GONZAGA UNIVERSITY

VHDL: Concurrent Coding vs. Sequential Coding

GONZAGA UNIVERSITY

Concurrent Coding

- Concurrent = parallel
- VHDL code is inherently concurrent
- Concurrent statements are adequate only to code at a very **low level** of abstraction
- Use concurrent coding with extreme moderation (i.e. avoid it with rare exceptions !!)
- In general, with concurrent code we can **only** build **combinational logic circuits** (the only exception is when guarded blocks are used)

GONZAGA UNIVERSITY

Concurrent statements

- simple WHEN (WHEN/ELSE)
- selected WHEN (WITH/SELECT/WHEN)
- operators
  (NOT, AND, NAND, OR, NOR, XOR, NXOR, +, −, etc.)
- generate
- block (simple and guarded)

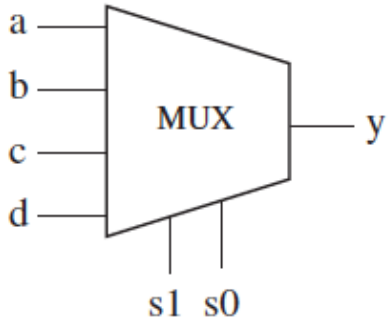
I can't remember how to use them and I do not want to know !!!

But …
if you care: see Pedroni, Circuit Design and Simulation with VHDL, 2/e, MIT Press for examples

GONZAGA UNIVERSITY

Example: mux using operators

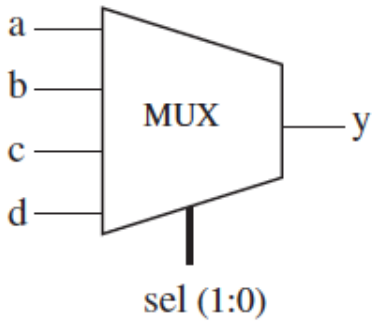
```
1  ---------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7            y: OUT STD_LOGIC);
8  END mux;
9  ---------------------------------------
10 ARCHITECTURE pure_logic OF mux IS
11 BEGIN
12    y <=  (a AND NOT s1 AND NOT s0) OR
13          (b AND NOT s1 AND s0) OR
14          (c AND s1 AND NOT s0) OR
15          (d AND s1 AND s0);
16 END pure_logic;
17 ---------------------------------------
```



GONZAGA UNIVERSITY

Example: Mux using WHEN/ELSE

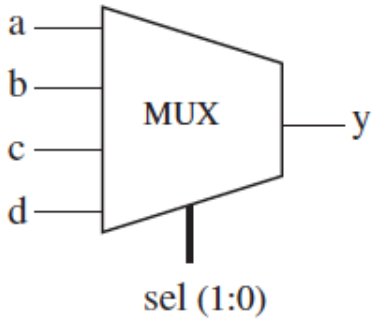
```
1  ------- Solution 1: with WHEN/ELSE --------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d: IN STD_LOGIC;
7            sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8            y: OUT STD_LOGIC);
9  END mux;
10 ---------------------------------------
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13    y <=  a WHEN sel="00" ELSE
14          b WHEN sel="01" ELSE
15          c WHEN sel="10" ELSE
16          d;
17 END mux1;
18 ---------------------------------------
```



GONZAGA UNIVERSITY

Example: Mux using WITH/SELECT/WHEN

```
1  --- Solution 2: with WITH/SELECT/WHEN -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d: IN STD_LOGIC;
7            sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8            y: OUT STD_LOGIC);
9  END mux;
10 ----------------------------------------
11 ARCHITECTURE mux2 OF mux IS
12 BEGIN
13    WITH sel SELECT
14       y <=  a WHEN "00",      -- notice "," instead of ";"
15             b WHEN "01",
16             c WHEN "10",
17             d WHEN OTHERS;    -- cannot be "d WHEN "11" "
18 END mux2;
19 ----------------------------------------
```



Whenever WITH / SELECT / WHEN is used, all permutations must be tested, so the keyword OTHERS is often useful. Another important keyword is UNAFFECTED, which should be used when no action is to take place.

```
---- With WITH/SELECT/WHEN -------------------
WITH control SELECT
   output <= "000" WHEN reset,
             "111" WHEN set,
             UNAFFECTED WHEN OTHERS;
----------------------------------------------
```

