# FPGA Lab

## 1 Background

### 1.1 FPGA - Field Programmable Gate Array

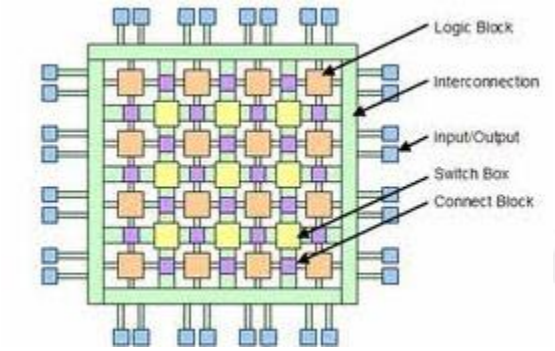
The Field-Programmable Gate Array (**FPGA**) is a semiconductor device that can be programmed after manufacturing. Instead of being restricted to any predetermined hardware function, an FPGA allows you to program product features and functions, adapt to new standards, and reconfigure hardware for specific applications even after the product has been installed in the field—hence the name "field-programmable". You can use an FPGA to implement any logical function that an application-specific integrated circuit (ASIC) could perform, but the ability to update the functionality after shipping offers advantages for many applications.

FPGAs contain an array of **configurable logic blocks** and a hierarchy of **reconfigurable interconnects** that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform simple logic gates like AND and XOR or more complex combinational functions. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory, allowing the implementation of sequential logic.
Programming an FPGA is done by using special CAD tools which take a design described in Hardware Description Language (e.g. VHDL or Verilog), map it to the specific FPGA to generate a bit steam that will be used to configure the logic blocks and the interconnect to implement the required net-list. The tools can also generate timing information for the design.

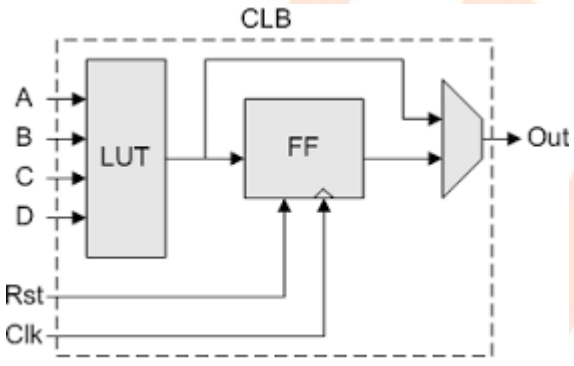

## 1.2 Architecture

### 1.2.1 Logic Blocks

The most common FPGA architecture consists of an array of logic blocks (called configurable logic block, CLB, or logic array block, LAB, depending on vendor), I/O pads, and routing channels. The interconnect network is using connection boxes and switches to route the signals between the logic blocks.

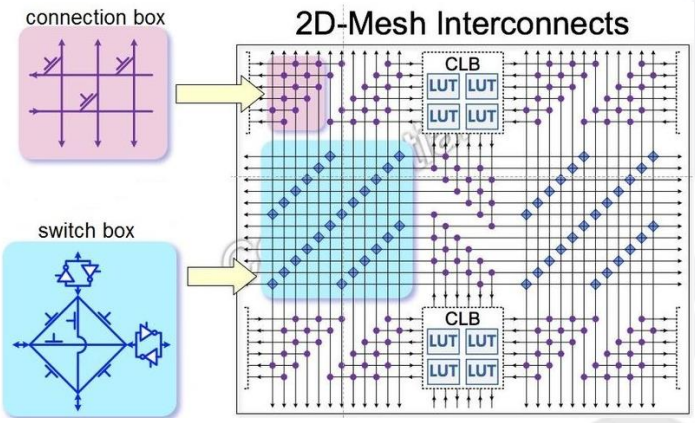



In general, a logic block (CLB or LAB) consists of a few logical cells. A simple cell that consists of 4-input LUT (Look-Up Table) and D-type flip-flop is shown below. The output of the cell can be either synchronous or asynchronous, depending on the programming of the mux. More complex cells may use multiple LUTs and include additional logic such as full-adder.



