

Common VHDL mistakes
**“It works perfect in simulation,
but not in the hardware!”**

Instructor: Dr. Phillip Jones
(phjones@iastate.edu)

Reconfigurable Computing Laboratory
Iowa State University
Ames, Iowa, USA

<http://class.ee.iastate.edu/cpre583/>

Overview

- Common VHDL mistakes
- What you should learn
 - What are the ~6 common mistakes
 - How to identify these mistakes
 - How to fix these mistakes

My design works in simulation, but not in hardware!!

- Clocked and non-clocked processes common issues
- Clean State machine design, using best practices
- Common VHDL Mistakes

Clocked vs. non-clock processes

Non-clocked process

(clock is **NOT** in the sensitivity list)

```
process (sel, a, my_data)
begin

  --default all driven signals
  a_out      <= x"00";
  data_out <= x"00";

  if (sel = '1') then
    a_out      <= a;
    data_out <= my_data;
  end if;
end process;
```

Clocked process

(clock is **ONLY** in the sensitivity list)

```
process (clk)
begin
  -- check for rising edge of clk
  if (clk'event and clk = '1') then

    -- initialize all driven
    signals during reset

    if (reset = '1') then
      a_out      <= x"00";
      data_out <= x"00";
    else
      if (sel = '1') then
        a_out      <= a;
        data_out <= my_data;
      end if;
    end if;

  end if;
end process;
```

Clean Finite State Machine (FSM) Design

- 3 processes
 - **1) Clocked process:** Assign next-state to state
 - **2) Combinational process:** Compute next-state, and outputs
 - **3) Clocked process:** Implement counters and registers that are controlled by the FSM outputs

Clean Finite State Machine (FSM) Design

-- 1) Assign Next_STATE to STATE

```
process (clk)
begin
-- check for rising edge of the clk
if(clk'event and clk = '1') then
  -- Initialize all driven signals
  during reset
  if(reset = '1') then
    STATE <= S1;
  else
    STATE <= Next_STATE;
  end if;
end if;
end process;
```

Has memory
(e.g. flip-flops)

-- 2) Compute next_state & outputs

```
process (STATE, x)
begin
  -- defaults
  z <= '1';
  Next_STATE <= STATE;

  case STATE is
  when S1 =>
    if(x = '0') then
      z <= '1';
      Next_STATE <= S1;
    else
      z <= '0';
      Next_STATE <= S2;
    end if;
  when S2 =>
    z <= '1';
    Next_State <= S1;
  end case
end process;
```

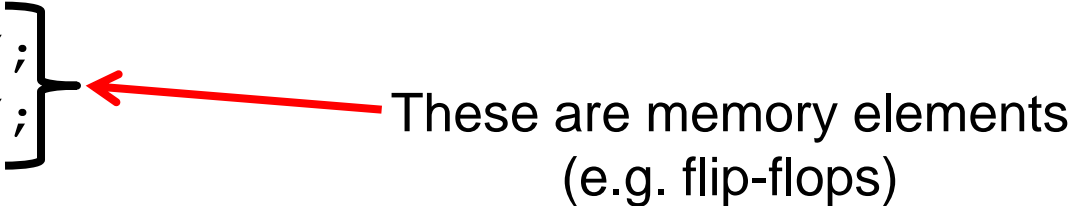
No memory!!!!

Clean Finite State Machine (FSM) Design

-- 3) Manage Registers/Counters controlled by FSM outputs

```
process (clk)
begin
  if(clk'event and clk = '1') then
    -- initialize all driven signals during reset
    if(reset = '1') then
      store_x_reg <= x"00";
      counter_1   <= x"00";
    else
      -- update registers & counters based on FSM outputs
      if(z = '1') then
        store_x_reg <= new_val;
      end if;

      if(z = '0') then
        counter_1 <= new_count;
      end if;
    end if;
  end if;
end process;
```



These are memory elements
(e.g. flip-flops)

Good papers on state machine design

- FSM “good practices” paper (Note: in Verilog)
- <http://www.sunburst-design.com/papers/>
 - [The Fundamentals of Efficient Synthesizable Finite State Machine](#) (2002)
 - [Synthesizable Finite State Machine Design Techniques](#) (2003)