GONZAGA UNIVERSITY

simple WHEN vs. select WHEN

WHEN / ELSE:

```
assignment WHEN condition ELSE
assignment WHEN condition ELSE
...;
```

WITH / SELECT / WHEN:

```
WITH identifier SELECT
assignment WHEN value,
assignment WHEN value,
...;
```

WHEN value can take up to three forms:

```
WHEN value                    -- single value
WHEN value1 to value2         -- range, for enumerated data types
                              -- only
WHEN value1 | value2 |...     -- value1 or value2 or ...
```

GONZAGA UNIVERSITY

Sequential Coding

- Can be used for building both combinational and sequential logic circuits
- PROCESS
- FUNCTION
- PROCEDURE

Any section of code inside a PROCESS, FUNCTION, or PROCEDURE is executed sequentially. However, as whole any of these "sections" is still concurrent with any other statement outside it

* … more about them when we switch our focus to testbenches (and System-Level Design)
* For now the focus is circuit design (synthesizable coding)

Statements allowed only in sequential coding

GONZAGA UNIVERSITY

- IF
- WAIT
- CASE
- LOOP

PROCESS

- A process is executed every time a signal in the sensitivity list changes (or the condition related to WAIT is full filled)

Do not initialize sythesizable code !!

```
[label:] PROCESS (sensitivity list)
  [VARIABLE name type [range] [:= initial_value;]]
BEGIN
  (sequential code)
END PROCESS [label];
```

Objects allowed only in sequential coding

GONZAGA UNIVERSITY

- VARIABLE

Variables can never be global so their value cannot be passed out directly

Signals can be global

Signals can be declared in a PACKAGE, ENTITY or ARCHITECTURE

Variables can be declared in a PROCESS, FUNCTION or PROCEDURE

variable vs. signal

- New value updated immediately
- The value is always local
- :=

- New value updated after the conclusion of the present run of the PROCESS (FUNCTION or PROCEDURE)
- The value can be global
- <=

GONZAGA UNIVERSITY

IF statement

```
IF conditions THEN assignments;
ELSIF conditions THEN assignments;
...
ELSE assignments;
END IF;
```

Example:

```
IF (x<y) THEN temp:="11111111";
ELSIF (x=y AND w='0') THEN temp:="11110000";
ELSE temp:=(OTHERS =>'0');
```

GONZAGA UNIVERSITY

Example: Counter

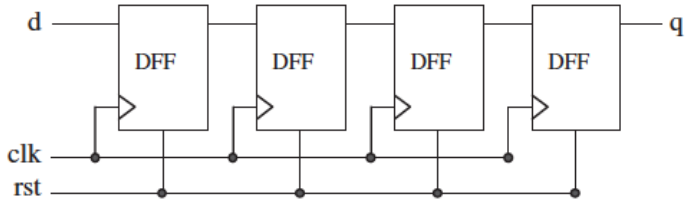
```
1  ---------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------
5  ENTITY counter IS
6     PORT (clk : IN STD_LOGIC;
7           digit : OUT INTEGER RANGE 0 TO 9);
8  END counter;
9  ---------------------------------------------
10 ARCHITECTURE    rtl    OF counter IS
11 BEGIN
12    count: PROCESS(clk)
13       VARIABLE temp : INTEGER RANGE 0 TO 10;
14    BEGIN
15       IF (clk'EVENT AND clk='1') THEN
16          temp := temp + 1;
17          IF (temp=10) THEN temp := 0;
18          END IF;
19       END IF;
20       digit <= temp;
21    END PROCESS count;
22 END    rtl   ;
23 ---------------------------------------------
```

constructive criticism:
cnt_v is a way better name for the variable temp

GONZAGA UNIVERSITY

Example: Shift Register

```
1  ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY shiftreg IS
6     GENERIC (n: INTEGER := 4);     -- # of stages
7     PORT (d, clk, rst: IN STD_LOGIC;
8           q: OUT STD_LOGIC);
9  END shiftreg;
10 ------------------------------------------------
11 ARCHITECTURE     rtl     OF shiftreg IS
12    SIGNAL internal: STD_LOGIC_VECTOR (n-1 DOWNTO 0);
13 BEGIN
14    PROCESS (clk, rst)
15    BEGIN
16       IF (rst='1') THEN
17          internal <= (OTHERS => '0');
18       ELSIF (clk'EVENT AND clk='1') THEN
19          internal <= d & internal(internal'LEFT DOWNTO 1);
20       END IF;
21    END PROCESS;
22    q <= internal(0);
23 END    rtl    ;
24 ------------------------------------------------
```



