LED Color Controller FPGA Implementation

The aim of this lab is to implement in VHDL the PWM circuit that you have designed on paper in Exercise 2 and to test it on the FPGA board. It is strongly recommended that you finish Exercise 2 and Lab 1 before starting this lab.

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!

FPGA:

- Remember to use the reset button/switch on the FPGA to reset your design and to turn this off again if using a switch.
- Connect the FPGA board's Ethernet port 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.

VHDL:

- 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 solution for further explanation.
- Never write to the same signal in multiple concurrent statements! This means that if you assign to a signal in a process that signal can only be assigned to in that process and nowhere else. The only place you are allowed to assign multiple times to a signal is inside the (single) process where it is assigned to.

Virtual machines:

- 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.

Windows users:

- Avoid spaces and special characters in your filepaths! Vivado projects can become corrupted if you have spaces and special characters in your filepaths.
- You may have to disable your antivirus tool before running simulations in Vivado.

For common questions/hints to this lab, please see the last page of this document which contains various hints and best-practices.

VHDL Coding of The Circuit

Download the handout `.zip` file from moodle. The file contains the following files:

- `pwm.vhdl`: Template for the circuit that generates the PWM pulse to control each color.
- `toplevel.vhdl`: Template for the top-level that instantiates three PWM circuits to control the three colors of the RGB LED.
- `pwm_tb.vhdl`: The testbench which automatically checks the PWM pulse width.

Task 1: PWM Pulse Generator Implementation

Implement the PWM pulse generation from Exercise 2 in `pwm.vhdl`. The input `PushxSI` is from a push button and the output `LedxSO` controls an LED color. Do not forget to stick to your block diagrams! The counter for the PWM and the threshold should be 20-bits and you should increment the threshold counter by $2^{17} = 131072$ every time the button is pressed.

Circuit designer's toolbox

In Exercise 1b and 2 we worked on understanding the problem at hand that we are solving and devising a plan. In this lab, we are now carrying out this plan by implementing. In general, problem solving can be divided into 4 steps:

1. **Understanding the problem:** This step can involve writing out the ports and their purpose, making a timing diagram, etc. Anything that furthers your understanding of the problem. Many students are halfway through writing code before discovering they do not really understand the problem and then have to start over.
2. **Devising a plan:** In circuit design, this means drawing a schematic (before writing the code!). Note that this 2nd step and the 1st step have some overlap.
3. **Carrying out the plan:** This step involves writing the actual VHDL code that implements the circuit and verifying its correctness.
4. **Looking back:** Think about how you solved the problem, could you have done anything simpler? How did others solve the problem? In this class we will do this together by looking at some of the solutions handed in.

While the process of problem solving may seem trivial to some of you, we find that many students have some bad habits in this area that should be unlearned.

Task 2: PWM Pulse Generator Verification

You can verify you implementation for the PWM pulse generation using the provided testbench `pwm_tb.vhdl`, which is added in the same way as the testbench from the first lab session. Please make sure that only the `pwm_tb.vhdl` file is listed as a top-level file for simulation.

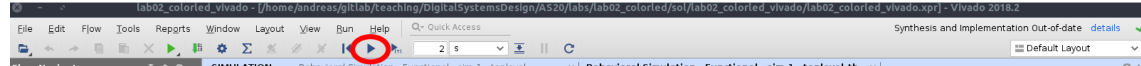

Figure 1: Press the **Run All** button at the top of the Vivado window to run a full simulation.

You must press **Run All** at the top of Vivado, as shown in Figure 1, to run the full simulation. The testbench verifies the pulse width and prints the result to the Tcl Console, as shown in Figure 2. However, you should still look at the waveforms in the waveform viewer to verify that everything is correct. See the hintbox on the next page for help with Vivado simulations.

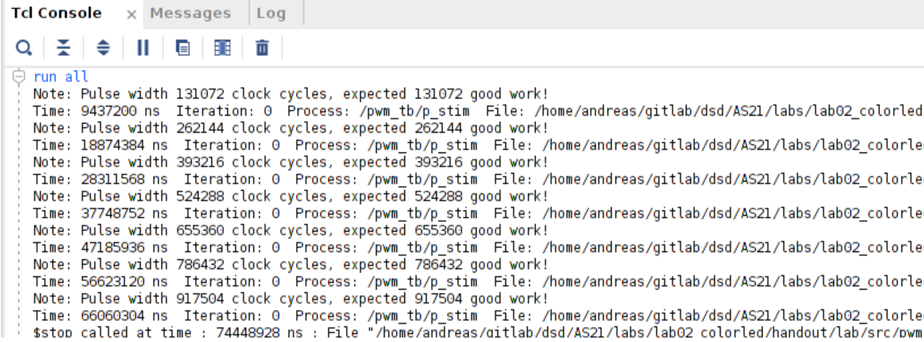

Figure 2: Tcl console with print out from the testbench. The testbench checks the width of your PWM pulses and reports the result. Pulse widths should be integer multiples of $2^{17} = 131072$.

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 3. 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.

