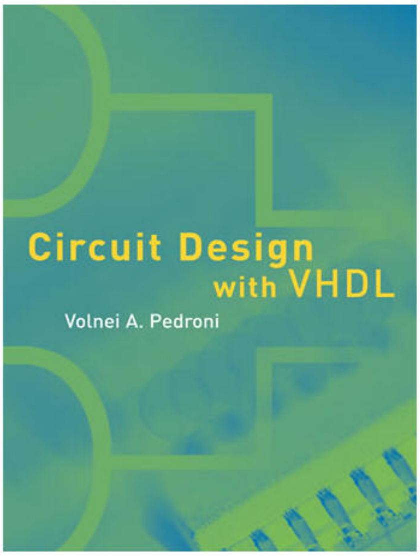


Circuit Design with VHDL

1st Edition

Volnei A. Pedroni, MIT Press, 2004

Selected Exercise Solutions

Problem 2.1: Multiplexer

```
---------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------------
ENTITY mux IS
   PORT (
      a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
      c: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END mux;
---------------------------------------------------
ARCHITECTURE example OF mux IS
BEGIN
   PROCESS (a, b, sel)
   BEGIN
      IF (sel="00") THEN
         c <= "00000000";
      ELSIF (sel="01") THEN
         c <= a;
      ELSIF (sel="10") THEN
         c <= b;
      ELSE
         c <= (OTHERS => 'Z'); --or c<="ZZZZZZZZ";
      END IF;
   END PROCESS;
END example;
---------------------------------------------------
```

Problem 3.2: Dealing with data types

First, recall figure 3.1, which shows four types of data structures. From it, we conclude the following:
a: a scalar of type BIT
b: a scalar of type STD_LOGIC
x: a 1D array (a vector) of type ARRAY1, whose 8 individual elements are of type STD_LOGIC
y: a 2D array (a matrix) of type ARRAY2, whose 4x8=32 individual elements are of type STD_LOGIC
w: a 1Dx1D array (another matrix) of type ARRAY3, whose 4 individual 8-element vectors are of type ARRAY1
z: another 1D array (another vector) whose 8 individual elements are again of type STD_LOGIC
Therefore:

`a <= x(2);`
a: scalar, type BIT
x(2): scalar, type STD_LOGIC
Assignment is illegal (type mismatch)

`b <= x(2);`
b: scalar, type STD_LOGIC
x(2): scalar, type STD_LOGIC
Assignment is legal

`b <= y(3,5);`
b: scalar, type STD_LOGIC
y(3,5): scalar, type STD_LOGIC, with valid indexing
Assignment is legal

`b <= w(5)(3);`
b: scalar, type STD_LOGIC
w(5)(3): scalar, type STD_LOGIC, but "5" is out of bounds
Assignment is illegal

`y(1)(0) <= z(7);`
y(1)(0): scalar, type STD_LOGIC, but indexing is incorrect because *y* is 2D (it should be *y*(1,0))
z(7): scalar, type STD_LOGIC
Assignment is illegal

`x(0) <= y(0,0);`
x(0): scalar, type STD_LOGIC
y(0,0): scalar, type STD_LOGIC, valid indexing
Assignment is legal

`x <= "1110000";`
x: 8-bit vector (1D)
Assignment would be legal if it contained 8 values instead of 7

`a <= "0000000";`
a: scalar, so can only have one bit
Assignment is illegal

`y(1) <= x;`
y(1): in principle, an 8-element vector, extracted from a 2D matrix, whose individual elements are of type STD_LOGIC; however, the indexing (slicing) of *y* is not valid, because the matrix is 2D, not 1Dx1D
x: an 8-element vector of type ARRAY1
Assignment is illegal (invalid slicing + type mismatch)

`w(0) <= y;`
w(0): row 0 of a 1Dx1D matrix, which is an 8-element vector of type ARRAY1
y: a 4x8 (2D) matrix
Assignment is illegal (size + type mismatches)

`w(1) <= (7=>'1', OTHERS=>'0');`
w(1): row 1 of a 1Dx1D matrix
Assignment is legal (*w*(1)<="10000000")

