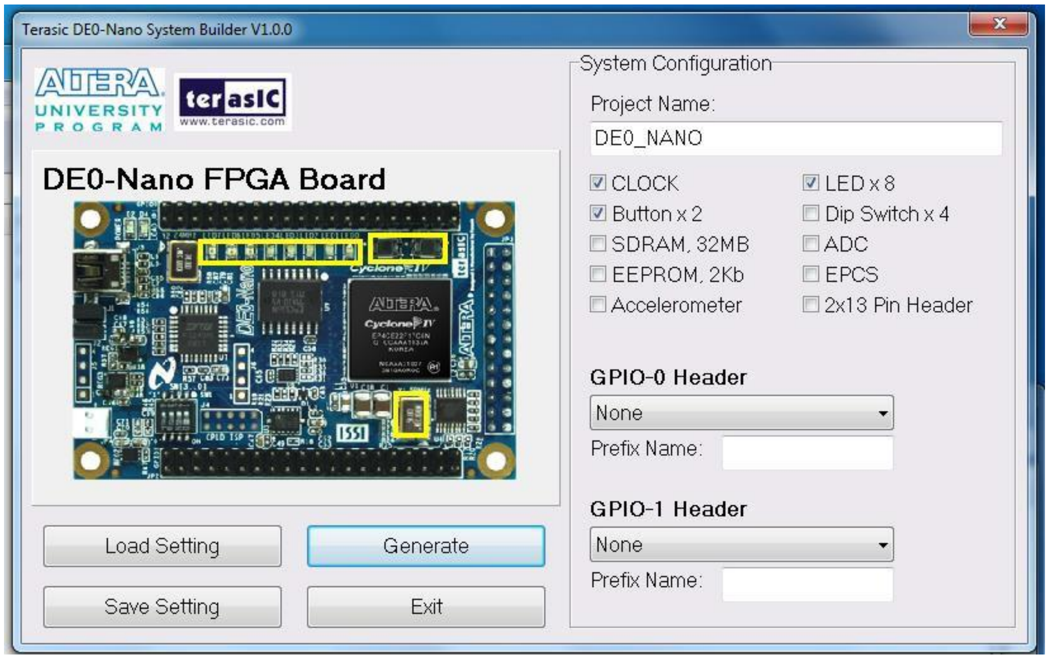
Programming the DE0-Nano Board, Part B

Implementing a design on the DE0 board

1. In order to implement your (System)Verilog design on your FPGA, you will need to map the name of the signals of the top-level entity to the physical pins of the FPGA. These are 'constraints' given to the synthesis software, those that we didn't have in the previous lab. However, since this is fairly repetitive and tedious work, terasIC has developed a tool to do this for us and to create the associated Quartus project. Launch the **DE0-Nano System Builder** program now (located on the CD-ROM in Tools/DE0_Nano_SystemBuilder).


Terasic DE0-Nano System Builder V1.0.0

System Configuration

Project Name: DE0_NANO

- [x] CLOCK
- [x] LED x 8
- [x] Button x 2
- [ ] Dip Switch x 4
- [ ] SDRAM, 32MB
- [ ] ADC
- [ ] EEPROM, 2Kb
- [ ] EPCS
- [ ] Accelerometer
- [ ] 2x13 Pin Header

GPIO-0 Header: None
Prefix Name:

GPIO-1 Header: None
Prefix Name:

2. For this example, and for your homework, you will not need many of the peripherals, so check only the CLOCK, LED and BUTTON options.

3. Press **Generate**, and save the Quartus Project File ".qpf" to a new directory, *with no spaces or special characters in the path*. This will create five files. Following the descriptions below, take time to fully understand their purpose:

   a. **DE0_NANO.htm**: this is an html file that shows an overview of the input/output connections of the top-level instance of the circuit, as generated by the tool (later manual changes are not back-annotated).

   b. **DE0_NANO.qpf**: the Quartus project file. This will keep track of the different files you use, and general project options.

c. **DE0_NANO.qsf**: this file contains notably the pin assignments, using two main commands:
   i. `set_location_assignment PIN_A15 -to LED[0]`: this tells the tool that the first bit of the LED signal in your top-level verilog module should be connected to the A15 pin of the FPGA. As it turns out, on the DE0-Nano card, this pin is itself connected to the LED: you can check that in the DE0-Nano user manual.

   ii. `set_instance_assignment –name IO_STANDARD "3.3-V LVTTL" -to LED[0]`: this tells the tool that the LED[0] signal should work at 3.3V with certain standard skew rates. Each pin can operate at different voltages, but we will only use 3.3V for this card.