### 1.2.2 Interconnect

To accommodate a wide variety of circuits, the interconnect structure must be flexible enough to support widely varying local and distant routing demands together with the design goals of speed performance and power consumption. There is a different number of the routing architecture strategies (hierarchical, island-style etc). The basic programmable elements in the interconnect logic are the **connection box** and the **switch box** as depicted in the figure below. The topology and architecture of the interconnect network has a major impact on the performance and the density of the FPGA.




### 1.2.3 Hard Blocks

Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic DSP blocks, embedded processors, high speed I/O logic and embedded memories.

Higher-end FPGAs can contain high speed multi-gigabit transceivers and hard IP cores such as processor cores, Ethernet MACs, PCI/PCI Express controllers, and external memory controllers. These cores exist alongside the programmable fabric, but they are built out of transistors instead of LUTs so they have ASIC level performance and power consumption while not consuming a significant amount of fabric resources, leaving more of the fabric free for the application-specific logic. The multi-gigabit transceivers also contain high performance analog input and output circuitry along with high-speed serializers and de-serializers, components which cannot be built out of LUTs. Higher-level PHY layer functionality such as line coding may or may not be implemented alongside the serializers and de-serializers in hard logic, depending on the FPGA.

## 2 Setup the Environment

First of all, in order to work with the FPGA, we need to install some tools. We are going to use the Altera Cyclone-IV FPGA which works with "***ModelSim***" and "***Intel Quartus Prime***".

***ModelSim*** is a simulation environment of hardware description languages such as VHDL and Verilog (a lot like SimVision of Cadence which you probably used in the past). With this tool we can simulate our design before implementing it to hardware.

***Intel Quartus Prime*** is programmable logic device design software which enables analysis and synthesis of HDL designs in an FPGA (specifically Altera FPGAs). This tool first compiles the design, then synthesizes and implements it to actual hardware in the FPGA and eventually converts it into a bit-stream file that contains the programming information for the FPGA. When programming the device, it loads the bit-stream file to configure the logic elements of the FPGA according to the design we created.

You can download the ***Quartus Prime Lite Edition*** and ***ModelSim*** together from:
`http://fpgasoftware.intel.com/?edition=lite`

In "Select release" select 18.1 and choose Windows as the operating system. When try to download the files it will ask you to sign in, so since you probably don't have an Intel account, choose to "Sign up for a basic account" and after you verify the registration in your mail you should be able to sign in and download the files. It might take the system a few minutes to realize you verified the registration. If it doesn't work, try to close the browser and enter again. Inside 'Individual Files', download the following:

- **Quartus Prime (includes Nios II EDS)** (1.7GB)
- **ModelSim-Intel FPGA Edition (includes Starter Edition)** (1.1GB)
- **Cyclone IV device support** (466.6MB)

Notice the files are quite large, so it could take some time to complete the download.

After the download is complete, install both ***ModelSimSetup*** and ***QuartusLiteSetup*** (in ModelSim choose the starter addition that doesn't require a license).

As we said earlier, after creating a bit-stream, we need to load it into the FPGA. To do so, we connect the board using the USB-Blaster cable. In order to the computer to identify the connection, you need to install the driver which is inside the Quartus folder you installed. Go to: `C:\intelFPGA_lite\18.1\quartus\drivers` and install: `DPInst.exe`.

Also, in order to communicate with FPGA and provide it with inputs, we'll use the UART interface. To do so, we also connect the board with a USB TTL Serial cable. The driver for this cable is at:
`https://www.ftdichip.com/Drivers/CDM/CDM21228_Setup.zip`

Last thing we need is a terminal to provide the inputs through the serial COM (UART). You can download a simple terminal from: `https://www.compuphase.com/software/termite-3.4.zip`

# 3 Example exercise

Download **"FPGA_lab.zip"** from the Moodle. Inside you'll find 4 folders:

- Docs: Where a copy of this and other guiding documents are located.
- Fpga_project:

