# vhdl-verilog-examples

# by


# Miljko Bobrek, Ph.D.

# (Adapted by Don Bouldin on 29 October 2008)


*Non-synthesizable Code*

Non-synthesizable code used for simulation needs to reside in simulation-only files or test benches. Typical examples of non-synthesizable codes include wait, after, and delay statements as shown below

```
--VHDL

wait for 10 ns;
Q <= '0' after 20 ns;
```

```
//Verilog

# 10;
assign #20 Q=0;
```

*Implementing Resets*

Resets need to be synchronous unless the reset signal comes from a different clock region or is an external signal. In that case, the asynchronous reset signal needs to be resynchronized to avoid metastability when the reset is released.

```
--VHDL synchronous reset

process (CLK)
begin
   if CLK'event and CLK = '1' then
      if RST = '1' then
         Q <= '0';
      else
         Q <= D;
      end if;
   end if;
end process;
```

```
//Verilog synchronous reset

always @ (posedge CLK)
begin
   if  (RST)
      Q = 1'b0;
   else
      Q = D;
end
```

```
--VHDL asynchronous reset

process (CLK,RST)
begin
   if RST = '1' then
      Q <= '0';
   elsif CLK'event and CLK = '1' then
      Q <= D;
   end if;
end process;
```

```
//Verilog asynchronous reset

always @ (posedge CLK or posedge RST)
begin
   if  (RST)
      Q = 1'b0;
   else
      Q = D;
end
```

```
--VHDL reset synchronizer1

process (CLK)
begin
   if CLK'event and CLK = '1' then
      RST1 <= RST_EXT;
      RST <= RST1;
   end if;
end process;
```

```
//Verilog reset synchronizer1

always @ (posedge CLK)
begin
   RST = RST1;
   RST1 = RST_EXT;
end
```

```
--VHDL reset synchronizer2

process (CLK,RST_EXT)
begin
   if RST_EXT = '1' then
      RST1 <= '1';
      RST<= '1';
   elsif CLK'event and CLK = '1' then
      RST1 <= '0';
      RST <= RST1;
   end if;
end process;
```

```
//Verilog reset synchronizer2

always @ (posedge CLK or posedge RST_EXT)
begin
   if (RST_EXT) begin
      RST = 1'b1;
      RST1 = 1'b1;
      end
   else begin
      RST = RST1;
      RST1 = 0;
      end
end
```

*Multiplexers/Encoders*

Below are the examples of **if**- and **case**-based multiplexers and encoders. If priority is not needed, use **case** statements instead of **if** statements to minimize the logic created by synthesis tools

```
--VHDL 4-to-1 priority mux

process (RST,SEL,IN1,IN2,IN3,IN4)
begin
   if RST = '1' then O <= '0';
   elsif SEL = "00" then O <= IN1;
   elsif SEL = "01" then O <= IN2;
   elsif SEL = "10" then O <= IN3;
   else O <= IN4;
   end if;
end process;
```

```
//Verilog 4-to-1 priority mux

always @ (RST or SEL or IN1 or IN2 or IN3 or IN4)
   begin
      if (RST == 1'b1) O = 1'b0;
      else if (SEL == 2'b00) O = IN1;
      else if (SEL == 2'b01) O = IN2;
      else if (SEL == 2'b10) O = IN3;
      else O = IN4;
   end
```

```
--VHDL 4-to-1 mux using case statement

process (SEL,IN1,IN2,IN3,IN4)
begin
   case SEL is
      when "00" => O <= IN1;
      when "01" => O <= IN2;
      when "10" => O <= IN3;
      when others => O <= IN4;
   end case;
end process;
```

```
//Verilog 4-to-1 mux using case statement

always @ (SEL or IN1 or IN2 or IN3 or IN4)
   begin
      case (SEL)
         2'b00 : O = IN1;
         2'b01 : O = IN2;
         2'b10 : O = IN3;
         2'b11 : O = IN4;
      endcase
   end
```

```
--VHDL 4-to-2 priority encoder

process (RST,IN1)
begin
   if RST = '1' then O <= "00";
   elsif IN1 = "0000" then O <= "00";
   elsif IN1 = "0010" then O <= "01";
   elsif IN1 = "0100" then O <= "10";
   elsif IN1 = "1000" then O <= "11";
   else O <= "00";
   end if;
```

```
//Verilog 4-to-2 priority encoder

always @ (RST or IN1)
   begin
      if (RST == 1'b1) O = 2'b00;
      else if (IN1 == 4'b0001) O = 2'b00;
      else if (IN1 == 4'b0010) O = 2'b01;
      else if (IN1 == 4'b0100) O = 2'b10;
      else if (IN1 == 4'b1000) O = 2'b11;
      else O = 2'b00;
   end
```

```
--VHDL 4-to-2 encoder using case statement

process (IN1)
begin
   case IN1 is
      when "0001" => O <= "00";
      when "0010" => O <= "01";
      when "0100" => O <= "10";
      when "1000" => O <= "11";
      when others  => O <= "00";
   end case;
end process;
```

```
//Verilog 4-to-2 encoder using case statement

always @ (IN1)
   begin
      case (IN1)
         4'b0001 : O = 2'b00;
         4'b0010 : O = 2'b01;
         4'b0100 : O = 2'b10;
         4'b1000 : O = 2'b11;
         default   : O = 2'b00;
      endcase
   end
```

