Vivado Quick Start Tutorial for the Digilent BASYS-3 board

This tutorial starts with the installation of the Vivado tools, followed by a step-by-step explanation of the implementation of a 16-bit up/down counter in a Xilinx BASYS-3 Starter board from Digilent, using the new Vivado design tools. The up/down counter uses the 16 LEDs of the board as outputs, one of the board slide switches as the up/down selector and another to enable/disable the counter.

1. Installation

Start by accessing the Xilinx site (http://www.xilinx.com/).

- In the menu choose **Products → Developer Zone → Hardware Developer Zone**
- In the new page menu choose **Vivado HLx and ISE Design Suites**
- In the new page choose **Download Vivado Design Suite - HLx Editions**
- In the new **Downloads** page choose the window **Vivado**

Select the most recent version and download the **Vivado HLx 201x.x: : WebPACK and Editions - Windows Self Extracting Web Installer** or **Linux Self Extracting Web Installer**, depending on your operating system.

Execute the downloaded file. You will be asked for your Xilinx user ID and password. If you do not have one, follow the instructions to create it first. It is free!

Then choose your download preference and agree with all Terms and Conditions. Then choose the **Vivado HL Design Edition** and in the next window, under **Design Tools** menu, select “Software Development Kit (SDK)”. Accept the remaining default options.

After installing the software, choose your preferred license option. You may get a free WebPack license from the Xilinx website or ask the Professor for one full license. In this last case, you need to give him the following information:

- Host Name: obtain the Host Name of your machine by running the Manage Xilinx Licenses application on the machine you wish to license (select **View Host Information** under **View System Information** menu)
- Operating System: 32-bit or 64-bit versions of Windows, 32-bit or 64-bit versions of Linux, or Sun Solaris
- Host ID Value: Ethernet MAC Address (notice that the wireless board is not supported) of Windows or Linux machines or Solaris Host ID; notice that several MAC Addresses for different Network Interface Card (NIC) may appear; choose the one belonging to

the Local Area Network adaptor; notice that wireless network adaptors are not supported; in doubt, and if you are running windows, open a DOS shell command window – cmd – and in the command line write **ipconfig /all**; look in the list for the MAC Address of the Local Area Network adaptor.

In case of doubts, watch the **Installation Overview Video** available at the Xilinx **Downloads** page (or look for **Vivado Design Suite Installation Overview** in the Xilinx channel at YouTube).

You may later add new design tools or devices selecting **Add Design Tools or Devices 201x.x** in the Windows **Start** menu or by selecting **Help → Add Design Tools or Devices** in the Vivado menu bar.

2. The Xilinx Documentation Navigator

When installing the **Vivado HL Design Edition** a full installation of the **Documentation Navigator** is also included. This application provides access to Xilinx Technical Documentation and can be run in your machine by selecting **Start → All Programs → Xilinx Design Tools → DocNav** or by selecting **Help → Documentation and Tutorials…** A list of document links to PDF and HTML pages are provided in the main window. Check this list whenever you have a doubt or question about Xilinx tools or hardware.

Now, let's start exploring Vivado tools!

3. The Digilent Vivado Board Files

In our tutorial, we are going to use a prototyping board from Digilent. Vivado needs to know which FPGA the board is using to be able to place and route our design inside the FPGA. These board definitions are not part of the original Vivado installation. Go to http://www.digilentinc.com/ and choose **Documentation**. Look for **FPGA → Software → Vivado**. In the new page, select **Design Resources → Board Files –7-Series**. In the new **Vivado Version 2015.1 and Later Board File Installation** page just follow the instructions.

4. Starting the Vivado Software

To start Vivado, double-click the desktop icon,

or start Vivado from the Start menu by selecting:

Start → All Programs → Xilinx Design Tools → Vivado201x.x

Note: Your start-up path is set during the installation process and may differ from the one above.

5. Accessing Help

At any time during the tutorial, you can access online help for additional information about the Vivado tools by selecting **Help → Documentation and Tutorials….**

6. Create a New Project

Create a new Vivado project targeting the FPGA device on the Basys-3 board from Digilent.

To create a new project:

1. In the **Quick Start** toolbar click **Create Project** or select **File → New Project…** in the Vivado menu bar; the **New Project** wizard appears (Figure 1)

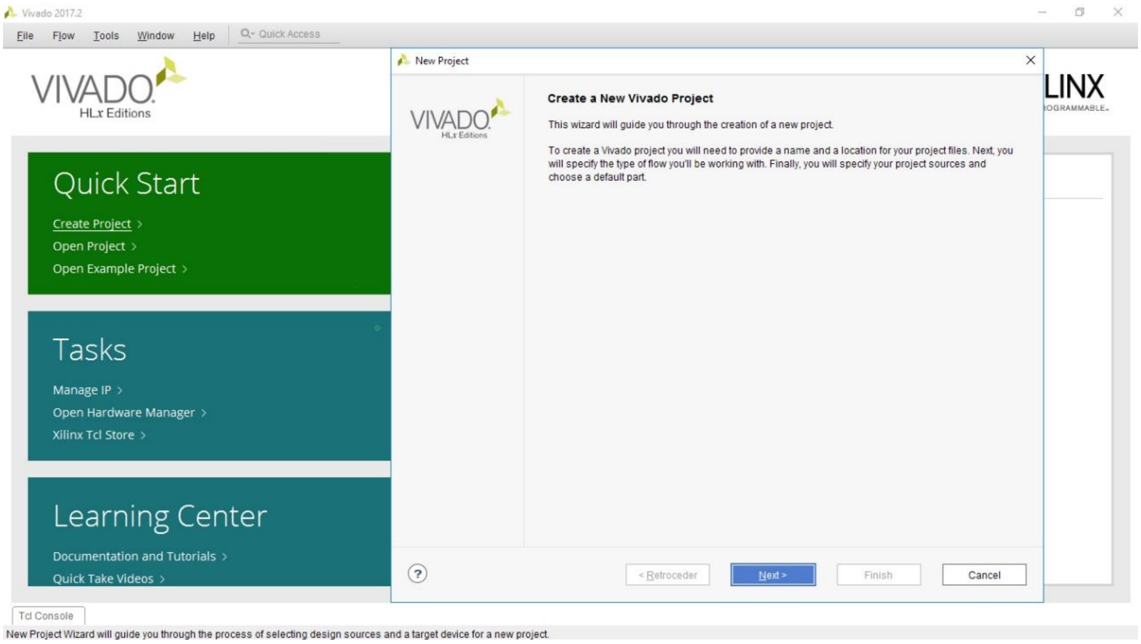

Figure 1

2. Click **Next**
3. Type **tutorial** in the **Project name** field
4. Enter or browse to a location (directory path) for the new project. A tutorial subdirectory is created if the **Create project subdirectory** box is checked

   WARNING: when using any of the Xilinx applications keep your directory path as close as possible to the root, and directory and file names as short as possible! Xilinx applications do not deal well with long directory paths and long directory or file names. Long paths or names usually lead to too hard to detect errors during development. AND IT DOES NOT TOLERATE WHITE SPACES!!!
5. Click **Next**
6. In the **Project Type** window select **RTL Project** and check the **Do not specify sources at this time** box (Figure 2)

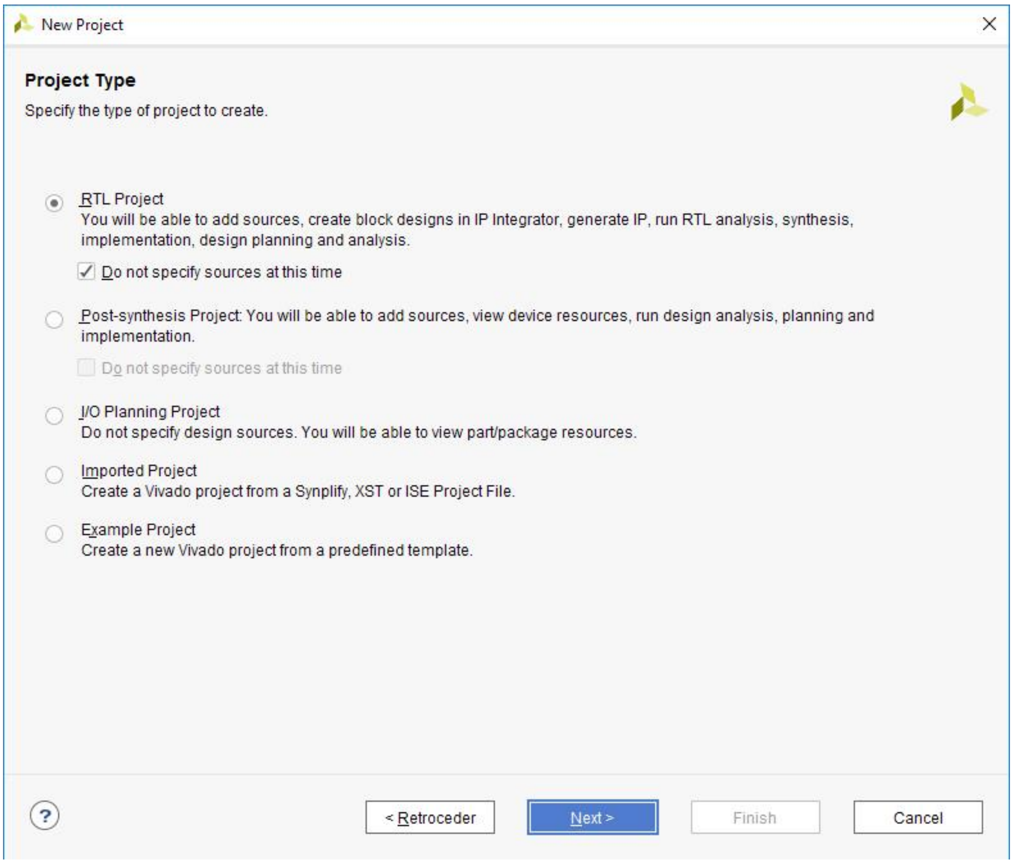

Figure 2

7. In the next window select **Boards**, choose **Vendor: digilentinc.com** and select **Basys3** from the list

   Notice: if the Digilent vendor does not appear in the list, go to section 3.

   Alternatively, we may keep the default selection - Parts – and fill in the Filters using the following settings:

   - Product Category: General Purpose
   - Family: Artix-7
   - Package: cpg236
   - Speed Grade: -1
   - Temp grade: All Remaining

   From the remaining parts on the list, select **xc7a35tcpg236-1**

8. Click **Next** to view the **New Project Summary**
9. Check if everything is correct and then click **Finish**

The **Project Manager**, where we manage all our project, opens.

7. Create a VHDL Source

In this section, we learn how to create a VHDL design source.

Create a VHDL source file for the project as follows:

1. In the **Flow Navigator** window, in the left side of the **Project Manager** main window, in the **Project Manager** section**,** click **Settings**
2. In the **Project Settings** menu of the new **Settings** window, click **General**
3. Choose **VHDL** as **Target language** and click **OK** (Figure 3)

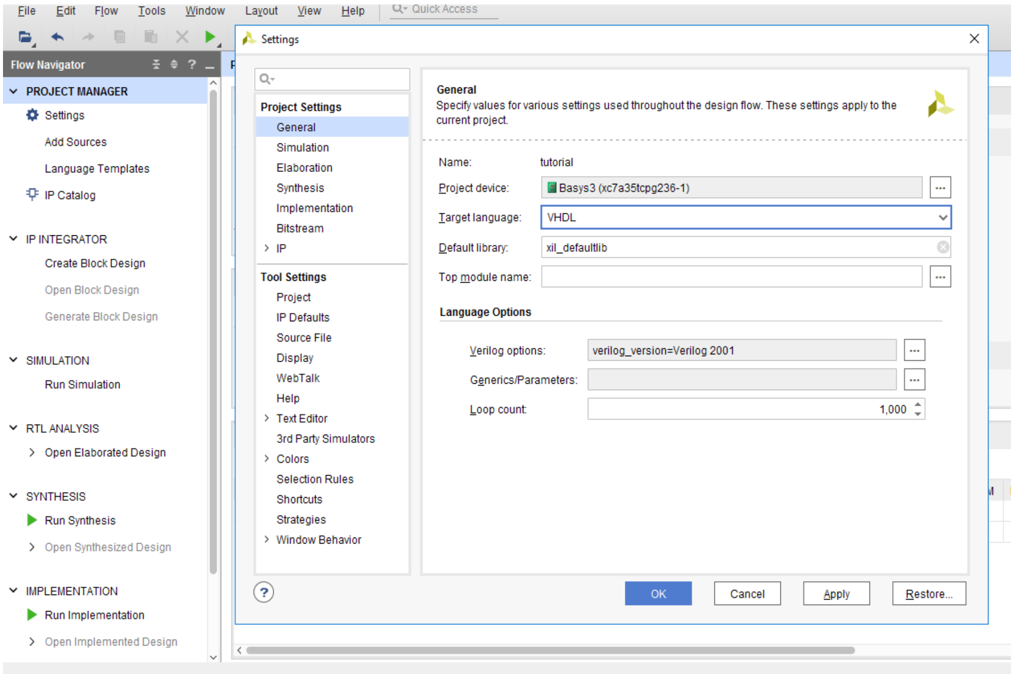

Figure 3

4. Again in the **Flow Navigator** window, click **Add Sources**
5. Select **Add or create design sources** (Figure 4)

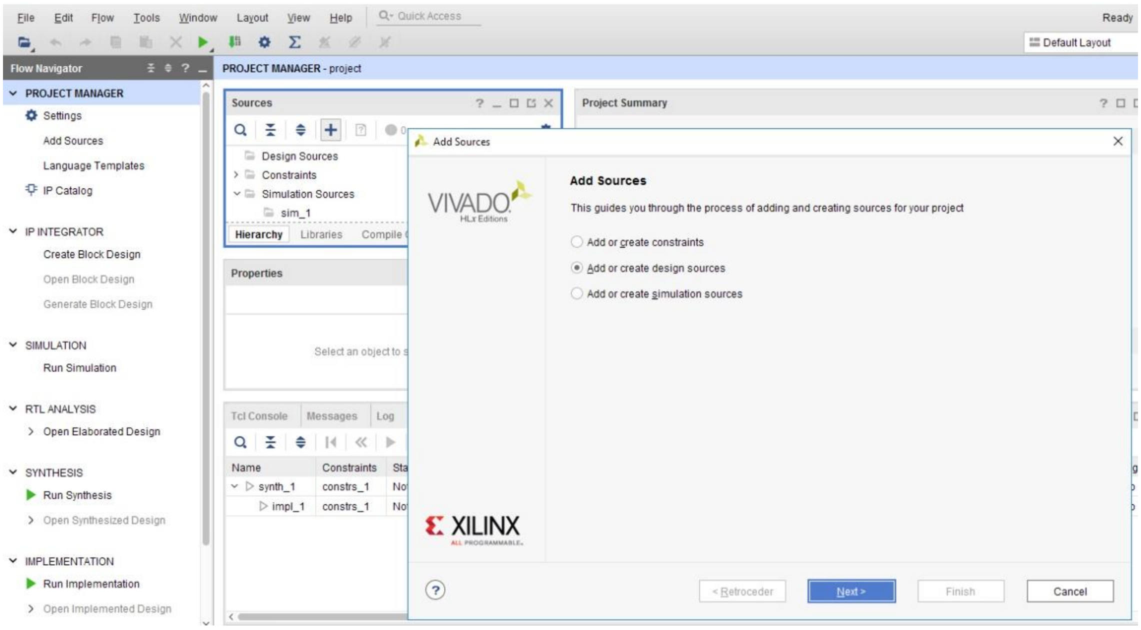

Figure 4

6. Click **Next**
7. In the new window click **Create File**
8. In the **Create Source File** window:
    - File type: **VHDL**
    - File name: **counter**
    - File location: **<Local to Project>**
9. Click **OK**

   A new design source file **counter.vhd** is added to the project
10. Click **Finish**
11. A new window opens, where we may define our module I/O ports

    Our up/down counter needs as module inputs a clock to run, a reset to initialize its register, an enable signal to enable the increment of the counter and a switch to select the counter direction – up or down. The 16-bit output of our counter is connected to the 16 LEDs available on our board. In this way, we have a visual output of the counter behavior. These input and output ports may be declared directly into the VHDL file or may be declared using the **Define Module** wizard.
12. Declare the **counter** design ports by filling in the port information as shown in Figure 5

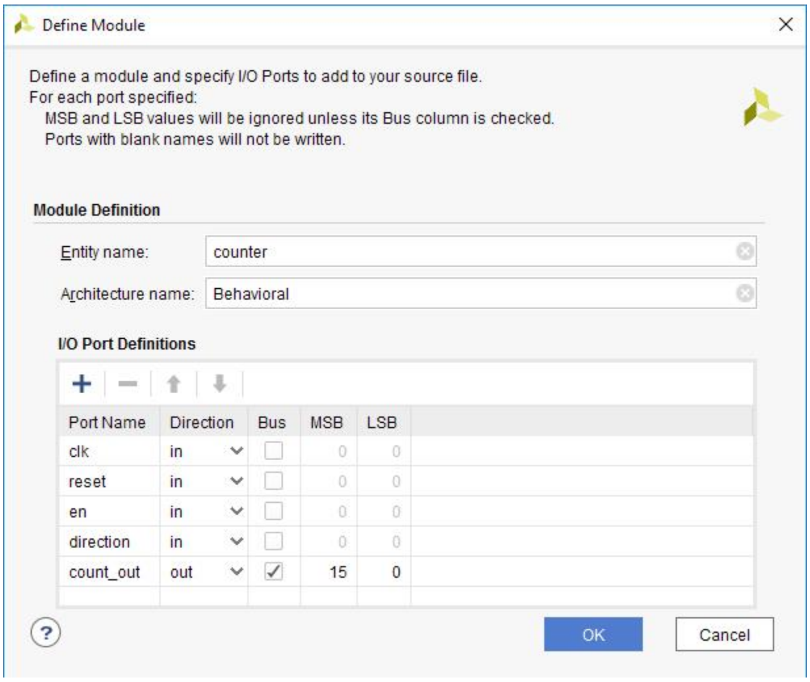

Figure 5

13. Click **OK**

    Our VHDL design source file, which contains our VHDL entity/architecture pair, is now added to the **Sources** window in our **Project Manager** (Figure 6)

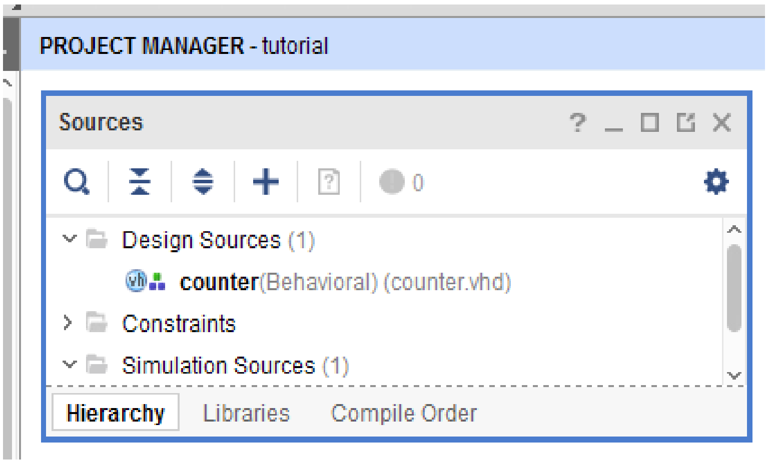

Figure 6

8. Using Language Templates (VHDL)

The next step in creating the new source is to add the behavioral description to the counter design source. To do this we use a simple counter code example from the Vivado Language Templates and customize it according to our counter design.