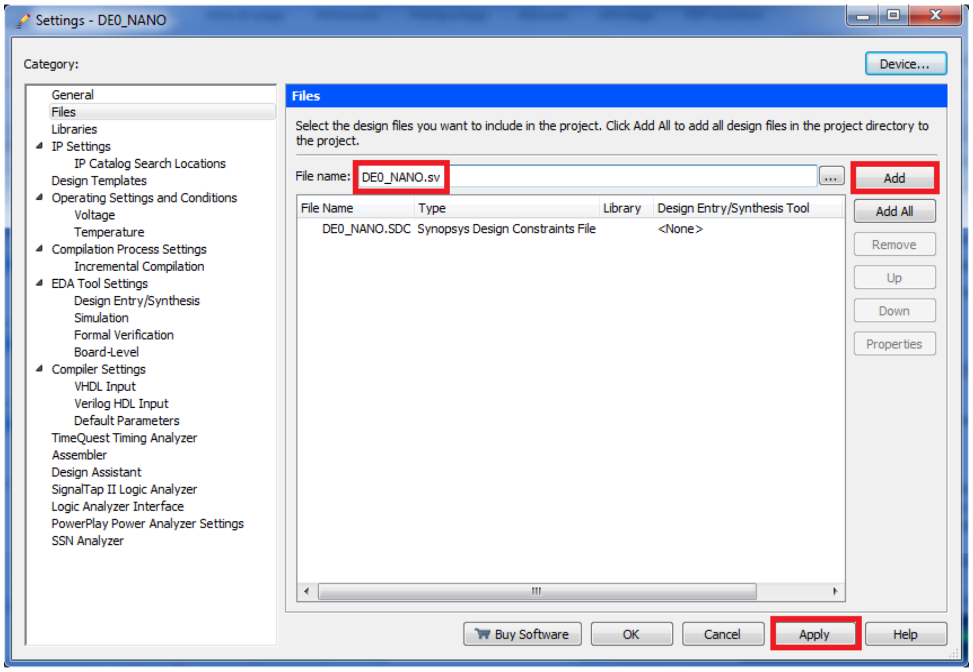
d. **DE0_NANO.sdc**: this file gives synthesis design constraints, in particular with regards to **timing**. The handling of this file is outside the scope of this course (see ELEC 2570 next year for more on this) but it is interesting for you to get a feel of its purpose already. For example, it contains the following commands:
   i. `create_clock -period 20 [get_ports CLOCK_50]`: this tells the tool that any ports matching the name "CLOCK_50" (there is only one in this design) will be the source of a clock with a period of 20ns. The tool will now know this clock period and will be able to correctly compute the setup timing slack of the different paths in your design. This `period` corresponds to the physical crystal oscillator on the board, so changing the period here will not change the operating frequency (instead, you should use a PLL).

   ii. `derive_pll_clocks`: tells the tool that any output of the PLLs should be assumed to be a clock.

e. **DE0_NANO.v**: an automatically generated Verilog top-level module for your design.

4. Replace the DE0_NANO.v file with the one we have given you (DE0_NANO.sv). Look at the source of that file, and try to understand what it does.

5. Now, double-click on the DE0_NANO.qpf file to launch Quartus. If this does not work on your setup, open Quartus manually, and open that file as a project. Do not forget to add the new DE0_NANO.sv file to the Quartus project: just right click on the 'Files' folder in the 'Project Navigator' sub-window. When providing the path to the file, do not forget to press the 'Add' button before closing.



Files

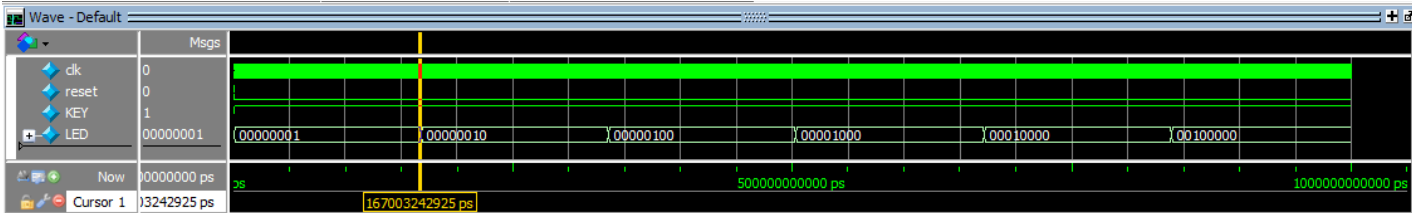
Select the design files you want to include in the project. Click Add All to add all design files in the project directory to the project.

File name: DE0_NANO.sv

File Name	Type	Library	Design Entry/Synthesis Tool
DE0_NANO.SDC	Synopsys Design Constraints File		<None>

6. Similarly to what you have done in the previous lab, set ModelSim as the simulator with SystemVerilog as the used HDL (`Assignments > Settings > EDA Tool Settings`. Then, add and configure the testbench (`Assignments > Settings > EDA Tool Settings > Simulation`). **Be careful with the names you provide**, they must match the corresponding module/instantiation names of your design! Also, today, we will need more simulation time (you can set at least 200ms and maximum 1s and use “ms” as the default time scale for the ModelSim wave).