*De-multiplexers/Decoders*

The **case** statements need to be used if priority is not required. Below are the examples of **if**- and **case**-based de-multiplexers and decoders.

```
--VHDL 1-to-4 priority demux

process (RST,SEL,IN1)
begin
   if RST = '1' then O <= "0000";
   elsif SEL = "00" then O <= "000"&IN1;
   elsif SEL = "01" then O <= "00"&IN1&'0';
   elsif SEL = "10" then O <= '0'&IN1&"00";
   else O <= IN1&"000";
   end if;
end process;
```

```
//Verilog 1-to-4 priority demux

always @ (RST or SEL or IN1)
   begin
      if (RST == 1'b1) O = 4'b0000;
      else if (SEL == 2'b00) O = {3'b000,IN1};
      else if (SEL == 2'b01) O = {2'b00,IN1,1'b0};
      else if (SEL == 2'b10) O = {1'b0,IN1,2'b00};
      else O = {IN1,3'b000};
   end
```

```
--VHDL1-to-4 demux using case statement

process (SEL,IN1)
begin
   case SEL is
      when "00" => O <= "000"&IN1;
      when "01" => O <= "00"&IN1&'0';
      when "10" => O <= '0'&IN1&"00";
      when others => O <= IN1&"000";
   end case;
end process;
```

```
//Verilog 1-to-4 demux using case statement

always @ (SEL or IN1)
   begin
      case (SEL)
         2'b00 : O = {3'b000,IN1};
         2'b01 : O = {2'b00,IN1,1'b0};
         2'b10 : O = {1'b0,IN1,2'b00};
         2'b11 : O = {IN1,3'b000};
      endcase
   end
```

```
--VHDL 2-to-4 priority decoder

process (RST,IN1)
begin
   if RST = '1' then O <= "0000";
   elsif IN1 = "00" then O <= "0111";
   elsif IN1 = "01" then O <= "0110";
   elsif IN1 = "10" then O <= "1100";
   else O <= "1110";
   end if;
end process;
```

```
//Verilog 2-to-4 priority decoder

always @ (RST or IN1)
   begin
      if (RST == 1'b1) O = 4'b0000;
      else if (IN1 == 2'b00) O = 4'b0111;
      else if (IN1 == 2'b01) O = 4'b0110;
      else if (IN1 == 2'b10) O = 4'b1100;
      else O = 4'b1110;
   end
```

```
--VHDL 2-to-4 decoder using case statement

process (IN1)
begin
   case IN1 is
      when "00" => O <= "0111";
      when "01" => O <= "0110";
      when "10" => O <= "1100";
      when others => O <= "1110";
   end case;
end process;
```

```
//Verilog 4-to-2 encoder using case statement

always @ (IN1)
   begin
      case (IN1)
         2'b00 : O = 4'b0111;
         2'b01 : O = 4'b0110;
         2'b10 : O = 4'b1100;
         2'b11 : O = 4'b1110;
      endcase
   end
```

*Comparators*

Before using the greater than (>) or less then (<) statement, designer needs to be aware of the format of the numbers being compared. As the example below shows, the result of the comparison depends whether the numbers are signed or unsigned.

```
--VHDL compare statement

O <= '1' when A > X"00" else '0';

-- if A = X"F0" is signed, O equals '0'
-- if A = X"F0" is unsigned, O equals '1'
```

```
//Verilog compare statement

assign O = (A > 8'h00) ? 1'b1 : 1'b0;

// if A = 8'hF0 is signed, O equals 1'b0
// if A = 8'hF0 is unsigned, O equals 1'b1
```

*Adder Trees*

Addition of multiple numbers needs to be implemented as synchronous cascade of two-number adders to avoid different synthesis implementations and to improve timing performance. Also, care needs to be taken to properly size the data-path to avoid overflows in the addition tree. Below are shown examples of 5-number adder trees.

```
--VHDL pipelined adder tree with 5 inputs and 3-cycle latency

process (CLK)
begin
   if CLK'event and CLK = '1' then
      LEVEL_ONE_SUM1 <= IN1 + IN2;
      LEVEL_ONE_SUM2 <= IN3 + IN4;
      LEVEL_ONE_SUM3 <= IN5;
      LEVEL_TWO_SUM1 <= LEVEL_ONE_SUM1 + LEVEL_ONE_SUM2;
      LEVEL_TWO_SUM2 <= LEVEL_ONE_SUM3;
      FINAL_SUM <= LEVEL_TWO_SUM1 + LEVEL_TWO_SUM2;
   end if;
end process;
```

```
//Verilog pipelined adder tree with 5 inputs and 3-cycle latency

always @ (posedge CLK)
begin
   FINAL_SUM = LEVEL_TWO_SUM1 + LEVEL_TWO_SUM2;
   LEVEL_TWO_SUM1 = LEVEL_ONE_SUM1 + LEVEL_ONE_SUM2;
   LEVEL_TWO_SUM2 = LEVEL_ONE_SUM3;
   LEVEL_ONE_SUM1 = IN1 + IN2;
   LEVEL_ONE_SUM3 = IN5;
   LEVEL_ONE_SUM2 = IN3 + IN4;
end
```

