### VHDL Operators

<table>
<thead>
<tr>
<th>VHDL Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication*</td>
</tr>
<tr>
<td>/</td>
<td>Division*</td>
</tr>
<tr>
<td>MOD</td>
<td>Modulus*</td>
</tr>
<tr>
<td>REM</td>
<td>Remainder*</td>
</tr>
<tr>
<td>&amp;</td>
<td>Concatenation – used to combine bits</td>
</tr>
<tr>
<td>SLL**</td>
<td>logical shift left</td>
</tr>
<tr>
<td>SRL**</td>
<td>logical shift right</td>
</tr>
<tr>
<td>SLA**</td>
<td>arithmetic shift left</td>
</tr>
<tr>
<td>SRA**</td>
<td>arithmetic shift right</td>
</tr>
<tr>
<td>ROL**</td>
<td>rotate left</td>
</tr>
<tr>
<td>ROR**</td>
<td>rotate right</td>
</tr>
<tr>
<td>=</td>
<td>equality</td>
</tr>
<tr>
<td>/=</td>
<td>Inequality</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less that or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>NOT</td>
<td>logical NOT</td>
</tr>
<tr>
<td>AND</td>
<td>logical AND</td>
</tr>
<tr>
<td>OR</td>
<td>logical OR</td>
</tr>
<tr>
<td>NAND</td>
<td>logical NAND</td>
</tr>
<tr>
<td>NOR</td>
<td>logical NOR</td>
</tr>
<tr>
<td>XOR</td>
<td>logical XOR</td>
</tr>
<tr>
<td>XNOR*</td>
<td>logical XNOR</td>
</tr>
</tbody>
</table>

*Not supported in many VHDL synthesis tools. In the Quartus II tools, only multiply and divide by powers of two (shifts) are supported. Mod and Rem are not supported in Quartus II. Efficient design of multiply or divide hardware typically requires the user to specify the arithmetic algorithm and design in VHDL.*

**Supported only in 1076-1993 VHDL.
Table 6.2 STD_LOGIC conversion functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONV_STD_LOGIC_VECTOR</strong> (integer, bits)</td>
<td><strong>CONV_STD_LOGIC_VECTOR</strong> (7, 4)</td>
</tr>
<tr>
<td>Converts an integer to a standard logic</td>
<td>Produces a standard logic vector of</td>
</tr>
<tr>
<td>vector. Useful to enter constants. **CONV</td>
<td>&quot;0111&quot;.</td>
</tr>
<tr>
<td>SIGNED** and <strong>CONV_UNSIGNED</strong> work in a</td>
<td></td>
</tr>
<tr>
<td>similar way to produce signed and unsigned</td>
<td></td>
</tr>
<tr>
<td>values.</td>
<td></td>
</tr>
<tr>
<td><strong>CONV_INTEGER</strong> (std_logic_vector)</td>
<td><strong>CONV_INTEGER</strong> (&quot;0111&quot;)</td>
</tr>
<tr>
<td>Converts a standard logic vector to an integer. Useful for array indexing when using a std_logic_vector signal for the array index.</td>
<td>Produces an integer value of 7.</td>
</tr>
</tbody>
</table>
LIBRARY IEEE; -- Include Libraries for standard logic data types
USE IEEE.STD_LOGIC_1164.ALL;
    -- Entity name normally the same as file name
ENTITY gate_network IS -- Ports: Declares module inputs and outputs
PORT( A, B, C : IN STD_LOGIC;
     -- Standard Logic Vector ( Array of 4 Bits )
     D : IN STD_LOGIC_VECTOR( 3 DOWNTO 0 );
     -- Output Signals
     X, Y : OUT STD_LOGIC );
END gate_network;

-- Defines internal module architecture
ARCHITECTURE behavior OF gate_network IS
BEGIN
    -- Concurrent assignment statements operate in parallel
    -- D(1) selects bit 1 of standard logic vector D
    X <= A AND NOT( B OR C ) AND ( D( 1 ) XOR D( 2 ) );
    -- Process must declare a sensitivity list,
    -- In this case it is ( A, B, C, D )
    -- List includes all signals that can change the outputs
    PROCESS ( A, B, C, D )
    BEGIN
        -- Statements inside process execute sequentially
        Y <= A AND NOT( B OR C ) AND ( D( 1 ) XOR D( 2 ) );
    END PROCESS;
END behavior;
LED_MSD_DISPLAY: -- BCD to 7 Segment Decoder for LED Displays
PROCESS (MSD)
BEGIN
  CASE MSD IS
    WHEN "0000" =>
      MSD_7SEG <= "1111110";
    WHEN "0001" =>
      MSD_7SEG <= "0110000";
    WHEN "0010" =>
      MSD_7SEG <= "1101101";
    WHEN "0011" =>
      MSD_7SEG <= "1111001";
    WHEN "0100" =>
      MSD_7SEG <= "0110011";
    WHEN "0101" =>
      MSD_7SEG <= "1011011";
    WHEN "0110" =>
      MSD_7SEG <= "1011111";
    WHEN "0111" =>
      MSD_7SEG <= "1110000";
    WHEN "1000" =>
      MSD_7SEG <= "1111111";
    WHEN "1001" =>
      MSD_7SEG <= "1111011";
    WHEN OTHERS =>
      MSD_7SEG <= "0111110";
  END CASE;
END PROCESS LED_MSD_DISPLAY;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY multiplexer IS
    PORT( A, B, Mux_Control : IN STD_LOGIC;
          Mux_Out1, Mux_Out2, Mux_Out3, Mux_Out4 : OUT STD_LOGIC );
END multiplexer;

