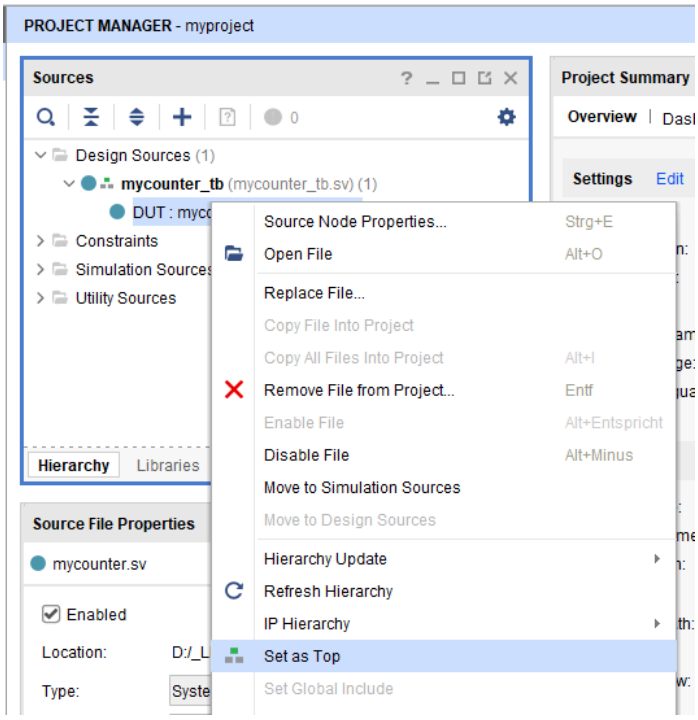
Vivado Tutorial 2 – Synthesis, Implementation & Bitstream generation

Open the project with the basic counter that we simulated in Tutorial 1 (or redo those steps).

Switch back to the “Sources” window by selecting the “Project manager” or just by closing the Simulation results.

There, open the “Design Sources” folder and set the counter module as the top module!



Now we want to realize this counter physically on the hardware.

We can do so by connecting all the inputs and outputs of the counter to physical inputs and outputs of the Basys3 board. We can, for example, connect the pin W5 of the FPGA to the internal clock signal, since the reference manual of the basys3 board tells us that a 100MHz clock signal is connected to the FPGA via this pin.

See page 7 (section 4) of the reference manual: https://virtueller-campus.fh-joanneum.at/2019-20/pluginfile.php/33544/mod_folder/content/0/basys3_rm.pdf?forcedownload=1

The outputs of the counter we can connect to some LEDs or to the signals of the PMOD edge connectors:

See page 15 (section 8) of the reference manual for all the LEDs, buttons, switches and the 7-Segment display. See page 17 (section 9) of the reference manual for the PMOD connectors.

However, we have a problem with the reset and the enable input signals. While we could just directly connect them to any button or switch directly, we would be connecting asynchronous signals to synchronous input ports. Even though it is unlikely that it would cause a huge problem with this simple circuit, neglecting to address such issues can and will cause problems in bigger, more interconnected systems.

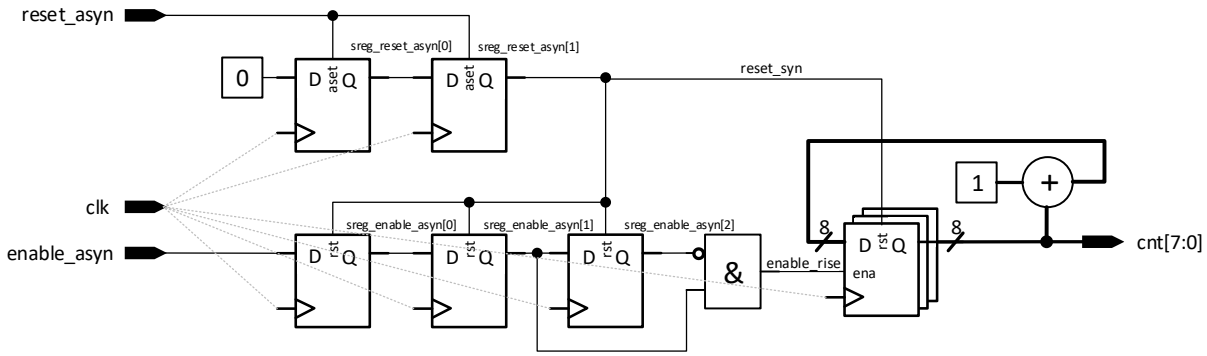
To solve this problem, we will synchronize the external signals via two stage shift registers. While the inputs will be synchronized to the core clock, they will still suffer from bouncing (multiple accidental actuations due to mechanical oscillations within buttons or switches).

Bouncing will not affect our reset in a bad way, because whenever we want to reset the circuit, we do not care if it is actually reset multiple times.

Bouncing will however become visible when we connect the enable of the counter to an edge detection circuit of a button. With such a circuit we could make the counter only count +1 whenever we are pressing the associated button instead of it incrementing constantly at 100MHz.

So we can expect that the counter might increment either once, twice or maybe even multiple times, as long as we do not make a de-bouncing circuit.

The circuit we want to implement:



Notice that our active high asynchronous reset „reset_asyn“ is synchronized via a two stage shift register. The supply based design of the synchronizer ensures that the asynchronous reset is kept asserted for at least two clock cycles and the internal reset will always assert in sync with the core clock. Therefore, we can connect the internal signal “reset_syn” to our synchronous reset inputs.

This internal “reset_syn” signal resets the other input shift register and the counter register.

When the output of the “enable_asyn” shift register is zero and the value in the flip flop before is one, a rising edge must currently be shifting in. With an AND and a NOT gate we can detect this situation and use it to enable the counter circuit.

OK now let’s describe this circuit:

Notice that we use the following vector assignment, combined with a concatenation to create the two shift registers:

```
...
else
  sreg_reset_asyn <= {sreg_reset_asyn[0], 1'b0};
  //this assignment takes the value stored in position 0 of the shift register
  //and writes it to position 1, it takes the constant 1’b0 and writes it into
  //position 0.
  //or in other words a new vector is created by concatenating the value in
  //position 0 with the constant zero, giving the value 2’bX0 (where X is
  //whatever was within position 0 before. the register is overwritten with this
  //new value, effectively resulting in a shift register.
```

Otherwise we would have to use a dedicated line of code for each flip flop within the shift register plus and additional begin end statement to fit everything into the else case of the if condition:

```
...
else
begin
  sreg_reset_asyn[0] <= 1'b0; //first stage of shift register, shifts in zeros
  sreg_reset_asyn[1] <= sreg_reset_asyn[0];   //second stage of shift register
end
```

Source code and Test bench

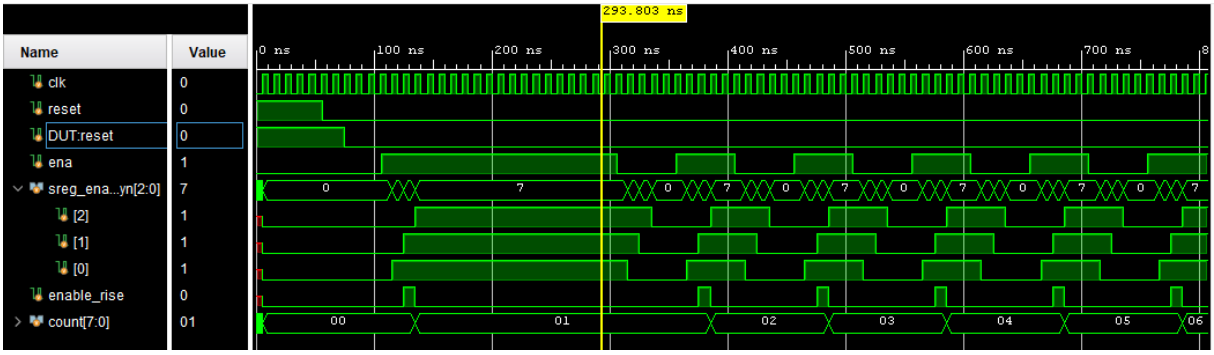
Modified source HDL code of our counter circuit

```
module mycounter (
     input clk, reset_asyn,     //clk and asynchronous active high reset
     input enable_asyn,
     output logic [7:0] cnt
     );
logic reset;                    //synchronized signal of reset_asyn input
logic [1:0] sreg_reset_asyn;    //shift register for reset_asyn
logic enable_rise;              //synchronized result of edge detection
                                //of enable_asyn input
logic [2:0] sreg_enable_asyn;   //shift register for enable_asyn


//logic to synchronize an asynchronous active high reset
//(reset of this block is asynchronous)
always_ff @(posedge clk, posedge reset_asyn)
begin
    if (reset_asyn)              //if the reset is asserted:
       sreg_reset_asyn <= 2'b11; //set both FFs of to one (assert output reset)
    else                         //else shift zeros into the shift register
        sreg_reset_asyn <= {sreg_reset_asyn[0], 1'b0};   //shift register
end
//output of shift register is our internal synchronous active high reset signal
assign reset = sreg_reset_asyn[1];  //connect to our internal reset signal.


//logic to synchronize an asynchronous input
//(reset of this block is synchronous)
always_ff @(posedge clk)
begin
    if (reset)
        sreg_enable_asyn <= 0;
    else
        sreg_enable_asyn <= {sreg_enable_asyn[1:0], enable_asyn};
end
//rising edge detection circuit for enable_asyncc
assign enable_rise = ~sreg_enable_asyn[2] & sreg_enable_asyn[1];


//counter circuit
//(reset of this block is synchronous)
always_ff @(posedge clk)
begin
    if (reset)
        cnt <= 0;
    else if (enable_rise)
        cnt <= cnt + 1;
end

endmodule
```

