#### **Process Blocks**

The last of the 'concurrent\_stmts' that we will consider is the **process** block

architecture arch\_name of entity\_name is

declarations

begin

concurrent\_stmt;

concurrent\_stmt;

end arch\_name;

The entire contents of a **process** block operate *in parallel* with other *concurrent\_stmt* 

The **process** block itself is a concurrent statement and should be thought of as a **subcircuit enclosed inside a black box** 

You will 'read elsewhere' that the **process** block is a container for a set of *sequential statements*, which 'executes' from top to bottom

And you will immediately be tempted to cut-and-paste your C code into one

I give you fair warning, there is nothing *sequential* about a **process** block nor about it *executing* and any attempt to treat it as a 'container' for C code will fail hard!

#### **Process Blocks**

The **process** block provides semantics that allow you to describe a circuit in a *recipe-type* fashion

VHDL synthesis tools obey special semantics when interpreting statements within a **process** block

The majority of your descriptions of combinational logic and ALL of your descriptions of storage elements (DFFs) will be done inside a **process** block

VHDL allows a lot of different types of constructs to show up in a **process** block

We will restrict the components that go into a **process** block to three components, NO exceptions!

- signal assignment
- *if stmts*
- case stmts

There are also two forms accepted by VHDL for a **process** block We will ONLY use the version with the *sensitivity list* (NO **wait** statements)

#### **General Form of the Process Block**

Our restricted form of the **process** block is given as follows

```
process(sensitivity_list)
```

begin

statement;

end process;

The *sensitivity\_list* lists the **input** signals to the sub-circuit you are describing

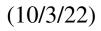
The following shows a simple example of a **process** block using *signal assignment* 

```
signal a, b, c, y: std_logic;
process(a, b, c)
    begin
    y <= a and b and c;</pre>
```

end process;

Although valid, you should consider *signal assignment* without the **process** block as an alternative, equivalent description of this 3-input AND gate

```
y <= a and b and c;
```



#### **Golden Rules of Process Blocks**

Before discussing the *if stmt* and the *case stmt*, let's cover my **golden rules** 

#### NEVER VIOLATE THESE RULES

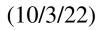
• Rule 1: All signals that are read with a process block must be included in the sensitivity list

This includes signals that appear on the right side of *assignment* statements AND those that appear within Boolean expressions in *if* and *case* statements

• Rule 2: The last assignment to an output signal takes precedence over ALL previous assignments (listed earlier in the **process** block) output\_signal <= value\_expression1;

```
...
if ( x = '1' ) then
    output_signal <= value_expression2;
end if;</pre>
```

If (x = '1') evaluates to be true, then *output\_signal* is assigned *value\_expression2* 



### **Golden Rules of Process Blocks**

NEVER VIOLATE THESE RULES

- Rule 3: All output signals MUST have an UNCONDITIONAL assignment You MUST include unconditional assignments to ALL output signals as the topmost set of assignment statements in your **process** block
- Rule 4: Never use a signal on both sides of an assignment statement (inside or outside of a **process** block)

tmp <= tmp or b;</pre>

As was true of signal assignment outside the **process** block, this results in a combinational loop with the output connected to one of the inputs

• Rule 5: Never assign to an output signal both *inside* and *outside* of a **process** block If you use a *simple signal, selected signal,* or *conditional signal assignment* to assign a value to a signal, do NOT assign to it within a **process** block

Turns out that multiple assignments to an output signal within a **process** block is allowed and is mandatory according to Rule 3 above!