1. Double click **counter – Behavioral (counter.vhd)** in the **Sources** window
2. The **counter** VHDL design source opens in the **Project Manager** window
3. Place the cursor just below the **begin** statement within the counter architecture
4. Open the **Language Templates** by selecting in the menu bar **Tools** → **Language Templates** or by selecting the **Language Templates** icon in the horizontal toolbar of the **counter** design source window (Figure 7)

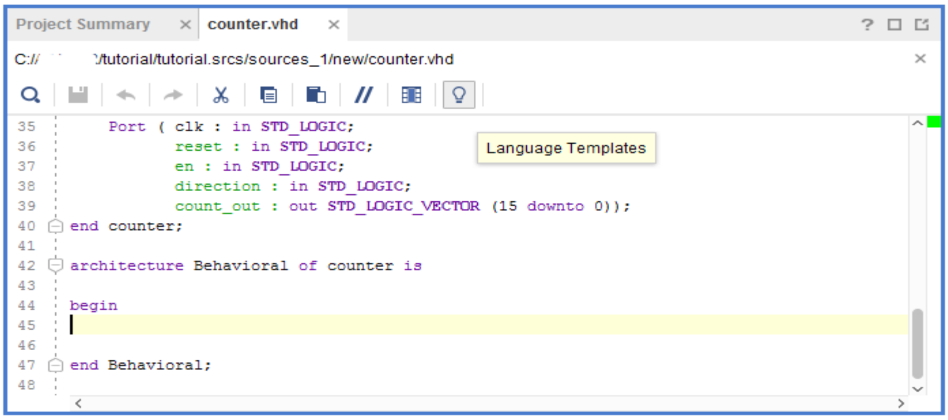

Figure 7

5. Using the **>** symbol, browse to the following code example:

   VHDL → Synthesis Constructs → Coding Examples → Counters → Binary → Up/Down Counters and select **/w CE and Sync Active High Reset**

6. To copy and paste the template to our design source, select the **Preview** window, right click to open a drop-down menu and select **Select All** followed by **Copy** (Figure 8) and paste the template into the **counter** design source

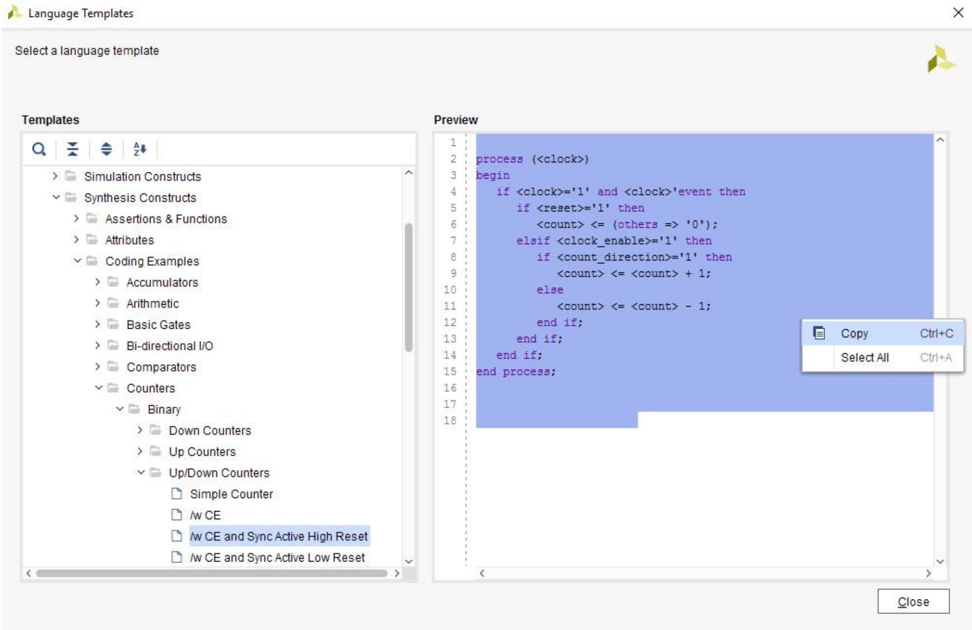

Figure 8

7. Close the **Language Templates** window
8. Go to the **counter.vhd** design source window, right click to open a drop-down menu and select **Paste**

9. Final Editing of the VHDL Source

1. In the **counter** VHDL design source uncomment the library declaration `use IEEE.NUMERIC_STD.ALL;`
2. Add the following signal declaration to handle the feedback of the counter output below the **architecture** declaration and above the first **begin** statement:

```
signal count_int : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
```

This statement creates a 16-bit register and initializes it to zero.

3. Customize the source file for the counter design by replacing the port and signal name placeholders with the actual ones as follows:
   - replace all occurrences of <clock> with clk
   - replace all occurrences of <reset> with reset
   4. replace all occurrences of <clock_enable> with en
   - replace all occurrences of <count_direction> with direction
   - replace all occurrences of <count> with count_int

5. Convert logic to arithmetic signals (`SIGNED (x)`), to be able to perform arithmetic operations over counter values, and back (`STD_LOGIC_VECTOR (x)`), by adding data type conversion functions:
   - `count_int <= STD_LOGIC_VECTOR (SIGNED (count_int) + 1);`
   - `count_int <= STD_LOGIC_VECTOR (SIGNED (count_int) - 1);`

   Conversions are needed because add operations are not defined in the `IEEE.NUMERIC_STD.ALL` library for `STD_LOGIC_VECTOR` types
6. To connect the counter register outputs to the output port, add the following line below the **end process** statement:

   `count_out <= count_int;`
7. Save the file by selecting **File → Save File** or click in the save file icon in the horizontal toolbar of the **counter** design source file window (Figure 9)

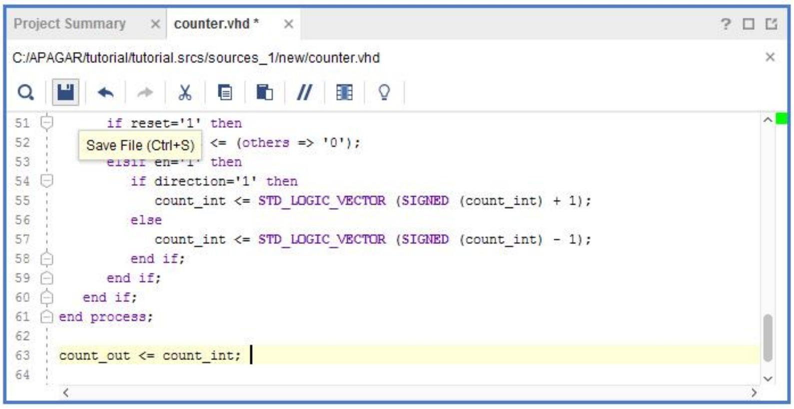

Figure 9

When we finish editing, the **counter** design source looks like the following (all comments were removed from the file):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           en : in STD_LOGIC;
           direction : in STD_LOGIC;
           count_out : out STD_LOGIC_VECTOR (15 downto 0));
end counter;
```

```
architecture Behavioral of counter is

signal count_int : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');

begin

process (clk)
begin
   if clk='1' and clk'event then
      if reset = '1' then
         count_int <= (others => '0');
      elsif en = '1' then
         if direction = '1' then
            count_int <= STD_LOGIC_VECTOR (SIGNED (count_int) + 1);
         else
            count_int <= STD_LOGIC_VECTOR (SIGNED (count_int) - 1);
         end if;
      end if;
   end if;
end process;

count_out <= count_int;

