Exercise 2: Generic Program Counter

After the first exercise introduced the Vivado Workflow, this second exercise is concerned with the first integral part of a CPU architecture: The *Program Counter*. The implementation of a program counter is divided into three parts, an adder implementation, a register and the overall integration. For this exercise, some skeleton files and constraints are given.

Task 1. Implement a generic adder. Setup a Vivado project and add the following files:

- `adder.vhd` (Design Source)
- `adder_tb.vhd` (Simulation Source)
- `constraints.xdc` (Constraint)

For this task, you need to implement an adder. `adder.vhd` is a skeleton of the adder design, there are comments at all positions where input is needed. Your design should satisfy the following requirements:

- There are three ports: two inputs for operands and one output
- The output is the sum of the two inputs
- The addition is sign-sensitive
- All ports, i.e. the operands and the results, have a bit-width of `bits`.

Once you finish the design, verify its functionality using the provided testbench (`adder_tb.vhd`).

Task 2. The second integral component to the program counter is the internal register. Within the same project created in the previous step, implement a register description.Add the provided sekelton of the register implementation and the corresponding testbench to your project:

- `reg.vhd` (Design Source)
- `reg_tb.vhd` (Simulation Source)

Implement you desing and verify your its functional correctness via simulation. For this, ensure, that the correct TOP file for simulation is selected before launching the behavioral simulation.

Task 3. Finish the program counter implementation. With both components, the adder and the internal register, completed, finish the overall program counter design. Again, there are provided files you should add to the project.

- `program_counter.vhd` (Design Source)
- `program_counter_tb.vhd` (Simulation Source)
- `program_counter_wrapper.vhd` (Design Source)

Complete the program counter description (`program_counter.vhd`). The implementation should satisfy the following requirements.

- The stored state can be reset to 0.
- The stored state can be set to a value which is input (`pc_in`) on the condition, that `load = '1'`
- If neither the state is reset (`rst = '1'`) or loaded, then it should be incremented by 4 on every clock cycle.
- The program counter should store an internal value with a bit-width of `bits`.

Verify the correctness of your design via simulation.

The provided wrapper connects the LEDs, switches and some buttons of the ZEDboard. Modify the constraint file from earlier by un-commmenting the board-parts you now need. This includes some LEDs and some buttons, as well as the global clock. Ensure, that `program_counter_wrapper` is selected as the top entity. Finish the workflow and program the FPGA. Now you should have a working program counter. Verify, that the state is incrementing, that it can be reset and that a new state can be loaded.

Now, you can test out your design on the FPGA. The switches of the ZEDBoard are connected to the state input of the program counter. The left and right button act as reset and load respectively. Meanwhile, the LEDs show the current state stored in the program counter.

Bonus Task 1. Optimize the implementation of the program counter. Can the design be improved, are there unneeded parts? What happens to these optimizations during synthesis?