`y(1) <= (0=>'0', OTHERS=>'1');`
y(1): in principle, row 1 of a matrix, but the indexing is invalid, because the matrix is 2D, not 1Dx1D
Assignment *y*(1)<="11111110" is illegal

```
w(2)(7 DOWNTO 0) <= x;
```
w(2)(7 DOWNTO 0): row 2 of a 1Dx1D matrix, which is an 8-element vector of type ARRAY1
x: an 8-element vector of type ARRAY1
Assignment is legal
Note: w(2) <= x would be fine too

```
w(0)(7 DOWNTO 6) <= z(5 DOWNTO 4);
```
w(0)(7 DOWNTO 6): the leftmost 2 elements of row 0 of a 1Dx1D matrix, being each row an 8-element vector of type ARRAY1
z(5 DOWNTO 4): 2 elements of an 8-element STD_LOGIC_VECTOR
Assignment is illegal (type mismatch)

```
x(3) <= x(5 DOWNTO 5);
```
x(3): a scalar of type STD_LOGIC
x(5 DOWNTO 5): also a scalar of type STD_LOGIC
Assignment is legal

```
b <= x(5 DOWNTO 5)
```
b: a scalar of type STD_LOGIC
x(5 DOWNTO 5): also a scalar of type STD_LOGIC
Assignment is legal

```
y <= ((OTHERS=>'0'), (OTHERS=>'0'), (OTHERS=>'0'), "10000001");
```
y is a 2D matrix
Assignment is legal.
Note: Since *y* is 2D, some older compilers might not accept the vector-like assignments above, thus requiring the assignment to be made element by element (with GENERATE, for example).
Note: The assignment below is also legal.
```
y <= (('0','0','0','0','0','0','0','0'),
      ('0','0','0','0','0','0','0','0'),
      ('0','0','0','0','0','0','0','0'),
      ('1','0','0','0','0','0','0','1'));
```

```
z(6) <= x(5);
```
z(6): scalar of type STD_LOGIC
x(5): also a scalar of type STD_LOGIC (though as a vector *x* is of type ARRAY1, as a scalar ("base" type) it is STD_LOGIC)

```
z(6 DOWNTO 4) <= x(5 DOWNTO 3);
```
z(6 DOWNTO 4): 3-element vector of type STD_LOGIC_VECTOR
x(5 DOWNTO 3): 3-element vector of type ARRAY1
Assignment is illegal (type mismatch)

```
z(6 DOWNTO 4) <= y(5 DOWNTO 3);
```
The indexing of *y* is invalid (slicing 2D array is generally not allowed)
Assignment is illegal

```
y(6 DOWNTO 4) <= x(3 TO 5);
```
The indexing of *y* is invalid (slicing 2D array is generally not allowed)
Indexing of *x* is in the wrong direction
Assignment is illegal

```
y(0, 7 DOWNTO 0) <= z;
```
y(0, 7 DOWNTO 0): in principle, row 0 of a matrix, but slicing 2D arrays is generally not supported
Assignment is illegal

```
w(2,2) <= '1';
```
w is 1Dx1D, so indexing should be *w*(2)(2)
Assignment is illegal

Problem 4.1: Operators

```
x1 <= a & c;                     --x1 = "10010"
x2 <= c & b;                     --x2 = "00101100"
x3 <= b XOR c;                   --x3 = "1110"
x4 <= a NOR b(3);                --x4 = '0'
x5 <= b sll 2;                   --x5 = "0000"
x6 <= b sla 2;                   --x6 = "0000"
x7 <= b rol 2;                   --x7 = "0011";
x8 <= a AND NOT b(0) AND NOT c(1); --x8 = '0'
d <= (5=>'0', OTHERS=>'1');      --d = "11011111"
```

Problem 5.1: Generic Multiplexer

Solution 1: In this solution, no package is employed. Notice however that *x* was not defined as a 1Dx1D or 2D structure (chapter 3); instead, it was specified as simply a long vector of length $m(2^n)$. Though this will not affect the result, such a "linearization" might be confusing sometimes, so is not recommended in general.

```
----------------------------------------------
ENTITY generic_mux IS
   GENERIC (
      n: INTEGER := 4;  --number of selection bits
      m: INTEGER := 8); --number of bits per input
   PORT (
      x: IN BIT_VECTOR (m*2**n-1 DOWNTO 0);
      sel: IN INTEGER RANGE 0 TO 2**n-1;
      y: OUT BIT_VECTOR (m-1 DOWNTO 0));
END generic_mux;
----------------------------------------------
ARCHITECTURE generic_mux OF generic_mux IS
BEGIN
   gen: FOR i IN m-1 DOWNTO 0 GENERATE
      y(i) <= x(m*sel+i);
   END GENERATE gen;
END generic_mux;
----------------------------------------------
```

