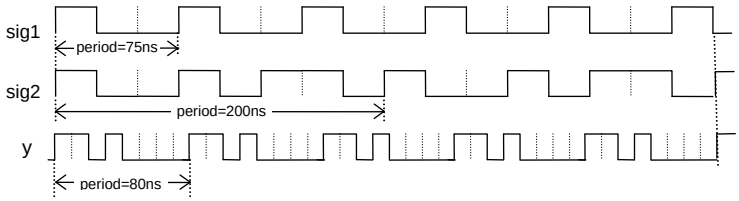
```
1  ----Main code:-----------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_package.all;
5  ------------------------------------------
6  ENTITY test IS
7     PORT (
8        x: IN STD_LOGIC_VECTOR(8 DOWNTO 1);
9        y: OUT STD_LOGIC_VECTOR(2 TO 9));
10 END ENTITY;
11 ------------------------------------------
12 ARCHITECTURE example OF test IS
13 BEGIN
14    y <= my_not(x);
15 END ARCHITECTURE;
16 ------------------------------------------
```

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

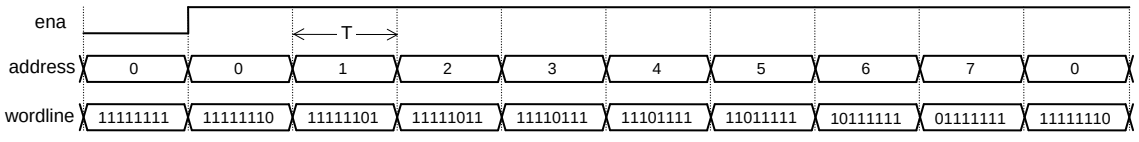
All three signals are depicted in the figure below, followed by a VHDL code that produces all three.
Note that the WAIT FOR statement was used in this solution. The reader is invited to redo it using AFTER. Which is simpler in this case?




```
 1  ------------------------------------------
 2  LIBRARY ieee;
 3  USE ieee.std_logic_1164.all;
 4  ------------------------------------------
 5  ENTITY stimuli_generator IS
 6  END ENTITY;
 7  ------------------------------------------
 8  ARCHITECTURE testbench OF stimuli_generator IS
 9     SIGNAL sig1, sig2, y: STD_LOGIC;
10  BEGIN
11     PROCESS
12     BEGIN
13        --signal sig1:--
14        sig1 <= '1';
15        WAIT FOR 25ns;
16        sig1 <= '0';
17        WAIT FOR 50ns;
18        --signal sig2:--
19        sig2 <= '1';
20        WAIT FOR 25ns;
21        sig2 <= '0';
22        WAIT FOR 50ns;
23        sig2 <= '1';
24        WAIT FOR 25ns;
25        sig2 <= '0';
26        WAIT FOR 25ns;
27        sig2 <= '1';
28        WAIT FOR 50ns;
29        sig2 <= '0';
30        WAIT FOR 25ns;
31        --signal y:--
32        y <= '1';
33        WAIT FOR 20ns;
34        y <= '0';
35        WAIT FOR 10ns;
36        y <= '1';
37        WAIT FOR 10ns;
38        y <= '0';
39        WAIT FOR 40ns;
40     END PROCESS;
41  END ARCHITECTURE;
42  ------------------------------------------
```

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.

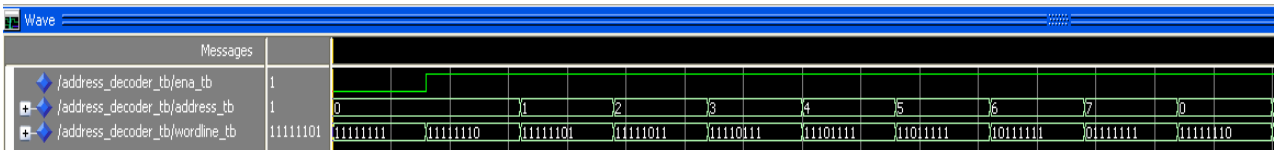
```
1  -----Design file (address_decoder.vhd):-----------------
```

```
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;
-------------------------------------------------------
```

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 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.

```
-----Testbench file (address_decoder_tb.vhd):--------------------------
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
      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;
-------------------------------------------------------------------------
```

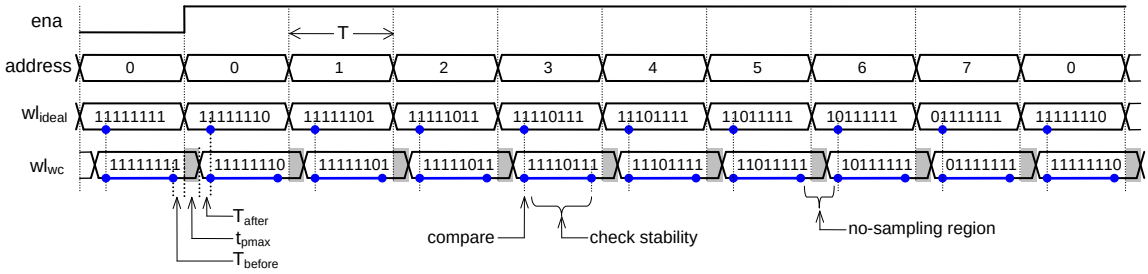
Simulation results (from ModelSim) are shown in the next figure. Note that, being it a type I (therefore functional) simulation, there are no propagation delays between the transitions of *address_tb* and *wordline_tb*. Observe also that the result produced by the circuit (*wordline_tb*) does match the 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, wl_{ideal}) is that also included in the figure, already seen in the solution to exercise 10.11. The last waveform (worst-case wordline, wl_{wc}) 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_{pmax}=15 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.

