More on Processes Behavior …

- All processes are executed once at start-up.
- After start-up only dependencies between signal values and events on these signals determine process execution.
- Signals behave differently from variables!
library IEEE;
use IEEE.std_logic_1164.all;
entity combo is
port (In1, In2 : in std_logic;
     z : out std_logic);
end entity combo;

architecture beh of combo is
  signal s1, s2, s3, s4: std_logic;
begin
  s1 <= not In1;
  s2 <= not In2;
  s3 <= not (s1 and In2);
  s4 <= not (s2 and In1);
  z <= not (s3 and s4);
end architecture beh;

library IEEE;
use IEEE.std_logic_1164.all;
entity combo is
port (In1, In2 : in std_logic;
     z : out std_logic);
end entity combo;

architecture beh of combo is
  signal s1, s2, s3, s4: std_logic;
begin
  sig_in_proc: process (In1, In2) is
  begin
    s1 <= not In1;
    s2 <= not In2;
    s3 <= not (s1 and In2);
    s4 <= not (s2 and In1);
    z <= not (s3 and s4);
  end process sig_in_proc;
end architecture beh;
Mis-using Signals in a Process (cont.)

Using CSA statements

Using signal assignment statements within a process
library IEEE;
use IEEE.std_logic_1164.all;
entity combo is
port (In1, In2: in std_logic;
     z       : out std_logic);
end entity combo;

architecture beh of combo is
variable s1, s2, s3, s4: std_logic;
begin
sig_in_proc: process (In1, In2) is
begin
s1 := not In1;
s2 := not In2;
s3 := not (s1 and In2);
s4 := not (s2 and In1);
z <= not (s3 and s4);
end process sig_in_proc;
end architecture beh;

Use variables for computing intermediate values
Signals vs. Variables

-- Process 1 – Correct Coding Style
proc1: process (x, y, z) is
variable var_s1, var_s2: std_logic;
begin
var_s1 := x and y;
var_s2 := var_s1 xor z;
res1 <= var_s1 nand var_s2;
end process;

Process 2 – Incorrect
proc2: process (x, y, z) is
begin
sig_s1 <= x and y;
sig_s2 <= sig_s1 xor z;
res2   <= sig_s1 nand sig_s2;
end process;
In Summary: The Process Statement …

- A process statement is a concurrent statement, but all statements contained in it are sequential statements (statements that execute serially, one after another).

- The use of processes allows to raise the level of abstraction (support advanced language constructs) usable, makes your code more modular, more readable, and allows you to separate combinational logic from sequential logic.
In Summary:
The sensitivity list ...

- List of all signals that the process is sensitive to. Sensitive means that a change in the value of these signals will cause the process to be invoked.
For Combinational Logic: the sensitivity list must be complete !!!

```
process (a)
variable a_or_b;
begin
  a_or_b := a or b;
  z <= a_or_b;
end process;
```

-- since b is not in the
-- sensitivity list, when
-- a change occurs on b
-- the process is not
-- invoked, so the value
-- of z is not updated
-- (still “remembering” the
-- old value of z)
Combinational Logic: incomplete sensitivity list effect

a

b

z

z

(VHDL)

(gate level)
For Combinational Logic:
What to put in sensitivity list?

- All signals you do a test on and all signals that are on the right side of an assignment.
- In other words all the signals you are “reading” in the value
- Don’t read and write a signal at the same time !!!
Bad coding example: delta time issues!!!

architecture bad of logic is
signal a_or_b : std_logic;
begin
logic_p: process(a,b,c)
begin
a_or_b <= a or b;
end process;
end bad;

Do not “read” and “write” a signal at the same time !!!

Let’s assume we enter the process at time “t”.

\[ a_{-}o_{-}r_{-}b \text{ is scheduled at } t + \Delta \]

This is the value at ”t” instead of the updated value (at ”t+\Delta ”)
How to fix the bad coding example

architecture good of logic is
variable a_or_b : std_logic;
begin
  logic_p: process(a,b,c)
    begin
      a_or_b := a or b;
      z <= a_or_b and c;
    end process;
end good;

Use Variables for intermediate operations