Solution 2: Here, a user-defined (in a package) type is employed

```
--- 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 (
      inputs: INTEGER := 16;     --number of inputs
      size: INTEGER := 8);       --size of each input
   PORT (
      x: IN MATRIX (0 TO inputs-1, size-1 DOWNTO 0);
      sel: IN INTEGER RANGE 0 TO inputs-1;
      y: OUT BIT_VECTOR (size-1 DOWNTO 0));
END generic_mux;
--------------------------------------------------------------
ARCHITECTURE arch OF generic_mux IS
BEGIN
   gen: FOR i IN size-1 DOWNTO 0 GENERATE
      y(i) <= x(sel, i);
   END GENERATE gen;
END arch;
--------------------------------------------------------------
```

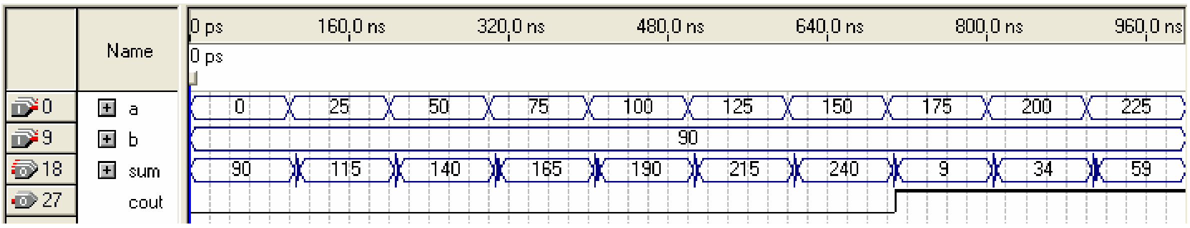
Problem 5.4: Unsigned adder

A possible solution is shown below (but see the NOTE that follows). The ports were considered to be of type STD_LOGIC (industry standard). Simulation results are included after the code.

```
-----------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all; --allows arith. operations w/ STD_LOGIC
-----------------------------------------------------------------------
ENTITY adder IS
   PORT (
      a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      sum: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
      cout: OUT STD_LOGIC);
END adder;
-----------------------------------------------------------------------
ARCHITECTURE adder OF adder IS
   SIGNAL long_a, long_b, long_sum: STD_LOGIC_VECTOR(8 DOWNTO 0);
BEGIN
   long_a <= '0' & a;
   long_b <= '0' & b;
   long_sum <= long_a + long_b;
   sum <= long_sum(7 DOWNTO 0);
   cout <= long_sum(8);
END adder;
-----------------------------------------------------------------------
```

NOTE: Even though the solution above is fine in principle, the <u>recommended</u> approach for arithmetic circuits is to explicitly convert the inputs to UNSIGNED or SIGNED (so it will be clear that the unsigned/signed issue was considered), do the computations, then return the result to STD_LOGIC_VECTOR (to be sent out).

The reader is invited to redo this problem taking into account the recommendation above.



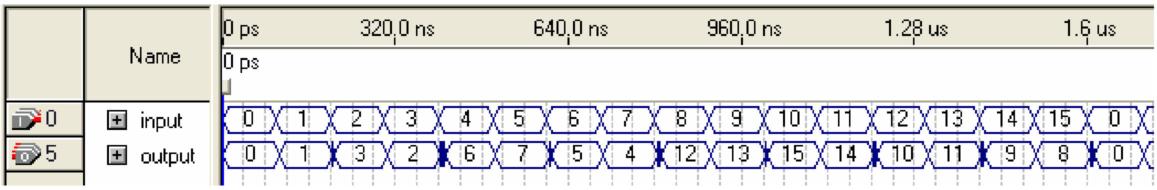
Simulation results for Problem 5.4.

Problem 5.6: Binary-to-Gray code converter

The gray codeword $g(N-1{:}0)$ corresponding to a regular binary codeword $b(N-1{:}0)$ can be obtained as follows:
For i = N−1: $g(i) = b(i)$
For i = N−2 ... 0: $g(i) = b(i) \oplus b(i+1)$
A corresponding *generic* VHDL code is presented below, followed by simulation results for N=4.

```
---------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------------------
ENTITY gray_encoder IS
   GENERIC (N: INTEGER := 4);
   PORT (b: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
         g: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END gray_encoder;
---------------------------------------------------------
ARCHITECTURE gray_encoder OF gray_encoder IS
BEGIN
   g(N-1) <= b(N-1);
   g(N-2 DOWNTO 0) <= b(N-2 DOWNTO 0) XOR b(N-1 DOWNTO 1);
END gray_encoder;
```

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



Simulation results for Problem 5.6.