7. Compile the design and carry out the **behavioral (RTL)** simulation as performed in the previous labs (as we run rather long simulation here, it can take up to a few minutes to get the full wave). Observe for the behavioral simulation the resulting waveform, and notice how the LED active bit is shifted to the left after 167ms (this long time is used to ensure that the switching is visible to the eye when it is implemented using the DE0-Nano board’s LEDs).



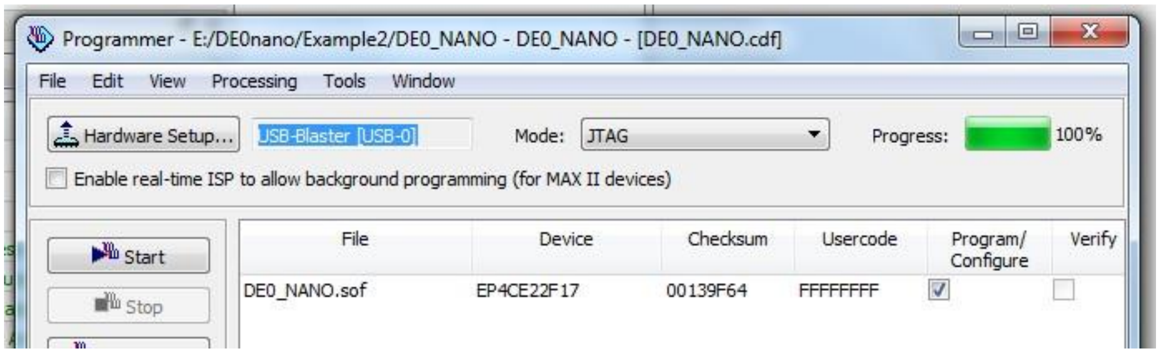
	Msgs	
clk	0	
reset	0	
KEY	1	
LED	00000001	00000001, 00000010, 00000100, 00001000, 00010000, 00100000
Now	0000000 ps	
Cursor 1	3242925 ps	

Notice that we ask you to observe this shift time of 167ms only for the behavioral simulation. Why?

Programming the DE0-Nano

Now that all the simulations are done, and things are looking good, let's put it on the board! Simply launch the programmer via the menu, the option in the synthesis window, or the toolbar button:

It should be automatically filled, but otherwise select the DE0_NANO.sof file (found in the root of your project directory) and program the device via JTAG by connecting the DE0-Nano board to the computer using the mini-USB cable. This requires a driver called "USB Blaster". If it is not detected, install the driver following the provided tutorial on Moodle ('Tutorial Install USB-Blaster Driver', written for the 15.0 Quartus version so please replace mentally every "15.0" occurrence by "18.0").


Programmer - E:/DE0nano/Example2/DE0_NANO - DE0_NANO - [DE0_NANO.cdf]
File Edit View Processing Tools Window
Hardware Setup...
Start
Stop

USB-Blaster [USB-0] Mode: JTAG Progress: 100%

Enable real-time ISP to allow background programming (for MAX II devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify
DE0_NANO.sof	EP4CE22F17	00139F64	FFFFFFFF	✓	☐

Once everything is set as above, you can press the "Start" button to program the FPGA. You can now play a bit with it (e.g. press the two buttons), and look delighted. Electricians are known for being extravagantly enthusiast over a single blinking LED, so try to uphold this tradition!

For information purpose, please note that programming the board using the ".sof" file is volatile and therefore lost once the device is powered off. You will learn later how to program the flash of the DE0-Nano with a ".jic" file so that your circuit is loaded each time the FPGA is powered on.

Signal Tap in-circuit logic analyzer

Now, let's play with the in-circuit logic analyzer. As you maybe noticed, we did not run a gate level (or post-layout) simulation this time, unlike we learned in lab 2. The reason for this is simple: for the time scale we use, it is prohibitively too slow, and also has to start at `t=0`. Signal Tap allows you to observe the signal in almost real-time, and specify advanced triggers to observe the physical behavior of your circuit.

Now, the key point: **Signal Tap waves are not a simulation, but a measurement!** While ModelSim allowed you to simulate your design, Signal Tap is a tool that inserts dedicated logic blocs into your FPGA design so that to carry the measurements. Consequently, when you add or modify a Signal Tap entity on your design, it needs to be re-compiled.

1. If your using Quartus without a license (i.e. "Lite Edition"), you have to enable the "Talkback" before being able to use Signal Tap. To do, go to `Tools > Options > Internet Connectivity > Talkback Options` and then check the box `Enable sending Talkback data to Altera` (this may not be the case anymore in Quartus 18.0, in this case, ignore this point).