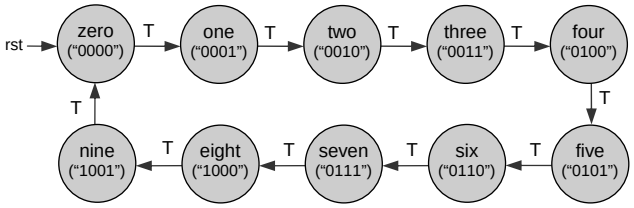
```
1  --Testbench file (address_decoder_tb.vhd)-----------------------------------------------
2  --Note 1: Because GENERIC parameters cannot be passed to the .vho file, if the value of
3  --N below must be changed, then the design must be resynthesized to get a new .vho file.
4  LIBRARY ieee;
5  USE ieee.std_logic_1164.all;
6  USE ieee.std_logic_arith.all;
7  ------------------------------------------------------------------------------------------
8  ENTITY address_decoder_tb IS
9     GENERIC (
10       N: NATURAL := 3;          --# of bits at input (see Note 1 above)
11       T: TIME := 80 ns;         --stimulus period
12       tpmax: TIME := 15 ns;     --maximum propagation delay
13       Tbefore: TIME := 1 ns;    --time margin before transition
14       Tafter: TIME := 1 ns;     --time margin after transition
15       Tfinal: TIME := 800 ns);  --end of simulation
16 END ENTITY;
17 ------------------------------------------------------------------------------------------
18 ARCHITECTURE testbench OF address_decoder_tb IS
19    --Constant and signal declarations:----------
20    CONSTANT T1: TIME := Tpmax + Tafter;    --16ns
21    CONSTANT T2: TIME := T - T1 - Tbefore;  --63ns
22    SIGNAL ena_tb: STD_LOGIC := '0';
23    SIGNAL address_tb: STD_LOGIC_VECTOR(N-1 DOWNTO 0) := (OTHERS => '0');
24    SIGNAL wordline_ideal: STD_LOGIC_VECTOR(2**N-1 DOWNTO 0) := (OTHERS => '1'); --ideal
25    SIGNAL wordline_tb: STD_LOGIC_VECTOR(2**N-1 DOWNTO 0); --actual circuit output
```

```
26     --DUT declaration:--------------------------
27     COMPONENT address_decoder IS  --see Note 1 above
28        PORT (
29           ena: STD_LOGIC;
30           address: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
31           wordline: OUT STD_LOGIC_VECTOR(2**N-1 DOWNTO 0));
32     END COMPONENT;
33  BEGIN
34     --DUT instantiation (see Note 1 above):------
35     dut: address_decoder PORT MAP (ena_tb, address_tb, wordline_tb);
36     --Generate enable:--------------------------
37     ena_tb <= '1' AFTER T;
38     --Generate address + expected ideal output:--
39     PROCESS
40        VARIABLE count: NATURAL RANGE 0 TO 2**N-1 := 0;
41     BEGIN
42        WAIT FOR T;
43        WHILE NOW<Tfinal LOOP --this loop could be unconditional
44           address_tb <= conv_std_logic_vector(count, N);
45           wordline_ideal <= (count=>'0', OTHERS=>'1');
46           WAIT FOR T;
47           IF (count < 2**N-1) THEN
48              count := count + 1;
49           ELSE
50              count := 0;
51           END IF;
52        END LOOP;
53     END PROCESS;
54     --Comparison + stability test:---------------
55     PROCESS
56     BEGIN
57        IF (NOW<Tfinal) THEN
58           WAIT FOR T1;
59           ASSERT (wordline_tb=wordline_ideal)
60              REPORT "Error: Signal values are not equal!"
61              SEVERITY FAILURE;
62           WAIT FOR T2;
63           ASSERT wordline_tb'STABLE(T2)
64              REPORT "Error: Signal is unstable!"
65              SEVERITY FAILURE;
66           WAIT FOR Tbefore;
67        ELSE
68           ASSERT FALSE
69              REPORT "END OF SIMULATION: No error found!"
70              SEVERITY FAILURE;
71        END IF;
72     END PROCESS;
73  END ARCHITECTURE;
74  ---------------------------------------------------------------------
```

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_{pmax} 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.