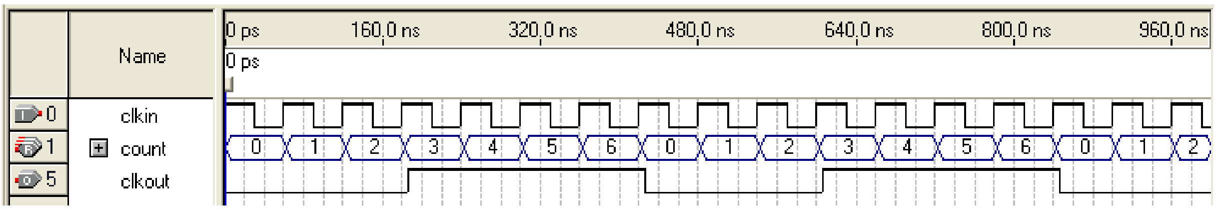
Problem 6.2: Four-stage shift register

```
--------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--------------------------------
ENTITY shift_reg IS
   PORT (
      clk, din: IN STD_LOGIC;
      dout: OUT STD_LOGIC);
END shift_reg;
--------------------------------
ARCHITECTURE shift_reg OF shift_reg IS
   SIGNAL d: STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
   PROCESS(clk)
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         d <= din & d(3 DOWNTO 1);
      END IF;
   END PROCESS;
   dout <= d(0);
END shift_reg;
--------------------------------
```

Problem 6.4: Generic frequency divider

```
----Clock frequency is divided by N-----------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
----------------------------------------------
ENTITY clock_divider IS
   GENERIC (N: POSITIVE := 7);
   PORT (clkin: IN STD_LOGIC;
         clkout: OUT STD_LOGIC);
END ENTITY;
----------------------------------------------
ARCHITECTURE clock_divider OF clock_divider IS
BEGIN
   PROCESS (clkin)
      VARIABLE count: INTEGER RANGE 0 TO N;
   BEGIN
      IF (clkin'EVENT AND clkin='1') THEN
         count := count + 1;
         IF (count=N/2) THEN
            clkout <= '1';
         ELSIF (count=N) THEN
            clkout <= '0';
            count := 0;
         END IF;
      END IF;
   END PROCESS;
END ARCHITECTURE;
---------------------------------------------
```

NOTE: In the solution above the duty cycle is not symmetric when *N* is odd (see simulation results below). Can you rewrite this code such that the duty cycle is always 50% (for both even and odd values of *N*)?



Simulation results for *N*=7 in Problem 6.4.

Problem 6.12: Vector shifter

```
---------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY vector_shifter IS
   GENERIC (n: INTEGER := 7); -- number of bits at input
   PORT (
      input: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
      sel: IN INTEGER RANGE 0 TO n;
      output: OUT STD_LOGIC_VECTOR(2*n-1 DOWNTO 0));
END vector_shifter;
---------------------------------------
ARCHITECTURE behavior OF vector_shifter IS
BEGIN
   PROCESS (input, sel)
      VARIABLE temp: STD_LOGIC_VECTOR(output'LEFT DOWNTO 0);
   BEGIN
      temp := (OTHERS => '0');
      FOR i IN input'RANGE LOOP
         temp(sel+i) := input(i);
      END LOOP;
      output <= temp;
   END PROCESS;
END behavior;
---------------------------------------
```

Problem 6.17: DFF

Architecture 1: A truly D-type flip-flop with asynchronous reset (same as in Example 6.1) .

Architecture 2: Only the clock appears in the sensitivity list, causing the reset to be *synchronous*.

Architecture 3: clk'EVENT is not associated with a test (AND clk='1', for example). Such test is mandatory in most compilers, so this is nor a correct code.

Architecture 4: Here the contents of sections 6.10 and 7.5 are helpful. Notice that no signal assignment is made at the transition of another signal, signifying that, in principle, no flip-flop is wanted (thus a combinational circuit). However, only one of the input signals appears in the sensitivity list, and an incomplete truth table for *q* is specified in the code, causing the inference of a latch to hold the value of *q* (a "pseudo" combinational circuit).

Architecture 5: The situation here is even more awkward than that above. Besides inferring again a latch, changes of *d* also cause the process to be run. The resulting circuit is very unlikely to be of any interest.

Problem 7.1: Legal/illegal assignments

Suggestion: Before solving this problem, review the solution of Problem 3.2.

