



# Revisiting Important VHDL Concepts

# Announcements

- Homework #5 due Wednesday
- Quiz Wednesday on VHDL arithmetic
  - If you missed last Wednesday, read the notes and ask questions
- Group homework Evals due 9/6
  - Please fill out one for each group member.
  - If all group members agree, you may remain in your current groups.

# From the top

- What VHDL is
  - A hardware description language used to describe digital hardware
  - It can also be used for \_\_\_\_\_?
- What VHDL is not
  - It is not a programming language like C, Java or Python

# Major difference from programming language

- Programming languages are sequential
- VHDL is \_\_\_\_\_?
  - What does \_\_\_\_\_ mean?
  - How does that affect the order of statements in VHDL?
  - When is VHDL sequential?

# Sequential Statements

- IF and CASE statements need to go in a \_\_\_\_\_?
- The statement gets executed when a signal in the \_\_\_\_\_ changes.
  - Which signals go in this list?
    - Any signal on the right hand side of an assignment
    - Any signal being evaluated in the IF or CASE
  - Which signals do not go in this list?
    - Output signals (on the left hand side of an assignment statement)

# A Common Mistake

- A common problem is to not include a needed signal on a sensitivity list:

```
process (A, S)
begin
  if ( S = '1' ) then
    Y <= A;
  else
    Y <= B;
  end process;
```

- This is implementing a 2 to 1 mux. Signal B has been left off the sensitivity list by mistake.
- if  $S = 0$  already and a change occurs on B, this change will not be propagated to the Y output!
- This can be hard to debug – be careful with sensitivity lists!

# Signal assignments in Processes

- When are outputs assigned in the process?
- Is this allowed? Why or why not?

```
ARCHITECTURE xyz OF xor_2 IS
```

```
BEGIN
```

```
    PROCESS (a, b)
```

```
    BEGIN
```

```
        c <= '0';
```

```
        IF a /= b then
```

```
            c <= '1';
```

```
        END IF;
```

```
    END PROCESS;
```

```
END xyz;
```

# Why order matters

```
comb: process(nickel_in, dime_in, quarter_in, money, dispense_in, coin_return,  
current_state)
```

```
begin
```

```
  case current_state is
```

```
    when wait1 =>
```

```
      if nickel_in = '1' then
```

```
        next_state <= nickel;
```

```
      elsif dime_in = '1' then
```

```
        next_state <= dime;
```

```
      elsif quarter_in = '1' then
```

```
        next_state <= quarter;
```

```
      elsif coin_return = '1' then
```

```
        next_state <= change;
```

```
      elsif money > 75 then
```

```
        next_state <= enough;
```

```
      else next_state <= wait1;
```

```
    end if;
```

This is a real example. What happens if money is greater than 75 but the user is trying to insert a nickel?

# Rules to live by

- One output per process
  - This is a standard adopted by the ECTET department.
  - Processes are free – use as many as you want
  - The number of processes does not affect the final implementation. However ....
    - Less chance of unintentionally creating a latch
    - Higher probability that that process can be re-used in a future design

# Rules to live by

- If you assign a signal in one branch of a case or an if you must assign it in all branches.
  - If not, latch is created
  - Latches are not good
    - Can lead to instability
    - Can lead to undesired functionality
  - Check you synthesis WARNINGS. If it indicates a latch, you must fix it.

# Bad Example

- What is wrong with the following?

```
ARCHITECTURE behavioral OF fulladd IS
    SIGNAL inputs : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
    inputs <= X & Y & Cin;
bad: PROCESS (inputs) IS
    BEGIN
        CASE inputs IS
            WHEN "001" | "010" | "100" | "111" =>
                S <= '1';
            WHEN "011" | "101" | "110" | "111" =>
                Cout <= '1';
            WHEN OTHERS =>
                S <= '0';
                Cout <= '0';
        END CASE;
    END PROCESS;
END behavioral;
```

# Good Example

- Longer code does not necessarily mean more circuitry.

```
ARCHITECTURE behavioral OF fulladd IS
  SIGNAL inputs : STD_LOGIC_VECTOR(2 DOWNT0 0);
BEGIN
  inputs <= X & Y & Cin;
  Ess: PROCESS (inputs) IS
  BEGIN
    CASE inputs IS
      WHEN "001" | "010" | "100" | "111" =>
        S <= '1';
      WHEN OTHERS =>
        S <= '0';
    END CASE;
  END PROCESS;
  Carryout: PROCESS (inputs) IS
  BEGIN
    CASE inputs IS
      WHEN "011" | "101" | "110" | "111" =>
        Cout <= '1';
      WHEN OTHERS =>
        Cout <= '0';
    END CASE;
  END PROCESS;
END behavioral;
```

# Re-Use

- Think of each process as a functional block
  - If it only has one output, there is more of a chance you can use it again in another project
  - If it has two unrelated outputs, it becomes specific to the current application

# Output Ports

- Output ports represent a pin
  - They have no memory
  - They cannot be read

# Bad Example

- This will generate the following error:

```
e_detector_bad.vhd
```

```
interface object "First" of mode out cannot be read. Change object mode to buffer.  
or, 0 warnings  
ings
```



```
library IEEE;  
use IEEE.std_logic_1164.all;  
  
]Entity Name_Detector IS  
]   Port( b6, a, b, c, d, e       :IN STD_LOGIC;  
-   First                        :OUT STD_LOGIC);  
-   END Name_Detector;  
-   -----  
]Architecture Mix of Name_Detector IS  
]   Begin  
-   First <= NOT(a OR b OR d) AND (c AND e);  
-   First <= First AND b6;  
-   END Mix;
```