*Arithmetic Overflow*

To avoid overflow during addition or subtraction, the most significant bit (MSB) padding needs to be applied to the operands before the operation is executed. The examples

below show the MSB padding together with the bit-trimming that is often needed to keep the data-path from unnecessary growth.

```
--VHDL MSB padding when adding two 8-bit numbers
-- otputs are trimmed to 8-bit numbers

process (CLK)
begin
   if CLK'event and CLK = '1' then
      FULL_SIZE_SUM <= (IN1(7)&IN1) + (IN2(7)&IN2);
      FULL_SIZE_DIFF <= (IN3(7)&IN3) - (IN4(7)&IN4);
   end if;
end process;
FINAL_SUM <= FULL_SIZE_SUM(8 downto 1);
FINAL_DIFF <= FULL_SIZE_DIFF(8 downto 1);
```

```
//Verilog MSB padding when adding two 8-bit numbers
// otputs are trimmed to 8-bit numbers

always @ (posedge CLK)
begin
   FULL_SIZE_SUM = {IN1[7],IN1} + {IN2[7],IN2};
   FULL_SIZE_DIFF = {IN3[7],IN3} - {IN4[7],IN4};
end
assign FINAL_SUM = FULL_SIZE_SUM[8:1];
assign FINAL_DIFF = FULL_SIZE_DIFF[8:1];
```

*Binary Scaling*

Binary scaling is prone to errors due to potential mishandling of the 2's complement numbers. A proper scaling requires that the most significant bit is preserved during the binary up- and down-scaling as shown below.

```
--VHDL scaling of a 16-bit number

A_DIV_BY4 <= A(15)&A(15)&A(15 downto 2);
A_MULT_BY2 <= A(15)&A(13 downto 0)&'0';
```

```
//Verilog scaling of a 16-bit number

assign A_DIV_BY4 = {A[15],A[15],A[15:2]};
assign A_MULT_BY2 = {A[15],A[13:0],1'b0};
```

*Counters*

The examples below show the binary counter implementation.

```
--VHDL 8-bit counter

process (CLK)
begin
   if CLK'event and CLK = '1' then
      if RST = '1' then
         CNT <= X"00";
      else
         CNT <= CNT + '1';
      end if;
   end if;
end process;
```

```
//Verilog 8-bit counter

always @ (posedge CLK)
begin
   if  (RST)
      CNT = 8'h00;
   else
      CNT = CNT + 1;
end
```