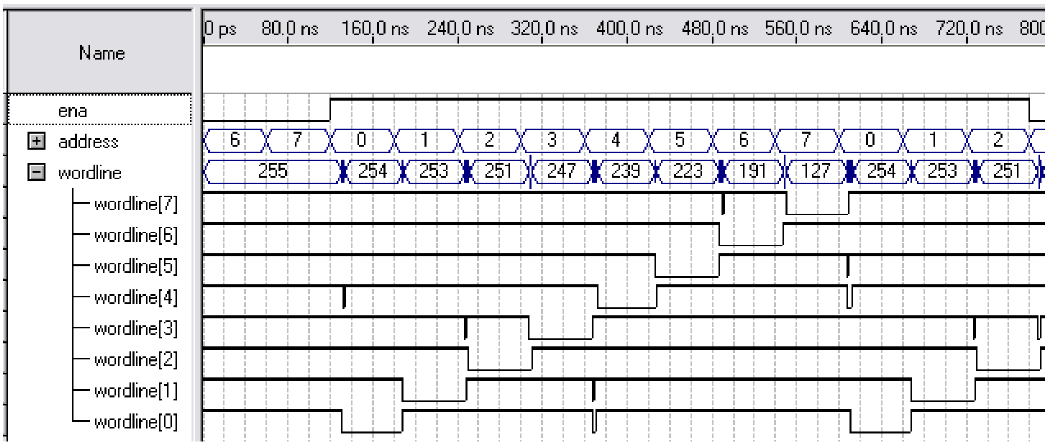
```
x <= 5;                             --legal
x <= y(5);                          --illegal (type mismatch)
z <= '1';                           --illegal (should be ":=" for variable)
z := y(5);                          --legal
```

```
WHILE i IN 0 TO max LOOP...         --legal (both bounds are static)
FOR i IN 0 TO x LOOP...             --illegal (bounds must be static)
G1: FOR i IN 0 TO max GENERATE...   --legal (both bounds are static)
G1: FOR i IN 0 TO x GENERATE...     --illegal (bounds must be static)
```

Problem 7.6: Generic address decoder

In the solution below, all ports are of type STD_LOGIC (industry standard). Simulation results (for *N*=3) are included after the code. Regarding the glitches in the simulation results, recall that they are normal in combinational circuits when a signal depends on multiple bits, because the values of such bits do not change all at exactly the same time.

```
----------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
----------------------------------------------------------
ENTITY address_decoder IS
   GENERIC (N: POSITIVE := 3); --# of input bits
   PORT (ena: IN STD_LOGIC;
         address: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
         wordline: OUT STD_LOGIC_VECTOR(2**N-1 DOWNTO 0));
END ENTITY;
----------------------------------------------------------
ARCHITECTURE address_decoder OF address_decoder IS
BEGIN
   PROCESS (ena, address)
      VARIABLE internal: STD_LOGIC_VECTOR(2**N-1 DOWNTO 0);
      VARIABLE addr: NATURAL RANGE 0 TO 2**N-1;
   BEGIN
      addr := CONV_INTEGER(address);
      internal := (OTHERS => '1');
      IF (ena='1') THEN
         internal(addr) := '0';
      END IF;
      wordline <= internal;
   END PROCESS;
END ARCHITECTURE;
---------------------------------------------------------
```



Simulation results for Problem 7.6.