Common Mistakes in more detail

1. Incomplete sensitivity list for non-clocked process

- All signals that impact the output of a non-clocked process must appear in the process's Sensitive list (Note: your clock should not be included)
- Example mistake:

```
process (a, b)
begin
    output <= '0';
    if(sel = '0') then
        output <= a;
    else
        output <= b;
    end if;
end process;
```

- Types of observed issues: Can cause **MANY** hours in debug time
 - Simulation behave in unexpected manner (e.g. signals not updating as you think they should)
 - Everything “work” in simulation, but not when deployed to hardware

1. Incomplete sensitivity list for non-clocked process

- All signals that impact the output of a non-clocked process must appear in the process's Sensitive list (Note: your clock should not be included)
- Corrected Example:

```
process (sel, a, b)
begin
    output <= '0';
    if(sel = '0') then
        output <= a;
    else
        output <= b;
    end if;
end process;
```

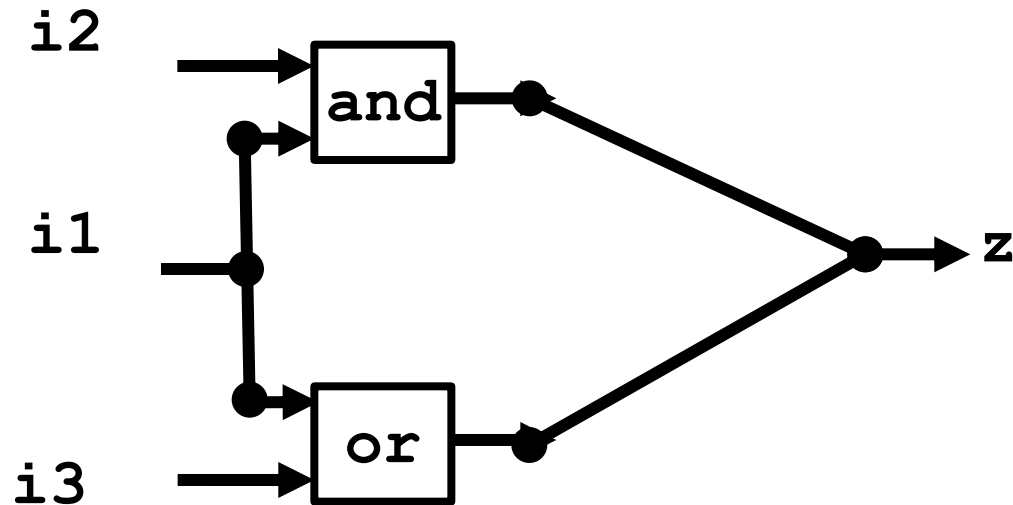
- Types of observed issues: Can cause **MANY** hours in debug time
 - Simulation behave in unexpected manner (e.g. signals not updating as you think they should)
 - Everything “work” in simulation, but not when deployed to hardware

2. Signal driven by multiple Processes / Components

- A signal can only be driven (i.e., assigned) by a single source (i.e., driver). If a signal is driven by more than one source you may observe:
 - X's assigned to the signal being driven by multiple drivers
 - A compile-time, simulation-time, synthesis-time error/warning message
- Example: Signal (e.g., **z**) having multiple drivers:

```
C1: process (i1, i2)
begin
  z <= i1 and i2;
end process: C1;

C2: process (i1, i3)
begin
  z <= i1 or i3;
end process: C2;
```