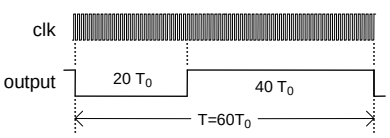
```
1  ------------------------------------------------------
2  ENTITY slow_counter IS
3     GENERIC (
4        T: NATURAL := 50_000_000); --T=fclk -> 1s
5     PORT (
6        clk, rst: IN BIT;
7        output: OUT BIT_VECTOR(6 DOWNTO 0));
8  END ENTITY;
9  ------------------------------------------------------
10 ARCHITECTURE fsm OF slow_counter IS
11    TYPE state IS (zero, one, two, three, four, five,
12       six, seven, eight, nine);
13    SIGNAL pr_state, nx_state: state;
14    SIGNAL counter_output: NATURAL RANGE 0 TO 9;
15 BEGIN
16    ----Lower section of FSM:-----------
17    PROCESS (rst, clk)
18       VARIABLE count: NATURAL RANGE 0 TO T;
19    BEGIN
20       IF (rst='1') THEN
21          pr_state <= zero;
22       ELSIF (clk'EVENT AND clk='1') THEN
23          count := count + 1;
24          IF (count=T) THEN
25             count := 0;
26             pr_state <= nx_state;
27          END IF;
28       END IF;
29    END PROCESS;
30    ----Upper section of FSM:-----------
31    PROCESS (pr_state)
32    BEGIN
33       CASE pr_state IS
34          WHEN zero =>
35             counter_output <= 0;
36             nx_state <= one;
37          WHEN one =>
38             counter_output <= 1;
39             nx_state <= two;
40          WHEN two =>
41             counter_output <= 2;
42             nx_state <= three;
43          WHEN three =>
44             counter_output <= 3;
45             nx_state <= four;
46          WHEN four =>
47             counter_output <= 4;
48             nx_state <= five;
49          WHEN five =>
50             counter_output <= 5;
51             nx_state <= six;
52          WHEN six =>
53             counter_output <= 6;
54             nx_state <= seven;
55          WHEN seven =>
56             counter_output <= 7;
57             nx_state <= eight;
58          WHEN eight =>
59             counter_output <= 8;
60             nx_state <= nine;
```

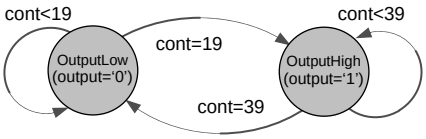
```
61          WHEN nine =>
62             counter_output <= 9;
63             nx_state <= zero;
64       END CASE;
65    END PROCESS;
66    ----BCD-to-SSD converter:-----------
67    PROCESS (counter_output)
68    BEGIN
69       CASE counter_output IS
70          WHEN 0 => output<="0000001";  --"0" on SSD
71          WHEN 1 => output<="1001111";  --"1" on SSD
72          WHEN 2 => output<="0010010";  --"2" on SSD
73          WHEN 3 => output<="0000110";  --"3" on SSD
74          WHEN 4 => output<="1001100";  --"4" on SSD
75          WHEN 5 => output<="0100100";  --"5" on SSD
76          WHEN 6 => output<="0100000";  --"6" on SSD
77          WHEN 7 => output<="0001111";  --"7" on SSD
78          WHEN 8 => output<="0000000";  --"8" on SSD
79          WHEN 9 => output<="0000100";  --"9" on SSD
80          WHEN OTHERS => output<="0110000"; --"E"rror
81       END CASE;
82    END PROCESS;
83 END ARCHITECTURE;
84 -------------------------------------------------------
```

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



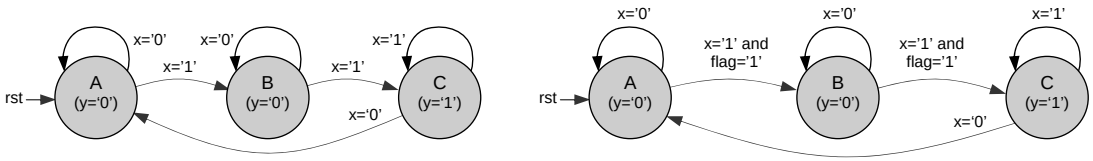


a) The state transition diagram is included in the figure above. A 0-to-39 counter was chosen to control the state transitions (it counts from 0-to-19 in one state, hence 20 clock periods, then 0-to-39 in the other state, thus 40 clock cycles), but recall that any counter with 40 states would do [Pedroni 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.)

```
1  -------------------------------------------------------------------
2  ENTITY signal_generator IS
3     GENERIC (
4        PulsesLow: NATURAL := 20;     --time in state OutputLow
5        PulsesHigh: NATURAL := 40);   --time in state OutputHigh
6     PORT (
7        clk, rst: IN BIT;
8        output: OUT BIT);
9  END ENTITY;
10 -------------------------------------------------------------------
11 ARCHITECTURE fsm OF signal_generator IS
12    TYPE state IS (OutputLow, OutputHigh);
13    SIGNAL pr_state, nx_state: state;
14    SIGNAL number_of_pulses: NATURAL RANGE 0 TO PulsesHigh;
15 BEGIN
16    ----Lower section of FSM:-----------
```

```
   PROCESS (rst, clk)
      VARIABLE count: NATURAL RANGE 0 TO PulsesHigh;
   BEGIN
      IF (rst='1') THEN
         pr_state <= OutputLow;
      ELSIF (clk'EVENT AND clk='1') THEN
         count := count + 1;
         IF (count=number_of_pulses) 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 OutputLow =>
            output <= '0';
            number_of_pulses <= PulsesLow;
            nx_state <= OutputHigh;
         WHEN OutputHigh =>
            output <= '1';
            number_of_pulses <= PulsesHigh;
            nx_state <= OutputLow;
      END CASE;
   END PROCESS;
END ARCHITECTURE;
-------------------------------------------------------------------
```

Exercise 11.11: Preventing state-bypass with a flag #2



(a) Original FSM. (b) FSM with a flag to avoid state-bypass.

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".

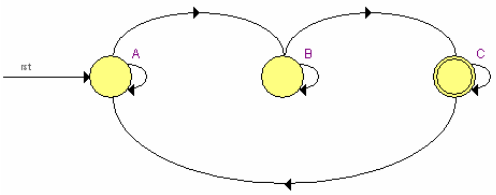
In figure (a) above, if the machine is in state A and a long *x*='1' occurs, it goes to B, then moves immediately to C at the next clock edge (that is, it stays in B during only one clock period). Sometimes this is exactly what we want, so there is no actual state bypass. However, in many cases (like 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).