  There are two important files – fpga_lab.qpf , fpga_lab.qsf
  - The .qpf file is our project in Quartus. When you open it, it is already loaded with all the modules of the design.
  - The .qsf file is actually a text file generated by Quartus. Inside you can see the Verilog files of your design, and also the pins in the FPGA which the I/Os were assigned to. (You shouldn't make changes there, just know it exists).
- Sim:
  - Under src_list you'll find an .f file which contains all the Verilog files required for the simulation. Notice the testbench files are there, unlike the .qsf file which only needs the synthesizable modules.
  - The modelsim directory will be your work area for the simulations.
- Src:

  There you'll find all the modules of the design. You don't need to change anything in most of them, only in "my_design.sv" (your personal design) and "my_ design_intrfc_wrap.sv" (the interface of your design) as required in your assignments. In the simulation part you might need to add some changes in the "tb.sv".

Notice that the files are in a System-Verilog (.sv) format, not the usual Verilog you are used to. You are not required to know System-Verilog since the regular Verilog works fine in System-Verilog. If you have any trouble understanding some of the code because of the small changes between the formats we are here to help ☺

- Python:

  An optional advanced Python remote interface, as referred later in this document, usage of the Python interface is not mandatory for the lab, we'll discuss this at the lab.

If you open your interface ("my_ design_intrfc_wrap.sv") you'll see that your interface is:

**Inputs:**

- `clk` – The clock period is defined in the simulation to be 40ns (25MHz), similar to the clock period of the FPGA board.
- `rst_n` – connected to PIN_T1 in the board. In the simulation it is activated in the beginning of the simulation.
- `gp_button` – connected to PIN_N22 in the board. Can be used as you wish. Notice that as long as you press the button, `gp_button` will be at '1'.
- `intrfc_regs_in` – eight registers, 32-bit each, to which you can write your inputs. It's up to you to decide how many registers to use in your design.
- `intrfc_regs_in_valid_pulse` – each input register has a pulse signal indicating that the data that was written to it is valid. You might find it useful building your design.

**Outputs:**

- `intrfc_regs_out` – eight registers, 32-bit each, which you can get your outputs from. It's up to you to decide how many registers to use in your design.
- `led` – this output is connected to the 4 LEDs on the board.

Inside this module we instantiate our design and connect the input and output registers to the relevant signals in our design (for example, the first input register is connected to '`in`').

Let's take a look at "my_design_example.sv".

The first block of the design uses 2 input registers, their valid pulses and 1 output register. This is basically an accumulator which either adds the input to the result, or subtracts the input from the result, depending on the '`select`' signal.

The second block simply counts the number of times we press the button and display the result on the LEDs. Notice we sample `gp_button` to identify a single press so the counter won't proceed counting while the button is pressed.

Now open "tb.sv" and go to the end of the file. There you'll see two blocks – the first one simulates the board clock and the reset. The second one simulates the button.

## 3.1 Simulation

After all this introduction, we can finally run the simulation:

Open ModelSim.