Let's also modify the test bench, since the port names of mycounter have changed and since our single actuation of the enable signal would only increment the counter once.

```
1  module mycounter_tb ( );
2
3  logic clk = 0, reset = 1, ena = 0;
4  logic [7:0] count;
5
6  always #5ns clk = ~clk;  //create clk
7
8  mycounter DUT(
9       .clk(clk),
10      .reset_asyn(reset),  //port name has changed
11      .enable_asyn(ena),   //port name has changed
12      .cnt(count));
13
14 initial
15 begin
16
17   $dumpvars(1, mycounter);
18   $dumpfile("dump.vcd");
19   #56ns;
20   reset = 0;
21   #50ns;
22   ena = 1;
23   #200ns;
24   repeat(10)        //repeat toggling the ena signal 10 times
25     begin           //this should increment our counter 5 more times.
26       ena = ~ena;
27       #50ns;
28     end
29   $finish;
30 end
31 endmodule
```

Rerunning the simulation and adding the internal signals "sreg_enable_async","enable_rise" and the internal synchronous reset signal of the DUT, the wave form viewer will give us the following result:



Since the internal synchronized reset has the same signal name as the external asynchronous reset coming from the test bench, I have renamed the internal signal to "DUT:reset" within the waveform viewer.

Notice how each flip flop of the shift register contains a time delayed version of the original input data connected to "ena" (enable_asyn) input. Notice as well how the "enable_rise" signal is asserted for a single clock cycle, whenever a rising edge was detected.

Notice as well how the internal reset is delayed in relation to the externally applied one.

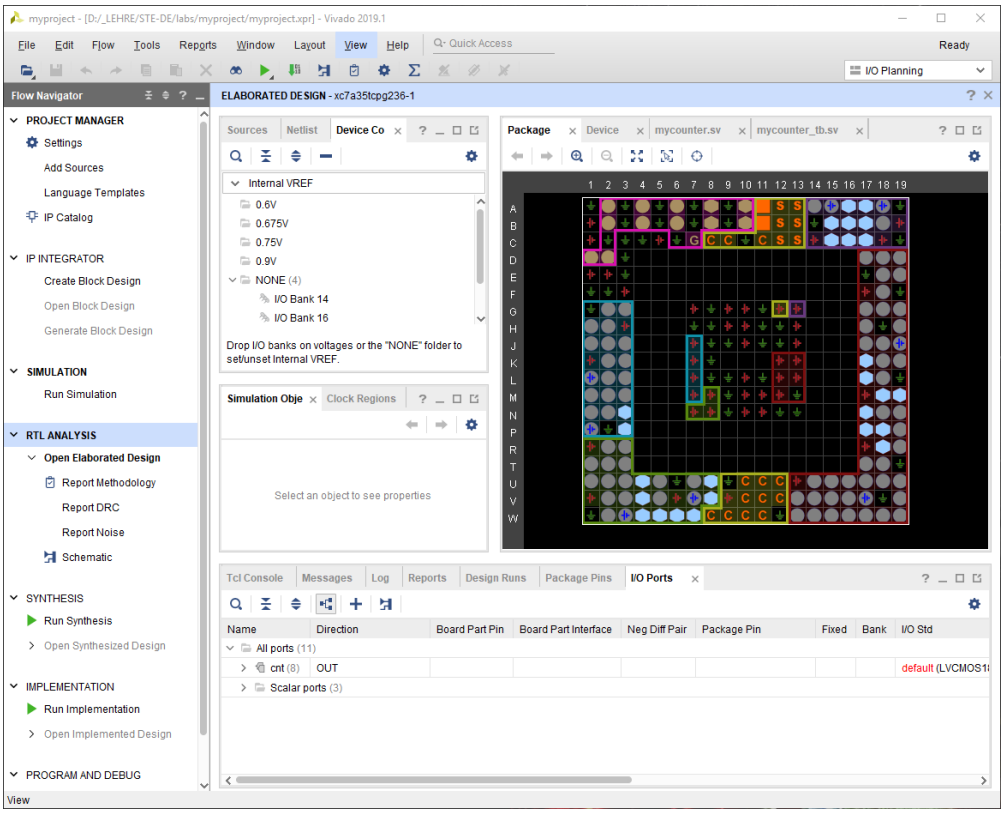
After every assertion of "enable_rise" the counter increments in the follow-up cycle.

Beware that now we have duplicate names for some signals. The signals in the test bench can be called differently than the ports to whom they connect!

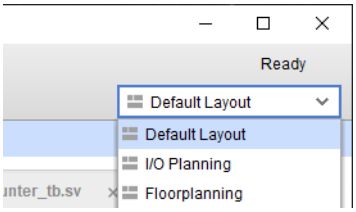
Since the circuit is still functional after our modifications we could now continue to bring it to the physical hardware of the basys3 board.

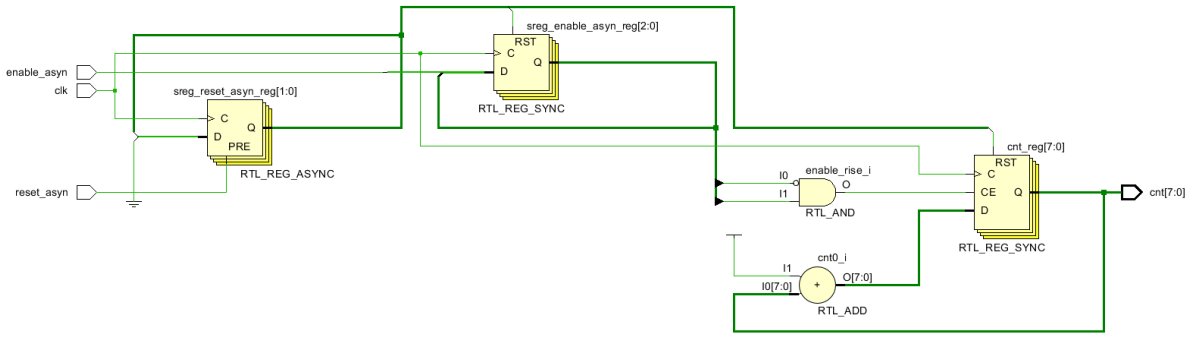
Elaboration and Constraints

Click on “Open Elaborated Design” and the following view will appear:



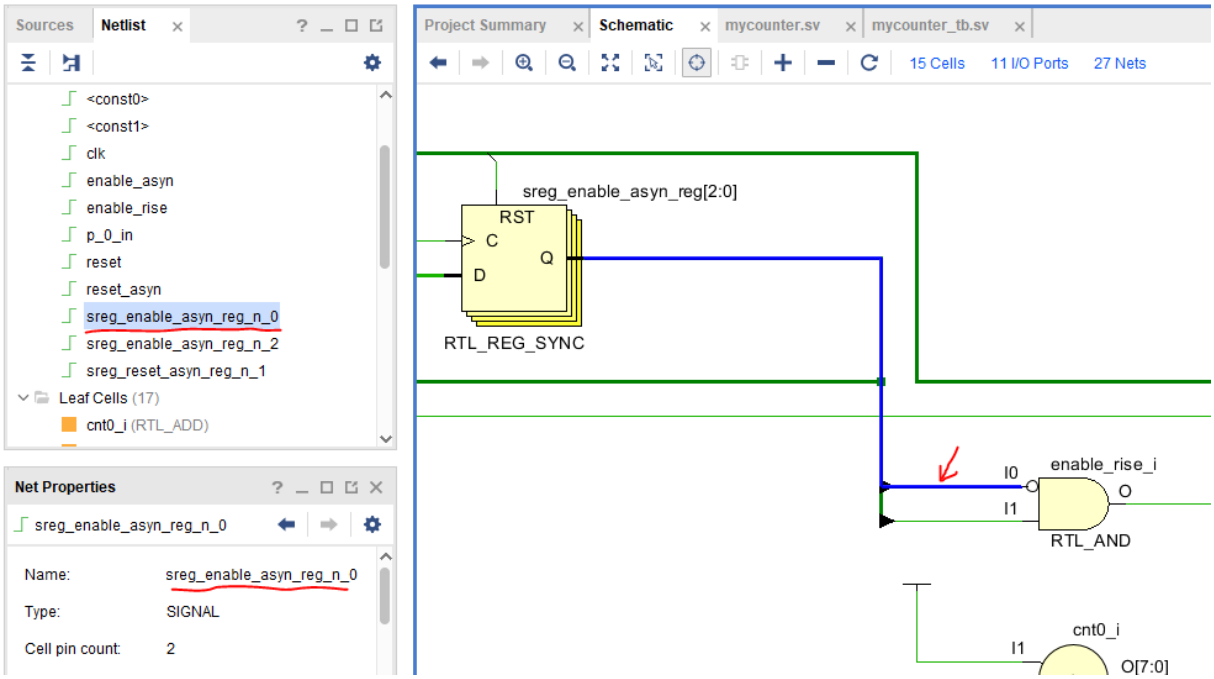
Switch the view to “Default Layout” to see the inferred schematic:



