

Digital Electronic System Design – VIVADO notes

 Un ringraziamento speciale va ad Lucilla Bernardini e a Simone Messina che mi hanno aiutato nella scrittura e revisione di alcune parti del documento.

In questo file abbiamo sbobinato ed integrato tutte le lezioni di Vivado dell'Anno Accademico 2024/2025, in modo da avere dei passaggi passo-passo (con anche i pulsanti da usare nella GUI ecc...) per sfruttare tutte le funzioni che offre Vivado e che vengono usate nel corso.

NB: in questo file potrebbero esserci errori, imprecisioni, parti che potrebbero essere riscritte meglio etc. In caso vogliate contribuire in qualsiasi modo, contattatemi via la mail riportata.

 A special thanks to Lucilla Bernardini and Simone Messina that helped me in the writing and in the review of this document

In this file we have transcribed and integrated all the Vivado lectures from the Academic Year 2024/2025, so as to provide step-by-step instructions (including the buttons to use in the GUI, etc.) to make the most of all the functions offered by Vivado and used in the course

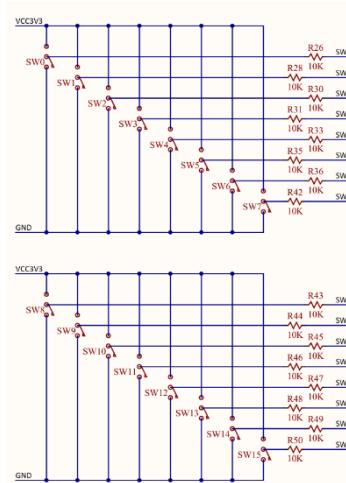
Note: This file may contain errors, inaccuracies, or sections that could be written more clearly, etc. If you would like to contribute in any way, please contact me at the provided email.

01_Vivado_Basics

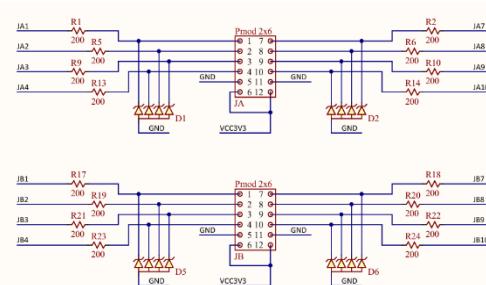
- [schematics of the board](#)
- [Datasheet of the board](#)

Port 13: JTAG port to program the FPGA.

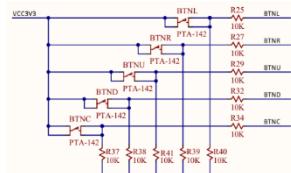
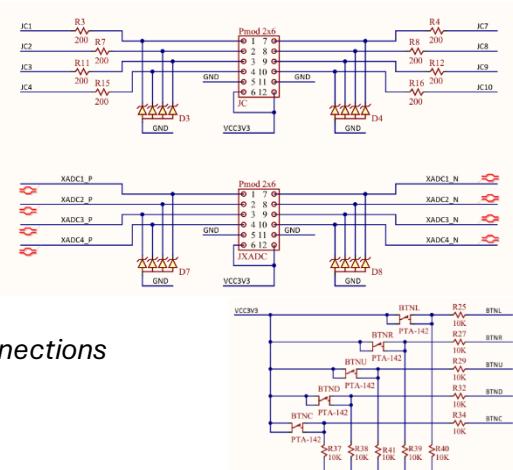
Schematic description



Switches in the schematic

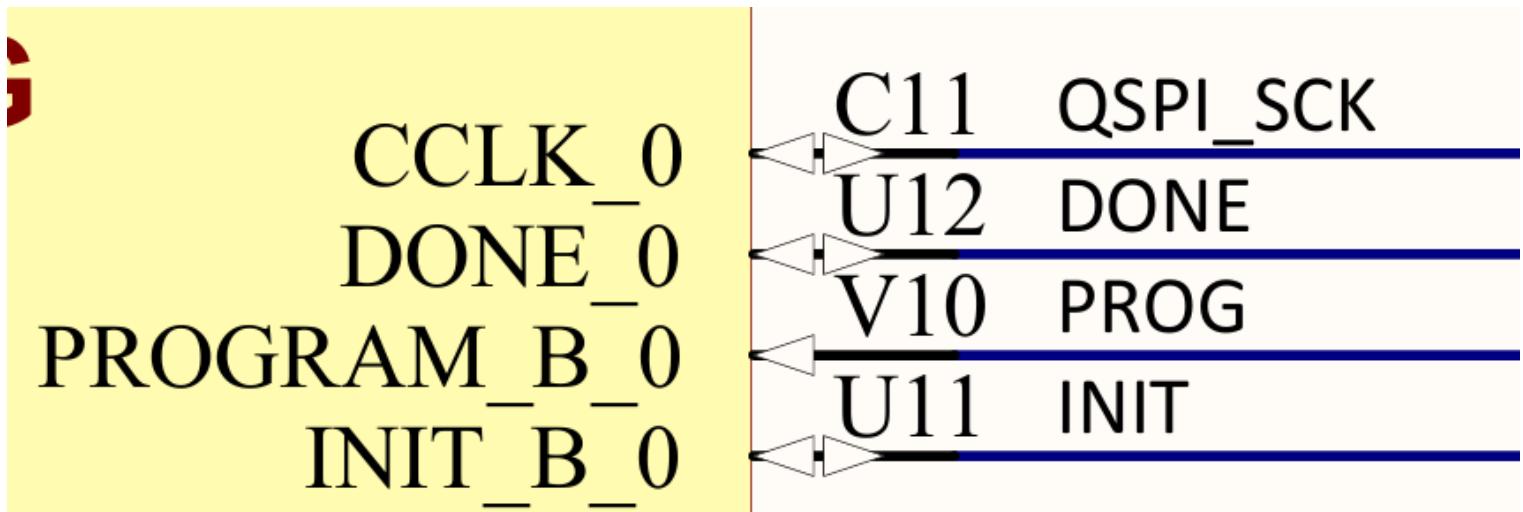


pushbuttons connections

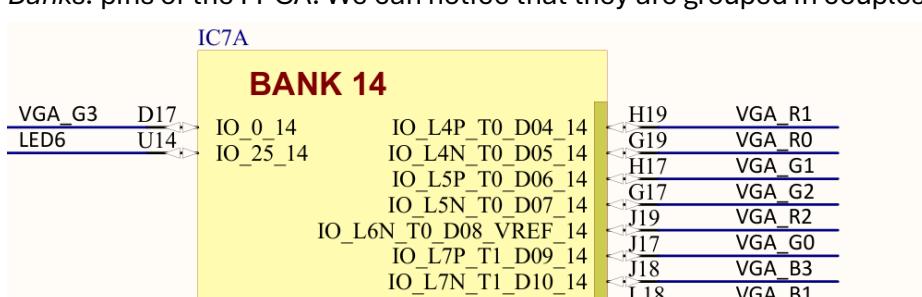


Each of the labels on the schematic represent a connection to a specific pin, in detail to the one of the FPGA

Example: quad SPI connection for SPI connection to the memory where the program is stored



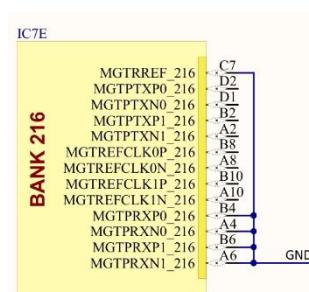
Banks: pins of the FPGA. We can notice that they are grouped in couples.

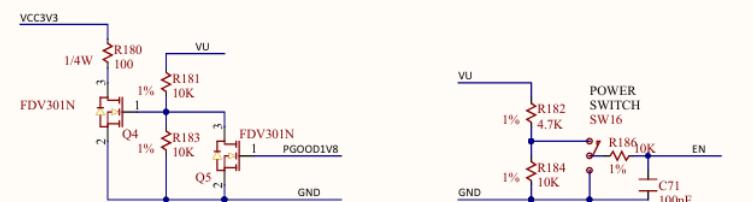
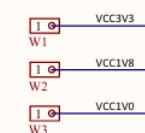
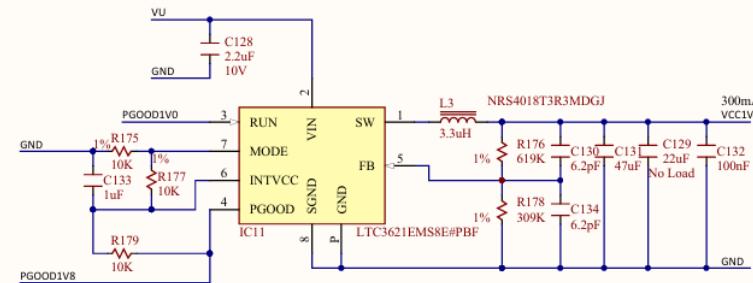
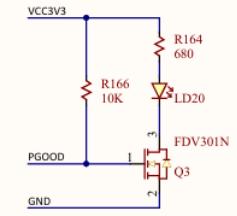
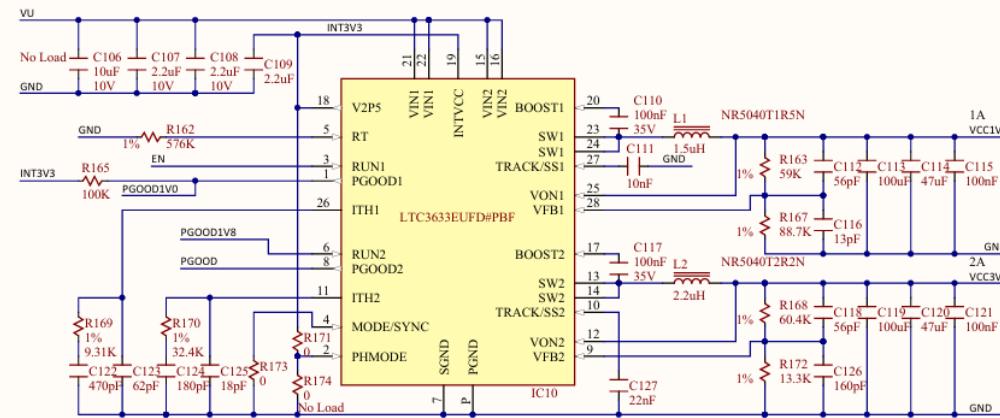
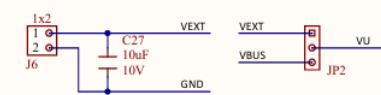
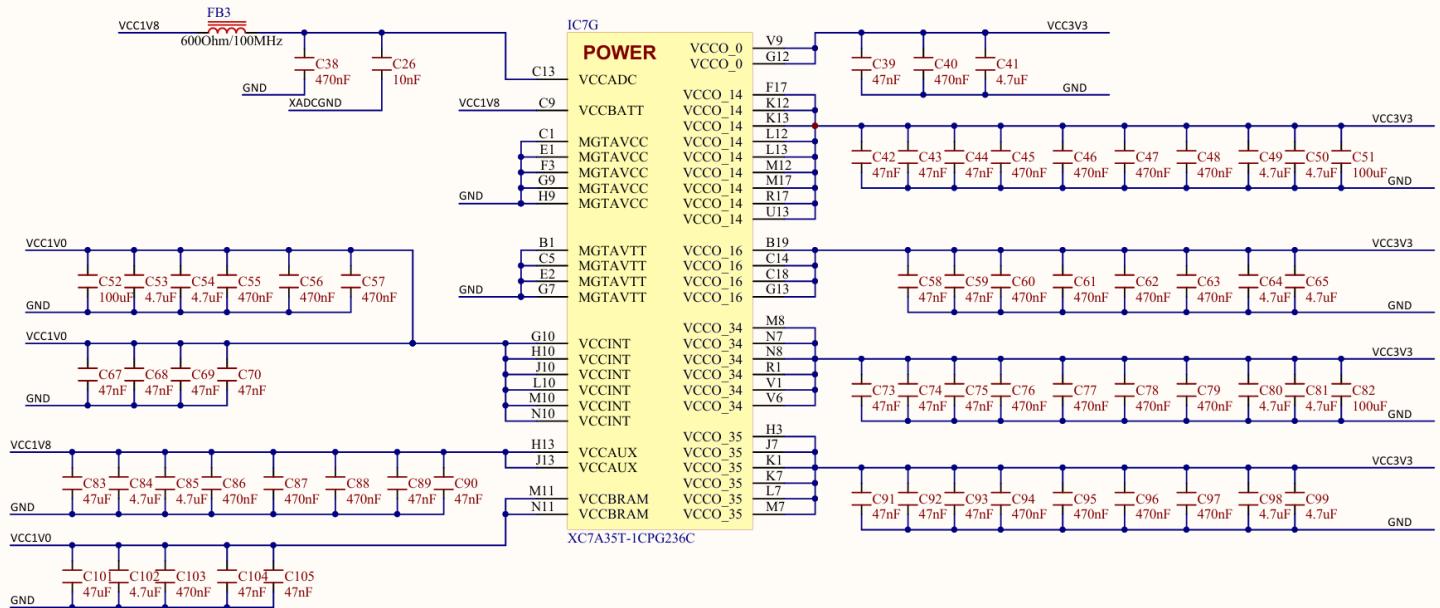


We can see that if we want to use H9 as digital single ended input it is D04 otherwise if we want to use it as a differential couple, we have I/O L4P as positive input and L4N as negative input of the differential pair.

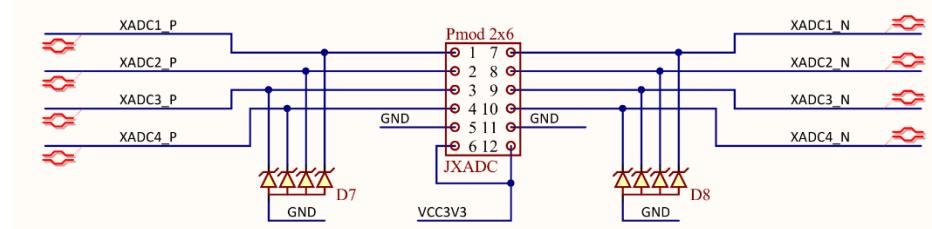
As we can see from the schematic, all the pins are connected to a different peripheral, in the previous case it was connected to the VGA.

We can see a transceiver here used for advanced features and we can notice that it is connected to ground, so it is turned off to avoid statical power consumption.





Power supply section of the FPGA, in particular we see that we have an ADC inside the FPGA and it has its own dedicated power supply and we see that we have a dedicated pmod with ADC rails.



VIVADO Project Wizard creation

Interesting things

- Vivado can understand VHDL, Verilog and System Verilog, the only constraint is *one language for each file*
 - FPGA model: *xc7a35tcpg236-1*
 - where each part stands for
 - *xc*: stands for Xilinx
 - *7*: 28 nm technology
 - *a*: artix
 - *35*: dimension of the board
 - *t*: logic of the transceivers
 - *cpg236*: package for the pinout
 - *-1*: speed grade, quality of Silicon
 - All the computations of the simulator go in the *.sim* folder
 - All the source files are in the *.srcs* folder

Tcl Console × Messages Log Reports Design Runs ? _ □

Q | X | ⌂ | || | D | M | B |

```
start_gui
create_project basic_project C:/DESD/basics/basic_project -part xc7a35tcpg236-1
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository 'C:/Xilinx/Vivado/2020.2/data/ip'.
file mkdir C:/DESD/basics/basic_project/basic_project.srcs/sources_1/new
set_property target_language VHDL [current_project]
close [ open C:/DESD/basics/basic_project/basic_project.srcs/sources_1/new/top.vhd w ]
add_files C:/DESD/basics/basic_project/basic_project.srcs/sources_1/new/top.vhd
update_compile_order -fileset sources_1
```

Each command executed is reported in the console, every operation is reported here. The console keeps track of every operation executed.

Report	Type	Options	Modified	Size
✓ Synthesis				
✓ Synth Design (synth_design)				
📄 Utilization - Synth Design	report_utilization			
📄 synthesis_report				
✓ Implementation				
✓ impl_1				
✓ Design Initialization (init_design)				
📄 Timing Summary - Design Initialization	report_timing_summary	max_paths = 10;		
✓ Opt Design (opt_design)				
📄 DRC - Opt Design	report_drc			
📄 Timing Summary - Opt Design	report_timing_summary	max_paths = 10;		
✓ Power Opt Design (power_opt_design)				
📄 Timing Summary - Power Opt Design	report_timing_summary	max_paths = 10;		
✓ Place Design (place_design)				
📄 IO - Place Design	report_io			
📄 Utilization - Place Design	report_utilization			
📄 Control Sets - Place Design	report_control_sets	verbose = true;		

the report of the analysis and of the timing simulations are reported here.

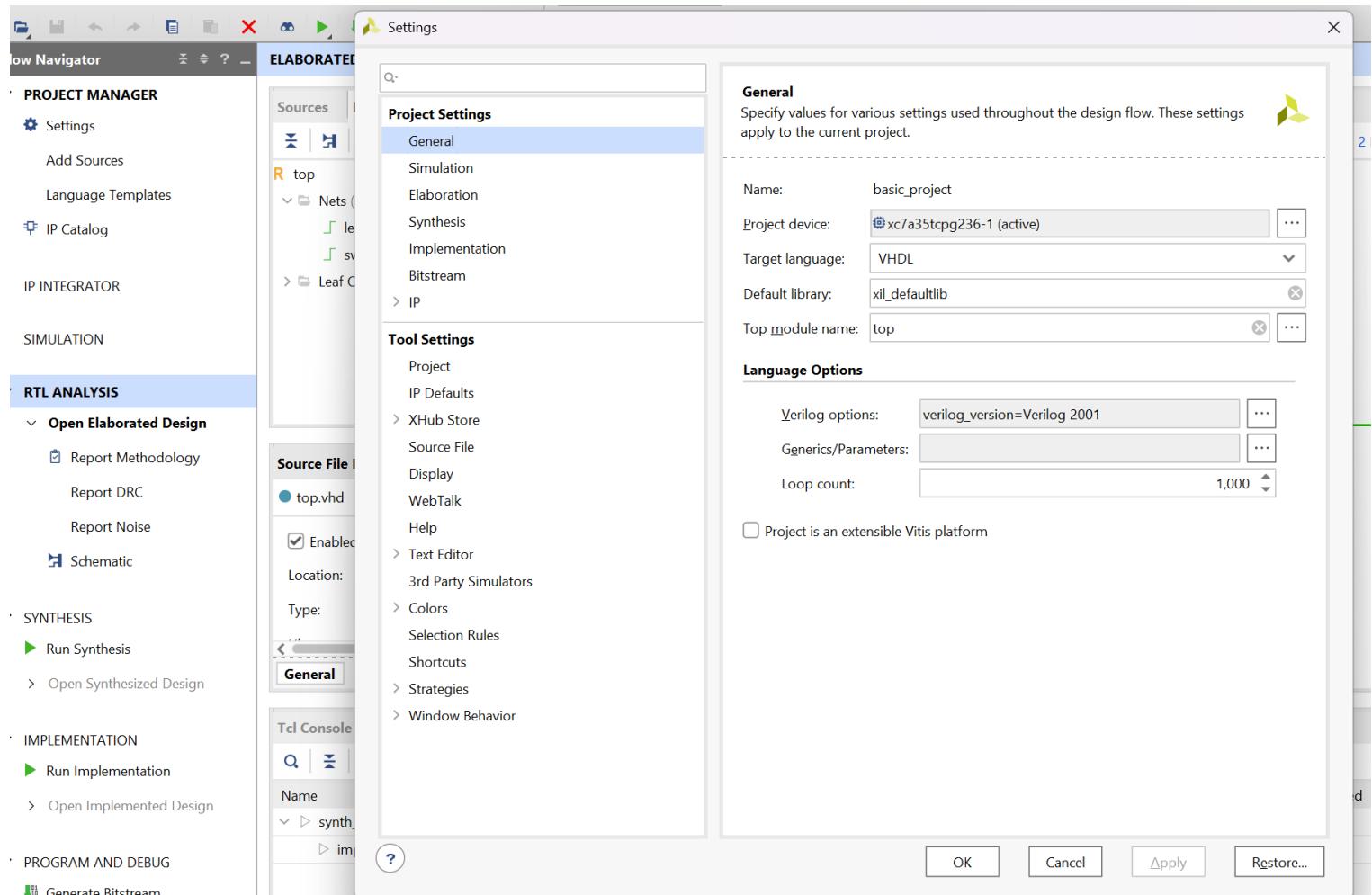
Tcl Console	Messages	Log	Reports	Design Runs	x	?	—	□									
Design Runs															?	—	□
	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
▼ ▷ synth_1	constrs_1	Not started															Vivado Synthesis Defaults (Vivado Synthesis 2020)
▷ impl_1	constrs_1	Not started															Vivado Implementation Defaults (Vivado Implement

Status and progress of the execution, it's the completed percentage etc

Elaborated design and windows

The RTL design is *technology independent*. Then the devices will be implemented in the FPGA we selected.

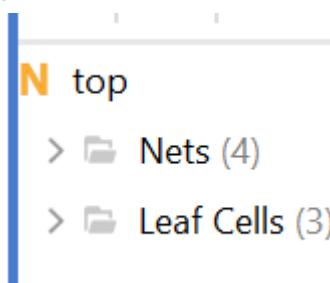
If we selected the wrong FPGA



settings → Project settings → Project device

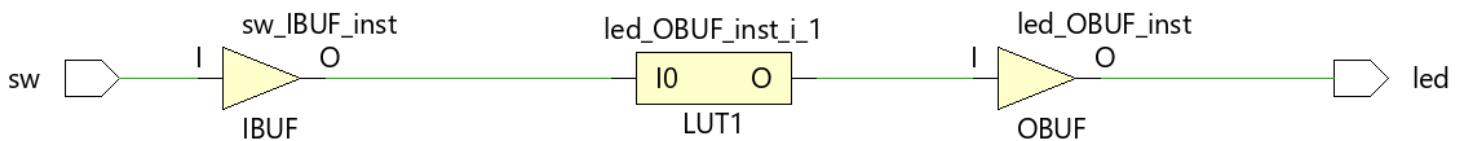
when we synthesize it is a strongly device-dependent process, so it's mandatory to select the correct device.

Synthesis



When we synthesize the netlist becomes *N top* that means that it is a post synthesis design while after the elaborated design we have *R top* because that is a post RTL design.

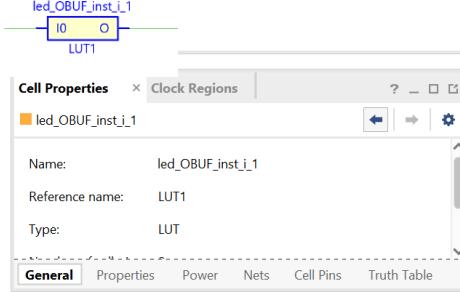
- by pressing *F4* we can see the schematic of the synthesized design



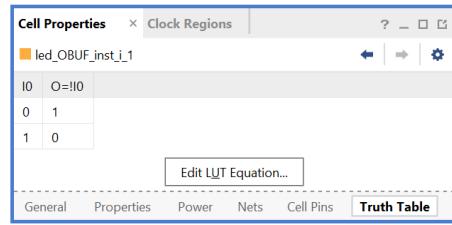
we can see that now we do not have the logic schematic as in the RTL design but we have *switch*, *input buffer*, *lut*, *output buffer* and *led*. If I have an input that is a switch, I need an input buffer that converts the electrical signal coming from the switch into a digital logic one that is in the core of the FPGA. We do not have a NOT inside our FPGA but we have a LUT that is programmed to implement the NOT logic function.

- to see what truth table is implemented in the LUT

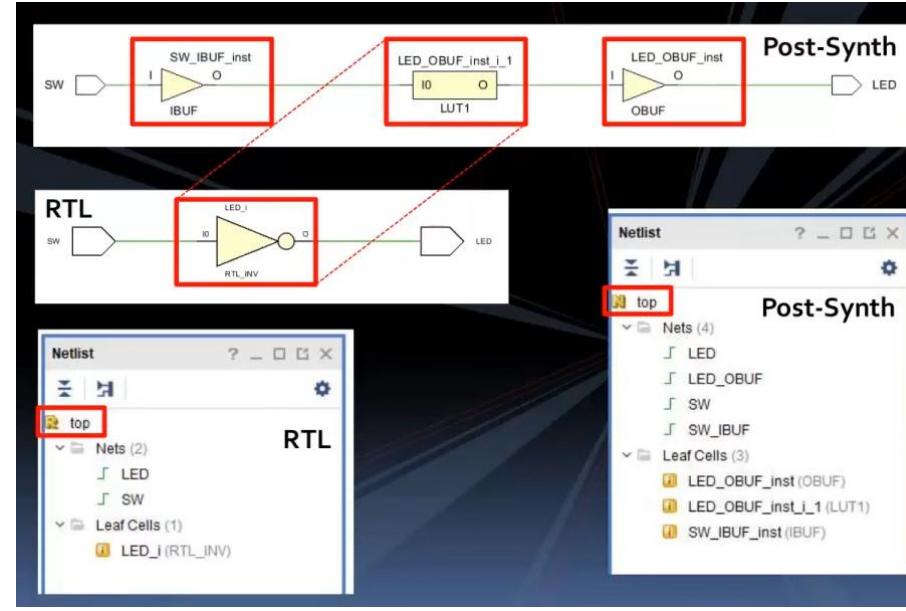
click on the LUT



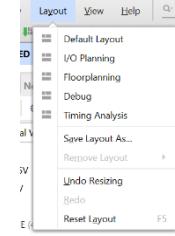
go into the *cell properties*



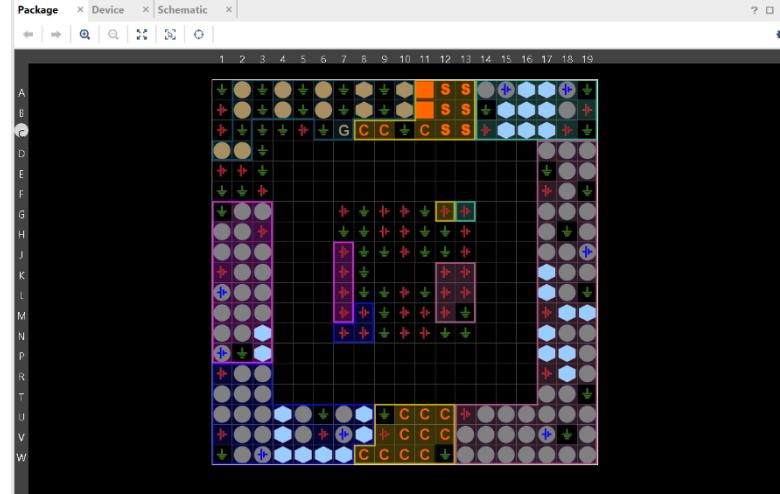
go to *truth table*



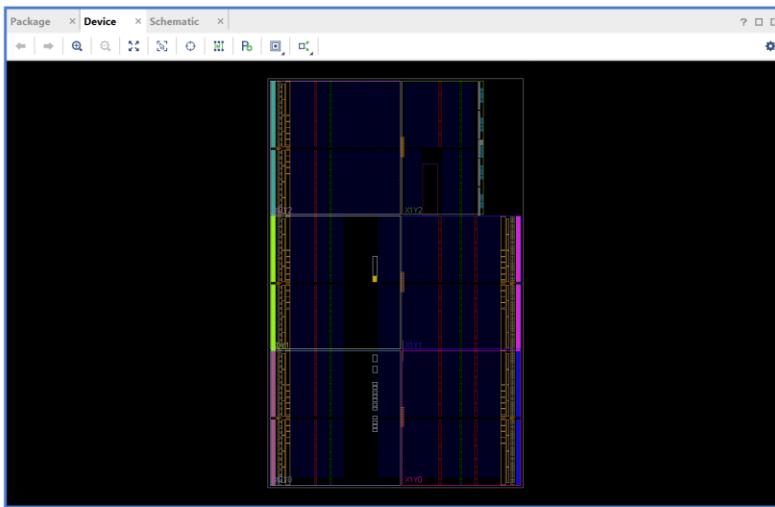
After the synthesis we can move to the device and we can see



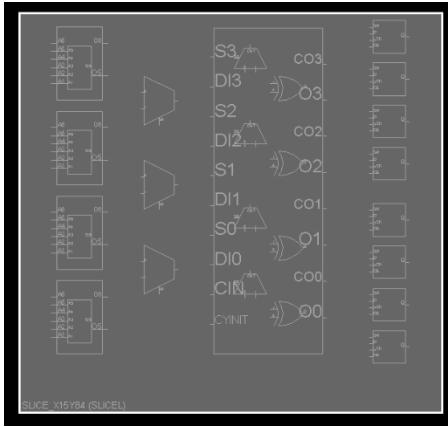
by clicking on *layout* and *I/O planning*



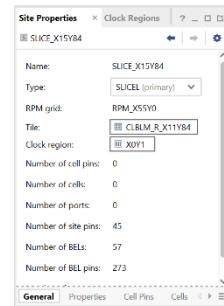
we can see the package of the device



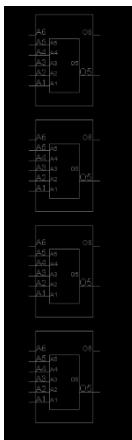
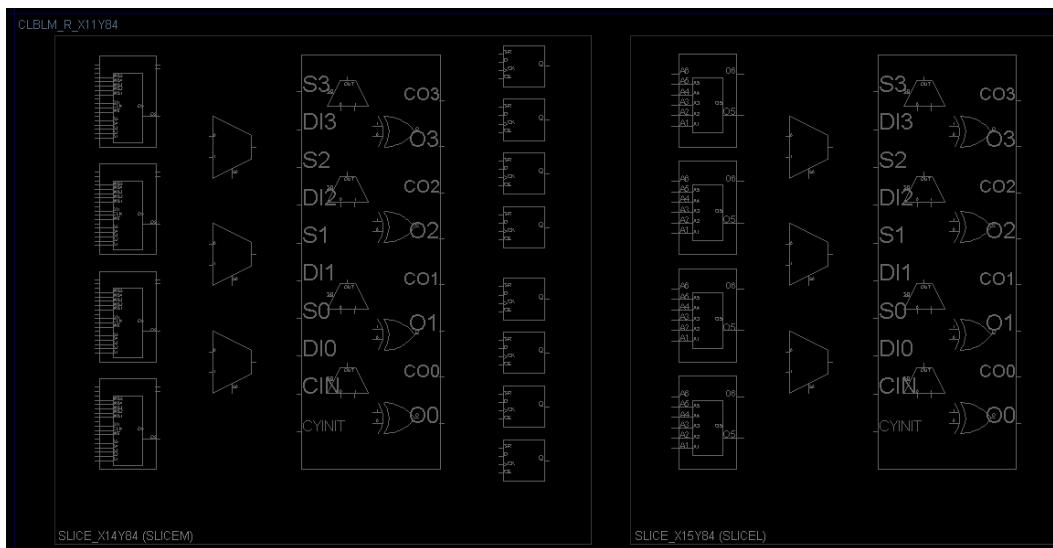
while this is a representation of my FPGA and I can see the CLB where my logic function has been implemented.



