VHDL

תיאור פקודות לקורס תכן לוגי

Entity and Architecture

Entity is a declaration of a component, architecture is the implementation of the component. It is possible to have several different architectures for one entity (choose which one with Configuration).

```
entity AndGate is
  port(
    a, b : in  std_logic;   -- inputs
    c    : out std_logic);  -- outputs
end AndGate;

architecture beh of AndGate is
begin
  c <= a and b;
end beh;
```

Architecture with Components

It is possible to build a component using other sub-components defined elsewhere. This enables a modular design (divide one large component into several smaller ones).

```
entity NandGate is
  port(
    A, B : in  std_logic;   -- inputs
    C    : out std_logic);  -- outputs
end NandGate;

architecture struct of NandGate is
  signal my_sig : std_logic;  -- we need an extra signal for connecting both

  component AndGate  -- this is a copy of the entity of AndGate
    port(
      a, b : in  std_logic;   -- inputs
      c    : out std_logic);  -- outputs
  end component;

  component NotGate  -- this is a copy of the entity of NotGate
    port(
      in1  : in  std_logic;   -- inputs
      out1 : out std_logic);  -- outputs
  end component;

begin
  my_and : AndGate                               -- one instance of type AndGate
    port map( a => A , b => B , c => my_sig );  -- connections: component => external
  my_not : NotGate                               -- one instance of type NotGate
    port map( in1 => my_sig , out1 => C );      -- connections: component => external
end struct;
```

Architecture with Processes

It is possible to build a component using sequencial logic (similar to a programming language). It is possible to have several processes, and also to combine both processes and sub-components and normal combinational logic.

Sensitivity List: A process runs according to a sensitivity list. If one signal in the list is updated, the process re-runs. If the sensitivity list is empty, the process will loop forever. Each process runs once on simulation start. If the process updates one of its own sensitivity list signals, it will run again when it terminates (sensitivity updated).

Process Run Algorithm: Run once on simulation start. When finished running, check if sensitivity signals are different from when we started this run. If so, run again from the start. If not, wait until a sensitivity signal is updated. If the sensitivity list is empty, always run again (loop forever).

Process Signal Assignments: (1) If a signal has several different assignments, only the last one is relevant. (2) All signal assignments are executed together when the process finishes. So all calculations using signals in the middle of the process are always done with the original signal value from when the process started. (3) Signals that are changed in a process are <u>always</u> changed (even if they are inside an IF that is not executed). If it is not stated to which value the signal should be changed, it will change to its original value from the start of the run.

Multiple Processes: All processes run together in the same time, collisions will result 'X' values.

```
architecture arch of NandGate is
begin

  process(A, B)   -- sensitivity list, run process again if any of these change
  begin
    C <= not ( A and B );
  end process;

end arch;
```

Signals

A signal is the basic combinational data type. All the inputs and outputs of a component are signals.

Signal Bit Values: The standard values for a single bit are '0' and '1' (boolean values). The additional simulation values are 'X' and 'Z'. If the same signal is assigned by 2 different components different values it will receive the value 'X' to indicate an error (example: 2 components write to a bus in the same time). When you want to indicate that a signal does not have a value (simply not connected) assign it the 'Z' value. This means "high Z" (used in tri-state buffers). In a bus, most components will assign the bus 'Z' except the current talker which will assign '0' or '1'.

```
architecture arch of Something
  signal one_bit : std_logic;                          -- extra signal declarations here
  signal b1, b2  : std_logic;                          -- defined in IEEE.std_logic_1164
  signal bit_vec : std_logic_vector(31 downto 0);   -- a bit vector with 32 bits
  signal v1, v2  : std_logic_vector(7 downto 0);    -- bit vectors with 8 bits

begin
  one_bit         <= '0';               -- assign 0
  b1              <= 'Z';               -- assign "high z"
  b2              <= '1' after 15 ns;   -- assign after some delay
  bit_vec         <= "11110000111100001111000011110000";
  bit_vec(0)      <= '1';    -- assign only one bit (the lsb) in the entire vector
  v1              <= bit_vec(10 downto 3);
  v1(7 downto 5)  <= "101";
  v2              <= ( 2 => '1' , others => '0');  -- assign the value 0x04
  v2              <= "101" & v1(7) & bit_vec(31 downto 30) & "00";
  v2              <= X"3A";  -- assign 0x3a

  b1              <= not ( (b1 and b2) or (b1 xor '1') or (b2 nand '0') );
  v1              <= not ( (v1 and v2) or (v1 xor X"FF") or (v2 nand X"00") );
  v1(3 downto 0)  <= v2(2 downto 0) + "002";           -- keep the carry
  v1              <= v2 + X"30" - v1;                  -- ignore the carry
  v1              <= v1(3 downto 0) * v2(3 downto 0);  -- need double the bits for res

  b1 <= '0' when b2 = '1' else b2 when v1(0) /= '0' else 'Z';

  with bit_vec(1 downto 0) select
    v1 <= "00000000" when "00",
          "00000001" when "01",
          v2         when "10",
          "11111111" when others;
```