Why d is not in the sensitivity list ?

concurrent execution

concurrent statement

GONZAGA UNIVERSITY

WAIT statement

- When WAIT is employed the PROCESS cannot have a sensitivity list

```
WAIT UNTIL signal_condition;
```

```
WAIT ON signal1 [, signal2, ... ];
```

```
WAIT FOR time;
```

WAIT FOR is intended for simulation only
(NOT synthesizable)

The use of WAIT statements is **highly discouraged** when writing synthesizable code (but … it is very useful when writing TBs)

GONZAGA UNIVERSITY

Examples: WAIT statement

Example: 8-bit register with synchronous reset.

```
PROCESS          -- no sensitivity list
BEGIN
   WAIT UNTIL (clk'EVENT AND clk='1');
   IF (rst='1') THEN
      output <= "00000000";
   ELSIF (clk'EVENT AND clk='1') THEN
      output <= input;
   END IF;
END PROCESS;
```

NOT RECOMMENDED

```
-- Recommended Coding Style
PROCESS (clk)
BEGIN
   IF (clk='1' AND clk'EVENT) THEN
      IF (rst='1') THEN
          output => (others => '0');
      END IF;
      output <= input;
   END IF;
END PROCESS;
```

Example: 8-bit register with asynchronous reset.

```
PROCESS
BEGIN
   WAIT ON clk, rst;
   IF (rst='1') THEN
      output <= "00000000";
   ELSIF (clk'EVENT AND clk='1') THEN
      output <= input;
   END IF;
END PROCESS;
```

NOT RECOMMENDED

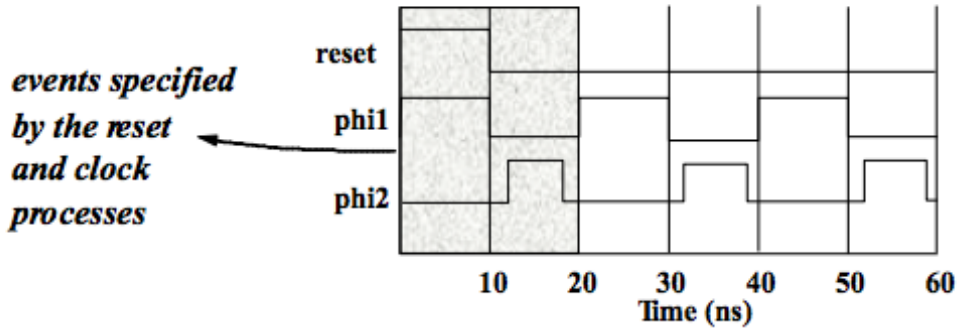
```
-- Recommended Coding Style
PROCESS (clk,rst)
BEGIN
   IF (rst='1') THEN
      output => (others => '0');
   ELSIF (clk='1' AND clk'EVENT) THEN
      output <= input;
   END IF;
END PROCESS;
```

WAIT Statement: Waveform Generation

```
library IEEE;
use IEEE.std_logic_1164.all;
entity two_phase is
port(phi1, phi2, reset: out std_logic);
end entity two_phase;

architecture behavioral of two_phase is
begin
rproc: reset <= '1', '0' after 10 ns;
clock_process: process is
begin
phi1 <= '1', '0' after 10 ns;
phi2 <= '0', '1' after 12 ns,
        '0' after 18 ns;
wait for 20 ns;
end process clock_process;

end architecture behavioral;
```



NOTE: the perpetual nature of a process without sensitivity list

GONZAGA UNIVERSITY

CASE statement

```
CASE identifier IS
    WHEN value => assignments;
    WHEN value => assignments;
    ...
END CASE;
```

Example:

```
CASE control IS
   WHEN "00" => x<=a; y<=b;
   WHEN "01" => x<=b; y<=c;
   WHEN OTHERS => x<="0000"; y="1111";
END CASE;
```

Useful KEYWORD to use when no action take place:
NULL

WHEN value can take up to 3 forms:

```
WHEN value                  -- single value
WHEN value1 to value2       -- range, for enumerated data types
                            -- only
WHEN value1 | value2 |...   -- value1 or value2 or ...
```

