Electronic Lock System FPGA Implementation

The aim of this lab is to implement the electronic lock system from Exercise 4 in VHDL and to test it on the FPGA board. Therefore, you have to finish Exercise 4 before starting this lab. In this lab, it is assumed that you are familiar with the design steps in Vivado from Lab 1.

Hand-in instructions: Prepare a small report with block diagrams. Submit the report as a PDF with the source code files through the lecture moodle per the moodle submission deadline.

Important information ⚠

Please always check these things before you start a lab or when you have issues:

- EDA server users must start Vivado with `vivado -source load_board_files.tcl` as described in Lab 1. If you do not see the board files in the Vivado GUI, you have likely used the command with a spelling error or something similar.
- Avoid spaces and special characters in your filepaths! Vivado projects can become corrupted if you have spaces and special characters in your filepaths.
- Remember to use the reset button/switch on the FPGA to reset your design.
- When the reset is mapped to a switch, check that this switch is properly turned off before deciding the FPGA is broken.
- Remember to connect the Ethernet port of the FPGA board to a computer, router, or any other device with an Ethernet port that can power the Ethernet chip on the FPGA board (no internet connection is needed). See the Lab 2 manual for an explanation.
- For combinational logic, use `process(all)`. Do not write your own sensitivity lists! Please remember to change files using `process(all)` to VHDL 2008. This is done by selecting the file in the `Source` tab. Look at the `Source File Properties` tab and change the type to `VHDL 2008` by clicking on the 3 dots in the `Type` field.
- For defining registers, use the clocked process style `process(CLKxCI, RSTxRI)`. Do not define combinational logic like `CNTxDP <= CNTxDP + AxDI` inside a clocked process! See Task 4 in Exercise 3 and its accompanying solution for an example on why this is bad.

For common questions to this lab, please see the last page of this document.

Task 1: RTL Design

Download the provided `.zip` file from moodle. This time, the handout only contains a proposed work directory structure, the standard `.xdc` file, and the board-files (for those using the servers). The remaining parts, the source code, the completed `.xdc` file, and the testbench, you will implement in this lab. For the source code, we suggest to create two files:

- `key_lock_timed.vhdl`: This file contains your implementation of the lock with the extended opening from Exercise 4.
- `toplevel.vhdl`: The top-level wraps your design from `key_lock_timed.vhdl` for implementation on the FPGA. On the FPGA, we will use the four push-buttons, with

push-buttons 0 to 3 corresponding to the *Key* signal from Exercise 4 being equal to the push-button number. Therefore, you have to map a press on the buttons to the key code, i.e., when push-button 3 is pressed, the *Key* signal is 3.

As the *Key* signal has to have a unique value, you have to ensure that the *KeyValid* signal is only '1' if one key is pressed. If multiple or no keys are pressed, *KeyValid* has to be '0'.

For your toplevel interface, we propose a naming scheme as shown in Listing 1. As we are using all four push-buttons you have to map the `RSTxRI` signal to a switch in the `.xdc` file. Please note that while VHDL itself is case-insensitive, **`.xdc` constraints are case sensitive**. Therefore, the naming in your `.xdc` file has to match the naming in your VHDL code.

```
entity toplevel is
  port (
    CLKxCI : in std_logic;
    RSTxRI : in std_logic;

    Push0xSI : in std_logic;
    Push1xSI : in std_logic;
    Push2xSI : in std_logic;
    Push3xSI : in std_logic;

    RLEDxSO : out std_logic;
    GLEDxSO : out std_logic
  );
end toplevel;
```

Listing 1: Entity declaration for top-level of of `key_lock_timed.vhdl`.

Task 2: Simulation

Implement a testbench for `toplevel.vhdl`. For inspiration, you can look at previously provided testbenches and the slides. To reduce the simulation time, modify the counter limit so the lock only stays open for a few milliseconds instead of 2 seconds. Remember to change this back before you move to Task 3! **See the hintbox on the next page for help with Vivado simulations.**

Circuit designer's toolbox

Specifying and writing test-cases is a critical step in the verification of your system. Even a simple testbench with a few test-cases that allow you to view the state of the signals in your design can go a long way in helping you figure out if something is wrong.

When writing basic testbenches you can derive test-cases as follows:

- **FSMs:** For an FSM, your testbench should make the FSM go into every possible state as a minimum. Normally, you should also test every possible state-to-state transition.
- **Arithmetic circuits:** Stimulate the circuit with all possible inputs (or a subset selected at random) and check the output in VHDL or save the output for later comparison with a reference model in C, Python, or MATLAB. We will do this for one of the labs related to the final project.

For more details you can refer to Lecture 4 on VHDL for simulation and testbenches. Note that while there are more advanced ways of doing verification beyond simulation, such as formal verification, this is outside the scope of this class.