*Shift Registers*

Examples below show implementations of the parallel-to-serial shift registers, and the serial-to-parallel shift registers.

```
--VHDL P-to-S 8-bit shift register, MSB first

process (CLK)
begin
   if CLK'event and CLK = '1' then
      if RST = '1' then
         SHIFT <= X"00";
         SHIFT_CNT <= "000";
      elsif LOAD = '1' then
         SHIFT <= PDATA;
         SHIFT_CNT <= "000";
      elsif SHIFT_CNT > "000" then
         SHIFT(7 downto 1) <= SHIFT(6 downto 0);
         SHIFT_CNT <= SHIFT_CNT + '1';
      end if;
   end if;
end process;
SDATA <= SHIFT(7);
```

```
//Verilog P-to-S shift register, MSB first

always @ (posedge CLK)
begin
   if  (RST) begin
      SHIFT = 8'h00;
      SHIFT_CNT = 3'b000;
      end
   else if (LOAD == 1'b1) begin
      SHIFT = PDATA;
      SHIFT_CNT = 3'b000;
      end
   else if (SHIFT_CNT > 3'b000) begin
      SHIFT[7:1] = SHIFT[6:0];
      SHIFT_CNT = SHIFT_CNT + 1;
      end
end
assign SDATA = SHIFT[7];
```

```
--VHDL P-to-S 8-bit shift register, LSB first

process (CLK)
begin
   if CLK'event and CLK = '1' then
      if RST = '1' then
         SHIFT <= X"00";
         SHIFT_CNT <= "000";
      elsif LOAD = '1' then
         SHIFT <= PDATA;
         SHIFT_CNT <= SHIFT_CNT + '1';
      elsif  SHIFT_CNT > "000" then
         SHIFT(6 downto 0) <= SHIFT(7 downto 1);
         SHIFT_CNT <= SHIFT_CNT + '1';
      end if;
   end if;
end process;
SDATA <= SHIFT(0);
```

```
//Verilog P-to-S shift register, LSB first

always @ (posedge CLK)
begin
   if  (RST) begin
      SHIFT = 8'h00;
      SHIFT_CNT = 3'b000;
      end
   else if (LOAD == 1'b1) begin
      SHIFT = PDATA;
      SHIFT_CNT = SHIFT_CNT + 1;
      end
   else if (SHIFT_CNT > 3'b000) begin
      SHIFT[6:0] = SHIFT[7:1];
      SHIFT_CNT = SHIFT_CNT + 1;
      end
end
assign SDATA = SHIFT[0];
```

```
--VHDL S-to-P 8-bit shift register, MSB first

process (CLK)
begin
   if CLK'event and CLK = '1' then
      if RST = '1' then
         SHIFT <= X"00";
         SHIFT_CNT <= "000";
      else
         SHIFT(7 downto 1) <= SHIFT(6 downto 0);
         SHIFT(0) <= SDATA;
         SHIFT_CNT <= SHIFT_CNT + '1';
      end if;
      if SHIFT_CNT = "000" then
         PDATA <= SHIFT;
         PDATA_EN <= '1';
      else
         PDATA_EN <= '0';
      end if;
   end if;
end process;
```

```
//Verilog S-to-PS shift register, MSB first

always @ (posedge CLK)
begin
   if  (RST) begin
      SHIFT = 8'h00;
      SHIFT_CNT = 3'b000;
      end
   else begin
      SHIFT[7:1] = SHIFT[6:0];
      SHIFT[0] = SDATA;
      SHIFT_CNT = SHIFT_CNT + 1;
      end
   if (SHIFT_CNT == 3'b000) begin
      PDATA = SHIFT;
      PDATA_EN = 1'b1;
      end
   else
       PDATA_EN = 1'b0;
end
```

```
--VHDL S-to-P 8-bit shift register, LSB first

process (CLK)
begin
   if CLK'event and CLK = '1' then
      if RST = '1' then
         SHIFT <= X"00";
         SHIFT_CNT <= "000";
      else
         SHIFT(6 downto 0) <= SHIFT(7 downto 1);
         SHIFT(7) <= SDATA;
         SHIFT_CNT <= SHIFT_CNT + '1';
      end if;
      if SHIFT_CNT = "000" then
         PDATA <= SHIFT;
         PDATA_EN <= '1';
      else
         PDATA_EN <= '0';
      end if;
   end if;
end process;
```

```
//Verilog S-to-PS shift register, LSB first

always @ (posedge CLK)
begin
   if  (RST) begin
      SHIFT = 8'h00;
      SHIFT_CNT = 3'b000;
      end
   else begin
      SHIFT[6:0] = SHIFT[7:1];
      SHIFT[7] = SDATA;
      SHIFT_CNT = SHIFT_CNT + 1;
      end
   if (SHIFT_CNT == 3'b000) begin
      PDATA = SHIFT;
      PDATA_EN = 1'b1;
      end
   else
       PDATA_EN = 1'b0;
end
```

*ROM Design*

A Read Only Memory (ROM) can be implemented as a look-up table using a **case** statement or as a synchronous ROM. Also, ROM can be implemented using vendor specific core generation tools such as CORE Generator from Xilinx, MegaWizard from Altera, or SmartGen from Actel. The generated ROM components are then instantiated in the HDL code. The ROM cores generated by these tools need to be simulated before instantiation to avoid different interpretations of the ROM design by different vendors.

```
--VHDL ROM design using case statement

process (ADDRESS)
begin
   case ADDRESS is
      when "00" => DATA <= X"7";
      when "01" => DATA <= X"6";
      when "10" => DATA <= X"C";
      when others => DATA <= X"E";
   end case;
end process;
```

```
//Verilog ROM design using case statement

always @ (ADDRESS)
   begin
      case (ADDRESS)
         2'b00 : DATA = 4'h7;
         2'b01 : DATA = 4'h6;
         2'b10 : DATA = 4'hC;
         2'b11 : DATA = 4'hE;
      endcase
   end
```

```
--VHDL synchronous ROM

entity syncc_ROM is
   port (CLK     : in std_logic;
         ADDR : in std_logic_vector(3 downto 0);
         DOUT : out std_logic_vector(7 downto 0));
end syncc_ROM;

architecture behavioral of syncc_ROM is
   type rom is array(15 downto 0) of std_logic_vector(7 downto 0);
   constant SROM  : rom :=(X"01", X"20", X"AA", X"A0",
                           X"0F", X"1C", X"34", X"C4",
                           X"99", X"32", X"70", X"DD",
                           X"E3", X"B6", X"DB", X"12");
begin

process (CLK)
begin
   if CLK'event and CLK = '1' then
      DOUT <= SROM(conv_integer(ADDR));
   end if;
end process;
end behavioral;
```

```
//Verilog synchronous ROM

module SROM (CLK,ADDR,DOUT);
   input CLK;
   input [7:0] ADDR;
   output [7:0] DOUT;
   reg [7:0] SROM [255:0];
   reg [7:0] DOUT;
   initial begin
     $readmemb("rom_table.list",SROM);
   end
   always @ (posedge CLK)
      begin
         DOUT = SROM[ADDR];
      end
endmodule
```