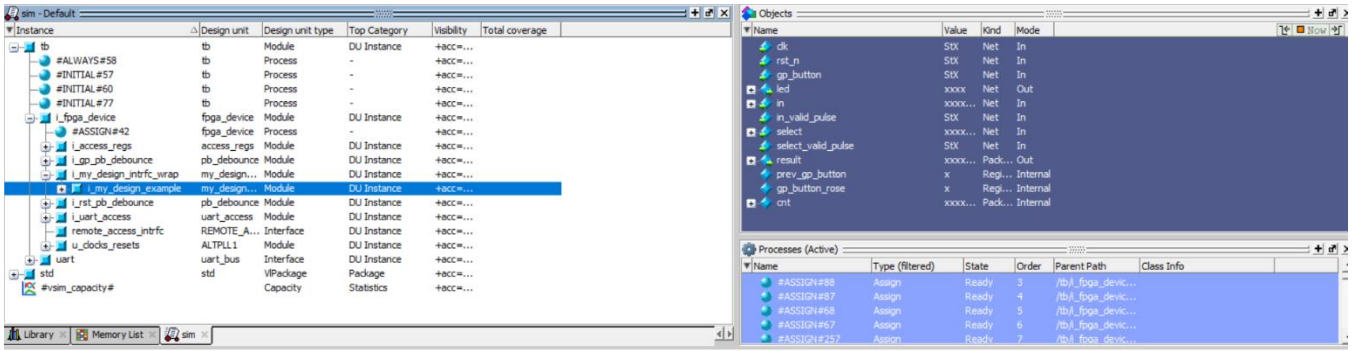
In the Transcript window at the bottom, run the following commands:

```
cd <path to your lab_area>/sim/modelsim
vlib work    // Create your work area for the simulation – Run only the first time.
vlog –f ../src_list/fpga_lib_sim.f     // Load the Verilog files.
vsim tb      // Open a simulation session with "tb" as the top module.
```

```
Transcript
# Reading C:/intelFPGA/18.1/modelsim_ase/tcl/vsim/pref.tcl
ModelSim> cd  C:/Users/jgrinbla/Documents/fpga_lab_local_7Oct19_1657_valid_OK/fpga_lab_local/laptop_local_fpga_lab/sim/modelsim
ModelSim> vlib work
ModelSim> vlog -f ../src_list/fpga_lab_sim.f
# Model Technology ModelSim - Intel FPGA Edition vlog 10.5b Compiler 2016.10 Oct  5 2016
# Start time: 12:09:51 on Nov 05,2019
# vlog -reportprogress 300 -f ../src_list/fpga_lab_sim.f
# -- Compiling module ALTPLL1
# -- Compiling interface REMOTE_ACCESS_INTRFC
# -- Compiling module fpga_device
# -- Compiling module my_design_example
# -- Compiling module my_design_intrfc_wrap
# -- Compiling interface uart_bus
# -- Compiling module access_regs
# -- Compiling module uart_access
# -- Compiling module uart_rx
# -- Compiling module uart_tx
# -- Compiling module pb_debounce
# -- Compiling module tb
#
# Top level modules:
#       tb
# End time: 12:09:51 on Nov 05,2019, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
ModelSim> vsim tb
# vsim tb
# Start time: 12:09:56 on Nov 05,2019
# Loading sv_std.std
# Loading work.tb
# Loading work.fpga_device
# Loading work.ALTPLL1
# Loading work.pb_debounce
# Loading work.REMOTE_ACCESS_INTRFC
# Loading work.uart_access
# Loading work.uart_rx
# Loading work.uart_tx
# Loading work.access_regs
# Loading work.my_design_intrfc_wrap
# Loading work.my_design_example
# Loading work.uart_bus
```

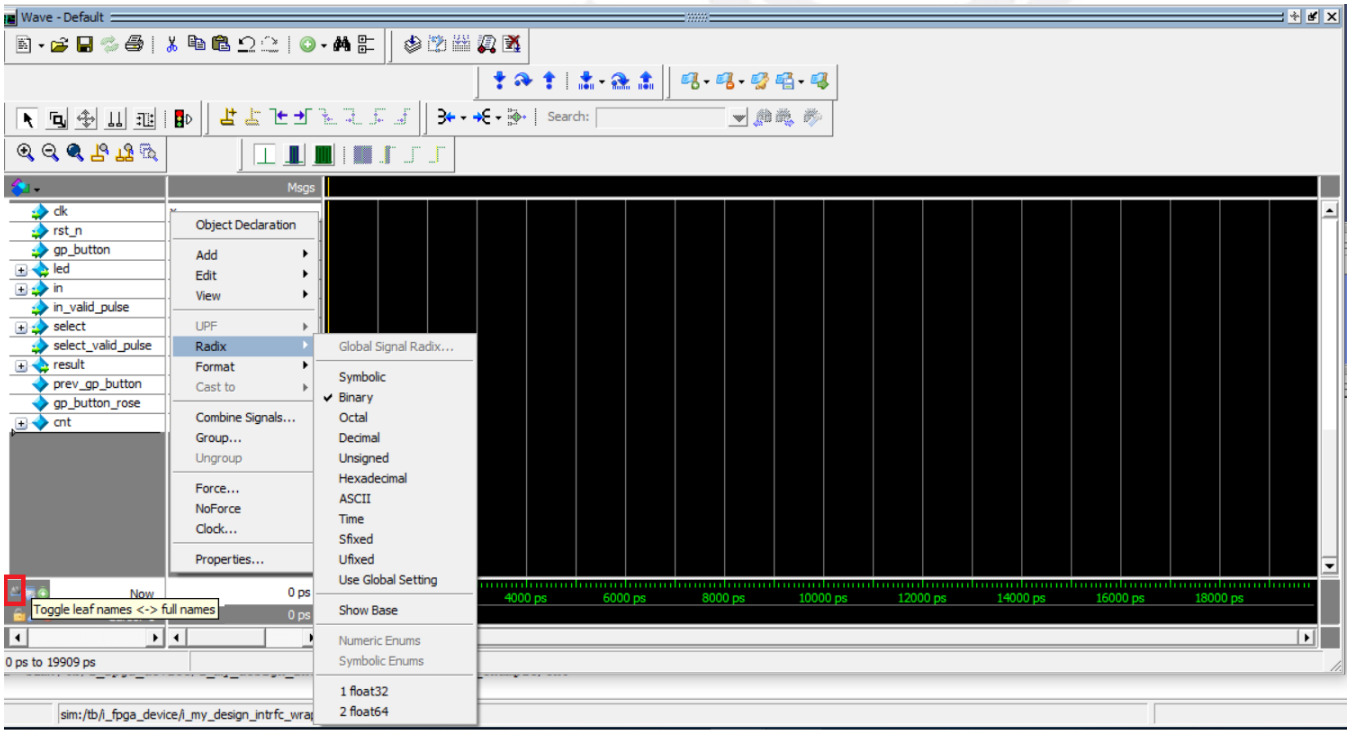
At the upper left side you can see the hierarchy of your design.