CASE vs. IF

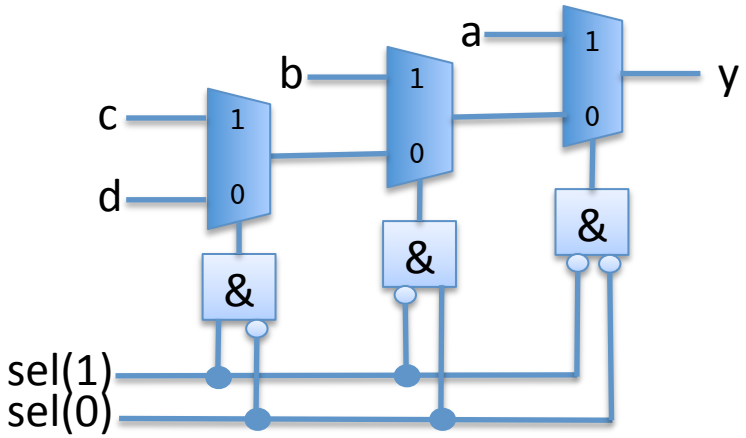
multiplexer ← → Priority encoder

- Think Hardware !!!

```
PROCESS(sel,a,b,c,d)
BEGIN
CASE sel IS
  WHEN "00"    =>  y <= a;
  WHEN "01     =>  y <= b;
  WHEN "10"    =>  y <= c;
  WHEN others  =>  y <= d;
END CASE;
END PROCESS;
```

```
PROCESS(sel,a,b,c,d)
BEGIN
IF sel="00"     THEN y <= a;
ELSIF sel="01" THEN y <= b;
ELSIF sel="10   THEN y <= c;
ELSE y <= d;
END IF;
END PROCESS;
```

c and d are late arriving signals !!



GONZAGA UNIVERSITY

LOOP Statement

FOR/LOOP: the loop is repeated a fixed number of times

```
[label:] FOR identifier IN range LOOP
  (sequential statements)
END LOOP [label];
```

← SYNTHESIZABLE

WHILE/LOOP: the loop is repeated until a condition no longer hold

```
[label:] WHILE condition LOOP
  (sequential statements)
END LOOP [label];
```

← NOT SYNTHESIZABLE

EXIT: used to end a loop

```
[label:] EXIT [label] [WHEN condition];
```

← NOT SYNTHESIZABLE

NEXT: used for skipping loop steps

```
[label:] NEXT [loop_label] [WHEN condition];
```

← NOT SYNTHESIZABLE

GONZAGA UNIVERSITY

Loop examples

```
FOR i IN 0 TO 5 LOOP
    (statements)
END LOOP;
```

```
WHILE (i < 10) LOOP
    (statements)
END LOOP;
```

```
FOR i IN data'RANGE LOOP
   CASE data(i) IS
      WHEN '0' => count:=count+1;
      WHEN OTHERS => EXIT;
   END CASE;
END LOOP;
```

```
FOR i IN 0 TO 15 LOOP
   NEXT WHEN i=skip;    -- jumps to next iteration
      (...)
END LOOP;
```

GONZAGA UNIVERSITY

Clocking Issues

```
PROCESS (clk)
BEGIN
   IF(clk'EVENT AND clk='1') THEN
      counter <= counter + 1;
   ELSIF(clk'EVENT AND clk='0') THEN
      counter <= counter + 1;
   END IF;
   ...
END PROCESS;
```

BAD Clocking: It is not possible to synthesize code that contain assignments on both transitions of the clock signal, in the same process. In addition, in this example the signal counter is assumed to be multiply driven

```
PROCESS (clk)
BEGIN
   IF(clk'EVENT) THEN
      counter := counter + 1;
   END IF;
   ...
END PROCESS;
```

BAD Clocking: the attribute EVENT cannot be used by itself. It must be Related to a test condition.

Working with both edges of a clock is a highly unreliable practice !!!
Duty Cycle may not be 50%

GONZAGA UNIVERSITY

Clocking Issues: Solution

- Work with a clock twice the speed of the original clock and use only one clock edge.

```
-----------------------------------------
PROCESS (clkby2)
BEGIN
  IF (clkby2='1' AND clkby2'EVENT) then
    counter <= counter + 1;
  END IF;
END PROCESS
-----------------------------------------
```

Signals

- A VHDL signal models a physical signal (you can think of it like a piece of wire)
- A signal is a sequence of time-value pairs
- A signal assignment takes effect only after a certain delay (the smallest possible delay is called a “delta time”).