end Behavioral;
```

We have now created the VHDL source for the tutorial project. Add the comments you wish to explain the code behavior.

10. Improving Code Type Productivity

To help you write the code, you may configure Vivado to display a list of code completion suggestions.

You may configure your preferences for activating the code completion drop-down, choosing to use a shortcut key, to display as you type, or to disable code completion if you do not want to use it. You can also set whether to use the Tab key or Space bar to select the displayed value.

To configure this feature:

1. In the **Flow Navigator** window, on the left side of the **Project Manager** main window, in the **Project Manager** section, click **Settings**
2. In the **Project Settings** menu of the new **Settings** window, in the **Tool Settings** section and choose **Text Editor**
3. Select **Code Completion** to see the full set of text editor settings available

11. Checking the Syntax of the New counter.vhd Module and generating the RTL schematic

While saving the file, Vivado automatically checks the design for syntax errors and typos. If a critical error is detected, the faulty design source file appears in the **Sources** window under a **Non-module File** branch (Figure 10) (in this case we only have one file but if we had more, only the faulty one(s) would appear in the branch). This means that due to the syntax error the file is no longer recognised as a valid design source.

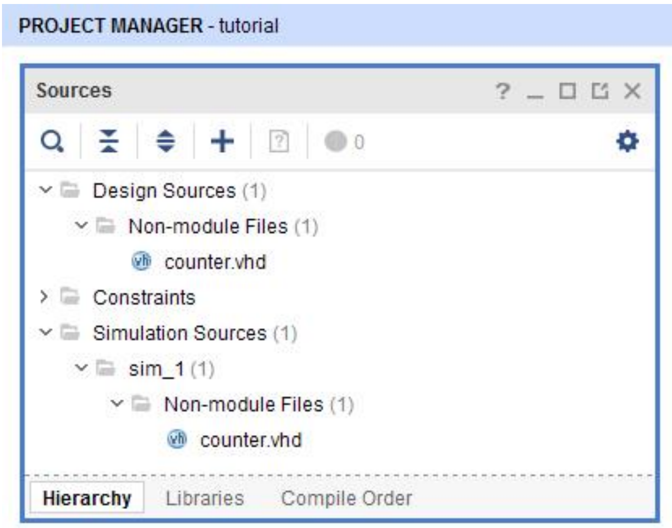

Figure 10

1. Hover the mouse cursor over the highlighted statements in the design file
2. Check the status window message for clues about the current critical error in our design (Figure 11)

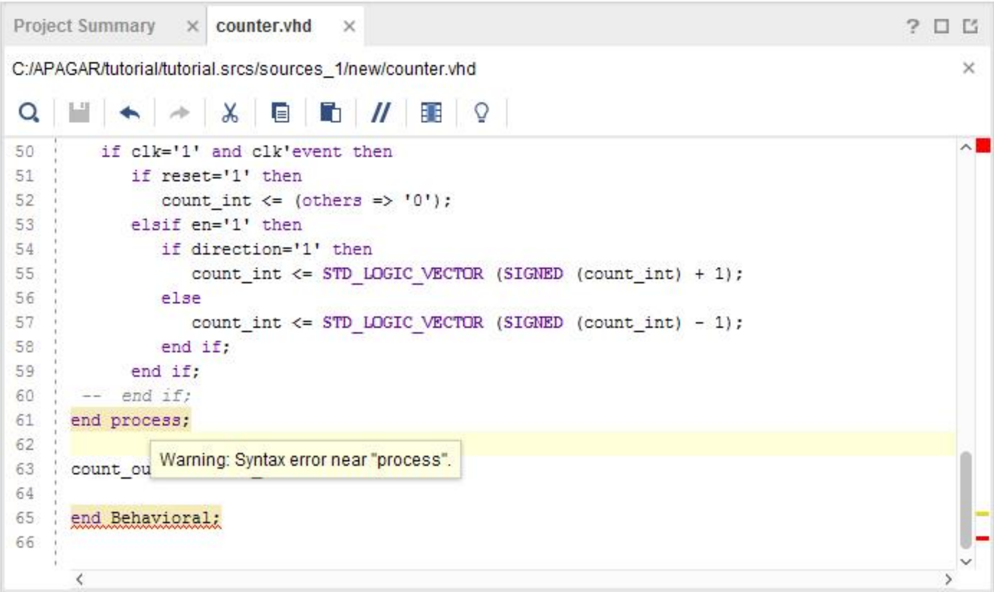

Figure 11

3. If the syntax error does not prevent the design source to be recognised as a valid design, it does not appear under the **Non-module File** branch; still, a syntax error in any line or lines of our VHDL description is underlined with a red line
4. Hover the mouse cursor over the underlined line; a status window error message appears (Figure 12)

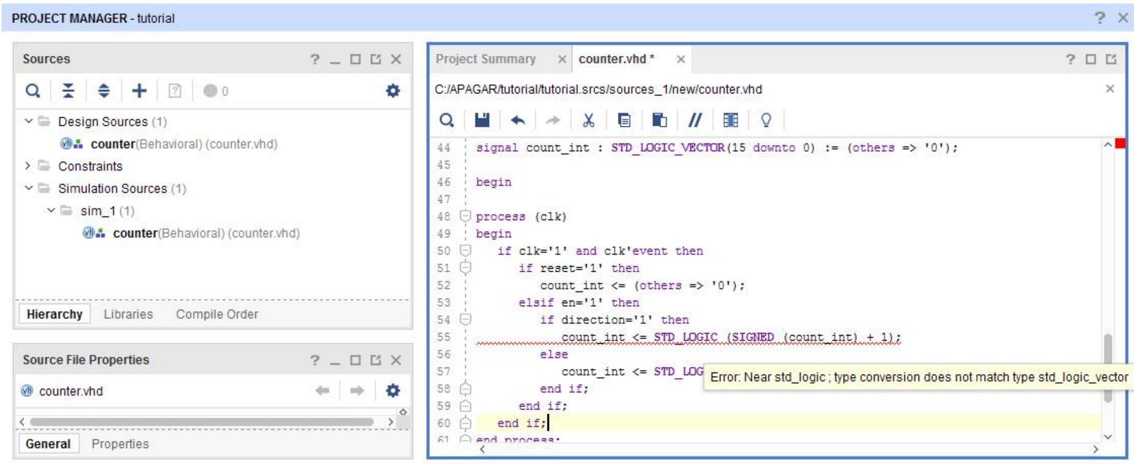

Figure 12

5. Fix any errors and typos before proceeding

However, checking the design for syntax errors and typos is not enough to guarantee that our VHDL description is grammatically correct. For that, we need to parse our design.

6. In the **Flow Navigator** window, in the **RTL Analysis** section, click **Open Elaborated Design**

7. This launches the first part of synthesis (called elaboration), parsing our RTL (Register Transfer Level) design and turning it into a netlist of generic technology cells (abstractions of AND, OR, MUX, ADDER, COMPARATOR, etc... cells)
8. If there is a syntax error in our RTL, the parsing fails, and we get error messages telling us what is wrong (Figure 13)

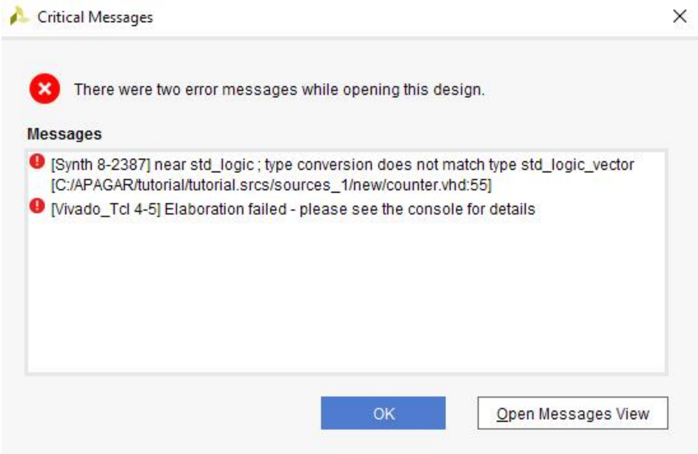

Figure 13

9. If it passes, not only it confirms that our RTL is correct, but we may now view and explore this generic technology representation of our design
10. Figure 14 shows the **Schematic** representation of our description at the RTL level; this representation should open automatically in the **Elaborated Design** window, now the main window of our project; if it does not open, go to the **Flow Navigator** window and in the section **RTL Analysis → Elaborated Design**, click **Schematic**

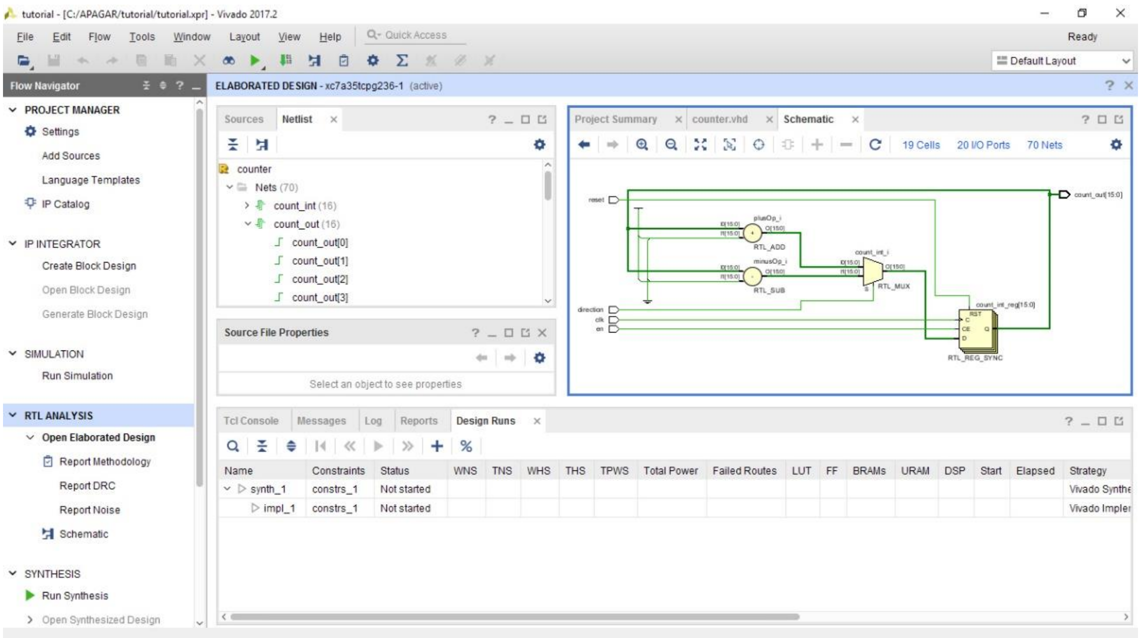

Figure 14

11. In the schematic, when we select one of the logic objects, it is highlighted in the **RTL Netlist** on the left of the **Schematic** window

12. Synthesize the Design

Now that we have created and analyzed the design, the next step is to synthesize the design. During synthesis, the HDL files are translated into gates and optimized for the target architecture.

13. Entering Synthesis Options

Synthesis options enable us to modify the behavior of the synthesis tool to make optimizations according to the needs of our design. One commonly used option is to control synthesis to make optimizations based on area or speed. Other options include controlling the maximum fanout of a flip-flop output or the sharing of arithmetic operators between different signals (check Xilinx's User Guide UG901 for a detailed explanation of each one of the options).

To enter synthesis options:

4. In the **Flow Navigator** window, in the left side of the **Project Manager** main window, in the **Project Manager** section, click **Settings**
5. In the **Project Settings** menu of the new **Settings** window, click **Synthesis**
6. This allows us to view the full set of process properties available (Figure 15)

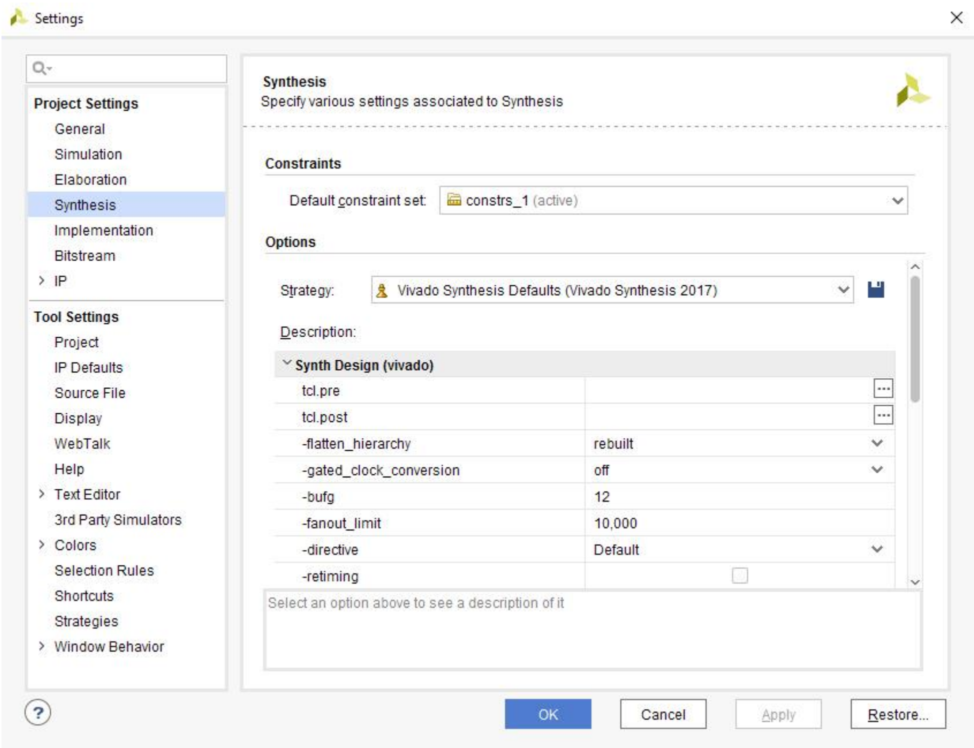

Figure 15

7. In **Options → Strategy** we can view and select from the drop-down menu a predefined synthesis strategy to use for the synthesis run; there are different preconfigured strategies, as shown in Figure 16; in our case, leave default options unchanged
8. Click **OK**

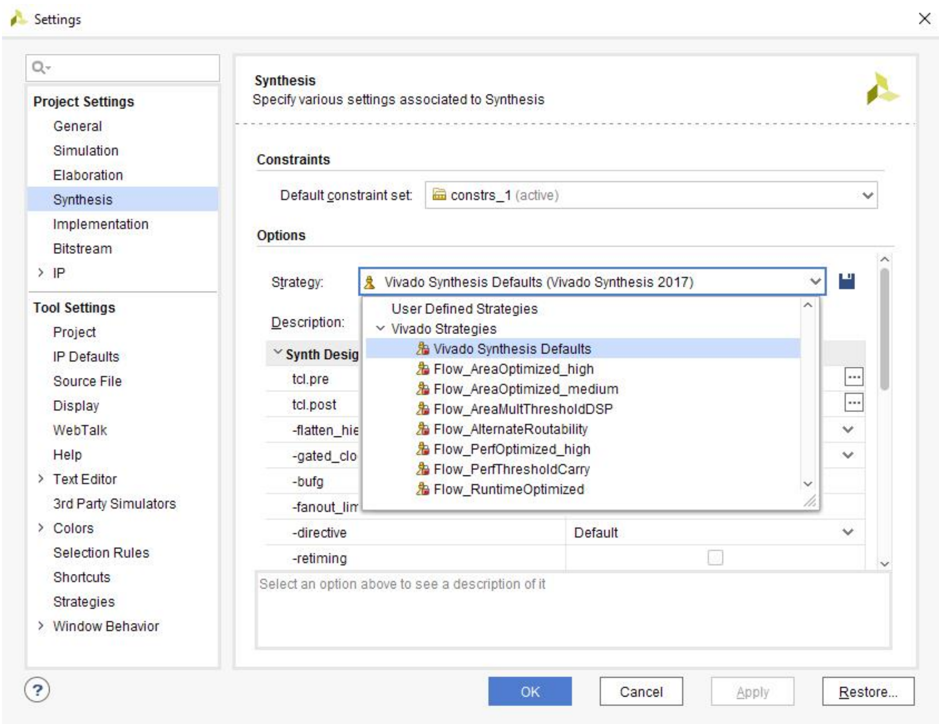

Figure 16

14. Synthesizing the Design

Now we are ready to synthesize our design. To take the VHDL code and generate a compatible netlist:

1. In the **Sources** window, select **counter – Behavioral (counter.vhd**)
2. In the **Flow Navigator** window, in the **Synthesis** section, click **Run Synthesis**
3. Check the **Log** window at the bottom of the main window to follow synthesis evolution (Figure 17)

Figure 17

4. In the end, a new window opens stating that the synthesis was successfully completed and showing the next steps (Figure 18). This type of windows appears after completing each one of the implementation steps. We may disable these windows by checking **Don't show this dialog again**. Notice that the options we see in the window are always available in the **Flow Navigator** window

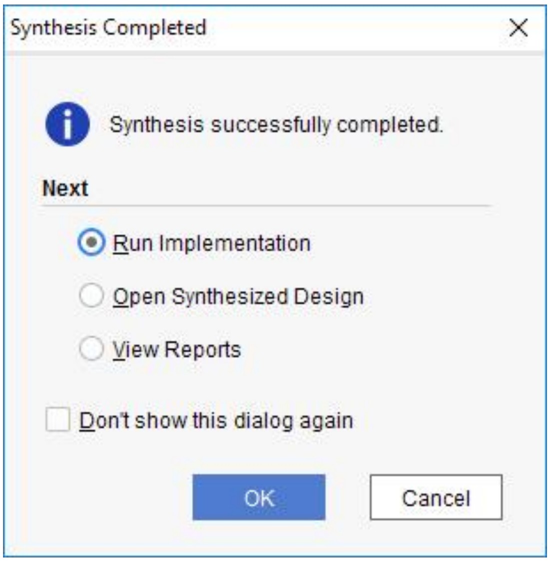

Figure 18

5. In the window choose **Open Synthesized Design** and click **OK**
6. A **Device** window opens in the new **Synthetized Design** window, showing the FPGA resources distribution (Figure 19)(if the **Device** window is not visible, check for the **Device** tab in the new **Synthetized Design** window). However, no resources are yet occupied by our design. This happens only after the **Implementation** step

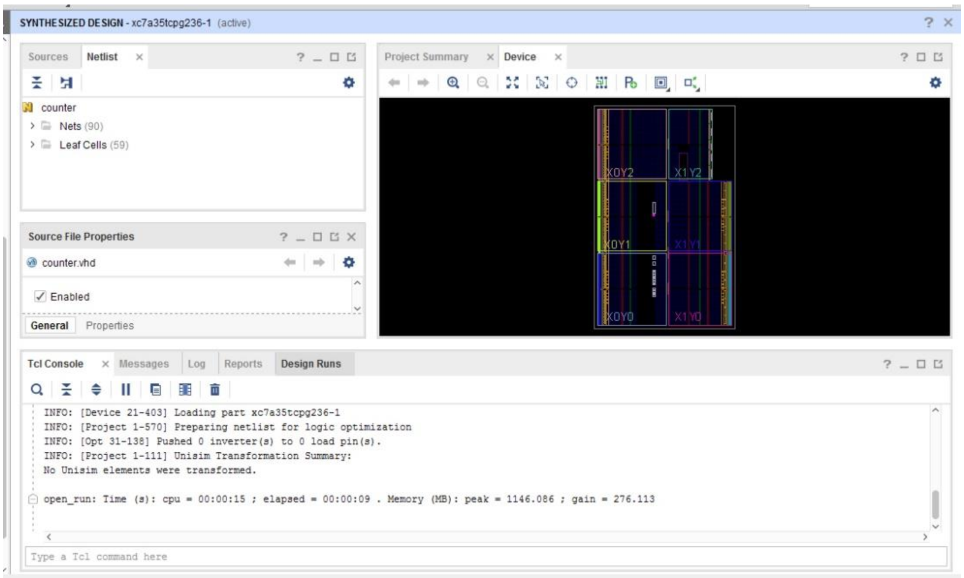

Figure 19

15. The RTL / Technology Viewer

A schematic representation of our VHDL code is generated during synthesis. A schematic view of the code helps us to analyze our design by displaying the graphical connection among the various logic objects that the synthesizer has inferred. There are two forms of schematic representation:

- RTL View - Pre-optimization of the HDL code we saw in section 11
- Technology View - Post-synthesis view of the HDL design mapped to the target technology

To view a schematic representation of our HDL code:

1. In the **Flow Navigator** window, in the **Synthesis** section, click **Open Synthesized Design** to expand the process hierarchy
2. Click **Schematic;** the schematic representation of our design opens in the **Synthetized Design** window, now the main window of our project
3. The **Netlist** window on the left side of the **Synthetized Design** window displays the logic hierarchy of our synthesized design; we can expand and select any logic instance or net within the netlist
4. Selecting a logic object in the **Schematic** or in the **Netlist** window gives us access to its properties; this information is displayed in the **Instance**, **Cell**, **Bus Net** or **Net Properties** (depending on the logic object type) windows, just below the **Netlist** window
5. For example, by selecting one of the Look-Up Tables (LUT) we have access to the truth table it implements; choose the **Truth Table** tab available on the **Cell Properties** window in the left side of our **Schematic** window, just below the **Netlist** window, and we may even edit the LUT equation by clicking on **Edit LUT Equation…** (Figure 20)
6. As we select logic objects in other windows, the **Netlist** window expands automatically to display the selected logic objects
7. The **Vivado Synthesis Report**, where any synthesis errors or warnings are reported, and the **Utilization Report,** containing a list of the FPGA resources used by our counter, are available in the **Reports** tab at the bottom of the **Synthetized Design** window; just double-click over the report name to open it

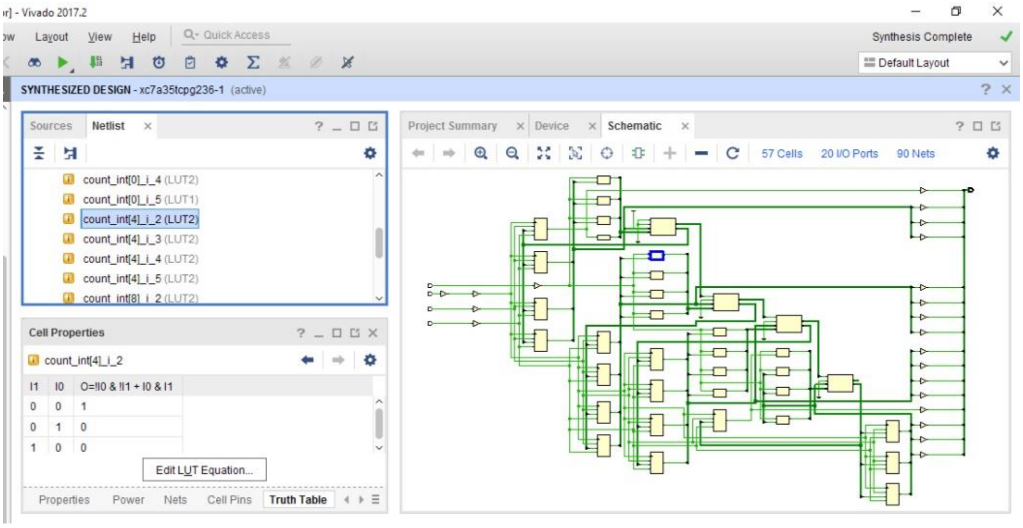

Figure 20

16. Creating a Test Bench File

In order to simulate the design, a test bench file is required to provide stimulus to the design.

To perform simulation, we need to create a test bench file and to add it to our project:

1. In the **Flow Navigator** window, in the left side of the **Project Manager** window, click **Add Sources** or select **File → Add Sources…** in the Vivado menu bar
2. Select **Add or create simulation sources**
3. Click **Next**
4. In the new window click **Create File**
5. In the **Create Source File** window:
    - File type: **VHDL**
    - File name: **counter_tb**
    - File location: **<Local to Project>**
6. Click **OK**

    A new design source file **counter_tb.vhd** is added to the project
7. Click **Finish**
8. A new window opens, where we may define our module I/O ports. Since test benches have no entity ports, click **OK → Yes**
9. The new **counter_tb** simulation file is added to the **Simulation Sources** tree in the **Sources** window (Figure 21)

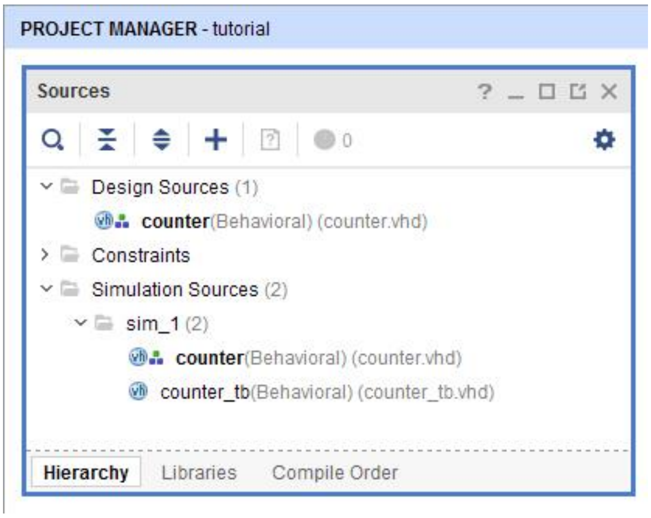

Figure 21

10. The next step is to customize the test bench file in the text editor
11. Double click **counter_tb – Behavioral (counter_tb.vhd)** in the **Sources** window
12. The **counter_tb** VHDL design source opens in the main window
13. Select all the code lines in the design source and delete them
14. Then add the following code lines

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY counter_tb IS
END counter_tb;

ARCHITECTURE behavior OF counter_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT counter
    PORT(
         clk : in  STD_LOGIC;
         reset : in STD_LOGIC;
         en : in STD_LOGIC;
         direction : in STD_LOGIC;
         count_out : out STD_LOGIC_VECTOR(15 downto 0)
        );
    END COMPONENT;

```

```
   --Inputs
   signal clk : STD_LOGIC := '0';
   signal reset : STD_LOGIC := '1';
   signal en : STD_LOGIC := '0';
   signal direction : STD_LOGIC := '0';

   --Outputs
   signal count_out : STD_LOGIC_VECTOR(15 downto 0);

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: counter PORT MAP (
          clk => clk,
          reset => reset,
          en => en,
          direction => direction,
          count_out => count_out
        );

   -- Clock process definitions
   clk_process :process
   begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      reset <= '1';
      en <= '0';
```

```
58          -- hold reset state for 100 ns.
            wait for 100 ns;
60          reset <= '0';

62          wait for 200 ns;
            en <= '1';
64
            -- insert stimulus here
66
            DIRECTION <= '1' after 800 ns;
68
            wait;
70      end process;
    END;
```

15. Save the file by selecting **File → Save File** or click in the save file icon in the corner of the **counter_tb** design source file window
16. While saving the file, Vivado automatically checks the design for syntax errors and typos, in the same way as described in section 11, and automatically associates it to the **uut** (unit under test) **counter**

The VHDL testbench has the same structure as any VHDL design source code. However, there are a few different things that require an explanation. After the library declarations, note that the Entity declaration on lines 4 and 5 is left empty. The Unit Under Test (UUT; or the VHDL code being simulated) is instantiated as a component declaration from lines 11 to 19. Lines 35 to 41 contain the port declaration for the UUT. Lines 51 to 70 define the stimuli generation. Line 71 ends the testbench.

17. Entering Simulation Options

Now that we have a test bench in our project, we can perform behavioral simulation on the design using the XSIM simulator.

To select XSIM as our project simulator and define the simulation runtime we must configure our simulation settings:

1. In the **Flow Navigator** window, in the left side of the **Project Manager** main window, in the **Project Manager** section, click **Settings**
2. In the **Project Settings** menu of the new **Settings** window, click **Simulation**

3. In **Simulation** dialog box, check if the **Vivado Simulator** is selected in the **Target simulator** field (Figure 22) (check in the drop-down menu the list of external simulators that may be used with Vivado)

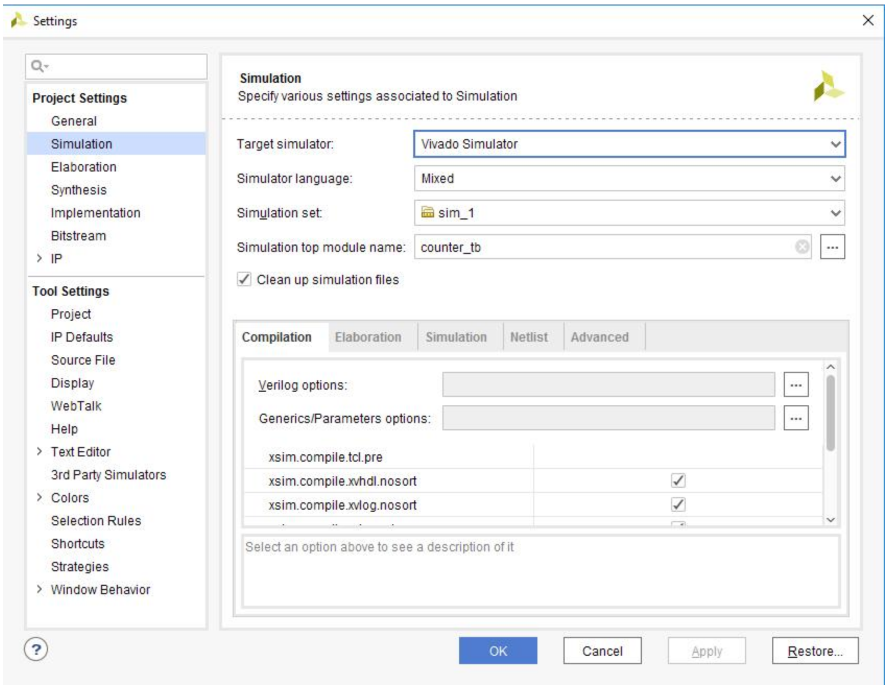

Figure 22

4. Select the **Simulation** tab
5. Change the **xsim.simulate.runtime** parameter to **2000ns**
6. Click **OK**

18. Design Simulation

Simulation may be run at different design stages. Depending on those stages, different kinds of simulation are available – Behavioral, Post-Synthesis Functional, Post-Synthesis Timing, Post-Implementation Functional, and Post-Implementation Timing Simulation. Behavioral Simulation is just a functional representation of our RTL. Post-synthesis Simulation performs the simulation of our synthesized netlist. If there is any optimization we may see differences between Behavioral and Post-Synthesis simulation. This may happen because sometimes synthesis remove unnecessary or equivalent logic from the code so as to improve the QOR (Quality Of Results) and to obtain a more efficient netlist. This process is called trimming and may cause differences in functionality. Post-Implementation simulation is the most important as we get a clear picture about what we are targeting and what exactly is generated. Timing simulation is the closest emulation to actually downloading a design to a device. It allows us to check that the implemented design meets all functional and timing requirements and behaves as we expect in the device. Performing a thorough timing simulation ensures that the finished design is free of defects that can easily be missed, such as dual-port RAM collisions, missing or improperly applied timing constraints or operation of asynchronous paths.

At this step, and since we already ran synthesis, Behavioral, Post-Synthesis Functional and Post-Synthesis Timing simulation are available. Since we did not yet define any timing constraints, let's run a Post-Synthesis Functional Simulation.