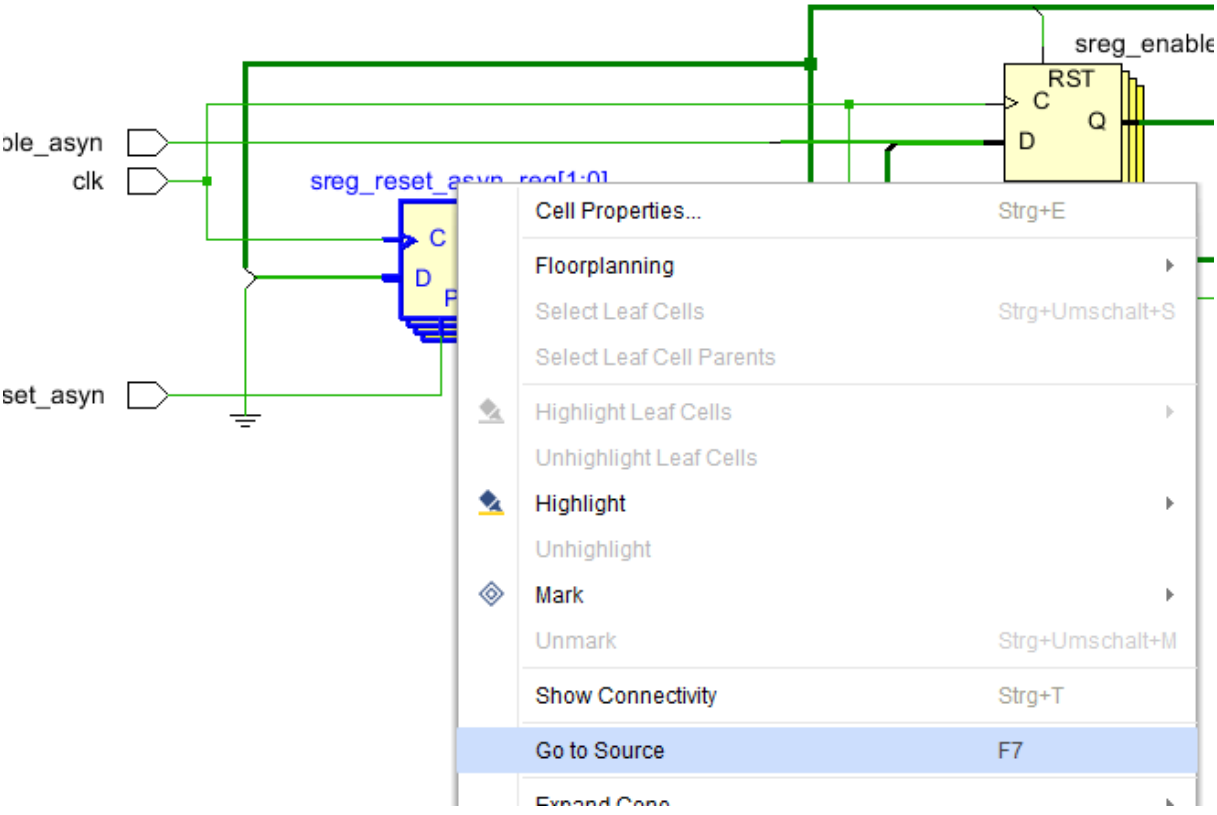


Notice how both shift registers as well as the counter are recognized as registers. While it is nice that the registers are shown as one piece the connections between them are somewhat hard to read, because the separate bits of the shift registers are always displayed as a bus.

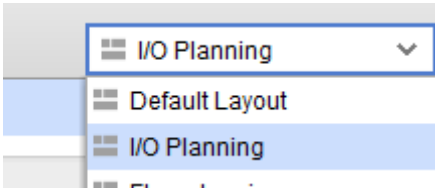
However, when clicking on any signal, the property window will show the signals name and the appropriate net will be highlighted within the netlist:

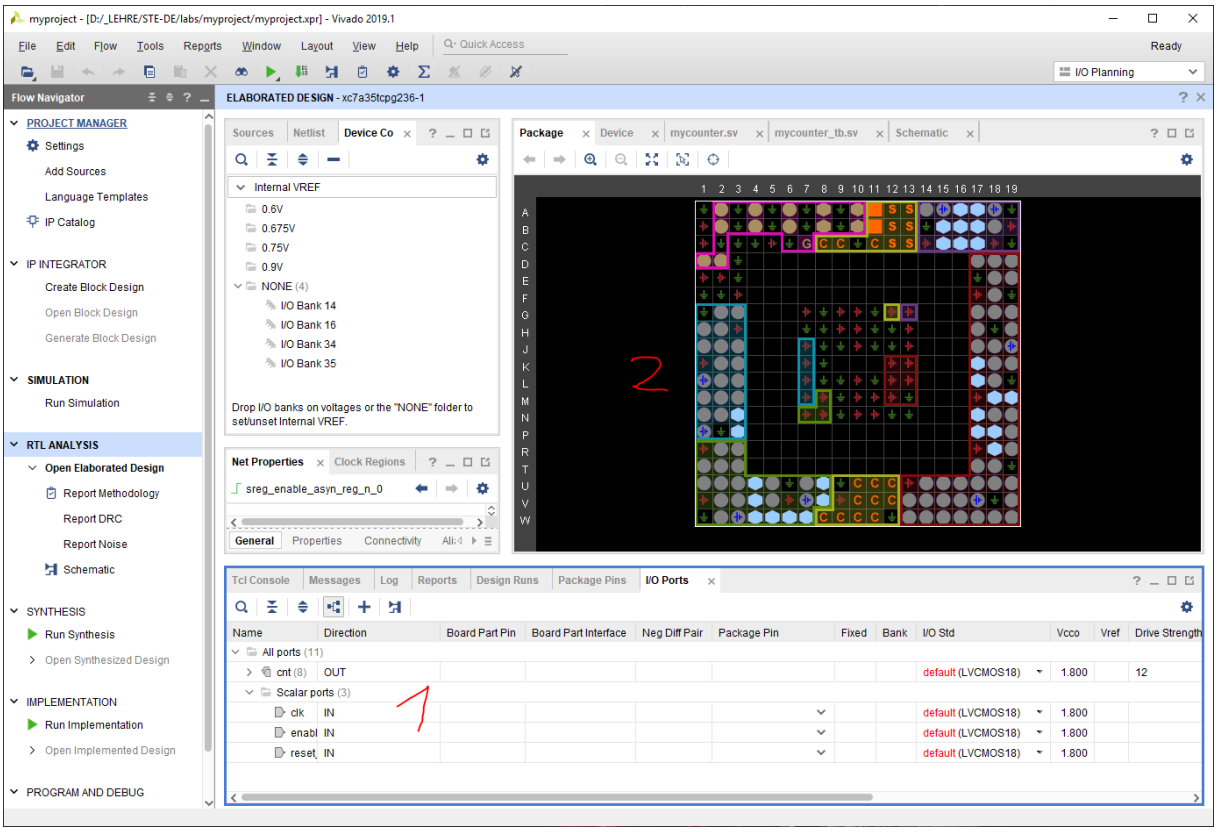


When doing a RMB klick on any component you can immediately go to the corresponding line within the source code.



Let’s now switch back to the I/O Planning layout view :





In the bottom window (1) in the tab “I/O Ports” we can see all input and output signals we have in our top level design unit “mycounter”. In the main window (2) we can see the top view of all the solder pads on the bottom of the FPGA device. Notice that all the FPGA pins are grouped into so called banks. Each bank has supply and ground pins as well as a set of input/output pins (grey circles). Clock capable input pins are shown in blue hexagonals.

