Common VHDL mistakes

(Trying to avoid the statement “It’s working fine in simulation, but not in the hardware”)

It is important to understand the difference between a clocked and non-clocked process. The main difference is a clocked process can only update signals on a clock edge (flip-flop, register behavior), while a non-clocked process updates its signals anytime any signal in its sensitivity list changes.

Example of a non-clocked process (Note: clk is not in the sensitivity list):

```
process (sel, a, my_data)
begin
   a_out <= 0x00;      -- default assignment
  data_out <= 0x00;  -- default assignment
  if (sel = ‘1’) then
    a_out <= a;
    data_out <= my_data;
  end if;
end process;
```

Example of a clocked process (Note: clk is the only signal in the sensitivity list):

```
process (clk)
begin
  if(clk’event and clk = ‘1’) then  -- check for the rising edge of the clock
    if(reset = ‘1’) then -- initialize all driven signals during reset
      a_out <= 0x00;
      data_out <= 0x00;
    else
      if (sel = ‘1’) then
        a_out <= a;
        data_out <= my_data;
      end if;
    end if;
  end if;
end process;
```

The following are the most common mistakes I have noticed students make while first learning VHDL.

1. **Incomplete sensitivity list for non-clocked processes**

All signals that control the output of a non-clock process must appear in the process's sensitivity list (Note: your clock should not be included). Example:

```
process (sel, a, my_data)
begin
  a_out <= 0x00;
  data_out <= 0x00;
  if (sel = '1') then
    a_out <= a;
    data_out <= my_data;
  end if;
end process;
```

If you do not have a complete sensitivity list, then the simulation will not correctly emulate the behavior of the hardware. Which can result in you spending many hours trying to figure out why:

a. Simulation is not behaving like you expect (e.g. signals are not updating when you think they should
b. Everything appears to work in simulation, but not in hardware

2. **Signal driven by multiple processes, or components**

Each signal can only be driving by a single source. If you drive a signal with more than one source, then you may see:

a. X's being assigned to the signal that has multiple drivers
b. A compile-time or simulation-time error/warning message

Example of a signal with multiple drivers ("out"):

```
process( i1, i2)
begin
  out <= i1 and i2;
end process;

process (i1, i3)
begin
  out <= i1 or i3;
end process;
```

In this case “out” is being driven by two different processes. This is not allowed. If a signal is being driven by multiple sources, then the simulator does not know what value to assign to the signal. Therefore you will likely see the signal show up as an “X”.

3. **Forgot to assign a default value to every signal being driven by a non-clock process (inferring a latch)**

Forgetting to assign a default value to every signal driven by a non-clocked process can result in interfering a latch. This basically means that you are 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 your hardware. Which can cost you many hours trying to figure out why things work in simulation and not in hardware.

Example of inferring a latch by not defaulting a signal:

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

This non-clocked processes will try to hold (latch) the last value assigned to “out” when sel is not “0”. If you want this behavior then you must place “out” in a clocked process, so that “out” becomes a register.

This inferred latch can be avoided by assigning “out” a default value.

Correct Example:

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

Another common error is to try to default a signal with another signal from the same or different non-clocked process. In the following example the designer is trying to hold/store the value of "out" into "out_reg". Again it is very important to remember you can NOT create registers (flip-flops) in a non-clocked process:

Example:

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

Strictly speaking you do not have to default all signals, but you do need to make sure EVERY logical branch of a non-clocked process gives every signal that is being driven an explicit value. This however can be very error prone. The safest way to ensure you do not infer a latch in a non-clocked process is to default all signals that are driven at the top of the non-clock process.

4. **Trying to create a counter (or register) in a non-clocked process. (inferring a latch)**

Counters (registers) should only be placed in a clocked process. A counter adds/subtracts from its previous value, therefore the signal being used to store the count value must be a register.

3 common incorrect ways of trying to create a counter in a non-clocked process (counters must be put in a clocked process!):

a. **Leaving the counter signal out of the 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 uncontrollable rate in hardware when "increment" is set equal to one. Also this will infer a latch! Note: A latch is inferred because "counter" is in a non-clocked process and does not have a default assignment.

b. **Same as a) but signal is placed in the non-clocked process's sensitivity list. This typically causes the simulator to error out. 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 sensitivity list ("counter") changes. Which is what will happen in hardware as well. And again a latch is inferred.**

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

c. **Similar to b) but the designer tries to default "counter" with "counter". Again 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;
```

Important!! Counters and registers should only be placed in clocked processes.

Correct Example:

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

5. Forgetting to initialize all signals that are being driven by a clocked process.

By not initializing all signals in a clocked process to some value, you basically put your hardware in a random state after reset. This can lead to differences between simulation and hardware. Which in turn can lead to you spending many hours trying to figure out why things work in simulation, but not in hardware. This difference is due to the fact that your VHDL code typically has statements like:

```
if (sel = "0") then
  out <= '0';
else
  out <= '1';
end if;
```

Therefore in simulation if sel is equal to "U" after reset, then out <= '1'. But in hardware there is no such thing as "U". Therefore it is possible that after reset sel may be equal to '0', which would give out equal to '0', which is a different behavior. So in a clocked make sure to default the signals you are driving.
Correct Example:

```
process(clk)
begin
  if(clk'event and clk = '1') then
    if(reset = '1') then
      out <= 0x00;  -- initial signal during reset
    else
      if(sel = '0')
        out <= '0';
      else
        out <= '1';
      end if;
    end if;
  end if;
end process;
```

Note: Do not initialize signals in the declarations section (i.e. signal my_signal : std_logic := '0'). The ":=" for initialization of a signal has no meaning in hardware. You must explicitly initialize your signals in their associated clocked process. You can use the ":=" if the signal you are declaring is a CONSTANT.

6. **Do not place clk in the sensitivity list of a non-clocked process, unless clk is being used to drive a signal (this should be very rare, and you better know what you are doing if you use clk to drive another signal in a process).**