#### Case Stmts Within a Process Block

The *case stmt* is equivalent to the *selected signal assignment* discussed earlier but is more general (similar in appearance to programming languages)

```
case case_expression is
```

```
when choice_1 =>
```

statements;

• • •

when others =>

statements;

#### end case;

As was true for *selected signal assignment*, *choice\_x* terms must be **mutually exclusive** and **all inclusive** 

```
x <= z;
case s is -- Creates a MUX structure in the
when "00" => -- hardware, similar to selected
x <= a; -- signal assignment
when "01" =>
```

#### If Stmts Within a Process Block

The *if stmt* is equivalent to the *conditional signal assignment* discussed earlier but is more general (also similar in appearance to programming languages)

if boolean\_expr\_1 then

statements;

elsif boolean\_expr\_2 then

statements;

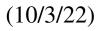
• • •

#### else

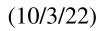
statements;

end if;

```
You can use the if stmt to describe the 4-to-2 priority encoder discussed earlier
entity prio_encoder42 is
port (
    r: in std_logic_vector(3 downto 0);
    code: out std_logic_vector(1 downto 0);
    active: out std_logic);
end prio_encoder42;
```



```
If Stmts Within a Process Block
    architecture if_arch of prio_encoder42 is
        begin
        process(r)
           begin
           code <= "00";
           if ( r(3)='1' ) then
              code <= "11";
           elsif (r(2)='1') then
              code <= "10";
           elsif (r(1)='1') then
              code <= "01";
           else
              code <= "00";
           end if;
        end process;
        active \leq r(3) or r(2) or r(1) or r(0);
    end if_arch;
```



#### If Stmts Within a Process Block

*if stmts* can be nested, as shown here when finding the max of *a*, *b* and *c* **process**(a, b, c) -- NOTE: if stmts create a priority **begin** -- network, i.e., a layered sequence max <= a; -- of MUXes as shown earlier</pre> if (a > b) then if (a > c) then max <= a;else max <= c;end if; else if (b > c) then max <= b;</pre> else max <= c;end if; end if; end process;

**Consequences of Violating the Golden Rules of Process Blocks** 

• Violating Rule 1: Failing to include a signal that is read in the sensitivity list usually results in a 'WARNING' from the synthesis tool

But can (technically) result in the tool adding a memory element for the output signal **process**(a)

begin
y <= a and b and c;
end process;</pre>

Signals *b* and *c* are not listed as inputs to this circuit, which means that *y* needs to maintains its value (FF) until *a* changes (*a* is treated as a clock that samples *b* and *c*)

- Violating Rule 2: There is no way to violate this rule regarding multiple assignments to an output signal within a **process** block
- Violating Rule 3: Failing to assign an output signal a *default* value causes real problems for students, with only an 'inferred xxx' warning from the tool
   A storage element is added to *preserve* the output signal value when none of the Boolean expressions are true (no assignment is made to the output signal)

## **Consequences of Violating the Golden Rules of Process Blocks**

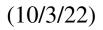
#### • Violating Rule 3:

The following 'instructs' the synthesis tool to add a storage element for eq

```
process(a, b)
    begin
    if (a = b) then
        eq <= '1';
    end if;
end process;</pre>
```

No assignment is made when a does not equal b -- the synthesis tool infers this as:

```
process(a, b)
begin
if (a = b) then
eq <= '1';
else
eq <= eq;
end if;
end process;</pre>
```



# Consequences of Violating the Golden Rules of Process Blocks • Violating Rule 3:

The synthesis tool issues 'inferred xxx' warning, and happily creates a storage element to store eq until the Boolean condition (a = b) is true again!

In fact, a variation of this syntax is used to describe valid FFs so it commonly used (as we will soon see)

```
THIS IS A COMMON BUG so beware!!!!
process(a, b) -- THIS VIOLATES GOLDEN RULE 3!
begin          -- MUST INCLUDE DEFAULT ASSIGNMENTS HERE
if (a > b) then
    gt <= '1';
elsif (a = b) then
    eq <= '1';
else
    lt <= '1';
end if;
end process;</pre>
```

#### **Consequences of Violating the Golden Rules of Process Blocks**

- Violating Rule 4: Creating combinational loops generally cause the synthesis tool to generate a 'WARNING', but beware!
- Violating Rule 5: Assigning to an output signal more than once outside a process block, or both inside and outside a process block generates a syntax error