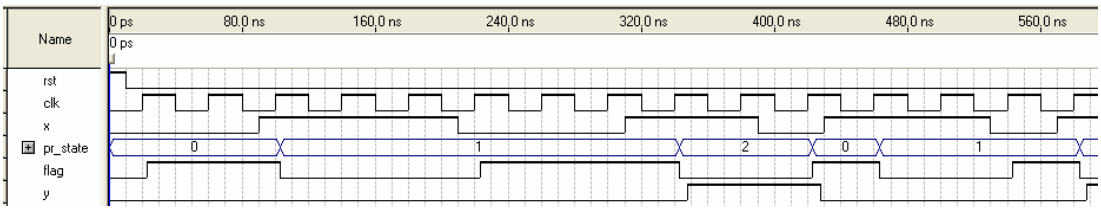
```
-------------------------------------------------------
ENTITY fsm_without_bypass IS
   PORT (
      clk, rst, x: IN BIT;
      y: OUT BIT);
END ENTITY;
-------------------------------------------------------
ARCHITECTURE fsm OF fsm_without_bypass IS
   TYPE state IS (A, B, C);
   SIGNAL pr_state, nx_state: state;
   ATTRIBUTE enum_encoding: STRING;
   ATTRIBUTE enum_encoding OF state: TYPE IS "sequential";
   SIGNAL flag: BIT;
BEGIN
   ----Flag generator:--------------
```

```
16     PROCESS (clk, rst)
17     BEGIN
18        IF (rst='1') THEN
19           flag <= '0';
20        ELSIF (clk'EVENT AND clk='1') THEN
21           IF (x='0') THEN
22              flag <= '1';
23           ELSE
24              flag <= '0';
25           END IF;
26        END IF;
27     END PROCESS;
28     ----Lower section of FSM:--------
29     PROCESS (rst, clk)
30     BEGIN
31        IF (rst='1') THEN
32           pr_state <= A;
33        ELSIF (clk'EVENT AND clk='1') THEN
34           pr_state <= nx_state;
35        END IF;
36     END PROCESS;
37     ----Upper section of FSM:--------
38     PROCESS (pr_state, x, flag)
39     BEGIN
40        CASE pr_state IS
41           WHEN A =>
42              y <= '0';
43              IF (x='1' AND flag='1') THEN nx_state <= B;
44              ELSE nx_state <= A;
45              END IF;
46           WHEN B =>
47              y <= '0';
48              IF (x='1' AND flag='1') THEN nx_state <= C;
49              ELSE nx_state <= B;
50              END IF;
51           WHEN C =>
52              y <= '1';
53              IF (x='0') THEN nx_state <= A;
54              ELSE nx_state <= C;
55              END IF;
56        END CASE;
57     END PROCESS;
58  END ARCHITECTURE;
59  -------------------------------------------------------------
```

The view produced by the state machine viewer (with the *enum_encoding* attribute commented out) is shown below.



Simulation results:



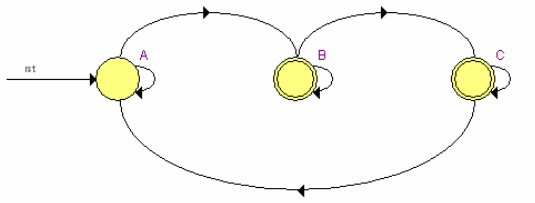
Exercise 11.15: FSM with embedded timer #2

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.

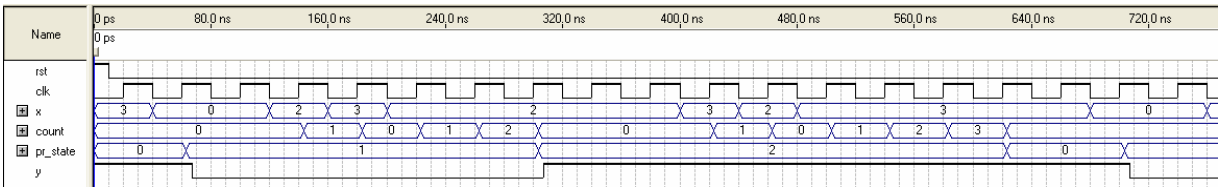
i) The machine must stay *time1* seconds in B and then evaluate *x*; if *x*=2 after that time interval, then the machine must proceed to state C.
ii) The value of *x* must be *x*=2 for at least *time1* seconds; therefore, if *x* changes its value before *time1* has been completed, the time counting must 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 "pr_state <= nx_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).

```
1  ------------------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------------------
5  ENTITY fsm IS
6     GENERIC (
7        time1: NATURAL := 3;  -- # of clock periods in B
8        time2: NATURAL := 4); -- # of clock periods in C
9     PORT (
10       clk, rst: IN STD_LOGIC;
11       x: IN NATURAL RANGE 0 TO 3;
12       y: OUT STD_LOGIC);
13 END ENTITY;
14 ------------------------------------------------------------
15 ARCHITECTURE fsm OF fsm IS
16    TYPE state IS (A, B, C);
17    SIGNAL pr_state, nx_state: state;
18    ATTRIBUTE enum_encoding: STRING;
19    ATTRIBUTE enum_encoding OF state: TYPE IS "sequential";
20    SIGNAL T: NATURAL RANGE 0 TO time2;
21    SIGNAL input1, input2: NATURAL RANGE x'RANGE;
22 BEGIN
23    ----Lower section of FSM:---------
24    PROCESS (rst, clk)
25       VARIABLE count: NATURAL RANGE 0 TO time2;
26    BEGIN
27       IF (rst='1') THEN
28          pr_state <= A;
29       ELSIF (clk'EVENT AND clk='1') THEN
30          IF (x=input1 OR x=input2) THEN
31             count := count + 1;
32             IF (count=T) THEN
33                count := 0;
34                pr_state <= nx_state;
35             END IF;
36          ELSE
37             count := 0;
38          END IF;
39       END IF;
40    END PROCESS;
41    ----Upper section of FSM:---------
42    PROCESS (pr_state, x)
43    BEGIN
44       CASE pr_state IS
45          WHEN A =>
46             y <= '1';
47             T <= 1;
48             input1<=0; input2<=1;
49             nx_state <= B;
50          WHEN B =>
51             y <= '0';
52             T <= time1;
53             input1<=2; input2<=2;
54             nx_state <= C;
55          WHEN C =>
56             y <= '1';
57             T <= time2;
58             input1<=3; input2<=3;
59             nx_state <= A;
60       END CASE;
61    END PROCESS;
62 END ARCHITECTURE;
63 ------------------------------------------------------------
```