We can now map the inputs and outputs of our counter module via drag and drop of the port names to the desired FPGA pin. Or, which works much better: we can assign them via their FPGA package pin name.

The Pins of the FPGA are labelled in a similar fashion as a chess board is labelled. Colums have numbers and rows have letters.

Now consult the reference manual to find out which hardware elements are connected to which package pins:

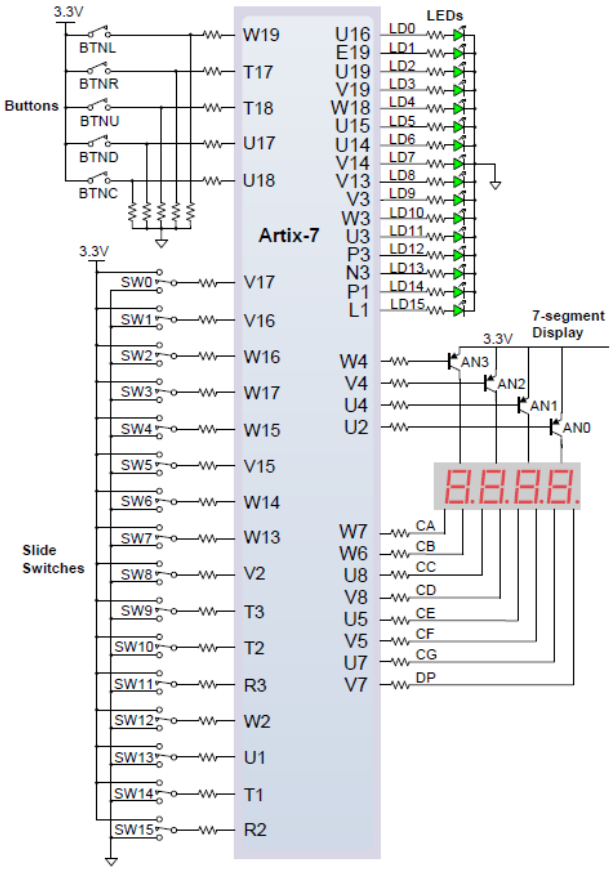
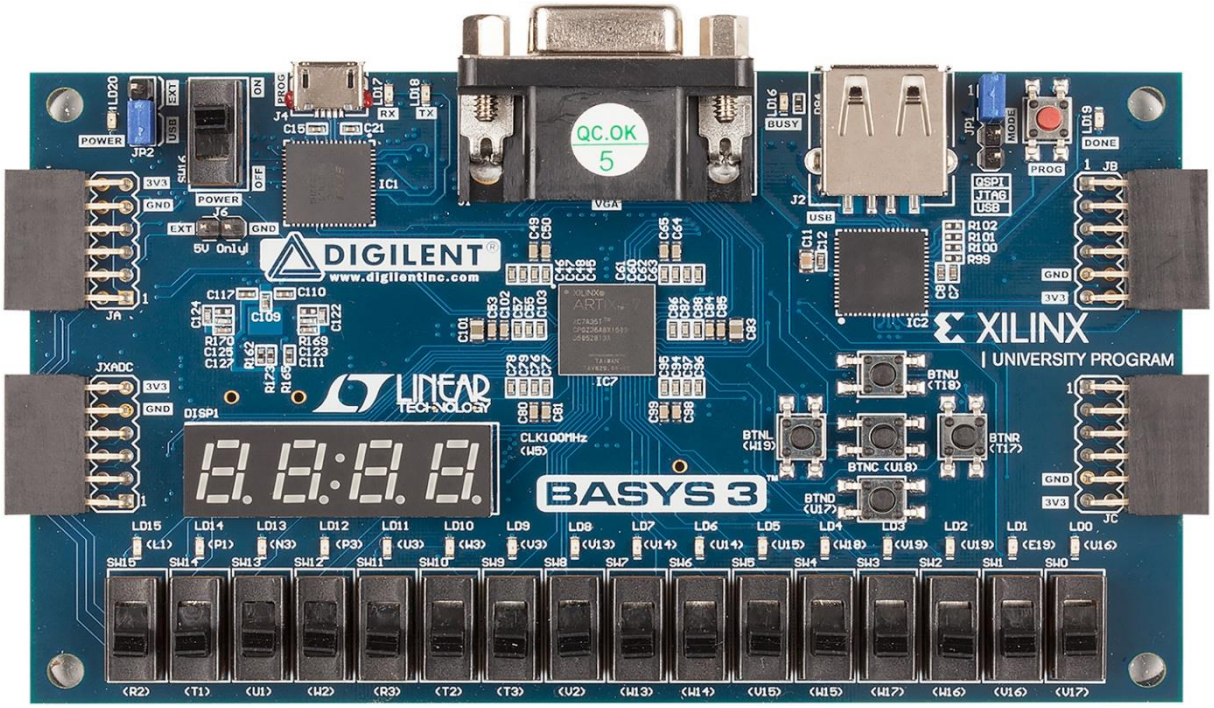


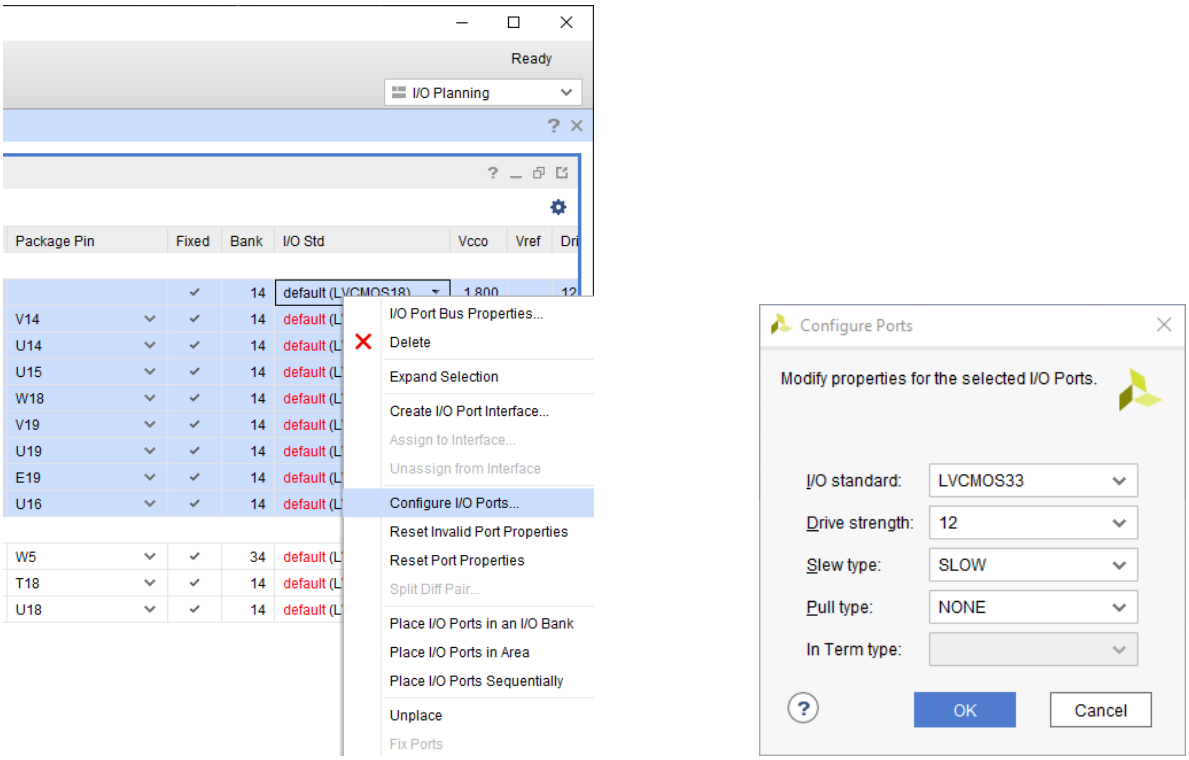
Figure 16. General purpose I/O devices on the Basys 3.