ARCHITECTURE behavior OF multiplexer IS
BEGIN

    -- selected signal assignment statement…
    Mux_Out1 <= A WHEN Mux_Control = '0' ELSE B;
    -- … with Select Statement
    WITH mux_control SELECT
        Mux_Out2 <= A WHEN '0', B WHEN '1', A WHEN OTHERS;
        -- OTHERS case required since STD_LOGIC
        -- has values other than "0" or "1"

    PROCESS ( A, B, Mux_Control )
    BEGIN
        IF Mux_Control = '0' THEN
            Mux_Out3 <= A;
        ELSE
            Mux_out3 <= B;
        END IF;

        CASE Mux_Control IS
            WHEN '0' =>
                Mux_Out4 <= A;
            WHEN '1' =>
                Mux_Out4 <= B;
            WHEN OTHERS =>
                Mux_Out4 <= A;
        END CASE;
    END PROCESS;
END behavior;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY tristate IS
  PORT( A, Control : IN STD_LOGIC;
        Tri_out : INOUT STD_LOGIC); -- Use Inout for bi-directional tri-state
                                 -- signals or out for output only
END tristate;

ARCHITECTURE behavior OF tristate IS
BEGIN
  Tri_out <= A WHEN Control = '0' ELSE 'Z'; -- Assignment of 'Z' value generates
                                           -- tri-state output
END behavior;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY DFFs IS
  PORT( D, Clock, Reset, Enable : IN STD_LOGIC;
        Q1, Q2, Q3, Q4 : OUT STD_LOGIC );
END DFFs;

ARCHITECTURE behavior OF DFFs IS
BEGIN
  PROCESS
    -- Positive edge triggered D flip-flop
    -- If WAIT is used no sensitivity list is used
    -- If WAIT is used no sensitivity list is used
    WAIT UNTIL ( Clock 'EVENT AND Clock = '1' );
    Q1 <= D;
  END PROCESS;

END DFFs;
PROCESS -- Positive edge triggered D flip-flop
BEGIN -- with synchronous reset
  WAIT UNTIL ( Clock 'EVENT AND Clock = '1' );
  IF reset = '1' THEN
    Q2 <= '0';
  ELSE
    Q2 <= D;
  END IF;
END PROCESS;

PROCESS (Reset,Clock) -- Positive edge triggered D flip-flop
BEGIN -- with asynchronous reset
  IF reset = '1' THEN
    Q3 <= '0';
  ELSIF ( clock 'EVENT AND clock = '1' ) THEN
    Q3 <= D;
  END IF;
END PROCESS;

PROCESS (Reset,Clock) -- Positive edge triggered D flip-flop
BEGIN -- with asynchronous reset and enable
  IF reset = '1' THEN
    Q4 <= '0';
  ELSIF ( clock 'EVENT AND clock = '1' ) THEN
    IF Enable = '1' THEN
      Q4 <= D;
    END IF;
  END IF;
END PROCESS;
END behavior;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY ilatch IS
  PORT( A, B       : IN  STD_LOGIC;
        Output1, Output2 : OUT STD_LOGIC );
END ilatch;

ARCHITECTURE behavior OF ilatch IS
BEGIN
  PROCESS ( A, B )
  BEGIN
    IF A = '0' THEN
      Output1 <= '0';
      Output2 <= '0';
    ELSE
      IF B = '1' THEN
        Output1 <= '1';
        Output2 <= '1';
      ELSE
        -- Latch inferred since no value is assigned
        Output1 <= '0';
        -- to output2 in the else clause!
      END IF;
    END IF;
  END PROCESS;
END behavior;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY Counter IS
  PORT( Clock, Reset : IN STD_LOGIC;
        Max_count : IN STD_LOGIC_VECTOR( 7 DOWNTO 0 );
        Count : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 ) );
END Counter;