The view produced by the state machine viewer (with the *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.)

```
-----Package with function:-----------------------------------------------
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;
--------------------------------------------------------------------------------
```

```
------Main code:----------------------------------------------------------
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;
               IF (counter1=fclk/2) THEN
```

```
                  counter1 := 0;
                  counter2 := counter2 + 1;
               END IF;
            END IF;
         ELSE
            IF (counter2 /= 0) THEN
               counter1 := counter1 + 1;
               IF (counter1=fclk/2) THEN
                  counter1 := 0;
                  counter2 := counter2 - 1;
               END IF;
            END IF;
         END IF;
      END IF;
      counter_output <= counter2;
   END PROCESS;
   -----LCD driver [Pedroni 2008]:----------
   lcd_input <= integer_to_lcd(counter_output);
   LCD_ON <= '1'; BKL_ON <= '1';
   PROCESS (clk)
      ... (insert code for LCD driver seen in sec. 12.2)
   END PROCESS;
END ARCHITECTURE;
--------------------------------------------------------------------------------
```

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.

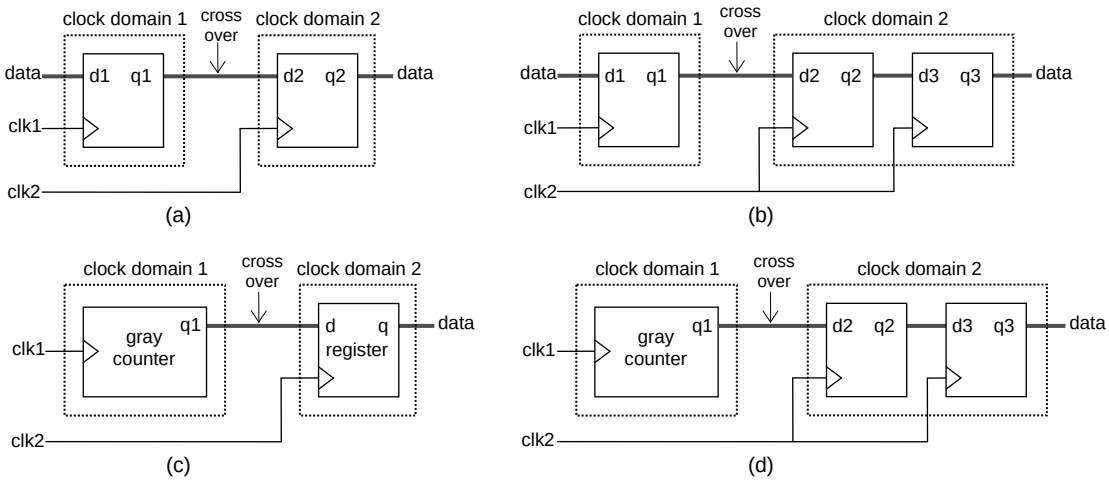


Figure (a) shows two interconnected domains. They operate with distinct (asynchronous with respect to each other) clocks, called *clk1* and *clk2*. The problem is that a positive edge in *clk2* can occur while the output of DFF1 (*q1*) is changing its value; in such a case, because the setup/hold times 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-

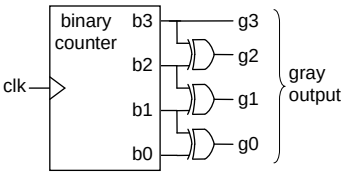
flops, but does require additional combinational logic, consisting of a layer with *N*-1 XOR gates. Obviously, the solution in figure (d) requires the sum of both.