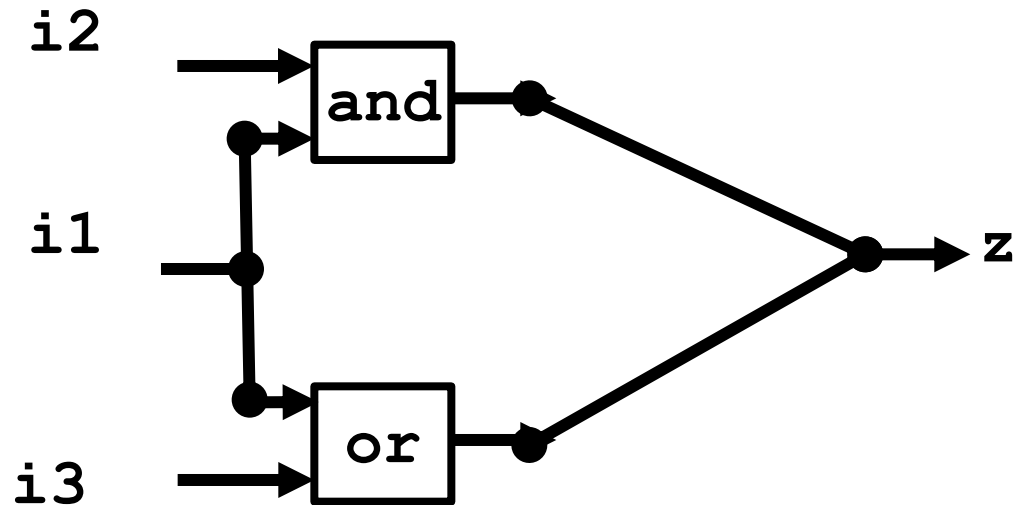


2. Signal driven by multiple Processes / Components

- A signal can only be driven (i.e., assigned) by a single source (i.e., driver). If a signal is driven by more than one source you may observe:
 - X's assigned to the signal being driven by multiple drivers
 - A compile-time, simulation-time, synthesis-time error/warning message
- Example: Signal (e.g., **z**) having multiple drivers:

```
C1: process (i1, i2)
begin
  z <= i1 and i2;
end process: C1;

C2: process (i1, i3)
begin
  z <= i1 or i3;
end process: C2;
```



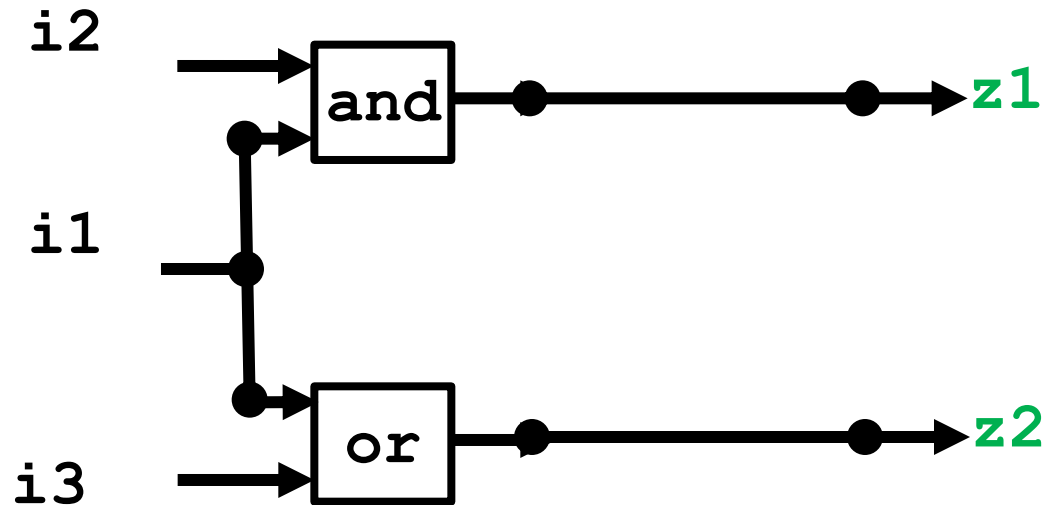
- Two solutions:
 - If a typo caused two signals to have the same name, then just fix typo
 - If meant to drive the same signal, rename signals and add a multiplexer

2. Signal driven by multiple Processes / Components

- A signal can only be driven (i.e., assigned) by a single source (i.e., driver). If a signal is driven by more than one source you may observe:
 - X's assigned to the signal being driven by multiple drivers
 - A compile-time, simulation-time, synthesis-time error/warning message
- Example: Signal (e.g., **z**) having multiple drivers:

```
C1: process (i1, i2)
begin
  z1 <= i1 and i2;
end process: C1;

C2: process (i1, i3)
begin
  z2 <= i1 or i3;
end process: C2;
```