Under: tb -> i_fpga_device you'll find "my_design". When you click on it you can see on the right side in the 'Objects' sub-window the signals of your design.




In order to be able to probe signals during the simulation, click on 'View -> Wave'. Drag the signals from 'Objects' inside the waveform window that opened.

You can always go inside other modules in the hierarchy and drag any signal you want into the waveform.




[You can change the radix as you like. If you click at the bottom left (marked in red in the picture) the names will appear without the entire path of the hierarchy].

After probing all the signals we needed, we can start the simulation by going to 'Simulate -> Run -> Run –All'.

In the Transcript you can see that the UART is ready to accept an input.

The inputs are being written to the input registers. When we write, we tell the UART to which register we want to write. Therefore it's important to know which register is connected to each signal in our design. For example, signal '`select`' is connected to `intrfc_reg_in[1]` (look at "my_ design_intrfc_wrap.sv")

**When you enter an input, it's *very important* to follow the correct syntax (Don't use CAPS and make sure there is no space between '#' and w/r):**

- Write execution: `#w <number of input register> <input data (in hexa)>`
- Read execution: `#r <number of output register>`

For example:

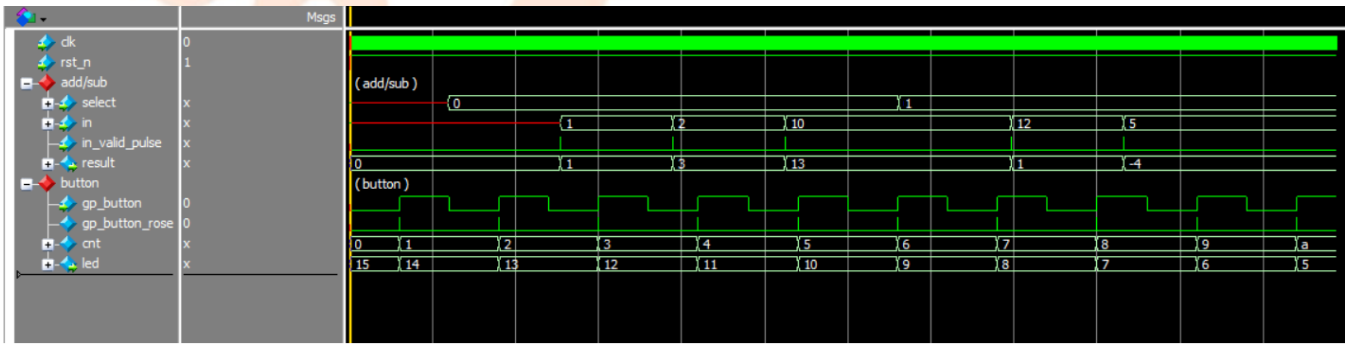
```
Ready> >> #w 1 0          // select = 0 – adds the input to the result
Ready> >> #w 0 1          // write 1 to input register [0] (in)
Ready> >> #w 0 2
Ready> >> #w 0 a
Ready> >> #w 1 1          // select = 1 – subtracts the input from the result
Ready> >> #w 0 c
Ready> >> #w 0 5
Ready> >> #r 0            // read from output register [0] (result)
# fffffffc                // result = fffffffc (equals to -4)
```