*RAM Design*

Random Access Memories (RAM) come in different forms including single-port synchronous, dual-port synchronous, and dual-port asynchronous. The single-port synchronous RAM can be used whenever simultaneous writes and reads from two different memory locations are not required. This is the simplest RAM implementation that can be easily tested and verified. If simultaneous reads and writes are necessary, the dual-port synchronous RAM should be used.

```
--VHDL single-port synchronous RAM

entity sp_sync_RAM is
    port (CLK  : in std_logic;
          WE   : in std_logic;
          ADDR : in std_logic_vector(7 downto 0);
          DIN  : in std_logic_vector(7 downto 0);
          DOUT : out std_logic_vector(7 downto 0));
end sp_sync_RAM;

architecture behavioral of sp_sync_RAM is
    type ram is array(255 downto 0) of std_logic_vector(7 downto 0);
    signal SP_RAM  : ram;
begin

process (CLK)
begin
    if CLK'event and CLK = '1' then
        if WE = '1' then
            SP_RAM(conv_integer(ADDR)) <= DIN;
        end if;
        DOUT <= SP_RAM(conv_integer(ADDR));
    end if;
end process;
end behavioral;


//Verilog single-port synchronous RAM

module sp_sync_RAM (CLK, WE, ADDR, DIN, DOUT);
    input CLK;
    input WE;
    input [3:0] ADDR;
    input [7:0] DIN;
    output [7:0] DOUT;
    reg [7:0] SP_RAM [15:0];
    reg [7:0] DOUT;
    always @ (posedge CLK)
        begin
            DOUT = SP_RAM[ADDR];
            if  (WE) SP_RAM[ADDR] = DIN;
        end
endmodule
```

```
--VHDL dual-port read-first synchronous RAM

entity dp_sync_RAM is
    port (CLK     : in std_logic;
          WE      : in std_logic;
          RE      : in std_logic;
          WADDR : in std_logic_vector(7 downto 0);
          RADDR : in std_logic_vector(7 downto 0);
          DIN     : in std_logic_vector(7 downto 0);
          DOUT : out std_logic_vector(7 downto 0));
end dp_sync_RAM;

architecture behavioral of dp_sync_RAM is
    type ram is array(255 downto 0) of std_logic_vector(7 downto 0);
    signal DP_RAM  : ram;
begin

    process (CLK)
       begin
          if CLK'event and CLK = '1' then
             if WE = '1' then
                DP_RAM(conv_integer(WADDR)) <= DIN;
             end if;
             if RE = '1' then
                DOUT <= DP_RAM(conv_integer(RADDR)) ;
             end if;
          end if;
    end process;
end behavioral;
```

```
//Verilog dual-port read-first synchronous RAM

module dp_sync_RAM (CLK, WE, RE,  WADDR, RADDR, DIN, DOUT);
    input CLK;
    input WE;
    input RE;
    input [7:0] WADDR;
    input [7:0] RADDR;
    input [7:0] DIN;
    output [7:0] DOUT;
    reg [7:0] DP_RAM [255:0];
    reg [7:0] DOUT;

    always @ (posedge CLK)
       begin
          if  (WE) DP_RAM[WADDR] = DIN;
       end
    always @ (posedge CLK)
       begin
          if (RE) DOUT = DP_RAM[RADDR];
       end
endmodule
```

