```
-------------------------------------------------------------------
-- n-bit Register (ESD book figure 2.6)
-- by Weijun Zhang, 04/2001
--
-- KEY WORD: concurrent, generic and range
-------------------------------------------------------------------

library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-------------------------------------------------------------------

entity reg is

generic(n: natural :=2);
port(   I:      in std_logic_vector(n-1 downto 0);
        clock:  in std_logic;
        load:   in std_logic;
        clear:  in std_logic;
        Q:      out std_logic_vector(n-1 downto 0)
);
end reg;

-------------------------------------------------------------------

architecture behv of reg is

    signal Q_tmp: std_logic_vector(n-1 downto 0);

begin

    process(I, clock, load, clear)
    begin

        if clear = '0' then
            -- use 'range in signal assigment
            Q_tmp <= (Q_tmp'range => '0');
        elsif (clock='1' and clock'event) then
            if load = '1' then
                Q_tmp <= I;
            end if;
        end if;

    end process;

    -- concurrent statement
    Q <= Q_tmp;

end behv;

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

```
-- Example of doing multiplication showing
-- (1) how to use variable with in process
-- (2) how to use for loop statement
-- (3) algorithm of multiplication
--
-- by Weijun Zhang, 05/2001

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- two 4-bit inputs and one 8-bit outputs
entity multiplier is
port(   num1, num2:     in std_logic_vector(3 downto 0);
        product:        out std_logic_vector(7 downto 0)
);
end multiplier;

architecture behv of multiplier is

begin
process(num1, num2)

  variable num1_reg: std_logic_vector(8 downto 0);
  variable product_reg: std_logic_vector(5 downto 0);

begin

  num1_reg := '0' & num1;
  product_reg := "0000" & num2;

  -- use variables doing computation
  -- algorithm is to repeat shifting/adding
  for i in 1 to 3 loop
    if product_reg(0)='1' then
          product_reg(5 downto 3) := product_reg(5 downto 3)
          + num1_reg(2 downto 0);
        end if;
        product_reg(5 downto 0) := '0' & product_reg(5 downto 1);
  end loop;

  -- assign the result of computation back to output signal
  product <= product_reg(3 downto 0);

end process;

end behv;
```

Full Adder

```
-------------------------------------------------------------
-- VHDL code for n-bit adder (ESD figure 2.5)
-- by Weujun Zhang, 04/2001
--
-- function of adder:
-- A plus B to get n-bit sum and 1 bit carry
-- we may use generic statement to set the parameter
-- n of the adder.
-------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-------------------------------------------------------------

entity ADDER is

generic(n: natural :=2);
port(   A:      in std_logic_vector(n-1 downto 0);
        B:      in std_logic_vector(n-1 downto 0);
        carry:  out std_logic;
        sum:    out std_logic_vector(n-1 downto 0)
);

end ADDER;

-------------------------------------------------------------

architecture behv of ADDER is

-- define a temparary signal to store the result

signal result: std_logic_vector(n downto 0);

begin

    -- the 3rd bit should be carry

    result <= ('0' & A)+('0' & B);
    sum <= result(n-1 downto 0);
    carry <= result(n);

end behv;

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

ALU

```
-------------------------------------------------
-- Simple ALU Module (ESD book Figure 2.5)
-- by Weijun Zhang, 04/2001
--
-- ALU stands for arithmatic logic unit.
-- It perform multiple operations according to
-- the control bits.
-- we use 2's complement subraction in this example
-- two 2-bit inputs & carry-bit ignored
-------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

-------------------------------------------------

entity ALU is

port(   A:      in std_logic_vector(1 downto 0);
        B:      in std_logic_vector(1 downto 0);
        Sel:    in std_logic_vector(1 downto 0);
        Res:    out std_logic_vector(1 downto 0)
);

end ALU;

-------------------------------------------------

architecture behv of ALU is
begin

    process(A,B,Sel)
    begin

        -- use case statement to achieve
        -- different operations of ALU

        case Sel is
            when "00" =>
                Res <= A + B;
            when "01" =>
                Res <= A + (not B) + 1;
            when "10" =>
                Res <= A and B;
            when "11" =>
                Res <= A or B;
            when others =>
                Res <= "XX";
        end case;

    end process;

end behv;

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

Decoder

```
-------------------------------------------------------
-- 2:4 Decoder (ESD figure 2.5)
-- by Weijun Zhang, 04/2001
--
-- decoder is a kind of inverse process
-- of multiplexor
-------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

-------------------------------------------------------

entity DECODER is
port(   I:      in std_logic_vector(1 downto 0);
        O:      out std_logic_vector(3 downto 0)
);
end DECODER;

