## Exercise 2:

### a) Using Design Hierarchy in VHDL

Design Hierarchy is one of the key concepts in VHDL development. Now that you have learned to test your code with ModelSim, you will learn to construct an easy hierarchical design with the modules you have developed. For this, you must declare components of the modules and connect instances of these components as shown in figure 1.

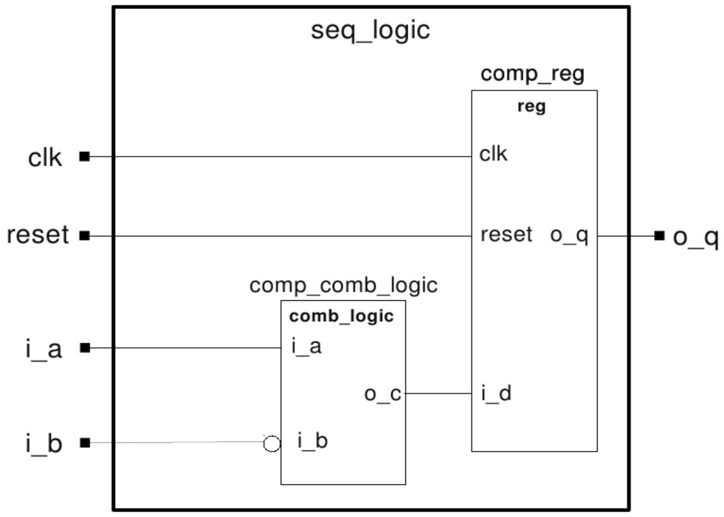

*Figure 1: seq_logic*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *i_a* : in std_logic
- *i_b* : in std_logic

We could also integrate the functionality of the combinational logic into the part of the code describing the register. Instead, however, we will use the VHDL component construct. One advantage of using components is the possibility for code reuse. Also, the simulation and debugging of the code is simplified, since you can now simulate small parts of your design. This is an important point, since you can easily have several thousand signals in a design.

To build a component, you must declare the component in the declaration part (the same part where you have to declare your signals) of your architecture first. The name of the component has to be the name of the entity you want to use in this case, the entities of the modules of Exercise 1 (*comb_logic*, *reg*). The generics and ports must be declared in the same way they are in the entities. Because no generics are used in this exercise, this declaration can be omitted.

```
-- component declaration
component entity_name is
      generic (generic_declarations );
      port (input_and_output_declarations);
end component entity_name;
```

Now, the component with its generics and ports is known to your architecture, but not yet used. To use this component, you must instantiate it inside your architecture.

First of all, you can label the component with a name deviating from the entity name (*comp_comb_logic*, *comp_reg*). Now you can connect the ports of the instantiated component with signals used in your architecture as shown in the figure. Ports of your entity can be used directly in the port map of the component. Again, you can omit the generic map. (If you want to use a generic map, be aware of the syntax! There is no semicolon between a generic map and a port map!)

```
-- component instantiation
label : entity_name
       generic map (component_generic     => generic)
       port map (component_input => input / signal,
                 component_output => output / signal);
```

Since you connect two components directly, you must declare an additional signal to connect the output o_c from *comp_comb_logic* with the input i_d from *comp_reg*.

Furthermore, a direct instantiation of components was defined in VHDL-93. This instantiation does not require a component declaration and has the following syntax.

```
-- direct component instantiation
label : entity library_name.entity_name(architecture_name)
         generic map (component_generic   => generic)
         port map (component_input => input / signal,
                   component_output => output / signal);
```

If you use this instantiation, consider that you must specify the library in which your entity resides.

Now set up a new ModelSim project with the Project Navigator and simulate and verify your design with the given testbench! Do not forget to compile the files of Exercise 1 to your ***work*** library, otherwise your components will be unbound and the simulation will not work.

**b) Debugging of a counter**

The next task is to debug a given six-bit counter.

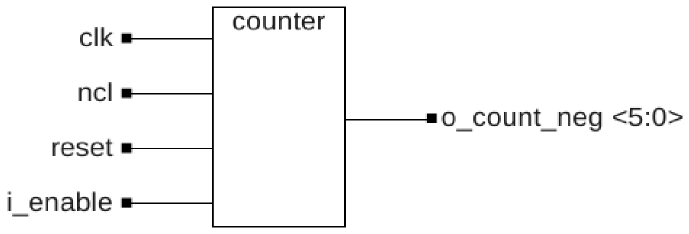

*Figure 2: counter*

**Port declaration:**

| | |
|---|---|
| ● *clk* | : in std_logic |
| ● *ncl* | : in std_logic |
| ● *reset* | : in std_logic |
| ● *i_enable* | : in std_logic |
| ● *o_count_neg* | : out std_logic_vector (5 downto 0) |

The counter should implement the following functions:

- count up from 0  to 63 in steps of 1.
- counting must be enabled by setting the input *i_enable* to 1, else the counter retains its old value.
- the input *reset* works as a synchronous and the input *ncl* as an asynchronous reset. Both set the counter back to 0.
- the output *o_count_neg* is the inverted value of the actual counter.

Write a simulation script and correct the mistakes in the module. You will find syntactic, as well as functional mistakes.