Sequential Statements in Processes

Since a process runs all statements in sequence, it has special sequencial syntax which can't be used in normal combinational logic. Processes can also have variables which are special virtual data holders that exist only inside the process (just like variables in a programming language) – variables and signals are different.

```
process (b1, b2)
  variable var : std_logic_vector(3 downto 0);  -- all variable declarations here
  variable j   : integer := 0;

begin
  var := "1101";  -- variables use := for assignment (signals use <= )

  if (b1 ='0') or ((b2 /= '1') and (b1 = 'Z')) then b1 <= '1';
  elsif (v1 > v2) or (v1 <= "00001111") then b1 <= '0';
  end if;

  if (v1(3 downto 0) = "0000") then v1 <= "0001"; else v1 <= "1111";
  end if;

  for i in 3 downto 0 loop
    var(i) := v2(i);
  end loop;

  for i in 7 - conv_integer(v2) + 1 to 7 loop
    v1(i) <= v1(7 - i);
  end loop;

  while (j < 256) loop
    j := j + 1;
  end loop;

  case v1 is
    when "0000" => var := v2(3 downto 0);
    when "0001" => var := v1(3 downto 0);
    when "0101" => v2 <= "0000" & var;
    when others => var := (others => 'X');
  end case;

  wait for 10 ns;  -- use wait; to wait endlessly
```

Mux

A simple example of a 2 to 1 mux with N bits as inputs and outputs (for example choose between two 32 bit values).

```
entity Mux_2to1 is
  generic( WIDTH : integer := 32);
  port(
    sel          : in  std_logic;
    d_in1, d_in2 : in  std_logic_vector( (WIDTH-1) downto 0 );
    d_out        : out std_logic_vector( (WIDTH-1) downto 0 ));
end Mux_2to1;

architecture beh of Mux_2to1 is
begin
  with sel select
    d_out <= d_in1 when '0',
             d_in2 when '1',
             (others => 'X') when others;
end beh;
```

Register

A simple example of a register with N bits and support for load.

```
entity Reg_with_load is
  generic( WIDTH : integer := 32);
  port(
    rst, clk, load : in  std_logic;
    d_in           : in  std_logic_vector( (WIDTH - 1) downto 0);
    d_out          : out std_logic_vector( (WIDTH - 1) downto 0));
end Reg_with_load;

architecture beh of Reg_with_load is
begin

  process(clk, rst)
  begin
    if (rst = '1') then d_out <= (others => '0');
    elsif (clk'event and clk = '1') then
      if (load = '1') then
        d_out <= d_in;
      end if;
    end if;
  end process;

end beh;
```

Finite State Machine

A simple example of how to make a controller with a finite number of states.

```
entity Controller is
  port(
    rst, clk : in  std_logic;
    input    : in  std_logic;   -- inputs (from the datapath or external)
    output   : out std_logic);  -- outputs (controls to the datapath or external)
end Controller;

architecture beh of Controller is
  type state_type is (S0, S1, S2, S3, S4);
  signal curr_state, next_state : state_type;

process (clk, rst)  -- this process is the state register
begin
  if (rst = '1') then curr_state <= S0;  -- upon reset enter state S0
  elsif (clk'event and clk = '1') then curr_state <= next_state;
  end if;
end process;

process (rst, input, curr_state)  -- this process is the ROM table
begin
  output <= '0';           -- init default values for outputs in each state
  case curr_state is       -- handle each state (S0, S1, S2, S3, S4)
    when S0 =>
      output <= '0';       -- choose output for this state
      next_state <= S1;    -- choose next state
    when S1 =>
      output <= '1';       -- choose output for this state
      next_state <= S3;    -- choose next state
    when others =>
  end case;
end process

end beh;
```