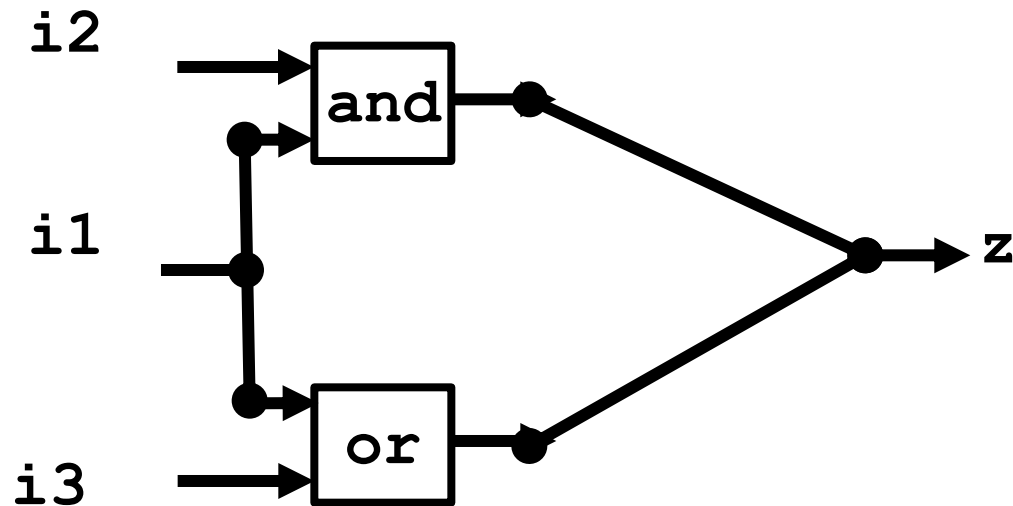
- Two solutions:
 - If a typo caused two signals to have the same name, then just fix typo
 - If meant to drive the same signal, rename signals and add a multiplexer

2. Signal driven by multiple Processes / Components

- A signal can only be driven (i.e., assigned) by a single source (i.e., driver). If a signal is driven by more than one source you may observe:
 - X's assigned to the signal being driven by multiple drivers
 - A compile-time, simulation-time, synthesis-time error/warning message
- Example: Signal (e.g., **z**) having multiple drivers:

```
C1: process (i1, i2)
begin
  z <= i1 and i2;
end process: C1;

C2: process (i1, i3)
begin
  z <= i1 or i3;
end process: C2;
```



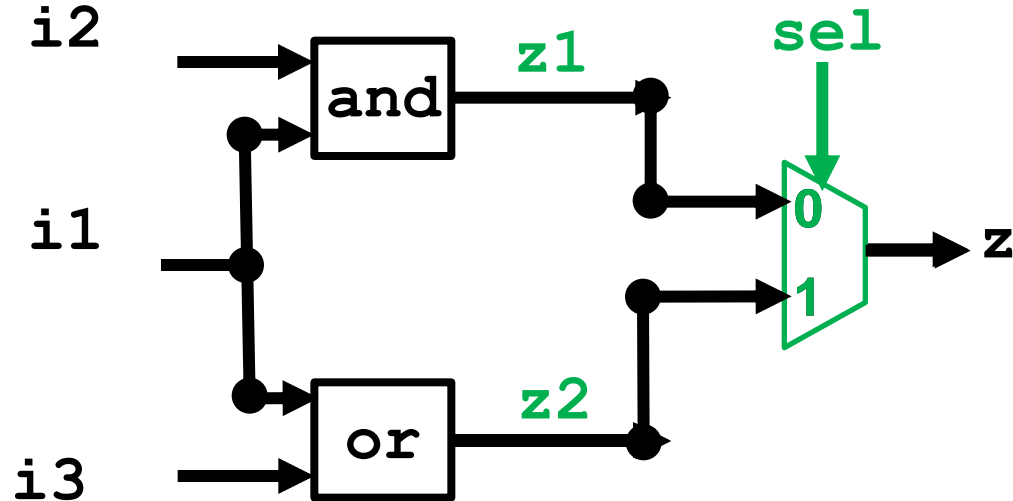
- Two solutions:
 - If a typo caused two signals to have the same name, then just fix typo
 - If meant to drive the same signal, rename signals and add a multiplexer

