VHDL Coding Styles for Synthesis

Dr. Aiman H. El-Maleh

Computer Engineering Department

King Fahd University of Petroleum & Minerals

Outline…

- **Synthesis overview**
- **Synthesis of primary VHDL constructs**
  - Constant definition
  - Port map statement
  - When statement
  - With statement
  - Case statement
  - For statement
  - Generate statement
  - If statement
  - Variable definition
- **Combinational circuit synthesis**
  - Multiplexor
  - Decoder
  - Priority encoder
  - Adder
  - Tri-state buffer
  - Bi-directional buffer

…Outline

- **Sequential circuit synthesis**
  - Latch
  - Flip-flop with asynchronous reset
  - Flip-flop with synchronous reset
  - Loadable register
  - Shift register
  - Register with tri-state output
  - Finite state machine
- **Efficient coding styles for synthesis**

General Overview of Synthesis…

- **Synthesis is the process of translating from an abstract description of a hardware device into an optimized, technology specific gate level implementation.**
- **Synthesis may occur at many different levels of abstraction**
  - Behavioral synthesis
  - Register Transfer Level (RTL) synthesis
  - Boolean equations descriptions, netlists, block diagrams, truth tables, state tables, etc.
- **RTL synthesis implements the register usage, the data flow, the control flow, and the machine states as defined by the syntax & semantics of the HDL.**

…General Overview of Synthesis

- **Forces driving the synthesis algorithm**
  - HDL coding style
  - Design constraints
    - Timing goals
    - Area goals
    - Power management goals
    - Design-For-Test rules
  - Target technology
    - Target library design rules
- **The HDL coding style used to describe the targeted device is technology independent.**
- **HDL coding style determines the initial starting point for the synthesis algorithms & plays a key role in generating the final synthesized hardware.**

VHDL Synthesis Subset

- **VHDL is a complex language but only a subset of it is synthesizable.**
- **Primary VDHL constructs used for synthesis:**
  - Constant definition
  - Port map statement
  - Signal assignment: A <= B
  - Comparisons: = (equal), /= (not equal), > (greater than), < (less than), >= (greater than or equal, <= (less than or equal)
  - Logical operators: **AND**, **OR**, **NAND**, **NOR**, **XOR**, **XNOR**, **NOT**
  - 'if' statement
    - **if** (presentstate = CHECK_CAR) **then** ....
    - **end if** | **elsif** ....
  - 'for' statement (used for looping in creating arrays of elements)
  - Other constructs are ‘**with’**, **’when’**, '**when else**', '**case**' , '**wait** '. Also "**:=**" for variable assignment.

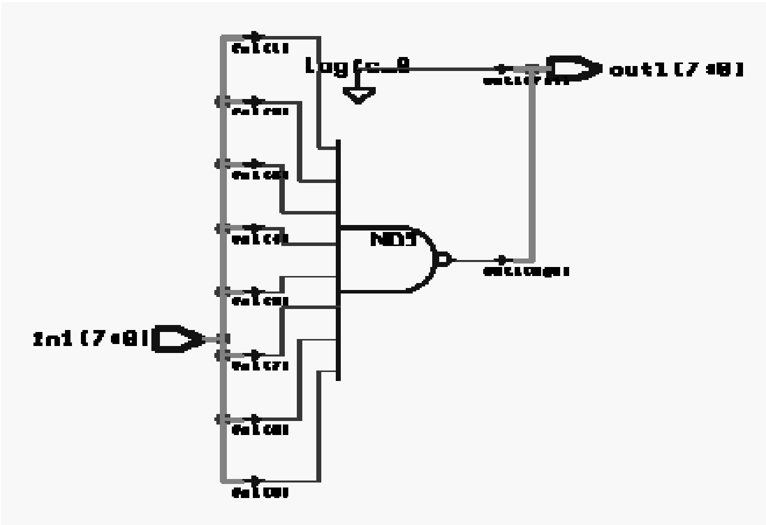
Outline

- **Synthesis overview**
- **Synthesis of primary VHDL constructs**
  - Constant definition
  - Port map statement
  - When statement
  - With statement
  - Case statement
  - For statement
  - Generate statement
  - If statement
  - Variable definition
- **Combinational circuit synthesis**
  - Multiplexor
  - Decoder
  - Priority encoder
  - Adder
  - Tri-state buffer
  - Bi-directional buffer

Constant Definition…

```
library ieee;
use ieee.std_logic_1164.all;
entity constant_ex is
    port (in1 : in std_logic_vector (7 downto 0); out1 : out
    std_logic_vector (7 downto 0));
end constant_ex;
 architecture constant_ex_a of constant_ex is
    constant A : std_logic_vector (7 downto 0) := "00000000";
     constant B : std_logic_vector (7 downto 0) := "11111111";
    constant C : std_logic_vector (7 downto 0) := "00001111";
 begin
    out1 <= A when in1 = B else C;
 end constant_ex_a;
```

…Constant Definition



Port Map Statement…

```
library ieee;
 use ieee.std_logic_1164.all;
entity sub is
    port (a, b : in std_logic; c : out std_logic);
 end sub;
architecture sub_a of sub is
begin
    c <= a and b;
end sub_a;
```

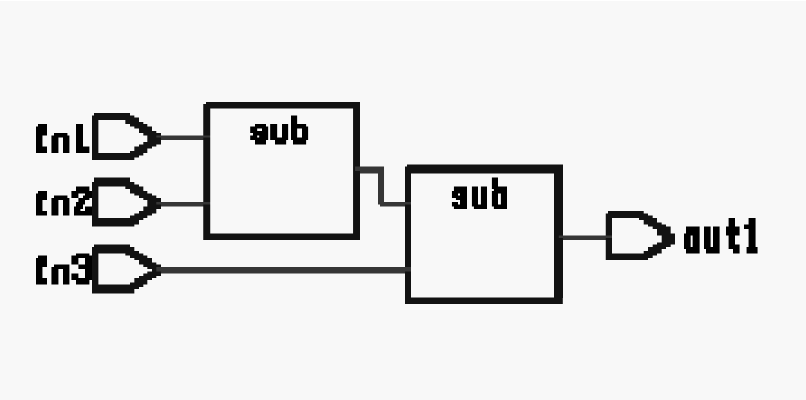
…Port Map Statement…

```
library ieee;
use ieee.std_logic_1164.all;
entity portmap_ex is
   port (in1, in2, in3 : in std_logic; out1 : out std_logic);
end portmap_ex;
architecture portmap_ex_a of portmap_ex is
   component sub
         port (a, b : in std_logic; c : out std_logic);
   end component;
   signal temp : std_logic;
```


…Port Map Statement…

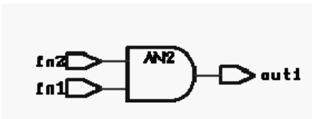
```
begin
   u0 : sub port map (in1, in2, temp);
   u1 : sub port map (temp, in3, out1);
end portmap_ex_a;
use work.all;
configuration portmap_ex_c of  portmap_ex is
    for portmap_ex_a
         for u0,u1 : sub use entity sub (sub_a);
         end for;
    end for;
end portmap_ex_c;
```