1. In the **Flow Navigator** window, in the **Simulation** section, click **Run Simulation**
2. A drop-down menu opens with the currently available simulation possibilities (Figure 23)

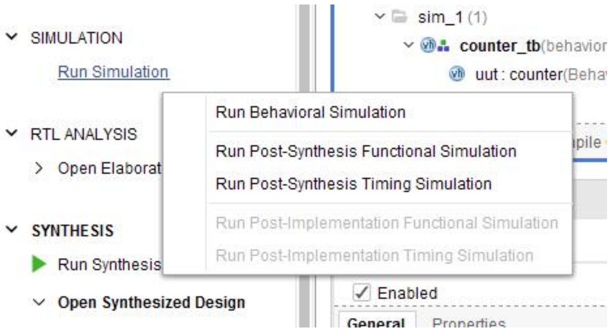

Figure 23

3. Choose **Run Post-Synthesis Functional Simulation**

The test bench and source files are compiled and the Vivado simulator runs (assuming no errors).

19. Simulator Waveform Interface

XSIM creates the work directory, compiles the source files, loads the design, and performs simulation for the time specified. Once simulation finishes we see four main views: **Scope**, where the test bench hierarchy is shown (in collapsed form); **Objects**, where top-level signals are displayed; the waveform window; and the **Tcl Console** where the simulation activities are displayed.

The top-level signals – input and output signals on the module under test – are automatically shown in the waveform window. These signals are displayed in the **Objects** window on the left. We may also add internal signals to our simulation:

1. In the **Scope** window, click **>** next to **counter_tb** to expand the hierarchy
2. Select **uut**
3. A list of other internal signals appears in the **Objects** window
4. Select, drag and drop in the waveform window under the list **Name** the signals you want to add (to select multiple signals, hold down the Ctrl key)

5. To view the new signal waveform, we need to restart simulation clicking on the **Restart** icon (Figure 24) and running simulation again by clicking on the **Run** icon (Figure 25)

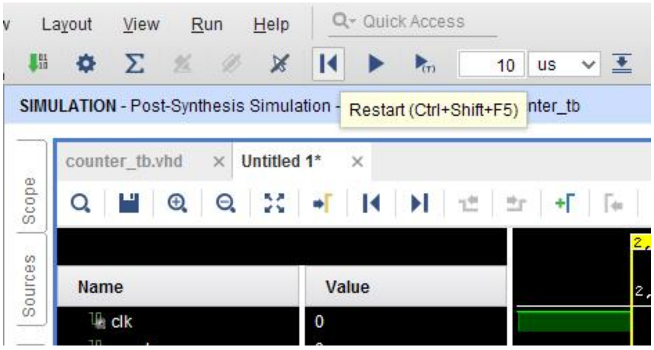

Figure 24

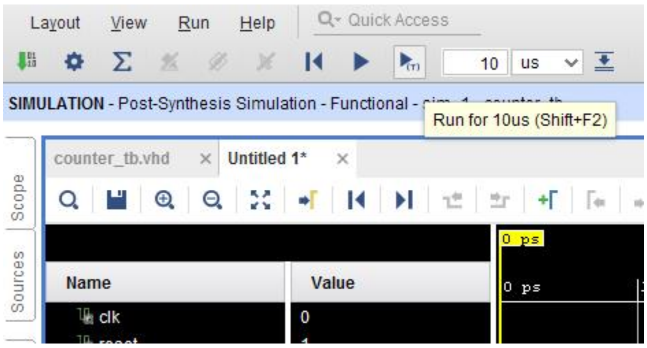

Figure 25

6. Notice that now the simulation runs for 10 μs, the time defined in the window just right of the **Run** icon; we may change the simulation runtime here
7. We may also run simulation until **Break** is pressed (Figure 26) by clicking on the **Run All** icon (Figure 27)
8. To delete signals, select the signal's name in the **Name** list and click DEL
9. By default, bus signals are presented aggregated and with their value represented using a hexadecimal notation. If you want to split the bus signals click **>** next to **count_out[15:0]**

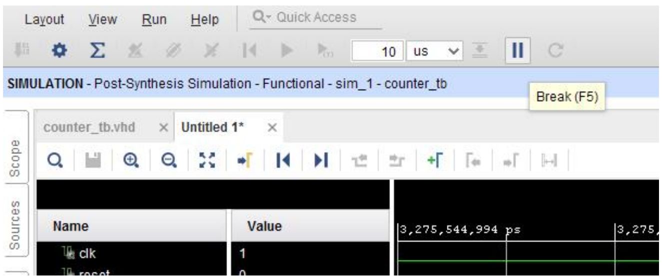

Figure 26

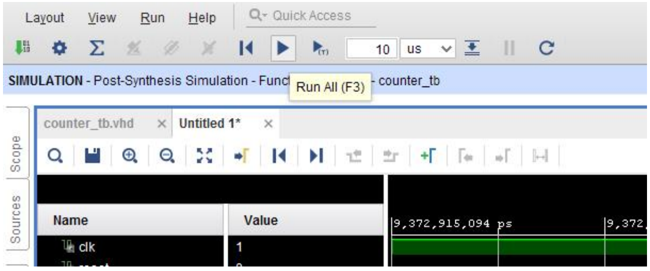

Figure 27

10. If instead of the hexadecimal notation you want to see the value of each one of the signal's vector bits, highlight **count_out[15:0]** to select it and right click to open a drop-down menu. In this menu choose **Radix → Binary** (Figure 28)

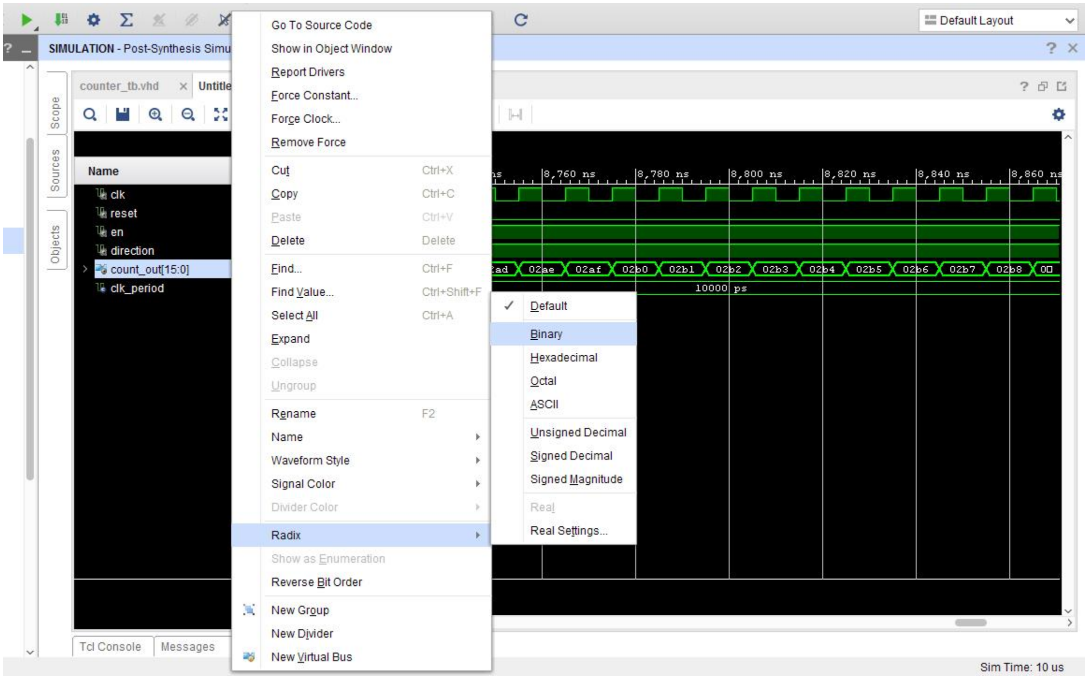

Figure 28

A set of icons in the hprizontal toolbar of the waveform window enables to define several waveform options, to save the waveform, to zoom in, out or fit the waveforms, to jump to the beginning or to the end of the simulation, to jump to the previous or to the next transition of the highlighted signal, to add markers and to jump to the previous or to the next marker.

20. Saving the wave configuration

A wave configuration refers to the customization of the waveform window. It is composed by:

- A list of signals and buses;
- Their properties, such as color, name style and radix value;
- Other wave objects, such as dividers, groups and markers.

You can completely customize a wave configuration by adding or removing signals and other wave objects, and use the wave configuration to examine the simulation results.

A .wcfg (waveform configuration) file saves the current configuration of the waveform window. A wave configuration can have a name or be untitled. The name shows on the title bar of the wave window. To save a wave configuration to a .wcfg file, select **File → Save Waveform Configuration As…** or click in the save file icon in the corner of the wave window.

Be careful, though: when you open a .wcfg file that contains references to HDL objects that are not present in a static simulation HDL design hierarchy, Vivado simulator ignores those HDL objects and omits them from the loaded waveform configuration. This means that if you change the module you are simulating without creating a new wave configuration, the new signals do not show up, since they are not part of the wave configuration you created before.

21. Analyzing the Functionality of our Design

Now the **counter** signals can be analyzed to verify if it works as expected. If not, correct the VHDL description accordingly and rerun the synthesis and simulation steps.

22. Design Implementation

Design Implementation is the process of translating, mapping, placing, routing, and generating a BIT file for our design. The design Implementation tools are embedded in Vivado for easy access and project management.

If you have followed the tutorial, you have created a project, written the source files, and synthesized the design. Now we need to add some physical constraints to our design. Constraints can include physical constraints that assign design logic to device resources, floorplanning and I/O planning constraints that assign logic elements to areas of the device, or timing constraints that characterize the skew and delay restrictions of the design. The order of files in a constraint set is important, as constraints are read in from the files in the order listed. So, be careful, as constraints defined early in the list can be overwritten by constraints defined later in the list. In our case, we are going to add a number of timing constraints related to the operation of our design, and I/O pin constraints related to the input and output FPGA pins we want to use to connect our signals to the board components (switches, buttons, LEDs, etc…).

23. Creating a Constraint File

To create a constraint file in our project:

1. In the **Flow Navigator** window and in the section **Synthesis → Open Synthetized Design**, click **Constraints Wizard**

2. In the new window click **Define Target**
3. In the **Define Constraints and Target** window click **Create File**
4. In the **Create Constraints File** window write **counter** as the File name and click **OK**
5. The newly created file appears in the **Define Constraints and Target** window
6. Check the checkbox under **Target** and click **OK**
7. The constraints file – **counter.xdc** – is added to the hierarchy of our design in the **Sources** window, under the default constraint set – **constrs_1 (1)**

24. Timing Constraints Wizard

To create our timing constraints, we are going to use the **Timing Constraints Wizard** available at Vivado. The Wizard analyzes the design for missing timing constraints and automatically suggests which signals in each situation should be constrained.

1. In the menu bar select the **Tools** menu and choose **Timing → Constraints Wizard…** (Figure 29)

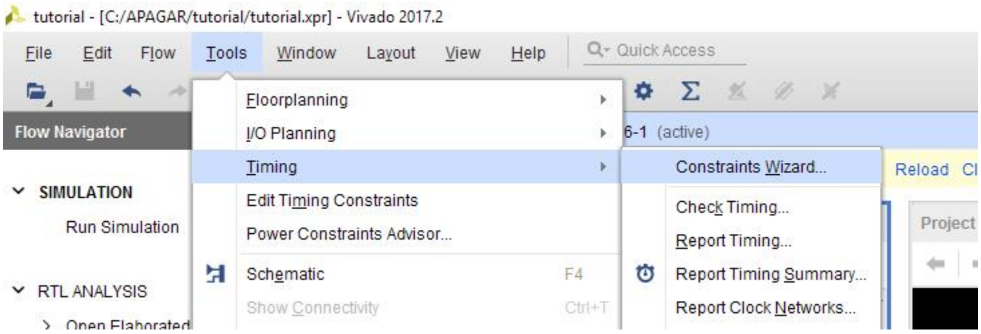

Figure 29

2. A new window opens – the **Timing Constraints Wizard** window. This new window contains a lot of useful information for you to understand how the wizard works and how you may use the information in each one of the following wizard pages to help you create the timing constraints your design needs. So, read it carefully. In the end, click **Next**
3. The first window of the wizard is the **Primary Clocks** window. Notice that there is a brief explanation of what Vivado means by 'Primary Clocks'. To know more about it, you have a **More info** link by the end of the paragraph. Go through it to deep your knowledge about the subject. Once clocks (including generated clocks if they exist) are completely constrained, all paths whose start and end-points are within the design (all register-to-register paths) are automatically constrained
4. Vivado automatically recommends constraining the **clk** signal. Enter the missing frequency value by clicking on the cell that shows 'undefined', and type **100**. Automatically, the

remaining cells – **Period**, **Rise At**, and **Fall At** – are filled in based on a 50% duty-cycle for the clock

5. You may also set **Jitter** to **0.5**

Jitter is defined as the variations in the significant instants of a clock or data signal. It refers to phase variations with respect to a perfect reference that occur in a clock or data signal as a result of noise, patterns, or other causes with a frequency of variation greater than a few tens of Hertz. Slower changes in phase due to temperature, voltage, and other physical changes are usually referred to as wander. In a digital system, a more useful measure is the period jitter. The period jitter is the difference between the longest period and the shortest period. We need to know it to ensure that there is adequate setup time for all of the signals. (from: Austin Lesea, "Jitter: Variations in the Significant Instants of a Clock or Data Signal", *White Paper: Virtex-4, Virtex-5, and Spartan-3 Generation FPGAs*, WP319 (v1.0) March 24, 2008)

6. Check if everything is correct (Figure 30) and click **Next**

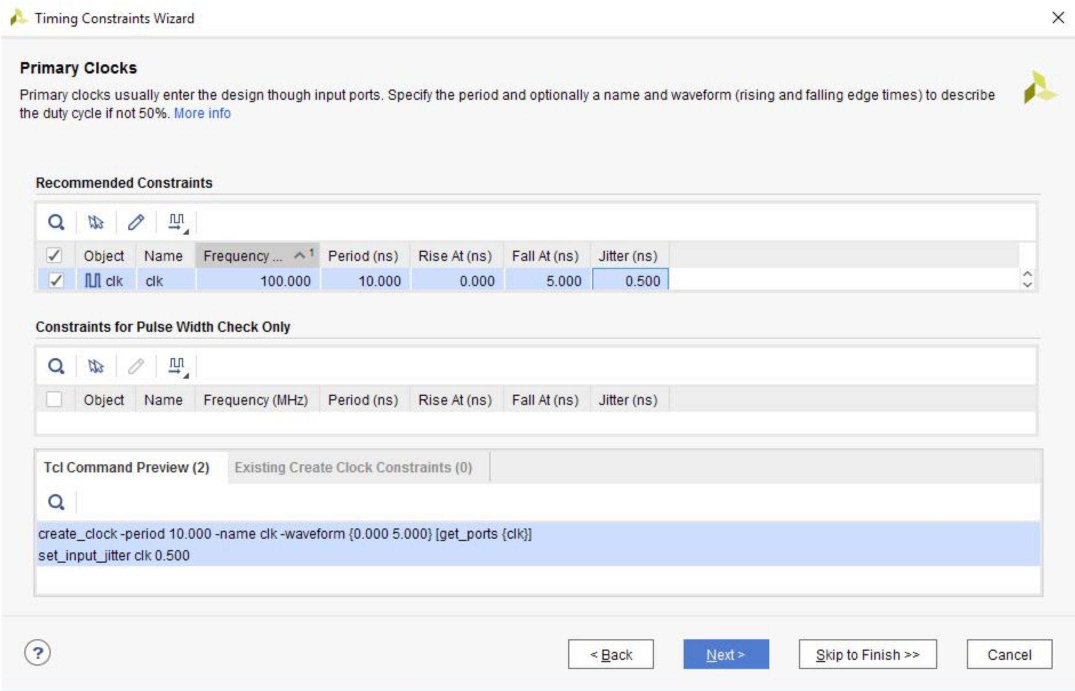

Figure 30

7. The second window deals with **Generated Clocks**, clocks that are derived from the master clock. Since we do not have any in our design, just click **Next**

8. The third window deals with **Forwarded Clocks**. While driving out the clock from an FPGA to off-chip devices like DDR memory, the clock forwarding technique enables to match clock and data path delays with both experiencing an equal amount of delay in the I/O. Since we do not have any in our design, just click **Next**

9. The forth window deals with **External Feedback Delays** needed in the case of designs using mixed-mode clock manager (MMCM) modules and/or phase-locked loop (PLL) modules, which serve as frequency synthesizers – frequency division, phase shifting, clock inversion. Since we are not using MMCM or PLL modules in our design, just click **Next**
10. The fifth window deals with **Input Delays**. It describes the delay (or relative phase) between the rising edge of our clock – **clk** – and the instant input signals change at the input of the device. Read the brief explanation provided and take a look at the **More info** link to better understand what it is and the importance of setting this timing constraint
11. Vivado automatically recommends to constrain the input signals **direction**, **en** and **reset**. To select all signals, right click to open a drop-down menu and select **Select All**. Fill in the right table with the values shown in Figure 31. Check if everything is correct and click **Next**
12. The sixth window deals with **Output Delays**. It describes the delay (or relative phase) between the rising edge of the board clock or of a forwarded clock – in our case the board clock is also our **clk** FPGA clock, but it may not be the case in all designs – and the instant output signals change at the output of the device. Read the brief explanation provided and take a look at the **More info** link to better understand what it is and the importance of setting this timing constraint
13. Vivado automatically recommends constraining all the output signals that are going to drive the LEDs. Select this bus signal and fill in the right table with the values shown in Figure 32. Check if everything is correct and click **Next**

    Refer to chapter 3 of the "UltraFast Design Methodology Guide, UG949" for further information about constraining input and output ports.
14. The seventh window – **Combinational Delays** – is about the constraining of combinational paths that cross the FPGA without being captured by any sequential element. Since we do not have any purely combinational path in our design, just click **Next**