2. Signal driven by multiple Processes / Components

- A signal can only be driven (i.e., assigned) by a single source (i.e., driver). If a signal is driven by more than one source you may observe:
 - X's assigned to the signal being driven by multiple drivers
 - A compile-time, simulation-time, synthesis-time error/warning message
- Example: Signal (e.g., **z**) having multiple drivers:

```
C1: process (i1, i2)
begin
  z1 <= i1 and i2;
end process: C1;

C2: process (i1, i3)
begin
  z2 <= i1 or i3;
end process: C2;
```



```
when sel = 0; z <= z1, else z <= z2; // Multiplexer
```

- Two solutions:
 - If a typo caused two signals to have the same name, then just fix typo
 - If meant to drive the same signal, rename signals and add a multiplexer

3. Did not assign a default value to each signal driven by a non-clocked process (i.e. inferring a latch!!)

- Forgetting to assign a default value to one or more signals driven by a non-clocked process can result in inferring a **latch**!!
 - You're trying to hold/store a value on a signal within a non-clocked process
 - Do not try to store values in a non-clocked process, it can (**and often does**) lead to unexpected behavior in hardware. Resulting in MANY hours debugging why things “work” in simulation, but not in hardware.
- Example of inferring a latch by not defaulting a driven signal:

```
process (sel)
begin
    if(sel = '0') then
        out <= '1';
    end if;
end process;
```

- **In this case the non-clocked process will try to hold (latch) the last value assigned to “out” when select is not ‘0’.**

3. Did not assign a default value to each signal driven by a non-clocked process (i.e. inferring a latch!!)

- Forgetting to assign a default value to one or more signals driven by a non-clocked process can result in inferring a **latch**!!
 - You're trying to hold/store a value on a signal within a non-clocked process
 - Do not try to store values in a non-clocked process, it can (**and often does**) lead to unexpected behavior in hardware. Resulting in MANY hours debugging why things “work” in simulation, but not in hardware.
- Example of inferring a latch by not defaulting a driven signal:

```
process (sel)
begin
    if(sel = '0') then
        out <= '1';
    end if;
end process;
```

- Two solutions:
 - Provide 'out' with a default value at the very top of the process
 - If you actually want 'out' to store a value, then used a clocked process

3. Did not assign a default value to each signal driven by a non-clocked process (i.e. inferring a latch!!)

- Forgetting to assign a default value to one or more signals driven by a non-clocked process can result in inferring a **latch**!!
 - You're trying to hold/store a value on a signal within a non-clocked process
 - Do not try to store values in a non-clocked process, it can (**and often does**) lead to unexpected behavior in hardware. Resulting in MANY hours debugging why things “work” in simulation, but not in hardware.
- Example of inferring a latch by not defaulting a driven signal:

```
process (sel)
begin
    out <= '0'; -- default assignment
    if(sel = '0') then
        out <= '1';
    end if;
end process;
```

- Two solutions:
 - Provide 'out' with a default value at the very top of the process
 - If you actually want 'out' to store a value, then used a clocked process

3. Did not assign a default value to each signal driven by a non-clocked process (i.e. inferring a latch!!)

- Forgetting to assign a default value to one or more signals driven by a non-clocked process can result in inferring a **latch**!!
 - You're trying to hold/store a value on a signal within a non-clocked process
 - Do not try to store values in a non-clocked process, it can (**and often does**) lead to unexpected behavior in hardware. Resulting in MANY hours debugging why things "work" in simulation, but not in hardware.
- Example of inferring a latch by not defaulting a driven signal:

```
process (clk)
begin
    if (clk'event and clk = '1') then    --Check for pos edge
        if (sel = '0') then
            out <= '1';
        end if;
    end if;
end process;
```

- Two solutions:
 - Provide 'out' with a default value at the very top of the process
 - If you actually want 'out' to store a value, then used a clocked process

3. Did not assign a default value to each signal driven by a non-clocked process (i.e. inferring a latch!!)

- Forgetting to assign a default value to one or more signals driven by a non-clocked process can result in inferring a **latch**!!
 - You're trying to hold/store a value on a signal within a non-clocked process
 - Do not try to store values in a non-clocked process, it can (**and often does**) lead to unexpected behavior in hardware. Resulting in MANY hours debugging why things "work" in simulation, but not in hardware.
- Example of inferring a latch by not defaulting a driven signal:

```
process (clk)
begin
    if (clk'event and clk = '1') then    --Check for pos edge
        --> place reset logic here <--
        if (sel = '0') then
            out <= '1';
        end if;
    end if;
end process;
```