Or look at the silk screen print on the PCB itself:

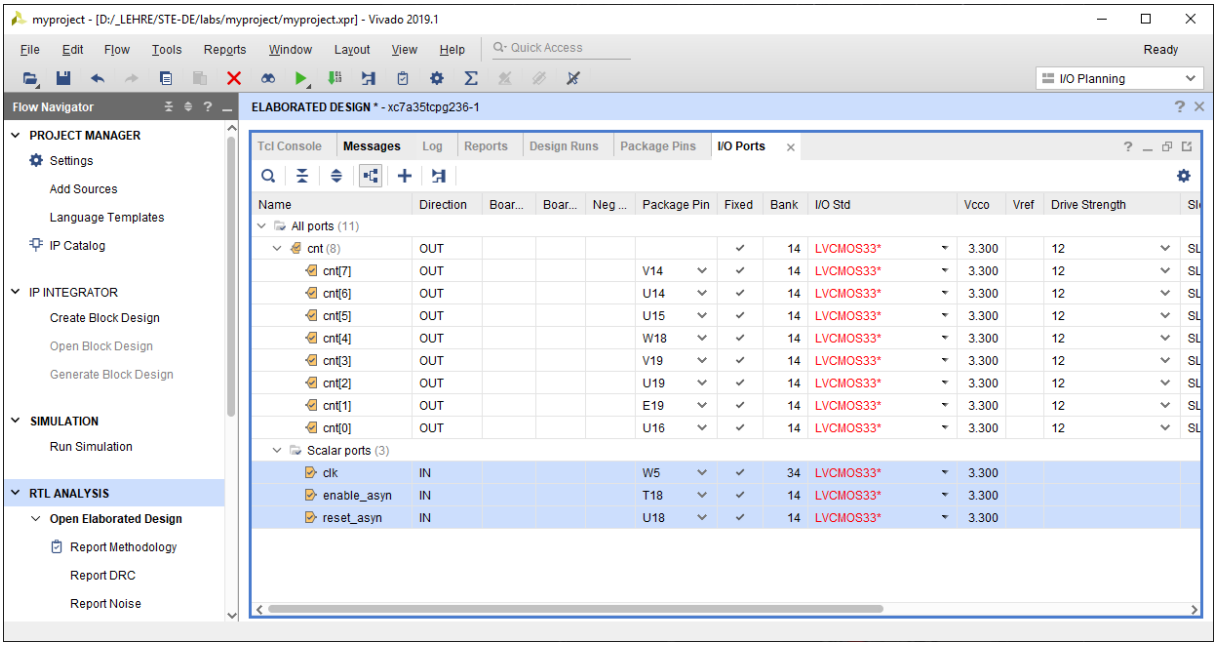


Be careful with the silk screen print. The letters U and V look very similar!

Let’s assign the counter outputs to LEDs LD0 to LD7, the reset to the center button and the enable to the up button. We must also configure the I/O Standard to be LVCMOS33, since the all the I/Os of the basys3 board work with 3.3V logic.



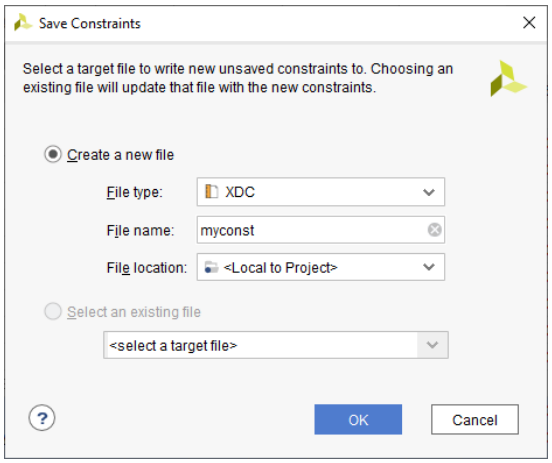
The result should look like this:



Now click save to store these settings. Vivado will ask you to give a name for this configuration file.

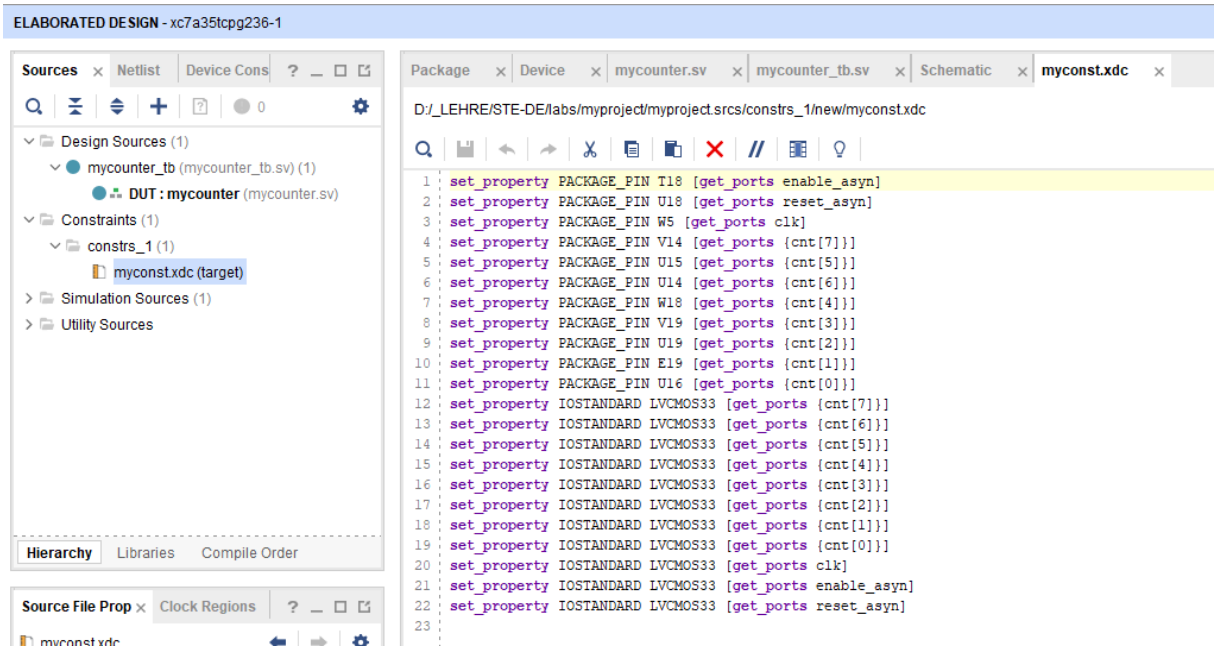
This configuration file contains all the constraints we apply to our design and therefore is called a constraint file (.xdc Xilinx constraint file).

Let's call it "myconst.xdc":



Vivado will create the file, add the appropriate instructions and add the file to the project.

Let us open this file and inspect it:



Notice how the set property command was executed for each signal twice, once to set the PACKAGE_PIN and once to set the IOSTANDARD.

However, there is no command that informs the tool about the clock frequency we are using.
Let's specify by ourselves by adding the line:

```
create_clock -period 10.000 -name clk100 -waveform {0.000 5.000} [get_ports clk]
```

Period is entered in nanoseconds, the waveform specifies the low and high transition points.
The name can be an arbitrary name

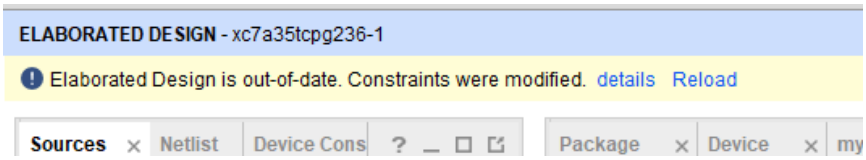
Furthermore, we can take the following configuration from the "Basys-3-Master.xdc" from Digilent:

```
## Configuration options, can be used for all designs
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]
```

It would also work without these but then the tool will give a warning!

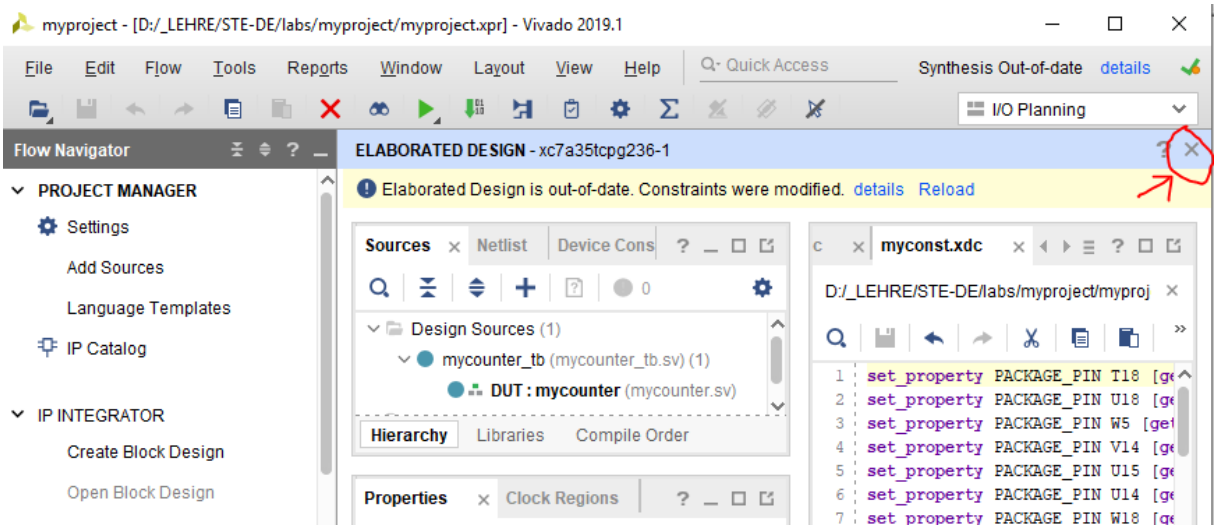
Save the modified constraints file.