Port Map Statement…



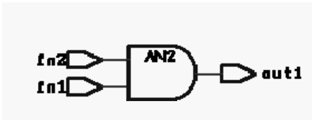
When Statement

```
library ieee;
use ieee.std_logic_1164.all;
entity when_ex is
    port (in1, in2 : in std_logic; out1 : out std_logic);
end when_ex;
architecture when_ex_a of when_ex is
begin
    out1 <= '1' when in1 = '1' and in2 = '1' else '0';
end when_ex_a;
```



With Statement

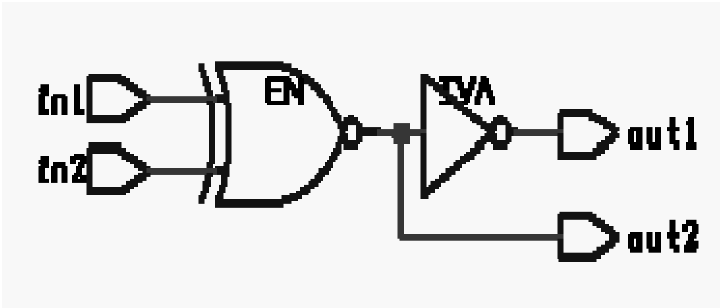
```
library ieee;
use ieee.std_logic_1164.all;
entity with_ex is
    port (in1, in2 : in std_logic; out1 : out std_logic);
end with_ex;
architecture with_ex_a of with_ex is
begin
    with in1 select out1 <= in2 when '1',
                             '0' when others;
end with_ex_a;
```



Case Statement…

```
library ieee;
use ieee.std_logic_1164.all;
entity case_ex is
     port (in1, in2 : in std_logic; out1,out2 : out std_logic);
end case_ex;
architecture case_ex_a of case_ex is
    signal b : std_logic_vector (1 downto 0);
begin
    process (b)
    begin
           case b is
                    when "00"|"11" => out1 <= '0'; out2 <= '1';
                    when others => out1 <= '1'; out2 <= '0';
          end case;
    end process;
     b <= in1 & in2;
end case_ex_a;
```

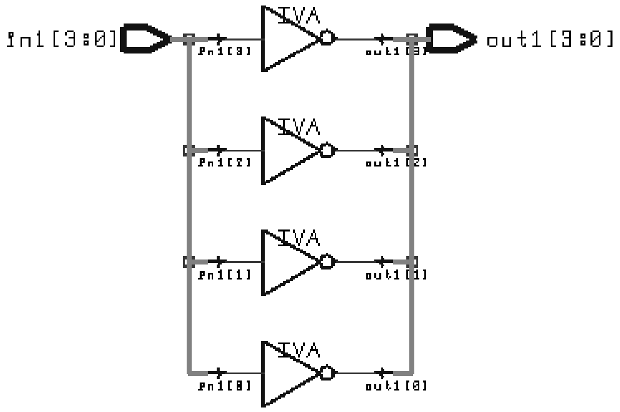
…Case Statement



For Statement…

```
library ieee;
use ieee.std_logic_1164.all;
entity for_ex is
   port (in1 : in std_logic_vector (3 downto 0); out1 : out
   std_logic_vector (3 downto 0));
end for_ex;
architecture for_ex_a of for_ex is
begin
   process (in1)
   begin
       for0 : for i in 0 to 3 loop
              out1 (i) <= not in1(i);
       end loop;
    end process;
end for_ex_a;
```

…For Statement

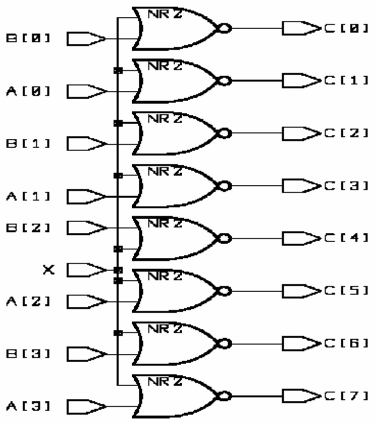


Generate Statement

signal A,B:BIT_VECTOR (3 downto 0);
signal C:BIT_VECTOR (7 downto 0);
signal X:BIT;

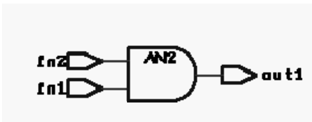
. . .

```
GEN_LABEL:
    for I in 3 downto 0 generate
        C(2*I+1) <= A(I) nor X;
        C(2*I) <= B(I) nor X;
    end generate GEN_LABEL
```



If Statement

```
library ieee;
use ieee.std_logic_1164.all;
entity if_ex is
   port (in1, in2 : in std_logic; out1 : out std_logic);
end if_ex;
architecture if_ex_a of if_ex is
   begin
       process (in1, in2)
       begin
              if in1 = '1' and in2 = '1' then out1 <= '1';
              else out1 <= '0';
              end if;
        end process;
end if_ex_a;
```



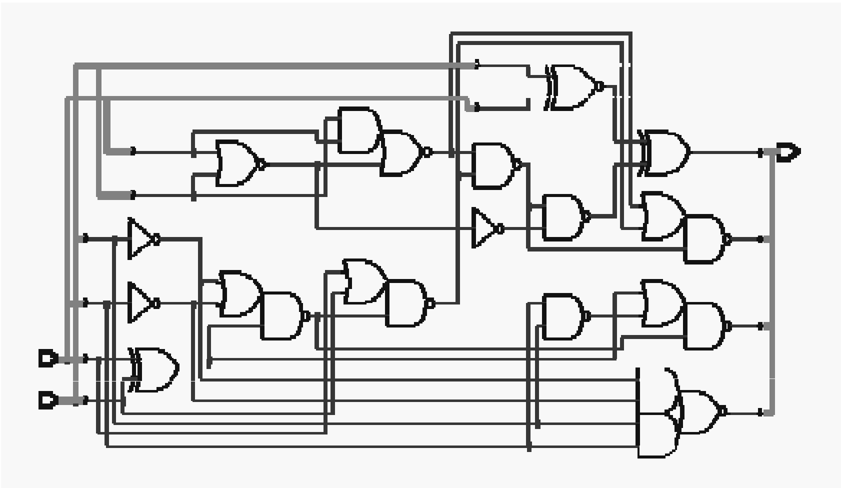
Variable Definition…

```
library ieee;
use ieee.std_logic_1164.all;
entity variable_ex is
   port ( a : in std_logic_vector (3 downto 0); b : in std_logic_vector
   (3 downto 0); c : out std_logic_vector (3 downto 0));
end variable_ex;
architecture variable_ex_a of variable_ex is
begin
   process (a,b)
        variable carry : std_logic_vector (4 downto 0);
        variable sum : std_logic_vector (3 downto 0);
```

…Variable Definition…

```
    begin
        carry (0) := '0';
        for i in 0 to 3 loop
            sum (i) := a(i) xor b(i) xor carry(i);
            carry (i+1) := (a(i) and b(i)) or (b(i) and carry (i))
                           or (carry (i) and a(i));
        end loop;
    c <= sum;
    end process;
end variable_ex_a;
```