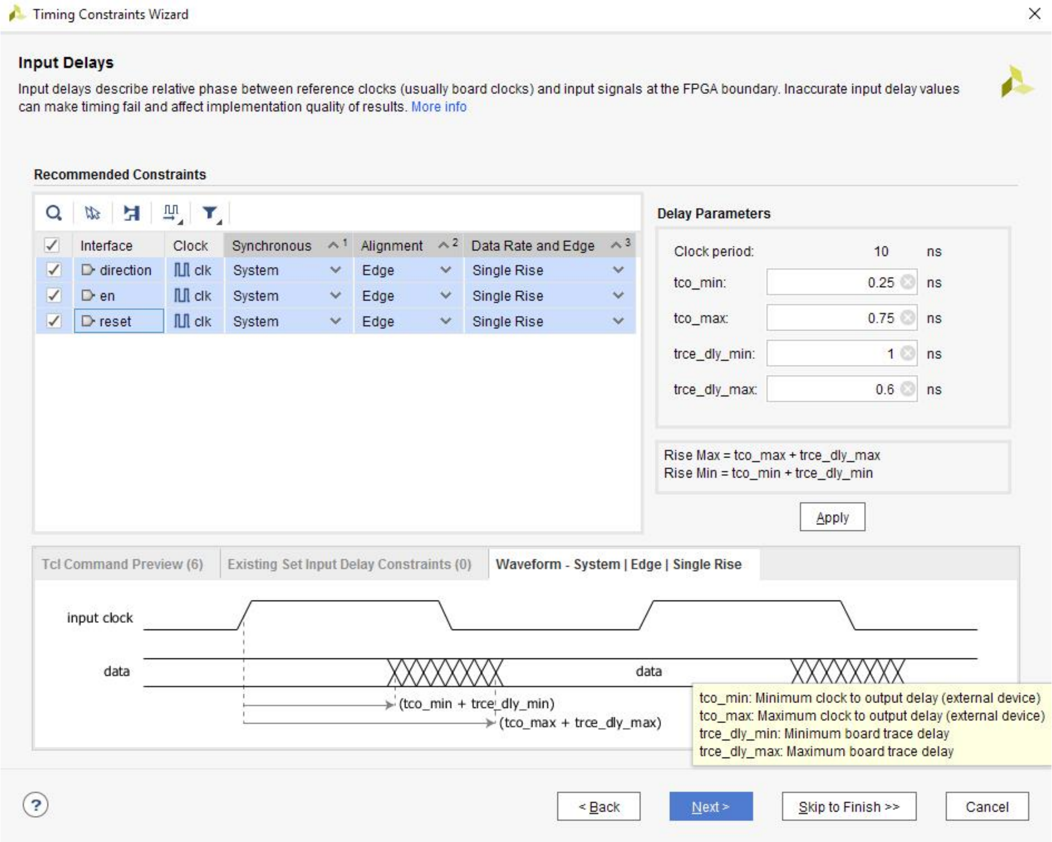

Figure 31

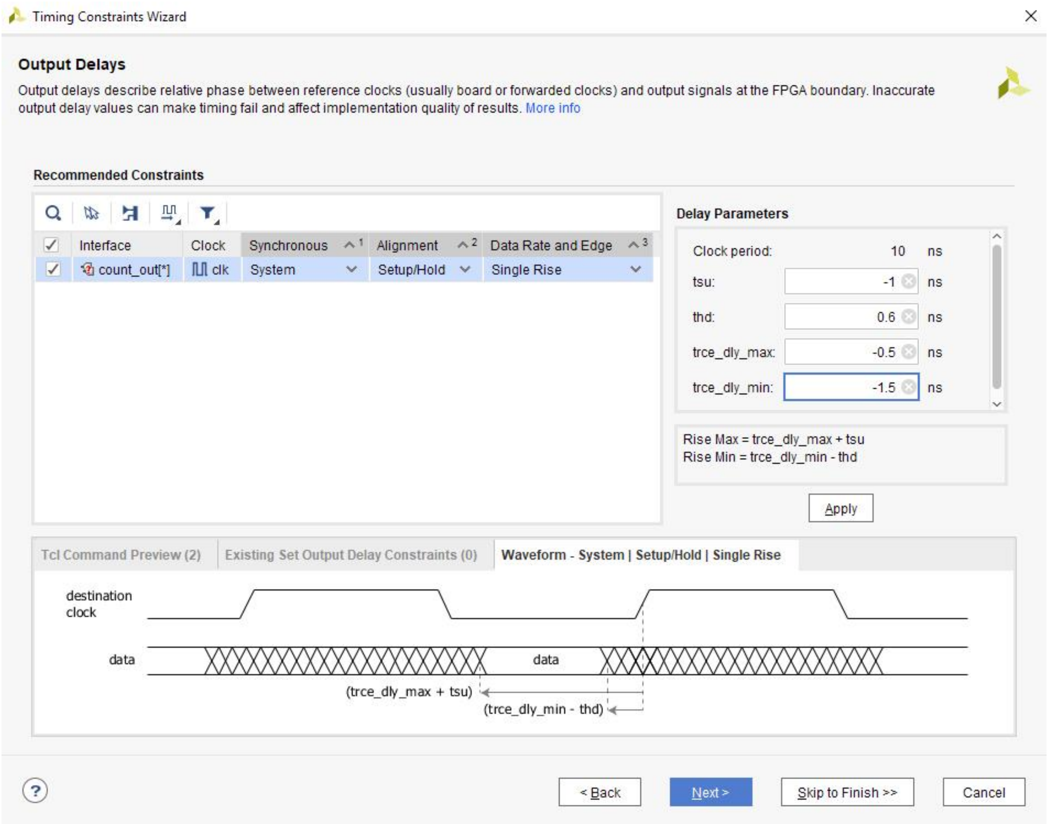

Figure 32

15. The eighth window – **Physically Exclusive Clock Groups** – deals with situations where the design has two primary clocks assigned to the same physical pin that, logically, cannot be used simultaneously. Since we only have one clock in our design, just click **Next**
16. The ninth window – **Logically Exclusive Clock Groups with No Interaction** – deals with situations where the design has more than one clock active at the same time that is defined on different source points but shares part of their clock tree, due to a multiplexer for example. Since we only have one clock in our design, just click **Next**
17. The tenth window – **Logically Exclusive Clock Groups with Interaction** – deals with situations where the design has more than one clock active at the same time except on shared clock tree sections. Logically exclusive clocks share timing paths in addition to their shared clock tree. Since we only have one clock in our design, just click **Next**
18. The eleventh window – **Asynchronous Clock Domain Crossings** – deals with situations where data is transferred between two different clock regions and whose clocks have no known phase relationship: they are asynchronous. Since we only have one clock in our design, just click **Next**
19. The last window shows a summary of the newly created ten timing constraints. To review the timing constraints, just clicked on each one of the hyperlinked lines
20. To keep these new timing constraints, click **Finish** to save them in the constraints file – **counter.xdc**

After creating the timing constraints, we must constrain our pin locations, to indicate to the design which pins each one of our **counter** signals should be connected.

25. Assigning I/O Pin Locations

We must now create pin assignments for all the input and output signals of our **counter** design.

1. Open the I/O Planning interface by selecting in the menu bar the **Layout** menu and choosing **I/O Planning** (Figure 33)

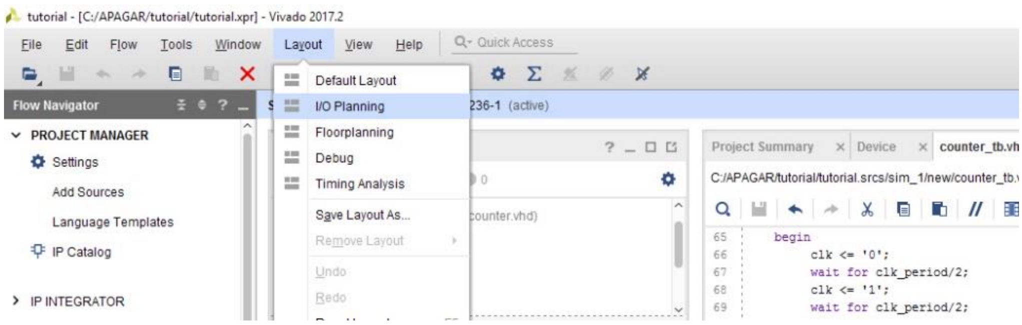

Figure 33

2. In the **I/O Ports** window, located in the bottom of the main window, we have a list of all I/O signals that are part of our **counter** design
3. Expand the **count_out(16)** output bus, to have access to all the individual bus signals, and the **Scalar Ports** trees under **All ports**
4. Locate and select the **clk** input signal under **Scalar Ports**, then attribute to this signal the pin location **W5** (GCLK signal on the Basys-3 board - see the Basys-3 FPGA Board Reference Manual to know the pin assignment in the board) by writing **W5** (or selecting it from the list) in the **Package Pin** column (Figure 34), as **I/O Std → LVCMOS33** and as **Vcco → 3.300** (this last value should change automatically after changing the **I/O Std** value)

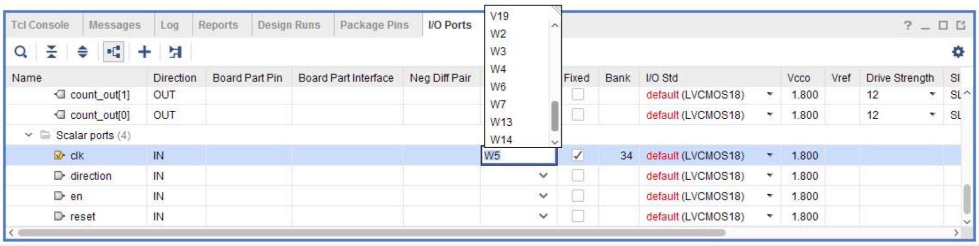

Figure 34

5. Locate the **reset** input signal under **Scalar Ports**, then attribute to this signal the pin location **U18**, connected to the BTNC push-button on the Basys-3 board, as **I/O Std → LVCMOS33** and as **Vcco → 3.300**
6. Locate the **en** input signal under **Scalar Ports**, then attribute to this signal the pin location **V16**, connected to the SW1 switch on the Basys-3 board, as **I/O Std → LVCMOS33** and as **Vcco → 3.300**
7. Locate the **direction** input signal under **Scalar Ports**, then attribute to this signal the pin location **V17**, connected to the SW0 switch on the Basys-3 board, as **I/O Std → LVCMOS33** and as **Vcco → 3.300**
8. Concerning the **count_out(16)** output bus signals, the aim is to connect the outputs of our counter to the 16 LEDs of our Basys-3 board. To do it, repeat the previous pin assignment step to place the following additional input and output pins using the following information from Board Reference Manual:
   - **count_out[15]** (LD15 LED on the board) **Package Pin → L1**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
   - **count_out[14]** (LD14 LED on the board) **Package Pin → P1**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
   - **count_out[13]** (LD13 LED on the board) **Package Pin → N3**, **I/O Std → LVCMOS33**, **Vcco → 3.300**

- **count_out[12]** (LD12 LED on the board) **Package Pin → P3**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[11]** (LD11 LED on the board) **Package Pin → U3**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[10]** (LD10 LED on the board) **Package Pin → W3**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[9]** (LD9 LED on the board) **Package Pin → V3**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[8]** (LD8 LED on the board) **Package Pin → V13**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[7]** (LD7 LED on the board) **Package Pin → V14**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[6]** (LD6 LED on the board) **Package Pin → U14**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[5]** (LD5 LED on the board) **Package Pin → U15**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[4]** (LD4 LED on the board) **Package Pin → W18**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[3]** (LD3 LED on the board) **Package Pin → V19**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[2]** (LD2 LED on the board) **Package Pin → U19**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[1]** (LD1 LED on the board) **Package Pin → E19**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[0]** (LD0 LED on the board) **Package Pin → U16**, **I/O Std → LVCMOS33**, **Vcco → 3.300**

9. Before proceeding, verify that the **Fixed** box is checked for all pins
10. Save the newly created **I/O Planning** constraints into the constraints file by selecting **File → Save Constraints**

26. Setting the Configuration Bank Voltage Select

Incorrect voltage supply for the configuration interfaces on board can result in configuration failure or device damage. The Vivado Design Rule Check (DRC) tool can check if the configuration interfaces of the device have correct voltage support based on the Configuration Bank Voltage Select (CFGBVS), CONFIG_VOLTAGE, and the CONFIG_MODE properties settings. Those properties tell Vivado how the device configuration interfaces are used and connected on board. If we do not set those properties during the design, Vivado will not perform the related DRC checks. Then the designer has to make sure that the device has the correct voltage support for configuration interfaces on board.

To configure the configuration interface:

1. If not open, **Open Synthesized Design** in the **Flow Navigator** window
2. In the menu bar select the **Tools** menu and choose **Edit Device Properties…**
3. In the vertical toolbar of the new **Edit Device Properties** window select **Configuration**
4. In the **Configuration Setup** section, go to **Configuration Voltage** and in the drop-down menu choose **3.3**
5. Then go to the **Configuration Bank Voltage Selection** and in the drop-down menu choose **VCCO** (see Figure 35)

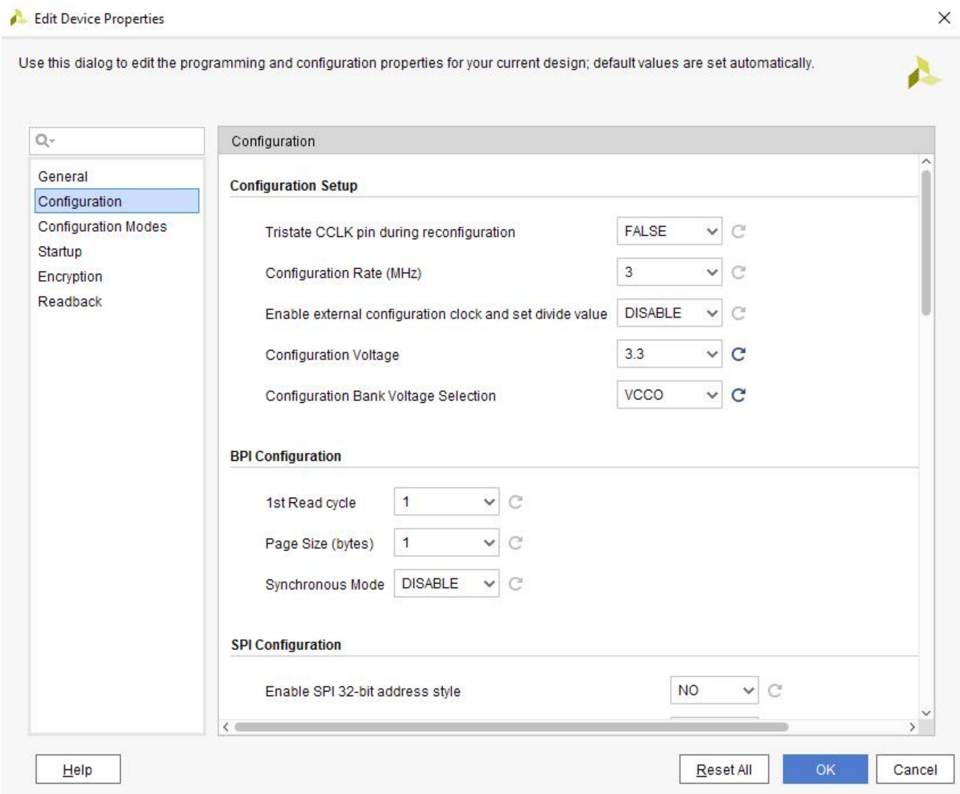

Figure 35

There is no need to configure the CONFIG_MODE property as JTAG/Boundary Scan (the configuration interface we are going to use with our board), because it is always selected by default. Otherwise, we must choose **Configuration Modes** in the vertical toolbar and select the desired configuration mode.

6. Click **OK**
7. Save these new constraints into the constraints file by selecting **File → Save Constraints**
8. Select the **Sources** tab and under the **Constraints** tree click on **counter.xdc (target)**
9. The **counter.xdc** file opens in the main window. Check that all the timing, I/O, and configuration voltage constraints appear in the file

Notice: Because XDC constraints are applied sequentially and are prioritized based on clear precedence rules, you must review the order of your constraints carefully. In a project flow without any IP, all the constraints are located in a constraints set. By default, the order of the XDC files displayed in Vivado defines the read sequence used by the tool when loading an elaborated or synthesized design into memory. The file at the top of the list is read in first, and the bottom one is read in last. We can change the order by simply selecting the file in the IDE, and moving it to the desired place in the list. Many IP cores are delivered with one or more XDC files. When such IP cores are generated within your RTL project, their XDC files are also used during the various design compilation steps. Refer to “Using Constraints, Vivado Design Suite User Guide, UG903” for further information. Whether you use one or several XDC files for your design, organize your constraints in the following sequence

```
## Timing Assertions Section
# Primary clocks
# Virtual clocks
# Generated clocks
# Clock Groups
# Bus Skew constraints
# Input and output delay constraints

## Timing Exceptions Section
# False Paths
# Max Delay / Min Delay
# Multicycle Paths
# Case Analysis
# Disable Timing

## Physical Constraints Section
# located anywhere in the file, preferably before or after the
timing constraints
# or stored in a separate constraint file
```

27. Implementation

Now that all the necessary constraints have been defined, let’s continue with the implementation of the design.

1. In the **Flow Navigator** window, in the left side of the **Project Manager** main window, in the **Project Manager** section, click **Settings**

2. In the **Project Settings** menu of the new **Settings** window, click **Implementation**
3. Take a look at the **Project Settings → Implementation** window, in particular to the **Strategy:** list. Open the drop-down menu to have an idea of the different implementation strategies we may choose when implementing our designs. Strategies are defined in pre-configured sets of options for the Vivado implementation features. Strategies are tool and version specific, so each major release of Vivado includes version-specific strategies that may be different from its predecessors. Refer to Appendix C of the “Implementation, Vivado Design Suite User Guide, UG904” for a complete description of the implementation strategies available in the current release
4. In our case, we use the **Vivado Implementation Defaults** as implementation strategy, so just take a look and accept the default settings by clicking **OK**
5. To implement the design, in the **Flow Navigator** window, in the **Implementation** section, click **Run Implementation**
6. At the bottom of the main window select **Log → Implementation** to follow implementation evolution
7. In the end, a new window opens stating that the implementation was successfully completed and showing the next steps. Click **OK** to open the **Implemented Design**
8. A window with a view of how our logic was placed and routed inside the **Device** appears in the main window. Use the different tools available in the horizontal toolbar to zoom in and out and to view the cell connections
9. In the **Flow Navigator** window, in the **Implementation** section, click **Open Implemented Design** to expand the process hierarchy
10. A number of different reports may be generated now that the implementation of the design in the FPGA is done. Notice that the same list also appeared after synthetizing the design, under the **Open Synthesized Design** entry in the **Flow Navigator**. These reports now contain the most accurate information about the real behavior of our design. Indeed, implementation is the closest emulation to actually downloading a design to a device

28. Post-Implementation Design Simulation

We may repeat the simulation of our design as it will now, after place and route, take into account the exact influence of the propagation delays inside our FPGA. Go to section 18 – Design Simulation – and repeat the indicated steps to simulate the design, but instead of choosing, in the third step, **Run Post-Synthesis Functional Simulation**, choose **Run Post-Implementation Timing Simulation**. Notice the waveform fluctuation of the values in the

count_out[15:0] output before stabilization after almost each increment of the counter (Figure 36). This behavior is caused by the different propagation delays each of the bus signals suffers between the output of the counter register and the output pin, something not taken into account when performing simulation after synthesis, because the signals' routing was then unknown.

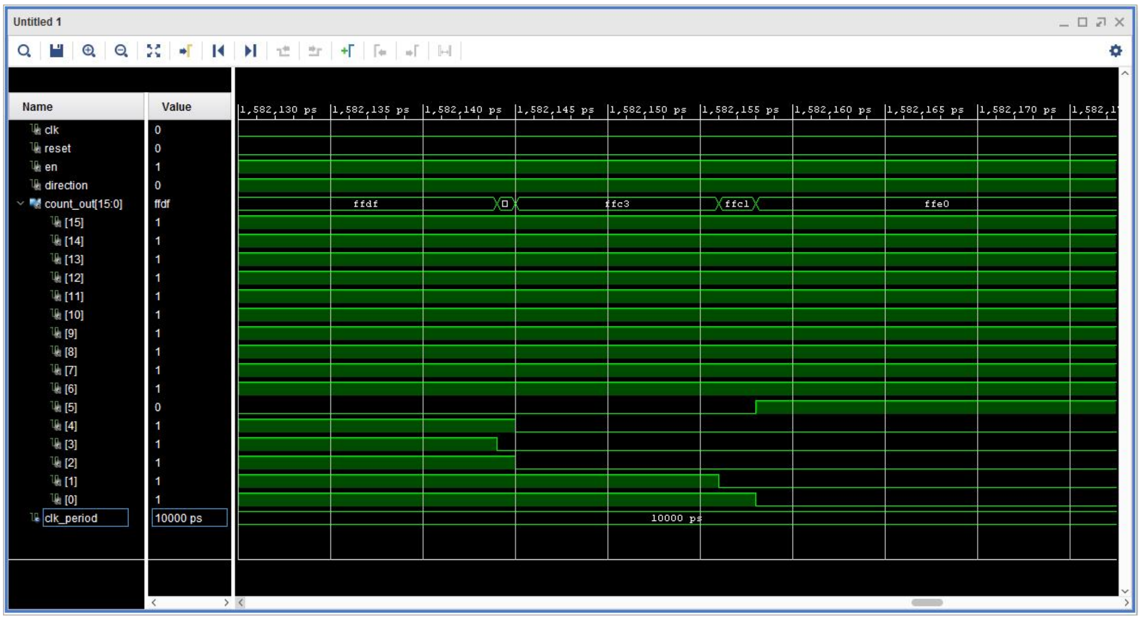

Figure 36

29. Creating Configuration Data and Device Configuration

After implementation we need to create the configuration data for our device. During the **Program and Debug** step a configuration bitstream is created for downloading to the target device.