We first entered the '`select`' so that the program will know what arithmetic action to do.

If you accidently make a mistake, you'll need to restart the simulation. Here is what you need to do:

- 'Simulate -> Break' and press <Enter> in the Transcript.
- **If you made changes in the Verilog files**, run again: `vlog –f ../src_list/fpga_lib_sim.f`
- 'Simulate -> Restart'.
- 'Simulate -> Run -> Run –All'.

You don't have to actually read from the output register, you can see the results in the waveform:



In the add/sub group signal you can see the input we inserted and how the result changes according to it depending on the value of `'select'`.

In the button group signal you can see how every time a rise of the button is detected, the counter goes up. (The `led` signal is defined as `~cnt` since the LEDs are on when they equal to 0).

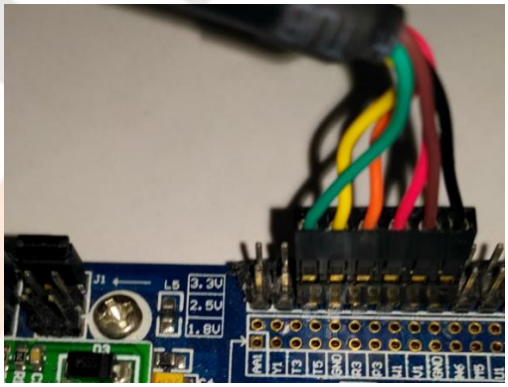
## 3.2 Run the program on the FPGA

Connect the FPGA cables as shown in the pictures:

**Power cable**

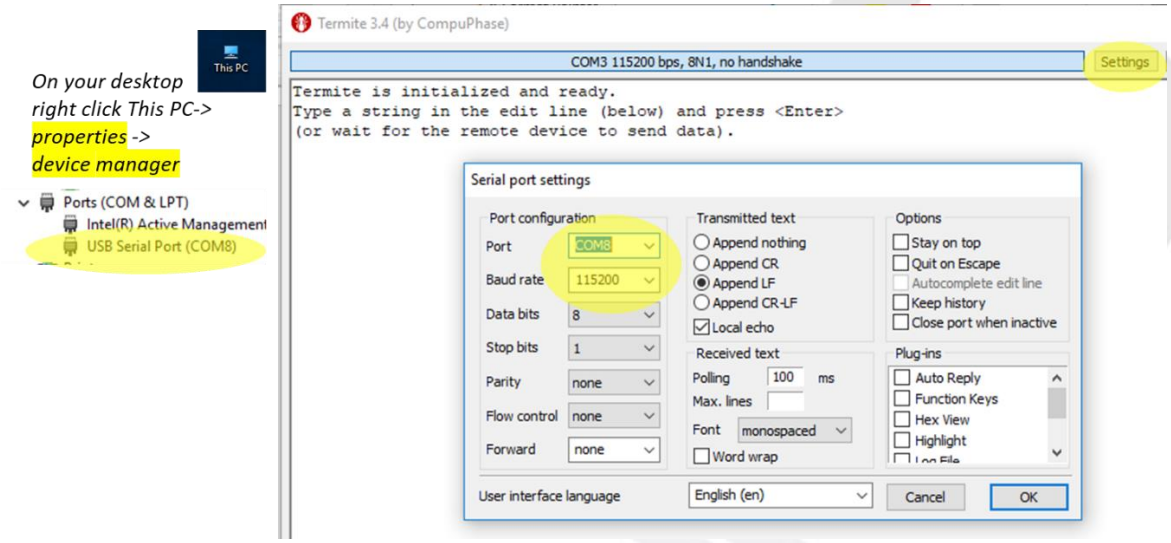
**USB-Blaster cable**


**USB TTL Serial cable**



Plugging to the exact header pins as in the picture is crucial, the cable should connect to the inner row of pins with the green wire align to the 3rd pin from the left, when looking from the board side (right picture above)