…Variable Definition

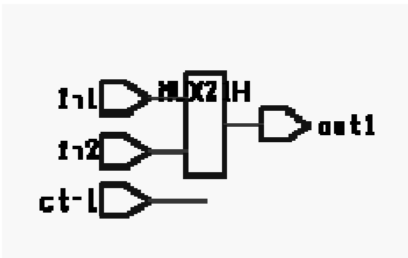


Outline

- **Synthesis overview**
- **Synthesis of primary VHDL constructs**
  - Constant definition
  - Port map statement
  - When statement
  - With statement
  - Case statement
  - For statement
  - Generate statement
  - If statement
  - Variable definition
- **Combinational circuit synthesis**
  - Multiplexor
  - Decoder
  - Priority encoder
  - Adder
  - Tri-state buffer
  - Bi-directional buffer

Multiplexor Synthesis…

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
   port (in1, in2, ctrl : in std_logic; out1 : out std_logic);
end mux;
architecture mux_a of mux is
begin
   process (in1, in2, ctrl)
   begin
        if ctrl = '0' then out1 <= in1;
        else out1 <= in2;
        end if;
   end process;
end mux_a;
```



…Multiplexor Synthesis

entity *mux2to1_8* is

port (signal *s*: in std_logic; signal *zero,one*: in std_logic_vector(7 downto 0); signal *y*: out std_logic_vector(7 downto 0));

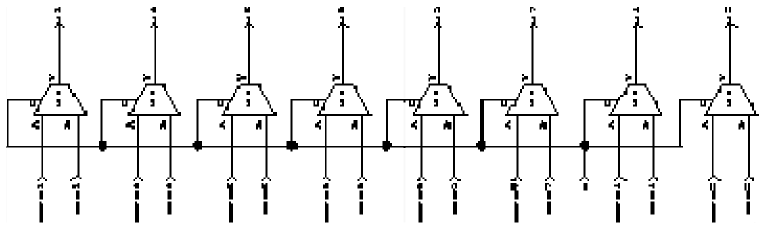
end *mux2to1_8*;

architecture *behavior* of *mux2to1* is

begin

***y* <= *one* when (*s* = '1') else *zero*;**

end *behavior*;



9-27

2x1 Multiplexor using Booleans

architecture *boolean_mux* of *mux2to1_8* is

signal *temp*: std_logic_vector(7 downto 0);

begin

temp <= (others => s);

***y* <= (*temp* and *one*) or (not *temp* and *zero*);**

end boolean_mux;

- **The *s* signal cannot be used in a Boolean operation with the *one* or *zero* signals because of type mismatch (*s* is a std_logic type, *one/zero* are std_logic_vector types)**
- **An internal signal of type std_logic_vector called *temp* is declared. The *temp* signal will be used in the Boolean operation against the *zero/one* signals.**
- **Every bit of *temp* is set equal to the *s* signal value.**

9-28

2x1 Multiplexor using a Process

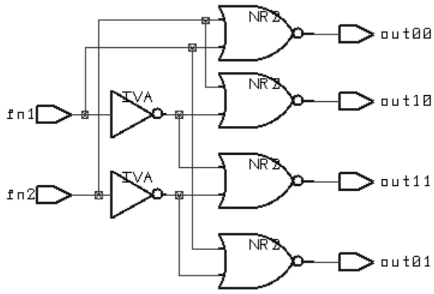
```
architecture process_mux of mux2to1_8 is
begin
    comb: process (s, zero, one)
    begin
        y <= zero;
        if (s = '1') then
           y <= one;
        end if;
    end process comb;
end process_mux ;
```


Decoder Synthesis…

```
library ieee;
use ieee.std_logic_1164.all;
entity decoder is
     port (in1, in2 : in std_logic; out00, out01, out10, out11 : out std_logic);
end decoder;
architecture decoder_a of decoder is
begin
    process (in1, in2)
    begin
        if in1 = '0' and in2 = '0' then out00 <= '1';
        else out00 <= '0';
        end if;
        if in1 = '0' and in2 = '1' then out01 <= '1';
        else out01 <= '0';
        end if;
```

…Decoder Synthesis

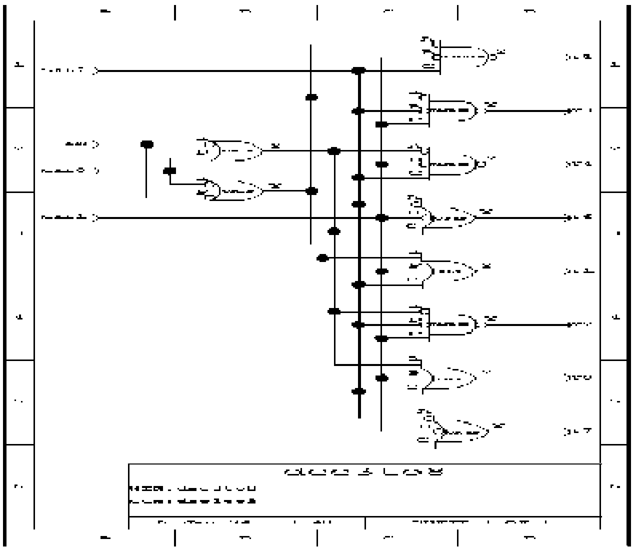
```
        if in1 = '1' and in2 = '0' then out10 <= '1';
        else out10 <= '0';
        end if;
        if in1 = '1' and in2 = '1' then out11 <= '1';
        else out11 <= '0';
        end if;
    end process;
end decoder_a;
```



3-to-8 Decoder Example…

```
entity dec3to8 is
    port (signal sel: in std_logic_vector(3 downto 0); signal en: in std_logic;
    signal y: out std_logic_vector(7 downto 0))
end dec3to8;
architecture behavior of dec3to8 is
begin
    process (sel, en)
        y <= "1111111";
        if (en = '1')  then
                case sel is
                        when "000" => y(0) <= '0';     when "001" => y(1) <= '0';
                        when "010" => y(2) <= '0';     when "011" => y(3) <= '0';
                        when "100" => y(4) <= '0';     when "101" => y(5) <= '0';
                        when "110" => y(6) <= '0';     when "111" => y(7) <= '0';
                end case;
        end if;
    end process;
end behavior;
```

…3-to-8 Decoder Example



Architecture of Generic Decoder