Notice the yellow warning we get displayed in the elaborated design window!



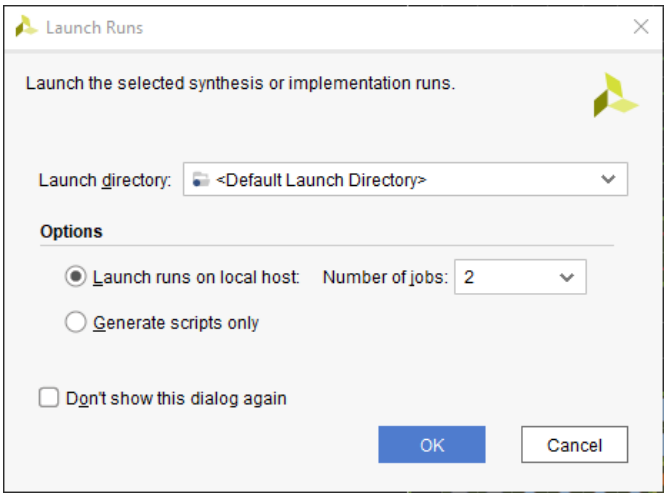
This informs you that you are looking at an outdated elaboration result. The new circuit could be different due to the constraints that we have made. (not in this case but in general)

We can ignore this and just close the elaborate design because we won't need it anymore:

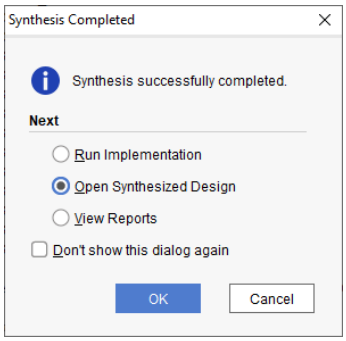


Synthesis

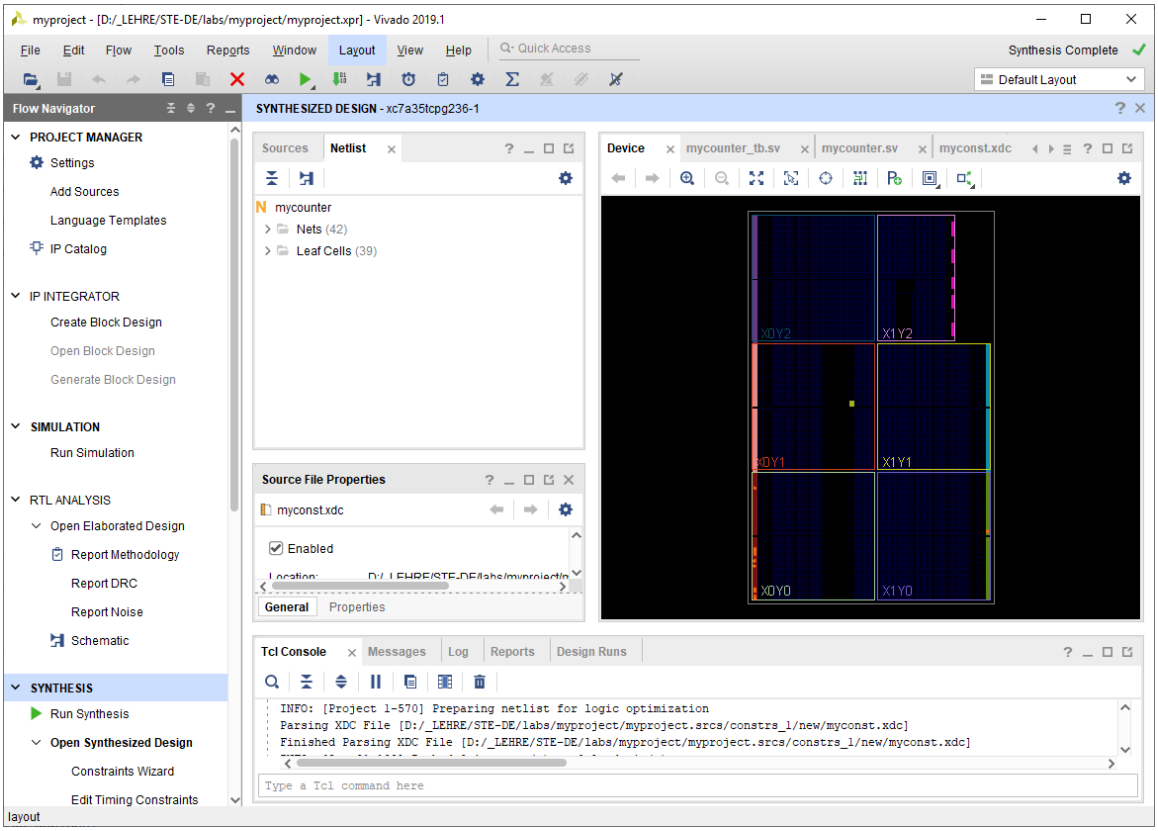
Now click on "Run Synthesis" (during Synthesis the elaboration step will be repeated, but our newly added constraints will be applied and then the circuit will be optimized:



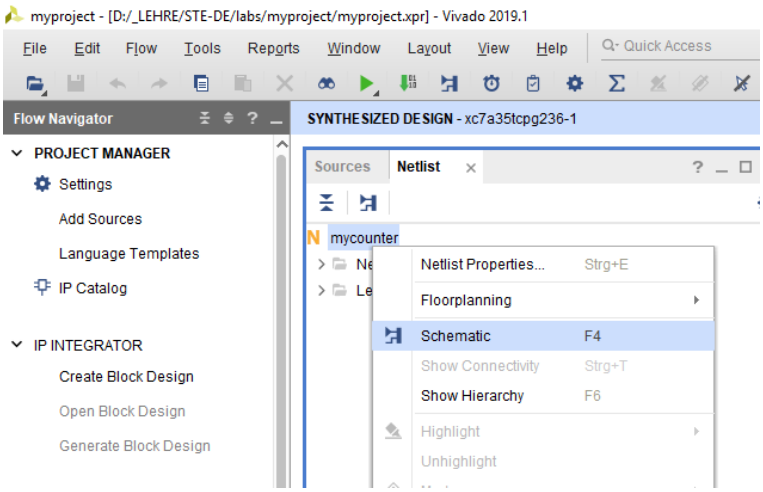
When the synthesis completed you can open the synthesized design:



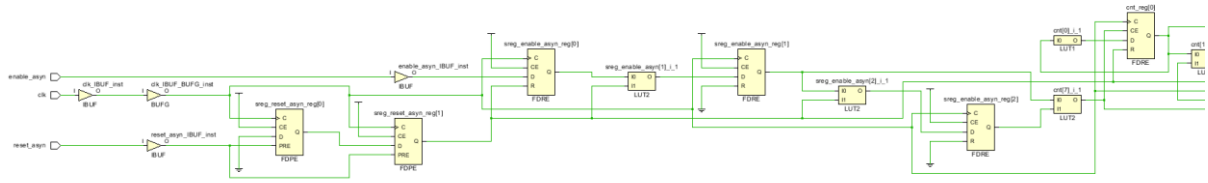
Here we see in the main window an overview of the different regions within the FPGA:

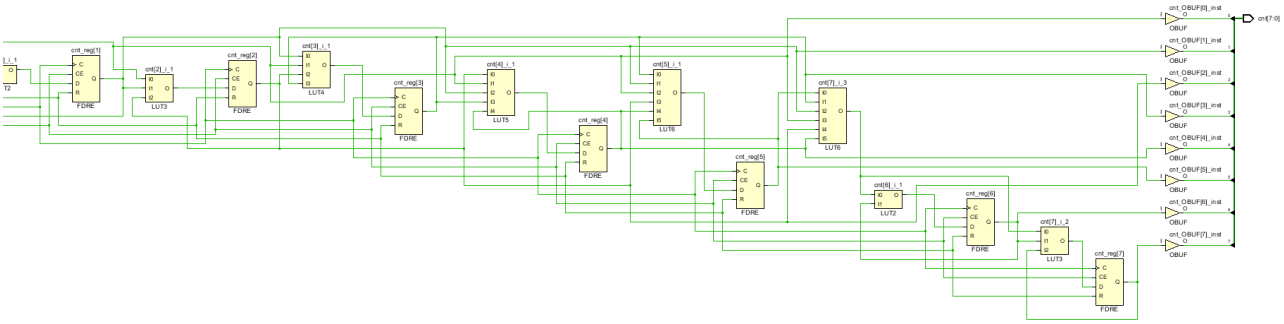


There is nothing to see yet within the “Device” view so let’s go to the schematic instead:



