

## **Lecture 7**

# **Introduction to VHDL: Processes (cont.)**

Khaza Anuarul Hoque  
ECE 4250/7250

# VHDL Processes

- A process with a sensitivity clause must not contain an explicit wait statement.
- Only static signal names for which reading is permitted may appear in the sensitivity list of a process statement.
- The execution of a process statement consists of the repetitive execution of its sequence of statements.

# VHDL Processes

- Simulator runs a process when any one of the signals in the sensitivity list changes.
- Process should either have a “sensitivity list” or a “wait” statement at the end.
- Only static signal names are allowed in the sensitivity list.

```
process (A,B)
begin
    if (A='1' or B='1') then
        Z <= '1';
    else
        Z <= '0';
    end if;
end process;
```

Suspends  
at bottom

```
process
begin
    if (A='1' or B='1') then
        Z <= '1';
    else
        Z <= '0';
    end if;
    wait on A, B;
end process;
```

Suspends  
at "wait"

# If statement


Syntax:

```
if condition1 then
    { sequential_statement }
elsif condition2 then
    { sequential_statement }
else
    { sequential_statement }
end if;
```

- “If” Statement evaluates each condition in order.
- Statements can be nested.

# If statement

```
process {A, B, C, X}
begin
    if {X = "0000"} then
        Z <= A;
    elsif {X <= "0101"} then
        Z <= B;
    else
        Z <= C;
    end if;
end process;
```

Always executes if  $x = "0000"$  

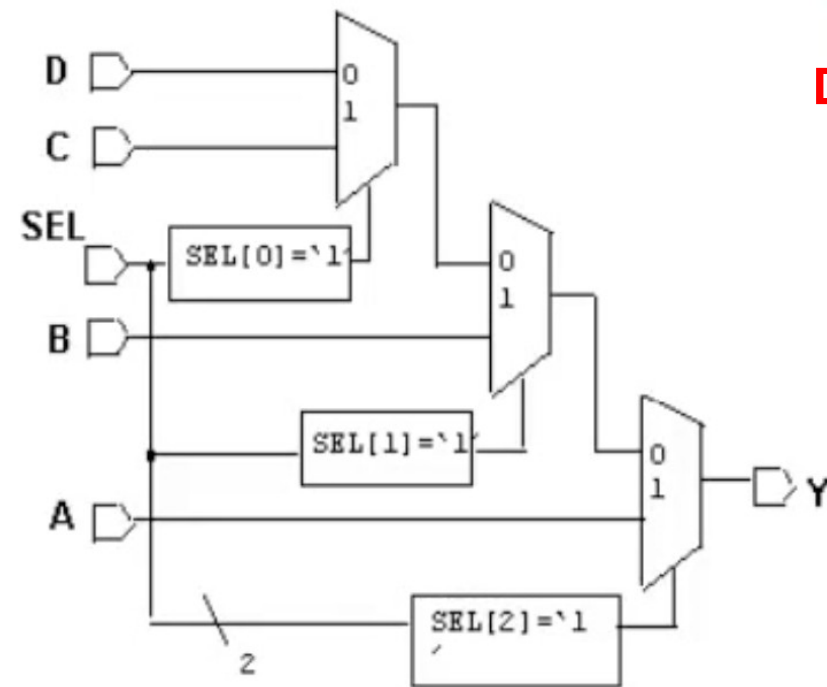
- Avoid using more than three levels Of **If...else** statements .
- When defining the condition, use parentheses to differentiate levels of operations on the condition.

# If statement – Implied Design

```

process (sel, a, b, c, d)
begin
  If sel(2) = '1' then
    y <= a;
  elsif sel(1) = '1' then
    y <= b;
  elsif sel(0) = '1' then
    y <= c;
  else
    y <= d;
  end if;
end process

```



- Generates a priority structure.
- Corresponds to “when-else” command in the concurrent part.