Check on the Device Manager which COM is your serial COM as follow, right click “This PC” icon on your desktop and go to properties -> device manager, scroll to ports and check your indicated USB serial port. Now open Termite that you installed. Inside 'Settings' choose the right COM and in the Baud-rate enter 115200.

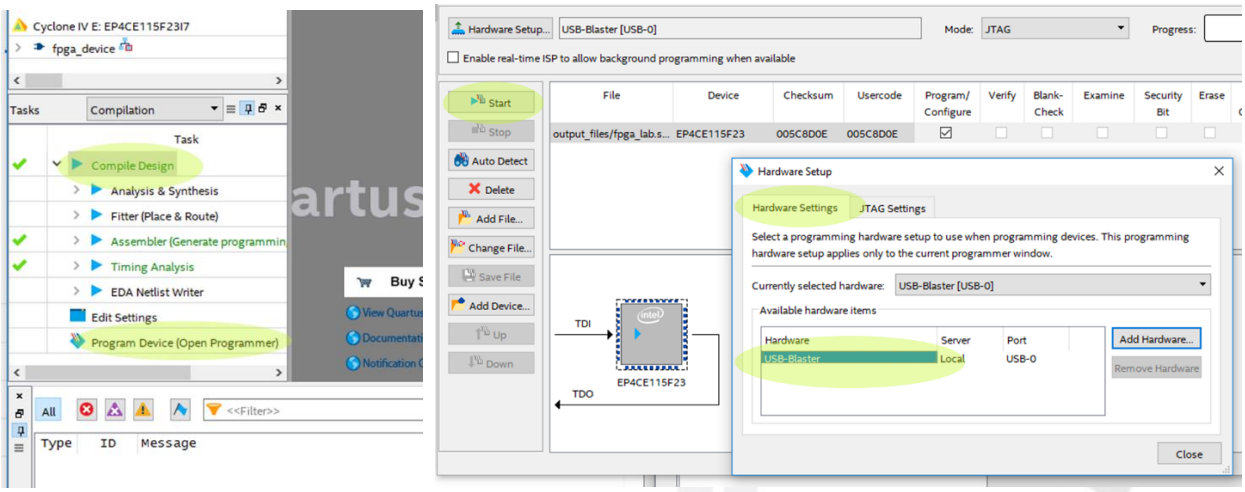



## 3.3 Program the FPGA

Now, open the **fpga_lab.qpf** file in the fpga_project folder. It will open Quartus.

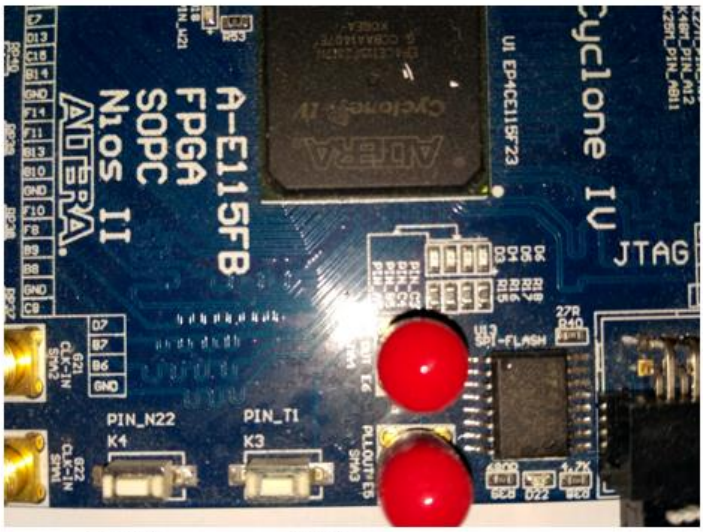
Double-click on **Compile Design** on the Tasks sub-window on the left, at this stage compilation errors and warnings will show up, while most warnings can be ignored, errors must be fixed.