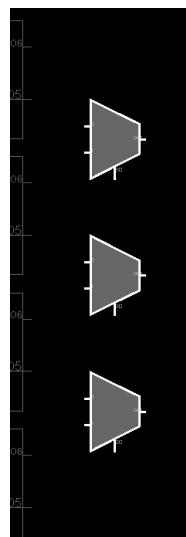
The FPGA is composed of 6 blocks, each one of them is a *clock region*. The bigger the FPGA the more clock regions we have. We can see that the FPGA is organized as a matrix and we can zoom up to see the SLICE.



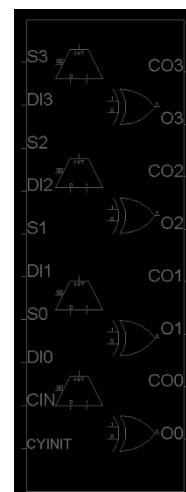
Here we can see the slice properties and by zooming a little more we can see that each SLICE is composed of two CLBs.



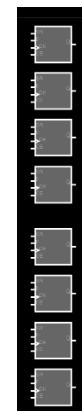
We can see that each CLB has 4 LUTs and each LUT has 6 inputs and 4 outputs or 5 inputs and 1 output based on the choice of the synthesizer



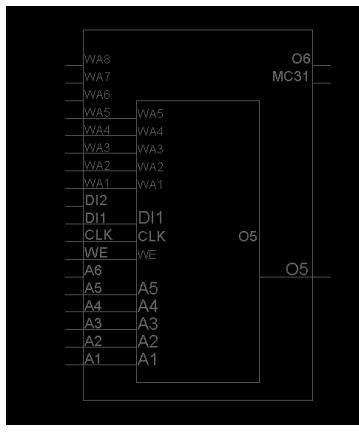
Here we can see the MUX that combines the results of the LUTs elaboration



We can see the bit carry element



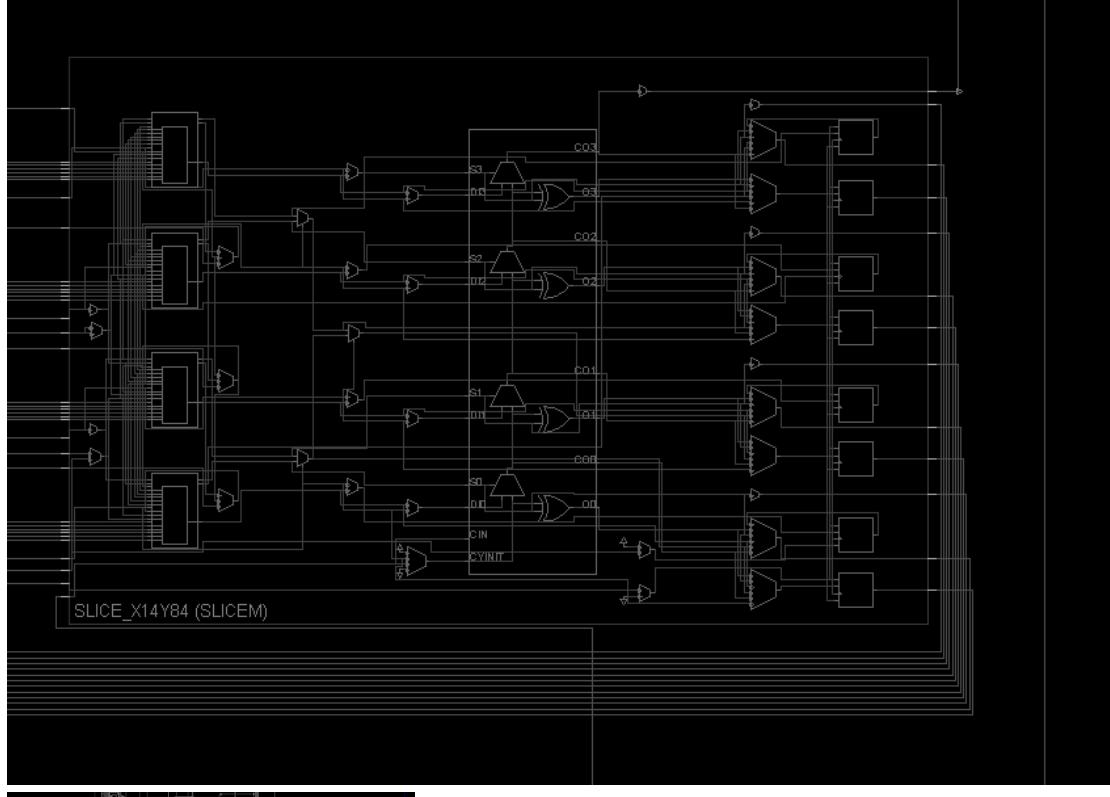
And the 4 by 4 registers



In the SLICEM we also have a memory inside the LUT, we have more pins because each LUT can be used as distributed memory.

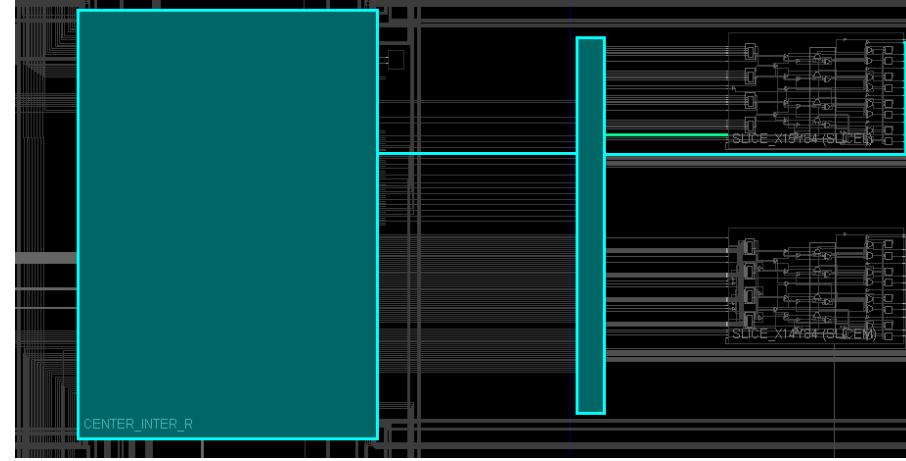
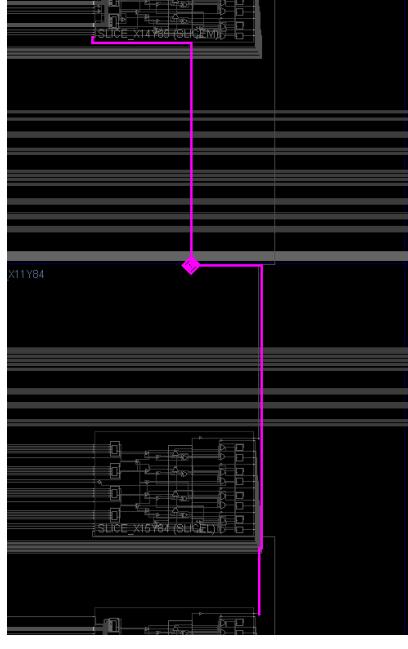


To see the complete routing of the FPGA we have to click on and the routing of the FPGA is shown. We can see the real schematic of the FPGA

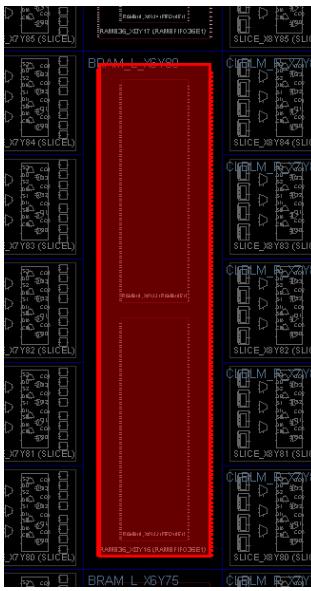


SLICEM with its own routing

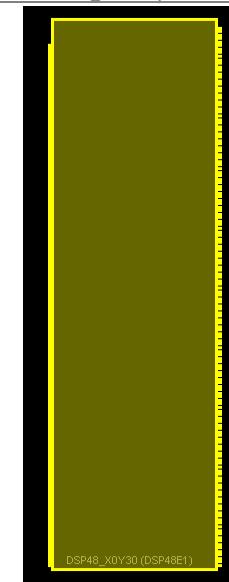
We can see that as we said in theory, the *carry out* signal of the CLB is connected directly between different blocks



While all the other signals go to the switching matrix for signal moving.

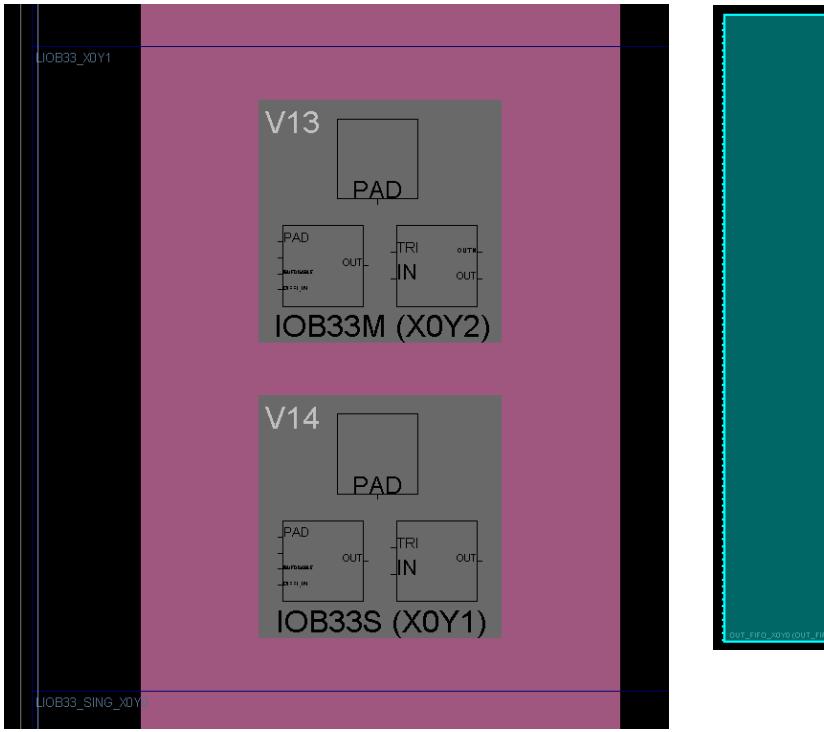


We also have columns of RAM



and columns of DSP

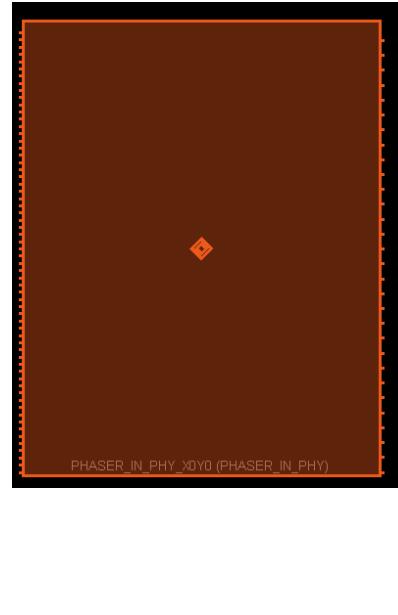
At the edge of the clock region there are the input and output banks.



We have here the input pin of the FPGA, with physical pin (the PAD), the input buffer, the output buffer and the input-output logic. We also have a FIFO for buffering input data.

MMCM: block that can be used as frequency synthesizer.

We also have a PLL.



Spare in the FPGA we also have the global clock buffers.

In package we have the pad of the device. This means that the input U14 is connected to the pin IO_25_14



We can see the banks

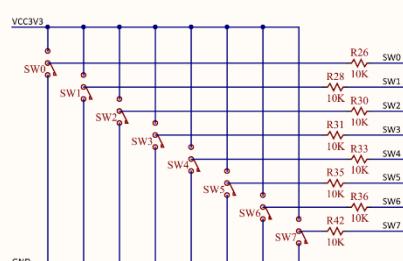


We can see that we have various pins with distributed ground and powering for signal integrity.

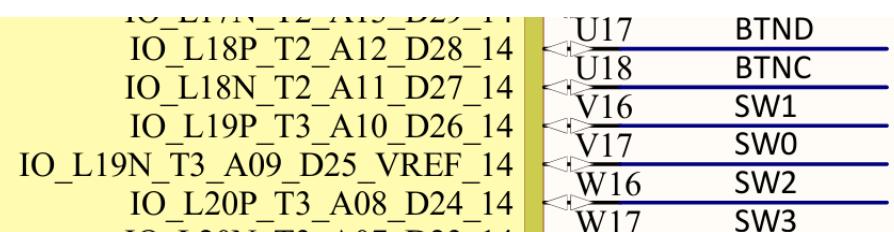
How can I map my pin on my FPGA?

Using package view and I/O port overview.

Let's suppose that *if switch zero is pushed I also see led0 on*

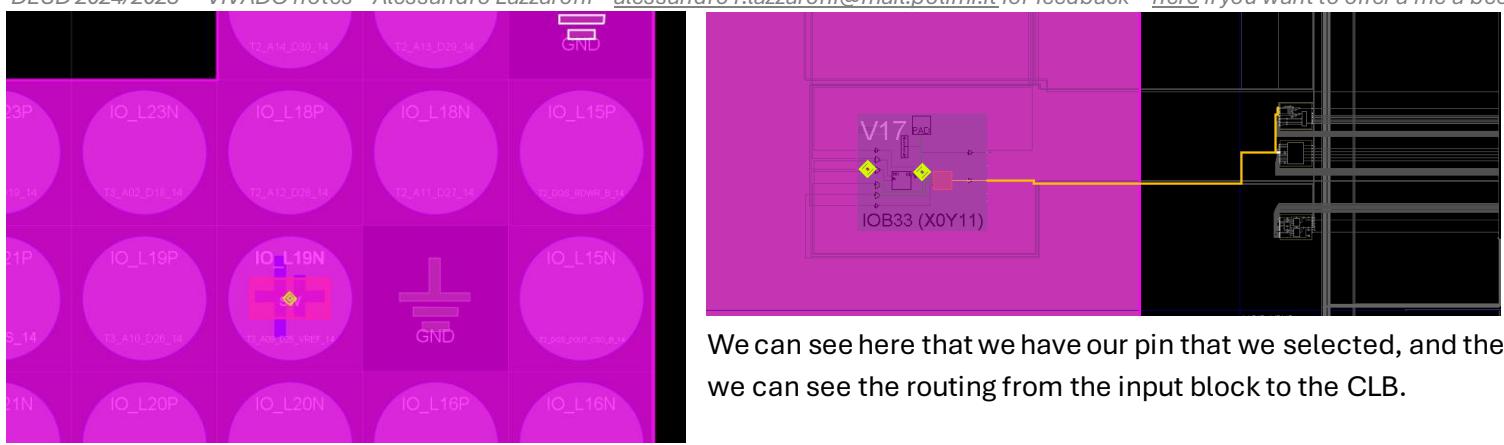


we see from the schematic that SW0 is connected to the V17 of the FPGA



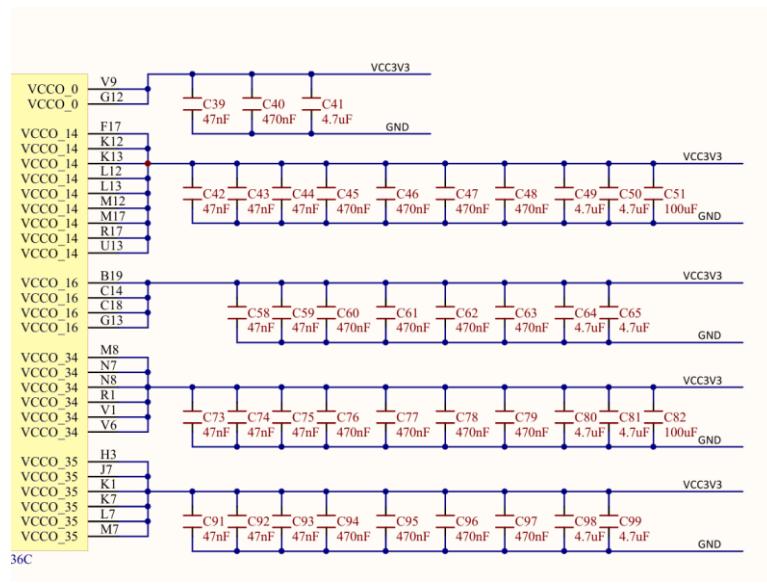
so I want that my input, that is SW0 is the V17 of the FPGA

Tcl Console	Messages	Log	Reports	Design Runs	Package Pins	I/O Ports	x
<input type="text"/> <input type="button"/> <input type="button"/> <input type="button"/> <input type="button"/> <input type="button"/> <input type="button"/>							
Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco
All ports (2)							
Scalar ports (2)							
OUT						def	1.800
IN			V17		14	def	1.800



We can see here that we have our pin that we selected, and then we can see the routing from the input block to the CLB.

After that we have to specify the I/O standard



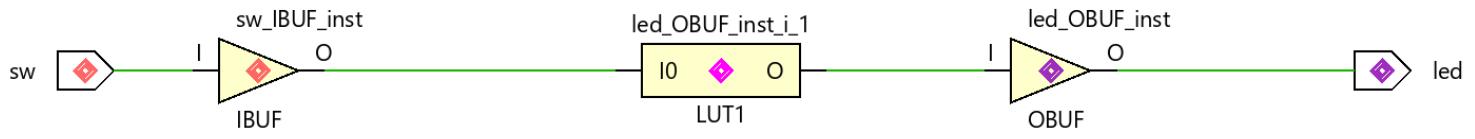
we can see the 3.3V power supply. So we communicate it with the standard *LVCMOS3.3*

I/O Ports														
Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination	IN_TERM	
All ports (2)														
Scalar ports (2)														
OUT						default (LVCMOS18)	1.800		12					
IN			V17	✓	14	LVCMOS33*	3.300							

we can see that two points have been highlighted meaning that the I/Os have been set



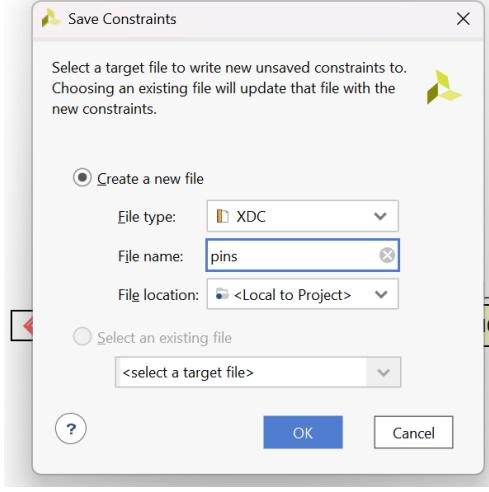
We can also mark from the synthesized view to see on the device the used buffer, LUT etc...



corresponds to



After we set some constraints and we need to save them, maybe to use them again, I need to save them into an XDC file



in this file we have all the informations that we imposed via the graphical user interface.

```

Package x Device x Schematic x pins.xdc x
C:/DESD/basics/basic_project/basic_project.srcs/constrs_1/new/pins.xdc

Q | H | ← | → | X | B | // | || | Q |
1 | set_property PACKAGE_PIN V17 [get_ports sw]
2 :

```

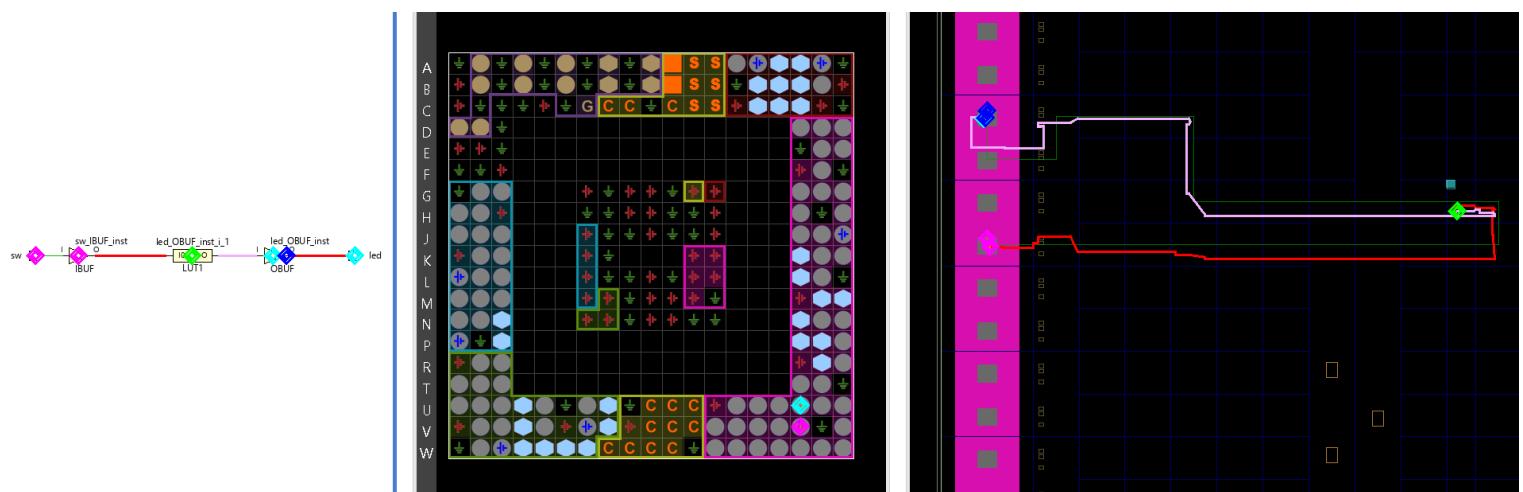
Implementation

If I click on *run implementation* the RTL and the synthesized design are implemented in our device

Report	Type	Options	Modified	Size
Synthesis				
Synth Design (synth_design)	Utilization - Synth Design	report_utilization	3/19/25, 3:52 PM	6.6 KB
	synthesis_report		3/19/25, 3:52 PM	14.3 KB
Implementation				
impl_1				
Design Initialization (init_design)	Timing Summary - Design Initialization	report_timing_summary max_paths = 10		

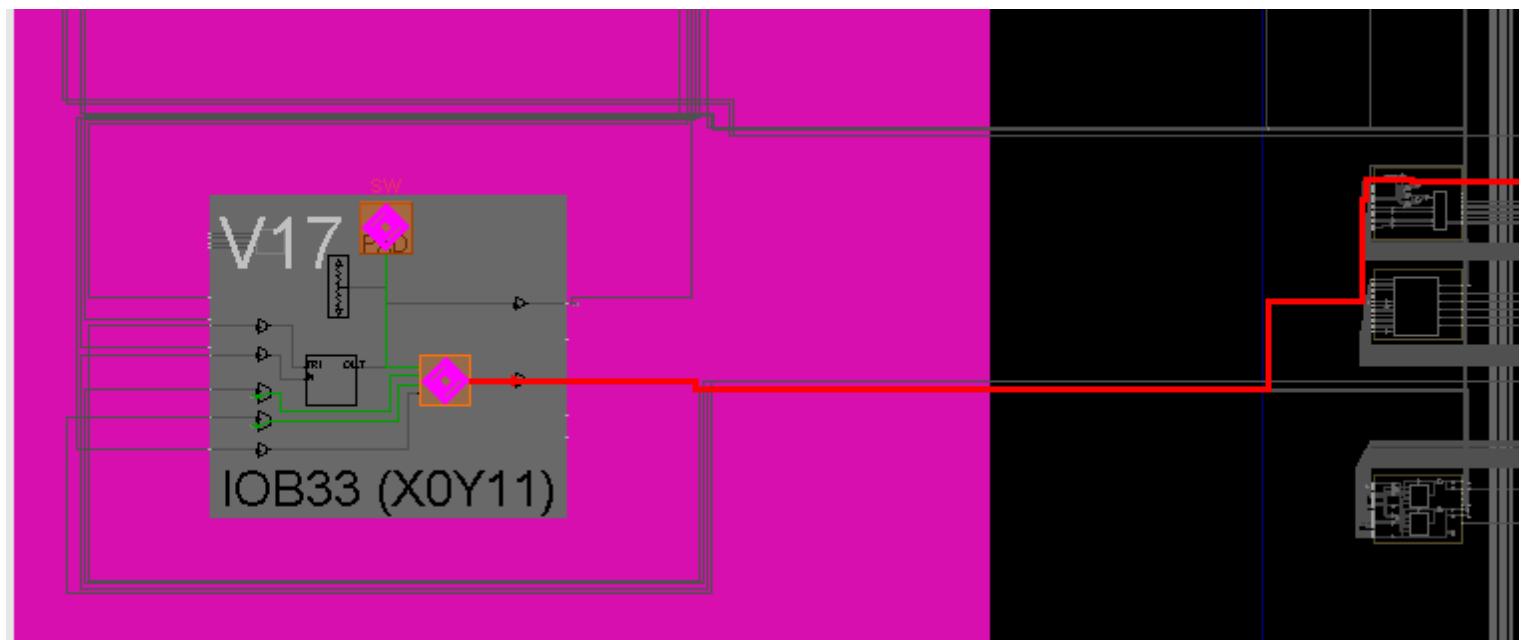
ex

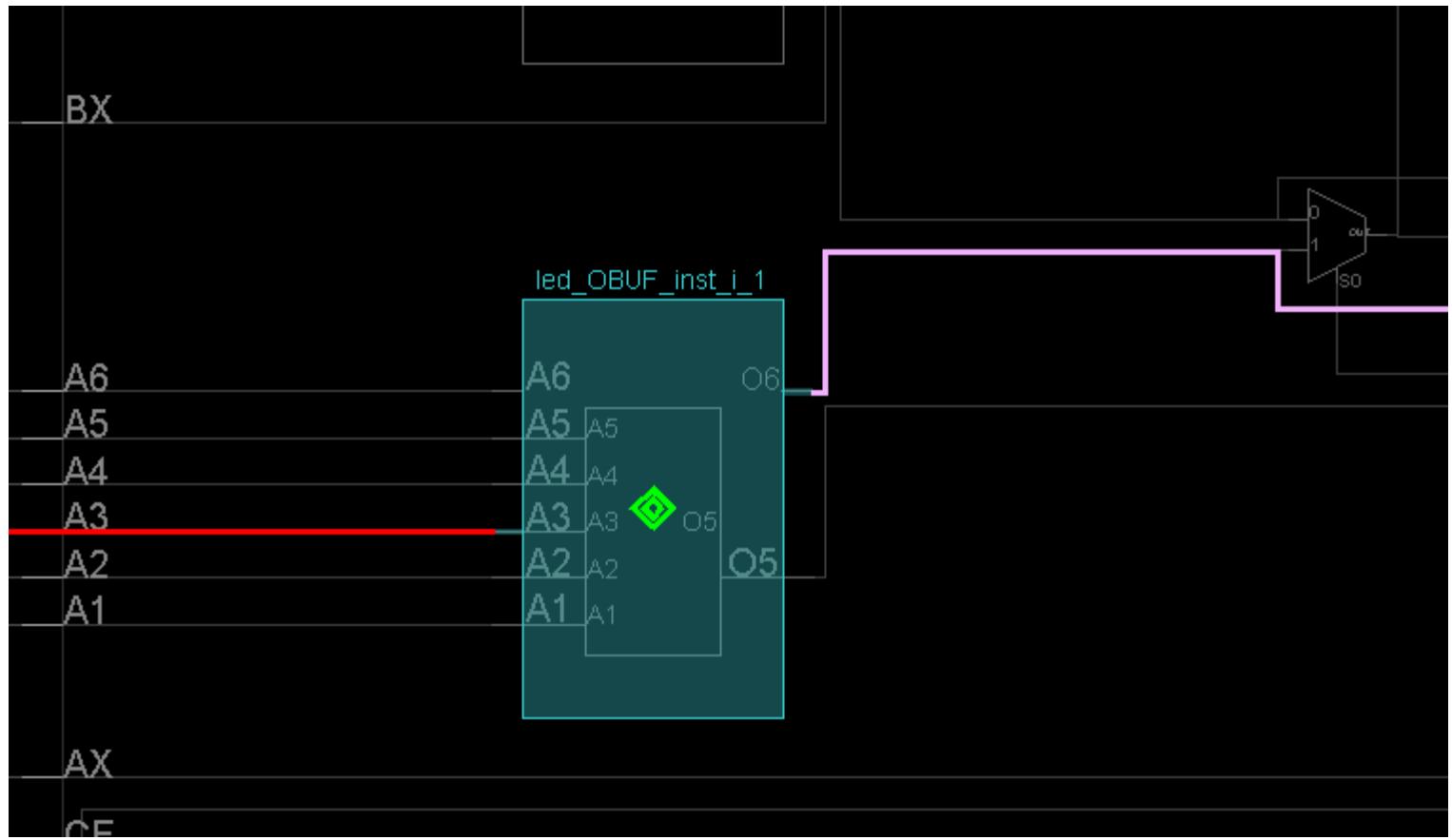
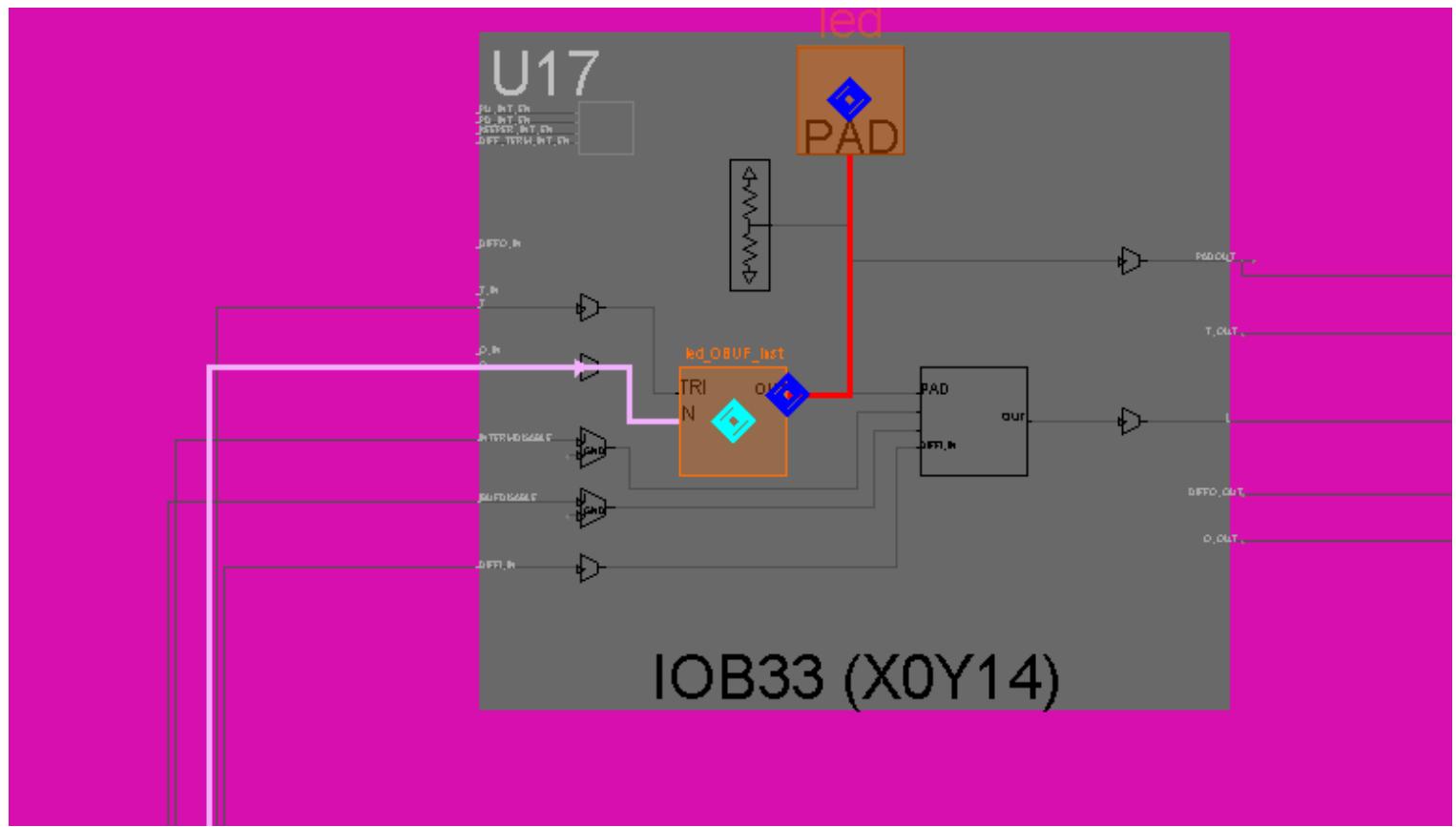
Site Type	Used	Fixed	Available	Util%
Slice LUTs*	1	0	20800	<0.01
LUT as Logic	1	0	20800	<0.01
LUT as Memory	0	0	9600	0.00
Slice Registers	0	0	41600	0.00
Register as Flip Flop	0	0	41600	0.00
Register as Latch	0	0	41600	0.00
F7 Muxes	0	0	16300	0.00
F8 Muxes	0	0	8150	0.00



In detail:

IBUF and its connections



LUT and its connections**OBUF and its connections****GENERATE BITSTREAM**

It's procedure to upload the project to the FPGA

We click *generate bitstream*

The bitstream is the .bit file in the implementation folder

XDC file editing

```

set_property PACKAGE_PIN V17 [get_ports sw]
set_property PACKAGE_PIN U17 [get_ports led]
set_property IOSTANDARD LVCMOS33 [get_ports led]
set_property IOSTANDARD LVCMOS33 [get_ports sw]

```

We can quickly switch the input pins by editing directly the XDC file

Modify the specific LUT

We can quickly change the LUT that is used, it can just simply drag-and-drop in another LUT but then we have to re-run the implementation because otherwise the connections to the LUT are broken.