Hints and common errors

By default, you only see the signals in your top-level entity, but from this it is almost impossible to identify most issues in larger designs. The simulator allows you to check and plot waveforms of any signal in your design as shown in Figure 1. For this, you have to:

1. Set the `Scope` to the entity in which you want to look at signals.
2. Drag and drop signals of interest into the Waveform viewer from the `Objects` tab.
3. You need to restart the simulation to view some signals as they were not saved earlier.

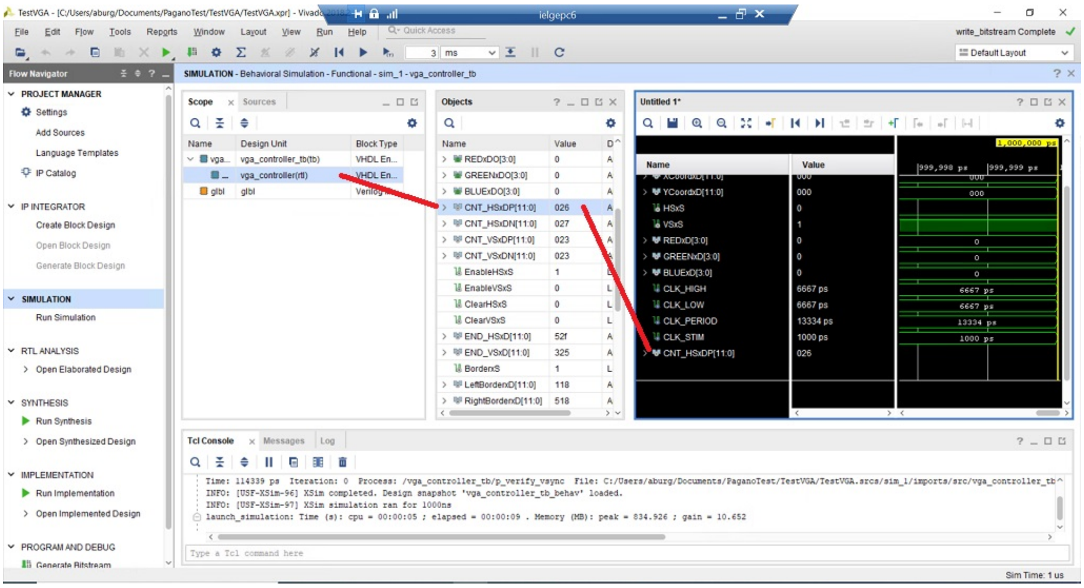

Figure 1: Click on `Scope` to unfold the top-level and see its sub-components. Any signal shown in the `Objects` tab can be dragged over to the Waveform viewer. Note that you have to restart the simulation to view some signals as they were not saved earlier.

Task 3: Implementation

Finally, run the full flow in Vivado to generate the bit-file and program the FPGA. Remember to check for the release of the key in your FSM. Note that sometimes you may fail to type the password as the buttons occasionally bounce and thus you type a key an extra time, but this happens rarely.

Important! For the design to be fully working, it is necessary to connect the FPGA with an Ethernet cable to a computer, router, or any other device with an active Ethernet port. This is because the FPGA clock comes from the Ethernet chip on the board.

Common Questions

Common questions/remarks for this lab are:

- **How do I create a VHDL file?** After creating a project in Vivado you can click `File` → `Add Sources` → `Add or create design sources`. Alternatively, just use your normal code editor and create new files with the `.vhdl` (recommended) or `.vhd` extensions.
- **How do I resolve the warning 'The PS7 cell must be used in this Zynq design ...?** This warning can be safely ignored as it's unrelated to what we do on the FPGA.

- **How do I resolve the error 'Unconstrained Logical Port'?** While VHDL itself is case-insensitive, **`.xdc` constraints are case sensitive** and your port names should match those in the `.xdc` file in case as well.
- **Outlook does not allow `.vhd` files:** You cannot send `.vhd` files in Outlook, try renaming to `.vhdl` as the `.vhd` extension is also used for **V**irtual **H**ard **D**isk on Windows.
- **Remember that order matters in processes!** Since the order of assignments are done sequentially in a process, meaning that in the example below `DxSO` is only assigned `AxSI and BxSI` and never `AxSI or BxSI`.

  Listing 2: This implementation ignores the line `AxSI or BxSI` as it is always overwritten by the final assignment to `DxSO`.

```
process(all)
begin
  if (CxSO = '0') then
    DxSO <= AxSI or BxSI;
  end if;

  CxSO <= not AxSI;
  DxSO <= AxSI and BxSI;
end process;
```

- **Remember to separate the description of the flip-flops from the combinational logic!** Use a single process for updating the flip-flops and a separate process or concurrent assignments for updating the adder as shown below. This is really important! We also discuss this in Exercise 3.

  Listing 3: VHDL code to show how to define flip-flops for a counter.

```
CNTxDN <= CNTxDP + 1; -- Increment outside clock-process

process(CLKxCI, RSTxRI)
begin
  if (RSTxRI = '1') then
    CNTxDP <= (others => '0');
  elsif CLKxCI'event and CLKxCI = '1' then
    CNTxDP <= CNTxDN;
  end if;
end process;
```