- What should you do? (hint do not change mode to buffer)

# Good Example

- Best solution is to work with internal signals and then assign internal signals to the outputs

```
library IEEE;
use IEEE.std_logic_1164.all;

Entity Name_Detector IS
    Port( b6, a, b, c, d, e      :IN STD_LOGIC;
          First                 :OUT STD_LOGIC);
END Name_Detector;
-----
Architecture Mix of Name_Detector IS
    Signal int_first : STD_LOGIC;
    Begin
        int_first <= NOT(a OR b OR d) AND (c AND e);
        First <= int_first AND b6;
    END Mix;
```

# Concatenation and reverse

- Concatenation combines individual signals into a vector (or bus)

```
SIGNAL buss : STD_LOGIC_VECTOR(5 downto 0);
```

```
:
```

```
Buss <= a & b & c & d & e & f;
```

- What if you want to reverse concatenate?
  - Say you have a vector internally but you have individual output ports

```
a <= buss(5);
```

```
b <= buss(4);
```

```
c <= buss(3);
```

```
d <= buss(2);
```

```
Etc;
```

# Synthesis

- What is synthesis?
- Can all VHDL be synthesized?
  - If not, what is the exception?
- If your code compiles and synthesizes does that mean it is correct and you are done?

# Which leads us to ....

- Simulation
  - Used to verify the operation of the circuit
  - Inputs can be in the form of:
    - Waveforms – cumbersome with large circuits
    - VHDL testbench – preferred method
  - Outputs can be in the form of:
    - Waveforms – inspect visually or with self checking testbench
    - File – we will not cover this

# What is important now

- Remember that the testbench code is separate from your design code (you will have 2 .vhd files)
- The module you are testing is brought into the testbench as a component.
  - Referred to as the UUT or DUT – Unit Under Test or Device Under Test.
- The component declaration must match your design entity exactly
- The UUT name must match the component name
- The port map signals must match the component ports exactly

# Example:

Just read the comments



```
ARCHITECTURE test OF phone_number_displayer_tb IS
```

```
-- Component Declaration for the Unit Under Test (UUT)
```

```
COMPONENT phone_number_displayer      --the component name must match your lab4 entity name
```

```
  PORT (
```

```
    x          : IN  std_logic;  --these port names must match your
```

```
    y          : IN  std_logic;  --entity port names exactly
```

```
    z          : IN  std_logic;
```

```
    w          : IN  std_logic;
```

```
    -----
```

```
    output_case : OUT std_logic_vector(6 downto 0);
```

```
    output_if   : OUT std_logic_vector(6 downto 0)
```

```
  );
```

```
END COMPONENT;
```

```
--Inputs
```

```
SIGNAL inputs : std_logic_vector(3 DOWNT0 0);      --this vector will drive the inputs
```

```
--Outputs
```

```
SIGNAL output_case_tb : std_logic_vector(6 downto 0);
```

```
SIGNAL output_if_tb   : std_logic_vector(6 downto 0);--these names can be anything
```

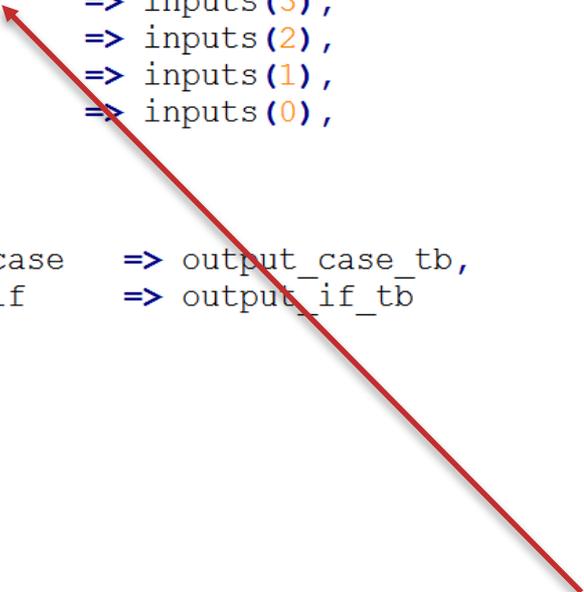
# Example (con't)

**BEGIN**

```
-- Instantiate the Unit Under Test (UUT)
-- on left of => are the names of your component's ports above
-- on the right of => are the signals that they map to
ut : phone_number_displayer PORT MAP (
  x           => inputs(3),
  y           => inputs(2),
  z           => inputs(1),
  w           => inputs(0),

  --

  output_case => output_case_tb,
  output_if   => output_if_tb
);
```



Change this name to match your component too

# Testbench code vs. design code

- Process triggering
  - A process can be triggered by resumption after a wait statement or by an event on a signal in its sensitivity list:

```
process (a, b, s)
  begin
  ....
end process;
```

Sensitivity list for process – process executed when an event occurs on any signal in this list

```
process
  begin
  ....
  Wait for 10 ns;
  ....
end process;
```

Process with no sensitivity list will always be triggered initially at time 0.

Suspend for 10 ns

# Rules for Processes

- If a process has a sensitivity list, then it cannot contain a 'wait' statement.
- If a process without a sensitivity list 'falls out the bottom' then it immediately loops back to the top until it hits a wait statement.
  - How do we prevent this in a testbench?



**Questions??**