# Case statement

Syntax:

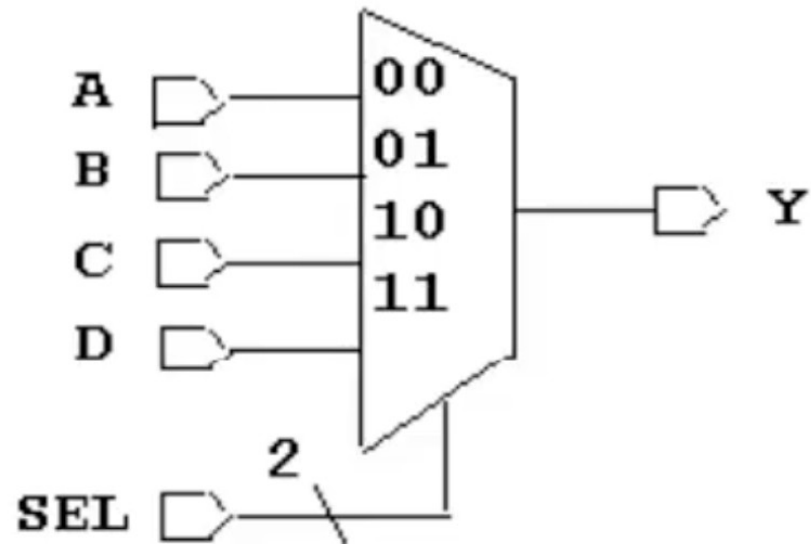
```
case expression is
    when choice1 => { statements }
    when choice2 => { statements }
    when others => { statements }
end case;
```

- “Case “ Statement is a series of parallel checks to check a condition.
- It selects, for execution one of a number of alternative sequences of statements.
- Statements following each “when” clause is evaluated, only if the choice value matches the expression value.



# Case statement

```
process (sel,a,b,c,d)
begin
  case sel is
    when 0=> y <=a;
    when 1=> y <=b;
    when 2=> y <=c;
    when others =>
      y<=d;
  end case;
end process;
```



- Does not result in prioritized logic structure unlike the if statement.
- Corresponds to “with...select” in concurrent statements.






# Case statement (rules)

```

process (A, B, C, X)
begin
  case X is
    when 0 to 4 =>
      Z <= B;
    when 5 =>
      Z <= C;
    when 7 | 9 =>
      Z <= A;
    when others =>
      Z <= 0;
  end case;
end process;

```

 range  
 list  
 others

- Every **possible** value of the case expression must be covered in one and only one *when* clause.
- Each choice can be either a static expression ( such as 3 ) or a static range ( such as 1 to 3 ). we cannot have a “when” condition that changes when it is being evaluated.

# Invalid case statements

```
signal VALUE: INTEGER range 0 to 15;
```

```
signal OUT_1: BIT;
```

EX1 : case VALUE is

end case

-- Must have at least one *when* clause

EX3: case VALUE is

when 0 to 10 =>

OUT\_1 <= '1';

when 5 to 15 =>

OUT\_1 <= '0';

end case

-- Choices 5 to 10 overlap

EX2 : case VALUE is

when 0 =>

OUT\_1 <= '1';

when 1 =>

OUT\_1 <= '0';

end case

-- Values 2 to 15 are not covered  
by choices

# Null statement

- Does not perform any action
- Can be used to indicate that when some conditions are met no action is to be performed
- Example:

```
case a is
  when "00" => q1 <= '1';
  when "01" => q2 <= '1';
  when "10" => q3 <= '1';
  when others <= null; -----Why?
end case;
```

# Processes again

- Two types of processes:

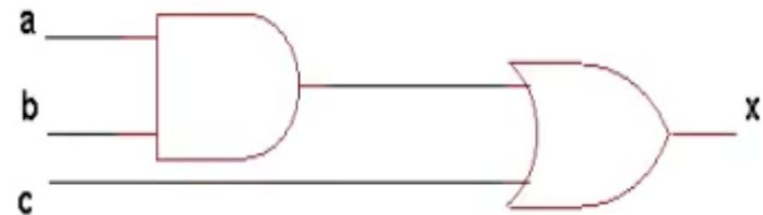
- Combinatorial
- Clocked



- Combinatorial Process

- Generates combinational logic
- All inputs must be present in the sensitivity list.