*FIFO Design*

FIFO is usually designed as dual-port synchronous RAM with write and read counters controlling the write and the read address.

```
--VHDL synchronous FIFO

   RAM_WRITE:process (CLK)
      begin
         if CLK'event and CLK = '1' then
            if WE = '1' and FF = '0' then
               FIFO_RAM(conv_integer(WADDR)) <= DIN;
            end if;
         end if;
   end process;

   RAM_READ:process (CLK)
      begin
         if CLK'event and CLK = '1' then
            if RE = '1' and EF = '0'  then
               DOUT <= FIFO_RAM(conv_integer(RADDR)) ;
            end if;
         end if;
   end process;
            end if;
         end if;
    end process;

READ_POINTER:process (CLK,CLR)
      begin
         if CLR = '1' then
            RADDR <= X"00";
         elsif CLK'event and CLK = '1' then
            if RE = '1' and EF = '1' then
               RADDR <= RADDR + '1';
            end if;
         end if;
      end process;
```

```
FFLAG:process (CLK,CLR) --active high
   begin
      if CLR = '1' then
         FF <= '0';
      elsif CLK'event and CLK = '1' then
         if RE = '1' then
            FF <= '0';
         elsif WE = '1' and WADDR = RADDR - '1'  then
            FF <= '1';
         end if;
      end if;
end process;

EFLAG:process (CLK,CLR) --active high
   begin
      if CLR = '1' then
         EF <= '1';
      elsif CLK'event and CLK = '1' then
         if WE = '1' then
            EF <= '0';
         elsif RE = '1' and WADDR = RADDR + '1' then
            EF <= '1';
         end if;
      end if;
end process;
```

```
//Verilog synchronous FIFO

always @ (posedge CLK)  // RAM write
      begin
         if  (WE) FIFO_RAM[WADDR] = DIN;
      end

   always @ (posedge CLK)  // RAM read
      begin
         if (RE) DOUT = FIFO_RAM[RADDR];
      end

   always @ (posedge CLK or posedge CLR)  // Write pointer
      begin
         if (CLR) WADDR = 8'h00;
         else if (WE & !FF) WADDR  = WADDR + 1;
      end

   always @ (posedge CLK or posedge CLR)  // Read pointer
      begin
         if (CLR) RADDR = 8'h00;
         else if (RE & !EF) RADDR  = RADDR + 1;
      end

  always @ (posedge CLK or posedge CLR)  // Full flag
      begin
         if (CLR) FF = 1'b0;
         else if (RE) FF = 1'b0;
         else if ((WE) & (WADDR == RADDR - 1)) FF = 1'b1;
      end

   always @ (posedge CLK or posedge CLR)  // Empty flag
      begin
         if (CLR) EF = 1'b1;
         else if (WE) EF = 1'b0;
         else if ((RE) & (WADDR == RADDR + 1)) EF = 1'b1;
      end
```

*State Machines*

If a state-machine contains states that transition to themselves for all input combinations (deadlock states), an external reset signal needs to be implemented to move the state-machine from the deadlock state. Furthermore, the reset signal is necessary to ensure that the state-machine initially starts in a known state. If the reset signal is not synchronous, it needs to be resynchronized properly.

```
--VHDL state machine with 4 states, two inputs and three outputs using if-elsif statements

process (CLK)
begin
   if CLK'event and CLK = '1' then
       if RST = '1' then
          O <= "000";
          STATE <= "00";
       elsif STATE = "00" then
          if IN1 = "00" then
             O <= "000";
             STATE <= "00";
          elsif IN1 = "01" then
             O <= "111";
             STATE <= "01";
          elsif IN1 = "10"  then
             O <= "101";
             STATE <= "00";
          else
             O <= "001";
             STATE <= "11";
          end if;
      elsif STATE = "01" then
          if IN1 = "00" then
             O <= "000";
             STATE <= "00";
          elsif IN1 = "01" then
             O <= "110";
             STATE <= "01";
          elsif IN1 = "10" then
             O <= "001";
             STATE <= "10";
          else
             O <= "110";
             STATE <= "11";
          end if;
      elsif STATE = "10" then
          if IN1 = "00" then
             O <= "000";
             STATE <= "10";
          elsif IN1 = "01" then
             O <= "101";
             STATE <= "11";
          elsif IN1 = "10" then
             O <= "010";
             STATE <= "00";
          else
             O <= "110";
             STATE <= "01";
          end if;
      elsif STATE = "11" then
          if IN1 = "00" then
             O <= "001";
             STATE <= "11";
          elsif IN1 = "01" then
             O <= "100";
             STATE <= "11";
          elsif IN1 = "10" then
             O <= "001";
             STATE <= "00";
          else
             O <= "101";
             STATE <= "01";
          end if;
       end if;
    end if;
end process;
```

```
--VHDL state machine with 4 states, two inputs and three outputs using case statements

process(CLK,RST)
begin
   if RST = '1' then
      O <= "000";
      STATE <= "00";
   elsif CLK'event and CLK = '1' then
      case STATE is
         when "00" =>
            case IN1 is
               when "00" =>
                  O <= "000";
                  STATE <= "00";
               when "01" =>
                  O <= "111";
                  STATE <= "01";
               when "10" =>
                  O <= "101";
                  STATE <= "00";
               when others =>
                  O <= "001";
                  STATE <= "11";
               end case;
         when "01" =>
            case IN1 is
               when "00" =>
                  O <= "010";
                  STATE <= "01";
               when "01" =>
                  O <= "111";
                  STATE <= "01";
               when "10" =>
                  O <= "101";
                  STATE <= "00";
               when others =>
                  O <= "110";
                  STATE <= "11";
               end case;
         when "10" =>
            case IN1 is
               when "00" =>
                  O <= "000";
                  STATE <= "10";
               when "01" =>
                  O <= "101";
                  STATE <= "01";
               when "10" =>
                  O <= "100";
                  STATE <= "00";
               when others =>
                  O <= "001";
                  STATE <= "11";
               end case;
         when others =>
            case IN1 is
               when "00" =>
                  O <= "101";
                  STATE <= "11";
               when "01" =>
                  O <= "110";
                  STATE <= "01";
               when "10" =>
                  O <= "000";
                  STATE <= "00";
               when others =>
                  O <= "001";
                  STATE <= "10";
               end case;
            end case;
         end if;

end process;
```