To create a bitstream for the target device, set the properties and run configuration as follows:

1. In the **Flow Navigator** window, in the **Program and Debug** section, click **Generate Bitstream**
2. At the bottom of the main window select **Log → Implementation** to follow bitstream generation evolution
3. In the end, a new window opens stating that the bitstream generation was successfully completed, which means that the bitstream file (in this tutorial, the counter.bit file) that contains the actual configuration data was created
4. Before proceeding, connect the Xilinx USB cable between a USB port of your computer and the **PROG** USB port of your board and switch **ON POWER**
5. Then, choose **Open Hardware Manager** and click **OK**

6. In the top left of the new **Hardware Manager** window, select **Open target → Auto Connect** (Figure 37)

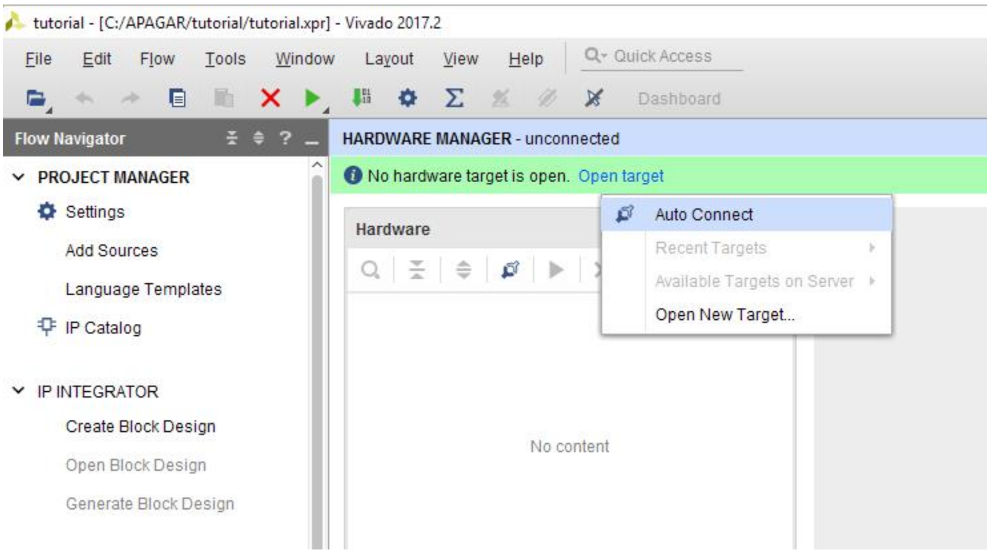

Figure 37

7. The target device is automatically connected and it is ready to be configured
8. In the top left of the **Hardware Manager** window, select **Program device**
9. In the new **Program Device** window, check that the configuration file is the **counter.bit** file and click **Program** (Figure 38)

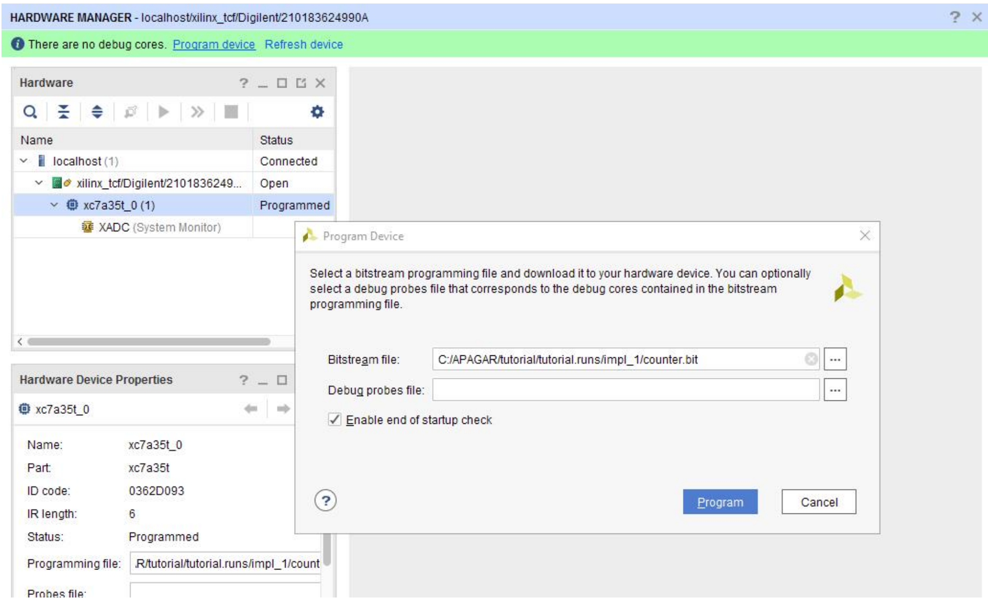

Figure 38

Check in the **Tcl Console** at the bottom of the main window the evolution of the device configuration. In the end, LED LD(15..0) on the board lit up, if the **en** signal is active (SW1), indicating that the counter is running.

30. Reducing the Counter Speed

Despite having rigorously followed all previous steps, the up/down counter seems not to work properly! When the enable signal **en** is active (SW1) all LEDs are lit simultaneously, despite their different brightness, and switching SW0 has no apparent effect on it. However, we are able to stop the counter changing the state of the switch SW1 that controls the enable input of the counter register, or by pressing the BTNC push-button which resets the counter. This apparent weird behavior is caused by the 100 MHz frequency of the clock signal available on board and applied to the counter, which is too high to let the sequence of LED lighting be perceptible by the human eye.

Therefore, the counter speed must be reduced in order to obtain a much lower counting frequency. To do this we need to add to the design a new module that generates an enable pulse with a much lower frequency than the frequency of the board clock – only 10 Hz. This enable pulse is applied to the counter's register clock enable inputs. Therefore, the counter's value is increased or decreased, depending on the **direction** signal, only once each 10,000,000 clock cycles.

31. Create a new VHDL Source

Create a new VHDL source file for the project as follows:

1. Start by closing the **Hardware Manager** window in the main design window
2. In the **Flow Navigator** window, in the left side of the **Project Manager** window, click **Add Sources**
3. Select **Add or create design sources**
4. Click **Next**
5. In the new window click **Create File**
6. In the **Create Source File** window:
   - File type: **VHDL**
   - File name: **clk10Hz**
   - File location: **<Local to Project>**
7. Click **OK**

   A new design source file **clk10Hz.vhd** is added to the project
8. Click **Finish**
9. A new window opens, where we may define our module I/O ports

   Our 10Hz enable pulse generator needs as module inputs a clock to run, a reset to initialize its register, an enable signal to enable the generation of the 10Hz signal and as sole output the 10Hz enable pulse that is applied to the counter. In this way, we obtain a perceptible visual output of the counter behavior. These input and output ports may be declared directly into the VHDL file or may be declared using the **Define Module** wizard.

10. Declare the **clk10Hz** design ports by filling in the port information as shown in Figure 39.

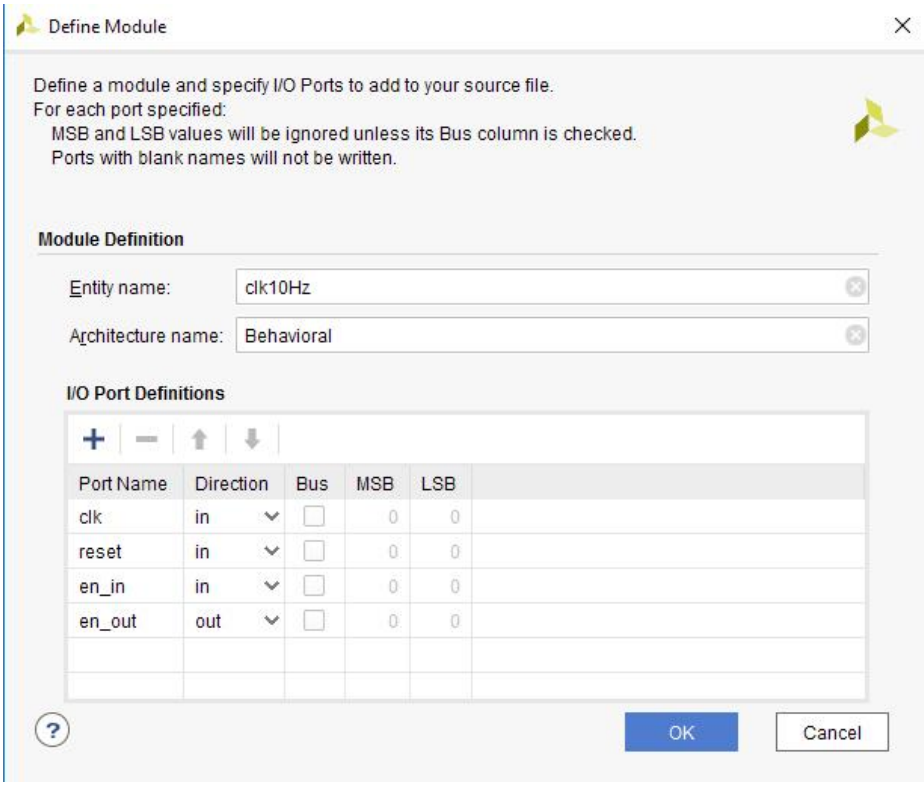

Figure 39

11. Click **OK**
12. Our new VHDL design source file, **clk10Hz.vhd**, which contains our VHDL entity/architecture pair, is now added to the **Sources** window in our **Project Manager**.

    The next step in creating the new source is to add the behavioral description for the 10Hz enable pulse generator. To generate the pulse, we just need to count 10,000,000 clock pulses and then generate the enable pulse.
13. Double click **clk10Hz – Behavioral (clk10Hz.vhd)** in the **Sources** window
14. The **clk10Hz.vhd** VHDL design source opens in the main window
15. Uncomment the declaration `use.IEEE.NUMERIC_STD.ALL`
16. After the architecture declaration and before the `begin` statement add the following VHDL line, which defines the clock pulse counter register and initializes it to zero

    `signal counter: INTEGER range 0 to 10000000 := 0;`
17. Between the `begin` statement and the `end Behavioral` statement, add the following VHDL lines

```
process (clk)
   begin

   if clk'event and clk = '1' then
         if reset = '1' then
               counter <= 0;
               en_out <= '0';

         elsif en_in = '1' then
               counter <= counter+1;

               if (counter = 10000000) then
                     en_out <= '1';
                     counter <= 0;
               else
                     en_out <= '0';
               end if;
         end if;
   end if;

end process;
```

18. After finishing editing the source file, save it by selecting **File → Save File** or click in the save file icon in the corner of the **clk10Hz** design source file window
19. Check for and correct any errors (see section 11 if needed)

We have now created the VHDL source for the 10Hz pulse enable generator. Add the comments you wish to explain the code behavior.

32. Creating a Test Bench File for the new clk10Hz Module

After creating the module, we need to simulate it, to verify if its functionality is according to the specification: a module that generates an enable pulse each 10,000,000 clock cycles.

To simulate the new module, we need to create a test bench file and to add it to our project:

1. In the **Flow Navigator** window, in the left side of the **Project Manager** window, click **Add Sources** or select **File → Add Sources…** in the Vivado menu bar
2. Select **Add or create simulation sources**
3. Click **Next**

4. In the new window click **Create File**
5. In the **Create Source File** window:
   - File type: **VHDL**
   - File name: **clk10Hz_tb**
   - File location: **<Local to Project>**
6. Click **OK**

   A new design source file **clk10Hz_tb.vhd** is added to the project
7. Click **Finish**
8. A new window opens, where we may define our module I/O ports. Since test benches have no entity ports, click **OK → Yes**
9. The new **clk10Hz_tb** simulation file is added to the **Simulation Sources** tree in the **Sources** window
10. The next step is to customize the test bench file in the text editor
11. Double click **clk10Hz_tb – Behavioral (clk10Hz_tb.vhd)** in the **Sources** window
12. The **clk10Hz_tb** VHDL design source opens in the main window
13. Select all the code lines in the design source and delete them
14. Then add the following code lines

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY clk10Hz_tb IS
END clk10Hz_tb;

ARCHITECTURE behavior OF clk10Hz_tb IS

     -- Component Declaration for the Unit Under Test (UUT)

     COMPONENT clk10Hz
     PORT(
          clk : in  STD_LOGIC;
          reset : in STD_LOGIC;
          en_in : in STD_LOGIC;
          en_out : out STD_LOGIC
          );
     END COMPONENT;
```

```
	--Inputs
	signal clk : STD_LOGIC := '0';
	signal reset : STD_LOGIC := '1';
	signal en_in : STD_LOGIC := '0';

	--Outputs
	signal en_out : STD_LOGIC;

	-- Clock period definitions
	constant clk_period : time := 10 ns;

begin

	-- Instantiate the Unit Under Test (UUT)
uut: clk10Hz PORT MAP (
			clk => clk,
			reset => reset,
			en_in => en_in,
			en_out => en_out
			);

	-- Clock process definitions
clk_process :process
begin
	clk <= '0';
	wait for clk_period/2;
	clk <= '1';
	wait for clk_period/2;
end process;

	-- Stimulus process
stim_proc: process
begin
	reset <= '1';
	en_in <= '0';

	-- hold reset state for 100 ns.
```

```
      wait for 100 ns;
      reset <= '0';

      wait for clk_period*10;

      -- insert stimulus here

      wait for 1 us;
      en_in <= '1';

      wait;
end process;