- Two solutions:

- Provide 'out' with a default value at the very top of the process
- If you actually want 'out' to store a value, then used a clocked process

3. Did not assign a default value to each signal driven by a non-clocked process (i.e. inferring a latch!!)

- Forgetting to assign a default value to one or more signals driven by a non-clocked process can result in inferring a **latch**!!
 - You're trying to hold/store a value on a signal within a non-clocked process
 - Do not try to store values in a non-clocked process, it can (**and often does**) lead to unexpected behavior in hardware. Resulting in MANY hours debugging why things “work” in simulation, but not in hardware.
- Example of inferring a latch by not defaulting a driven signal:

```
process (sel, out_not_a_reg, out)
begin
    out <= out_not_a_reg
    if(sel = '0') then
        out <= '1';
        out_not_a_reg <= out;
    end if;
end process;
```

- **In this case the non-clocked process will still try to hold (latch) the last value assigned to “out” when select is not ‘0’. But the designer is trying to use another signal from a combination process to “default” out.**

3. Did not assign a default value to each signal driven by a non-clocked process (i.e. inferring a latch!!)

- Forgetting to assign a default value to one or more signals driven by a non-clocked process can result in inferring a **latch**!!
 - You're trying to hold/store a value on a signal within a non-clocked process
 - Do not try to store values in a non-clocked process, it can (**and often does**) lead to unexpected behavior in hardware. Resulting in MANY hours debugging why things “work” in simulation, but not in hardware.
- Example of inferring a latch by not defaulting a driven signal:

```
process (sel, out_not_a_reg, out)
begin
    out <= out_not_a_reg
    if(sel = '0') then
        out <= '1';
        out_not_a_reg <= out;
    end if;
end process;
```

- **The output of a combinational process needs to be defaulted to a constant, or a register from a clocked process.**

3. Did not assign a default value to each signal driven by a non-clocked process (i.e. inferring a latch!!)

- Forgetting to assign a default value to one or more signals driven by a non-clocked process can result in inferring a **latch!!**
 - You're trying to hold/store a value on a signal within a non-clocked process
 - Do not try to store values in a non-clocked process, it can (**and often does**) lead to unexpected behavior in hardware. Resulting in MANY hours debugging why things “work” in simulation, but not in hardware.
- Example of inferring a latch by not defaulting a driven signal:

```
process (sel, out_reg)      process (clk)
begin                      begin
    out <= out_reg         if (clk'event and clk = '1') then
    if (sel = '0') then    out_reg <= out;
        out <= '1';       end if;
    end if;               end process;
end process;
```

- **The output of a combinational process needs to be defaulted to a constant, or a register from a clocked process.**

4. Trying to create a counter (or register) inside of a non-clocked process (i.e. inferring a latch!!)

- Counters (registers) should only be created within a **clocked** process. A counter adds/subtracts from its previous value, thus the signal being used to store the count value must be a register.

4. Trying to create a counter (or register) inside of a non-clocked process (i.e. inferring a latch!!)

- Counters (registers) should only be created within a clocked process. A counter adds/subtracts from its previous value, thus the signal being used to store the count value must be a register.
- **Incorrect:** The following illustrates 3 common incorrect ways of trying to create a counter within a non-clocked process.

4. Trying to create a counter (or register) inside of a non-clocked process (i.e. inferring a latch!!)

- Counters (registers) should only be created within a clocked process. A counter adds/subtracts from its previous value, thus the signal being used to store the count value must be a register.
- **Incorrect:** The following illustrates 3 common incorrect ways of trying to create a counter within a non-clocked process.
- **Correct:** Then a correct implementation of a counter is provided within a clocked process.