//Verilog state machine with 4 states, two inputs and three outputs using if-else if statements

```
always @ (posedge CLK)
   begin
      if (RST == 1'b1) begin
        O = 3'b000;
        STATE = 2'b00;
        end
      else if (STATE == 2'b00) begin
        if (IN1 == 2'b00) begin
           O = 3'b000;
           STATE = 2'b00;
           end
        else if (IN1 == 2'b01) begin
           O = 3'b111;
           STATE = 2'b01;
           end
        else if (IN1 == 2'b10) begin
           O = 3'b101;
           STATE = 2'b00;
           end
        else begin
           O = 3'b001;
           STATE = 2'b11;
           end
        end
      else if (STATE == 2'b01) begin
        if (IN1 == 2'b00) begin
           O = 3'b000;
           STATE = 2'b00;
           end
        else if (IN1 == 2'b01) begin
           O = 3'b100;
           STATE = 2'b01;
           end
        else if (IN1 == 2'b10) begin
           O = 3'b001;
           STATE = 2'b10;
           end
        else begin
           O = 3'b110;
           STATE = 2'b11;
           end
        end
      else if (STATE == 2'b10) begin
        if (IN1 == 2'b00) begin
           O = 3'b000;
           STATE = 2'b10;
           end
        else if (IN1 == 2'b01) begin
           O = 3'b101;
           STATE = 2'b11;
           end
        else if (IN1 == 2'b10) begin
           O = 3'b010;
           STATE = 2'b00;
           end
        else begin
           O = 3'b110;
           STATE = 2'b01;
           end
        end
      else if (STATE == 2'b11) begin
        if (IN1 == 2'b00) begin
           O = 3'b001;
           STATE = 2'b11;
           end
        else if (IN1 == 2'b01) begin
           O = 3'b100;
           STATE = 2'b11;
           end
        else if (IN1 == 2'b10) begin
           O = 3'b001;
           STATE = 2'b00;
           end
        else begin
           O = 3'b101;
           STATE = 2'b01;
           end
        end
      end
```

```
//Verilog state machine with 4 states, two inputs and three outputs using case statements

always @ (posedge CLK)
  begin
    if (RST == 1'b1) begin
      O = 3'b000;
      STATE = 2'b00;
      end
    else if (STATE == 2'b00) begin
      if (IN1 == 2'b00) begin
        O = 3'b000;
        STATE = 2'b00;
        end
      else if (IN1 == 2'b01) begin
        O = 3'b111;
        STATE = 2'b01;
        end
      else if (IN1 ==2'b10) begin
        O = 3'b101;
        STATE = 2'b00;
        end
      else begin
        O = 3'b001;
        STATE = 2'b11;
        end
      end
    else if (STATE == 2'b01) begin
      if (IN1 == 2'b00) begin
        O = 3'b000;
        STATE = 2'b00;
        end
      else if (IN1 == 2'b01) begin
        O = 3'b100;
        STATE = 2'b01;
        end
      else if (IN1 == 2'b10) begin
        O = 3'b001;
        STATE = 2'b10;
        end
      else begin
        O = 3'b110;
        STATE = 2'b11;
        end
      end
Pa
    else if (STATE == 2'b10) begin
      if (IN1 == 2'b00) begin
        O = 3'b000;
        STATE = 2'b10;
        end
      else if (IN1 == 2'b01) begin
        O = 3'b101;
        STATE = 2'b11;
        end
      else if (IN1 == 2'b10) begin
        O = 3'b010;
        STATE = 2'b00;
        end
      else begin
        O = 3'b110;
        STATE = 2'b01;
        end
      end
    else if (STATE == 2'b11) begin
      if (IN1 == 2'b00) begin
        O = 3'b001;
        STATE = 2'b11;
        end
      else if (IN1 == 2'b01) begin
        O = 3'b100;
        STATE = 2'b11;
        end
      else if (IN1 == 2'b10) begin
        O = 3'b001;
        STATE = 2'b00;
        end
      else begin
        O = 3'b101;
        STATE = 2'b01;
        end
      end
    end
```

To improve code readability, use named rather than positional association for the port mapping when instantiating a sub-module. Also, a single port mapping per line is preferred over the single-line port mapping.

```
--VHDL

INST0 : tbuf port map (
   I => DATA_IN,
  O => DATA_OUT,
  T => DATA_ENB
  );
```

```
//Verilog

tbuf INST0 (
    .I  (DATA_IN),
  .O (DATA_OUT),
  .T (DATA_ENB)
  );
```