-------------------------------------------------------

architecture behv of DECODER is
begin

    -- process statement

    process (I)
    begin

        -- use case statement

        case I is
            when "00" => O <= "0001";
            when "01" => O <= "0010";
            when "10" => O <= "0100";
            when "11" => O <= "1000";
            when others => O <= "XXXX";
        end case;

    end process;

end behv;

architecture when_else of DECODER is
begin

    -- use when..else statement

    O <=        "0001" when I = "00" else
                "0010" when I = "01" else
                "0100" when I = "10" else
                "1000" when I = "11" else
                "XXXX";

end when_else;

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

D Latch

```
-------------------------------------------------
-- Simple D Latch (ESD book Chapter 2.3.1)
-- by Weijun Zhang, 04/2001
--
-- latch is simply controlled by enable bit
-- but has nothing to do with clock sigal
-- notice this difference from flip-flops
-------------------------------------------------

library ieee ;
use ieee.std_logic_1164.all;

-------------------------------------------------

entity D_latch is
port(   data_in:        in std_logic;
        enable:         in std_logic;
        data_out:       out std_logic
);
end D_latch;

-------------------------------------------------

architecture behv of D_latch is
begin

    -- compare this to D flipflop

    process(data_in, enable)
    begin
        if (enable='1') then
            -- no clock signal here
            data_out <= data_in;
        end if;
    end process;

end behv;

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

```
-------------------------------------------------
-- VHDL code for 4:1 multiplexor
-- (ESD book figure 2.5)
-- by Weijun Zhang, 04/2001
--
-- Multiplexor is a device to select different
-- inputs to outputs. we use 3 bits vector to
-- describe its I/O ports
-------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

-------------------------------------------------

entity Mux is
port(   I3:     in std_logic_vector(2 downto 0);
        I2:     in std_logic_vector(2 downto 0);
        I1:     in std_logic_vector(2 downto 0);
        I0:     in std_logic_vector(2 downto 0);
        S:      in std_logic_vector(1 downto 0);
        O:      out std_logic_vector(2 downto 0)
);
end Mux;

-------------------------------------------------

architecture behv1 of Mux is
begin
    process(I3,I2,I1,I0,S)
    begin

        -- use case statement
        case S is
            when "00" =>        O <= I0;
            when "01" =>        O <= I1;
            when "10" =>        O <= I2;
            when "11" =>        O <= I3;
            when others =>      O <= "ZZZ";
        end case;

    end process;
end behv1;

architecture behv2 of Mux is
begin

    -- use when.. else statement
    O <=        I0 when S="00" else
                I1 when S="01" else
                I2 when S="10" else
                I3 when S="11" else
                "ZZZ";

end behv2;
-------------------------------------------------
```

RAM

```
-------------------------------------------------------------
-- a simple 4*4 RAM module (ESD book Chapter 5)
-- by Weijun Zhang
--
-- KEYWORD: array, concurrent processes, generic, conv_integer
-------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-------------------------------------------------------------

entity SRAM is
generic(        width:  integer:=4;
                depth:  integer:=4;
                addr:   integer:=2);
port(   Clock:          in std_logic;
        Enable:         in std_logic;
        Read:           in std_logic;
        Write:          in std_logic;
        Read_Addr:      in std_logic_vector(addr-1 downto 0);
        Write_Addr:     in std_logic_vector(addr-1 downto 0);
        Data_in:        in std_logic_vector(width-1 downto 0);
        Data_out:       out std_logic_vector(width-1 downto 0)
);
end SRAM;

-------------------------------------------------------------

architecture behav of SRAM is

-- use array to define the bunch of internal temparary signals

type ram_type is array (0 to depth-1) of
        std_logic_vector(width-1 downto 0);
signal tmp_ram: ram_type;

begin

    -- Read Functional Section
    process(Clock, Read)
    begin
        if (Clock'event and Clock='1') then
            if Enable='1' then
                if Read='1' then
                    -- buildin function conv_integer change the type
                    -- from std_logic_vector to integer
                    Data_out <= tmp_ram(conv_integer(Read_Addr));
                else
                    Data_out <= (Data_out'range => 'Z');
                end if;
            end if;
        end if;
    end process;

    -- Write Functional Section
    process(Clock, Write)
    begin
        if (Clock'event and Clock='1') then
            if Enable='1' then
```

```
            if Write='1' then
                tmp_ram(conv_integer(Write_Addr)) <= Data_in;
            end if;
        end if;
    end if;
end process;

end behav;
```

RAM code