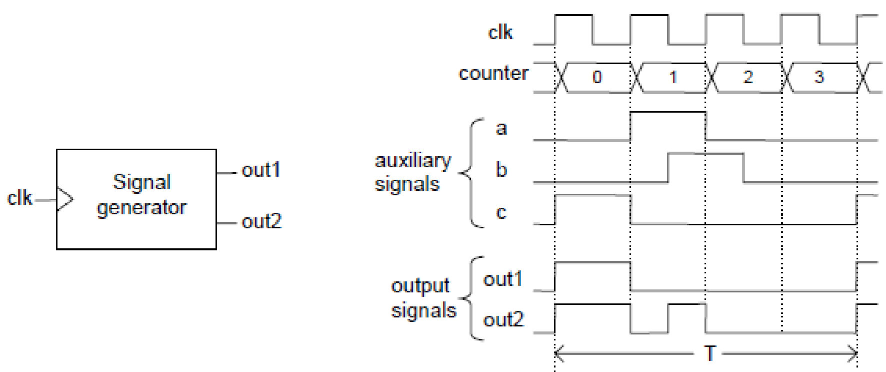
Problem 8.1: Finite State Machine

```
---------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY fsm IS
   PORT (
      inp, clk, rst: IN STD_LOGIC;
      outp: OUT STD_LOGIC_VECTOR (1 DOWNTO 0));
END fsm;
```

```
----------------------------------------------
ARCHITECTURE fsm OF fsm IS
   TYPE state IS (state1, state2, state3, state4);
   SIGNAL pr_state, nx_state: state;
BEGIN
   ----- Lower section: ----------------------
   PROCESS (rst, clk)
   BEGIN
      IF (rst='1') THEN
         pr_state <= state1;
      ELSIF (clk'EVENT AND clk='1') THEN
         pr_state <= nx_state;
      END IF;
   END PROCESS;
   ---------- Upper section: -----------------
   PROCESS (inp, pr_state)
   BEGIN
      CASE pr_state IS
         WHEN state1 =>
            outp <= "00";
            IF (inp='1') THEN
               nx_state <= state2;
            ELSE
               nx_state <= state1;
            END IF;
         WHEN state2 =>
            outp <= "01";
            IF (inp='1') THEN
               nx_state <= state4;
            ELSE
               nx_state <= state3;
            END IF;
         WHEN state3 =>
            outp <= "10";
            IF (inp='1') THEN
               nx_state <= state4;
            ELSE
               nx_state <= state3;
            END IF;
         WHEN state4 =>
            outp <= "11";
            IF (inp='1') THEN
               nx_state <= state1;
            ELSE
               nx_state <= state2;
            END IF;
      END CASE;
   END PROCESS;
END fsm;
----------------------------------------------
```

Problem 8.6: Signal generator without the FSM approach

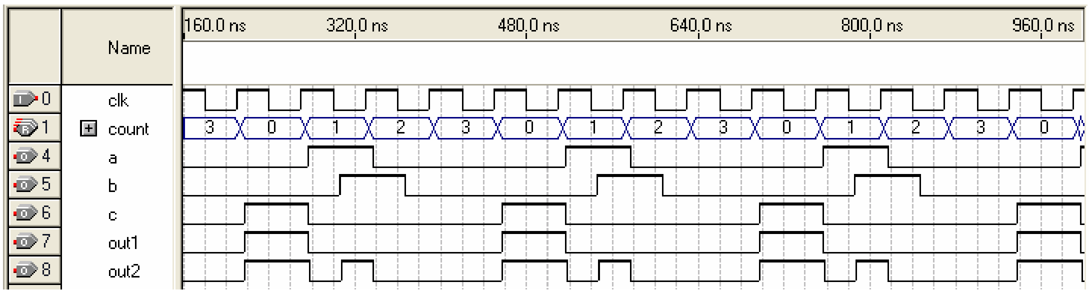
Physical circuits, design, and operation of sequential circuits are described in chapter 14 of [1]. The solution that follows is based on the theory presented there. The corresponding signals are depicted in the figure below.



Signals employed in the solution of Problem 8.6.

In this case (figure above), three auxiliary signals (*a*, *b*, *c*) are created, from which *out1* and *out2* are then derived using conventional gates. As described in chapter 14 of [1], the fundamental point here is to guarantee that the outputs are not prone to glitches. To guarantee glitch-free outputs, all three auxiliary signals are registered (i.e., stored in DFFs); consequently, given that no two signals that affect the gates' outputs change at the same clock edge, glitch-free outputs are automatically generated. A corresponding VHDL follows, along with simulation results.

```
----------------------------------------------------
ENTITY signal_generator IS
   PORT (clk: IN BIT;
         out1, out2: OUT BIT);
END ENTITY;
----------------------------------------------------
ARCHITECTURE signal_generator OF signal_generator IS
   SIGNAL a, b, c: BIT;
   SIGNAL counter: INTEGER RANGE 0 TO 3;
BEGIN
   ----Creating a counter:------------
   PROCESS (clk)
      VARIABLE count: INTEGER RANGE 0 TO 4;
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         count := count + 1;
         IF (count=4) THEN
            count := 0;
         END IF;
      END IF;
      counter <= count;
   END PROCESS;
   ----Generating signal a:-----------
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         IF (counter=0) THEN
            a <= '1';
         ELSE
            a <= '0';
         END IF;
      END IF;
   END PROCESS;
   ----Generating signal b:-----------
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT AND clk='0') THEN
         IF (counter=1) THEN
            b <= '1';
         ELSE
            b <= '0';
         END IF;
      END IF;
   END PROCESS;
   ----Generating signal c:-----------
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         IF (counter=3) THEN
            c <= '1';
         ELSE
            c <= '0';
         END IF;
      END IF;
   END PROCESS;
   ----Generating the outputs:--------
   out1 <= c;
   out2 <= (a AND b) OR c;
END ARCHITECTURE;
----------------------------------------------------
```



Simulation results for Problem 8.6.