If we change the I/O ports we also have to re-run the synthesis and then the implementation.

Correct procedure to share a project

File > Project > Archive

then we specify the archive name and the archive location. Oss: we can't just compress the folder from our pc and send it because it could happen that it won't work, while by doing this procedure Vivado creates some files etc. so that it works well.

02_Vivado_Simulation_Hardware_Debug

Simulation is wrote in VHDL so it's a file that has to be created in the `.srcs` folder of the project, while the results of the simulations are saved into the `.sim` folder.

Connecting the test bench to our entity

Like when using a component we have to open our test bench file, and below the architecture section initialize our device under test as a component, in order to be able to test it.

```

33 :
34 entity tb_count_1 is
35   -- Port ( );
36 end tb_count_1;
37 :
38 architecture Behavioral of tb_count_1 is
39 :
40 :
41 component counter is
42   Generic(
43     BIT_COUNT : INTEGER RANGE 0 TO 32 := 8;
44     INIT_COUNT : INTEGER := 1
45   );
46   Port (
47     reset : IN STD_LOGIC;
48     clk : IN STD_LOGIC;
49     count : OUT STD_LOGIC_VECTOR(BIT_COUNT-1 downto 0)
50   );
51 end component;
52 :
53 begin
54   -- generate reset
55   -- generate the clock
56   -- connect the clk and reset to the counter
57   -- connect the count to the output
58 end Behavioral;

```

oss: constant TB_BIT_COUNT : INTEGER RANGE 0 TO 32 := 8; is the dimension of the device under test that I want.

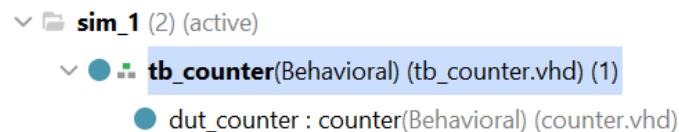
In fact, while writing the device under test, if we have any generic, we have to initialize them and so we have to create a constant that will be the initialization value for the generic.

```
40
41     component counter is
42         Generic(
43             BIT_COUNT    :  INTEGER RANGE 0 TO 32    := 8;
44             INIT_COUNT   :  INTEGER                  := 1
45         );
46         Port (
47
48             reset    :  IN STD_LOGIC;
49             clk      :  IN STD_LOGIC;
50
51             count    :  OUT STD_LOGIC_VECTOR(BIT_COUNT-1 downto 0)
52         );
53     end component;
54
55
56
57
58     constant TB_BIT_COUNT : INTEGER RANGE 0 TO 32 := 8;
59     constant TB_INIT_COUNT : INTEGER := 1;
60
61
62
63 begin
64
65
66     dut : counter
67         Generic Map(
68             BIT_COUNT    => TB_BIT_COUNT,
69             INIT_COUNT   => TB_INIT_COUNT
70         )
71         Port Map(
72
73             reset    =>
74             clk      =>
75
76             count    =>
77
78     );
79 
```

After that we have to create the signal to drive the simulation and the signals that we have to observe in the simulation.

```
56
57
58     constant TB_BIT_COUNT : INTEGER RANGE 0 TO 32 := 8;
59     constant TB_INIT_COUNT : INTEGER := 1;
60
61     signal tb_reset : std_logic;
62     signal tb_clk : std_logic;
63
64     signal tb_count : std_logic_vector(TB_BIT_COUNT-1 downto 0);
65
66
67 begin
68
69
70
71     dut : counter
72         Generic Map(
73             BIT_COUNT    => TB_BIT_COUNT,
74             INIT_COUNT   => TB_INIT_COUNT
75         )
76         Port Map(
77
78             reset    => tb_reset,
79             clk      => tb_clk,
80
81             count    => tb_count
82
83 );
```

here we can see the component initialized in the device under test



we can also initialize the signals with a standard value

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

by making the device as parametric as possible we have the possibility of quickly changing the dimension in the range that we defined, maybe to create different test benches in the begin in order to have different versions under test.

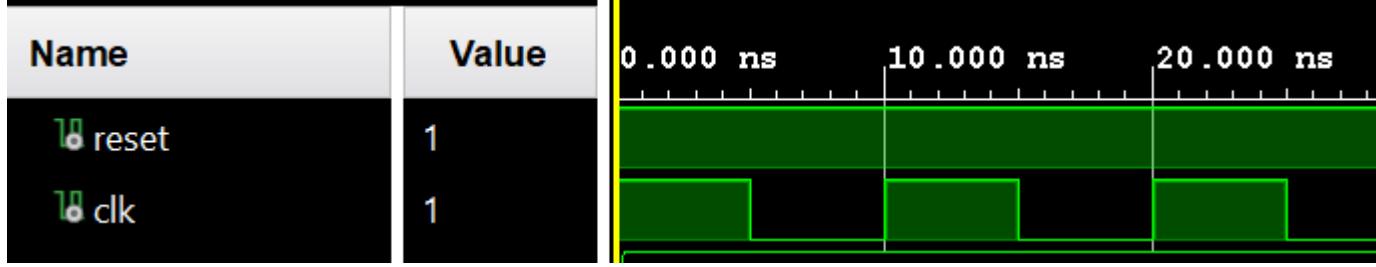
clock generator implementation

```
-- TB Clk Generation
clk <= not clk after CLK_PERIOD/2;
```

where we previously defined

```
-- Constant For Test Bench (TB)
constant CLK_PERIOD : time := 10 ns;
```

so that we obtain a square wave



if I want a faster clock we can just change the constant initialization value.

I might also initialize the initial value of the clock just by putting

```
signal clk : STD_LOGIC := '1';
```

reset process for simulation

```
-- TB Reset Generation
reset_wave : process
begin

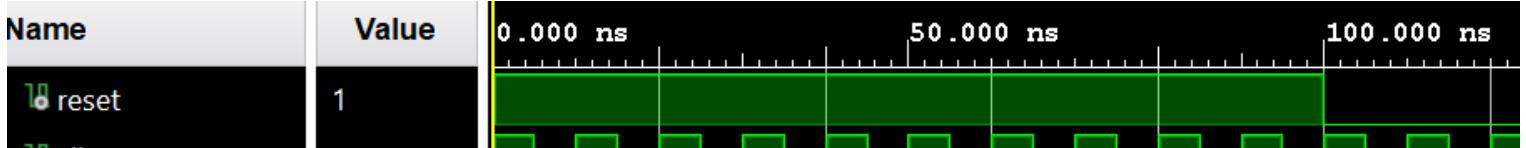
    reset <= '1';
    wait for RESET_WND;

    reset <= '0';
    wait;

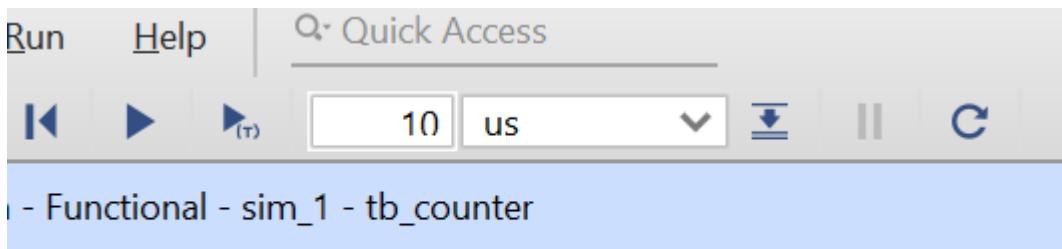
end process;
```

where we previously defined

```
constant RESET_WND : time := 100 ns;
```



how to change the scale of for how much time the simulation has to run

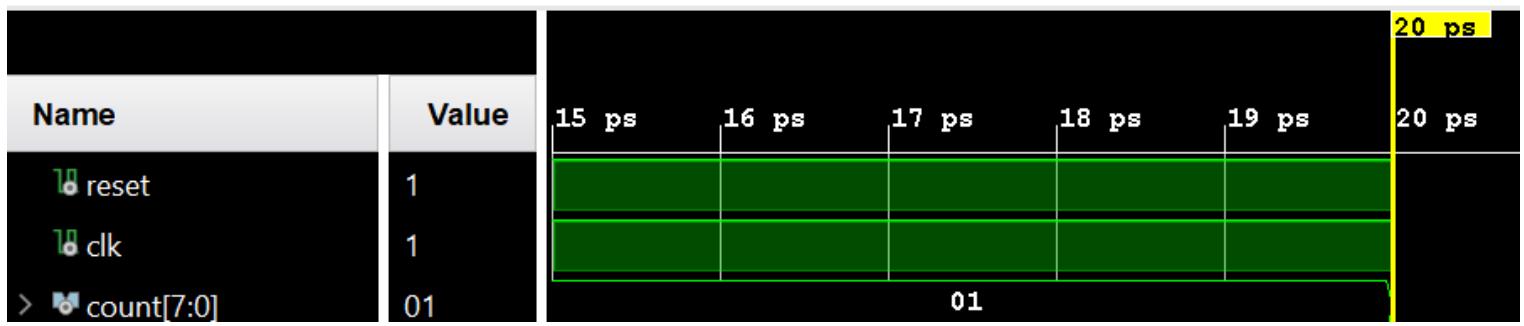


while being in the simulation section, change this parameter and the simulation will run for that much time.

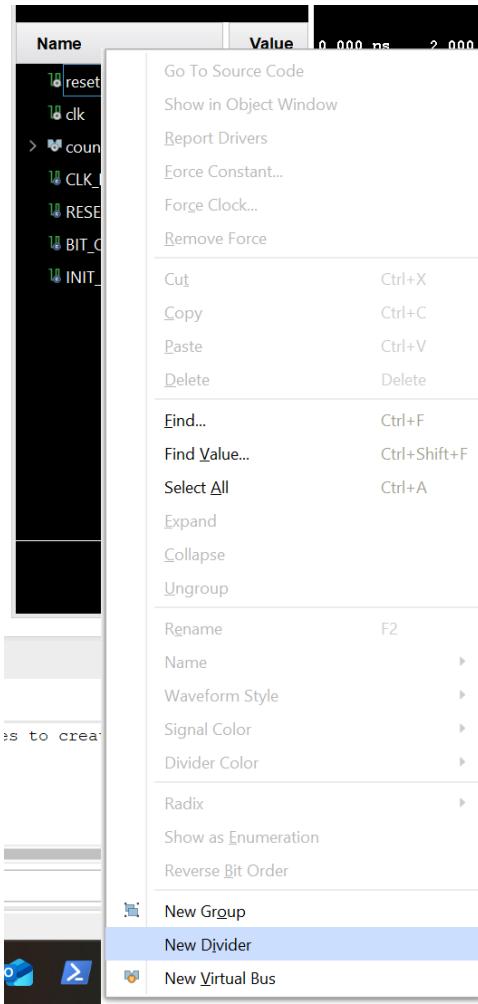
If we change the time scale, what we have to do to see the different results is

1. Relaunch Simulation
2. Restart (Ctrl+Shift+F5)
3. Run for 20ps (Shift+F2)

and we obtain the waveform of the new simulation

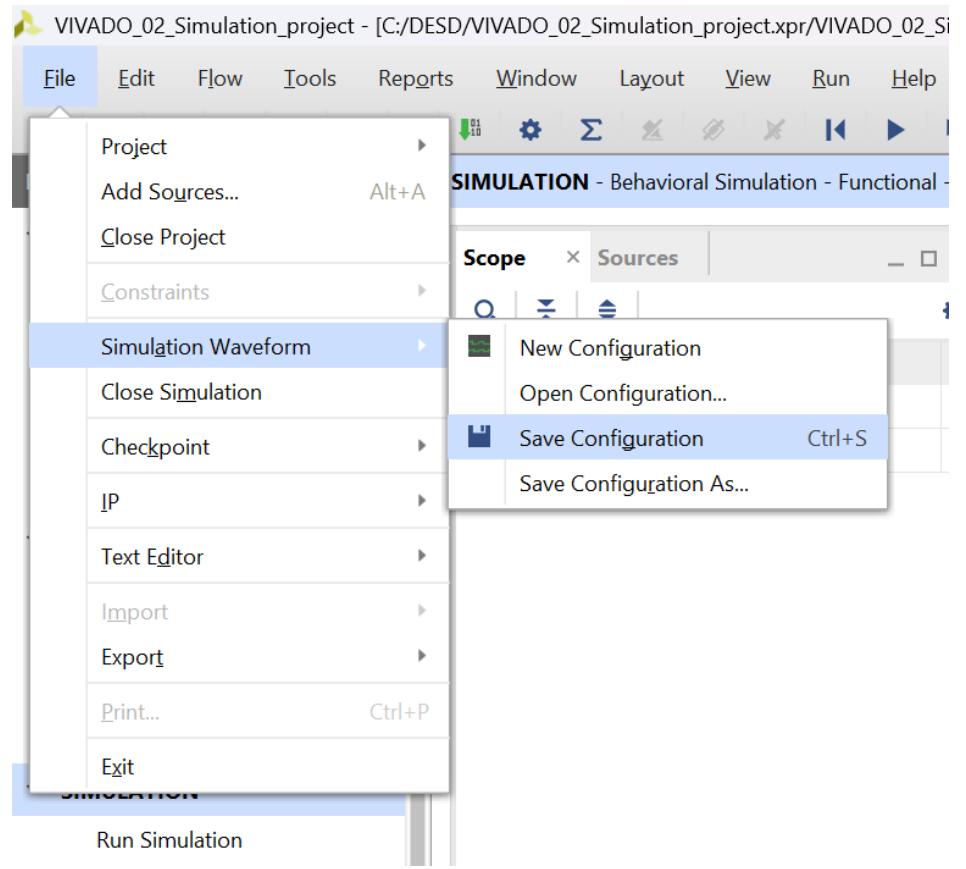


How to personalize and save the waveform



To create an understandable waveform create a divider and separate the various sections of signals.

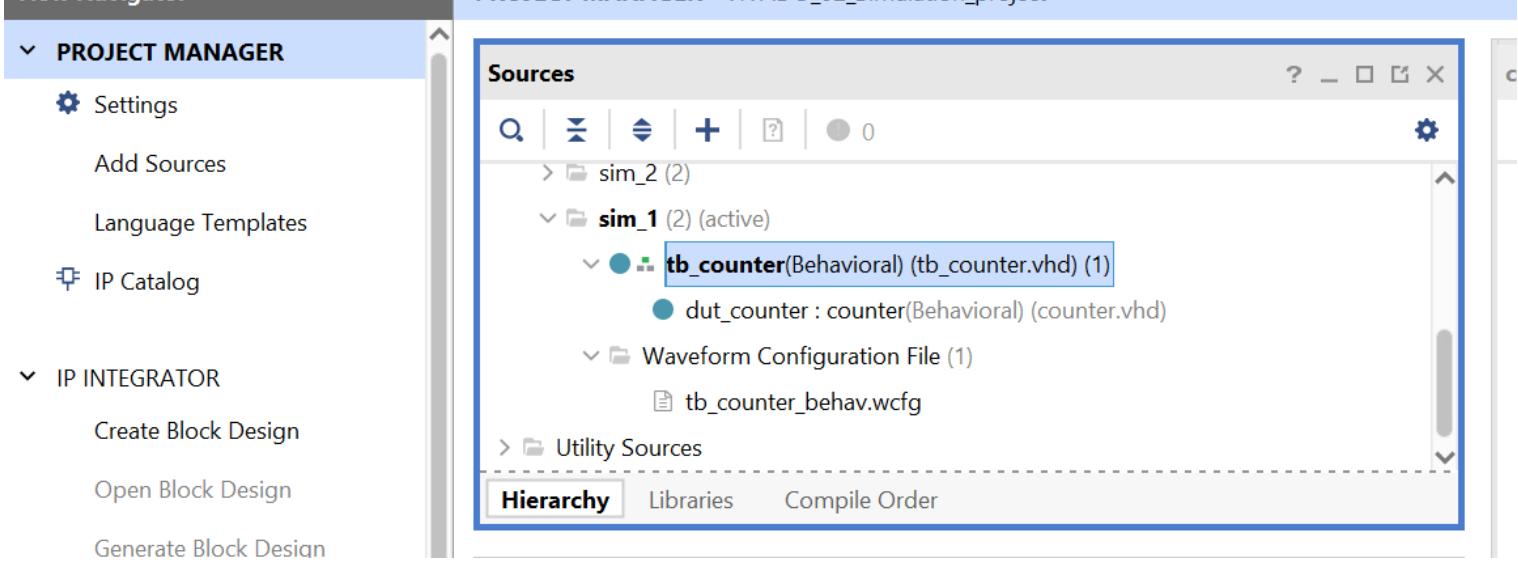
Then when I want to save



and a file in .sim folder is created

```
save_wave_config {C:/DESP/VIVADO_02_Simulation_project.xpr/VIVADO_02_Simulation_project/VIVADO_02_Simulation_project.srccs/sim_1/new/tb_counter_behav.wcfg}
```

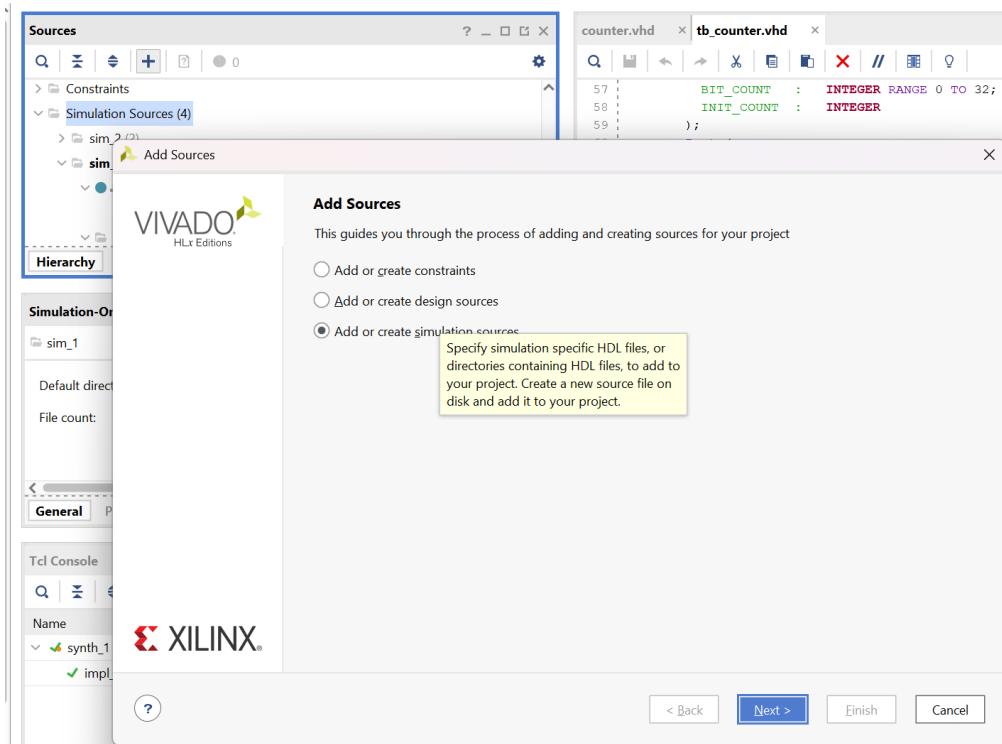
and from the project manager we can also see it in the *waveform configuration bits*



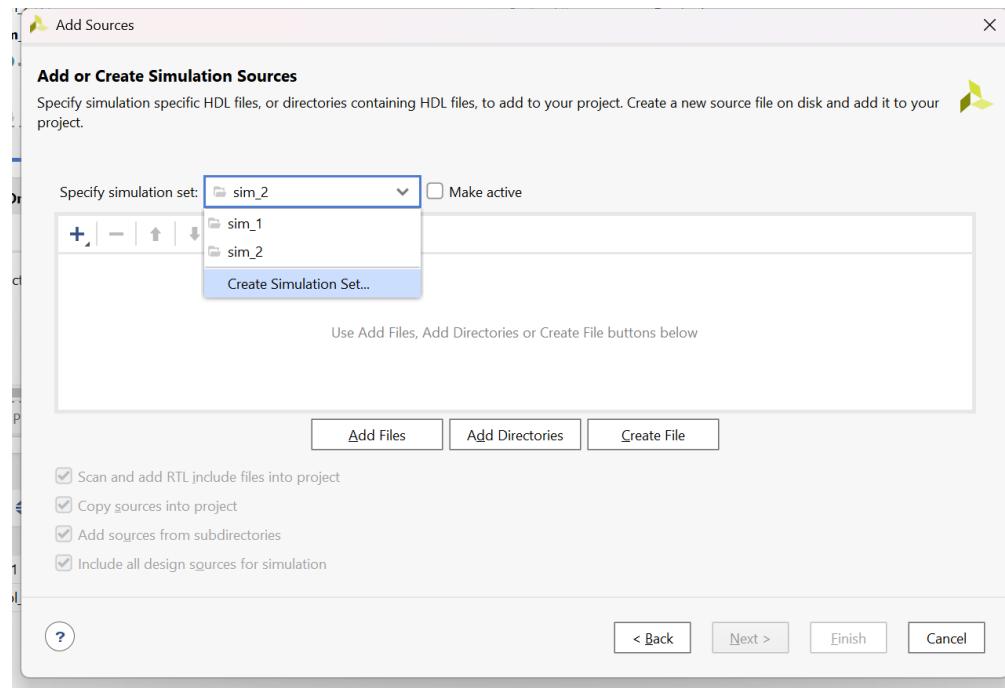
How to create a new simulation set

1. Create a new subfolder in the .srccs folder
2. Create there a new test bench with the new simulation set, new VHDLfile with all the new simulation parameters

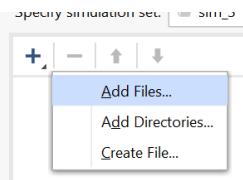
3. Import the test bench without modifying the previous so we go on the + button and then we add a simulation source



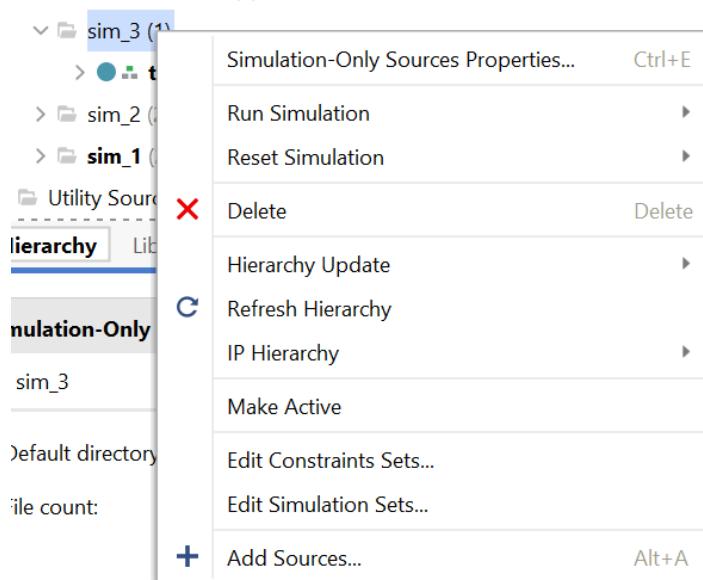
4. Create the new simulation set



5. Add files
6. Add the new simulation file
7. Do not copy the simulation sources in the project



8. Now I have to deactivate simulation one and activate simulation two by clicking *Make active*



Differences between “Behavioral”, “Post synthesis” and “Post implementation” simulation

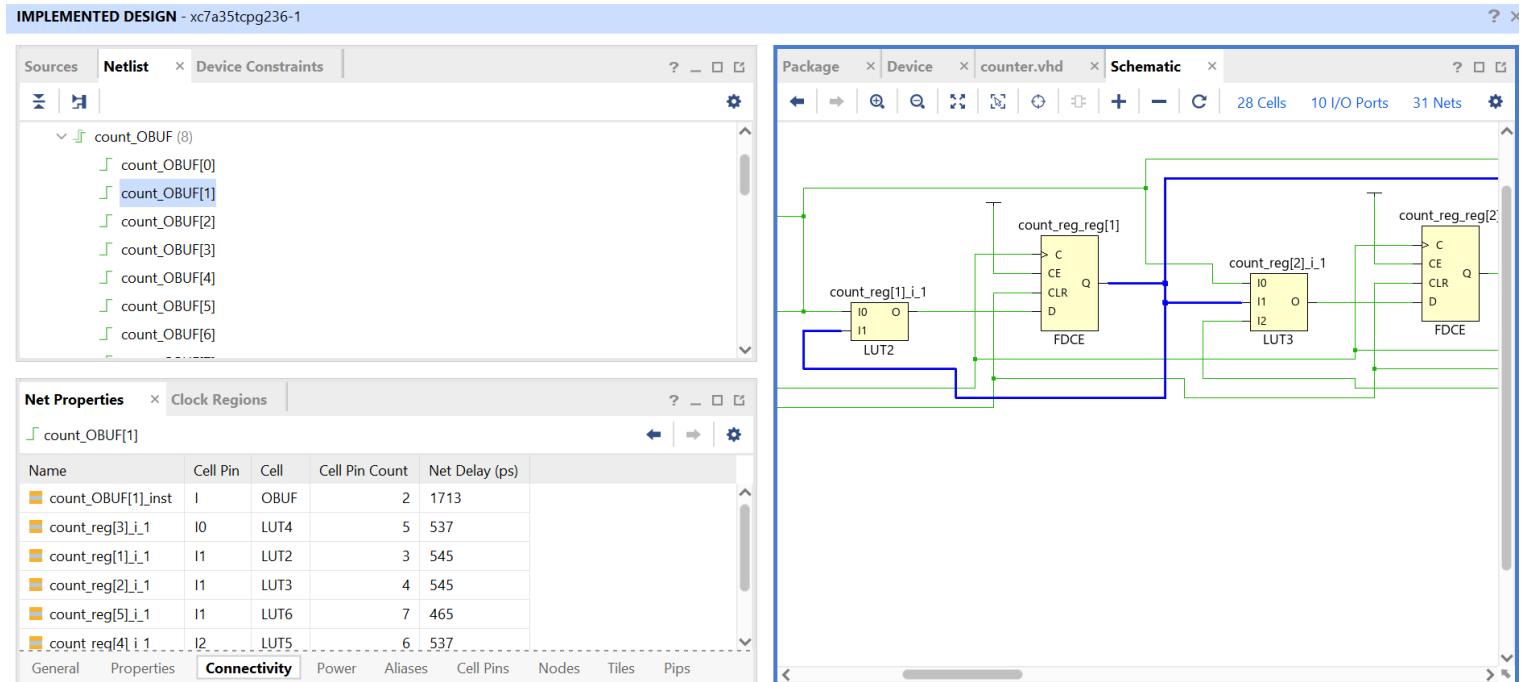
Since in the usual FPGA design we have three main steps

1. *RTL analysis*: general purpose I/O schematic, valid for any FPGA
2. *Synthesis*: the program is wrapped inside my device
3. *Post-implementation*: the schematic is placed and routed

then in the

1. *RTL behavioral simulation*: all the components are ideal, so the propagation and contamination delay are zero
2. *Post-synthesis simulation*: the components are not ideal anymore, propagation and contamination delays are considered, but here we consider the average value of these quantities
3. *Post-implementation simulation*: since the devices have been placed, this means that we have the (*computed*) “real” delays

example for post synthesis



we can see here that in this implemented design the cable introduces that specific delay

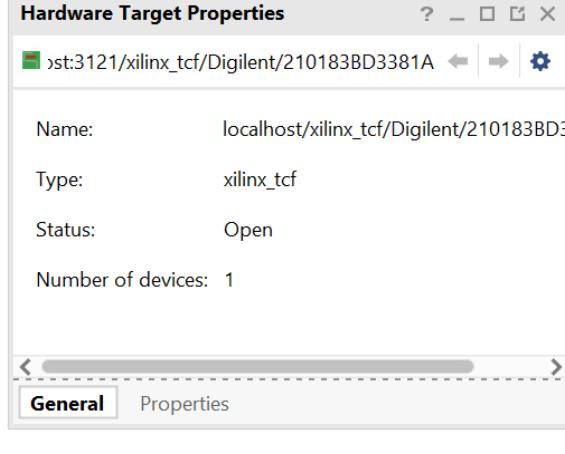
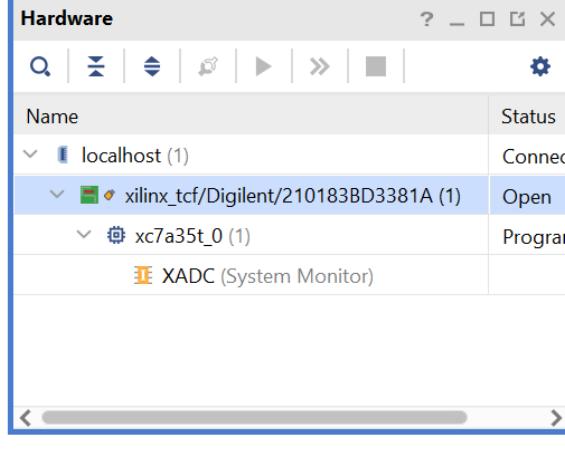
oss: obviously after I run the synthesis and the post-implementation the generics have to be fixed, otherwise I have to synthesize the new device with the new buffers, numbers of routes etc.