```
architecture behavior of generic_decoder is
begin
    process (sel, en)
    begin
        y <= (others => '1') ;
         for i in y'range loop
                if ( en = '1' and bvtoi(To_Bitvector(sel)) = i ) then
                        y(i) <= '0' ;
                end if ;
        end loop;
    end process;
end behavior;
```

bvtoi is a function to convert from bit_vector to integer

A Common Error in Process Statements…

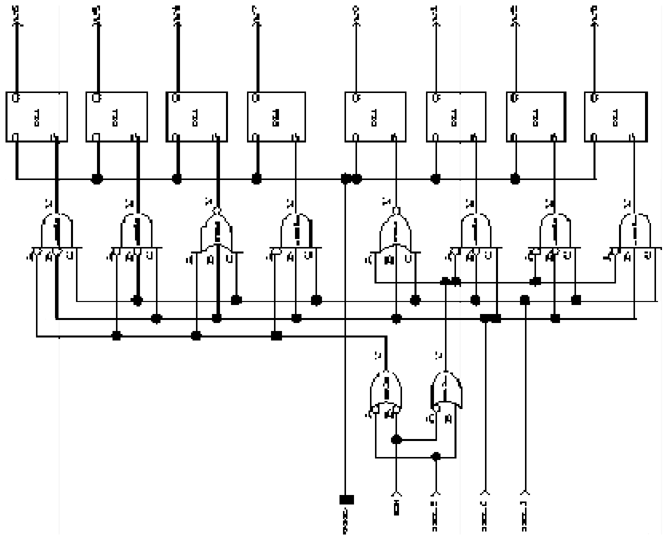
- **When using processes, a common error is to forget to assign an output a default value.**
  - ALL outputs should have DEFAULT values
- **If there is a logical path in the model such that an output is not assigned any value**
  - the synthesizer will assume that the output must retain its current value
  - a latch will be generated.
- **Example: In *dec3to8.vhd* do not assign *'y'* the default value of B"11111111"**
  - If *en* is 0, then 'y' will not be assigned a value
  - In the new synthesized logic, all '*y*' outputs are latched

…A Common Error in Process Statements…

```
entity dec3to8 is
    port (signal sel: in std_logic_vector(3 downto 0); signal en: in std_logic;
    signal y: out std_logic_vector(7 downto 0))
end dec3to8;
architecture behavior of dec3to8 is
begin
    process (sel, en)
    --    y <= "1111111";
          if (en = '1')  then
                 case sel is
                        when "000" => y(0) <= '0';    when "001" => y(1) <= '0';
                        when "010" => y(2) <= '0';    when "011" => y(3) <= '0';
                        when "100" => y(4) <= '0';    when "101" => y(5) <= '0';
                        when "110" => y(6) <= '0';    when "111" => y(7) <= '0';
                 end case;
          end if;
    end process;
end behavior;
```

No default value assigned to y!!

…A Common Error in Process Statements

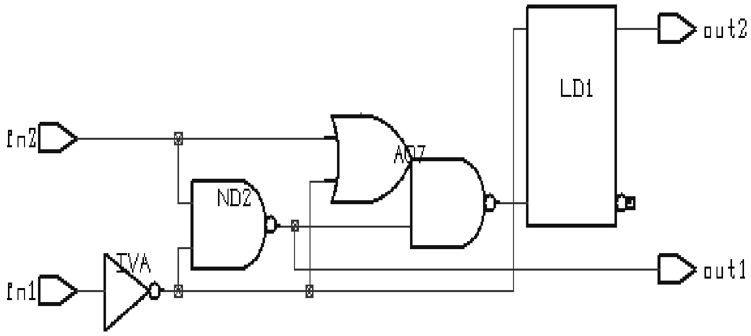


Another Incorrect Latch Insertion Example…

```
entity case_example is
    port (in1, in2 : in std_logic; out1, out2 : out std_logic);
end case_example;
architecture case_latch of case_example is
    signal b : std_logic_vector (1 downto 0);
begin
    process (b)
    begin
        case b is
            when "01" => out1 <= '0'; out2 <= '1';
            when "10" => out1 <= '1'; out2 <= '0';
            when others => out1 <= '1';
        end case;
    end process;
    b <= in1 & in2;
end case_latch;
```

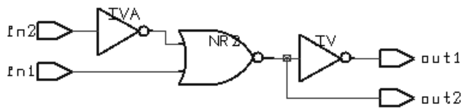
out2 has not been assigned a value for others condition!!

…Another Incorrect Latch Insertion Example



Avoiding Incorrect Latch Insertion

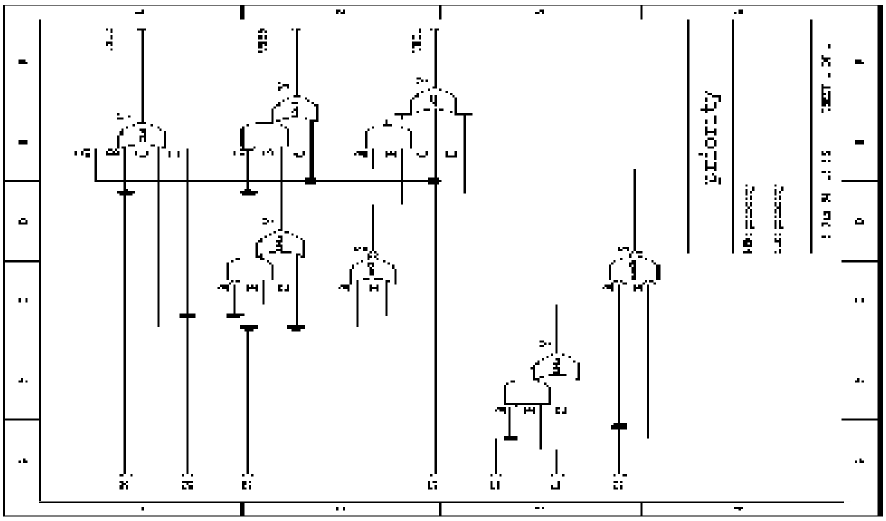
```
architecture case_nolatch of case_example is
    signal b : std_logic_vector (1 downto 0);
begin
    process (b)
    begin
        case b is
            when "01" => out1 <= '0'; out2 <= '1';
            when "10" => out1 <= '1'; out2 <= '0';
            when others => out1 <= '1'; out2 <= '0';
        end case;
    end process;
    b <= in1 & in2;
end case_nolatch;
```



Eight-Level Priority Encoder…

```
Entity priority is
   Port (Signal y1, y2, y3, y4, y5, y6, y7: in std_logic;
            Signal vec: out std_logic_vector(2 downto 0));
End priority;
Architecture behavior of priority is
Begin
   Process(y1, y2, y3, y4, y5, y6, y7)
   begin
       if (y7 = ‘1’)  then vec <= “111”;
       elsif (y6 = ‘1’)  then vec <= “110”;
       elsif (y5 = ‘1’)  then vec <= “101”;
       elsif (y4 = ‘1’)  then vec <= “100”;
       elsif (y3 = ‘1’)  then vec <= “011”;
       elsif (y2 = ‘1’)  then vec <= “010”;
       elsif (y1= ‘1’)  then vec <= “001”;
       else vec <= “000”;
       end if;
   end process;
End behavior;
```


…Eight-Level Priority Encoder…



Eight-Level Priority Encoder…

```
Architecture behavior2 of priority is
Begin
    Process(y1, y2, y3, y4, y5, y6, y7)
    begin
        vec <= “000”;
        if (y1 = ‘1’)  then vec <= “111”;  end if;
        if (y2 = ‘1’)  then vec <= “110”;  end if;
        if (y3 = ‘1’)  then vec <= “101”;  end if;
        if (y4 = ‘1’)  then vec <= “100”;  end if;
        if (y5 = ‘1’)  then vec <= “011”;  end  if;
        if (y6 = ‘1’)  then vec <= “010”;  end if;
        if (y7= ‘1’)  then vec <= “001”;   end if;
    end process;
End behavior2;
```

Equivalent 8-level priority encoder.

Ripple Carry Adder…