Notice how different the circuit now looks: The registers are now shown as dedicated flip flop components and the adder circuit is now represented by a couple of look up table components:

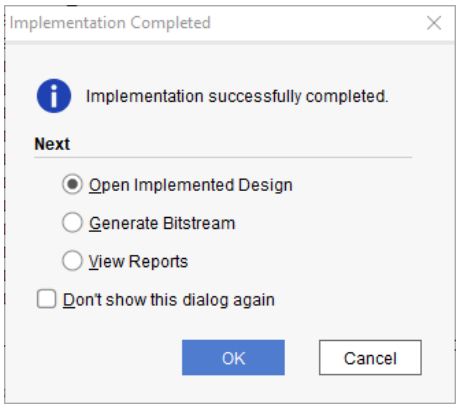




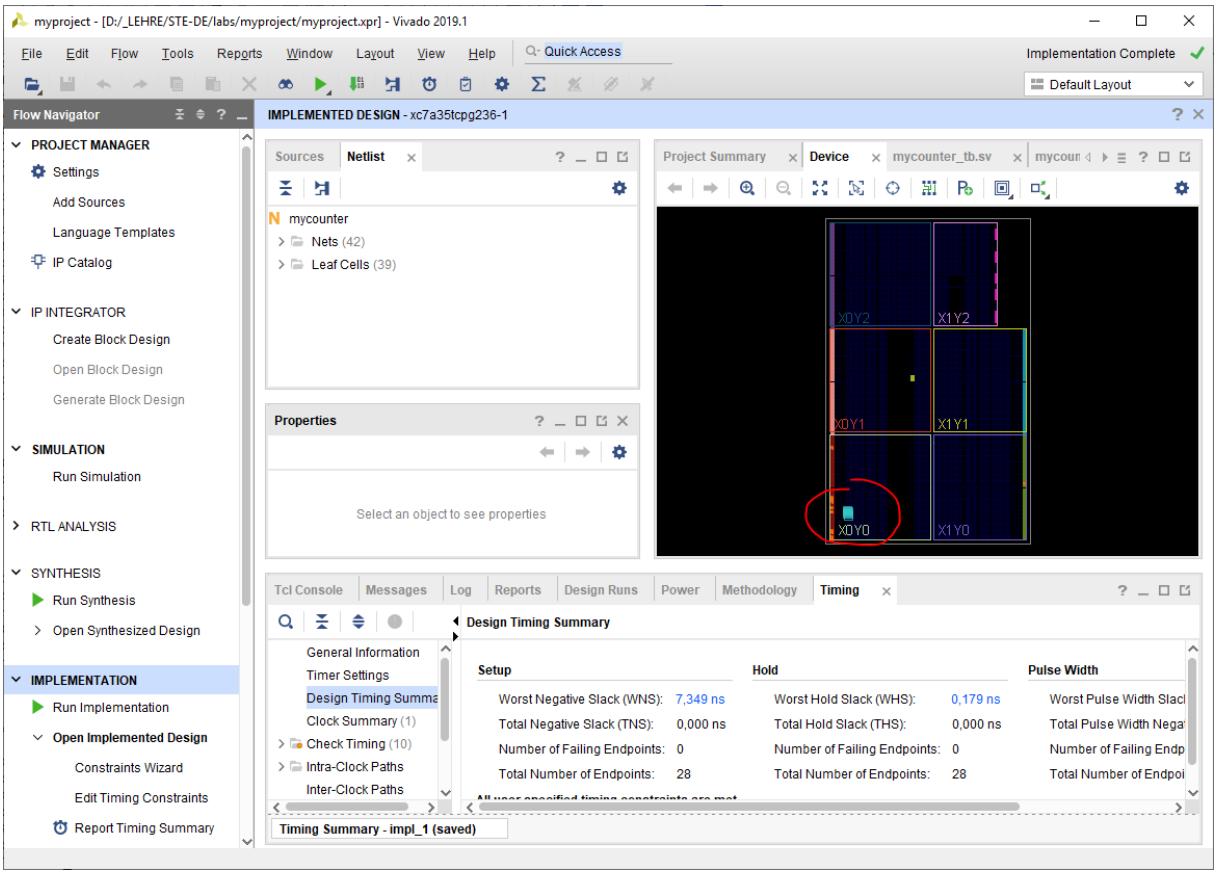
Implementation

Close the synthesized design and click on “Run Implementation”:

Open the implemented design:

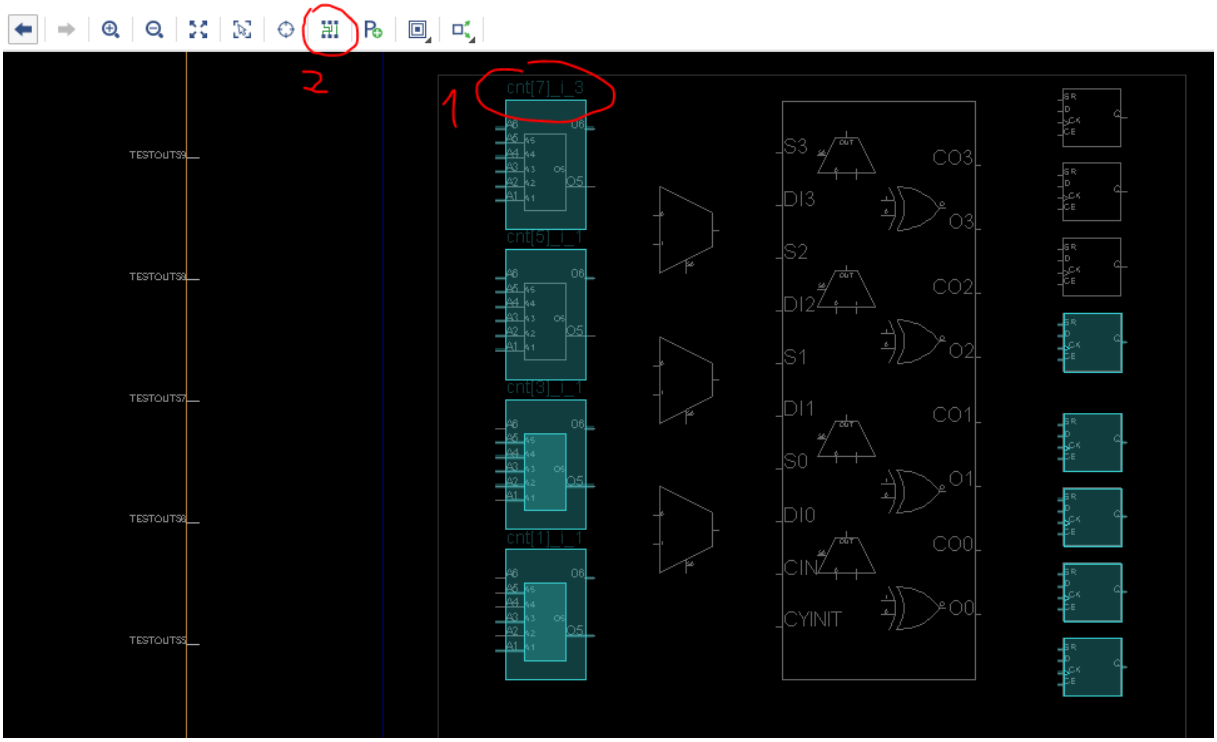


Notice the tiny circuit that appeared within the Device view:

