Exercise 3

a) Signals vs. Variables in VHDL

In the previous exercises, you used **signals** to describe your hardware. VHDL also offers the possibility to use **variables**, which have a different behavior.

The main functional difference between signals and variables is the “value history” associated with signals. Variables do not have a history of their previous values; a variable only knows its current value. This especially results in a different behavior when assigning a value within a process. The value of a signal will be updated at the end of the execution of the process, while variables will be updated immediately. Because of this property, variables can be overwritten with a sequential statement in a process. Overwriting a signal with multiple assignments in a sequential process will just use the last statement while ignoring all earlier assignments.
The behavior of variables can e.g. be exploited in complex calculations to hold intermediate values, but this can also lead to problems: It is not possible to see the intermediate changes of a variables' value over time in a waveform during simulation, which can complicate debugging. Also, it is not possible to use variables in the sensitivity list of the process, which can result in differences between the simulation results and the synthesized hardware (synthesizers mostly ignore sensitivity lists and just issue a warning).
For larger designs like memories, it might be advantageous to use variables, since they consume less memory resources during simulation than signals do.

As you can see in the example code below, the syntax for variables is slightly different than the syntax for signals.

```
-- usage of signals
architecture rtl of example is
      signal sig: std_logic_vector(3 downto 0) := (others => '0');
begin
      sig <= “0010”;
end;

--  usage of variables
process(sensitivity list) is
      variable var: integer;
begin
      var := var + 1;
end process;
```

In this exercise, you must design two modules which perform the same calculation. To clarify the differences between signals and variables, you must implement one of the modules using variables and the other using signals.

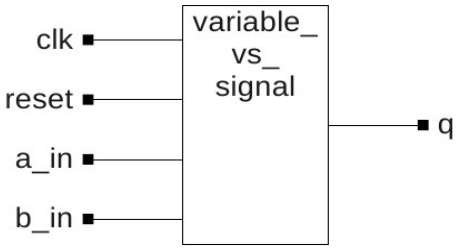

Figure 1: variable_vs_signal

The calculation shown in Figure 2 should be performed. The values of ***a_in*** and ***b_in*** must be fed to A and B, the result E must be fed to ***q***. Try to convert all the values from type std_logic_vector to type integer to perform the calculation. You can search on the internet for the conversion functions you will need.

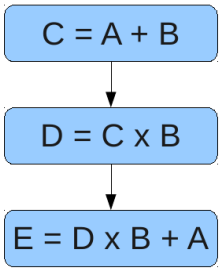

Figure 2:
Calculation rule

In this exercise, no explicit testbench is given. The stimuli for the inputs are provided by the ModelSim simulation script with the help of the ***force*** command. This is an easy approach to generate stimuli, but it will not be sufficient to test larger and more complex designs.

Try to find out the differences between variables and signals by analyzing the waveforms of both modules. If you understand the differences properly, you are free to use either signals or variables to describe your hardware in the upcoming exercises. But since we neither have complex calculations nor very large designs in our lab, the general recommendation is to use signals.

b) Serial To Parallel Conversion

This block should convert a serial data stream into a parallel data word using a shift register.

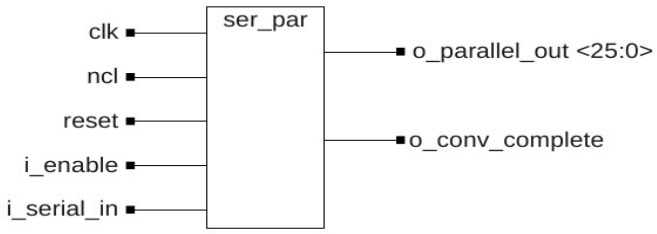

Figure 3: ser_par

Port declaration:

- *clk* : in std_logic
- *ncl* : in std_logic
- *reset* : in std_logic
- *i_enable* : in std_logic
- *i_serial_in* : in std_logic
- *o_parallel_out* : out std_logic_vector (25 downto 0)
- *o_conv_complete* : out std_logic

To do so, each new incoming bit must be shifted into the register on a rising clock edge when the *i_enable* signal is active. You can realize a shift register by using the concatenation operator **&.** This operator concatenates vectors, as shown in Figure 4.

Figure 4: Concatenation operation

After the conversion of each data word, a reset is triggered from the outside and the value of the shift register is set to 0. The serial data stream is sent to the converter with the MSB first and consists of 27 bits. The MSB is always 1 and is called the start bit of the data word.
When the whole data stream has been completely shifted into the register, this start bit has reached the MSB position of the shift register, which is assigned to the output o_*conv_complete*. This output signifies that a complete data word has been converted and that it is available at the output *o_parallel_out*.