```
library ieee;
use ieee.std_logic_1164.all;
entity adder4 is
    port (Signal  a, b:  in std_logic_vector (3 downto 0);
            Signal cin : in std_logic_vector;
            Signal  sum:  out std_logic_vector (3 downto 0);
            Signal cout : in std_logic_vector);
end adder4;
architecture behavior  of adder4 is
Signal  c:  std_logic_vector (4 downto 0);
begin
```

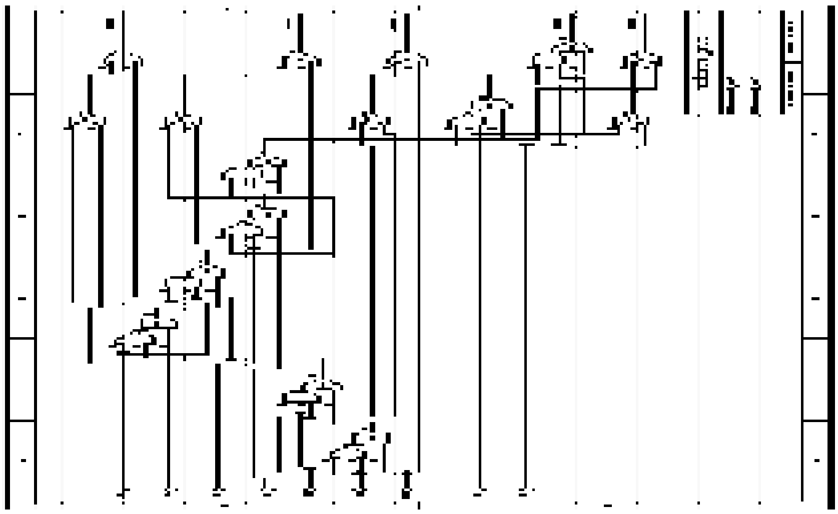
C is a temporary signal to hold the carries.

…Ripple Carry Adder…

```
    process (a, b, cin, c)
    begin
        c(0) <= cin;
        for I in 0 to 3 loop
            sum(I) <= a(I) xor b(I) xor c(I);
            c(I+1) <= (a(I) and b(I)) or (c(I) and (a(I) or b(I)));
        end loop;
    end process;
    cout <= c(4);
End behavior;
```

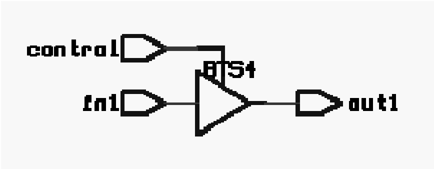
- *The Standard Logic 1164 package does not define arithmetic operators for the std_logic type.*
- *Most vendors supply some sort of arithmetic package for 1164 data types.*
- *Some vendors also support synthesis using the '+' operation between two std_logic signal types (Synopsis).*

…Ripple Carry Adder



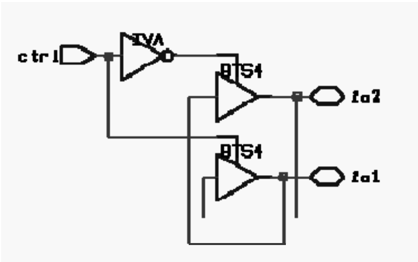
Tri-State Buffer Synthesis

```
library ieee;
use ieee.std_logic_1164.all;
entity tri_ex is
   port (in1, control : in std_logic; out1 : out std_logic);
end tri_ex;
architecture tri_ex_a of tri_ex is
begin
    out1 <= in1 when control = '1' else 'Z';
end tri_ex_a;
```



Bi-directional Buffer Synthesis

```
library ieee;
use ieee.std_logic_1164.all;
entity inout_ex is
   port (io1, io2 : inout std_logic; ctrl : in std_logic);
end inout_ex;
architecture inout_ex_a of inout_ex is
begin
   io1 <= io2 when ctrl = '1' else 'Z';
   io2 <= io1 when ctrl = '0' else 'Z';
end inout_ex_a;
```

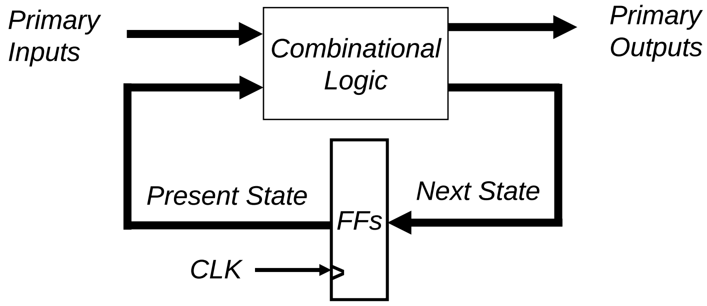


Outline

- **Sequential circuit synthesis**
  - Latch
  - Flip-flop with asynchronous reset
  - Flip-flop with synchronous reset
  - Loadable register
  - Shift register
  - Register with tri-state output
  - Finite state machine
- **Efficient coding styles for synthesis**

Sequential Circuits

- **Sequential circuits consist of both combinational logic and storage elements.**
- **Sequential circuits can be**
  - *Moore*-type: outputs are a combinatorial function of Present State signals.
  - *Mealy*-type: outputs are a combinatorial function of both Present State signals and primary inputs.



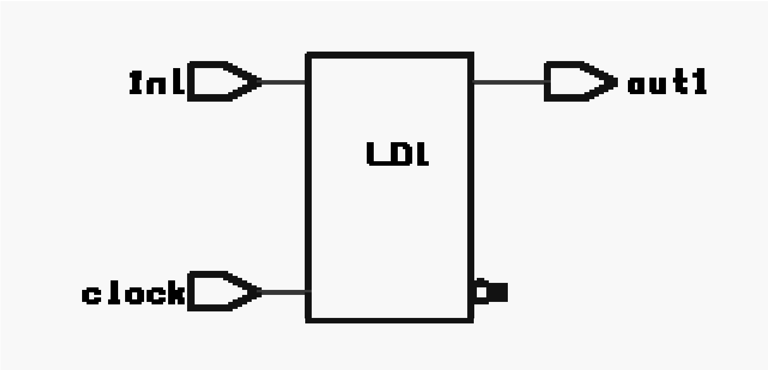
Template Model for a Sequential Circuit

```
entity model_name is
    port ( list of inputs and outputs );
end model_name;
architecture behavior of model_name is
    internal signal declarations
begin
    -- the state process defines the storage elements
    state: process ( sensitivity list -- clock, reset, next_state inputs)
    begin
        vhdl statements for state elements
    end process state;
    -- the comb process defines the combinational logic
    comb: process ( sensitivity list -- usually includes all inputs)
    begin
        vhdl statements which specify combinational logic
    end process comb;
end behavior;
```


Latch Synthesis…