Uploading code on the FPGA

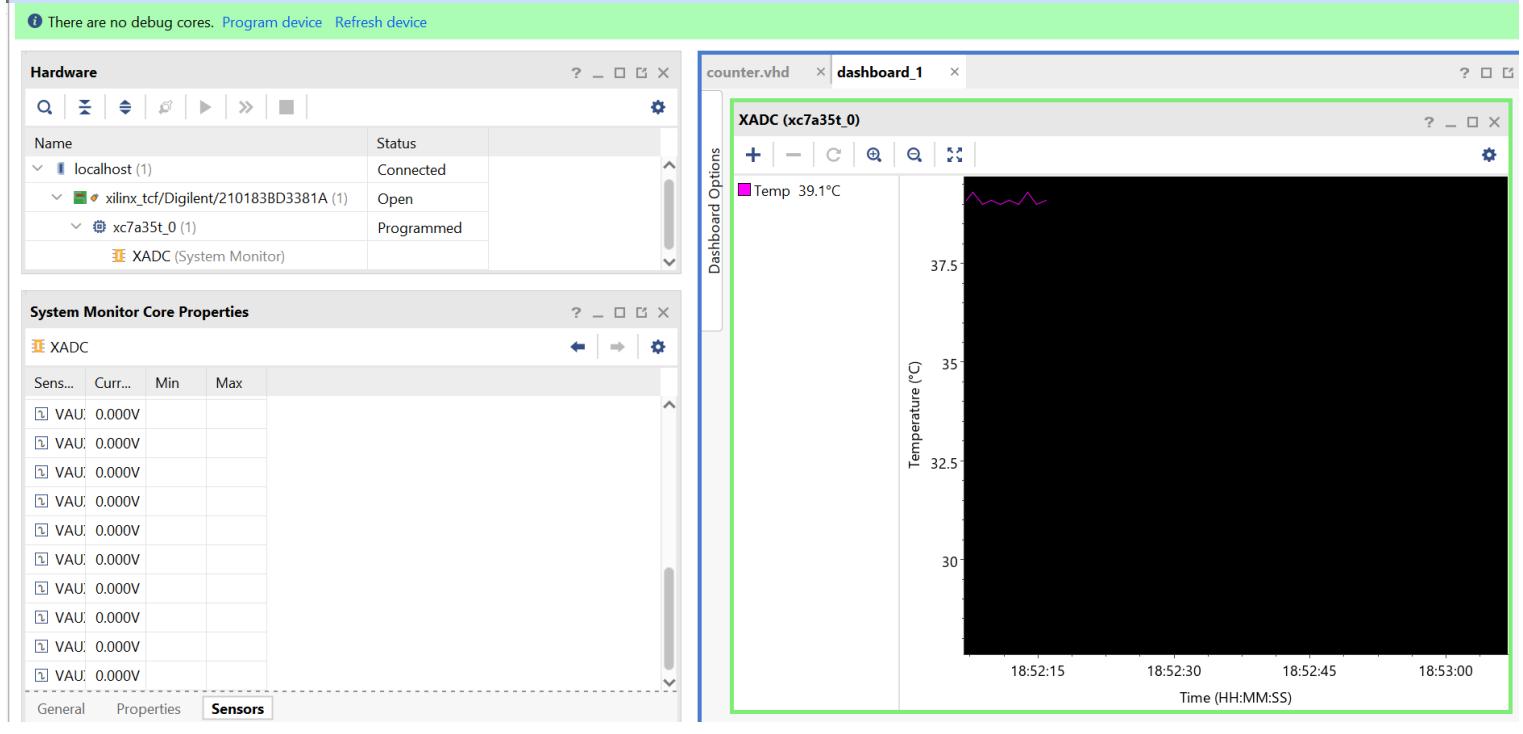
After we wrote the code and we are sure that it works correctly we can use the button “Generate bitstream”, the synthesis and the implementation are run.

oss: when we change the input and output ports the settings are saved into an *xdc* file, usually is embedded in the project.

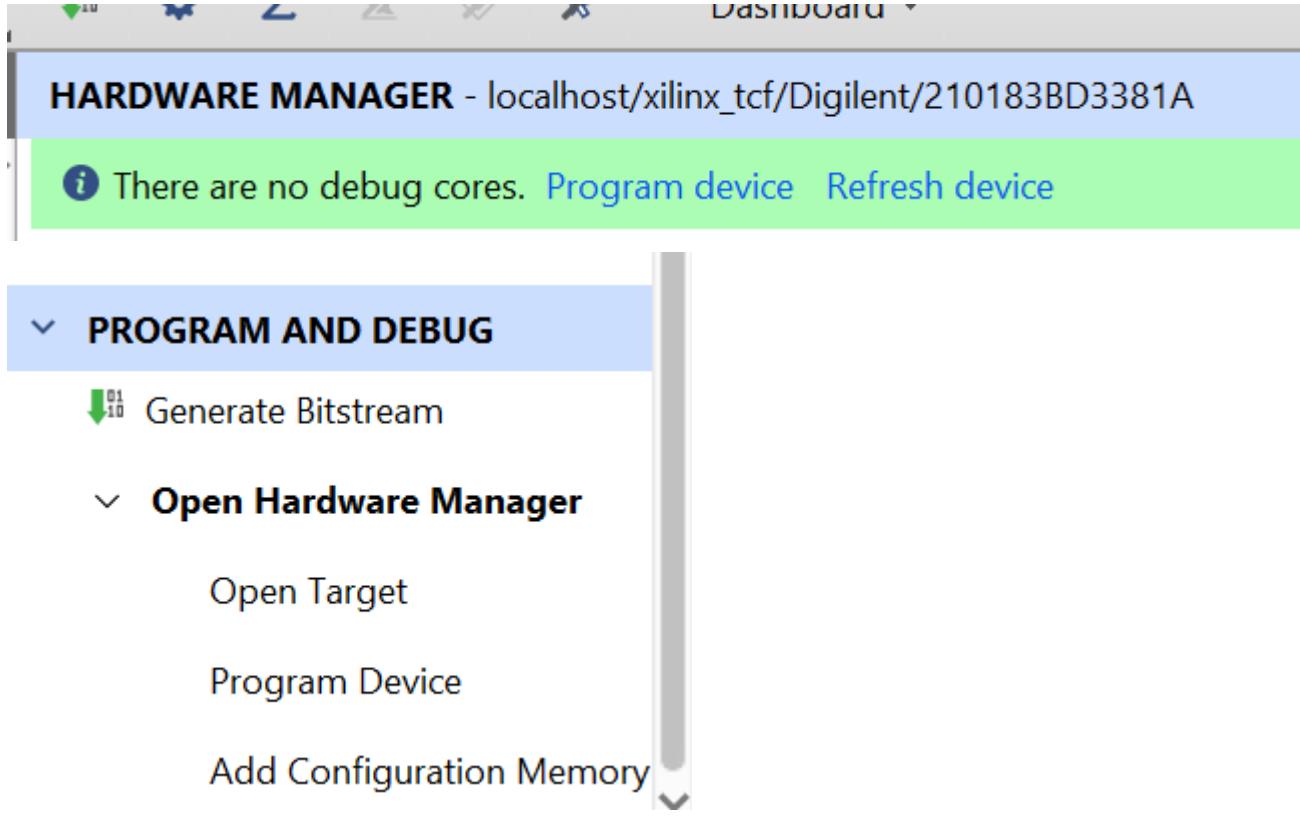
once we connected the board with the computer we can see it in the hardware manager



HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/210183BD3381A



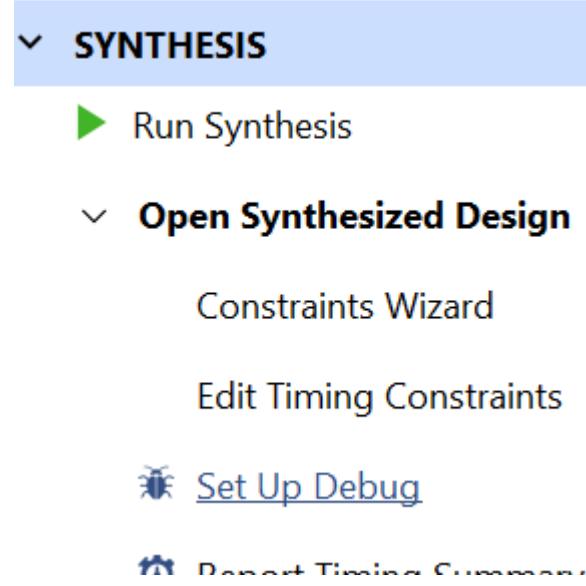
To upload the bitstream on the board we have to use the *Program device* button



03_Hardware_Debug

Technique to use and see the signals inside the FPGA like in an debugger way? Yes using the hardware debug. It can be used via GUI and is a program that samples and moves data from the FPGA to our computer and vice versa.

To do so we have to add the debugger in the synthesized design.



Here we have to choose the *nets* that we want to have in the debugger

Set Up Debug



Nets to Debug

The nets below will be debugged with ILA cores. To add nets click "Find Nets to Add". You can also select nets in the Netlist or other windows, then drag them to the list or click "Add Selected Nets".



Drag and drop nets from [Netlist View](#), [Find Results](#), or use [Find Nets to Add](#)

[Find Nets to Add...](#) Nets to debug: 0

< Back Next > Finish Cancel

ex: if we want to see the counter that we have in our setup, we have to do "*Find nets to add*" and then in finds nets we have to select our variable or signal

Find Nets

Find objects by filtering Tcl properties and objects.

Properties

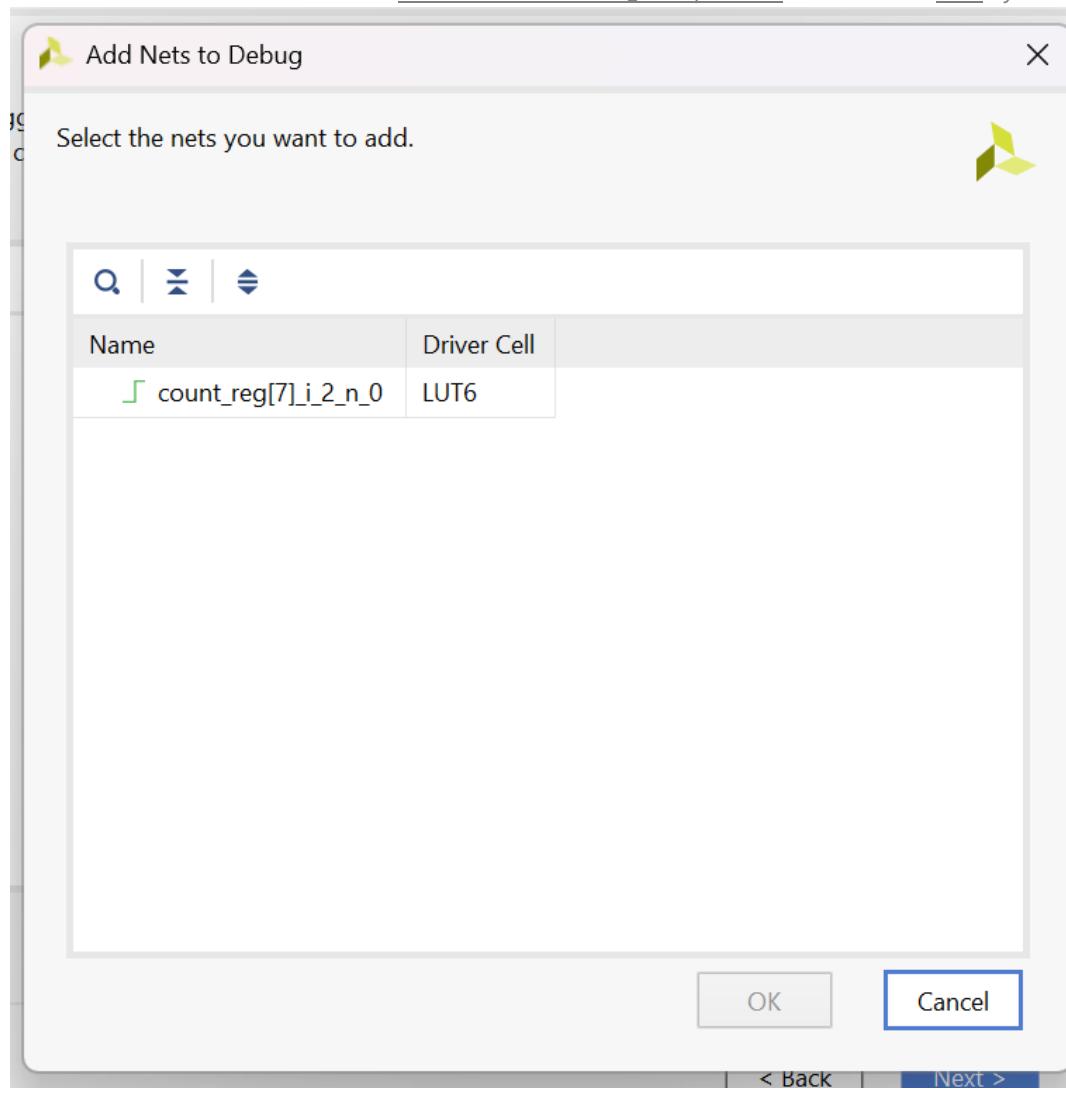
NAME contains *count_reg*

Regular expression Search hierarchically Display unique nets

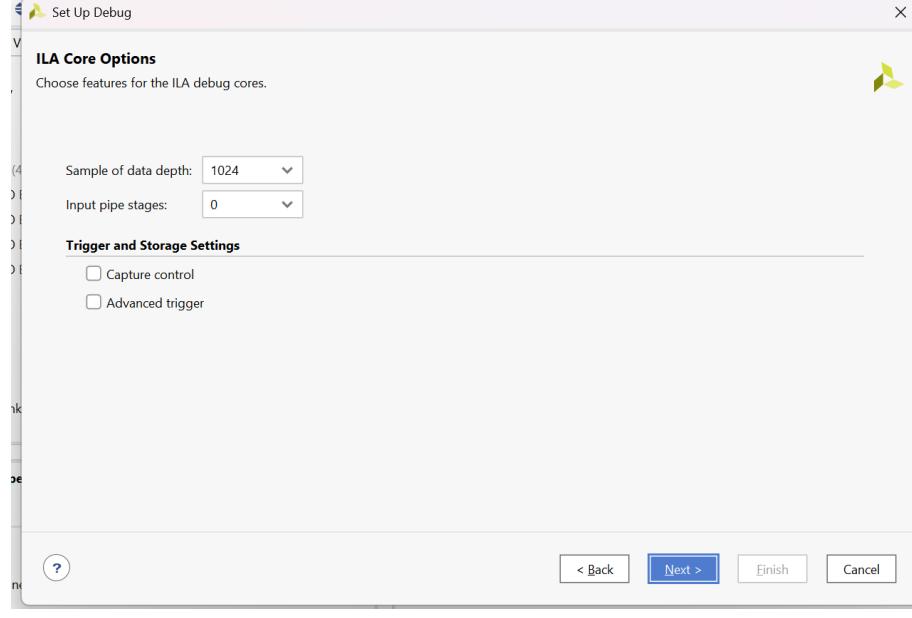
Of objects: ...

Command: show_objects -name NET_ONLY [get_nets -hierarchical -top_net_of_hierarchical_group "*"]

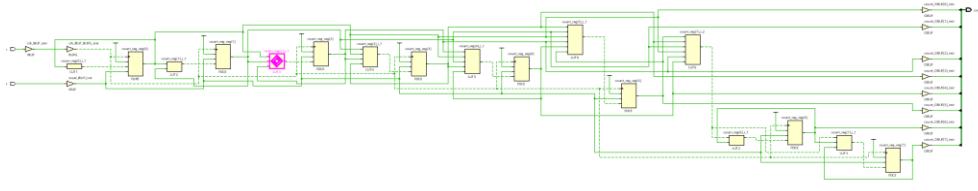
? OK Cancel



In the next section we can choose how many samples acquire in the acquisition window in the voice *sample of data depth*



we can see in the dashed line the *debugger* implemented in the synthesis layout



the debugger to sample the data uses a clock, in particular the *ch0*

clk (1)

Ch 0 (clk_IBUF_BUFG)

BUFG

O

the debugger is a synchronous module that uses that clock to sample data. We see that the *xdc* file changes and in particular under the *debug core* we have the constraints set used by the debugger

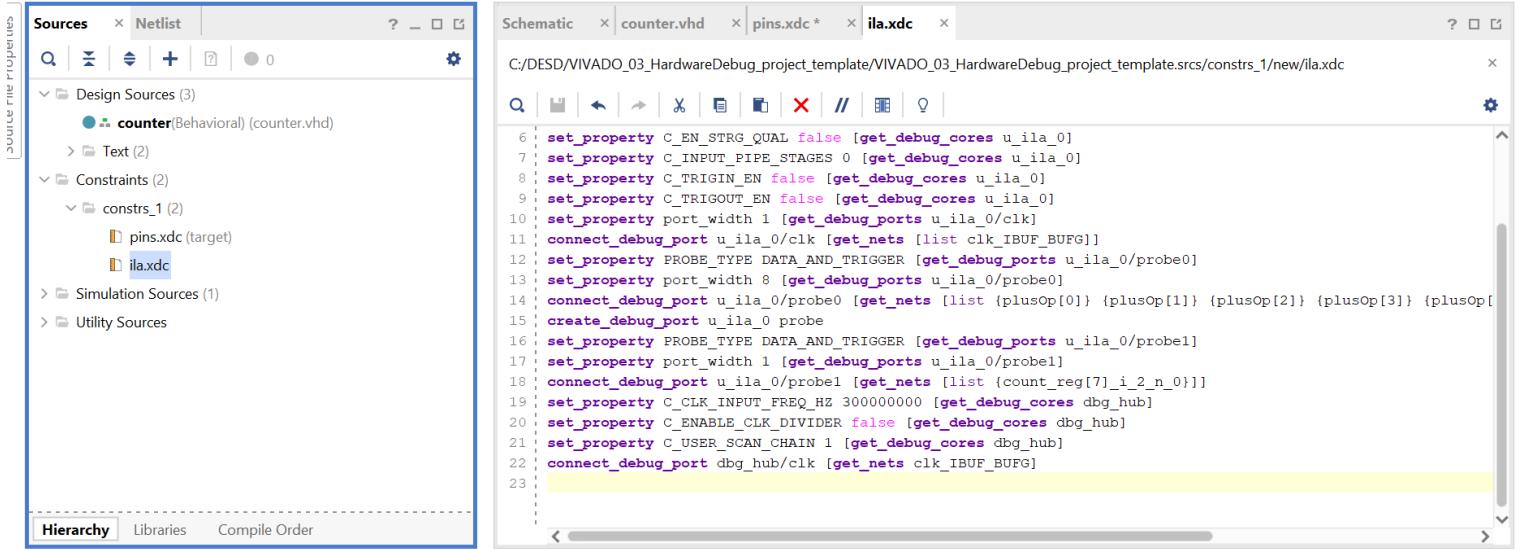
```

25 create_debug_core u_ilal_0 ila
26 set_property ALL_PROBE_SAME_MU true [get_debug_cores u_ilal_0]
27 set_property ALL_PROBE_SAME_MU_CNT 1 [get_debug_cores u_ilal_0]
28 set_property C_ADV_TRIGGER false [get_debug_cores u_ilal_0]
29 set_property C_DATA_DEPTH 1024 [get_debug_cores u_ilal_0]
30 set_property C_EN_STRG_QUAL false [get_debug_cores u_ilal_0]
31 set_property C_INPUT_PIPE_STAGES 0 [get_debug_cores u_ilal_0]
32 set_property C_TRIGIN_EN false [get_debug_cores u_ilal_0]
33 set_property C_TRIGOUT_EN false [get_debug_cores u_ilal_0]
34 set_property port_width 1 [get_debug_ports u_ilal_0/clk]
35 connect_debug_port u_ilal_0/clk [get_nets [list clk_IBUF_BUFG]]
36 set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ilal_0/probe0]
37 set_property port_width 8 [get_debug_ports u_ilal_0/probe0]
38 connect_debug_port u_ilal_0/probe0 [get_nets [list {plusOp[0]} {plusOp[1]} {plusOp[2]} {plusOp[3]} {plusOp[4]} {plusOp[5]} {plusOp[6]} {plusOp[7]}]]
39 create_debug_port u_ilal_0 probe

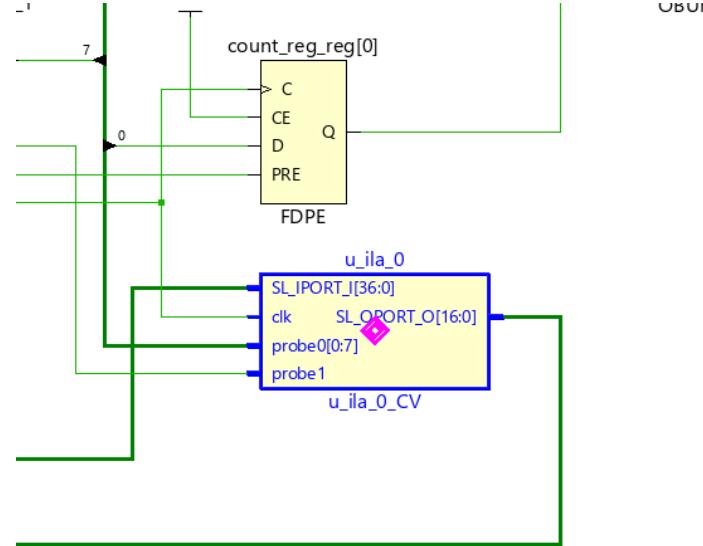
```

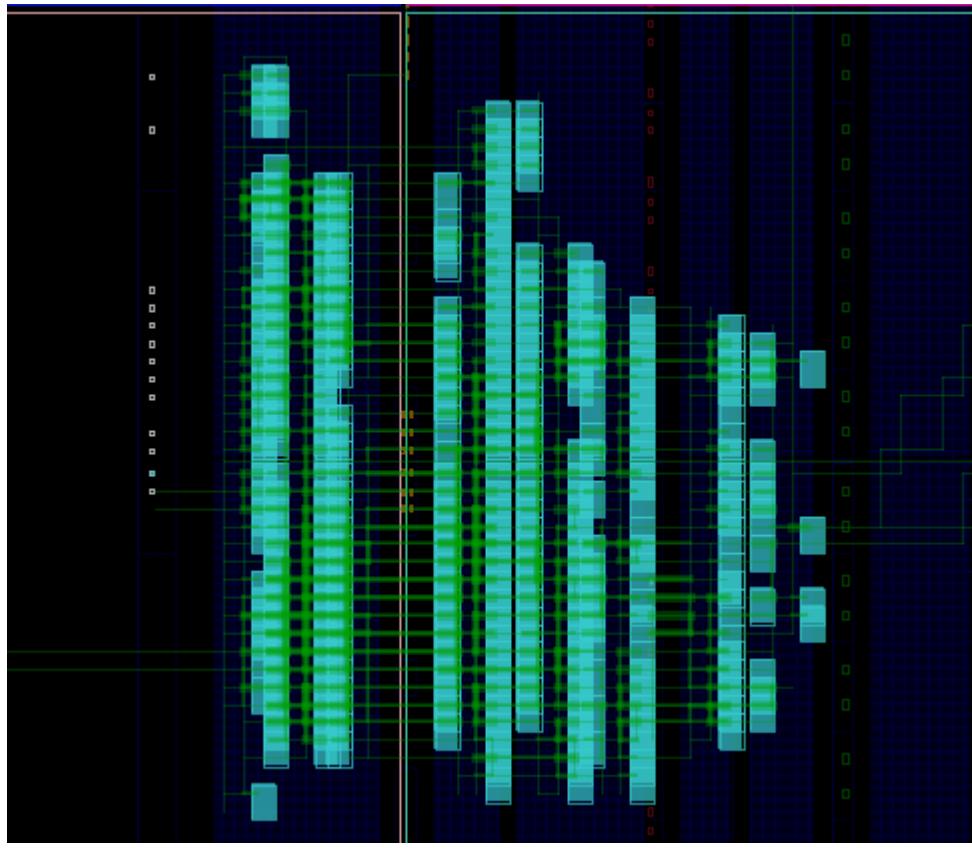
the first part is the I/O pins, in the second part we have the constraints of the debugger, as a matter of fact a part of hardware has to be set to use it. The code is human readable.

A good idea might be to create a new constraint file exclusively for the debugger.



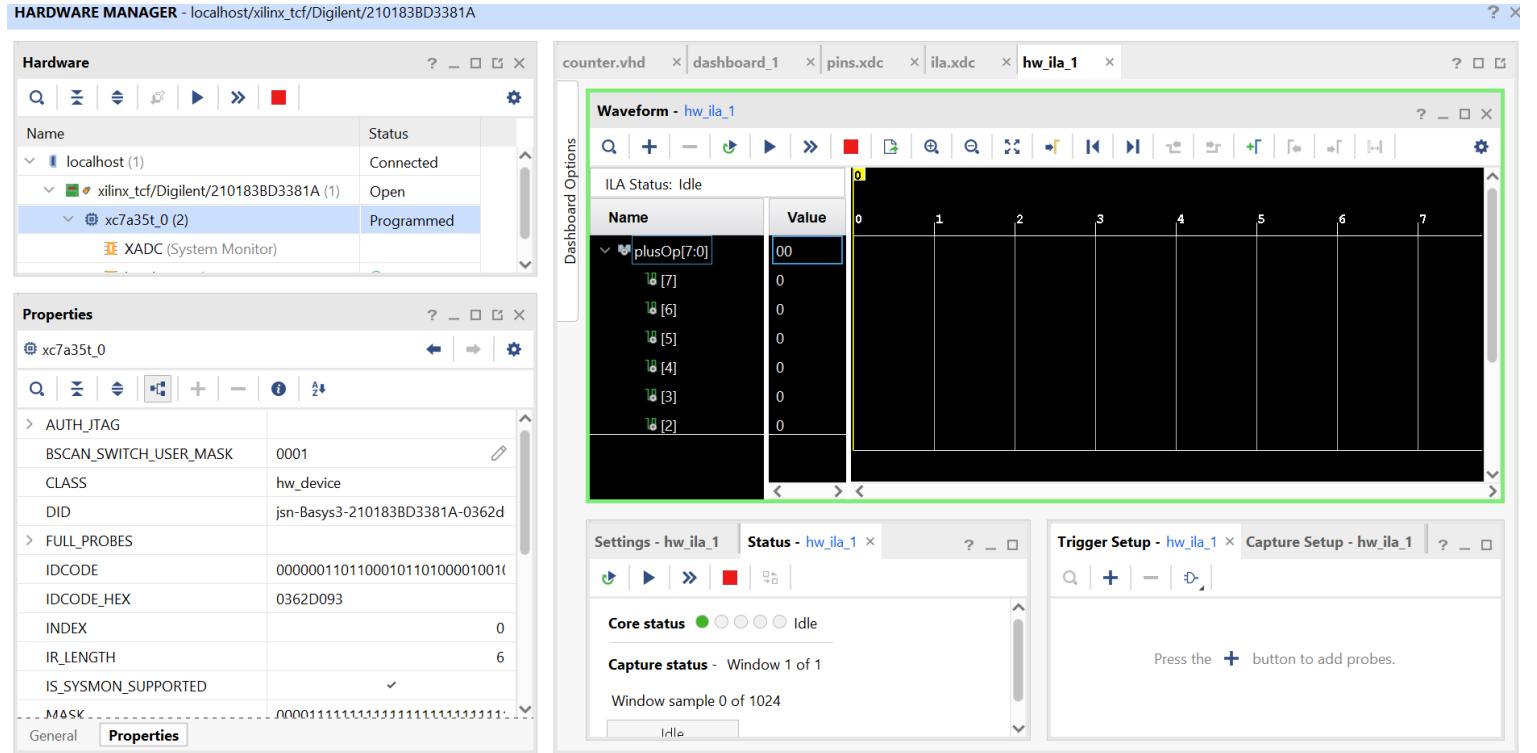
If the pins are defined and we work on the pins, we set as target the *pins.xdc* file, while if we modify the debugger properties or signal we have to set as target our *ila.xdc* file. By redoing both synthesis and implementation we can see the hardware dedicated to the debugger



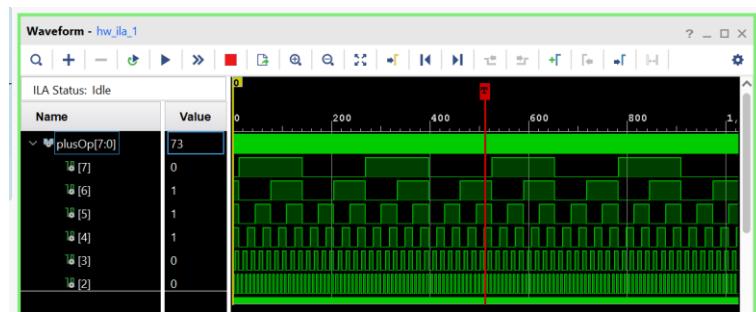


Here we can see in the implementation our ILA file and resources that we previously wrote for the debugger.