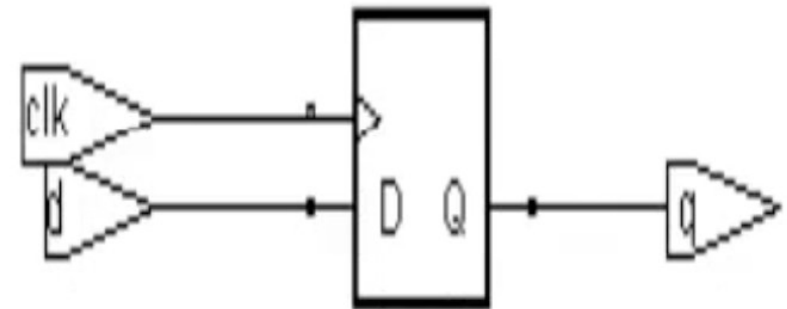
```
process (a,b,c)
begin
  x <= (a and b) or c;
end process;
```



# Clocked Processes

- Clocked Process:
  - Generates synchronous logic.

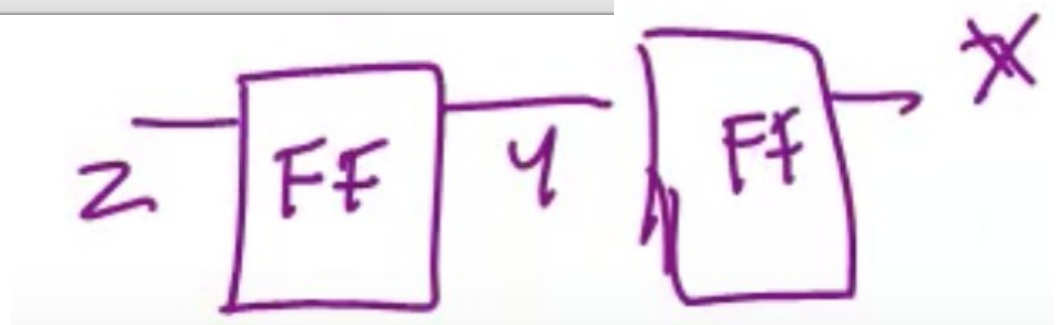
```
process (clk)
begin
  if (clk' event and clk = '1' ) then
    Q <= D;
  end if;
end process;
```



- Any signal assigned under a clk'event generates a Flip-flop.

# Clocked Processes

```
if (clk'event ...) then  
    x ≤ y;  
    y ≤ z;  
end if;
```



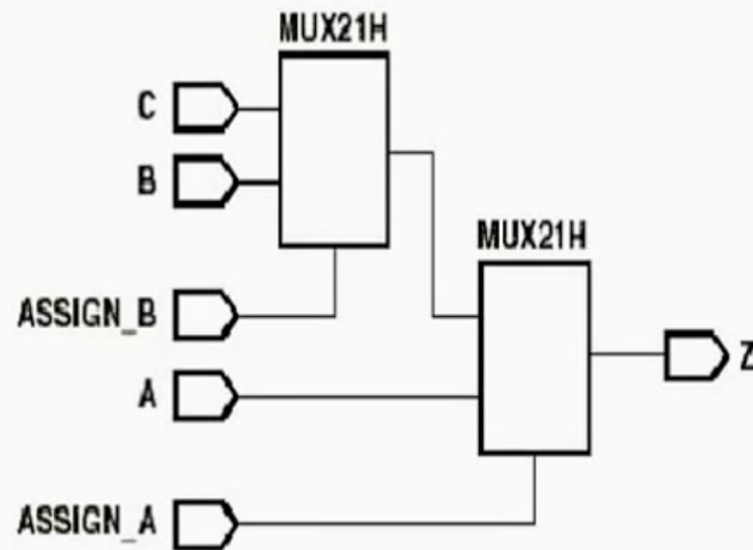
# Clocked Processes

if (clk'event ....) then  
 {     x ≤ y;     y -> [FF] -> x  
      m ≤ z;  
 end if;  
               z -> [FF] -> m