ARCHITECTURE behavior OF Counter IS
  SIGNAL internal_count: STD_LOGIC_VECTOR( 7 DOWNTO 0 );
BEGIN
  count <= internal_count;
  PROCESS ( Reset, Clock )
  BEGIN
    -- Reset counter
    IF reset = '1' THEN
      internal_count <= "00000000";
    ELSIF ( clock 'EVENT AND clock = '1' ) THEN
      IF internal_count < Max_count THEN -- Check for maximum count
        internal_count <= internal_count + 1; -- Increment Counter
      ELSE -- Count >= Max_Count
        internal_count <= "00000000"; -- reset Counter
      END IF;
    END IF;
  END PROCESS;
END behavior;
Figure 6.1 State Diagram for st_mach example
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY st_mach IS
  PORT( clk, reset : IN STD_LOGIC;
       Input1, Input2 : IN STD_LOGIC;
       Output1   : OUT STD_LOGIC);
END st_mach;

ARCHITECTURE A OF st_mach IS
  -- Enumerated Data Type for State
  TYPE STATE_TYPE IS ( state_A, state_B, state_C );
  SIGNAL state: STATE_TYPE;
BEGIN
  PROCESS ( reset, clk )
  BEGIN
    IF reset = '1' THEN -- Reset State
      state <= state_A;
    ELSIF clk 'EVENT AND clk = '1' THEN
CASE state IS
  -- Define Next State Transitions using a Case
  --       Statement based on the Current State
  WHEN state_A =>
    IF Input1 = '0' THEN
      state <= state_B;
    ELSE
      state <= state_C;
    END IF;
  WHEN state_B =>
    state <= state_C;
  WHEN state_C =>
    IF Input2 = '1' THEN
      state <= state_A;
    END IF;
  WHEN OTHERS =>
    state <= state_A;
END CASE;
END IF;
END PROCESS;

WITH state SELECT
  -- Define State Machine Outputs
  Output1 <= '0' WHEN state_A,
            '1' WHEN state_B,
            '0' WHEN state_C;
END a;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY ALU IS
  PORT( Op_code : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
        A_input, B_input : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        ALU_output : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) );
END ALU;

ARCHITECTURE behavior OF ALU IS
  -- Declare signal(s) internal to module here
  SIGNAL temp_output : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN

  PROCESS ( Op_code, A_input, B_input )
  BEGIN


CASE Op_Code ( 2 DOWNTO 1 ) IS
   -- Select Arithmetic/Logical Operation
   WHEN "00" =>
      temp_output <= A_input + B_input;
   WHEN "01" =>
      temp_output <= A_input - B_input;
   WHEN "10" =>
      temp_output <= A_input AND B_input;
   WHEN "11" =>
      temp_output <= A_input OR B_input;
   WHEN OTHERS =>
      temp_output <= "00000000";
END CASE;

   -- Select Shift Operation: Shift bits left with zero fill using concatenation operator
   -- Can also use VHDL 1076-1993 shift operator such as SLL

IF Op_Code( 0 ) = '1' THEN
   Alu_output <= temp_output( 6 DOWNTO 0 ) & '0';
ELSE
   Alu_output <= temp_output;
END IF;
END PROCESS;
END behavior;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY mult IS
  PORT( A, B : IN STD_LOGIC_VECTOR( 7 DOWNTO 0 );
       Product : OUT STD_LOGIC_VECTOR( 15 DOWNTO 0 ) );
END mult;

ARCHITECTURE a OF mult IS
BEGIN -- LPM 8x8 multiply function P = A * B
  multiply: lpm_mult
    GENERIC MAP ( LPM_WIDTHA => 8,
                   LPM_WIDTHB => 8,
                   LPM_WIDTHS => 16,
                   LPM_WIDTHP => 16,
                   LPM_REPRESENTATION => "UNSIGNED" )
    PORT MAP ( data => A,
                datab => B,
                result => Product );
END a;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY memory IS
PORT( read_data : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
     read_address : IN STD_LOGIC_VECTOR( 2 DOWNTO 0 );
     write_data : IN STD_LOGIC_VECTOR( 7 DOWNTO 0 );
     write_address : IN STD_LOGIC_VECTOR( 2 DOWNTO 0 );
     Memwrite : IN STD_LOGIC;
     Clock : IN STD_LOGIC );
END memory;

ARCHITECTURE behavior OF memory IS
  -- define new data type for memory array
  TYPE memory_type IS ARRAY ( 0 TO 7 ) OF STD_LOGIC_VECTOR( 7 DOWNTO 0 );
  SIGNAL memory : memory_type;
BEGIN
  -- Read Memory and convert array index to an integer with CONV_INTEGER
  read_data <= memory( CONV_INTEGER( read_address( 2 DOWNTO 0 ) ) );