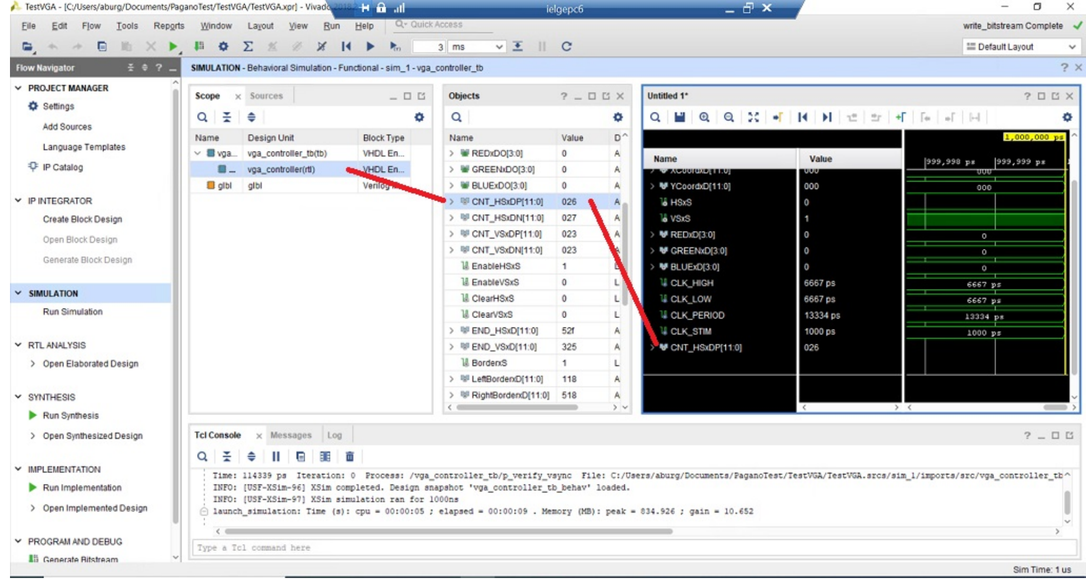

Figure 3: Click on `Scope` to unfold the top-level and see its sub-components. Any signal shown in the `Objects` tab can be dragged over to the Waveform viewer. Note that you have to restart the simulation to view some signals as they were not saved earlier.

Task 3: RGB PWM Pulse Controller

Implement the RGB PWM pulse controller inside `toplevel.vhdl` by instantiating the `pwm` component three times for each color. The component declaration is already included and you only need to assign the corresponding top-level ports to each instance.

Implementation on the Board

Task 4: FPGA Constraint File

Launch Vivado and create a new project using the name of the work directory in the handout. Add the previously described source files to your project. Then, add the `.xdc` file, provided on moodle, to the project. The `.xdc` file is in the `constr` directory as in the first lab. The provided `.xdc` file is the standard template for the board, and you need to correctly connect the buttons, clock and LEDs to your design as follows:

- Connect the clock port of the design to the external board clock source which has a frequency of 125 MHz.
- Connect the reset port of the design to BTN 0 on the board.
- Connect the RGB color controller inputs to BTN 1-3 on the board, with `PushRedxSI` connected to BTN 3, `PushGreenxSI` to BTN 2, and `PushBluexSI` to BTN 1.
- Connect the RGB output color to LED4 on the board. Note that this LED has 3 inputs as it is an RGB LED.

You can look at the `.xdc` file from Lab 1 better understand what changes to make when connecting your signals to the pins.

Task 5: FPGA Programming

Run synthesis, implementation and generate the bitstream. Program the FPGA with the bitstream and try the followings to verify your design:

- Press BTN 0 to reset all the counter values. The PWM pulses have duty cycle of 0 and the LED should be turned off (black).
- Press BTN 3 to increase the intensity of the red color. Press reset again and test the two other colors in a similar way.
- Try to generate other colors by mixing RGB according to their color codes.

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. Without this, you will see some flickering of the LED, even if your design is correct, as the clock will periodically stop. **For this lab, it is okay to NOT use an Ethernet cable, but it will be needed in future labs. The result of not using an Ethernet cable in this lab is a little pulsing of the LED.**

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 1: 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 2: 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;
```

- **Remember to never write to the same signal in multiple concurrent statements!** When assigning to a signal in a process, you can only assign to that signal in that (single) process. The code below in Listing 3 shows the signal `CNTxDN` being assigned to in two different concurrent statements, which is not allowed. With this code, you will see an 'X' for `CNTxDN` in the waveform viewer. The solution is to put the default assignment in a process like shown in Listing 4.

Listing 3: VHDL code which shows how not to assign to a signal!

```
CNTxDN <= CNTxDP;

process(all)
begin
  if (In0xSI = '1') then
    CNTxDN <= CNTxDP + 1;
  end if;
end process;
```

- **Use default values in your process!** To avoid introducing errors in your code from missing assignments to signals, you should always use a default value for all signals assigned to in a `process(all)` when describing combinational logic as shown in Listing 4. This is done as the first thing in a process.

Listing 4: VHDL code which shows the assignment of a default value.

```
process(all)
begin
  -- Default values
  CNTxDN <= CNTxDP;
  AxD    <= (others => '0');
  BxD    <= (others => '0');

  -- Actual logic after default values
  if (In0xSI = '1') then
    CNTxDN <= CNTxDP + 1;
    AxD    <= In1xSI;
  elsif (In1xSI = '1') then
    CNTxDN <= CNTxDP - 1;
    BxD    <= In1xSI;
  end if;
end process;
```