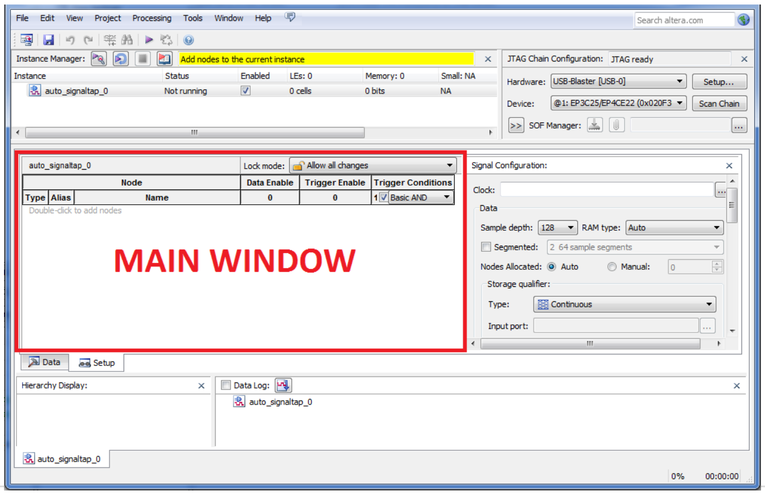
2. Now, open Signal Tap using `Tools > Signal Tap Logic Analyzer`.

3. Here you will need to specify three things:
   - The signals to observe

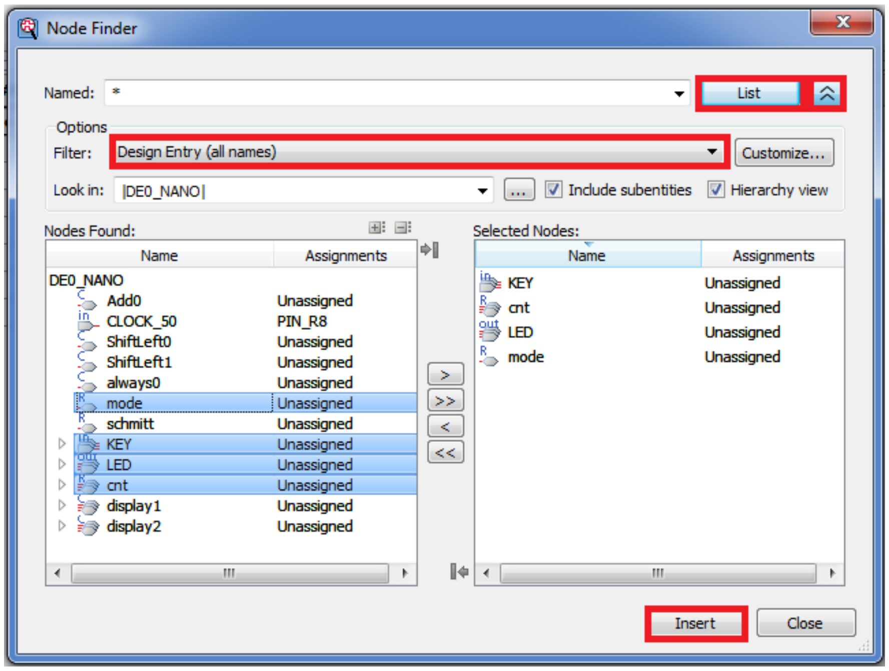
     Important note: some of the signals you wrote in your (System)Verilog code can be optimized out during the synthesis of the circuit by Quartus. As those nodes are simplified, they will not exist anymore as is in the implemented circuit and you won't be able to probe them. They will be marked in **red** when trying to select them.

   - The sampling clock
   - The trigger conditions

4. Double-click in the main window to add signals. There are several name types available (click on the double arrow next to the 'List' button to see them), we will be using "Design Entry" for now.



This searches for names as written in your (System)Verilog code. Use the 'List' button to find all signals matching your filter, and select those that interest you.



Named: *

Options

Filter: Design Entry (all names)

Look in: |DE0_NANO|

☑ Include subentities ☑ Hierarchy view

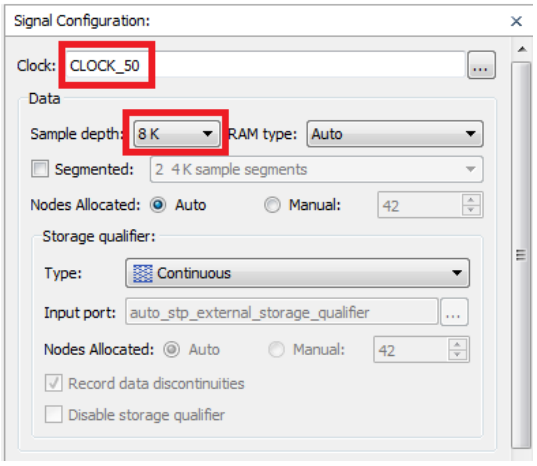
Nodes Found:

