Lab 1. Combinatorial logic and FPGA design in Vivado

http://www.sysf.physto.se/~silver/digsyst/lab1.html

Task:

In this lab exercise you will

- Learn to design, simulate and implement designs in Xilinx Vivado
- Design and simulate a 1-bit full adder
- Design and simulate a parallel 4-bit adder using four 1-bit full_adders.

Introductory steps:

In your home directory, create a vhdl folder (for example ~/vhdl). You can either do this from the desktop using the graphic interface, or from the command line. For the latter, open a terminal session and type:

```
mkdir vhdl
```

There should be no spaces in the path names.

In your vhdl folder, create a subdirectory for this lab exercise (e.g. ~/vhdl/lab1). To do this in the command line, type:

cd vhdl (to navigate to the vhdl subdirectory)
mkdir lab1

Task 1: Create a new project:

Set up and open the Xilinx Vivado design suite. For this you will need to open a terminal (if you have not already done so). From the command line enter start_vivado.

After Vivado has been opened, create a new project:

- From the opening window, click 'Create New Project...' to bring up a new project wizard.
- In the wizard, give the project a new name (e.g. Lab1), and point the project location and workind directory to the 'lab1' subdirectory that you just created. If yo do this, you don't need to create a project subdirectory
- Create an RTL project, because you are starting the project from the beginning and creating/adding your own sources.
- You don't need to add or create sources in this wizard, but do specify VHDL for both the target and simulator languages
- Similarly, it is not necessary to add intellectual property cores (IP) or user constraints. These can be added later.
- When asked to choose a default Xilinx part, you need to select the FPGA that the design is targeted for. For the FPGA on the Digilent board you are using:

  Product category: General purpose
  Family: Artix-7
  Package: cpg236 (236-pin ball-grid array)
  Speed grade: -1
  Temperature grade: C

  There will be three remaining devices displayed. Choose part: xc7a35tcpg236-1.

- When you reach the end of the wizard, review the project summary and click 'Finish' to open the new project.

Task 2: Create an AND gate:

The Vivado IDE will open, with the Project Manager for Lab 1 in the main window, and the Flow Navigator on the left. You can use the Flow Navigator to manage settings and execute the different steps for designing, simulation and implementation of your project.

In the Flow Navigator, begin by launching the "Add Sources" wizard under Project Manager.

Choose "Add or create design sources" and click "Next". You will then have the option to add files or directories, or create a new file. Choose "Create File". In the pop-up:

- Select source type 'VHDL Module'.
- Give the file a suitablename (for examplemy_and).
- Set the file location "Local to Project" (default) and click "OK".

After clicking "Finish" you will be prompted to define the module properties:

- Default names are given for the entity and architecture names.
- Define the ports. In our case, specify two "in" ports (a,b) and one "out" port (q).
- Review your choices and finish.

If you have done everything correctly, you will see a top-level VHDL design (my_and) in your Design Sources hierarchy. You can open and view the VHDL source, which will contain an entity declaration and an empty architecture. Make sure that the entity statement has correctly-declared input and output ports (a,b,q).

The next step is to write the architecture, describing the AND gate functionality. Use the native VHDL **and** operator. Save when you are finished.

Task 3: Simulate your AND gate:

The next step is to simulate your design file using a VHDL test bench. You can download a blank template at this link and save it to your local project directory under a convenient name (for instance, lab1_tb.vhd).

In your "Sources" window you will notice that your my_and design file is listed under both "Design Sources" and "Simulation Sources". Right-click "Simulation Sources" and select "Add Sources", which will launch a wizard that will allow you to add the test bench as a simulation source.

When you have imported the test bench file to your project, open it in the editor.

Declare your my_and module as a component, and declare "a" "b" and "q" as std_logic signals. Also, instantiate your my_and component as the "unit under test" (UUT), connecting signals a,b and q with the appropriate ports.

In the stimulus process, you need to provide different inputs to my_and to test the behavior. A simple stimulus code might look something like this:

```
a <= '0';
b <= '0';
wait for 10 ns;
a <= '1';
wait for 10 ns;
a <= '0';
b <= '1';
wait for 10 ns;
a <= '1';
wait;
```

There are no clocks in your design, but the test bench template includes a simple clock stimulus process anyway. You may choose to ignore or remove this code.

After saving and checking that there are no syntax errors, you can now run the simulation. In the Flow Navigator, locate and launch "Run Simulation" (choosing "Behavioural" simulation).

The simulation will compile and run, and XSim will open a new simulation view. Here you can see the design hierarchy, visible signals, and a waveform viewer.

Zoom out and/or scroll through the waveform window to check that your design works correctly.

When you are satisfied with your design, it is time to implement your AND gate in the FPGA.

Task 4: Assign user constraints to your design:

Under "RTL Analysis" in the Flow Navigator, launch "Elaborated Design" to analyze your design so that you can begin constraint assignment and project planning.

In the Schematic tab you can see the RTL schematic of the design you have created (essentially a single AND gate). Above the schematic you will see that the design uses one (logic) cell, three I/O ports and three nets (connections). If you click on the link to the 3 I/O ports, a new window will be launched that will allow you to assign these pins to physical I/O pins on the FPGA.