Variables

- All assignment to variables are scheduled (takes effect) immediately.
- If a variable is assigned a value, the corresponding location in memory is written with the new value while destroying the old value.
  - This effectively happen immediately so if the next executing statement in the program uses the value of the variable, it is the new value that is used.

GONZAGA UNIVERSITY

Signals vs. Variables

- Signals assignments are scheduled after a certain delay δ
- Variables assignments happen immediately, there is no delay

PROCESS Behavior in a nutshell

- A process statement is a concurrent statement, but all statements contained in it are sequential statement (statements that executes 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.

PROCESS behavior in a nutshell

- 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 !!!**

PROCESS behavior in a nutshell: The Sensitivity List

GONZAGA UNIVERSITY

- List of all signals the process is sensitive to.
- Sensitive means that a change in the value of these signals will cause the process to be invoked.

Combinational Logic (using processes)

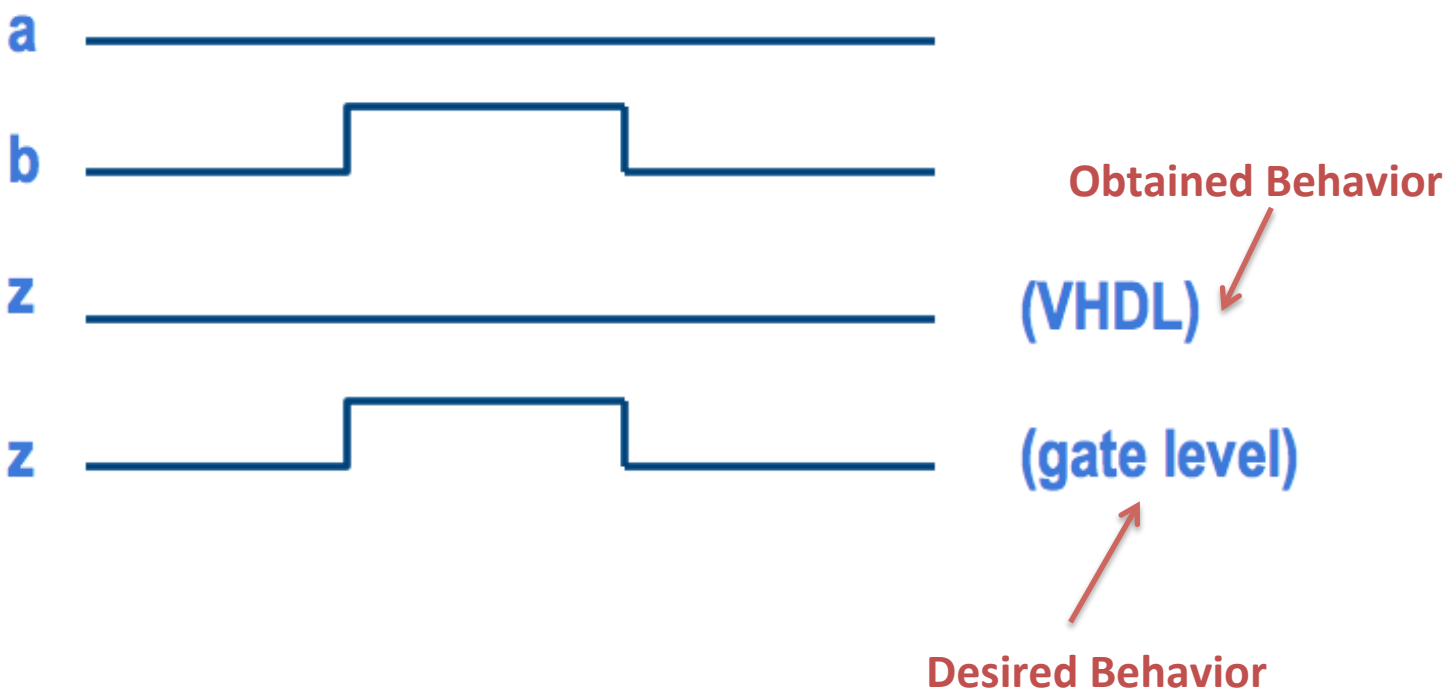
- **The sensitivity list must be complete !!!**

```