  PROCESS
  BEGIN
    -- Write Memory?
    WAIT UNTIL clock 'EVENT AND clock = '1';
    IF ( memwrite = '1' ) THEN
      -- convert array index to an integer with CONV_INTEGER
      memory( CONV_INTEGER( write_address( 2 DOWNTO 0 ) ) ) <= write_data;
    END IF;
  END PROCESS;
END behavior;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY memory IS
  PORT( read_data : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
        read_address : IN STD_LOGIC_VECTOR( 2 DOWNTO 0 );
        write_data   : IN STD_LOGIC_VECTOR( 7 DOWNTO 0 );
        write_address : IN STD_LOGIC_VECTOR( 2 DOWNTO 0 );
        Memwrite   : IN STD_LOGIC;
        clock,reset : IN STD_LOGIC );
END memory;

ARCHITECTURE behavior OF memory IS
  SIGNAL mem0, mem1 : STD_LOGIC_VECTOR( 7 DOWNTO 0 );

BEGIN
  PROCESS (read_address, mem0, mem1) -- Process for memory read operation
  BEGIN
    CASE read_address IS
      WHEN "000" =>
        read_data <= mem0;
      WHEN "001" =>
        read_data <= mem1;
      WHEN OTHERS => -- Unimplemented memory locations
        read_data <= X"FF";
    END CASE;
  END PROCESS;
END memory;
PROCESS
BEGIN
  WAIT UNTIL clock 'EVENT AND clock = '1';
  IF ( reset = '1' ) THEN
    mem0 <= X"55" ; -- Initial values for memory (optional)
    mem1 <= X"AA" ;
  ELSE
    IF memwrite = '1' THEN -- Write to memory?
      CASE write_address IS -- Use a flip-flop with
        WHEN "000" => -- an enable for memory
          mem0 <= write_data;
        WHEN "001" =>
          mem1 <= write_data;
        WHEN OTHERS => -- unimplemented memory locations
          NULL;
      END CASE;
    END IF;
  END IF;
END PROCESS;
END behavior;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
LIBRARY Altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY amemory IS
    PORT( read_data : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
         memory_address : IN STD_LOGIC_VECTOR( 2 DOWNTO 0 );
         write_data : IN STD_LOGIC_VECTOR( 7 DOWNTO 0 );
         Memwrite : IN STD_LOGIC;
         clock,reset : IN STD_LOGIC );
END amemory;

ARCHITECTURE behavior OF amemory IS
BEGIN
    data_memory: altsyncram -- Altsyncram memory function
        GENERIC MAP ( operation_mode => "SINGLE_PORT",
                       width_a => 8,
                       widthad_a => 3,
                       lpm_type => "altsyncram",
                       outdata_reg_a => "UNREGISTERED",
                       -- Reads in mif file for initial data values (optional)
                       init_file => "memory.mif",
                       intended_device_family => "Cyclone" )
        PORT MAP (wren_a => Memwrite, clock0 => clock,
                   address_a => memory_address( 2 DOWNTO 0 ),
                   data_a => write_data, q_a => read_data );
END behavior;
Figure 6.2 Schematic of Hierarchical Design
ENTITY hierarch IS
  PORT ( clock_25Mhz, pb1 : IN STD_LOGIC;
         pb1_single_pulse : OUT STD_LOGIC);
END hierarch;

ARCHITECTURE structural OF hierarch IS
  -- Declare internal signals needed to connect submodules
  SIGNAL clock_1MHz, clock_100Hz, pb1_debounced : STD_LOGIC;
  COMPONENT debounce
    -- Use Components to Define Submodules and Parameters
    PORT ( pb, clock_100Hz : IN STD_LOGIC;
           pb_debounced : OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT onepulse
    PORT (pb_debounced, clock : IN STD_LOGIC;
          pb_single_pulse : OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT clk_div
    PORT ( clock_25Mhz : IN STD_LOGIC;
           clock_1MHz : OUT STD_LOGIC;
           clock_100KHz : OUT STD_LOGIC;
           clock_10KHz : OUT STD_LOGIC;
           clock_1KHz : OUT STD_LOGIC;
           clock_100Hz : OUT STD_LOGIC;
           clock_10Hz : OUT STD_LOGIC;
           clock_1Hz : OUT STD_LOGIC);
  END COMPONENT;
BEGIN
  -- Use Port Map to connect signals between components in the hierarchy
  debounce1 : debounce PORT MAP ( pb => pb1, clock_100Hz => clock_100Hz,
                                 pb_debounced => pb1_debounced);
  prescalar : clk_div PORT MAP ( clock_25Mhz => clock_25Mhz,
                                 clock_1MHz => clock_1MHz,
                                 clock_100Hz => clock_100Hz);
  single_pulse : onepulse PORT MAP ( pb_debounced => pb1_debounced,
                                      clock => clock_1MHz,
                                      pb_single_pulse => pb1_single_pulse);
END structural;
Figure 6.3 Using a Testbench for automatic verification