Expand the list of ports to see the details. The column 'Package Pin' is empty at the beginning; here is where you specify where the I/Os are placed. Look at your Digilent FPGA board to help decide.

For inputs a and b, the switches on the bottom are suitable. You can see that SW0 and SW1 are connected to FPGA pins V17 and V16, respectively. For output q, the LEDs above the row of switches are a good choice. LD0 is connected to FPGA pin U16.

The FPGA I/O pins on your Digilent FPGA board run at 3.3V, so in the I/O Std column, choose LVCMOS33 for all pins in your design.

When you are done, click the "Save" icon at the top of the IDE window to save your design constraints. When prompted to provide a name for your constraint file, choose an appropriate name like Lab1.

Task 5: Implement the design:

There are three main implementation steps:

- Synthesis
- Implementation
- Bitstream generation (for programming)

Launchers for each of these steps are located in the Flow Navigator. You can execute them one at a time, or you can run all in sequence by clicking "Generate Bitstream". If you select a later step in the chain, Vivado will first run all previously uncompleted steps in order.

The final product will be an FPGA configuration file (my_and.bit) that you can download to the FPGA.

Before downloading the file to the board, it may be useful to look at some of the synthesis results. In the flow manager, you will find different schematic types. Look at the following two to see the difference.

- RTL schematic: RTL Analysis > Elaborated Design > Schematic
- Technology schematic: Synthesis > Synthesized Design > Schematic

Examine the design summary as well to see how many resources were used:

- Technology schematic: Synthesis > Synthesized Design > Report Utilization

Task 5: Download and test the design:

Connect the FPGA board to a USB port on your computer with the provided cable and turn on power. Make sure the jumper JP1 (next to the large USB port) is connected across the center two pins to select JTAG programming mode.

Double-click on "Open Hardware Manager" in the Flow Navigator. Near the top there will be an "Open Target" link, and a pop-up menu will let you select "Auto Connect". In the Hardware window you will then see a programming chain starting with the computer you are on (localhost), followed by the programming device (Digilent/serial #), and then the FPGA (xc7a35t).

Right-click on the FPGA and select "Program Device....". You will see a popup window where you can select the programming (.bit) file, and an optional "debug probes" file (which we will ignore for now).

When you have selected the desired .bit file, click on the "Program" button, and the configuration file will be loaded.

Congratulations! You have completed your first FPGA design. Test it by trying different on/off combinations with the two input switches, and see which ones cause the output to light.

Task 6: Create OR and XOR gates:

Now that you have gone through the complete design process once, create two-input AND and OR components (my_or, my_xor) by copying and modifying your original my_and code. You can use the same test bench to simulate all of the gates at the same time. Simply declare the other components as you did my_and, instantiate all three UUTs with different names (maybe UUT1, UUT2...) and declare and connect separate signals to the different gate output ports (for instance, q_and, q_or, q_or). All three components can share the same inputs a and b.

Your project will have three top-level files now. To implement and test the OR and XOR gates as you did before, in the Project Manager right-click on the entity you want to implmement and choose "Set as Top". Then follow the synthesis and implementation steps you did before.

Download the OR and XOR designs to the FPGA, test again with different input switch combinations, and verify that they work as they should.

Task 7: Design a full adder.

Now that you are familiar with the design process close the existing project and create a new project called 'full-adder'.

Create a new top-level VHDL module with three inputs (a, b, and carry_in) and two outputs (s and carry_out). You also need to create a second VHDL source called 'half-adder', with two inputs (a, b) and two outputs (s, carry). Additionally, you should copy and import your my_and, my_or and my_xor gates from the previous project.

In your 'half-adder' architecture, declare and instantiate your my_and and my_xor components, and use them to produce the outputs "s" and "carry" as covered in lecture. The output s is an exclusive OR (XOR) of the inputs a and b. carry is the AND of the two inputs.

In 'full-adder', declare and instantiate your half-adder twice. You will have to declare some internal signals in this top-level design. You can produce the OR of the carry bits either by instantiating your my_or component, or simply as a VHDL statement.

Create a new test bench and simulate full-adder. Make sure you test all combinations of a, b, and carry_in.

Finally, implement and test your design on the Digilent FPGA developer board. Use three switches as inputs (a, b, carry_in), and two LEDs as outputs (s and carry_out).

Task 8: Design a 4-bit adder from four full-adders.

Create a new project "adder_4b" with top-level ports:

a_in, b_in, s_out : std_logic_vector(3 DOWNTO 0)
carry_in, carry_out : std_logic

Import your full-adder and half-adder VHDL designs to the project. Instantiate four full-adder components and connect them correctly, using internal signals for the carry chain.

Simulate your design. It is not necessary to test all 512 possible input combinations, but testing a few is important. For very complex designs, full simulations become impossible, so the designer must choose tests that are capable of finding potential errors.

Implement the design using eight switches (a_in, b_in) and five LEDs (s_out and carry_out) as inputs and outputs. carry_in can be connected to one of the four input buttons on the side of the switches. Check that the adder works correctly.

Show your work to the instructor when you are finished.

"Extra credit":
If you like, you can extend your parallel adder to an 8-bit version. Can you use generics and generate statements to make an n-bit adder?