Name	Assignments
DE0_NANO	
Add0	Unassigned
CLOCK_50	PIN_R8
ShiftLeft0	Unassigned
ShiftLeft1	Unassigned
always0	Unassigned
mode	Unassigned
schmitt	Unassigned
KEY	Unassigned
LED	Unassigned
cnt	Unassigned
display1	Unassigned
display2	Unassigned

Selected Nodes:

Name	Assignments
KEY	Unassigned
cnt	Unassigned
LED	Unassigned
mode	Unassigned

5. Add a sampling clock (we will simply use CLOCK_50), and set the number of samples to store:



Signal Configuration:

Clock: CLOCK_50

Data

Sample depth: 8 K RAM type: Auto

☐ Segmented: 2 4 K sample segments

Nodes Allocated: ● Auto ○ Manual: 42

Storage qualifier:

Type: Continuous

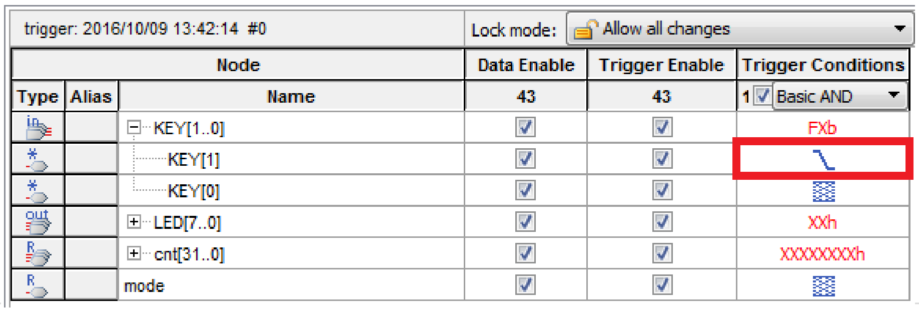
Input port: auto_stp_external_storage_qualifier

Nodes Allocated: ● Auto ○ Manual: 42

☑ Record data discontinuities

☐ Disable storage qualifier

6. Lastly, set triggers for your data. In this example, you can for example choose the **F**alling edge of

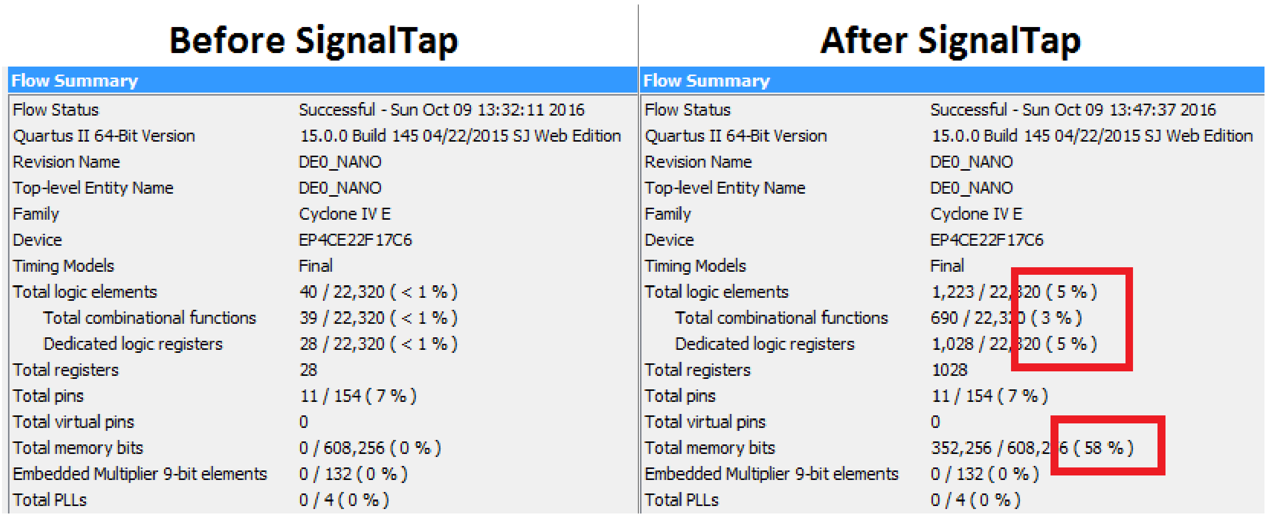


trigger: 2016/10/09 13:42:14 #0 Lock mode: Allow all changes

Type	Alias	Name	Data Enable	Trigger Enable	Trigger Conditions
			43	43	1 ☑ Basic AND
in		KEY[1..0]	☑	☑	FXb
		KEY[1]	☑	☑	⤵
		KEY[0]	☑	☑	▨
out		LED[7..0]	☑	☑	XXh
R		cnt[31..0]	☑	☑	XXXXXXXXh
R		mode	☑	☑	▨

KEY[1] by selecting KEY[1] and right click on trigger conditions and choose Falling edge, so probing will start only once that button is pressed.