After good compilation is achieved, turn on the board, make sure your usb-blaster cable is connected also to your computer, double-click on **Program Device**, and in the window that pops up click on Hardware Setup and select and double click USB-Blaster. Then click 'Start' on the previous window to load the program into the FPGA.




## 3.4 Run the program on the FPGA

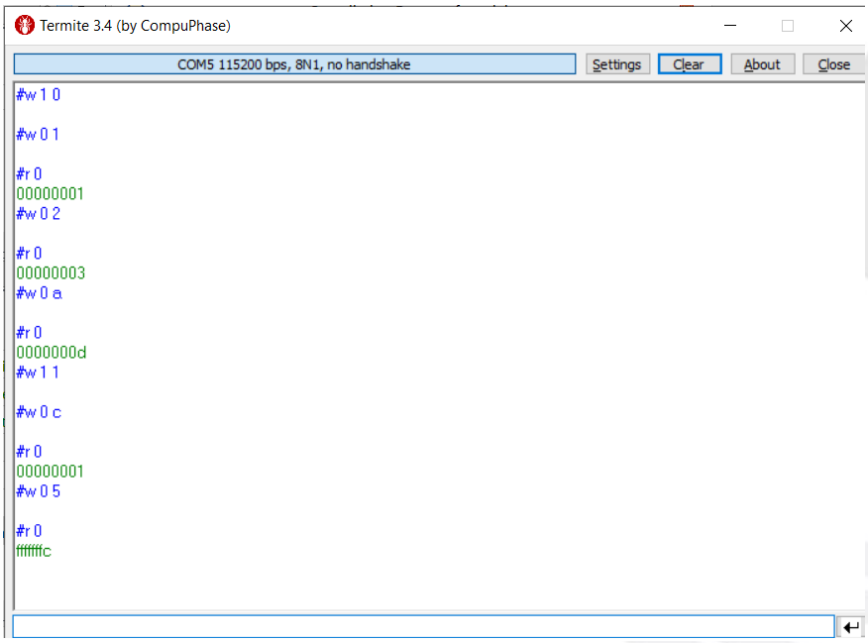
Now you can write and read through the terminal in the same manner as in the simulation console. You can also press on **PIN_N22** and check if the LEDs behave as expected. **Remember to press the reset button (PIN_T1) at the beginning to initialize the program.**




**PIN_N22**
**General Purpose**
**Currently effect lends**

**PIN_T1**
**Connected**
**to Reset**

The terminal should look something like this:




# 4 'Burning' the FPGA programming

Notice that upon following above instructions once you power-off the board your FPGA program will be lost, and you will need to re-program the FPGA each time you power it on.

To avoid that, once your design is stable you can 'burn' the compiled program (commonly named image or bit-stream) to a non-volatile flash memory on the board which the FPGA will automatically load upon power-up. This will eliminate the need to connect the USB-Blaster cable and re-program the FPGA upon power-up in case your program did not change, furthermore you will not need at all to bring-up the Quartos utility in such case.

See separate document for details on how to burn the FPGA bit stream.

# 5 Using the Python interface

The UART COM port interface provide the ability to communicate with the FPGA design but is very basic and limited, essentially it can only transmit entered characters and display received characters which are interpreted by your design wrapper logic interfacing with your design ports. Alternatively, a Python program interface infrastructure is available to allow coding a python application interface with the FPGA logic.

See separate document for details on the Python interface infrastructure.

# 6 Your personal assignment

Each pair of students will get a different assignment in the lab.

# 7 Submission instructions

In your final report, please attach the following:

- A **commented** Verilog code of your design. Explain what changes you had to make in each module.
- Screenshots of the simulation – relevant waveforms and transcript of ModelSim showing your design works properly. **Explain your results.**
- Screenshots of the terminal showing the design works properly on the FPGA as well.

You are not required to include the example exercise in your report.