```
library ieee;
use ieee.std_logic_1164.all;
entity latch_ex is
    port (clock, in1 : in std_logic; out1 : out std_logic);
end latch_ex;
architecture latch_ex_a of latch_ex is
begin
    process (clock)
    begin
        if (clock = '1') then
            out1 <= in1;
        end if;
    end process;
end latch_ex_a;
```

…Latch Synthesis

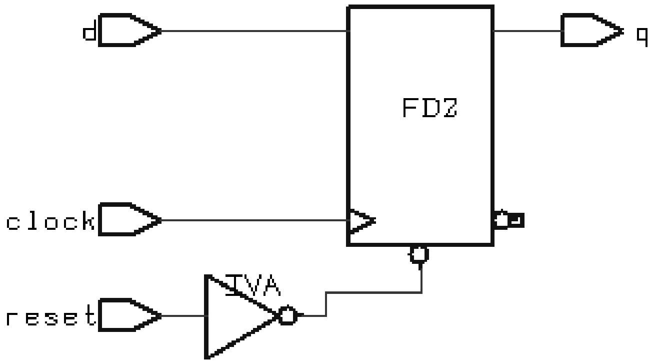


Flip-Flop Synthesis with Asynchronous Reset…

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_asyn is
    port( reset, clock, d: in std_logic; q: out std_logic);
end dff_asyn;
architecture dff_asyn_a of dff_asyn is
begin
    process
    begin
        if (reset = '1') then
                q <= '0';
        elsif clock = '1' and clock'event then
                q <= d;
        end if;
    end process;
end dff_asyn_a;
```

•*Note that the reset input has precedence over the clock in order to define the asynchronous operation.*

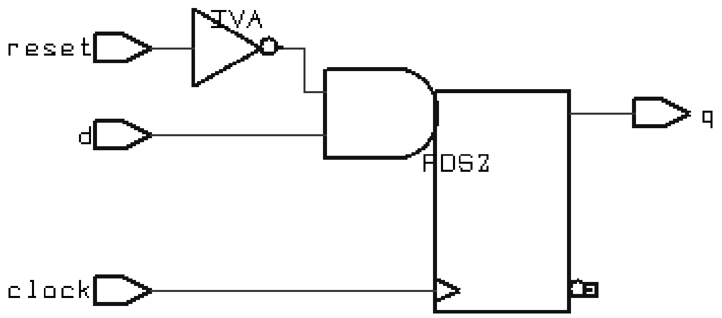
…Flip-Flop Synthesis with Asynchronous Reset



Flip-Flop Synthesis with Synchronous Reset…

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_syn is
    port( reset, clock, d: in std_logic; q: out std_logic);
end dff_syn;
architecture dff_syn_a of dff_syn is
begin
    process
    begin
        if clock = '1' and clock'event then
            if (reset = '1') then q <= '0';
            else q <= d;
            end if;
        end if;
    end process;
end dff_syn_a;
```

…Flip-Flop Synthesis with Synchronous Reset



8-bit Loadable Register with Asynchronous Clear…

```
library ieee;
use ieee.std_logic_1164.all;
entity reg8bit is
    port( reset, clock, load: in std_logic;
    signal din: in std_logic_vector(7 downto 0);
    signal dout: out std_logic_vector(7 downto 0));
end reg8bit;
architecture behavior of reg8bit is
    signal n_state, p_state: std_logic_vector(7 downto 0);
begin
    dout <= p_state;
    comb: process (p_state, load, din)
    begin
        n_state <= p_state;
        if (load = '1') then n_state <= din end if;
    end process comb;
```

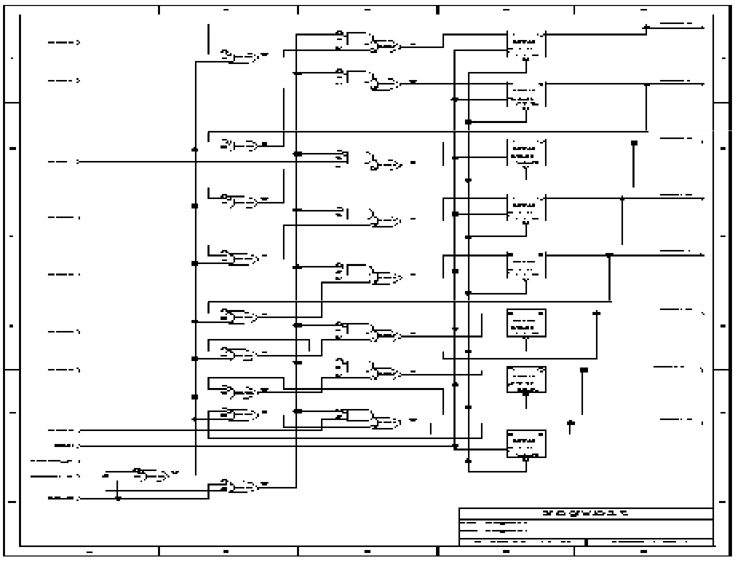
…8-bit Loadable Register with Asynchronous Clear…

```
state: process (clk, reset)
begin
    if (reset = '0') then   p_state   <= (others => '0');
    elsif (clock = '1' and clock'event) then
            p_state  <= n_state;
    end if;
end process state;
End behavior;
```

• The state process defines a storage element which is 8-bits wide, rising edge triggered, and had a low true asynchronous reset.

•Note that the reset input has precedence over the clock in order to define the asynchronous operation.

…8-bit Loadable Register with Asynchronous Clear



4-bit Shift Register…

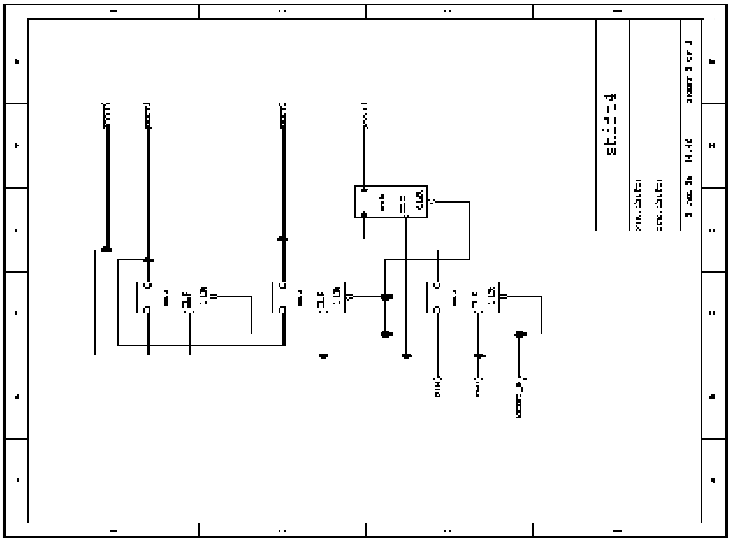
```
library ieee;
use ieee.std_logic_1164.all;
entity shift4 is
    port( reset, clock: in std_logic; signal din: in std_logic;
    signal dout: out std_logic_vector(3 downto 0));
end shift4;
architecture behavior of shift4 is
    signal n_state, p_state: std_logic_vector(3 downto 0);
begin
    dout <= p_state;
    state: process (clk, reset)
    begin
        if (reset = '0') then   p_state  <= (others => '0');
        elsif (clock = '1' and clock'event) then
                p_stateq <= n_state;
        end if;
    end process state;
```


…4-bit Shift Register…