4. Trying to create a counter (or register) inside of a non-clocked process (i.e. inferring a latch!!)

- **Error Example 1:**

Leaving the counter signal out of a non-clocked process's sensitivity list

```
process (increment)
begin
    if(increment = '1') then
        counter <= counter + 1;
    end if;
end process;
```

4. Trying to create a counter (or register) inside of a non-clocked process (i.e. inferring a latch!!)

- **Error Example 1:**

Leaving the counter signal out of a non-clocked process's sensitivity list

```
process (increment)
begin
    if(increment = '1') then
        counter <= counter + 1;
    end if;
end process;
```

This will appear to work in simulation, but has **NO** chance of working in hardware. The “counter” will count at an unknown and uncontrollable rate in the hardware when `increment` is set to 1. **Also this will infer a latch!!**

Note: A latch is inferred because “counter is in a non-clocked process and is not given a default assignment.

4. Trying to create a counter (or register) inside of a non-clocked process (i.e. inferring a latch!!)

- **Error Example 2:**

This is the same as Error Example 1, except the “counter” signal is placed in the non-clocked process’s sensitivity list.

```
process (increment, counter)
begin
    if(increment = '1') then
        counter <= counter + 1;
    end if;
end process;
```

4. Trying to create a counter (or register) inside of a non-clocked process (i.e. inferring a latch!!)

- **Error Example 2:**

This is the same as Error Example 1, except the “counter” signal is placed in the non-clocked process’s sensitivity list.

```
process (increment, counter)
begin
    if(increment = '1') then
        counter <= counter + 1;
    end if;
end process;
```

This typically causes the simulator to error out (i.e. die). This is due to the counter trying to count at an infinite rate in simulation, since every time the counter increments the process gets rerun because the value of a signal in the sensitivity list (“counter”) changes. Which is what will happen in the hardware as well. **And again a latch is inferred!!**

4. Trying to create a counter (or register) inside of a non-clocked process (i.e. inferring a latch!!)

- **Error Example 3:**

Similar to Error Example 2, except the designer tries to “default” the “counter” with “counter”. Produces the same error, and **a latch will get inferred** (Do not do this!!)

```
process (increment, counter)
begin
    counter <= counter;
    if(increment = '1') then
        counter <= counter + 1;
    end if;
end process;
```


4. Trying to create a counter (or register) inside of a non-clocked process (i.e. inferring a latch!!)

- **Correct Example:**

Important!!!: counters and registers should only be placed within **clocked** processes.

```
process (clk)
begin
    if(clk'event and clk = '1') then
        if(reset = '1') then
            counter <= (others => '0'); -- initialize counter
        else
            if(increment = '1') then
                counter <= counter + 1;
            end if;
        end if;
    end if;
end process;
```

5. Not initializing all signals driven by clocked process

If a **clocked** process does not have a “reset” condition to initialize all signals driven by that process (i.e., its registers), then the circuit is being placed in a random state after power up. This can lead to a difference between hardware simulation and hardware deployment.

```
process (clk)
begin
    if(clk'event and clk = '1') then
        if(reset = '1') then
            out_reg <= (others => '0'); -- initialize
        else
            if(sel = '0') then
                out_reg <= '0';
            else
                out_reg <= '1';
            end if;
        end if;
    end if;
end process;
```

6. Using 'event with a non-clock signal

- `if(clk'event and clk = '1')` is used to indicate the implementation of flip-flops. Flip-flop transitions are meant to be driven by a **clock**.
- **Issues:** Using 'event for non-clock signals can result in a flip-flop that is "clocked" by a noisy signal (e.g., causing flip-flops to update randomly), can cause synthesis tools to have issues meeting timing constraints, cause synthesis tools to allocate on-chip clocking resources (e.g., clock trees)

```
process (rand_sig) // Not using a clock
begin
    if(rand_sig'event and rand_sig = '1') then
        if(reset = '1') then
            counter <= (others => '0'); -- initialize counter
        else
            if(increment = '1') then
                counter <= counter + 1;
            end if;
        end if;
    end if;
end process;
```

6. Using 'event with a non-clock signal

- `if(clk'event and clk = '1')` is used to indicate the implementation of flip-flops. Flip-flop transitions are meant to be driven by a **clock**.
- **Issues:** Using 'event for non-clock signals can result in a flip-flop that is "clocked" by a noisy signal (e.g., causing flip-flops to update randomly), can cause synthesis tools to have issues meeting timing constraints, cause synthesis tools to allocate on-chip clocking resources (e.g., clock trees)

```
process (clk) // Using a clock
begin
    if(clk'event and clk = '1') then
        if(reset = '1') then
            counter <= (others => '0'); -- initialize counter
        else
            if(increment = '1') then
                counter <= counter + 1;
            end if;
        end if;
    end if;
end process;
```