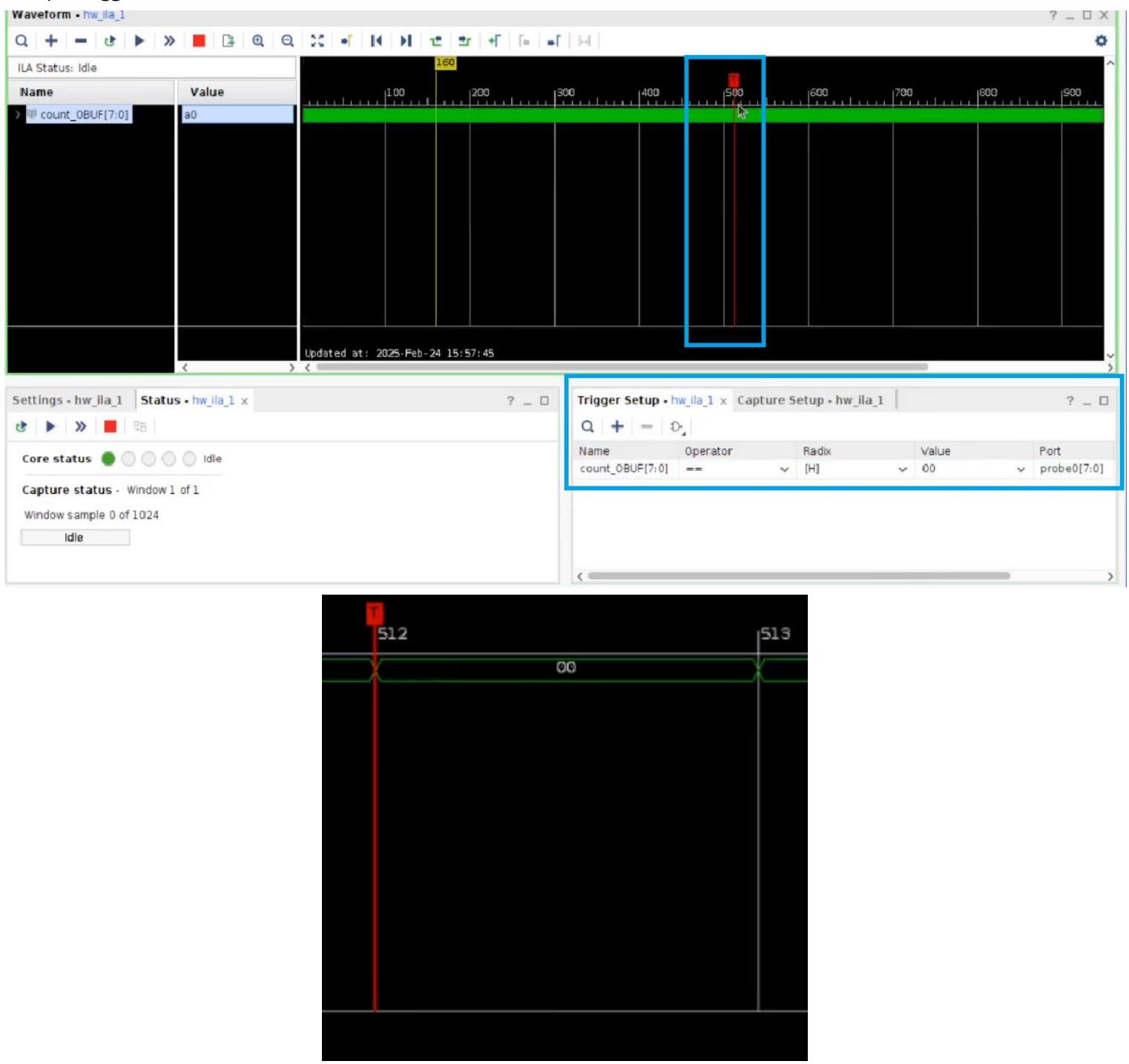
After that we generate the bitstream and load it on the device, we can see this window where we have the waveform of the debugger



here we see the signals that are read out of the FPGA.



DESD 2024/2025 – VIVADO notes – Alessandro Lazzaroni – alessandro1.lazzaroni@mail.polimi.it for feedback – [here](#) if you want to offer a me a beer we can also insert triggers, to do so we have to drag and drop the signal in the trigger setup and for example we have to setup a trigger whenever our counter reaches a certain value



If we want to remove the ILA just comment all the `ila.xdc` file and re-run the synthesis and implementation.

04_Vivado_IP_BD_BF

Board files: files that completely describe the board.

master xdc file: is a file where the mapping of GPIOs is already implemented, when we want to use a specific peripheral or device of the circuit, we can do so by uncommenting the corresponding line and putting it in an *xdc* file inside our project.

.xlm files: files that describe how a specific pin creates an interface. We have a part where a sort of translation of the *xdc* file while in the other part of the file we have all the information related to the specific interface.

How to use board files in our projects:

When Vivado asks which is the *default Xilinx part or board* that we want to use, do not use the parts selectors but go to *Boards* and search *basys 3*.

oss: obviously we previously have to integrate our board files into vivado by using the install/update boards

By using the board model we can avoid setting some constraints, like the pinout.

How is it possible to activate all the boards files?

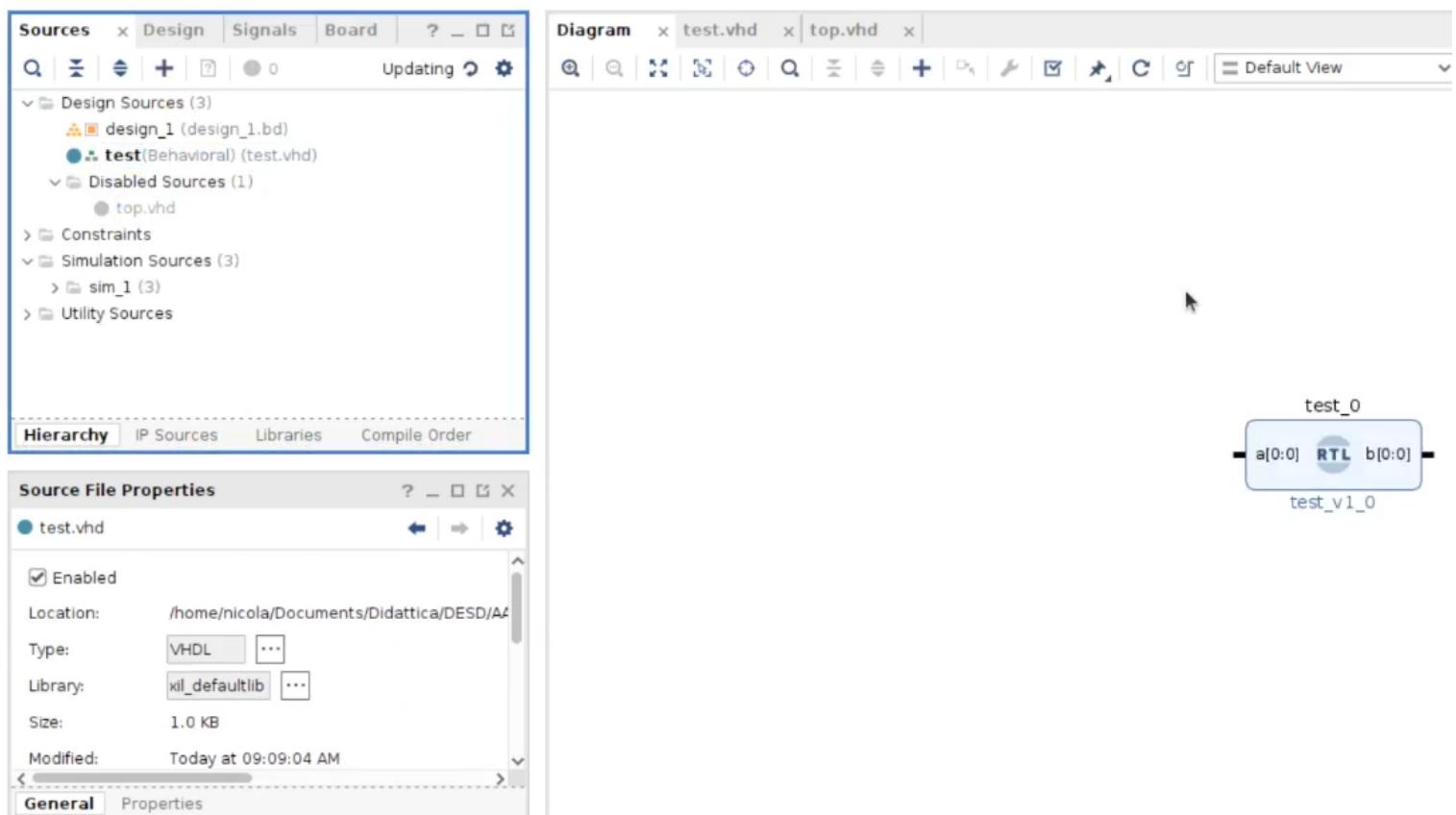
By default, only the FPGA model is used, we must use the **block design**. When for example we use a component that we previously wrote and we want to instantiate it in our VHDL module, we have to import it, do the port map etc... all these operations made with code can be performed via the GUI with block design, that is an abstraction of what we previously made.

To use block design we have to



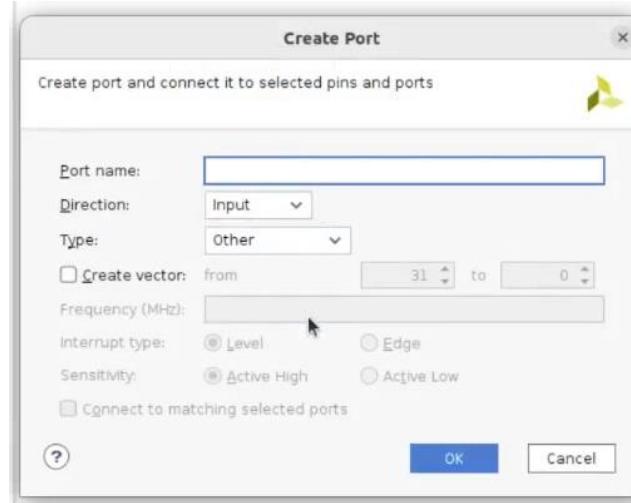
we leave it in its default folder.

In the board design we have the sources, signals and the board. By doing so we have activated our board file. To add an entity in our block design we have to disable our *top.vhd* file if we instantiated some components, so then we drag and drop the entity in our diagram.



this represent the device we wrote.

To create the connections in the diagram we have to perform *right click > create port*



ex: if we want a specific function to be performed, for example the concatenation of two vectors, we can create quickly our entity *concat* and instantiate it in our diagram.

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

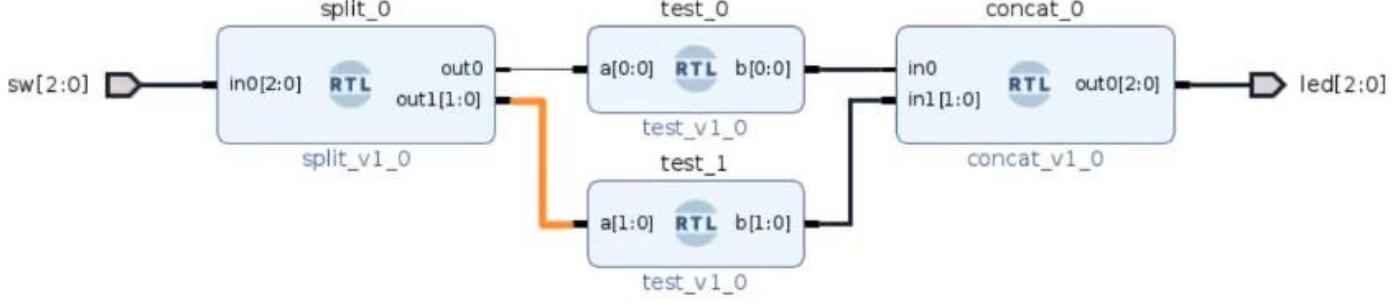
-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity split is
    Port ( in0 : in STD_LOGIC;
           in1 : in STD_LOGIC_VECTOR (1 downto 0);
           out0 : out STD_LOGIC_VECTOR (2 downto 0));
end split;

architecture Behavioral of split is
begin
    out0 <= in1&in0;
end Behavioral;
```

```
entity split is
    Port ( out0 : out STD_LOGIC;
           out1 : out STD_LOGIC_VECTOR (1 downto 0);
           in0 : in STD_LOGIC_VECTOR (2 downto 0));
end split;