```
        comb: process (p_state, din)
        begin
                n_state(0) <= din;
                for I in 3 downto 0 loop
                        n_state(I)  <= p_state(I-1);
                end loop;
        end process comb;
End behavior;
```

- *Serial input din is assigned to the D-input of the first D-FF.*
- *For loop is used to connect the output of previous flip-flop to the input of current flip-flop.*

…4-bit Shift Register



Register with Tri-State Output…

```
library ieee;
use ieee.std_logic_1164.all;
entity tsreg8bit is
    port( reset, clock, load, en: in std_logic;
    signal din: in std_logic_vector(7 downto 0);
    signal dout: out std_logic_vector(7 downto 0));
end tsreg8bit;
architecture behavior of tsreg8bit is
    signal n_state, p_state: std_logic_vector(7 downto 0);
begin
    dout <= p_state when (en=‘1’) else “ZZZZZZZZ”;
    comb: process (p_state, load, din)
    begin
        n_state <= p_state;
        if (load = '1') then n_state <= din end if;
    end process comb;
```

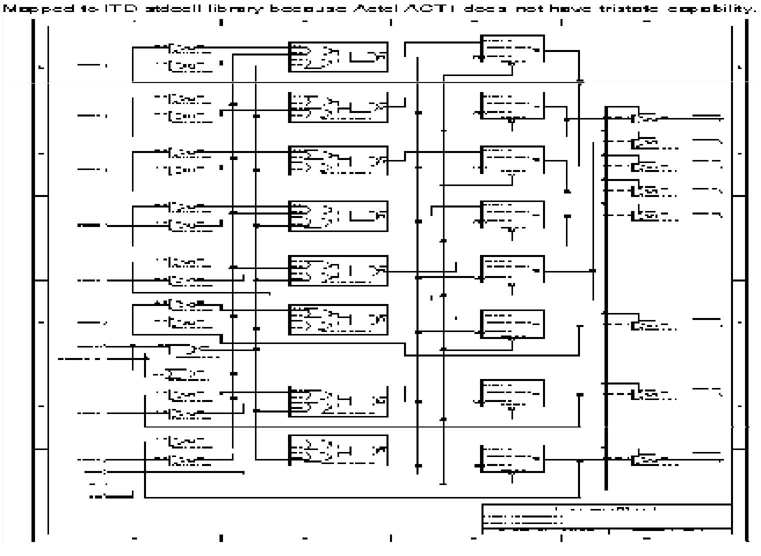
• *Z assignment used to specify tri-state capability.*

…Register with Tri-State Output…

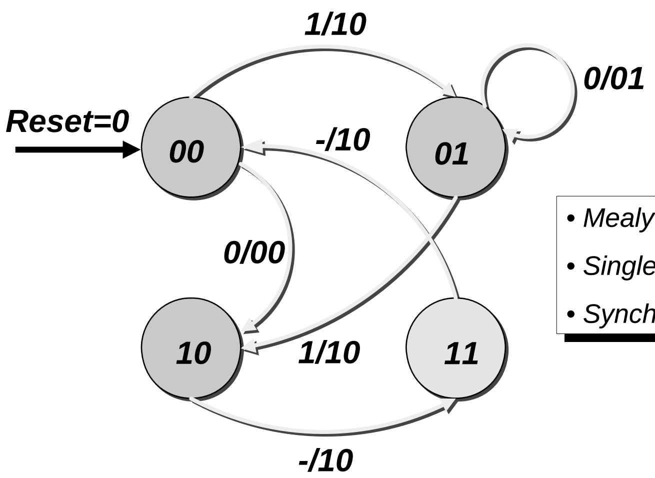
```
state: process (clk, reset)
       begin
              if (reset = ‘0') then   p_state   <= (others => '0‘);
              elsif (clock = '1' and clock'event) then
                     p_state  <= n_state;
              end if;
       end process state;
End behavior;
```


…Register with Tri-State Output

Mapped to ITD stdcell library because Actel ACT1 does not have tristate capability.



Finite State Machine Synthesis…



- *Mealy model*
- *Single input, two outputs*
- *Synchronous reset*

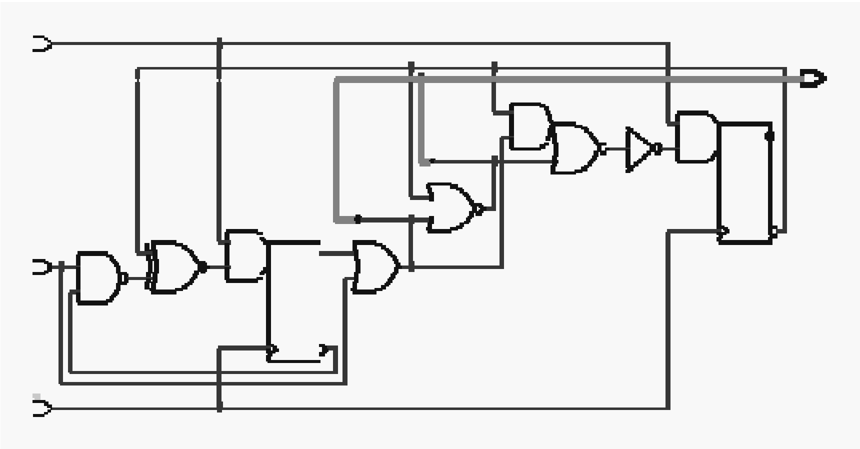
…Finite State Machine Synthesis…

```
library ieee;
use ieee.std_logic_1164.all;
entity state_ex is
     port (in1, clock, reset : in std_logic; out1 :
            out std_logic_vector (1 downto 0));
end state_ex;
architecture state_ex_a of state_ex is
    signal cur_state, next_state : std_logic_vector (1 downto 0);
begin
    process (clock, reset)
     begin
            if clock = '1' and clock'event then
                       if reset = '0' then cur_state <= "00";
                      else cur_state <= next_state;
                      end if;
           end if;
    end process;
```

…Finite State Machine Synthesis…