BONUS 1: Taking care with off-chip signals (Metastability)

- For signals coming from **off-chip**, a best practice is to send them through a couple of flip-flops before using. This helps avoid the signals being used becoming metastable.

```
// Mitigating metastability
process (clk)
begin
    if(clk'event and clk = '1') then
        if(reset = '1') then
            signal_d0 <= '0';
            signal_d1 <= '0';
        else
            signal_d0 <= off_chip_signal;
            signal_d1 <= d0;
        end if;
    end if;
end process;
```

BONUS 2: Trying to initialize a signal with “:=“

- **Do not** initialize signals in the declaration section of an Entity. The “:=“ is not guaranteed to be synthesized by the tools.
 - e.g. `signal my_signal : std_logic := 0.`
- Initialize your signal within their associated clocked process.
- **Note:** It is perfectly fine to use “:=“ if the signal you are declaring is of type CONSTANT.

BONUS 3: Having extra signals in sensitivity list

```
-- Combinational process
process (sel, a, my_data, a_out, data_out)
begin

    --default all driven signals
    a_out      <= x"00";
    data_out  <= x"00";

    if (sel = '1') then
        a_out      <= a;
        data_out  <= my_data;
    end if;

end process;
```

- **Not really a functional issue, but should raise yellow to red flags that the hardware designer may be new, and may be making many other mistakes in their HDL descriptions (of course everyone makes a typo occasionally).**

BONUS 3: Having extra signals in sensitivity list

```
-- Combinational process
process (sel, a, my_data, a_out, data_out)
begin

    --default all driven signals
    a_out      <= x"00";
    data_out  <= x"00";

    if (sel = '1') then
        a_out      <= a;
        data_out  <= my_data;
    end if;

end process;
```

- **Not really a functional issue, but should raise yellow to red flags that the hardware designer may be new, and may be making many other mistakes in their HDL descriptions (of course everyone makes a typo occasionally).**

BONUS 3: Having extra signals in sensitivity list

```
-- Combinational process
process (sel, a, my_data)    -- Correct
begin

    --default all driven signals
    a_out    <= x"00";
    data_out <= x"00";

    if (sel = '1') then
        a_out    <= a;
        data_out <= my_data;
    end if;

end process;
```

- **Not really a functional issue, but should raise yellow to red flags that the hardware designer may be new, and may be making many other mistakes in their HDL descriptions (of course everyone makes a typo occasionally).**

BONUS 3: Having extra signals in sensitivity list

-- Clocked process

```
process (clk, a, my_data)
begin
    if(clk'event and clk = '1') then

        if(reset = '1') then
            a_out      <= x"00";
            data_out <= x"00";
        else
            if (sel = '1') then
                a_out      <= a;
                data_out <= my_data;
            end if;
        end if;
    end if;
end process;
```

- **Not really a functional issue, but should raise yellow to red flags that the hardware designer may be new, and may be making many other mistakes in their HDL descriptions (of course everyone makes a typo occasionally).**

BONUS 3: Having extra signals in sensitivity list

-- Clocked process

```
process (clk, a, my_data)
begin
    if(clk'event and clk = '1') then

        if(reset = '1') then
            a_out      <= x"00";
            data_out <= x"00";
        else
            if (sel = '1') then
                a_out      <= a;
                data_out <= my_data;
            end if;
        end if;
    end if;
end process;
```

- **Not really a functional issue, but should raise yellow to red flags that the hardware designer may be new, and may be making many other mistakes in their HDL descriptions (of course everyone makes a typo occasionally).**

BONUS 3: Having extra signals in sensitivity list

-- Clocked process

```
process (clk) -- Correct
begin
    if (clk'event and clk = '1') then

        if (reset = '1') then
            a_out      <= x"00";
            data_out <= x"00";
        else
            if (sel = '1') then
                a_out      <= a;
                data_out <= my_data;
            end if;
        end if;
    end if;
end process;
```

- **Not really a functional issue, but should raise yellow to red flags that the hardware designer may be new, and may be making many other mistakes in their HDL descriptions (of course everyone makes a typo occasionally).**