7. Save ("Ctrl+S") the ".stp" file in your project root directory. Right afterwards, a pop-up window should appear, asking you if you want to add the ".stp" file to project: click on "Yes", as this file has to be part of the design in order for the compiler to insert the needed logic analyzer block. Then launch a full compilation. Take a look at the "Flow Summary" on Quartus, you should get the following:



Before SignalTap

Flow Summary	
Flow Status	Successful - Sun Oct 09 13:32:11 2016
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	DE0_NANO
Top-level Entity Name	DE0_NANO
Family	Cyclone IV E
Device	EP4CE22F17C6
Timing Models	Final
Total logic elements	40 / 22,320 (< 1 %)
Total combinational functions	39 / 22,320 (< 1 %)
Dedicated logic registers	28 / 22,320 (< 1 %)
Total registers	28
Total pins	11 / 154 (7 %)
Total virtual pins	0
Total memory bits	0 / 608,256 (0 %)
Embedded Multiplier 9-bit elements	0 / 132 (0 %)
Total PLLs	0 / 4 (0 %)

After SignalTap

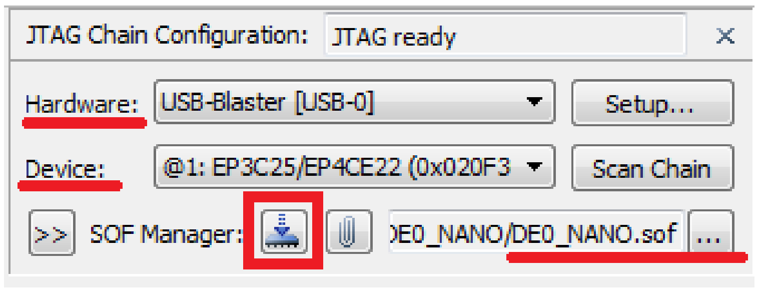
Flow Summary	
Flow Status	Successful - Sun Oct 09 13:47:37 2016
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	DE0_NANO
Top-level Entity Name	DE0_NANO
Family	Cyclone IV E
Device	EP4CE22F17C6
Timing Models	Final
Total logic elements	1,223 / 22,320 (5 %)
Total combinational functions	690 / 22,320 (3 %)
Dedicated logic registers	1,028 / 22,320 (5 %)
Total registers	1028
Total pins	11 / 154 (7 %)
Total virtual pins	0
Total memory bits	352,256 / 608,256 (58 %)
Embedded Multiplier 9-bit elements	0 / 132 (0 %)
Total PLLs	0 / 4 (0 %)

You see that after the addition of Signal Tap to our design, the resource utilization increases. There are two things to note here:

- For the '**Total logic elements**', the increase comes from the resources that Signal Tap adds in order to probe the value of the different nodes of your design.
- For the '**Total memory bits**', the increase depends on the 'Sample depth' that you have specified in Signal Tap at point 4. Memory resources are required in order to store the samples taken at each cycle of the sampling clock defined at point 4. The higher the specified sample depth and the higher the number of probed signals, the higher the memory resources utilized by Signal Tap: be careful so as not to exceed the maximum memory capacity of your FPGA!

It is **very important** to keep these considerations in mind when using Signal Tap.

8. Program the chip via the Signal Tap in-built programmer:



JTAG Chain Configuration: JTAG ready

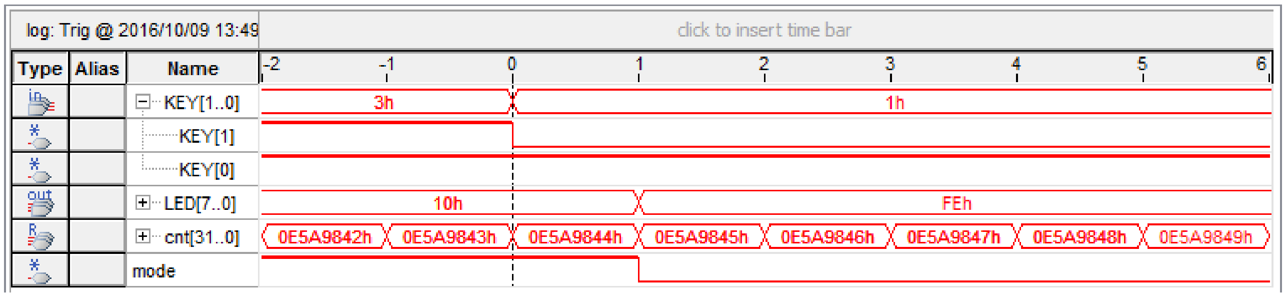
Hardware: USB-Blaster [USB-0]