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: Effect of an Incomplete Sensitivity List

GONZAGA UNIVERSITY



Combinational Logic: What to put in sensitivity list ?

GONZAGA UNIVERSITY

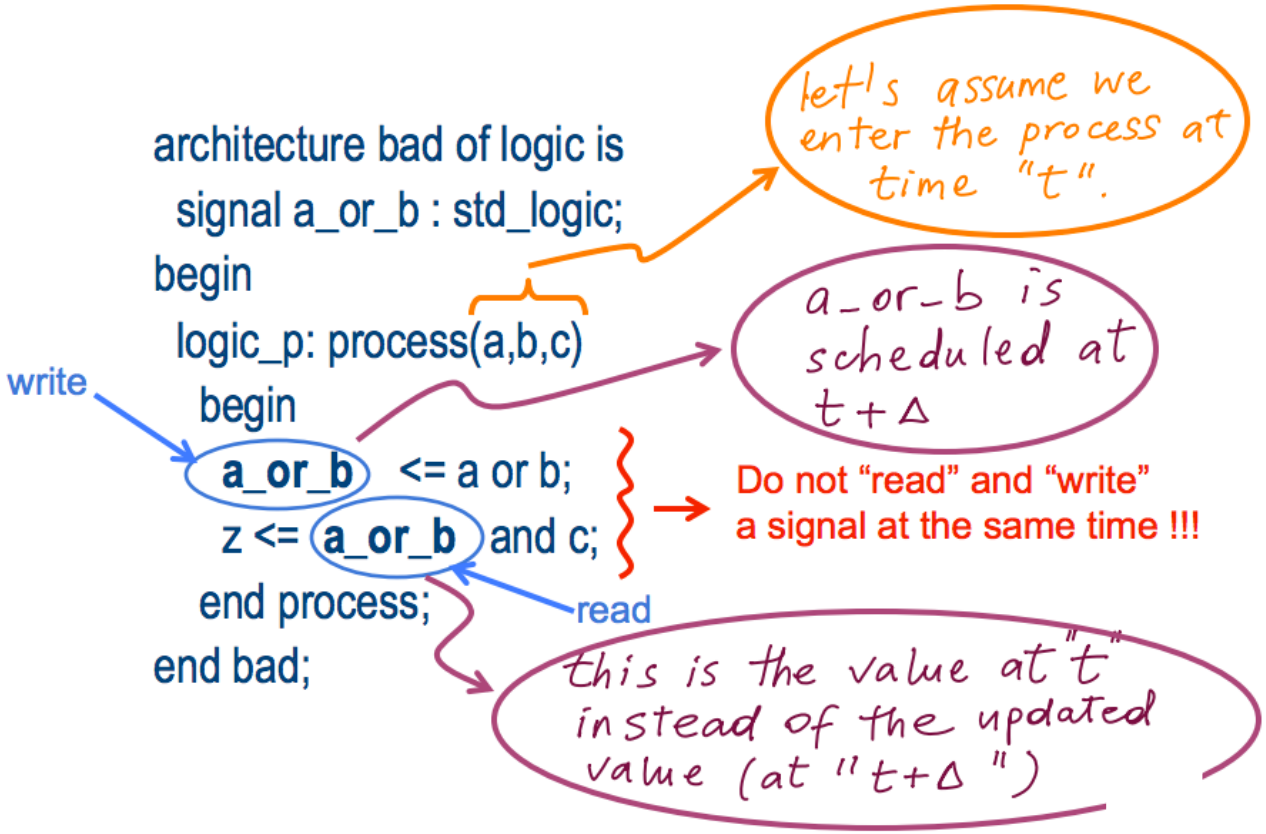
- All signals you do a test on and all signals that are on the right side of an assignment.
- In other words all the signals you are "reading in" the value

- **Don't read and write a signal at the same time !!!**

Combinational Logic: Bad coding example (δ time issues)

GONZAGA UNIVERSITY

Reading and Writing a signal at the same time



How to fix the bad coding example

```
architecture good of logic is


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

Using Sequential coding for combinational Logic

GONZAGA UNIVERSITY

- RULE 1.
  make sure all signals read in the process appear in the sensitivity list

  Incomplete sensitivity will result in the VHDL simulation not matching the gate level behavior

- RULE 2.
  make sure all signals driven in the PROCESS are defined for all possible input combinations

  Incomplete specification of the "output" signals causes the synthesis to infer latches

- RULE 3.
  Make sure no signal is driven by more than one PROCESS

  Failing RULE 3 causes Multiple drivers