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:

$g(N-1) = b(N-1)$

$g(i) = b(i) \oplus b(i+1)$ 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:

```
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 00 FF 1A 05 50 B0 00 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: I²C interface for an RTC

Note: Before examining this solution, please read the topic "Additional Details on the I²C 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.

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

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.

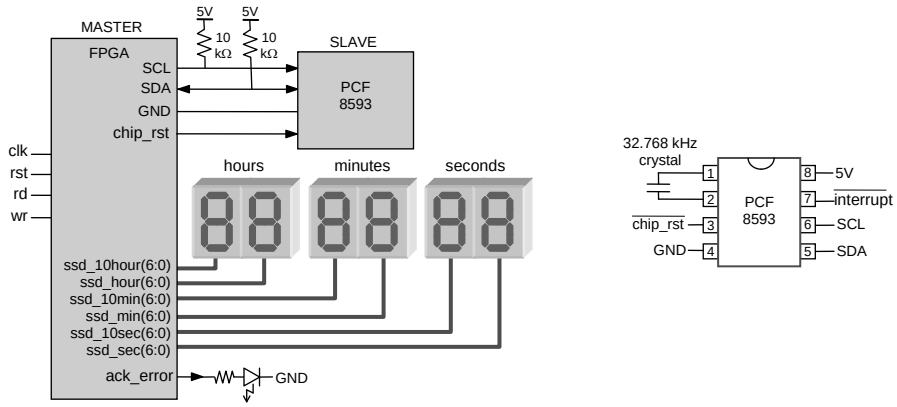


Figure 2. Setup for the experiments.

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 that in “Additional Details on the I^2C Interface” in the “Extra Material” part of the book website would also do.

```
----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
```

```
         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 bin_to_ssd;
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 (
      --Common write+read states:
      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,
```

```
58          --Read-only states:
59          initial_addr_rd, ack11, start_rd, slave_addr_rd, ack12, rd_sec, ack13, rd_min, ack14,
60          rd_hour, no_ack);
61       SIGNAL pr_state, nx_state: state;
62    BEGIN
63       chip_rst <= NOT rst; --to reset the chip (pin 3)
64       -------Display signals:--------------------
65       ssd_sec <= bin_to_ssd(sec(3 DOWNTO 0));
66       ssd_10sec <= bin_to_ssd(sec(7 DOWNTO 4));
67       ssd_min <= bin_to_ssd(min(3 DOWNTO 0));
68       ssd_10min <= bin_to_ssd(min(7 DOWNTO 4));
69       ssd_hour <= bin_to_ssd(hour(3 DOWNTO 0));
70       ssd_10hour <= bin_to_ssd("00" & hour(5 DOWNTO 4));
71       -------Auxiliary clock (400kHz):-----------
72       PROCESS (clk)
73          VARIABLE count1: INTEGER RANGE 0 TO divider;
74       BEGIN
75          IF (clk'EVENT AND clk='1') THEN
76             count1 := count1 + 1;
77             IF (count1=divider) THEN
78                aux_clk <= NOT aux_clk;
79                count1 := 0;
80             END IF;
81          END IF;
82       END PROCESS;
83       -------Bus & data clocks (100kHz):---------
84       PROCESS (aux_clk)
85          VARIABLE count2: NATURAL RANGE 0 TO 3;
86       BEGIN
87          IF (aux_clk'EVENT AND aux_clk='1') THEN
88             count2 := count2 + 1;
89             IF (count2=0) THEN
90                bus_clk <= '0';
91             ELSIF (count2=1) THEN
92                data_clk <= '1';
93             ELSIF (count2=2) THEN
94                bus_clk <= '1';
95             ELSE
96                data_clk <= '0';
97             END IF;
98          END IF;
99       END PROCESS;
100      -------Lower section of FSM:---------------
101      PROCESS (rst, data_clk)
102         VARIABLE error_flag: STD_LOGIC;
103      BEGIN
104         IF (rst='1') THEN
105            pr_state <= idle;
106            wr_flag <= '0';
107            rd_flag <= '0';
108            error_flag := '0';
109            i := 0;
110         --Enter data for I2C bus:
111         ELSIF (data_clk'EVENT AND data_clk='1') THEN
112            IF (i=timer-1) THEN
113               pr_state <= nx_state;
114               i := 0;
115            ELSE
116               i := i + 1;
117            END IF;
118         ELSIF (data_clk'EVENT AND data_clk='0') THEN
119            --Store data read from RTC memory:
120            IF (pr_state=rd_sec) THEN
121               sec(7-i) <= SDA;
122            ELSIF (pr_state=rd_min) THEN
123               min(7-i) <= SDA;
124            ELSIF (pr_state=rd_hour) THEN
125               hour(7-i) <= SDA;
126            END IF;
127            --Store write/read/error flags:
128            IF (pr_state=idle) THEN
129               wr_flag <= wr;
130               rd_flag <= rd;
131            ELSIF (pr_state=stop) THEN
132               wr_flag <= '0';
133               rd_flag <= '0';
134            ELSIF (pr_state=ack1 OR pr_state=ack2 OR pr_state=ack3 OR pr_state=ack4 OR
135               pr_state=ack5 OR pr_state=ack6OR pr_state=ack7 OR pr_state=ack8 OR pr_state=ack9
```