Device: @1: EP3C25/EP4CE22 (0x020F3

SOF Manager: DE0_NANO/DE0_NANO.sof

9. In Signal Tap, click "Run Analysis" , then press the button KEY[1] to trigger the waveform.

10. Eureka (zoom on the waveform (left click on the waves) to observe the same as below):



Now, why is it so complicated to observe a transition on the LEDs? Remember your behavioral simulation: a transition on the LEDs occurs every 167ms. In order to observe signals at a longer time scale on Signal Tap, there are three things you can do:

- Increase the sample depth:

  Let's understand what you can do with this 8K sample depth here (i.e. 8K = 8*1024 = 8192 samples), where each sample monitors 2-bit `KEY`, 8-bit `LED` and 32-bit `cnt` and 1-bit `mode`, so 43 bits of data in total. At each cycle of CLOCK_50 (i.e. every 20ns), you store a sample in memory. Therefore, an 8K sample depth allows you to monitor a time equivalent to 8192*20ns, which is about 164 microseconds. We are 3 orders of magnitude below the 167 milliseconds necessary to observe a LED transition! You won't be able to increase the sample depth by three orders of magnitude though, as the FPGA embedded memory resources are limited (see point 6).

- Change the triggering conditions:

  A quick and efficient solution is to modify the triggering conditions to monitor any change in the value of LED[7:0]: try it yourself!

- Use a slower clock:

  Here, you can use a sampling clock with a frequency lower than CLOCK_50 (for example thanks to a dedicated PLL, see the next section of this tutorial), but keep in mind that you will also be able to observe much less transitions in your signals. If you use a slower clock, it is not a good idea to monitor signals defined in the CLOCK_50 domain, such as the counter here.

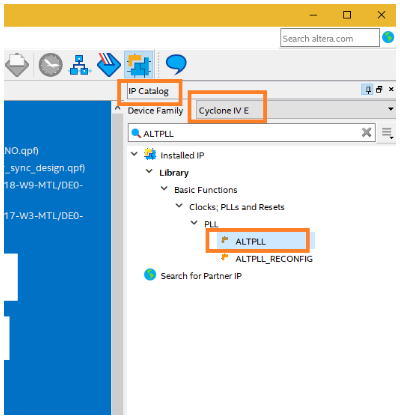
More things to check out!

Well, wasn't that all very exciting? But wait – there's more! As you may know, an FPGA is more than just a series of configurable logic units, there are also a number of built-in hard blocks (specific hardware available in the FPGA, the PLL is one of them) and soft blocks (modules previously designed in HDL for you, you can implement them in your design, you just may not have access to the original HDL code) can be used, they perform specific functions.

In your (System)Verilog code, this is done in the same way as instantiating another (System)Verilog module. For the hard blocks, the module is automatically recognized by Quartus as being a hardware component (this is called a "primitive"). However, many of such primitives have rather complicated options which require detailed knowledge of the datasheet in order to instantiate properly. Luckily, Quartus has a tool that spares you this effort! It's called the **IP Catalog**.

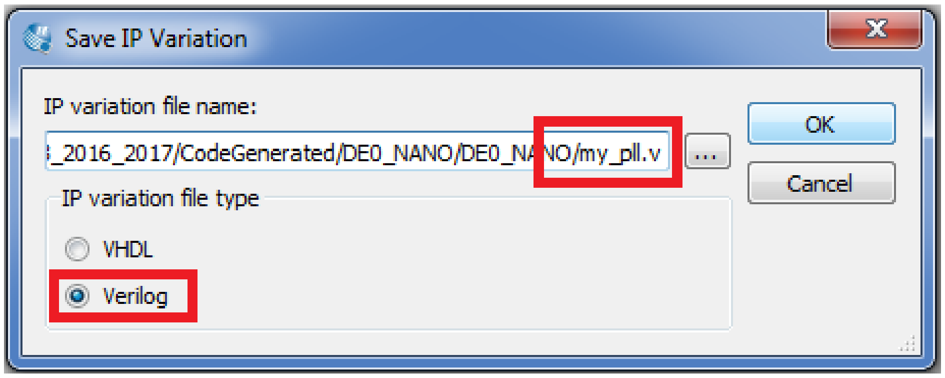
You can launch the IP Catalog via `Tools` > `IP Catalog`. A sub-window should open on the right of your main Quartus window and may actually already be opened by default.

Once you have launched the tool, be sure to have the "Cyclone IV E" selected for the device family, as in the figure below. Then go in `Library` > `Basic Functions` > `Clocks; PLLs and Resets` > `PLL` and double-click on the `ALTPLL` module: this is a primitive that can be handled by the wizard, and that allows you to later modify it as you like.