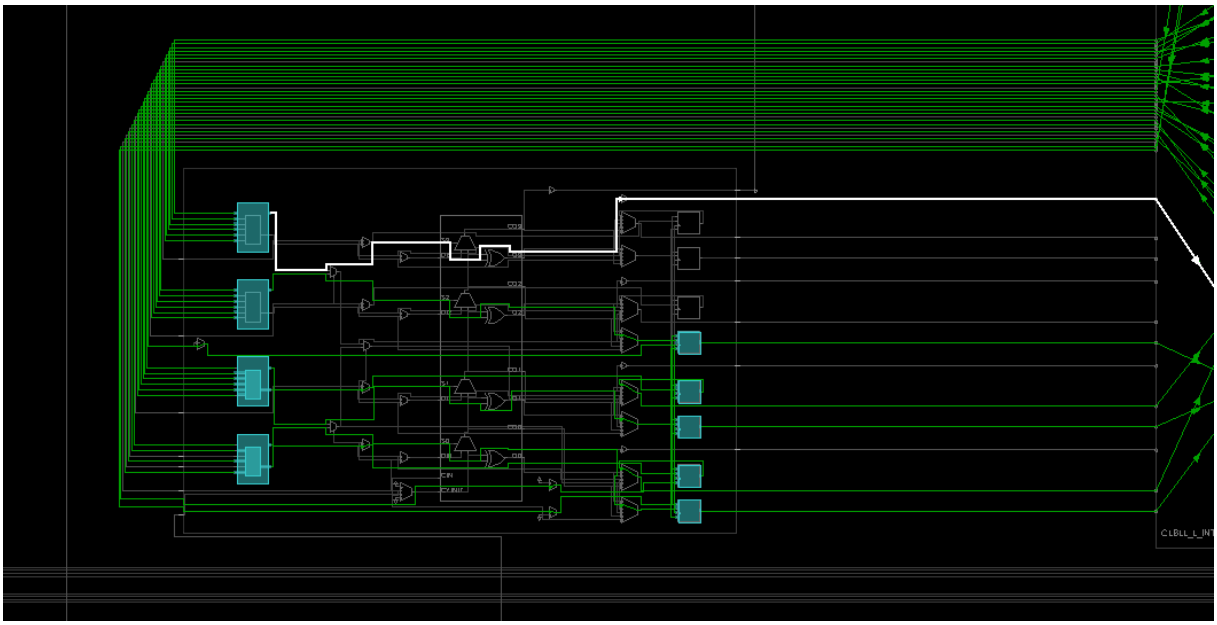


Let’s zoom into it!

Ah… we have found some lookup tables and some flipflops that belong to our counter circuit (1)!

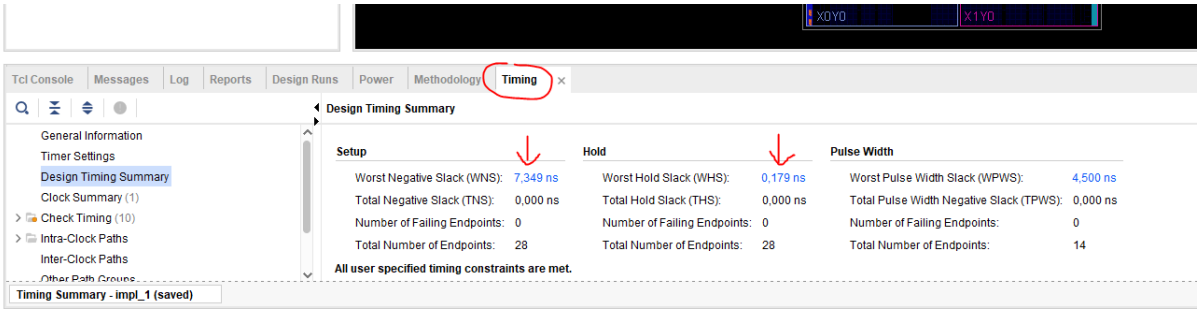


Click on the routing resources button (2) to see all the interconnections between the devices:



Timing analysis

Within the implemented design you should also take a look at the Timing report which you can find in the bottom status window:



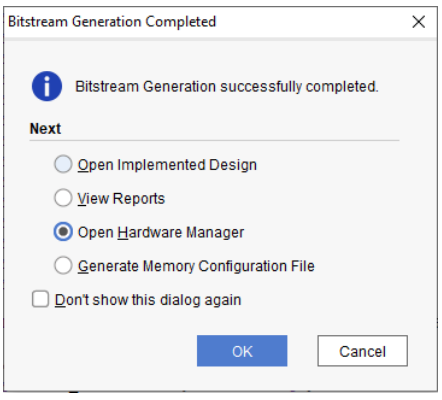
If the timing is ok the WNS and the WHS will be positive.

A negative WNS or WHS will be marked in red and you will get a warning message within the Message Window, saying that your design did not meet the timing requirements.

Since the timing is fine for our circuit let's generate a bitstream:

Generate Bitstream and Hardware Manager

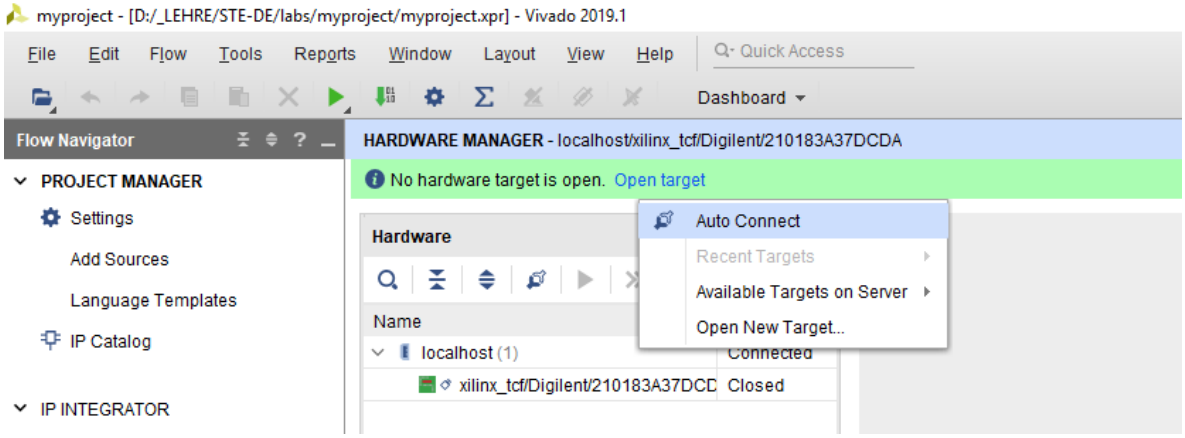
Close the implemented design and click on "Generate Bitstream"



Choose on “Open Hardware Manager” after the bitstream was completed.

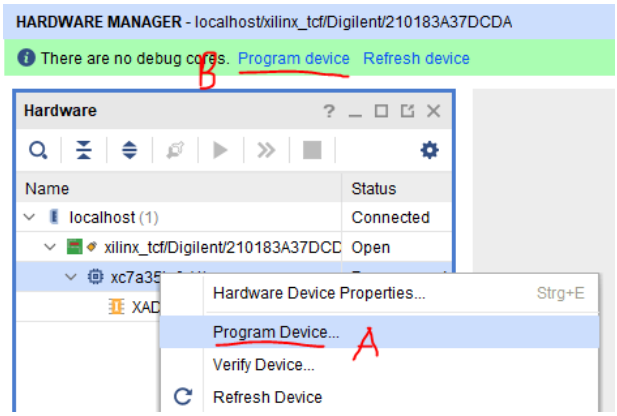
Connect the Basys3 board to the PC via USB and turn it on (Power switch: SW16)

Inside the hardware manager click on “Open target” and select “Auto Connect”:



The hardware will appear within the top left window

Open the context menu via RMB on the target device xc7a35 and select “Program Device” (A), or directly click “program Device” within the green notification bar:



The LED LD19 with the label DONE will be active after the FPGA is configured.

Now the counter is present within the FPGA. Try using the up-button to increment it. Observe the LEDs to see the binary value of the counter register. Does the counter increment by one on each actuation of the button, or does it sometimes increment by 2 or more?

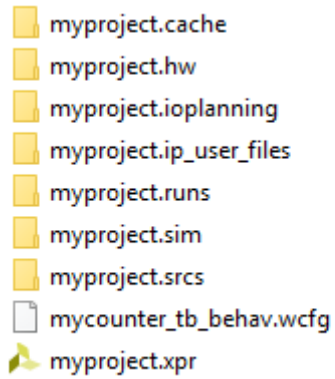
Try also to reset the counter via the center-button.

Remember, after power cycling the board it will have lost the configuration and it must be programmed again. To keep the bitstream permanently on the board, consider following the “Vivado Flash Bitstream Tutorial”.

Project folder structure:

Take a look at the folders that were created within the project folder through the Vivado toolchain.

Consider for example the .sim folder which was created after we started the simulation. It contains the simulation results and other data generated by the simulation run:



The .srcs folder was created after we imported or added the first source or constraints file.

The .runs folder was created after we did an elaboration/synthesis or implementation run. It contains all the netlists and reports that are generated after these runs completed. Also the .bit bitstream file can be found in here.

The other folders are used by Vivado to store other intermediate files.

Backup of Vivado Project files

Investigate the size of the .runs folder. Notice that all this generated files can consume a lot of disc space. If you want to copy or backup only the sources of your project you could delete all the generated files by deleting all folders except the .srcs folder. This will decrease the project folder to it's minimum size. When opening the project afterwards, all the synthesis and implementation runs will not be available anymore and these steps would have to be executed again