Problem 10.2: Carry-ripple adder constructed with components

```
-----The component (full-adder unit):-----------
ENTITY FAU IS
   PORT (a, b, cin: IN BIT;
         s, cout: OUT BIT);
END FAU;
------------------------------------------------
ARCHITECTURE full_adder OF FAU IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin) OR (b AND cin);
END full_adder;
------------------------------------------------

-----Main code:------------------------------------------
ENTITY carry_ripple_adder IS
   GENERIC (N : INTEGER := 8); --number of bits
   PORT (a, b: IN BIT_VECTOR(N-1 DOWNTO 0);
         cin: IN BIT;
         s: OUT BIT_VECTOR(N-1 DOWNTO 0);
         cout: OUT BIT);
END carry_ripple_adder;
-----------------------------------------------------------
ARCHITECTURE structural OF carry_ripple_adder IS
   SIGNAL carry: BIT_VECTOR(N DOWNTO 0);
   COMPONENT FAU IS
      PORT (a, b, cin: IN BIT; s, cout: OUT BIT);
   END COMPONENT;
BEGIN
   carry(0) <= cin;
   generate_adder: FOR i IN a'RANGE GENERATE
      adder: FAU PORT MAP (a(i), b(i), carry(i), s(i), carry(i+1));
   END GENERATE;
   cout <= carry(N);
END structural;
-----------------------------------------------------------
```

Problem 11.1: Conversion from INTEGER to STD_LOGIC_VECTOR

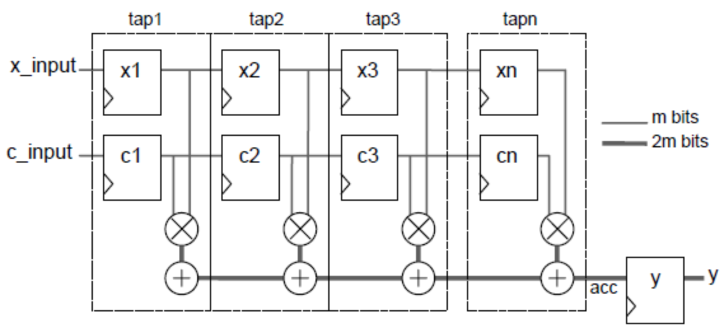
A VHDL code for this exercise is shown below. To test it, just a jumper from the input to the output was used, in which case the compiler must give equations of the type *output(i)=input(i)*, for *i*=0, 1, …, *N*-1. A similar function can be found in the standard package *std_logic_arith*, available in the library of your VHDL synthesis software. Note that in the solution below the function was located in the main code; to install it in a package, just follow the instruction in chapter 11 (see example 11.4, for example).

```
-------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------
ENTITY data_converter IS
   GENERIC (N: NATURAL := 4); --number of bits
   PORT (input: IN INTEGER RANGE 0 TO 2**N-1;
         output: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END data_converter;
```

```
-------------------------------------------------------
ARCHITECTURE data_converter OF data_converter IS
   SIGNAL x: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
   -------------------
   FUNCTION conv_std_logic(arg: INTEGER; size: POSITIVE)
   RETURN STD_LOGIC_VECTOR IS
      VARIABLE temp: INTEGER RANGE 0 TO 2**size-1;
      VARIABLE result: STD_LOGIC_VECTOR(size-1 DOWNTO 0);
   BEGIN
      temp := arg;
      FOR i IN result'RANGE LOOP
         IF (temp>=2**i) THEN
            result(i) := '1';
            temp := temp - 2**i;
         ELSE
            result(i) := '0';
         END IF;
      END LOOP;
      RETURN result;
   END conv_std_logic;
   -------------------
BEGIN
   output <= conv_std_logic(input, N);
END data_converter;
-------------------------------------------------------
```