```
    process (in1, cur_state)
    begin
        case cur_state is
            when "00" => if in1 = '0' then next_state <= "10"; out1 <= "00";
                         else next_state <= "01"; out1 <= "10";
                         end if;
            when "01" => if in1 = '0' then next_state <= cur_state;
                            out1 <= "01";
                         else next_state <= "10“; out1 <= "10";
                         end if;
            when "10" => next_state <= "11"; out1 <= "10";
            when "11" => next_state <= "00"; out1 <= "10";
            when others => null;
        end case;
    end process;
end state_ex_a;
```


…Finite State Machine Synthesis



Outline

- **Sequential circuit synthesis**
  - Latch
  - Flip-flop with asynchronous reset
  - Flip-flop with synchronous reset
  - Loadable register
  - Shift register
  - Register with tri-state output
  - Finite state machine
- **Efficient coding styles for synthesis**

Key Synthesis Facts

- **Synthesis ignores the after clause in signal assignment**
  - C <= A AND B after 10ns
  - May cause mismatch between pre-synthesis and post-synthesis simulation if a non-zero value used
  - The preferred coding style is to write signal assignments without the after clause.
- **If the process has a static sensitivity list, it is ignored by the synthesis tool.**
- **Sensitivity list must contain all read signals**
  - Synthesis tool will generate a warning if this condition is not satisfied
  - Results in mismatch between pre-synthesis and post-synthesis simulation

Synthesis Static Sensitivity Rule

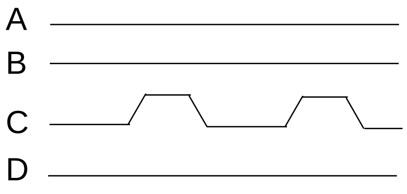
Original VHDL Code

```
Process(A, B)
Begin
    D <= (A AND B) OR C;
End process;
```

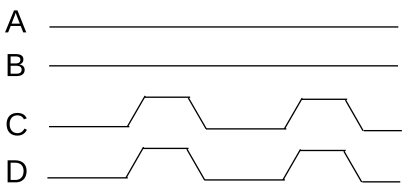
Synthesis View of Original VHDL Code

```
Process(A, B, C)
Begin
    D <= (A AND B) OR C;
End process;
```

Pre-Synthesis Simulation



Post-Synthesis Simulation



Impact of Coding Style on Synthesis Execution Time

Inefficient Synthesis Execution Time

```
Process(Sel, A, B, C, D)
Begin
    if Sel = “00 then  Out <= A;
    elsif Sel = “01” then Out<=B;
    elsif Sel = “10” then Out<=C;
    else  Out<=D;
    endif;
End process;
```

Efficient Synthesis Execution Time

```
Process(Sel, A, B, C, D)
Begin
    case Sel is
        when “00 =>  Out <= A;
        when “01” Out<=B;
        when “10” Out<=C;
        when “11”  Out<=D;
    end case;
End process;
```

- *Synthesis tool is capable of deducing that the if …elsif conditions are mutually exclusive but precious CPU time is required.*
- *In case statement, when conditions are mutually exclusive.*

Synthesis Efficiency Via Vector Operations

Inefficient Synthesis Execution Time

```
Process(Scalar_A, Vector_B)
Begin
    for k in Vector_B`Range loop
        Vector_C(k) <=Vector_B(k) and
            Scalar_A;
    end loop;
End process;
```

Efficient Synthesis Execution Time

```
Process(Scalar_A, Vector_B)
    variable Temp:
    std_logic_vector(Vector_B`Range);
Begin
    Temp := (others => Scalar_A);
    Vector_C <=Vector_B and Temp;
End process;
```

• Loop will be unrolled and analyzed by the synthesis tool.

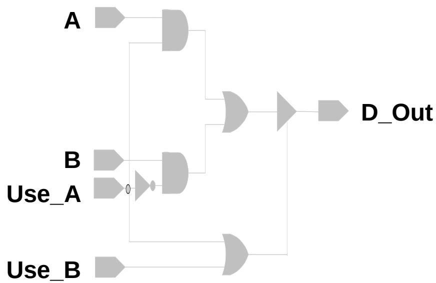
• Vector operation is understood by synthesis and will be efficiently synthesized.

Three-State Synthesis

- **A three-state driver signal must be declared as an object of type std_logic.**
- **Assignment of 'Z' infers the usage of three-state drivers.**
- **The std_logic_1164 resolution function, resolved, is synthesized into a three-state driver.**
- **Synthesis does not check for or resolve possible data collisions on a synthesized three-state bus**
  - It is the designer responsibility
- **Only one three-state driver is synthesized per signal per process.**

Example of the Three-State / Signal / Process Rule

```
Process(B, Use_B, A, Use_A)
Begin
    D_Out <= ‘Z’;
    if Use_B = ‘1’ then
        D_Out <= B;
    end if;
    if Use_A = ‘1’ then
        D_Out <= A;
    end if;
End process;
```



•Last scheduled assignment has priority

Latch Inference & Synthesis Rules…

- **A latch is inferred to satisfy the VHDL fact that signals and process declared variables maintain their values until assigned new ones.**
- **Latches are synthesized from if statements if all the following conditions are satisfied**
  - Conditional expressions are not completely specified
    - An else clause is omitted
  - Objects conditionally assigned in an if statement are not assigned a value before entering this if statement
  - The VHDL attribute `EVENT is not present in the conditional if expression
- **If latches are not desired, then a value must be assigned to the target object under all conditions of an if statement (without the `EVENT attribute).**

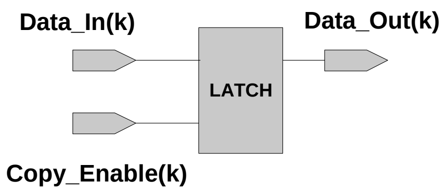
…Latch Inference & Synthesis Rules

- **For a case statement, latches are synthesized when it satisfies all of the following conditions:**
  - An expression is not assigned to a VHDL object in every branch of a case statement
  - VHDL objects assigned an expression in any case branch are not assigned a value before the case statement is entered.
- **Latches are synthesized whenever a for…loop statement satisfies all of the following conditions**
  - for…loop contains a next statement
  - Objects assigned inside the for…loop are not assigned a value before entering the enclosing for…loop

For…Loop Statement Latch Example

```
Process(Data_In, Copy_Enable)
Begin
    for k in 7 downto 0 loop
          next when Copy_Enable(k)='0'
       Data_Out(k) <= Data_in(k);
    end loop;
End process;
```

Seven latches will be synthesized



Flip-Flop Inference & Synthesis Rules…

- **Flip-flops are inferred by either**
  - Wait until….
    - Wait on… is not supported by synthesis
    - Wait for… is not supported by synthesis
  - If statement containing `EVENT
- **Synthesis accepts any of the following functionally equivalent statements for inferring a FF**
  - Wait until Clock='1'; (most efficient for simulation)
  - Wait until Clock`Event and Clock='1';
  - Wait until (not Clock`Stable) and Clock='1';

…Flip-Flop Inference & Synthesis Rules

- **Synthesis does not support the following Asynchronous description of set and reset signals**
  - Wait until (clock='1') or (Reset='1')
  - Wait on Clock, Reset
- **When using a synthesizable wait statement only synchronous set and reset can be used.**
- **If statement containing the VHDL attribute `EVENT cannot have an else or an elsif clause.**

Alternative Coding Styles for Synchronous FSMs

- **One process only**
  - Handles both state transitions and outputs
- **Two processes**
  - A synchronous process for updating the state register
  - A combinational process for conditionally deriving the next machine state and updating the outputs
- **Three processes**
  - A synchronous process for updating the state register
  - A combinational process for conditionally deriving the next machine state
  - A combinational process for conditionally deriving the outputs