END;
```

15. Save the file by selecting **File → Save File** or click in the save file icon in the corner of the **clk10Hz_tb** design source file window
16. While saving the file, Vivado automatically checks the design for syntax errors and typos, in the same way as described in section 11, and automatically associates it to the **uut** (unit under test) **clk10Hz**

33. Simulating the new clk10Hz Module

Now that we have a test bench for the new **clk10Hz** module, we can perform behavioral simulation on the design using the XSIM simulator.

1. In the **Flow Navigator** window, in the left side of the **Project Manager** main window, in the **Project Manager** section, click **Settings**
2. In the **Project Settings** menu of the new **Settings** window, click **Simulation**
3. Check if the **Vivado Simulator** is selected in the **Target simulator** field and change the **Simulation top module name:** to **clk10Hz_tb** by clicking on the **…** icon at the right end of this parameter line (Figure 40)
4. A new **Select Top Module** window opens. Select **clk10Hz_tb** and click **OK**
5. Select the **Simulation** tab
6. Change the **xsim.simulate.runtime** parameter to **850ms**
7. Select the **Elaboration** tab

8. Add to the **xsim.elaborate.xelab.more_options** line the command **-timeprecision_vhdl 1ns**

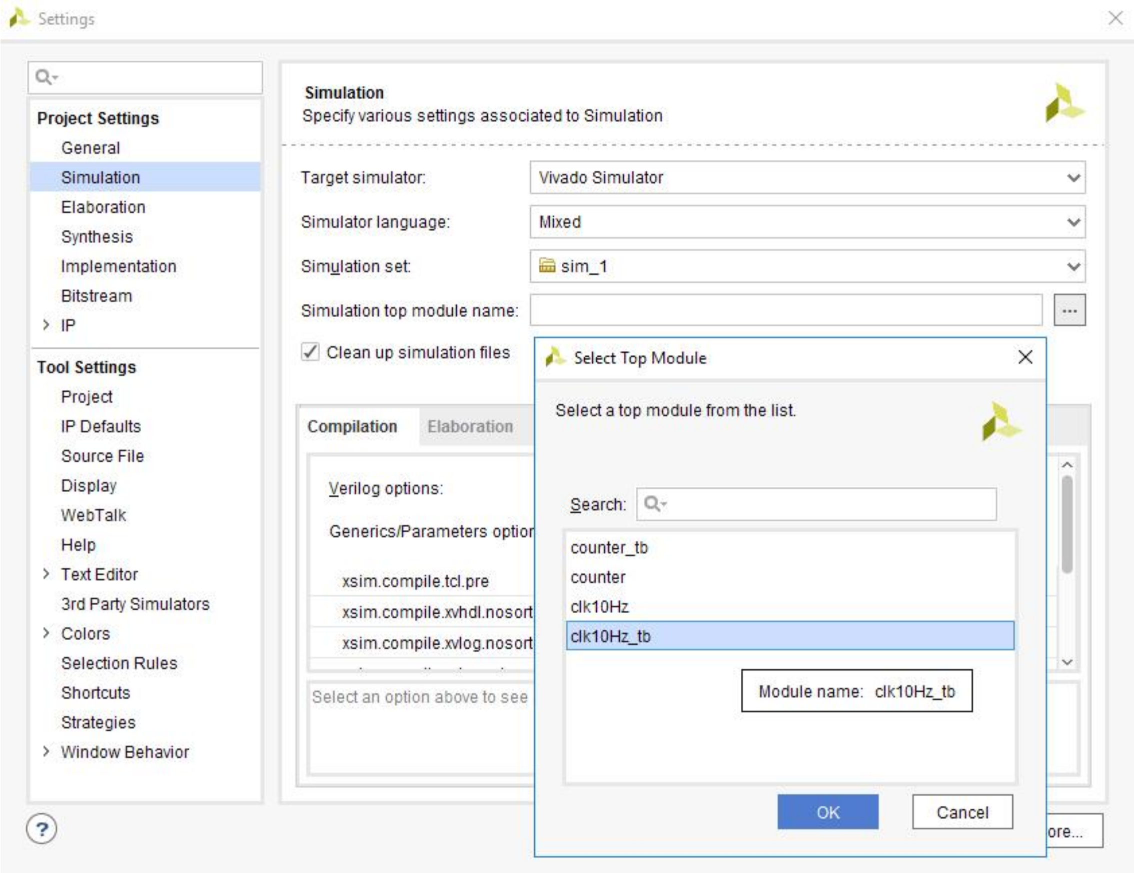

Figure 40

The simulation's default time resolution is 1ps. When the simulation runtime required to simulate our modules is long, we can (in certain circumstances) sacrifice precision to get faster simulation results. With this command we override the default time resolution, imposing a time resolution of 1ns. This is not always possible because some Xilinx primitive components, such as clock primitives, require a 1 ps resolution in order to work properly in either functional or timing simulation. In these cases, we get a simulation error. Anyway, it takes some time to simulate the **clk10Hz** module.

9. Click **OK**
10. In the **Flow Navigator** window, in the **Simulation** section, click **Run Simulation**
11. In the drop-down menu choose **Run Behavioral Simulation**
12. Wait for the simulation to finish and analyze the result to verify if the **clk10Hz** module works as expected. If not, correct the VHDL description accordingly and rerun simulation
13. Since the **en_out** pulse is produced only each 100ms, it is difficult to find it when looking at the simulation waveform. To find it easily, just select the signal's name **en_out** in the

waveform names list and click on the **Next Transition** icon in the waveform window's horizontal toolbar (Figure 41), and the yellow marker jumps to the next **en_out** signal transition

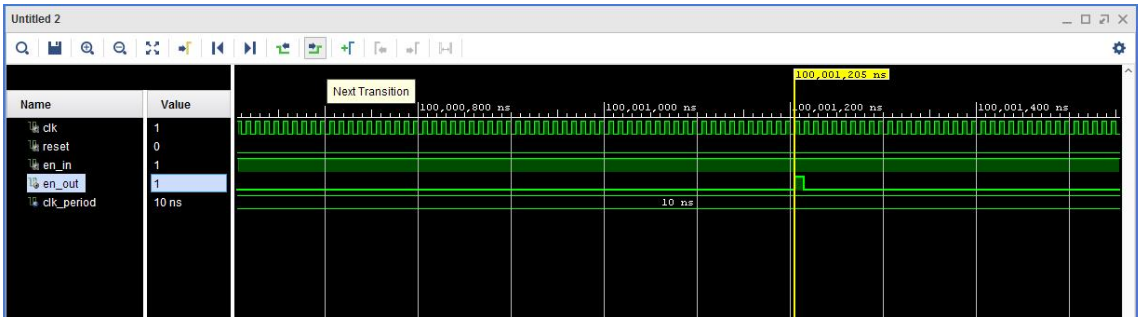

Figure 41

14. To return to a previous transition, click on the **Previous Transition** icon placed immediately to the left of the **Next Transition** icon
15. For other simulation functionalities, check section 19

Now that the 10Hz enable pulse generator is up and running, it is necessary to connect it to our **counter** module. To do that, we need to create a top-level module, a new module that contains these two together with a description of the interconnections between them and with the input and output signals.

34. Create a VHDL Top-level Module

To help creating the top-level module it is always useful to create a structural description of it based on component instantiation. A schematic representation of the final top-level module is shown in Figure 42

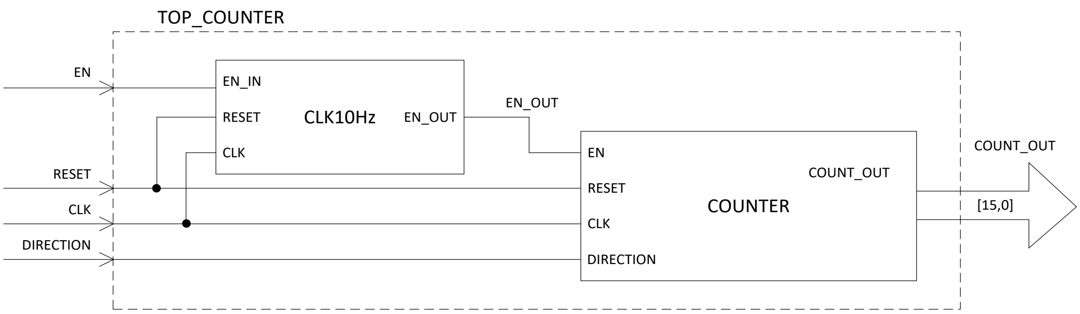

Figure 42

Create a new VHDL source file for the top-level module as follows:

1. Start by closing the **Behavioral Simulation** window
2. In the **Flow Navigator** window, in the left side of the **Project Manager** window, click **Add Sources**

3. Select **Add or create design sources**
4. Click **Next**
5. In the new window click **Create File**
6. In the **Create Source File** window:
    - File type: **VHDL**
    - File name: **top_counter**
    - File location: **<Local to Project>**
7. Click **OK**

    A new design source file **top_counter.vhd** is added to the project
8. Click **Finish**
9. A new window opens, where we may define our module I/O ports

    Our top-level module needs as module inputs a clock to run, a reset to initialize both **counter** and **clk10Hz** registers, an enable signal to enable the counting and as sole output the 16-bit output of the counter that connects to the 16 LEDs available on our board. These input and output ports may be declared directly into the VHDL file or may be declared using the **Define Module** wizard.
10. Declare the **top_counter** design ports by filling in the port information as shown in Figure 43.

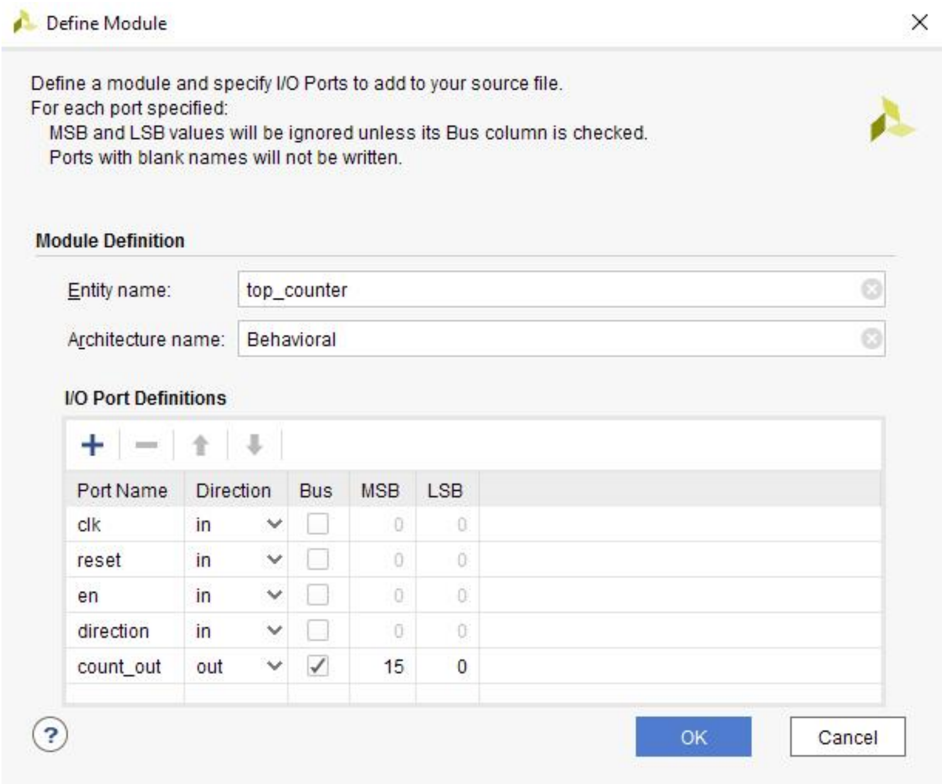

Figure 43

11. Click **OK**
12. Our new VHDL design source file, **top_counter.vhd**, which contains our VHDL entity/architecture pair, is now added to the **Sources** window in our **Project Manager**

    The next step in creating the new source is to add the structural description for the top-level module, according to the block diagram.
13. Double click **top_counter – Behavioral (top_counter.vhd)** in the **Sources** window
14. The **top_counter.vhd** VHDL design source opens in the main window
15. After the architecture declaration and before the `begin` statement we need to add the two component declarations and the enable interconnect wire declaration – **en_out**

```
COMPONENT clk10Hz
      PORT(
            clk : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            en_in : in  STD_LOGIC;
            en_out : out  STD_LOGIC
            );
END COMPONENT;

COMPONENT counter
      PORT(
            clk : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            en : in  STD_LOGIC;
            direction : in  STD_LOGIC;
            count_out : out STD_LOGIC_VECTOR (15 downto 0)
            );
END COMPONENT;

signal en_out : STD_LOGIC := '0';
```

16. Between the `begin` statement and the `end Behavioral` statement, we need to add the structural description of the top-level module

```
Inst_clk10Hz: clk10Hz PORT MAP(
                          clk => clk,
                          reset => reset,
                          en_in => en,
                          en_out => en_out
                          );

      Inst_counter: counter PORT MAP(
                          clk => clk,
                          reset => reset,
                          en => en_out,
                          direction => direction,
                          count_out => count_out
                          );
```

17. After finishing editing the source file, save it by selecting **File → Save File** or click in the save file icon in the corner of the **top_counter** design source file window
18. Check for and correct any errors (see section 11 if needed)
19. Look at the **Sources** window. Vivado assumes automatically the **top_counter** as the top-level module of our design in the **Design Sources** tree

We have now created the VHDL source for the top-level module. Add the comments you wish to explain the code behavior.

35. Synthesize the Full Design

Now that our design is complete, we may synthesize it.

1. In the **Sources** window, select **top_counter – Behavioral (top_counter.vhd**)
2. In the **Flow Navigator** window, in the **Synthesis** section, click **Run Synthesis**
3. Check the **Log** window at the bottom of the **Project Manager** window to follow synthesis evolution. Select the **Synthesis** tab if this is not the current selected tab
4. In the end, a new window opens stating that the synthesis was successfully completed and showing the next steps
5. Before proceeding to the Implementation step, we are going to simulate our design. Close the window by clicking **Cancel**

If you want, you may take a look at the schematic representation of our now complete design. Just follow the steps indicated in section 15.

36. Creating a Test Bench File for the new Top-level Module

After creating the top-level module, we need to simulate it, to verify the functionality of the whole design.

To simulate the new top-level module, we need to create a test bench file and to add it to our project:

1. In the **Flow Navigator** window, in the left side of the **Project Manager** window, click **Add Sources** or select **File → Add Sources…** in the Vivado menu bar
2. Select **Add or create simulation sources**
3. Click **Next**
4. In the new window click **Create File**
5. In the **Create Source File** window:
    - File type: **VHDL**
    - File name: **top_counter_tb**

- File location: **<Local to Project>**

6. Click **OK**

   A new design source file **top_counter_tb.vhd** is added to the project

7. Click **Finish**
8. A new window opens, where we may define our module I/O ports. Since test benches have no entity ports, click **OK → Yes**
9. The new **top_counter_tb** simulation file is added to the **Simulation Sources** tree in the **Sources** window
10. The next step is to customize the test bench file in the text editor
11. Double click **top_counter_tb – Behavioral (top_counter_tb.vhd)** in the **Sources** window
12. The **top_counter_tb** VHDL design source opens in the main window
13. Select all the code lines in the design source and delete them
14. Then add the following code lines

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY top_counter_tb IS
END top_counter_tb;

ARCHITECTURE behavior OF top_counter_tb IS

      -- Component Declaration for the Unit Under Test (UUT)

COMPONENT top_counter
      PORT(
            clk : in STD_LOGIC;
            reset : in STD_LOGIC;
            en : in STD_LOGIC;
            direction : in STD_LOGIC;
            count_out : out STD_LOGIC_VECTOR (15 downto 0)
            );
END COMPONENT;

      --Inputs

signal clk : STD_LOGIC := '0';
signal reset : STD_LOGIC := '1';
signal en : STD_LOGIC := '0';
signal direction : STD_LOGIC := '0';

      --Outputs

signal count_out : STD_LOGIC_VECTOR (15 downto 0);

      -- Clock period definitions

constant clk_period : time := 10 ns;
```

```
begin

      -- Instantiate the Unit Under Test (UUT)
uut: top_counter PORT MAP (
                           clk => clk,
                           reset => reset,
                           en => en,
                           direction => direction,
                           count_out => count_out
                           );

      -- Clock process definitions

clk_process :process
begin
      clk <= '0';
      wait for clk_period/2;
      clk <= '1';
      wait for clk_period/2;
end process;

      -- Stimulus process

stim_proc: process
begin

      reset <= '1';
      en <= '0';
      direction <= '0';

      -- hold reset state for 100 ns.

      wait for 100 ns;

      reset <= '0';

      wait for clk_period*10;

      -- insert stimulus here

      wait for 200 ms;

      en <= '1';

      wait for 400 ms;

      direction <= '1';

      wait;

end process;

END;
```

15. Save the file by selecting **File → Save File** or click in the save file icon in the corner of the **top_counter_tb** design source file window
16. While saving the file, Vivado automatically checks the design for syntax errors and typos, in the same way as described in section 11, and automatically associates it to the **uut** (unit under test) **top_counter**

37. Simulating the New Top-level Module

Now that we have a test bench for the new **top_counter** module and that we synthesized it, we can run a Post-Synthesis Functional Simulation on the design using the XSIM simulator.

1. In the **Flow Navigator** window, in the left side of the **Project Manager** main window, in the **Project Manager** section, click **Settings**
2. In the **Project Settings** menu of the new **Settings** window, click **Simulator**
3. Check if the **Vivado Simulator** is selected in the **Target simulator** field and change the **Simulation top module name:** to **top_counter_tb** by clicking on the **…** icon at the right end of this parameter line
4. A new **Select Top Module** window opens. Select **top_counter_tb** and click **OK**
5. There is no need to change the other simulation parameters, as the tool kept the last ones we changed, namely the 1ns time resolution
6. Click **OK**
7. In the **Flow Navigator** window, in the **Simulation** section, click **Run Simulation**
8. In the drop-down menu choose **Run Post-Synthesis Functional Simulation**
9. Wait for the simulation to finish (and it will take a while to finish, so you may want to go to have a coffee or other delicious beverage!) and analyze the result to verify if the **top_counter** module works as expected. If not, correct the VHDL description accordingly and rerun simulation
10. To remember simulation functionalities, check section 19

Now that the top-level module is up and running, the next step is to implement it in our FPGA.

38. Implementation

To implement our top-level design, we now need to add some physical constraints to it. Take a look at section 22 to remember why we need to add physical constraints to our design.

39. Creating a New Constraint Set for the Top-level Module

To create a new constraint set in our project:

1. Start by closing the **Post-Synthesis Simulation - Functional** window

2. In the **Flow Navigator** window, in the left side of the **Project Manager** window, click **Add Sources** or select **File → Add Sources…** in the Vivado menu bar
3. Select **Add or create constraints**
4. Click **Next**
5. In the new window, open the drop-down menu in front of the **Specify constraint set:** entry, and select **Create Constraint Set…** (Figure 44)

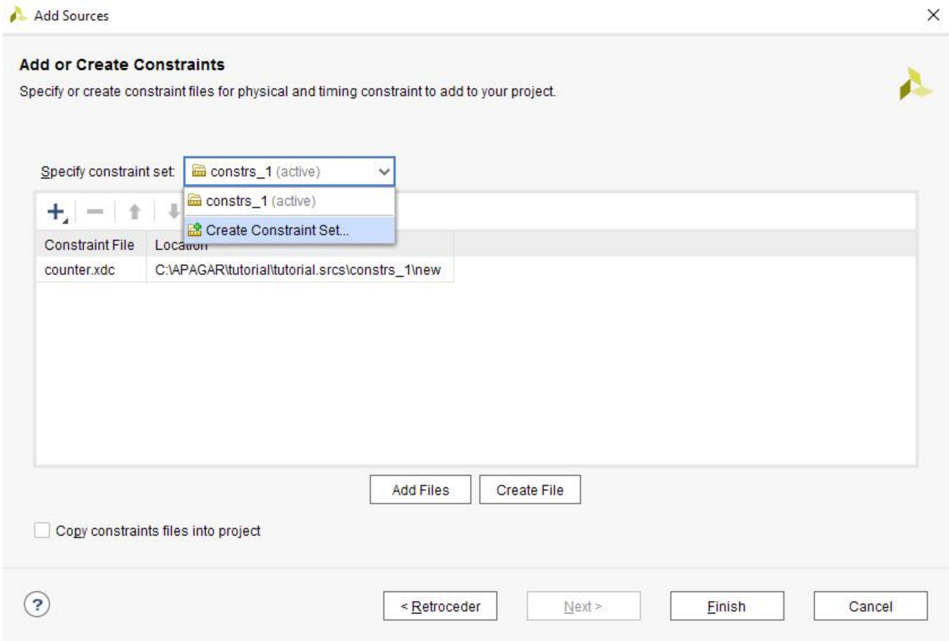

Figure 44

6. In the new **Create Constraint Set** window **Enter Constraint Set Name → top_counter** and click **OK** to return to the **Add Sources** window
7. In this window check **Make active** in front of the **top_counter** constraint set name (Figure 45)

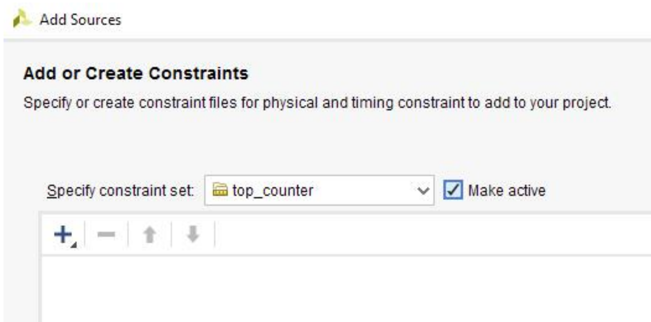

Figure 45

8. The constraint set **top_counter** is now our design's constraint set for the **top_counter** design implementation
9. Now we need to create the constraint file. Click **Create File**

10. In the **Create Source File** window:

    - File type: **XDC**
    - File name: **top_counter**
    - File location: **<Local to Project>**

11. Click **OK**
12. A new constraint file **top_counter.xdc** is added to the project
13. Click **Finish**
14. The new, and active, constraint set **top_counter** containing the new constraint file **top_counter.xdc** is added to the hierarchy of our design in the project **Sources** window (Figure 46)

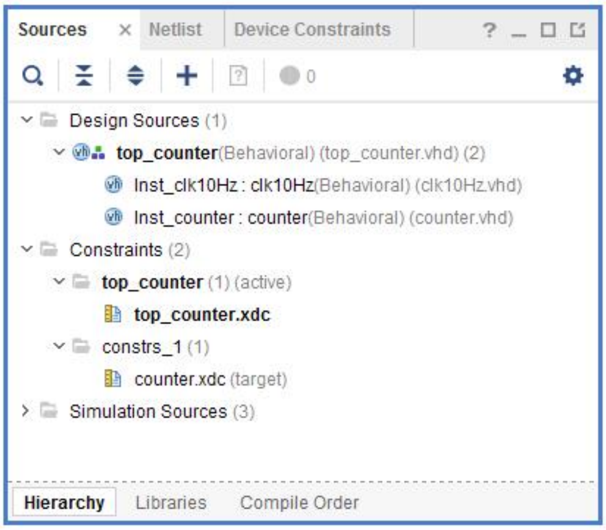

Figure 46

15. Since the synthesis and implementation settings changed with the addition of a new constraint file, Vivado asks to reload the design before proceeding. Click **OK** to reload and open the **Synthesized Design** window in the main window
16. Then, in the **Sources** window, select **top_counter.xdc** and make it the target constraint file by right-clicking and choosing **Set as Target Constraint File** (Figure 47)

The new **top_counter.xdc** file is now our target constraint file.

40. Adding Time Constraints to the New Constraint File

As we did in section 24, we start by creating a timing constraint to our clock.

1. In the menu bar select the **Tools** menu and choose **Timing → Constraints Wizard…**
2. A new window opens – the **Timing Constraints Wizard** window. Click **Next**
3. The first window of the wizard is the **Primary Clocks** window.