architecture Behavioral of split is
begin
    out0 <= in0(0);
    out1(1 downto 0) <= in0(1 downto 0);
end Behavioral;
```



When we finished our block design, then we have to click on to check if our design is correct, if the dimensions of the signals match etc. and then we must right click on design and click on *create HDL wrapper* that

wraps our block design. We have two options to click and we use . A VHDL file is generated that implements our block diagram.

What's the big advantage of the out of context module design?

If we modify just one module, the other modules are not modified, they still are synthesized, and I don't have to do everything again each time.

How to avoid writing constraints for our board

IP cores

Pre-designed blocks written by Xilinx that can be used in our *diagram design*.

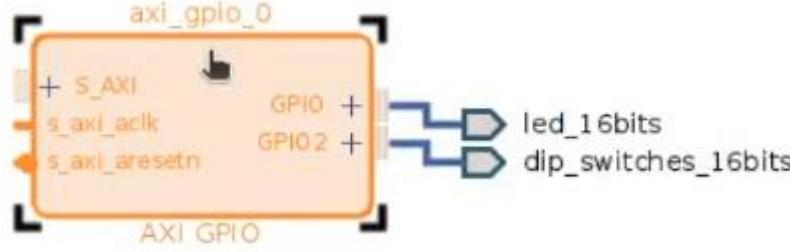
PLL IP core

The screenshot shows the Xilinx Vivado IP Integrator interface for a PLL IP core. The component name is set to `clk_wiz_0`. The configuration table for output clocks is as follows:

Output Clock	Port Name	Output Freq (MHz) Requested	Actual	Phase (degrees) Requested	Actual	Duty Cycle (%) Requested	Actual
<input checked="" type="checkbox"/> clk_out1	clk_out1	200.000	200.00000	0.000	0.000	50.000	50.0
<input checked="" type="checkbox"/> clk_out2	clk_out2	50.000	50.00000	0.000	0.000	50.000	50.0
<input type="checkbox"/> clk_out3	clk_out3	100.000	N/A	0.000	N/A	50.000	N/A
<input type="checkbox"/> clk_out4	clk_out4	100.000	N/A	0.000	N/A	50.000	N/A
<input type="checkbox"/> clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000	N/A
<input type="checkbox"/> clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000	N/A
<input type="checkbox"/> clk_out7	clk_out7	100.000	N/A	0.000	N/A	50.000	N/A

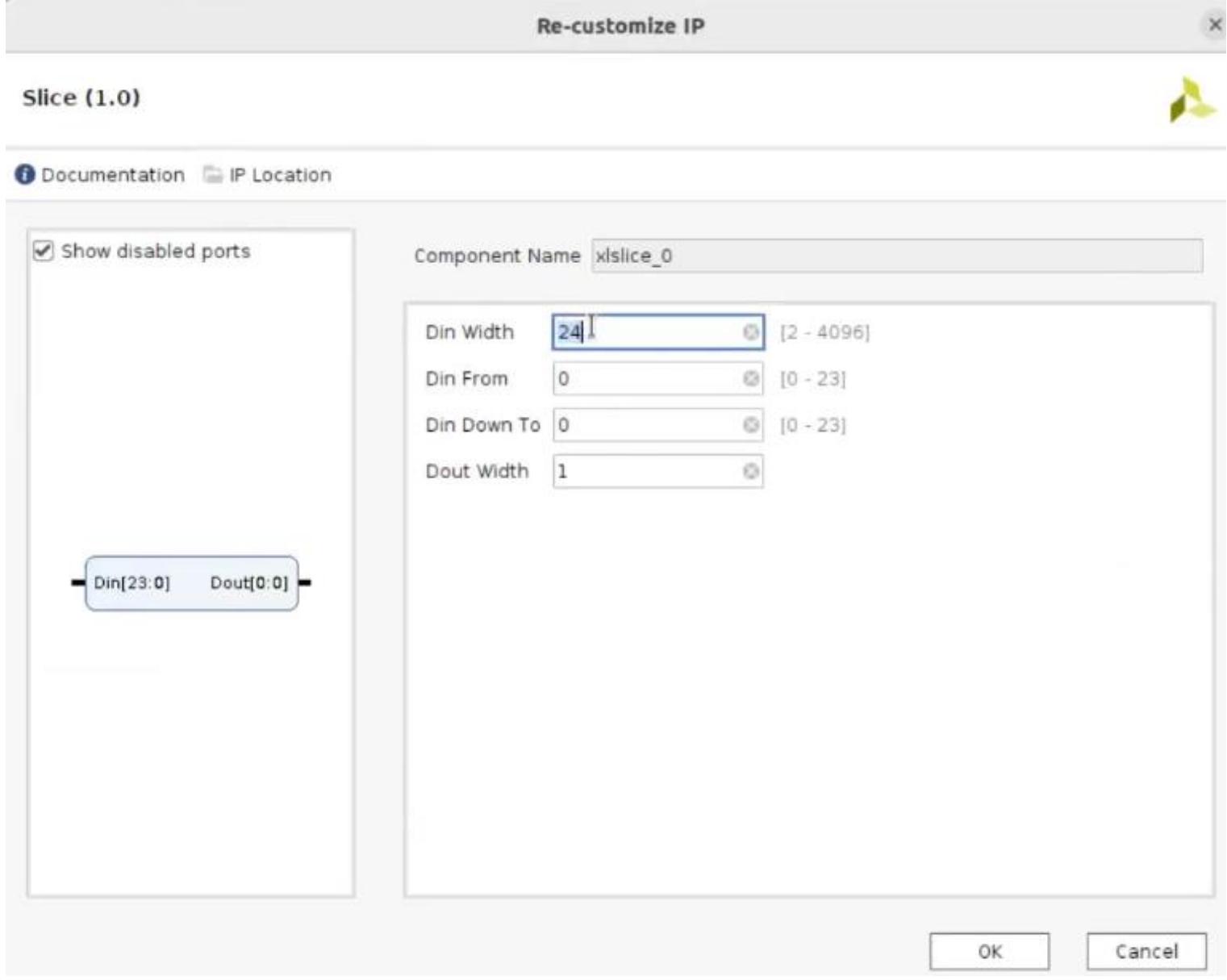
Below the table, there are sections for *USE CLOCK SEQUENCING* and *Clocking Feedback*. The *Clocking Feedback* section includes options for Source (Automatic Control On-Chip, Automatic Control Off-Chip, User-Controlled On-Chip, User-Controlled Off-Chip) and Signaling (Single-ended, Differential).

oss: when using GPIOs switches or leds we have see that an *axi_GPIO* module is introduced in the block diagram

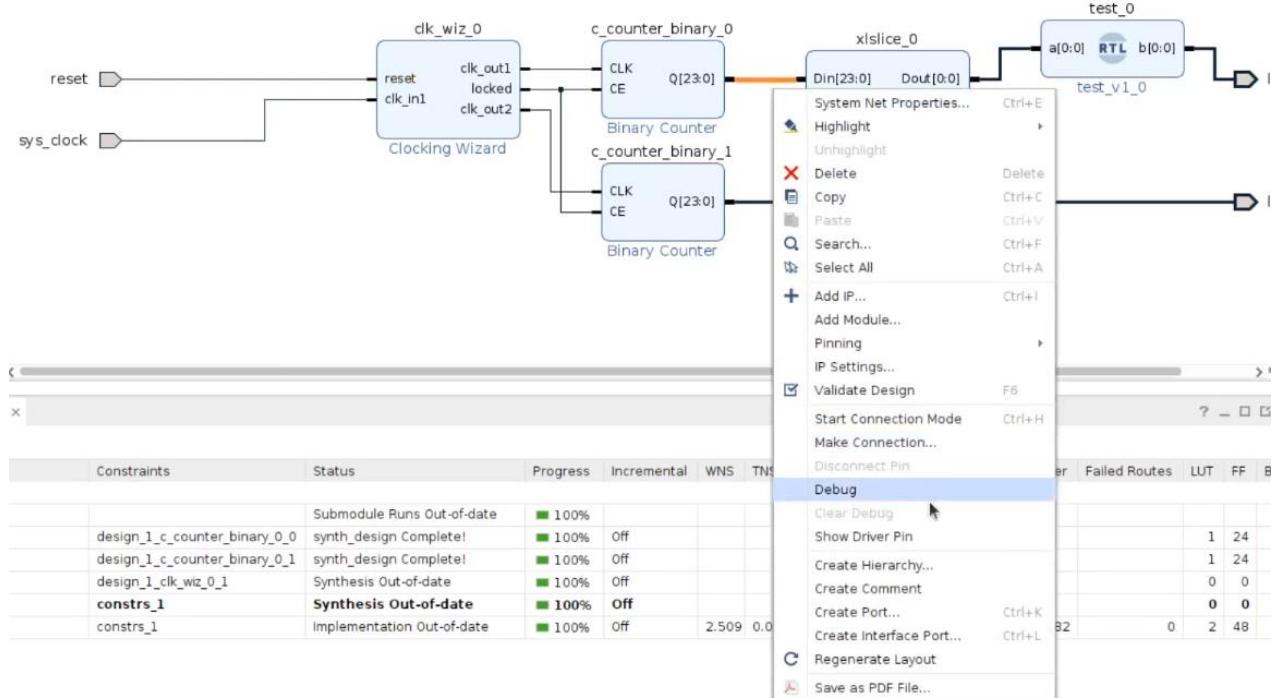


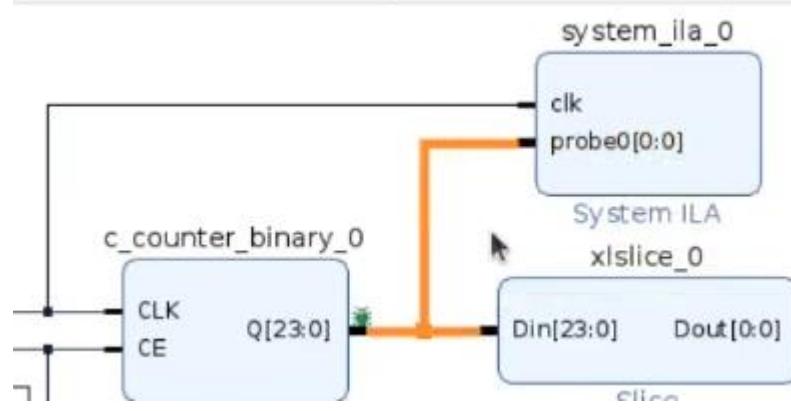
is an interface that maps leds/switches etc. into a memory, this is a memory mapped approach. To use them in the *usual* way we have to connect them with the contact.

Slice block implemented from Xilinx:



A very useful function of the block diagram approach is that if we want to debug a specific signal in our design, it is not necessary to go and during the synthesis activate the debugger and add it in the synthesized space, but it is enough to right click and select “Debug”

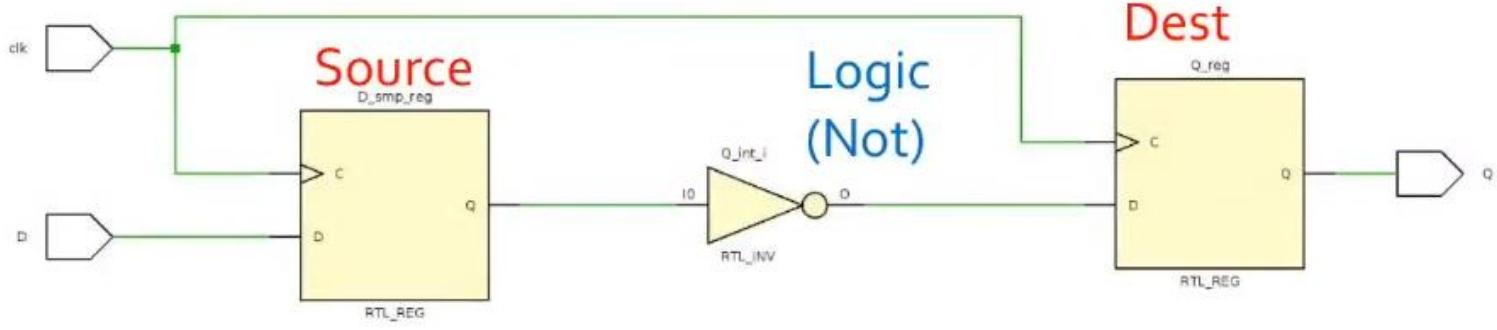




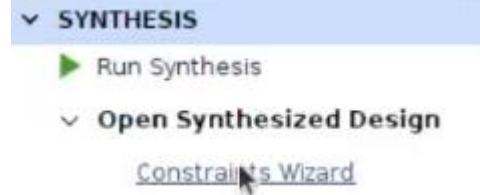
05_Vivado_Timing

How to perform the timing computation and have a report that explains each timing constraint.

Case study

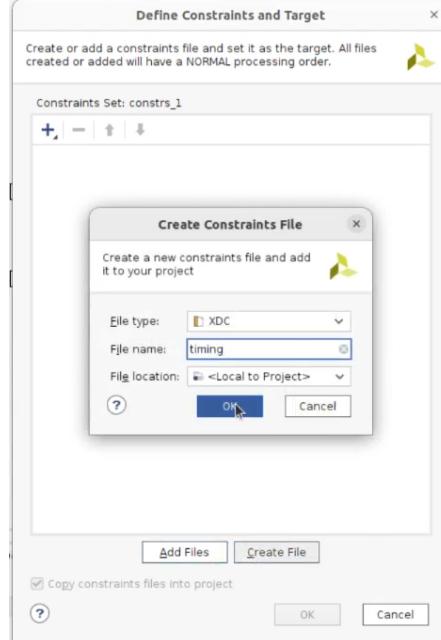


Approach #1 to timing analysis: Constraint Wizard



It is a GUI that helps us to specify the timing constraints. In this circuit everything has a propagation delay that is measured by Xilinx and is inserted into Vivado. It can also happen that the lines introduce a skew.

As we can see at the opening of the *constraints wizard* these are defined as a *xdc* file. Via this GUI we can define the target where we put all the information



oss: If we want to implement the I/O constraint or the debugger ones create different *xdc* files.

The timing constraint wizard then shows us all the types of timing constraints that we can add in the project



Timing Constraints Wizard

Identify and Recommend Missing Timing Constraints

The Timing Constraints Wizard guides you through creating timing constraints per Xilinx design methodology. It analyzes your design for missing timing constraints and makes recommendations. You need to review and understand all of the recommendations to ensure they are appropriate for your design.

Clocks:

- Primary Clocks
- Generated Clocks
- Forwarded Clocks
- External Feedback Delays

Input and Output Ports:

- Input Delays
- Output Delays
- Combinational Delays

Clock Domain Crossings:

- Physically Exclusive Clock Groups
- Logically Exclusive Clock Groups with No Interaction
- Logically Exclusive Clock Groups with Interaction
- Asynchronous Clock Domain Crossings

Clicking 'Next' on a page applies the constraints to the design in memory, so that missing constraints on subsequent pages can be identified. Each page may require considerable runtime to discover missing constraints.

The Clock Networks report is available on every page to help you review the constraints. Schematics and timing path reports are available on the Asynchronous Clock Domain Crossings page.

To leave the Wizard and automatically save the new constraints to the target XDC file, click Finish. To discard the new constraints click Cancel.

XILINX

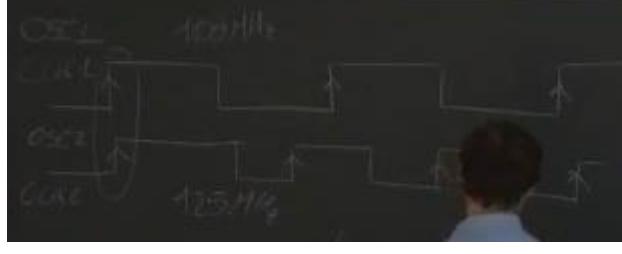
?

Next  Skip to Finish >> Cancel

oss: what does “clocks without synchronism” mean? We might have different oscillators that generate different clocks and have different floorplanning. If the clocks come from different sources

(ex one internal osc, the other one from external) they do not have a phase relation.

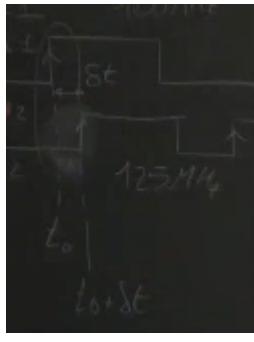
At the power up we have maybe



$$T_4 = n T_{osc1} + \phi$$

$$T_2 = n T_{osc2} + \phi$$

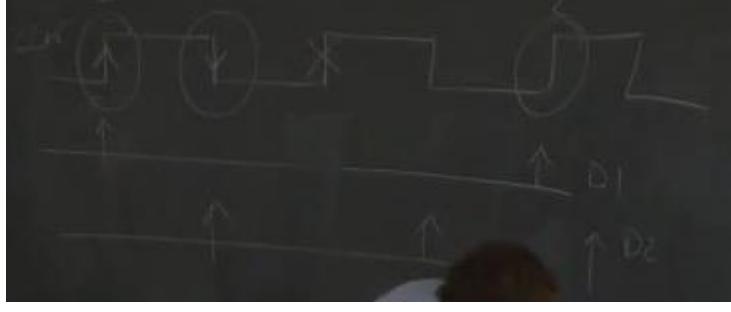
or maybe at the power up we have this situation



$$T_1 = n T_{clk} + t_0$$

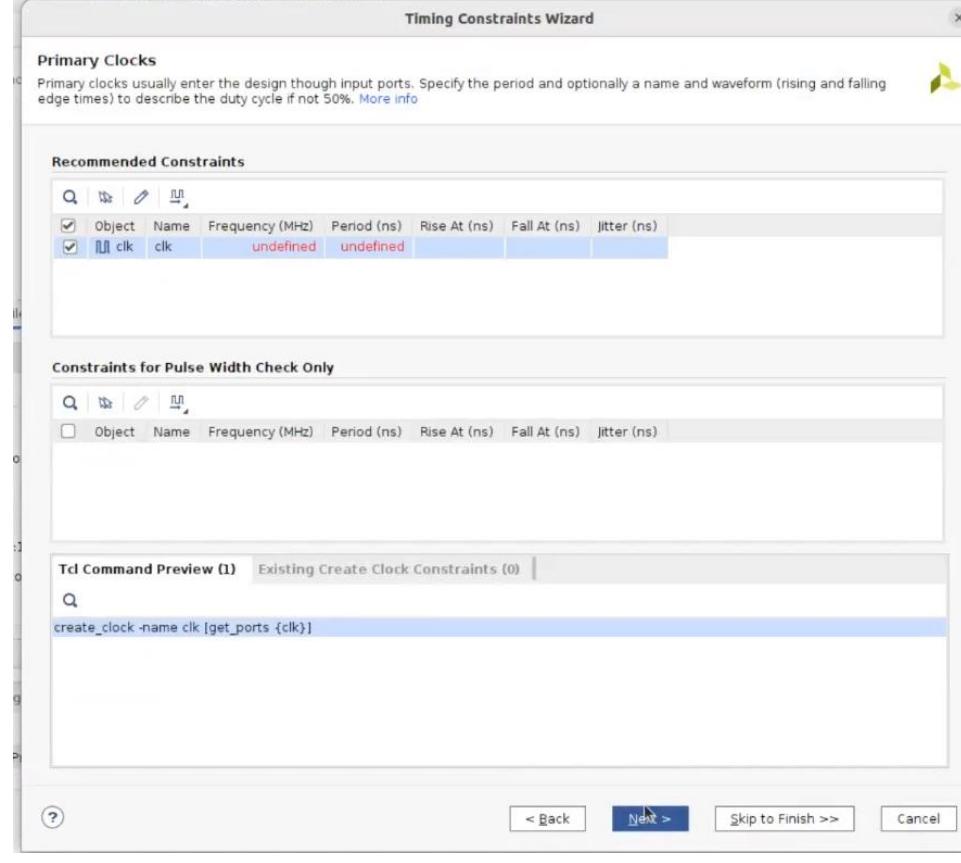
$$T_2 = n T_{clk} + t_0 + \delta t$$

and δt changes at each reset or power up of the system. This is the definition of clocks without phase relationship. We can also have *locked in phase* or *synchronous clock*, maybe we're using a PLL that generates the clock. The PLL guarantees that the δt is constant. In that case we talk about synchronous clocks because we know the relationship between the two waves. We can also have the *derived clocks* that are a subset of synchronous clocks in which we use just a single clock: maybe one domain works on the rising edge of the clock while the other one works on the falling edge of that clock.



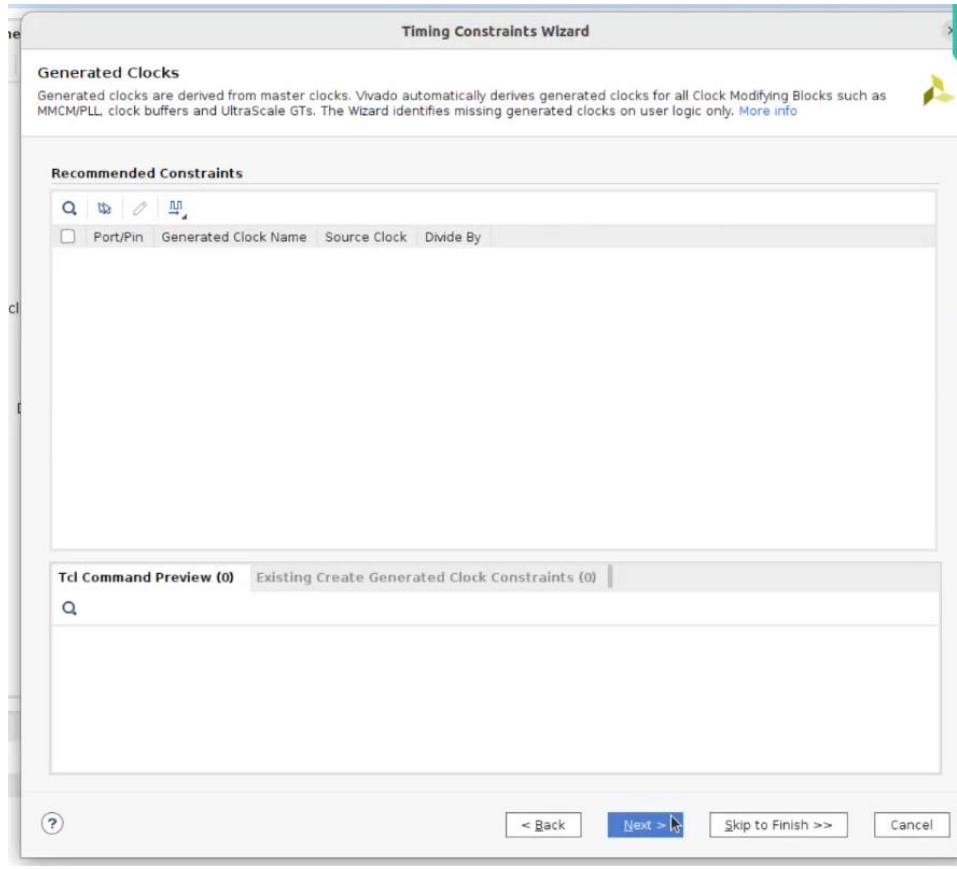
We can also find all these informations described in the timing wizard.

single clock domain

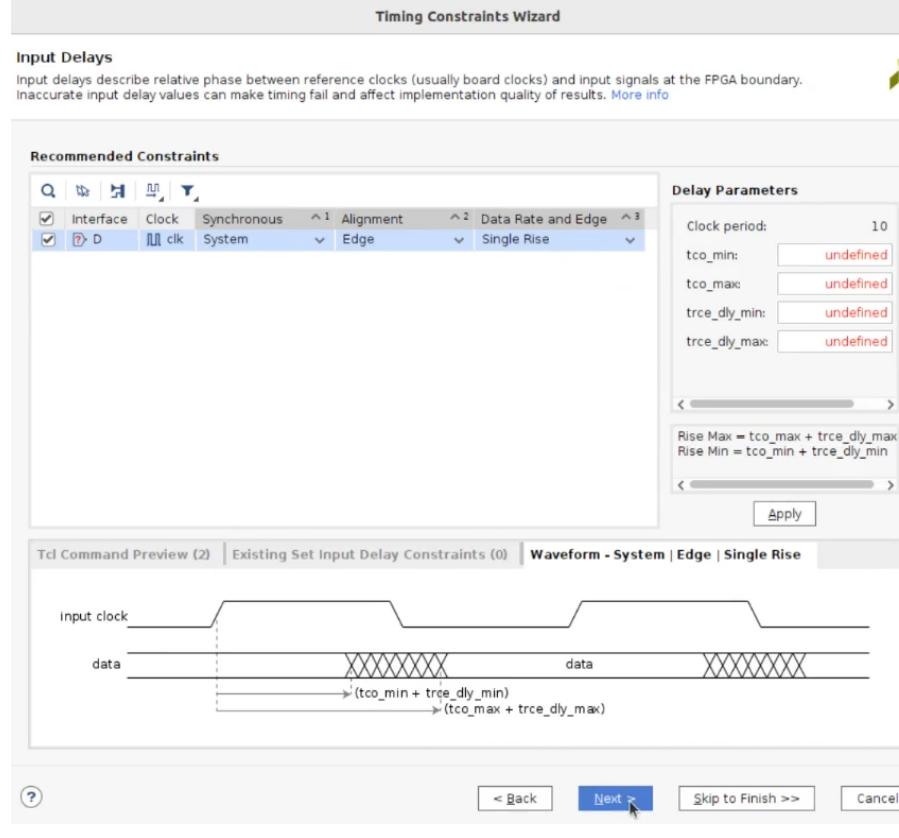


vivado knows which signal is a clk because it knows which signal is connected to the oscillator of the FPGA, it could be named whatever name.

Generated clocks is the wizard for the setup of asynchronous clocks or other clock domain. If we use a PLL here we have to insert our constraint, if we use the IP core it automatically manages all the constraints.

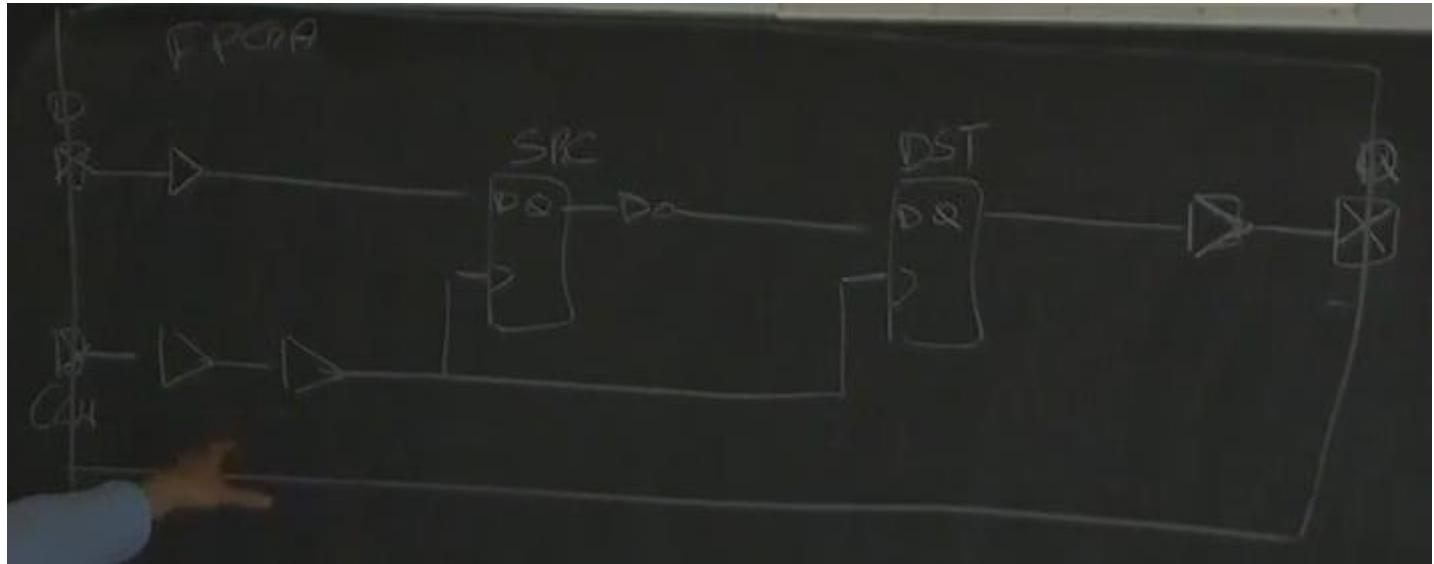


additional I/O delays

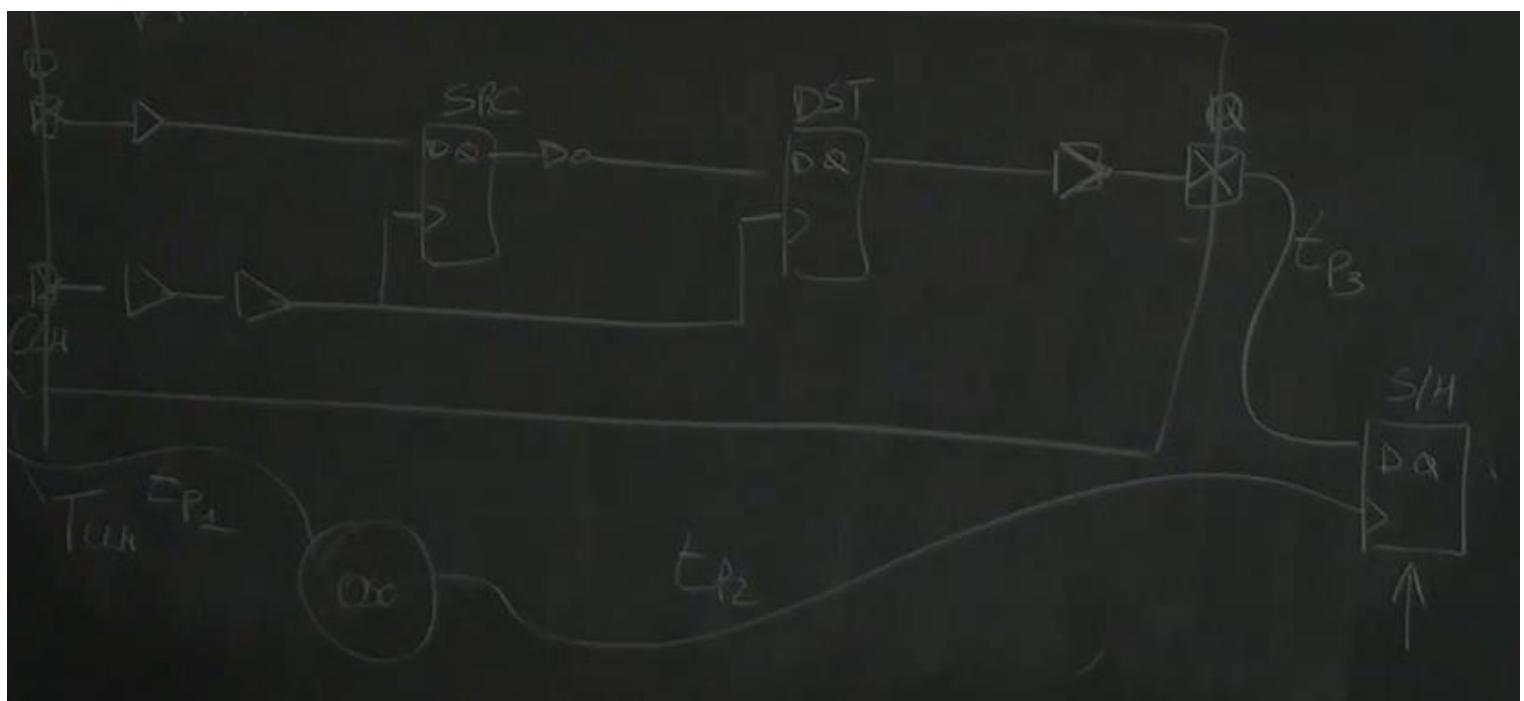


We said that Vivado knows all the delays of the circuit, that is true only for the input to output pin.

This means that we have our FPGA and Vivado knows it



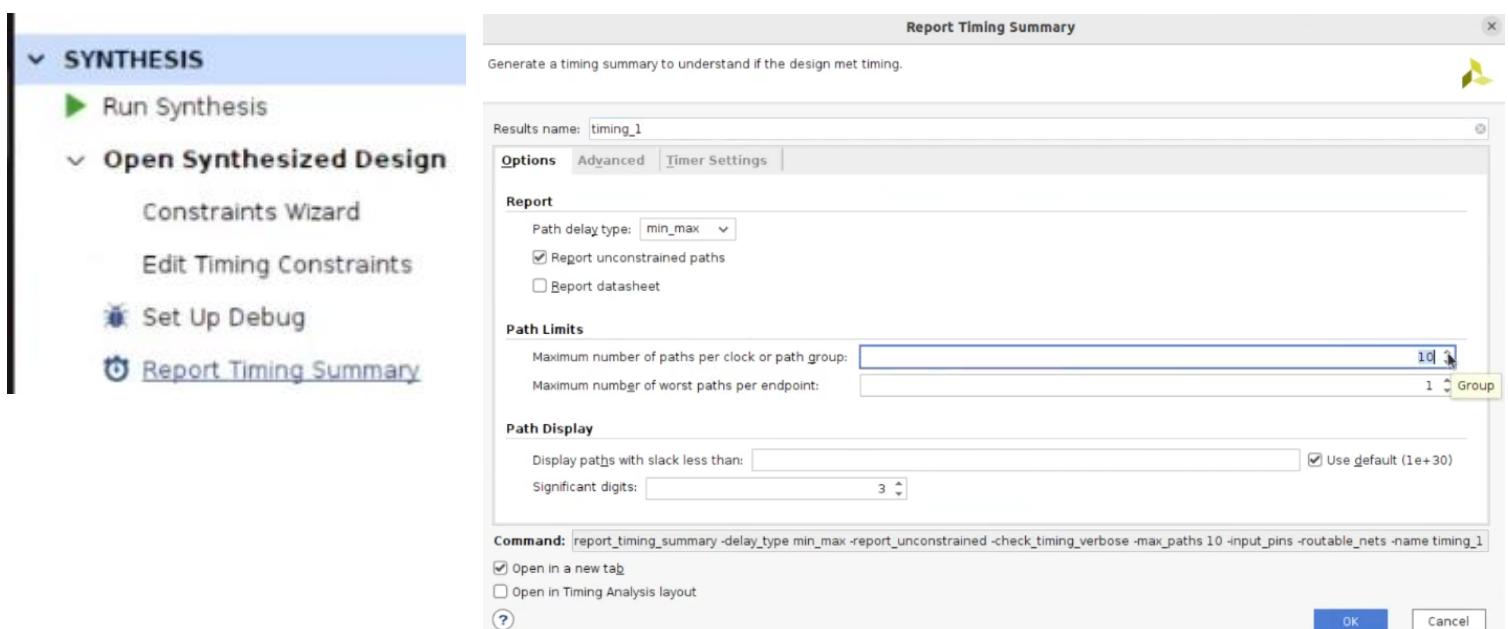
let's suppose that our Q output is connected to an external flip flop that is connected to the same clock of the FPGA.
Vivado can't know the propagation delays (tp_1, tp_2, tp_3, t_s and t_h) of the external circuit



using this wizard we can specify all these constraints for the I/O signals. These constraints are mandatory
for examples for communication interfaces.

How to perform the timing analysis – Synthesized Design

We click on *Report Timing Summary*



- *Maximum number of paths per clocks or path group:* reports the worst n cases of the timing
all the other parameters can be left on default

Then the *Timing report* opens

here we can see the *setup* and *hold* analysis

Timing Check	Count	Worst Severity
no_input_delay	1	High
no_output_delay	1	High
no_clock	0	
constant_clock	0	
pulse_width_clock	0	
unconstrained_internal_endpoints	0	
multiple_clock	0	
generated_clocks	0	
loops	0	
partial_input_delay	0	
partial_output_delay	0	
latch_loops	0	

here Vivado let us know that we did not setup the external constraints for other signals so probably it is not a general timing analysis.

The most interesting part is *Intra clock paths*

Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Δ_1 (TNS)	Failing Endpoints	Δ_2 (TNS)	Total Endpoints	Δ_3	Edges (WHS)	WHS (ns)	THS (ns)	Failing Endpoints	Total Endpoints (THS)	WPWS (ns)	TPWS (ns)	Failing Endpoints (TPWS)	Total Endpoints (TPWS)
clk	rise - rise	8.787	0.000		0		1		rise - rise	0.134	0.000	0	1	4.500	0.000	0	0

Clock: clk

Statistics

Type	Worst Slack	Total Violation	Failing Endpoints	Total Endpoints
Setup	8.787 ns	0.000 ns	0	1
Hold	0.134 ns	0.000 ns	0	1
Pulse Width	4.500 ns	0.000 ns	0	3

this is the timing analysis of the internal clock of the FPGA.