Device Family: Cyclone IV E

ALTPLL

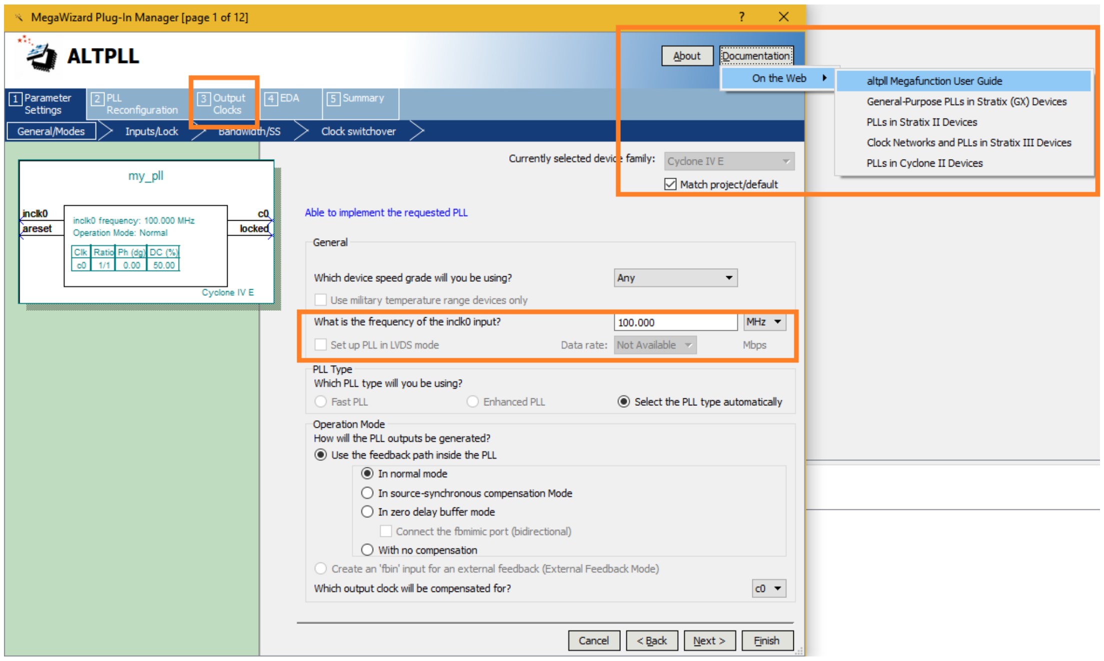
In the next screen, select 'Verilog' as the IP variation file type and give it the name `my_pll.v`, as in the screen below.


Save IP Variation

IP variation file name: _2016_2017/CodeGenerated/DE0_NANO/DE0_NANO/my_pll.v

IP variation file type: VHDL [✓ Verilog]

In this exercise, we ask you to create a PLL, instantiate it in your SystemVerilog code, and use it to halve the clock frequency of the example circuit given earlier in this tutorial. Compile, run, and embark on a wonderful journey of discovery by playing around with all the wonderful things you find! Try to get an idea of the different options and configure your PLL appropriately, do not hesitate to call an assistant and to browse the documentation.



Currently selected device family: Cyclone IV E

☑ Match project/default

Able to implement the requested PLL

General

Which device speed grade will you be using? Any

☐ Use military temperature range devices only

What is the frequency of the inclk0 input? 100.000 MHz

☐ Set up PLL in LVDS mode Data rate: Not Available Mbps

inclk0 frequency: 100.000 MHz
Operation Mode: Normal

Clk	Ratio	Ph (dg)	DC (%)
c0	1/1	0.00	50.00

PLL Type

Which PLL type will you be using?

○ Fast PLL ○ Enhanced PLL ● Select the PLL type automatically

Operation Mode

How will the PLL outputs be generated?

● Use the feedback path inside the PLL

● In normal mode

○ In source-synchronous compensation Mode

○ In zero delay buffer mode

☐ Connect the fbmimic port (bidirectional)

○ With no compensation

○ Create an 'fbin' input for an external feedback (External Feedback Mode)

Which output clock will be compensated for? c0

What now? Some ideas to go further on your own…

So, that was easy, wasn't it? Now that you have finished far earlier than expected, here are some more things you can do to occupy your mind:

- Try adding new triggers to Signal Tap. Can you observe the LED transitions?
- In the Signal Tap figure of point 5, you can select different 'Trigger Conditions' (Basic AND, Basic OR, Advanced): understand and explore the differences.
- What is your design's longest path (timing-wise)?
- Draw a block diagram of your SystemVerilog design.
- View the schematics: first use `Tools` > `Netlist Viewers` > `RTL viewer` to view a schematic version of your SystemVerilog design. Now, see what it was transformed into by using the Post-Fitting viewer. Anything different? :P
- Use new IP Catalog functions, and observe their effect!

And that's really it for now. Don't forget your homework (details on the Moodle page), hope you had fun!

That's all folks!