# When statement (concurrent part)

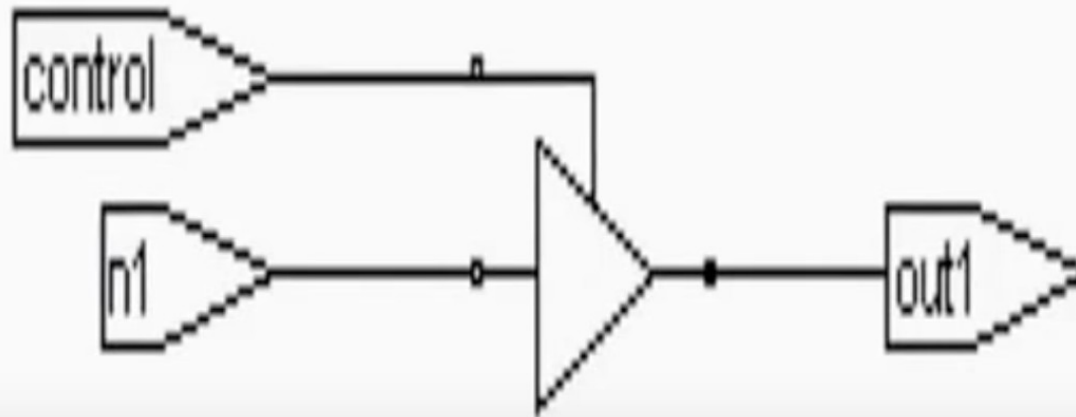
```
Z <= A when ASSIGN_A = '1' else  
      B when ASSIGN_B = '1' else  
      C;
```



# Tri-state buffer

## ■ Modeling Tri-state buffer

```
architecture tri_ex_a of tri_ex is
begin
    out1 <= in1 when control = '1' else
        'Z';
end tri_ex_a;
```



# With...select statement

- Syntax

```
with choice_expression select  
target <= expression 1 when choice 1  
           expression 2 when choice 2  
           expression N when choice N ;
```
- 'with...select' statement evaluates choice\_expression and compares that value to each choice value.
- In 'when' statement the matching choice value has its expression assigned to target.
- Each value in the range of the choice\_expression type must be covered by one choice.

# 4:1 MUX

```

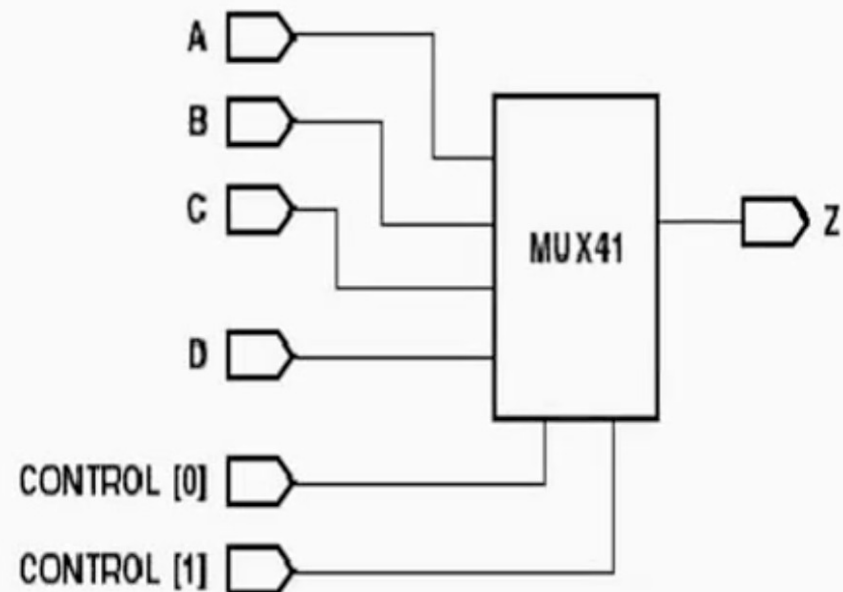
signal A, B, C, D, Z: std_logic;

signal CONTROL_0, CONTROL_1:
  std_logic;

with CONTROL select
  Z <= A when "00",
    B when "01",
    C when "10",
    D when "11",
    '0' when others;

  
```

Why "when others" clause?



- No two choices can overlap.
- All possible choices must be enumerated.
- Each choice can be either a static expression (such as 3) or a static range (such as 1 to 3).