```
136             OR pr_state=ack10) THEN
137             error_flag := error_flag OR SDA;
138          END IF;
139       END IF;
140       ack_error <= error_flag;
141    END PROCESS;
142    -------Upper section of FSM:---------------
143    PROCESS (pr_state, bus_clk, data_clk, wr_flag, rd_flag)
144    BEGIN
145       CASE pr_state IS
146          --common write-read states:
147          WHEN idle =>
148             SCL <= '1';
149             SDA <= '1';
150             timer <= 1;
151             IF (wr_flag='1' OR rd_flag='1') THEN
152                nx_state <= start_wr;
153             ELSE
154                nx_state <= idle;
155             END IF;
156          WHEN start_wr =>
157             SCL <= '1';
158             SDA <= data_clk; --or '0'
159             timer <= 1;
160             nx_state <= slave_addr_wr;
161          WHEN slave_addr_wr =>
162             SCL <= bus_clk;
163             SDA <= slave_addr_for_write(7-i);
164             timer <= 8;
165             nx_state <= ack1;
166          WHEN ack1 =>
167             SCL <= bus_clk;
168             SDA <= 'Z';
169             timer <= 1;
170             IF (wr_flag='1') THEN
171                nx_state <= initial_addr_wr;
172             ELSE
173                nx_state <= initial_addr_rd;
174             END IF;
175          --data-write states:
176          WHEN initial_addr_wr =>
177             SCL <= bus_clk;
178             SDA <= initial_address_wr(7-i);
179             timer <= 8;
180             nx_state <= ack2;
181          WHEN ack2 =>
182             SCL <= bus_clk;
183             SDA <= 'Z';
184             timer <= 1;
185             nx_state <= wr_control;
186          WHEN wr_control =>
187             SCL <= bus_clk;
188             SDA <= set_control(7-i);
189             timer <= 8;
190             nx_state <= ack3;
191          WHEN ack3 =>
192             SCL <= bus_clk;
193             SDA <= 'Z';
194             timer <= 1;
195             nx_state <= wr_subsec;
196          WHEN wr_subsec =>
197             SCL <= bus_clk;
198             SDA <= set_subsec(7-i);
199             timer <= 8;
200             nx_state <= ack4;
201          WHEN ack4 =>
202             SCL <= bus_clk;
203             SDA <= 'Z';
204             timer <= 1;
205             nx_state <= wr_sec;
206          WHEN wr_sec =>
207             SCL <= bus_clk;
208             SDA <= set_sec(7-i);
209             timer <= 8;
210             nx_state <= ack5;
211          WHEN ack5 =>
212             SCL <= bus_clk;
213             SDA <= 'Z';
```

```
214             timer <= 1;
215             nx_state <= wr_min;
216          WHEN wr_min =>
217             SCL <= bus_clk;
218             SDA <= set_min(7-i);
219             timer <= 8;
220             nx_state <= ack6;
221          WHEN ack6 =>
222             SCL <= bus_clk;
223             SDA <= 'Z';
224             timer <= 1;
225             nx_state <= wr_hour;
226          WHEN wr_hour =>
227             SCL <= bus_clk;
228             SDA <= set_hour(7-i);
229             timer <= 8;
230             nx_state <= ack7;
231          WHEN ack7 =>
232             SCL <= bus_clk;
233             SDA <= 'Z';
234             timer <= 1;
235             nx_state <= wr_date;
236          WHEN wr_date =>
237             SCL <= bus_clk;
238             SDA <= set_date(7-i);
239             timer <= 8;
240             nx_state <= ack8;
241          WHEN ack8 =>
242             SCL <= bus_clk;
243             SDA <= 'Z';
244             timer <= 1;
245             nx_state <= wr_month;
246          WHEN wr_month =>
247             SCL <= bus_clk;
248             SDA <= set_month(7-i);
249             timer <= 8;
250             nx_state <= ack9;
251          WHEN ack9 =>
252             SCL <= bus_clk;
253             SDA <= 'Z';
254             timer <= 1;
255             nx_state <= wr_timer;
256          WHEN wr_timer =>
257             SCL <= bus_clk;
258             SDA <= set_timer(7-i);
259             timer <= 8;
260             nx_state <= ack10;
261          WHEN ack10 =>
262             SCL <= bus_clk;
263             SDA <= 'Z';
264             timer <= 1;
265             nx_state <= stop;
266          --data-read states:
267          WHEN initial_addr_rd =>
268             SCL <= bus_clk;
269             SDA <= initial_address_rd(7-i);
270             timer <= 8;
271             nx_state <= ack11;
272          WHEN ack11 =>
273             SCL <= bus_clk;
274             SDA <= 'Z';
275             timer <= 1;
276             nx_state <= start_rd;
277          WHEN start_rd =>
278             SCL <= '1'; --or bus_clk;
279             SDA <= data_clk;
280             timer <= 1;
281             nx_state <= slave_addr_rd;
282          WHEN slave_addr_rd =>
283             SCL <= bus_clk;
284             SDA <= slave_addr_for_read(7-i);
285             timer <= 8;
286             nx_state <= ack12;
287          WHEN ack12 =>
288             SCL <= bus_clk;
289             SDA <= 'Z';
290             timer <= 1;
291             nx_state <= rd_sec;
```

```
292           WHEN rd_sec =>
293              SCL <= bus_clk;
294              SDA <= 'Z';
295              timer <= 8;
296              nx_state <= ack13;
297           WHEN ack13 =>
298              SCL <= bus_clk;
299              SDA <= '0';
300              timer <= 1;
301              nx_state <= rd_min;
302           WHEN rd_min =>
303              SCL <= bus_clk;
304              SDA <= 'Z';
305              timer <= 8;
306              nx_state <= ack14;
307           WHEN ack14 =>
308              SCL <= bus_clk;
309              SDA <= '0';
310              timer <= 1;
311              nx_state <= rd_hour;
312           WHEN rd_hour =>
313              SCL <= bus_clk;
314              SDA <= 'Z';
315              timer <= 8;
316              nx_state <= no_ack;
317           WHEN no_ack =>
318              SCL <= bus_clk;
319              SDA <= '1';
320              timer <= 1;
321              nx_state <= stop;
322           --Common write-read state:
323           WHEN stop =>
324              SCL <= bus_clk;
325              SDA <= NOT data_clk; --or '0'
326              timer <= 1;
327              nx_state <= idle;
328        END CASE;
329     END PROCESS;
330  END ARCHITECTURE;
331  -------------------------------------------------------------------------
```

Exercise 15.1: 800x600x75Hz SVGA interface

All that is needed is to replace the values of H_{pulse}, H_{BP}, H_{active}, H_{FP}, V_{pulse}, V_{BP}, V_{active}, and $V_{FP}$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).