NB: this analysis is made on the synthesized design, not on the implemented one. If we want to know the exact delay of all the paths we have to previously perform the floorplanning inside the board. By the way, the synthesis analysis is performed in a worst case scenario, so usually the implementation one is less restrictive than the synthesized one.

To be accurate we have to run the implementation and we obtain the report.

Slack and timing equations

When performing the timing analysis we do not see the equation that we used at lesson, so

$$t_{SETUP} < T_{CLK} + t_{SW} - t_j - t_{PATH}$$

but the equation we see is

$$SLACK = T_{CLK} + t_{SW} - t_j - t_{PATH} - t_{SETUP} = T_{REQUIRED} - T_{ARRIVED}$$

the *slack* represents the time required to respect the timing constraints

We realize that the stuff that helps us to respect the timing is $t_{jitter}, t_{path}, t_{SETUP}$ while the *clock period* and the *skew* increase the required time.

If the slack is positive it means that our setup time is lower than $t_{jitter}, t_{path}, t_{SETUP}$ so in this input to output connection we *have margin to make a slower path or increase the clock frequency*, while if the slack is negative we are not respecting the timing constraints.

How to perform the timing analysis – Implemented Design

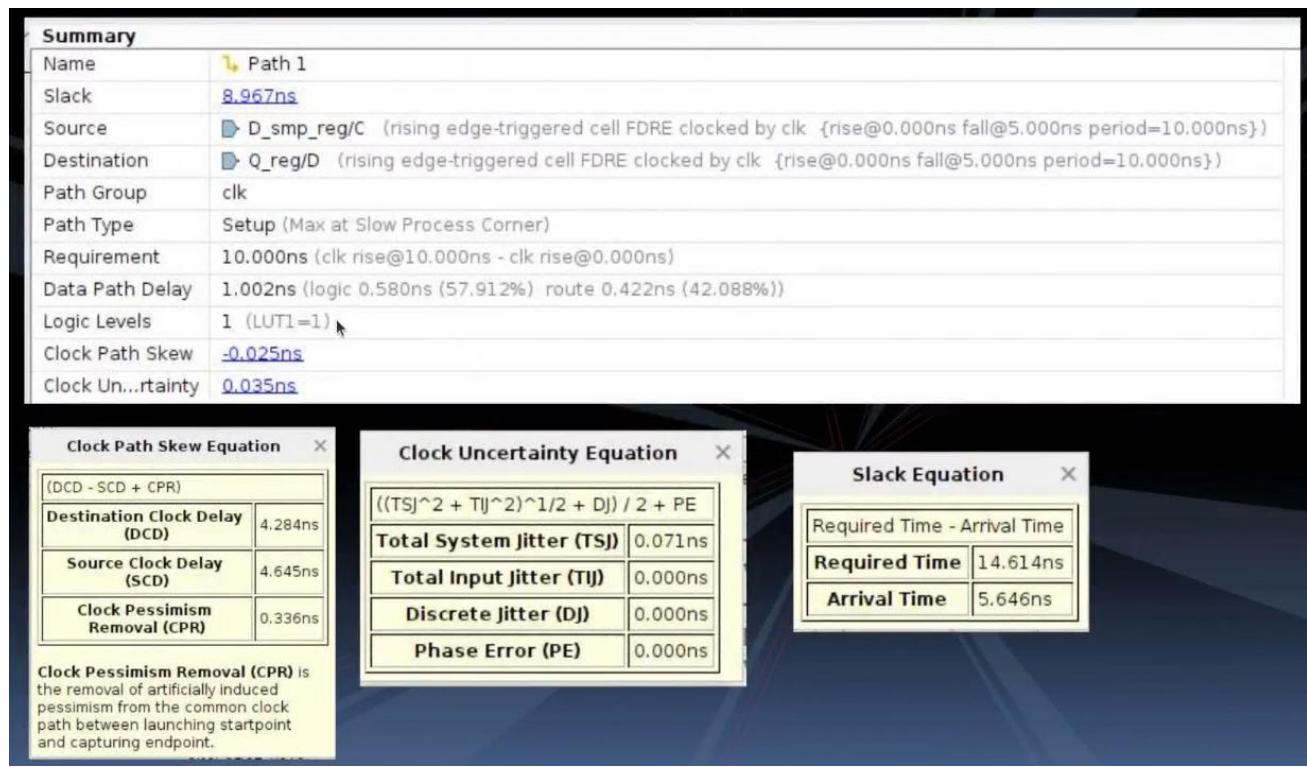
After running the implementation, we can click on *Report Timing Summary* and see all the reports

Design Timing Summary					
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8.967 ns	Worst Hold Slack (WHS):	0.236 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1	Total Number of Endpoints:	1	Total Number of Endpoints:	3

All user specified timing constraints are met.

here we have all the accurated timings.

The report is kind of complicated to read the first time, so in the slides is reported a detailed analysis



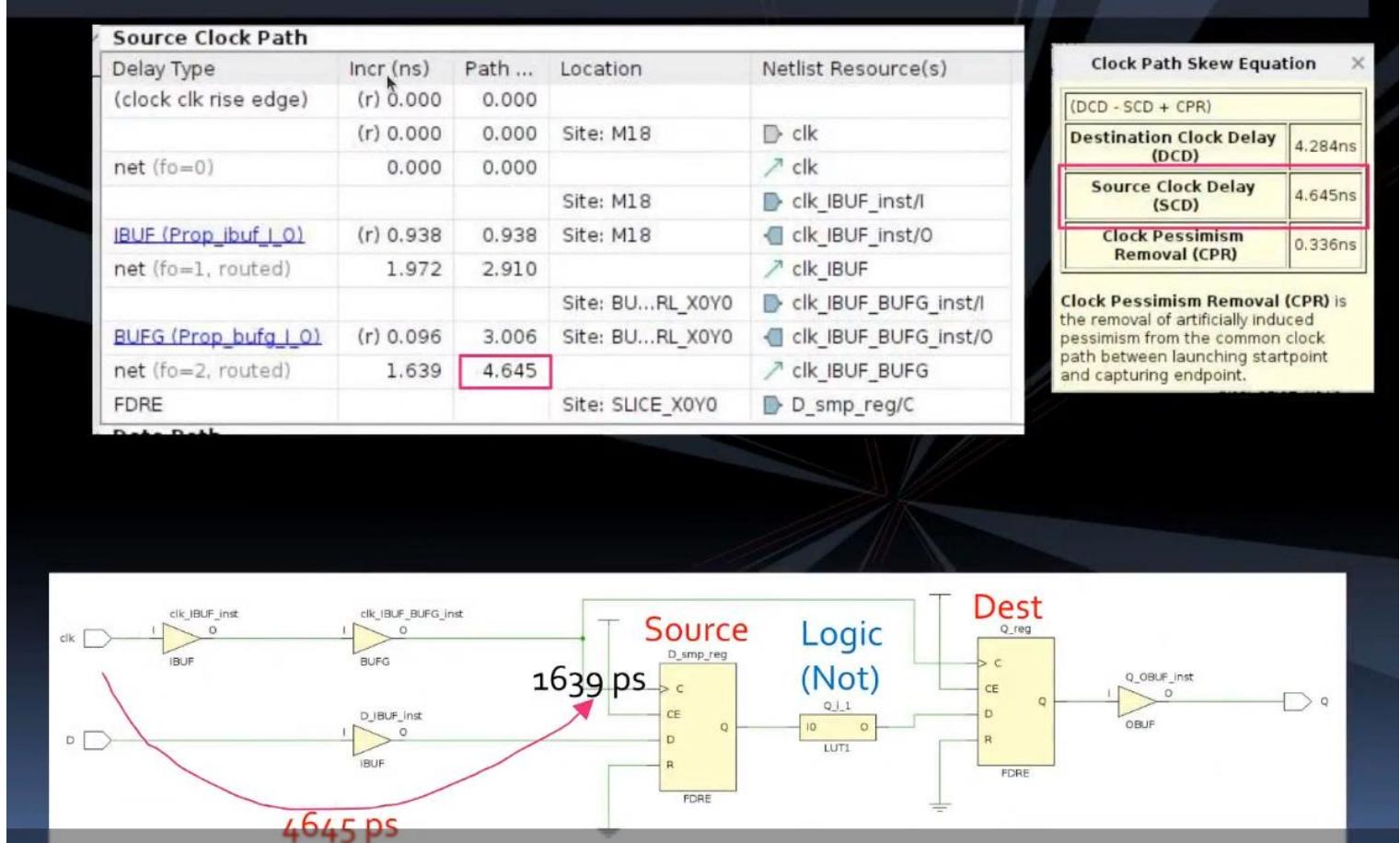
we can see the *Clock Path Skew* and the *Clock Uncertainty*. Note that we have a negative skew so this clock is in advance with respect to the reference clock. We can see that also the equations both for clock uncertainty and for the slack are reported. Note that we can speed up a little our circuit if the slack is positive.

oss: note that sometimes like in this case it would look like we could speed up our circuit up to

$$\frac{1}{\text{actual clock-slack}} = \frac{1}{(10-8.967)\text{ns}} = 968 \text{ MHz}$$

but the clock buffer has a bandwidth up to 600 – 700MHz so it would act like a low pass filter.

So we always have to compare the timing analysis result with the effective capability of the hardware.



The *incr* column tells us the contributions to the *source clock delay*

Vivado also performs the same computation from source to destination

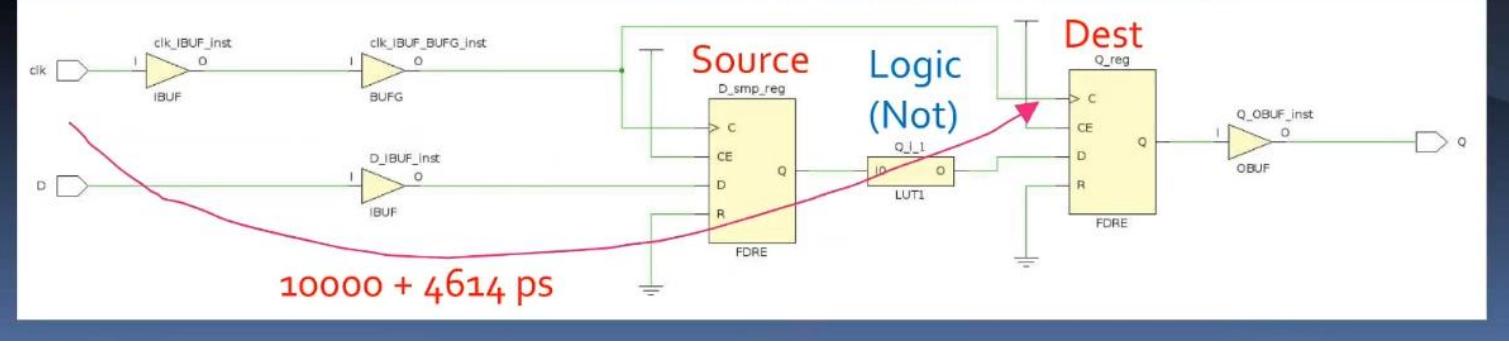
Bor

Setup – Destination Clock

Destination Clock Path				
Delay Type (clock clk rise edge)	Incr (ns) (r) 10.000	Path (...) 10.000	Location	Netlist Resource(s)
net (fo=0)	(r) 0.000	10.000	Site: M18	clk
			Site: M18	clk_IBUF_inst/l
<u>IBUF (Prop_ibuf_l_0)</u>	(r) 0.805	10.805	Site: M18	clk_IBUF_inst/0
net (fo=1, routed)	1.868	12.673		clk_IBUF
			Site: BU...RL_XOYO	clk_IBUF_BUFG_inst/l
<u>BUFG (Prop_bufg_l_0)</u>	(r) 0.091	12.764	Site: BU...RL_XOYO	clk_IBUF_BUFG_inst/0
net (fo=2, routed)	1.520	14.284		clk_IBUF_BUFG
FDRE			Site: SLICE_X0Y1	Q_reg/C
clock pessimism	0.336	14.620		
clock uncertainty	-0.035	14.585		
<u>FDRE (Set...dre_C_D)</u>	0.029	14.614	Site: SLICE_X0Y1	Q_reg
Required Time		14.614		

Clock Path Skew Equation	
(DCD - SCD + CPR)	
Destination Clock Delay (DCD)	4.284ns
Source Clock Delay (SCD)	4.645ns
Clock Pessimism Removal (CPR)	0.336ns

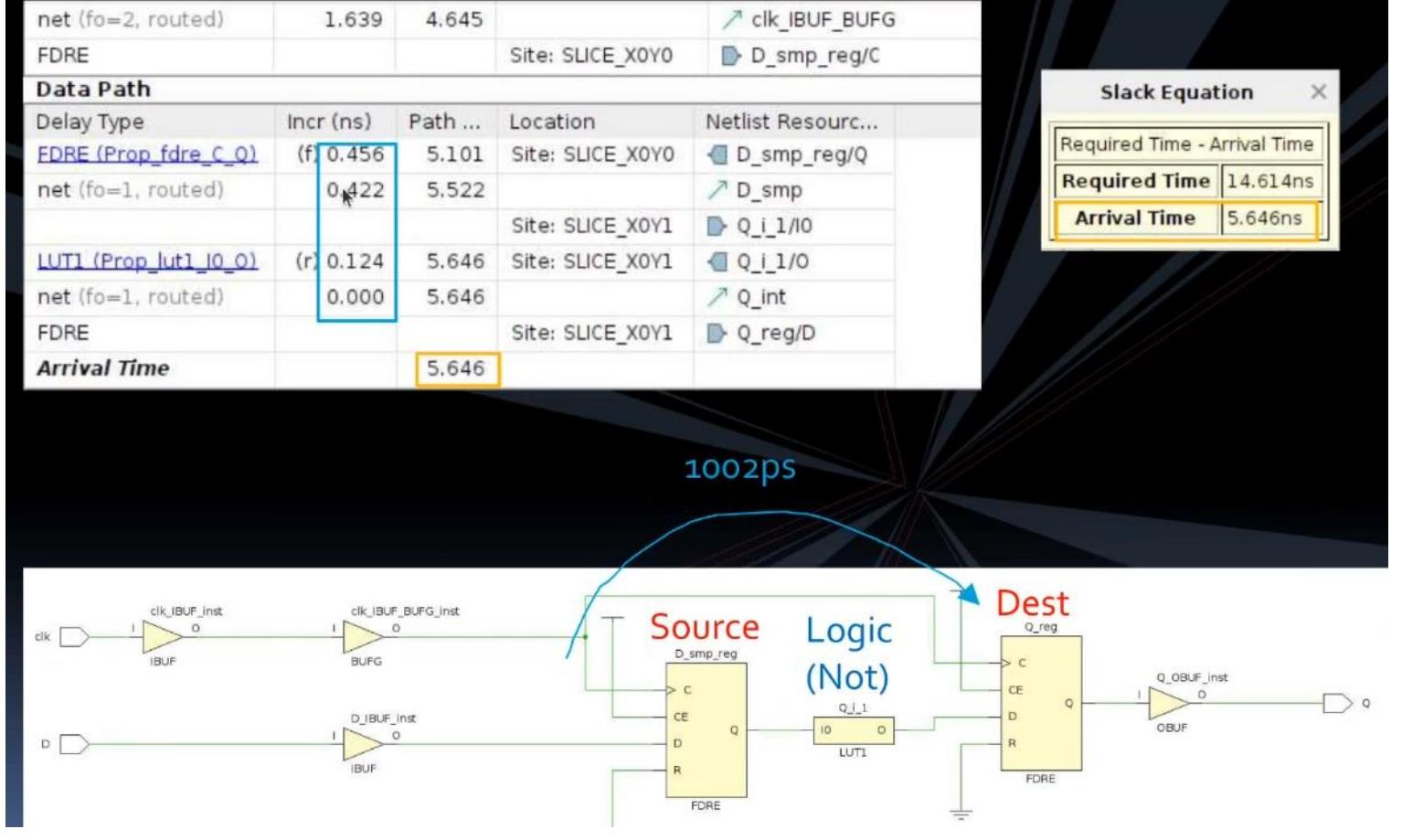
Clock Pessimism Removal (CPR) is the removal of artificially induced pessimism from the common clock path between launching startpoint and capturing endpoint.



After that we also see the clock jitters etc.

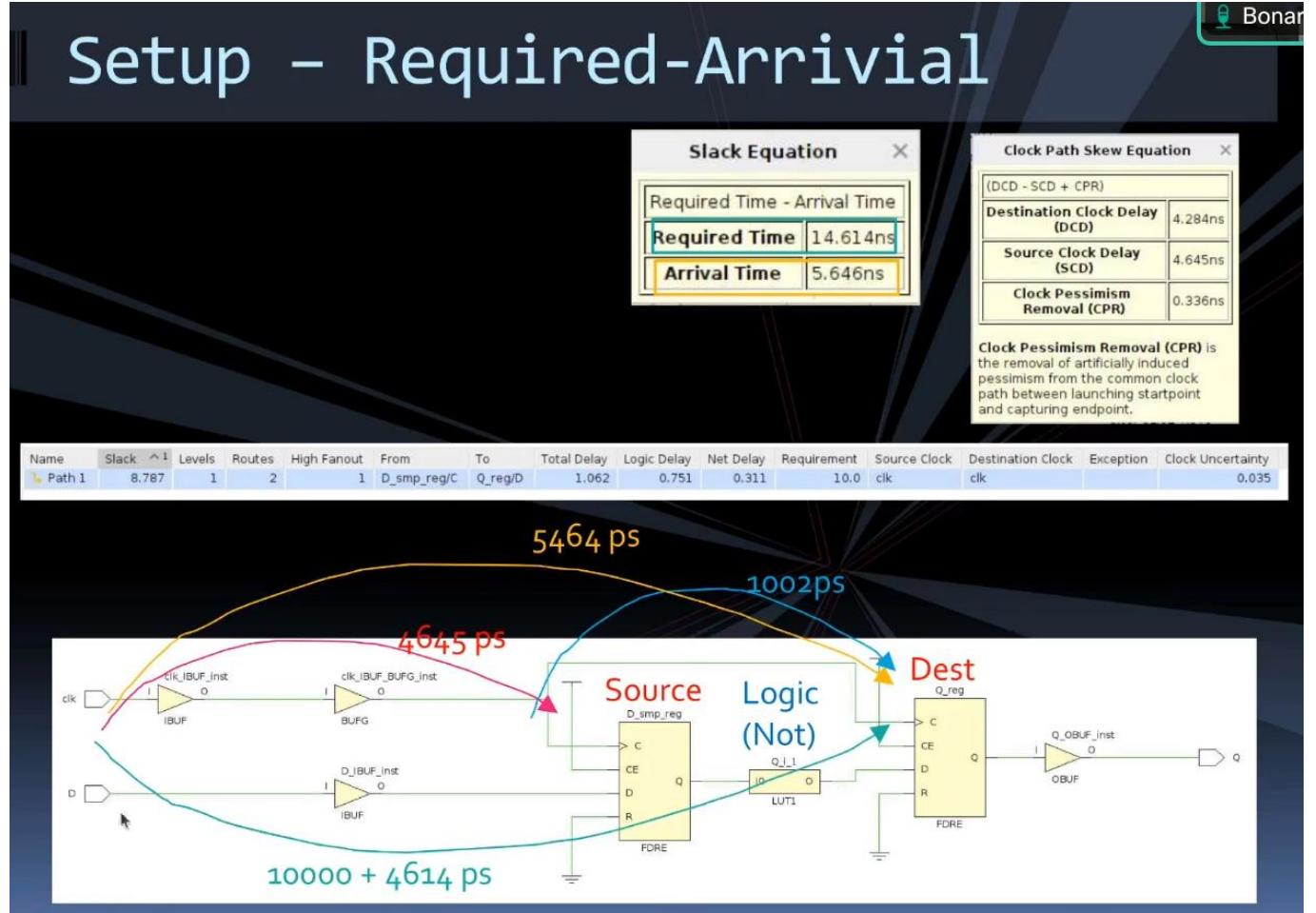


Setup – Data Path



This is the datapath, so from the source flip flop to the destination flip flop, we have all the propagation delays.

We can see a summary here with the computed constraints



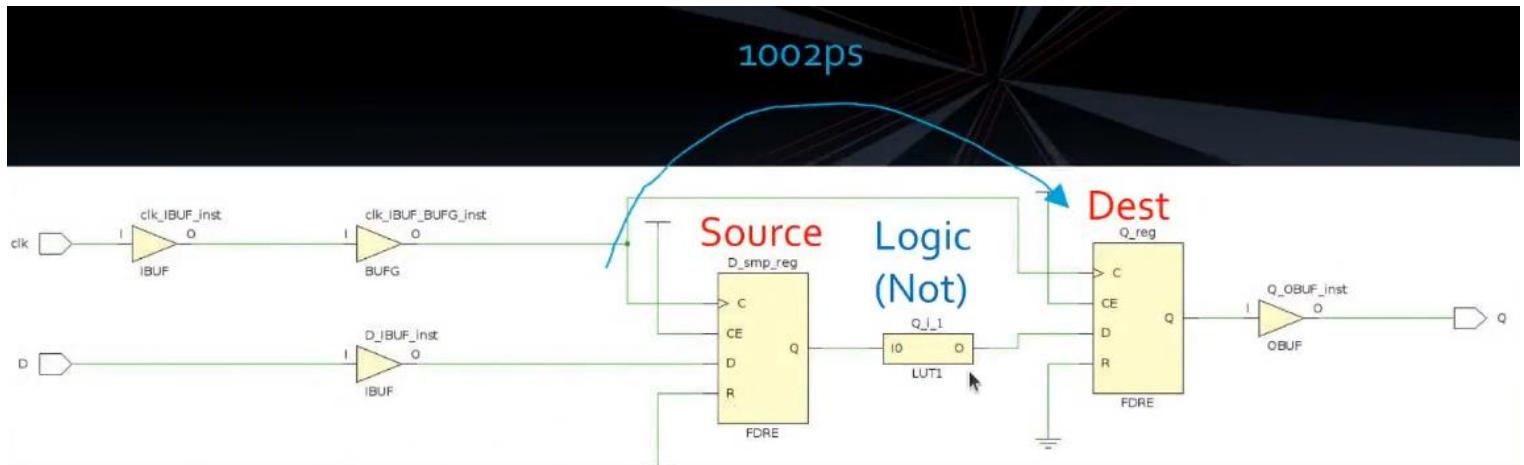
If we want to see it directly from the circuit, we have to highlight the specific path and then press **F4**. By double clicking we go to the previous node.

If we change the clocking, *etc* we have to re-run the implementation to verify again all the constraints. If a timing constraint is violated, an error is generated

Tcl Console	Messages	Log	Reports	Design Runs	Timing	x	Power	Methodology	DRC	Package Pins	I/O Ports
Q Intra-Clock Paths - clk - Pulse Width											
General Information					Check Type	Corner	Lib Pin	Reference Pin	Required	Actual	Slack
Timer Settings					Min Period	n/a	BUFG/l	n/a	2.155	1.600	-0.555
Design Timing Summary					Low Pulse Width	Slow	FDRE/C	n/a	0.500	0.800	0.300
Clock Summary (1)					Low Pulse Width	Fast	FDRE/C	n/a	0.500	0.800	0.300
Check Timing (2)					Low Pulse Width	Slow	FDRE/C	n/a	0.500	0.800	0.300
Intra-Clock Paths					Low Pulse Width	Fast	FDRE/C	n/a	0.500	0.800	0.300
clk					High Pulse Width	Slow	FDRE/C	n/a	0.500	0.800	0.300
Setup 0.711 ns (1)					High Pulse Width	Fast	FDRE/C	n/a	0.500	0.800	0.300
Hold 0.198 ns (1)					High Pulse Width	Slow	FDRE/C	n/a	0.500	0.800	0.300
Pulse Width -0.555 ns (11)					High Pulse Width	Fast	FDRE/C	n/a	0.500	0.800	0.300
Inter-Clock Paths					Min Period	n/a	FDRE/C	n/a	1.000	1.600	0.600
Other Path Groups					Min Period	n/a	FDRE/C	n/a	1.000	1.600	0.600
User Ignored Paths											
Unconstrained Paths											

usually when we have errors, it is enough to slow down the clock.

What happens if we have a setup violation? We have to slow down the clock. If we do not want to do so, we have to split the logic with pipeline. By doing so we reduce this *propagation delay*

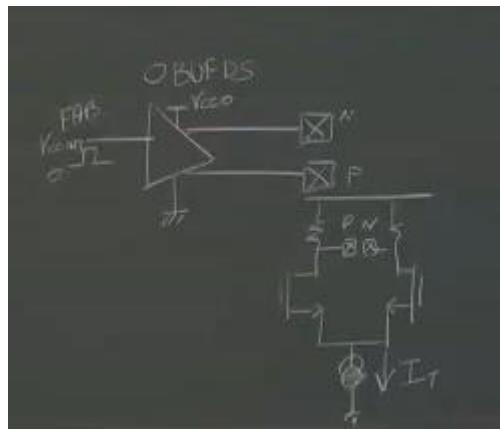


(we can't apply it in this case because the *not* cannot be split)

06_Vivado_Power_Reports

Vivado has a tool that has the possibility to study *static* and *dynamic power*

- static power: Vivado computes the static power considering also all the leakage currents that can flow in our circuit and considers also the fully differential peripherals that consume static power



- dynamic power: usual dynamic power so $\sum fC(V_{OH} - V_{OL})Vdd$ where f is the switching activity of the circuit

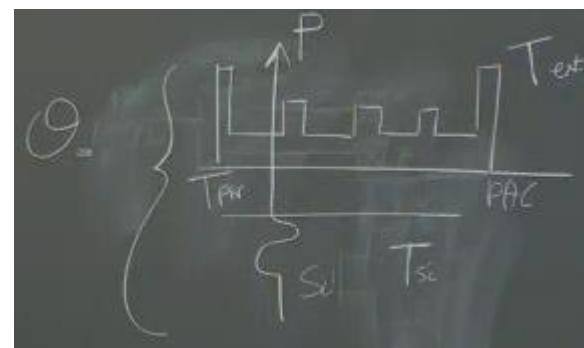
DESD 2024/2025 – VIVADO notes – Alessandro Lazzaroni – alessandro1.lazzaroni@mail.polimi.it for feedback – [here](#) if you want to offer a me a beer
 The total power is the sum, so $P = P_S + P_D$. In the report, the power consumption is also split between signals, so we know the contribution for each signal, furthermore this power data is also used to estimate the working temperature of the FPGA, Vivado does so by using the formula

$$T = P * \theta$$

where

- T [$^{\circ}\text{C}$] is the operating temperature of the FPGA
- P [W] is the power consumed
- θ [$\frac{^{\circ}\text{C}}{\text{W}}$] is the thermal resistance of the circuit

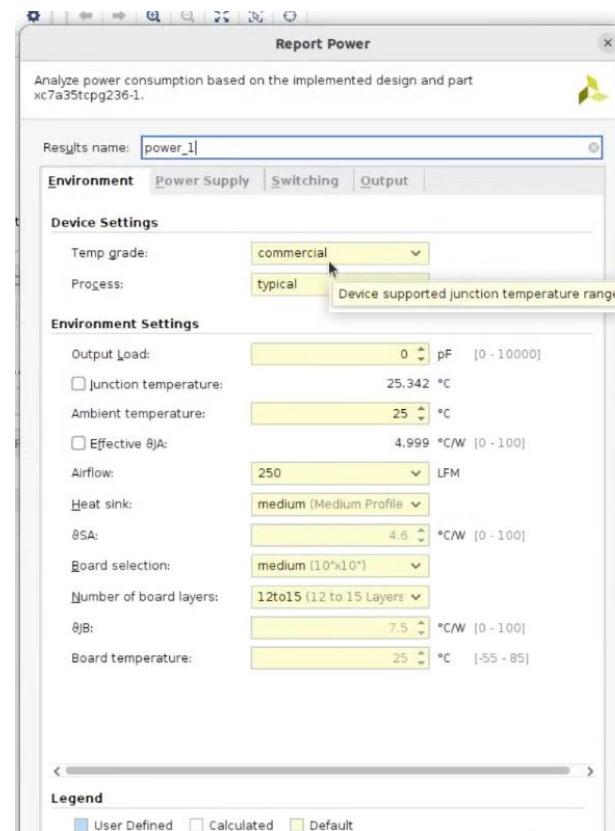
in reality the computation is a bit more complex because we have these parameters

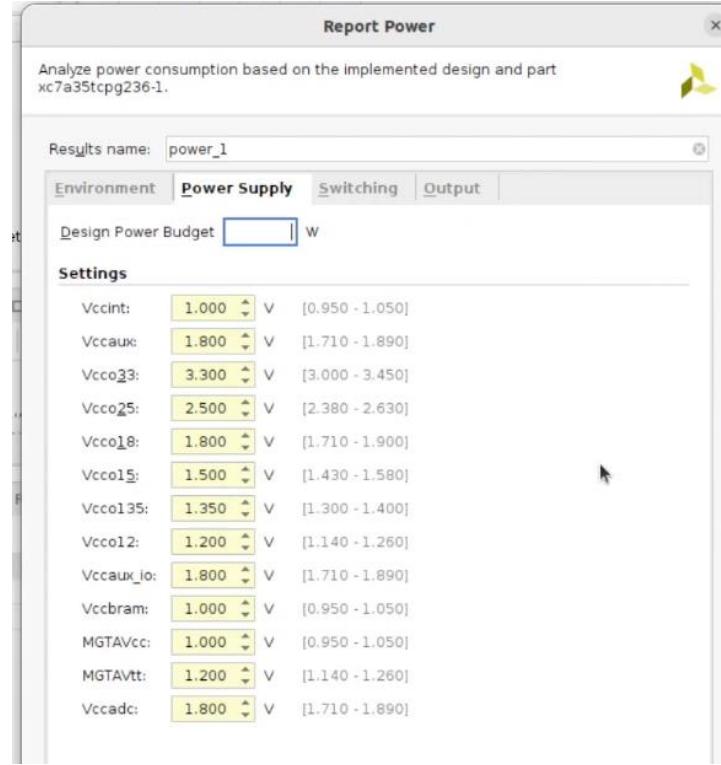


We start by using the implemented design and we move to the *Report Power* function

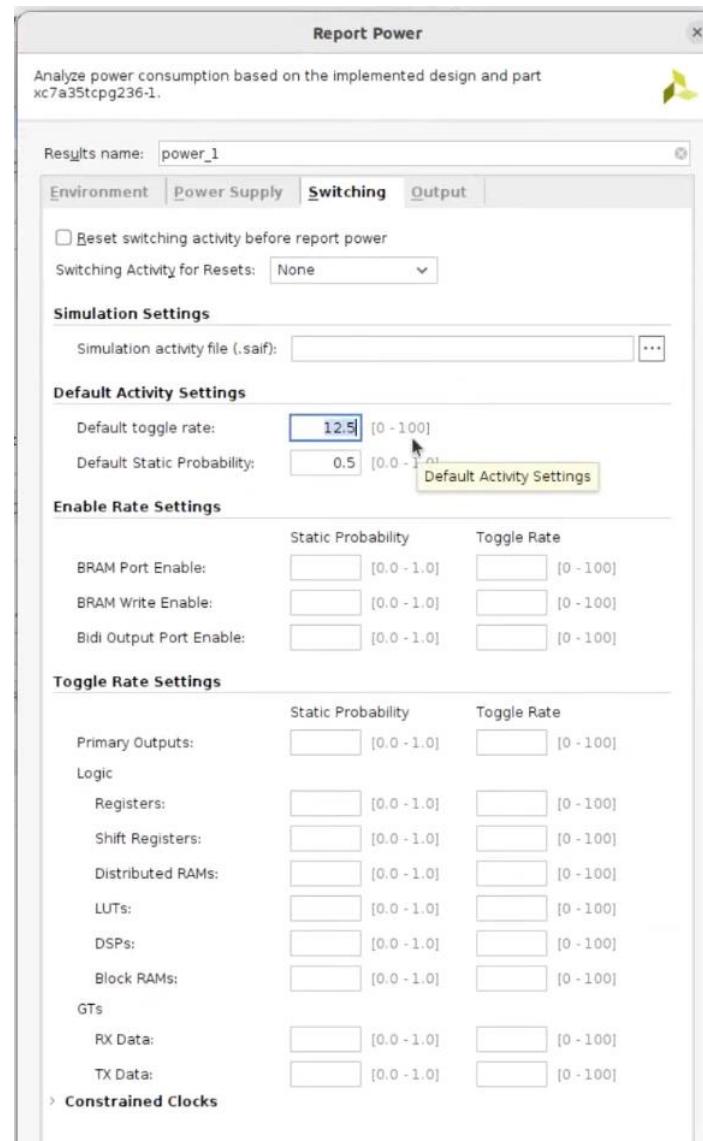


we can see these parameters



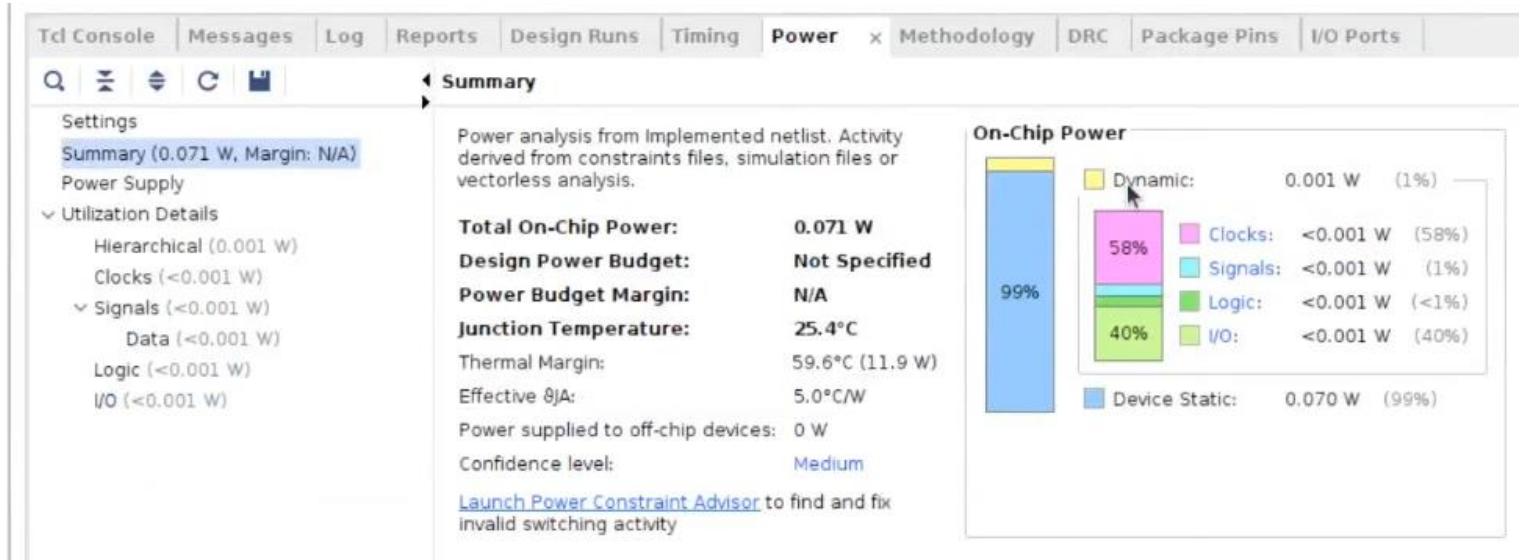


if we want a power budget, we can limit the power and if the total power is more than what we set we have an alert.



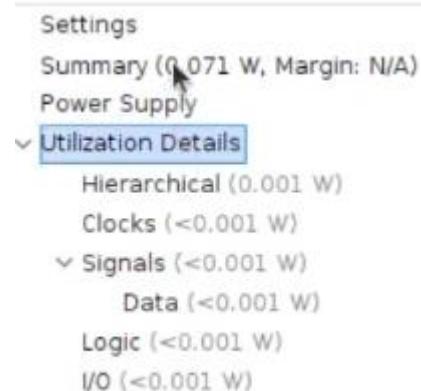
here we have the settings for the switching activity of the device under the voice *Default toggle rate* and we can set the parameters of the power analysis

Here we have the power report

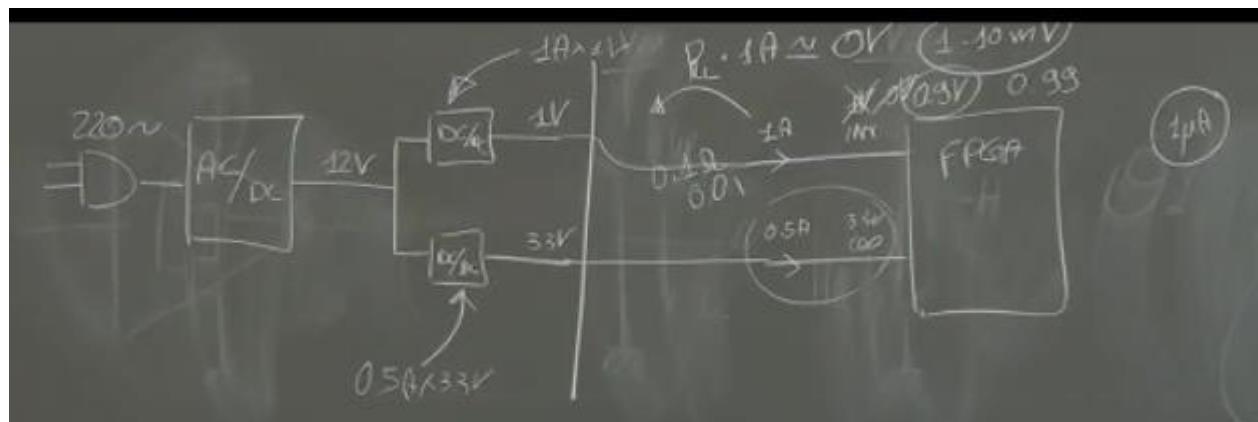


we do realize that most of the power dissipated is static power by devices that we are not using.

If we go in the *hierarchical* voice, we can see the power used by each single VHDL module. In clocks we have the power consumption of each clock of the FPGA.



we can also see the specific power absorbed by each power line, in particular we know that



A handwritten formula on a chalkboard:

$$\Delta T = \theta \cdot (I^2 R_c)$$

Below the formula, there is a note: $\theta \text{ Th } \text{Np} = \frac{I^2 \cdot R_c}{P_L}$

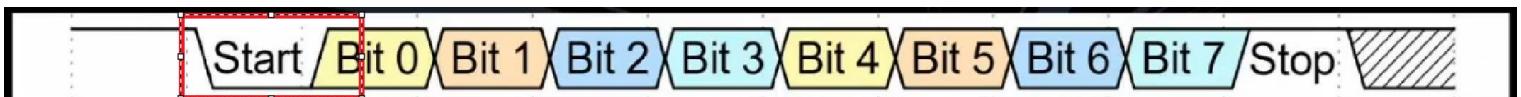
Below the note, there is a diagram showing a resistor R_c with current I flowing through it, and a power dissipation P_L .

each connector or line has a maximum operating temperature and power dissipation, here we can see the estimate.

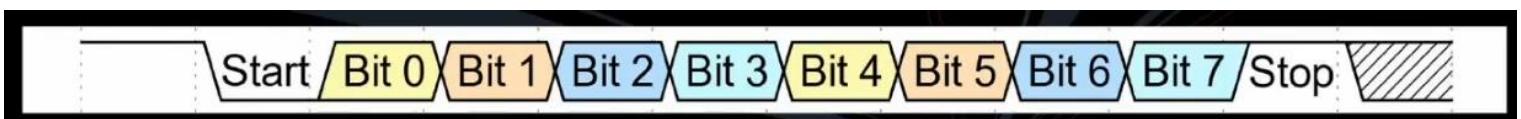
UART – PC FPGA Communication

UART stands for Universally Asynchronous protocol. We study it because we implement an application in our FPGA, then we need a way to communicate from the board to the outside world. One of the most known standards is in fact UART. It is a really standard protocol and are very cheap in terms of connections and area.