*Constants and Parameters*

Using constants and parameters to substitute numbers helps readability and portability of the code.

```
--VHDL

constant INIT : std_logic_vector(1 downto 0) := "00";
constant SLOW : std_logic_vector(1 downto 0) := "01";
constant FAST : std_logic_vector(1 downto 0) := "10";
signal STATE : std_logic_vector(1 downto 0);

begin
   if STATE = INIT then
      O <= '0';
   elsif STATE = SLOW then
      O <= IN1;
   elsif STATE = FAST then
      O <= not IN1;
   else
      O <= '1';
   end if;
```

```
//Verilog

parameter INIT = 2'b00;
parameter SLOW = 2'b01;
parameter FAST = 2'b10;
wire STATE;

always @ (posedge CLK)
begin
   if (STATE == INIT) begin
      O = 1'b0;
      end
   else if (STATE == SLOW) begin
      O =;
      end
   else if (STATE == FAST) begin
      O = !IN;
      end
   else begin
      O = 1'b1;
      end
end
```

*Indenting and Spacing*

To have better readability of the code and to reduce coding errors, use proper indentation and spacing. The following example shows proper code indentation.

```
--VHDL

process (CLK,RST)
begin
   if RST = '1' then
      O <= '0';
   elsif CLK'event and CLK = '1' then
      if SEL = '1' then
         O <= I1;
      else
         O <= I2;
      end if;
   end if;
end process;
```

```
//Verilog

always @ (posedge CLK or posedge RST)
begin
   if (RST)
      O = 0;
   else if (SEL == 1'b1)
      O = I1;
   else
      O = I2;
end
```

*Dynamic Parameters*

Implementing variable bus and array widths using generics and parameters helps code reuse and readability.

```
--VHDL

entity adder is
   generic (
            IN_WIDTH : integer := 16;
            OUT_WIDTH : integer := 16
            );
port (
      IN1 : in std_logic_vector(IN_WIDTH-1 downto 0);
      IN2 : in std_logic_vector(IN_WIDTH-1 downto 0);
      O : out std_logic_vector(OUT_WIDTH-1 downto 0);
      );
end adder;
architecture behavioral of adder is

begin

   O <= IN1 + IN2;

end behavioral;
```

```
//Verilog

module (IN1, IN2, OUT);
parameter IN_WIDTH = 16;
parameter OUT_WIDTH = 16;
input IN1 [IN_WIDTH-1 : 0];
input IN2 [IN_WIDTH-1 : 0];
output OUT [OUT_WIDTH-1 : 0];

always @ (IN1, IN2)
   begin
      OUT = IN1 + IN2;
   end
end module
```

*Hierarchical Code*

Avoid using flat-module designs where all the code resides in a single file. Using hierarchical design makes the code easier to read, trace, and verify. Also, it makes the team work on large designs easier. In the hierarchical code, use the top-level code for component/module declaration and instantiation. The behavioral code should generally be placed at the lowest hierarchical level.

```
--VHDL file top.vhd

entity top is
port (
      IN1 : in std_logic_vector(15 downto 0);
      IN2 : in std_logic_vector(15 downto 0);
      IN3 : in std_logic_vector(15 downto 0);
      O : out std_logic_vector(15 downto 0)
      );
end top;
architecture Behavioral of top is

component adder is
port (
      IN1 : in std_logic_vector(15 downto 0);
      IN2 : in std_logic_vector(15 downto 0);
      O : out std_logic_vector(15 downto 0)
      );
end component;

signal SUM1 : std_logic_vector(15 downto 0);

begin

INST0 : adder port map (
   IN1 => IN1,
   IN2  => IN2,
   O => SUM1
  );

INST1 : adder port map (
   IN1 => IN3,
   IN2  => SUM1,
   O => O
  );

end Behavioral;
```

```
--VHDL file adder.vhd

entity adder is
port (
      IN1 : in std_logic_vector(15 downto 0);
      IN2 : in std_logic_vector(15 downto 0);
      O : out std_logic_vector(15 downto 0)
      );
end adder;
architecture Behavioral of adder is

begin

O <= IN1 + IN2;

end Behavioral;
```

```
//Verilog file top.v

module (IN1, IN2, IN3, OUT);
input IN1 [15 : 0];
input IN2 [15 : 0];
input IN3 [15 : 0];
output OUT [15 : 0];
wire SUM1 [15 :0];
always @ (IN1, IN2, IN3)
   begin

   adder INST0 (
      .IN1 (IN1),
      .IN2 (IN2),
      .OUT (SUM1)
      );

   adder INST1 (
      .IN1 (IN1),
      .IN2 (SUM1),
      .OUT (OUT)
      );
   end
end module;
```

```
//Verilog file adder.v

module (IN1, IN2, OUT);
input IN1 [15 : 0];
input IN2 [15 : 0];
output OUT [15 : 0];
always @ (IN1, IN2)
   begin
      OUT = IN1 + IN2;
   end
end module
```