```
1     ------------------------------------------
2     --Part 2: IMAGE GENERATOR
3     ------------------------------------------
4     PROCESS (pixel_clk, Hsync, dena)
```

```
5        VARIABLE x: NATURAL RANGE 0 TO Hd;
6        VARIABLE y: NATURAL RANGE 0 TO Vd;
7     BEGIN
8        ----Count columns:--------
9        IF (pixel_clk'EVENT AND pixel_clk='1') THEN
10          IF (Hactive='1') THEN x := x + 1;
11          ELSE x := 0;
12          END IF;
13       END IF;
14       ----Count lines:----------
15       IF (Hsync'EVENT AND Hsync='1') THEN
16          IF (Vactive='1') THEN y := y + 1;
17          ELSE y := 0;
18          END IF;
19       END IF;
20       --Generate the image:-----
21       IF (dena='1') THEN
22          IF (((x-320)**2+(y-240)**2)<14400) THEN
23             R <= (OTHERS => '1');
24             G <= (OTHERS => '1');
25             B <= (OTHERS => '0');
26          ELSE
27             R <= (OTHERS => red_switch);
28             G <= (OTHERS => green_switch);
29             B <= (OTHERS => blue_switch);
30          END IF;
31       ELSE
32          R <= (OTHERS => '0');
33          G <= (OTHERS => '0');
34          B <= (OTHERS => '0');
35       END IF;
36    END PROCESS;
37    -----------------------------------------------
```

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).

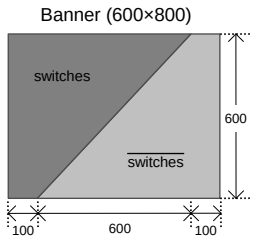
Exercise 16.4: Image generation with hardware #3 (filling with green)

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).

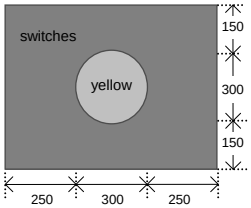
```
1  -----Image generator:--------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------------------
5  ENTITY image_generator IS
6     PORT (
7        red_switch, green_switch, blue_switch: IN STD_LOGIC;
8        clk50, Hsync, Vsync, Hactive, Vactive, dena: IN STD_LOGIC;
9        R, G, B: OUT STD_LOGIC_VECTOR(5 DOWNTO 0));
10 END image_generator;
11 ------------------------------------------------------------
12 ARCHITECTURE image_generator OF image_generator IS
13 BEGIN
14    PROCESS (clk50, dena)
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).

"Sun in the sky" (600×800)




```
-----Image generator:-----------------------------------------
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
```

```
         IF (((x-400)**2+(y-300)**2)<22500) THEN
            R <= (OTHERS => '1');
            G <= (OTHERS => '1');
            B <= (OTHERS => '0');
         ELSE
            R <= (OTHERS => red_switch);
            G <= (OTHERS => green_switch);
            B <= (OTHERS => blue_switch);
         END IF;
      ELSE
         R <= (OTHERS => '0');
         G <= (OTHERS => '0');
         B <= (OTHERS => '0');
      END IF;
   END PROCESS;
END image_generator;
-------------------------------------------------------------
```

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).

```
-----Image generator:-------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------------
ENTITY image_generator IS
   PORT (
      red_switch, green_switch, blue_switch: IN STD_LOGIC;
      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 (Hsync, Vsync, dena)
      VARIABLE y: INTEGER RANGE 0 TO 600;
      VARIABLE limit: INTEGER RANGE 0 TO 600;
      VARIABLE flag: STD_LOGIC;
   BEGIN
      --Count lines:-------------------
      IF (Hsync'EVENT AND Hsync='1') THEN
         IF (Vactive='1') THEN
            y := y + 1;
         ELSE
            y := 0;
         END IF;
      END IF;
      --Generate vert. limit:----------
      IF (Vsync'EVENT AND Vsync='1') THEN
         limit := limit + 1;
         IF (limit=600) THEN
            limit := 0;
            flag := NOT flag;
         END IF;
      END IF;
      --Generate the image:------------
      IF (dena='1') THEN
         IF (flag='0') THEN
            IF (y<=limit) THEN
               R <= (OTHERS => '0');
               G <= (OTHERS => '1');
               B <= (OTHERS => '0');
            ELSE
               R <= (OTHERS => red_switch);
               G <= (OTHERS => green_switch);
               B <= (OTHERS => blue_switch);
            END IF;
         ELSE
            IF (y>limit) THEN
               R <= (OTHERS => '0');
               G <= (OTHERS => '1');
```

```
51                B <= (OTHERS => '0');
52             ELSE
53                R <= (OTHERS => red_switch);
54                G <= (OTHERS => green_switch);
55                B <= (OTHERS => blue_switch);
56             END IF;
57          END IF;
58       ELSE
59          R <= (OTHERS => '0');
60          G <= (OTHERS => '0');
61          B <= (OTHERS => '0');
62       END IF;
63    END PROCESS;
64 END image_generator;
65 -------------------------------------------------------------
```