Since it is *asynchronous* it does not use a dedicated clock line, we have two lines, TX and RX for the duplex version of the protocol and it's a serial protocol, so the bits of our data will be transmitted one by one. The bus is an electrical line, we have an high line and when something has to be transmitted the line is lowered



we have eight bits of information and a stop bit, that will rise back the line.



so the *start* and *stop* bits are used.

There are various implementations of the protocol

- based on the *baud rate* so is the total number of bits that can be transmitted per second, we will use the 115200 so we transmit this number of bits each second (taking into account the start and stop pin)
- number of databits
- parity: it may be implemented like the number of ones or of zeros in the packet are counted and then at the end of the packet the number of bits counted is attached to the data packet
- number of stop bits

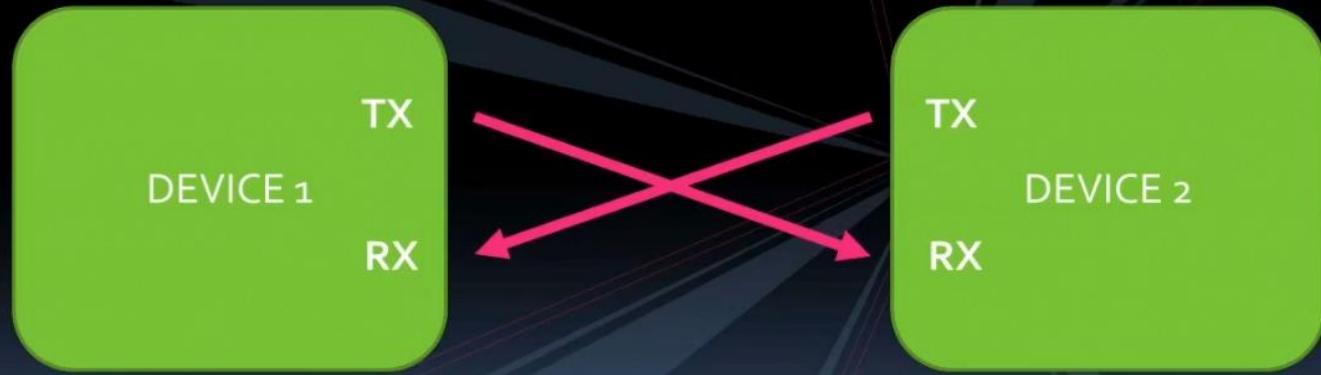
For now, we will just use the most common version of the asynchronous serial communication:

- 115200 bits per second
- 8 data bits per packet
- no parity
- 1 stop bit

often called:

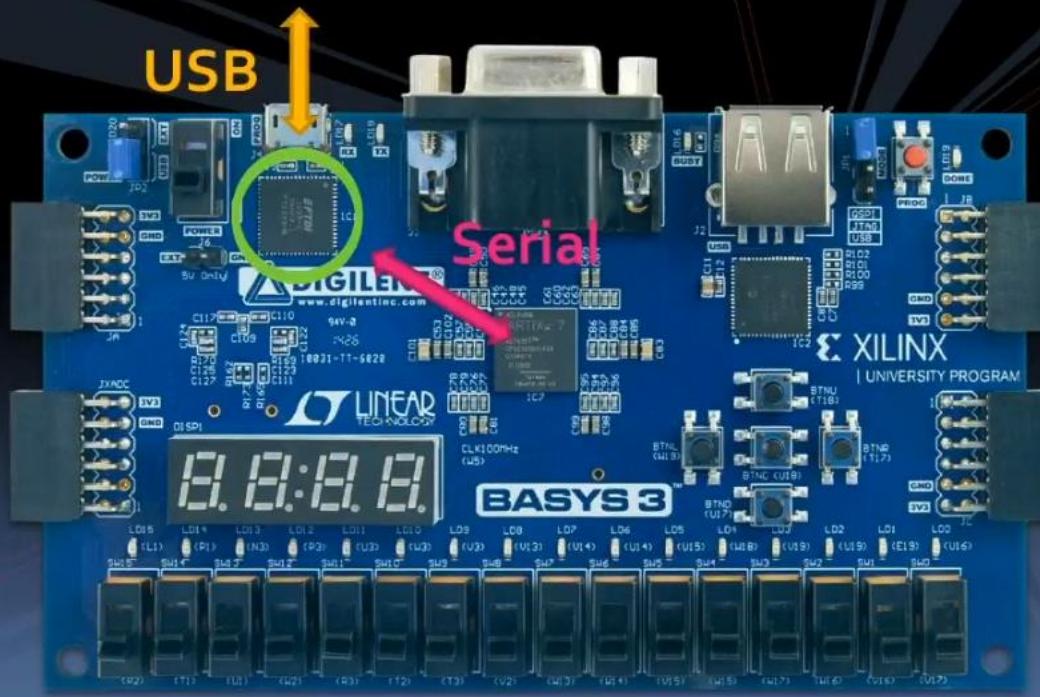
115200 8N1

The two most important signals are the TX and RX signals (sometimes called TXD and RXD), which transmit the data.



The TX of one device must be connected with the RX of the other one!

On the Digilent Basys 3 board there is a USB to serial converter (FTDI FT2232H):

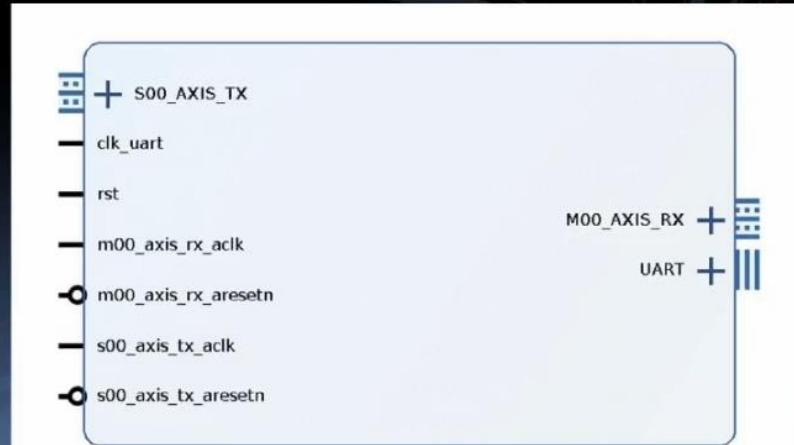


so this chip implements both the UART and the JTAG programming communication.

oss: the RX and TX are related only to the USB communication

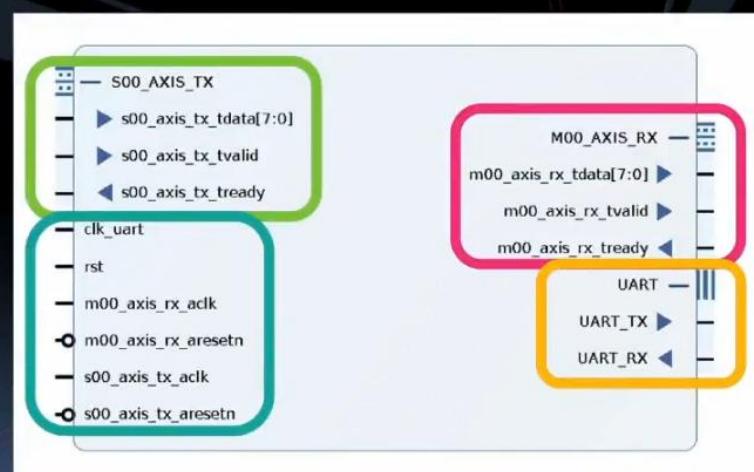
IP core that converts from UART to AXIS

To “talk” the asynchronous serial protocol we will use a UART (Universal Asynchronous Receiver-Transmitter) IP-Core with an AXI4-Stream interface:



Slave AXI4-Stream port:
data coming into this port is converted and then transmitted through TX

Master AXI4-Stream port:
data received from the RX signal is converted and then transmitted from this port

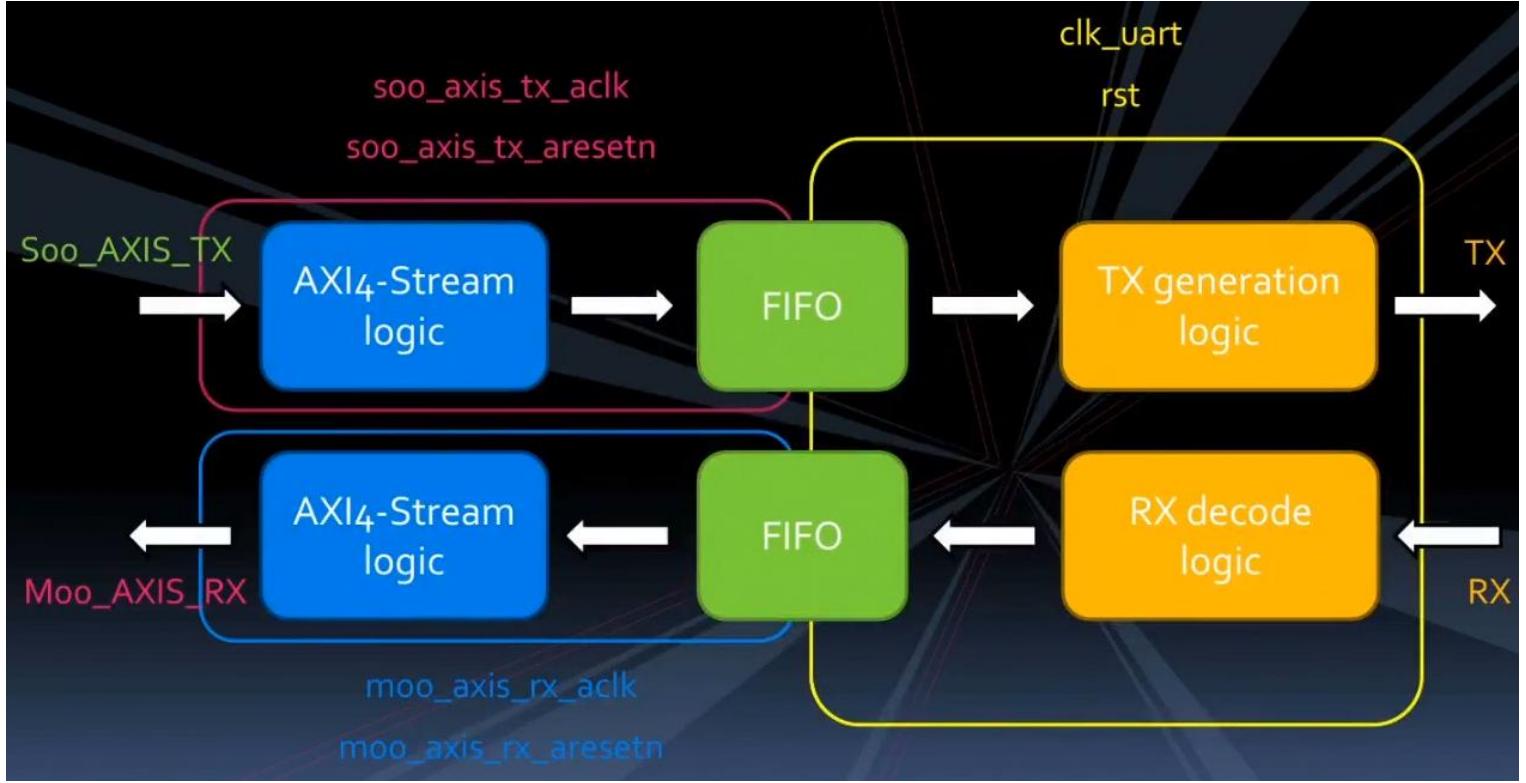


Clocks and reset signals

Asynchronous serial signals, to be connected to the FPGA I/O pins

Every interface here has a different clock so we have to take into account it and connect it to the correct pin

How is the IP core inside:



we have these three clock domains that are interfaced one to the other via asynchronous FIFOs.

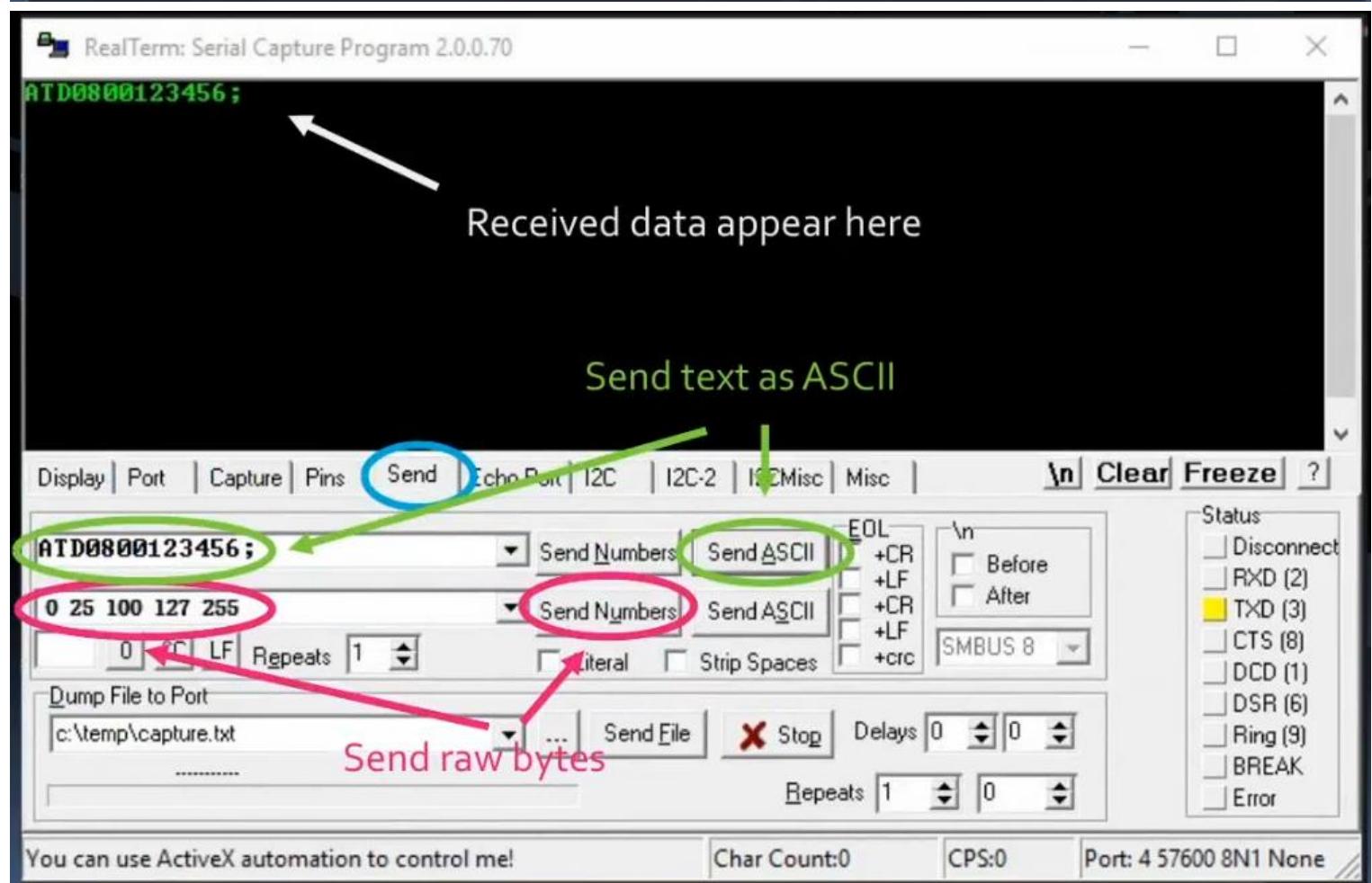
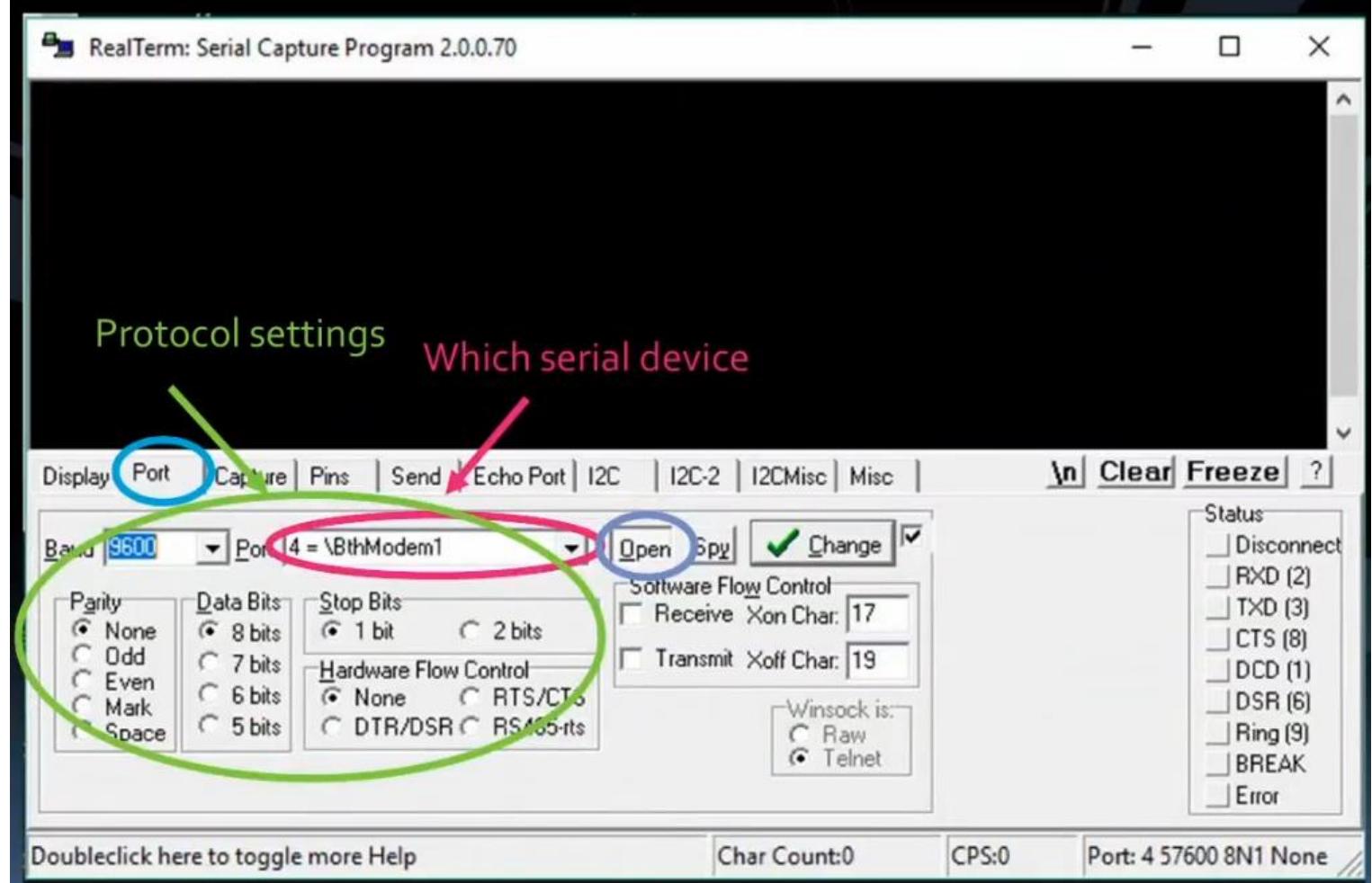
On the PC, USB-serial converters are seen as:

- Windows: **COMx** ports
- Linux: **/dev/ttyUSBx** devices
- Mac OS: **/dev/tty.usbserial-x** devices

To interact with a serial port, you need to write your own software or use a Serial terminal like:

- **RealTerm** (Windows)
- **Cutecom** (Linux)

these software are required to open a virtual serial port and implement it with our parameters.



A first way to check if the device is working well we usually have a loopback configuration, so we send and receive the same data in a loop.

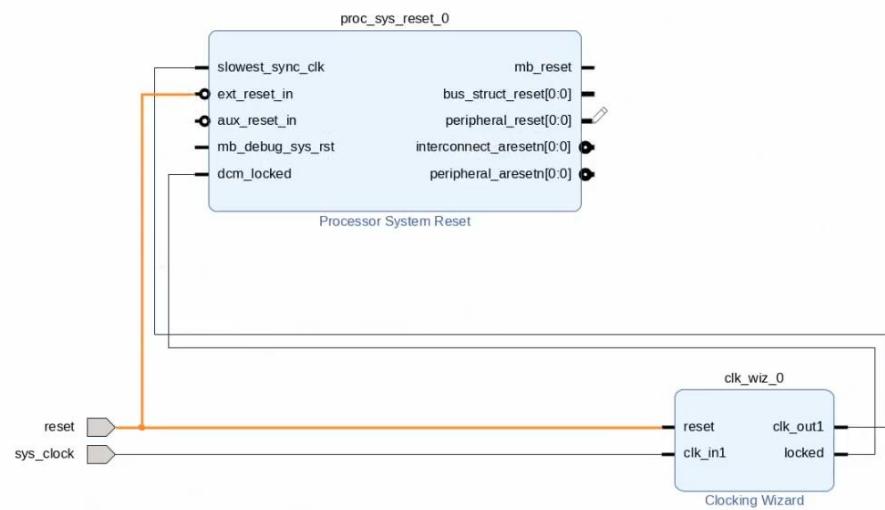
To use the given IP the first thing to do is *Settings > IP > Repository*

and add the IP

repository. Then we create the board design and we will see how the board file helps us.

And then the system clock is selected.

Another IP core that is needed is the *Processor System Reset* and helps synchronizing the reset interfaces of the board



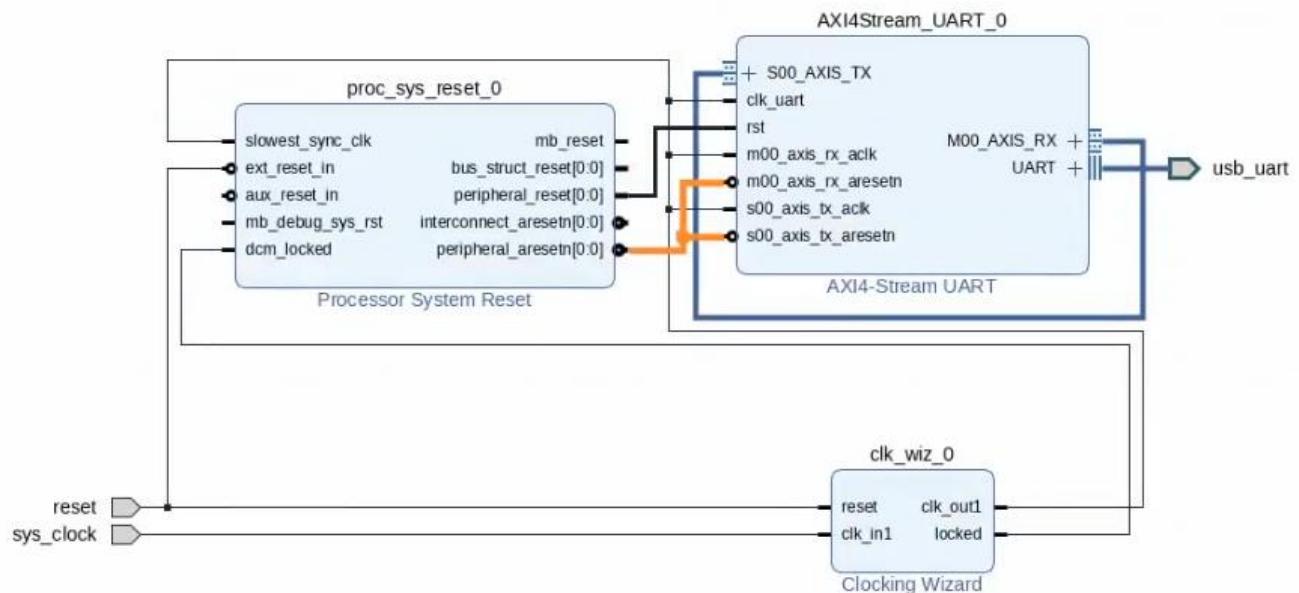
Then we must add the AXI4Stream IP core

How does the USART protocol works?

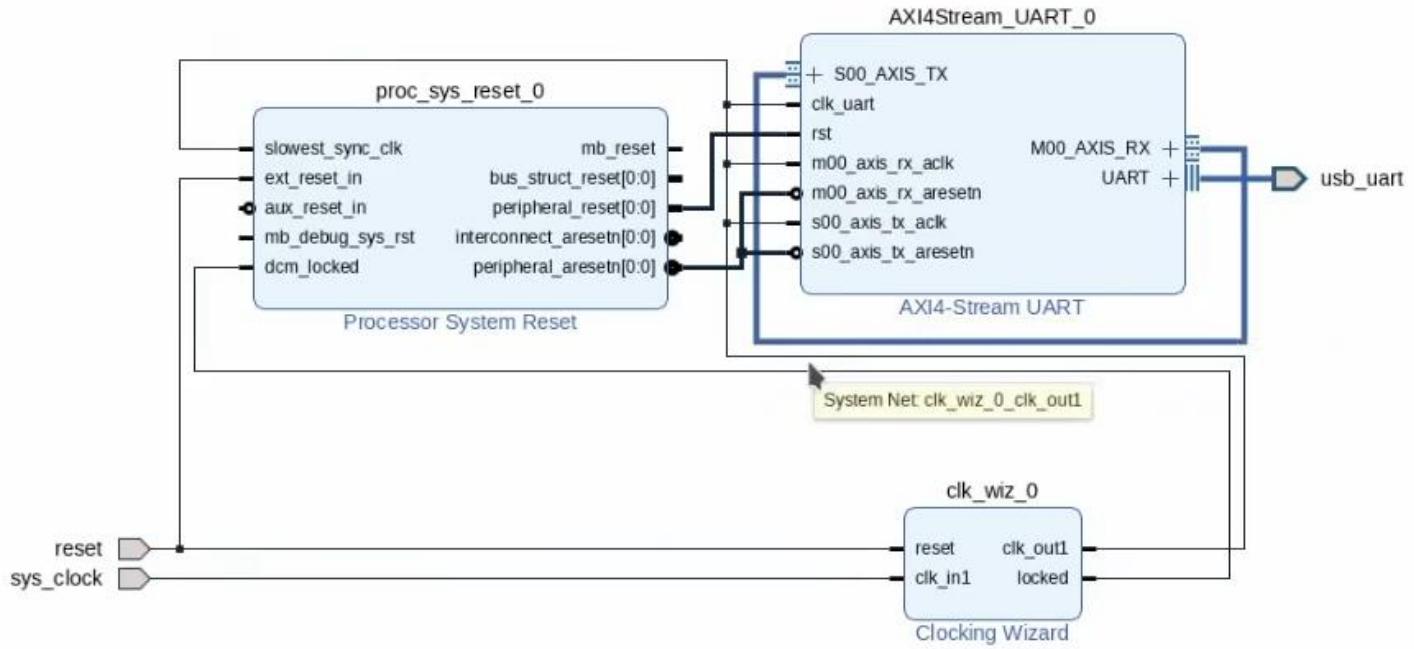
We need the board to sample the data because we need to understand where the transition is. By sampling it at higher rate than the baud rate we are able to know that value is a 1 or a 0



so in our IP core works for 100 MHz but it is not guaranteed that it works with other frequencies due this sampling constraint.

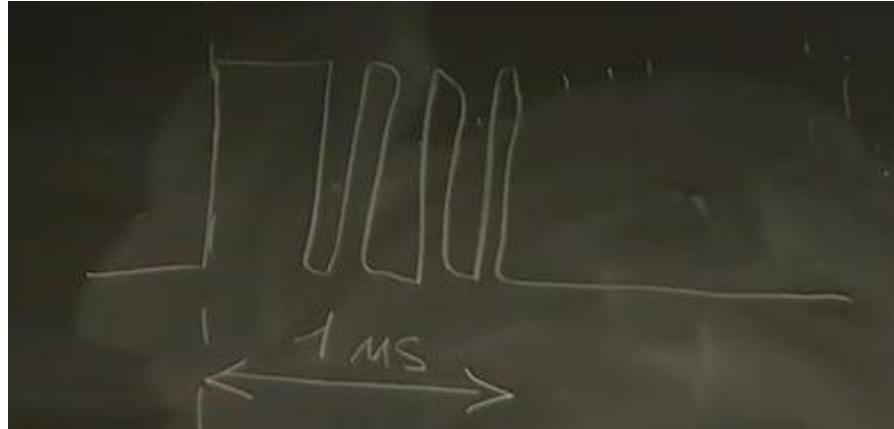


Then we have to create our HDL wrapper this will take our block design and bound it into our design into our HDL file.



Debouncer, AXI4stream_data_sampler and edge_detector

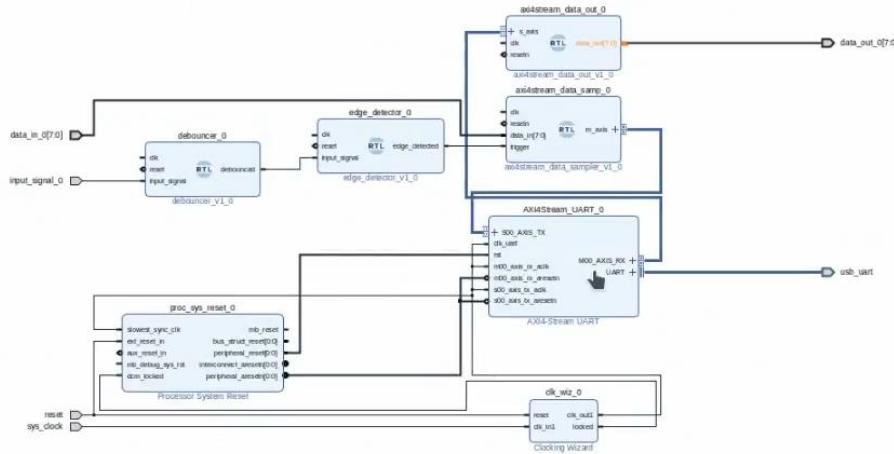
With the debouncer we have to clear the spurious bounces of the pressing of the button

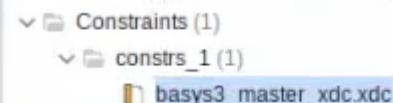


this is the input coming to the IP core and this is the signal that will come out



so it basically senses a rising edge and a falling edge to decouple the module.





file that contains all the I/O ports constraints
to use the constraint file we have to put the same as the one
the *xdc* file, so like →

sw[7:0]

BtnD

analysis of the .vhd files

Joystick + SPI

oss: we have to add an ILA for each clock domain that we have inside the design.

SPI

The Serial Peripheral Interface (SPI) is a very popular synchronous protocol for off-chip communication.

In its basic form it is composed by 4 signals:

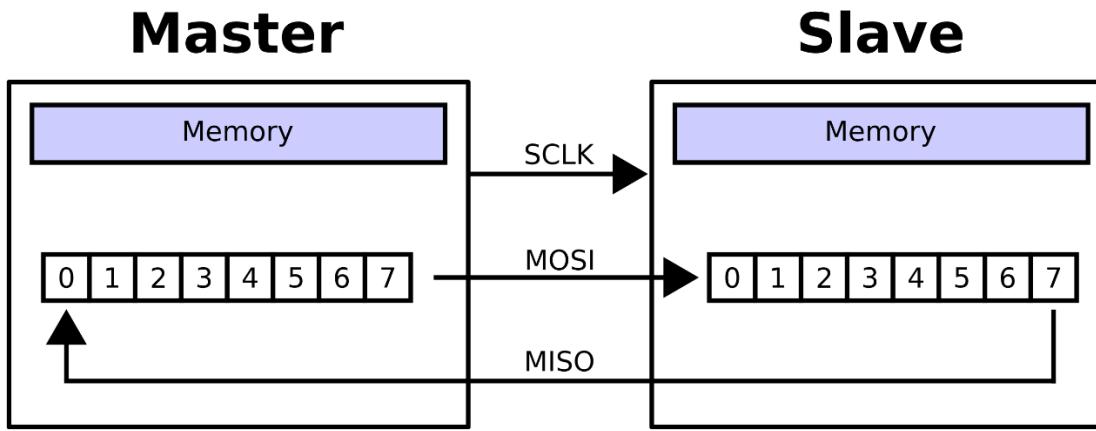
- Serial CLocK (SCLK)
- Master-Out Slave-In (MOSI)
- Master-In Slave-Out (MISO)
- Chip Select (CS)

You will find the details of its behavior in the README file. In short:

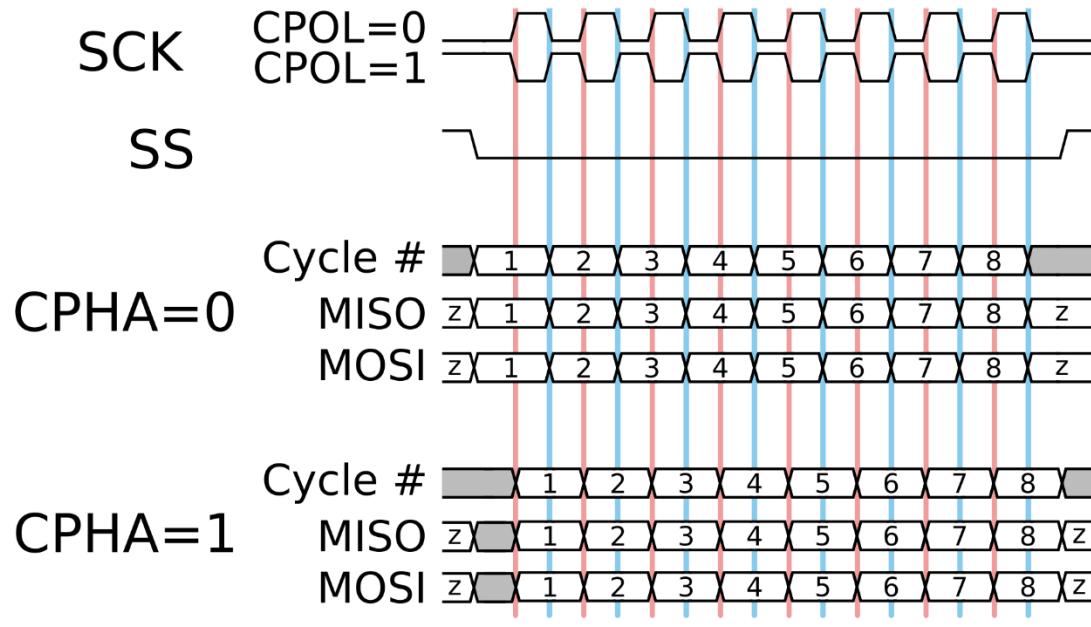
- Whatever you send to S_AXIS is sent to the SPI “slave”.
- Whatever is received from the SPI “slave” is sent to you through the M_AXIS interface.



How to use the protocol



we can imagine that our Master is the FPGA and the Slave is the joystick. The idea is that we have a shift register inside the two modules (it's a shift register in most of the cases) and being a serial protocol what happens is that when we want to write something to the slave we will push out of the master register one bit at a time, it is pushed clock cycle by clock cycle out. This is a timing diagram of a SPI communication



SPI timing diagram for both clock polarities and phases. Data bits output on blue lines if CPHA=0, or on red lines if CPHA=1, and sample on opposite-colored lines. Numbers identify data bits. Z indicates high impedance.

both polarities and phase of the signal are represented.

To understand how to communicate with the joystick, we have to read the Digilent documentation for the joystick.

Interfacing with the Pmod

The Pmod JSTK2 communicates with the host board via the SPI protocol with SPI Mode 0, CS active low, a 1 MHz clock, and each data byte organized MSb first. With the Pmod JSTK2, there are two types of data packet protocols: the standard data packet of 5 bytes and an extended data packet with 6 or 7 bytes in total. With the standard 5 byte protocol, users may use the old code from the Pmod JSTK without any syntax errors. The 5 byte packet structure is provided in the image below:

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
MOSI	COMMAND / 0	PARAM1 / DUMMY	PARAM2 / DUMMY	PARAM3 / DUMMY	PARAM4 / DUMMY
MISO	smpX (Low Byte)	smpX (High Byte)	smpY (Low Byte)	smpY (High Byte)	fsButtons

we will use the standard 5 bytes data packet.

In the first byte we have to put the command described in the documentation to perform the specific operation, then in the other 5 bytes we can put either parameters or dummy bytes

Why do we need dummy bytes?

Because maybe we want to send just a command not all the parameters and so we need some dummy data, some unuseful data to complete the data packet.

fsButtons	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	EXTPKT	0	0	0	0	0	TRIGGER	JOYSTICK

EXTPKT: Extended Packet Status Bit

1 = additional data corresponding to the command byte is available and may be retrieved after this byte

0 = standard response packet, no additional data follows this byte

TRIGGER: Trigger Button Status Bit

1 = trigger button is currently pressed

0 = trigger button is not being pressed

JOYSTICK: Joystick Center Button Status Bit

1 = joystick center button is currently pressed

0 = joystick center button is not being pressed

the only command that we will use it is

cmdSetLedRGB	(0x84)	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
		1	0	0	0	0	1	0	0

Parameters

PARAM1 – Red LED duty cycle

PARAM2 – Green LED duty cycle

PARAM3 – Blue LED duty cycle

PARAM4 – ignored

Set the duty cycles for the Red, Green, and Blue LEDs.

A very important part is that we have timing constraints to respect for the SPI

SPI Timing Requirements

The embedded PIC16F1618 requires certain SPI timing requirements in order for successful communication to occur. When the Chip Select line is brought low, users must wait at least 15 µS before sending the first byte of data. An interbyte delay of at least 10 µS is required when transferring multiple bytes. When the Chip Select line is brought high after the last byte has been transferred, at least 25 µS is required before users may bring the Chip Select line low again to initiate another communication session.