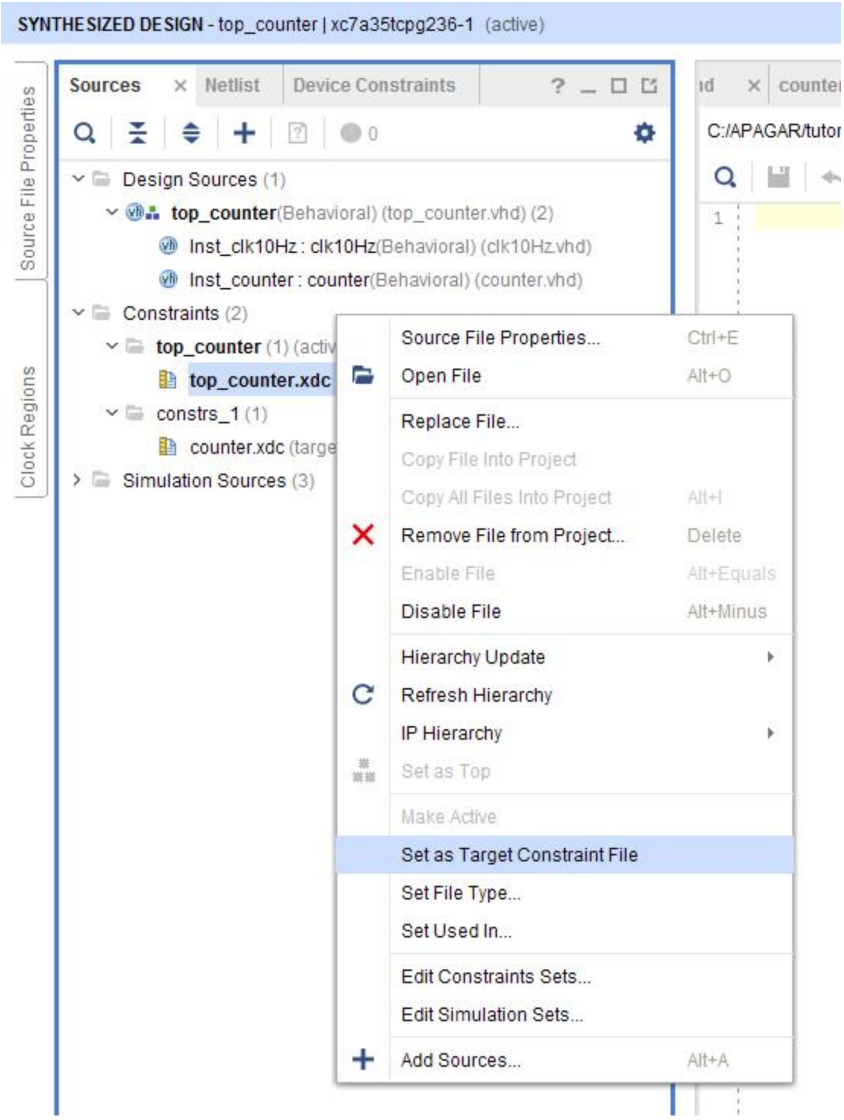

Figure 47

4. Vivado automatically recommends constraining the **clk** signal. Enter the missing frequency value by clicking on the cell that shows ‘undefined’, and type **100**. Automatically, the remaining cells – **Period**, **Rise At**, and **Fall At** – are filled in based on a 50% duty-cycle for the clock
5. You may also set **Jitter** to **0.5**
6. Check if everything is correct (Figure 48) and click **Next**
7. The second window deals with **Generated Clocks**, clocks that are derived from the master clock. Since we do not have any in our design, just click **Next**
8. The third window deals with **Forwarded Clocks**. Since we do not have any in our design, just click **Next**

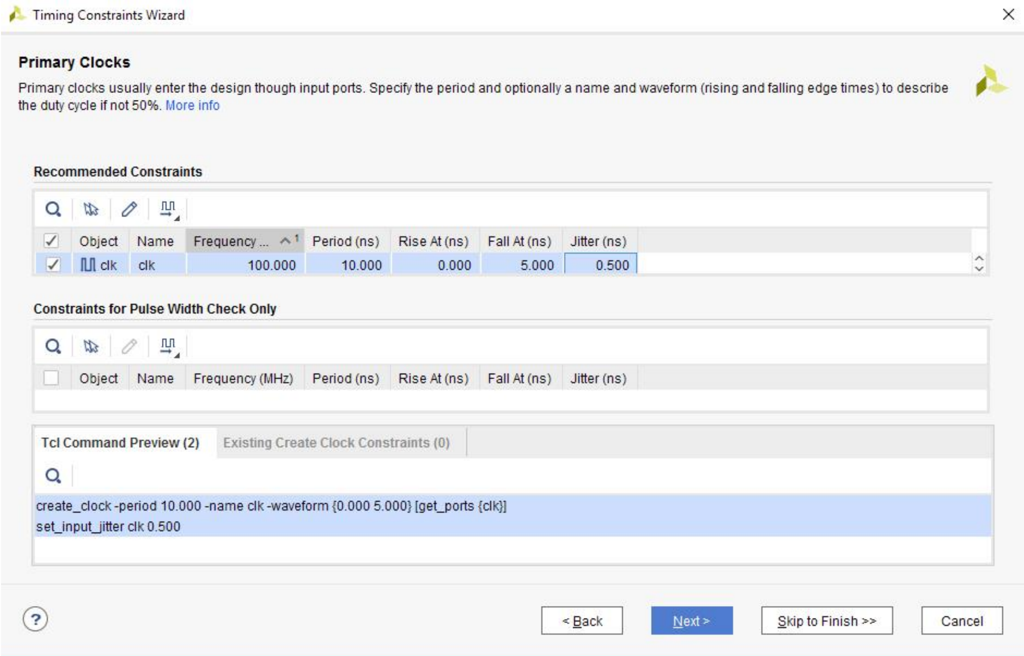

Figure 48

9. The forth window deals with **External Feedback Delays**. Since we are not using MMCM or PLL modules in our design, just click **Next**
10. The fifth window deals with **Input Delays**. Vivado automatically recommends to constrain the input signals **direction**, **en** and **reset**. To select all signals, right click to open a drop-down menu and select **Select All**. Fill in the right table with the values shown in Figure 49. Check if everything is correct and click **Next**

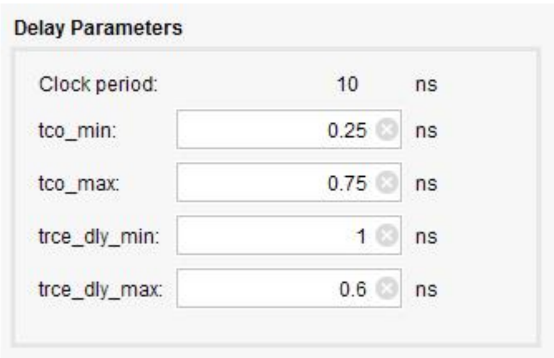

Figure 49

11. The sixth window deals with **Output Delays.** Vivado automatically recommends constraining all the output signals that are going to drive the LEDs. Select this bus signal and fill in the right table with the values shown in Figure 50. Check if everything is correct and click **Next**

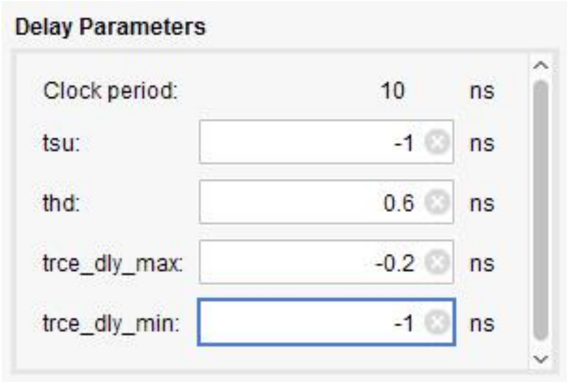

Figure 50

12. The seventh window – **Combinational Delays** – is about the constraining of combinational paths that cross the FPGA without being captured by any sequential element. Since we do not have any purely combinational path in our design, just click **Next**
13. The eighth window – **Physically Exclusive Clock Groups** – deals with situations where the design has two primary clocks assigned to the same physical pin that, logically, cannot be used simultaneously. Since we only have one clock in our design, just click **Next**
14. The ninth window – **Logically Exclusive Clock Groups with No Interaction** – deals with situations where the design has more than one clock active at the same time that is defined on different source points but shares part of their clock tree, due to a multiplexer for example. Since we only have one clock in our design, just click **Next**
15. The tenth window – **Logically Exclusive Clock Groups with Interaction** – deals with situations where the design has more than one clock active at the same time except on shared clock tree sections. Logically exclusive clocks share timing paths in addition to their shared clock tree. Since we only have one clock in our design, just click **Next**
16. The eleventh window – **Asynchronous Clock Domain Crossings** – deals with situations where data is transferred between two different clock regions and whose clocks have no known phase relationship: they are asynchronous. Since we only have one clock in our design, just click **Next**
17. The last window shows a summary of the newly created ten timing constraints. To review the timing constraints, just clicked on each one of the hyperlinked lines
18. To keep these new timing constraints, click **Finish** to save them in the constraints file – **top_counter.xdc**

After creating the timing constraints, we must constrain our pin locations, to indicate to the design which pins each one of our **counter** signals should be connected.

41. Assigning I/O Pin Locations to the Top-level Module

We must now create pin assignments for all the input and output signals of our **top_counter** module, the same way we did in section 25.

1. If the I/O ports window is not visible in the bottom of the main window, open it by selecting in the menu bar the **Layout** menu and choosing **I/O Planning** (if **I/O Planning** is not visible in the drop-down menu in the main toolbar, in the **Flow Navigator** window, under the **Synthesis** section, choose **Open Synthesized Design**; if the synthesis is out-of-date, it asks to run synthesis again before proceeding and to open the **Synthesized Design** window in the main window)
2. In the **I/O Ports** window, located in the bottom of the main window, we have a list of all I/O signals that are part of our **top_counter** design
3. Expand the **count_out(16)** output bus, to have access to all the individual bus signals, and the **Scalar Ports** trees under **All ports**
4. Locate and select the **clk** input signal under **Scalar Ports**, then attribute to this signal the pin location **W5** (GCLK signal on the Basys-3 board - see the Basys-3 FPGA Board Reference Manual to know the pin assignment in the board) by writing **W5** (or selecting it from the list) in the **Package Pin** column, as **I/O Std → LVCMOS33** and as **Vcco → 3.300** (this last value should change automatically after changing the **I/O Std** value)
5. Locate the **reset** input signal under **Scalar Ports**, then attribute to this signal the pin location **U18**, connected to the BTNC push-button on the Basys-3 board, as **I/O Std → LVCMOS33** and as **Vcco → 3.300**
6. Locate the **en** input signal under **Scalar Ports**, then attribute to this signal the pin location **V16**, connected to the SW1 switch on the Basys-3 board, as **I/O Std → LVCMOS33** and as **Vcco → 3.300**
7. Locate the **direction** input signal under **Scalar Ports**, then attribute to this signal the pin location **V17**, connected to the SW0 switch on the Basys-3 board, as **I/O Std → LVCMOS33** and as **Vcco → 3.300**
8. Concerning the **count_out(16)** output bus signals, the aim is to connect the outputs of our counter to the 16 LEDs of our Basys-3 board. To do it, repeat the previous pin assignment step to place the following additional input and output pins using the following information from Board Reference Manual:
   - **count_out[15]** (LD15 LED on the board) **Package Pin → L1**, **I/O Std → LVCMOS33**, **Vcco → 3.300**

- **count_out[14]** (LD14 LED on the board) **Package Pin → P1**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[13]** (LD13 LED on the board) **Package Pin → N3**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[12]** (LD12 LED on the board) **Package Pin → P3**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[11]** (LD11 LED on the board) **Package Pin → U3**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[10]** (LD10 LED on the board) **Package Pin → W3**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[9]** (LD9 LED on the board) **Package Pin → V3**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[8]** (LD8 LED on the board) **Package Pin → V13**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[7]** (LD7 LED on the board) **Package Pin → V14**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[6]** (LD6 LED on the board) **Package Pin → U14**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[5]** (LD5 LED on the board) **Package Pin → U15**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[4]** (LD4 LED on the board) **Package Pin → W18**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[3]** (LD3 LED on the board) **Package Pin → V19**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[2]** (LD2 LED on the board) **Package Pin → U19**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[1]** (LD1 LED on the board) **Package Pin → E19**, **I/O Std → LVCMOS33**, **Vcco → 3.300**
- **count_out[0]** (LD0 LED on the board) **Package Pin → U16**, **I/O Std → LVCMOS33**, **Vcco → 3.300**

9. Before proceeding, verify that the **Fixed** box is checked for all pins
10. Save the newly created **I/O Planning** constraints into the constraints file by selecting **File → Save Constraints**

11. Select the **Sources** tab and under **Constraints (2) → top_counter (1) (active)** click on **top_counter.xdc (target)**
12. The **top_counter.xdc** file opens in the main window. Check that all the timing and I/O constraints appear in the file

Notice that the constraints we defined for the **counter** implementation are exactly the same we need for the **top_counter** implementation, since we are using in the **top_counter** module the same clock – the board clock defined as **clk** signal in both the **counter** VHDL description and the **top_counter** VHDL description – and the same I/O pins we used in the **counter** implementation. Therefore, we may open both files – **counter.xdc** and **top_counter.xdc** – in the main project window and just copy and paste the content of the **counter.xdc** file to the **top_counter.xdc**. In the end, save the file by selecting **File → Save Constraints** or click in the save file icon in the corner of the **top_counter.xdc** constraint file window.

42. Adding the Configuration Bank Voltage Select to the new Constraint File

To configure the configuration interface:

1. If not open, **Open Synthesized Design** in the **Flow Navigator** window
2. In the menu bar select the **Tools** menu and choose **Edit Device Properties…**
3. In the vertical toolbar of the new **Edit Device Properties** window select **Configuration**
4. In the **Configuration Setup** section, go to **Configuration Voltage** and in the drop-down menu choose **3.3**
5. Then go to the **Configuration Bank Voltage Selection** and in the drop-down menu choose **VCCO** (see Figure 35 if needed)
6. Click **OK**
7. Save these new constraints into the constraints file by selecting **File → Save Constraints**

43. Top-level Module Implementation

Now that all the necessary constraints have been defined, let's continue with the implementation of the design. For more details, review section 27.

1. In our case, we use the **Vivado Implementation Defaults** as implementation strategy, so there is no need to change any **Implementation Settings**
2. Check only that in the **Synthesis Settings**, accessible in the **Flow Navigator** window, in the **Project Manager** section, **Settings → Project Settings → Synthesis**, the **Default constraint set:** active is the **top_counter**, to be sure that the new constraint set defined for the **top_counter** design is the one the tool is going to use in the implementation (Figure 51)

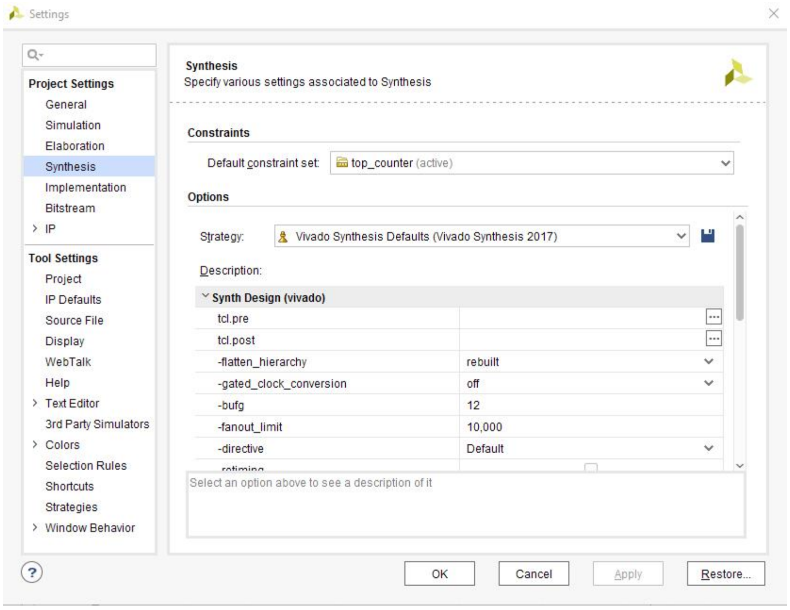

Figure 51

3. We may now proceed and run the implementation
4. In the **Flow Navigator** window, in the **Implementation** section, click **Run Implementation**
5. At the bottom of the main window select **Log → Implementation** to follow implementation evolution
6. In the end, a new window opens stating that the implementation was successfully completed and showing the next steps. Click **OK** to open the **Implemented Design**
7. A window with a view of how our logic was placed and routed inside the **Device** appears in the main window. Use the different tools available in the horizontal toolbar to zoom in and out and to view the cell connections
8. In the **Flow Navigator** window, in the **Implementation** section, click **Open Implemented Design** to expand the process hierarchy
9. A number of different reports may be generated now that the implementation of the design in the FPGA is done. Notice that the same list also appeared after synthetizing the design, under the **Open Synthesized Design** entry in the **Flow Navigator**. These reports now contain the most accurate information about the real behavior of our top-level design

44. Top-level Module Post-Implementation Design Simulation

We may repeat the simulation of our design as it will now, after place and route, take into account the exact influence of the propagation delays inside our FPGA. Go to section 37 – Simulating the New Top-level Module – and repeat the indicated steps to simulate the design, but instead of choosing, in the seventh step, **Run Post-Synthesis Functional Simulation**, choose **Run Post-Implementation Timing Simulation** (you may now go for a full lunch!). Notice the waveform fluctuation of the values in the **count_out[15:0]** output before stabilization after almost each increment of the counter

45. Creating Top-level Module Configuration Data and Device Configuration

After implementation we need to create the configuration bitstream for downloading to the target device. For details, review section 29.

1. In the **Flow Navigator** window, in the **Program and Debug** section, click **Generate Bitstream**
2. At the bottom of the main window select **Log → Implementation** to follow bitstream generation evolution
3. In the end, a new window opens stating that the bitstream generation was successfully completed, which means that the bitstream file (in this tutorial, the **top_counter.bit** file) that contains the actual configuration data was created
4. Before proceeding, connect the Xilinx USB cable between a USB port of your computer and the **PROG** USB port of your board and switch **ON POWER**
5. Then, choose **Open Hardware Manager** and click **OK**
6. In the top left of the new **Hardware Manager** window, select **Open target → Auto Connect** (see Figure 37)
7. The target device is automatically connected and it is ready to be configured
8. In the top left of the **Hardware Manager** window, select **Program device** and choose the target device
9. In the new **Program Device** window, check that the configuration file is the **top_counter.bit** file and click **Program**

Check in the **Tcl Console** at the bottom of the main window the evolution of the device configuration. In the end, LED LD(15..0) on the board lit up, indicating that the counter is running. If not, remember that when the enable signal **en** is disabled (SW1) the LEDs do not lit, so switch SW1. Switching SW0 the counting direction changes. We are able to stop the counter changing the state of switch SW1. Pressing the BTNC push-button resets the counter.

Finally, everything seems to be working as expected!

You have completed the Vivado Quick Start Tutorial for the Digilent Basys-3 board. For an in-depth explanation of the Vivado design tools, check the references below in the Xilinx web site

References:

Design Flows Overview, *Vivado Design Suite User Guide*, UG892 (v2017.2) June 7, 2017

Getting Started, *Vivado Design Suite User Guide*, UG910 (v2017.2) June 7, 2017

Using the Vivado IDE, *Vivado Design Suite User Guide*, UG893 (v2017.2) June 7, 2017

System-Level Design Entry, *Vivado Design Suite User Guide*, UG895 (v2017.2) June 7, 2017

Logic Simulation, *Vivado Design Suite User Guide*, UG900 (v2017.2) June 7, 2017

Logic Simulation, *Vivado Design Suite Tutorial*, UG937 (v2017.2) June 7, 2017

Synthesis, *Vivado Design Suite User Guide*, UG901 (v2017.2) June 7, 2017

Using Constraints, *Vivado Design Suite User Guide*, UG903 (v2017.2) June 7, 2017

I/O and Clock Planning, *Vivado Design Suite User Guide*, UG899 (v2017.2) June 7, 2017

Implementation, *Vivado Design Suite User Guide*, UG904 (v2017.2) June 7, 2017

UltraFast Design Methodology, *Vivado Design Suite User Guide*, UG949 (v2017.2) June 7, 2017

Design Analysis and Closure Techniques, *Vivado Design Suite User Guide*, UG906 (v2017.2) June 7, 2017

Using Constraints, *Vivado Design Suite Tutorial*, UG945 (v2017.2) June 7, 2017

Version 4

Manuel Gericota – October 2017

Revision table

v. 1	Introduction of the section “19. Saving the wave configuration”	October 2016
v. 2	Updated Editing Timing Constraints section New notice about constraints precedence rules	February 2017
v. 3	Updated version for Vivado 2017.2 Updated reference list	July 2017
v. 4	Revision of the output delays constraint values New figure 32 New “Improving code type productivity” section	October 2017