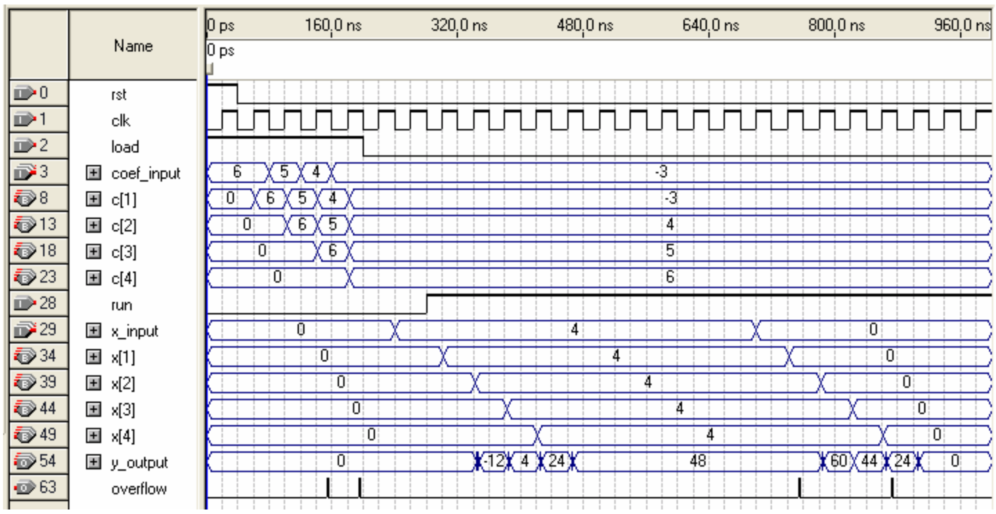
Problem 12.4: General-purpose FIR filter

In this example, only integers are employed. A block diagram for the circuit is shown below. It contains two shift registers, which store the input values (*x*) and the filter coefficients (*c*). It contains also a register to store the accumulated (*acc*) value, producing the filter output (*y*). The total number of taps is *n*, with *m* bits used to represent *x* and *c*, and 2*m* bits for the after- multiplication paths. Hence a total of 2*m*(n+1) flip-flops are required. Observe that the taps are all alike, so COMPONENT can be used to easily implement this circuit. A slightly more difficult (non-structural) solution is shown below; the structural option (with COMPONENT) is left to the reader. Note that the code includes overflow check. Simulation results are also included.




```
-----------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all; --package needed for SIGNED
-----------------------------------------------------------------
ENTITY FIR IS
   GENERIC (n: INTEGER := 4;  --number of coefficients
            m: INTEGER := 4); --number of bits per coefficient
   PORT (clk, rst: IN STD_LOGIC;
         load: STD_LOGIC;  --to enter new coefficient values
         run: STD_LOGIC;   --to compute the output
         x_input, coef_input: IN SIGNED(m-1 DOWNTO 0);
         y: OUT SIGNED(2*m-1 DOWNTO 0);
         overflow: OUT STD_LOGIC);
END FIR;
-----------------------------------------------------------------
ARCHITECTURE FIR OF FIR IS
   TYPE internal_array IS ARRAY (1 TO n) OF SIGNED(m-1 DOWNTO 0);
   SIGNAL c: internal_array;  --stored coefficients
   SIGNAL x: internal_array;  --stored input values
BEGIN
```

```
   PROCESS (clk, rst)
      VARIABLE prod, acc: SIGNED(2*m-1 DOWNTO 0) := (OTHERS=>'0');
      VARIABLE sign_prod, sign_acc: STD_LOGIC;
   BEGIN
      --Reset:--------------------------------
      IF (rst='1') THEN
         FOR i IN 1 TO n LOOP
            FOR j IN m-1 DOWNTO 0 LOOP
               x(i)(j) <= '0';
            END LOOP;
         END LOOP;
      --Shift registers:----------------------
      ELSIF (clk'EVENT AND clk='1') THEN
         IF (load='1') THEN
            c <= (coef_input & c(1 TO n-1));
         ELSIF (run='1') THEN
            x <= (x_input & x(1 TO n-1));
         END IF;
      END IF;
      --MACs and output (w/ overflow check):---
      acc := (OTHERS=>'0');
      FOR i IN 1 TO n LOOP
         prod := x(i)*c(i);
         sign_prod := prod(2*m-1);
         sign_acc := acc(2*m-1);
         acc := prod + acc;
         IF (sign_prod=sign_acc AND acc(2*m-1)/=sign_acc) THEN
            overflow <= '1';
         ELSE
            overflow <= '0';
         END IF;
      END LOOP;
      IF (clk'EVENT AND clk='1') THEN
         y <= acc;
      END IF;
   END PROCESS;
END FIR;
-----------------------------------------------------------------
```



Simulation results for Problem 12.4.