Oblig 5
Functions, packages and a self-checking testbench

IN3160/4160

Version 2023-12-07

In this exercise, you will explore VHDL functions and create a VHDL package. You will also build a self-checking testbench with Cocotb.
Optionally, a2) will highlight some architecture features in FPGA design.

a) Simulate the attached code in **pargen.vhd** and **tb_pargen.py** (the given **Makefile** may be used). Here two 16-bit vectors *(indata1* and *indata2)* are read in and a parity signal *(par)* is generated.
Create a PNG-image of the simulation result.
(gtkwave: File->Grab to file)

b) Modify the code in **pargen.vhd** to encapsulate the two different methods in separate functions for creating parity calculation.

c) Move the functions in b) to a subprog_pck package. Modify **pargen.vhd** from part b of the exercise to use the functions from the subprog_pck package.

d) Modify the given **tb_pargen.py** and complete the function *stimuli_generator()* so that it generates randomized stimuli for *indata1* while *indata2 is counted up*. At least 20 random values should be applied.

e) Modify the given **tb_pargen.py** and complete the function *compare(dut) so that the output "par" and the internal signals "xor_parity" and "toggle_parity" are checked each clock cycle*. Use assertions on the monitored signals. (*If you have made changes to their names, check the new corresponding signals*). You may use the provided functions *parity(value)* and *predict(dut)* to calculate the expected values.

- *During simulation, values should be checked after all delta-delays caused by the trigger have been simulated*. In cocotb, this can typically be achieved using **await** ReadOnly() after awaiting the clock edge.
- *Hint: output par updates only on rising_edge(mclk).*
- *The check itself may run indefinitely, as long as it is disjoint from stimuli generation.*

Hint #1: Type conversions can be a bit tricky in VHDL. Going from integer to std_logic_vector requires deciding on whether you will use signed data or not. Here is an example on how to convert from integer to an arbitrary length std_logic_vector using numeric_std and std_logic_1164 libraries:

```
my_var := std_logic_vector( to_unsigned(i, my_var'length) );
```

Approval:

- Simulation result (png) for a) and e)
- VHDL source file for the individual questions.
- Modified tb_pargen.py
- Optional: Answers to optional questions (below).

This assignment may be approved in lab. When approved in lab, comments and feedback are given orally, and there is no need for uploading content. The lab supervisor will mark the assignment as demonstrated in lab in the learning platform (Canvas).

a2) (optional addition after a))

Create a project using the pargen.vhd file *before* modifications) in Vivado. (Remember to select VHDL 2008 for both sources).
Open the *RTL-analysis->Elaborated Design*, and look at the schematic generated. Zoom in and compare the paths of the parity_toggle and the XOR_Parity (RTL_REDUCTION_XOR).

- How many gates are required for each path?
  (Not counting inverters, only (N)AND/ (N)OR/ X(N)OR).
- If we would implement this schematic in a full custom ASIC, which version would be favorable? (consider how many gate delays they will induce)

Synthesize the design, and open the synthesized design schematic. Note that the differences you will see compared to the RTL schematic are mostly due to the FPGA architecture consisting of logical blocks having mostly look-up tables (LUTs) and flip-flops.

Does it seem to make any difference whether our code indicates the use of multiplexers or reduction XOR if we implement it on a Xilinx Zynq-series FPGA?