

UNIVERSITY OF LIEGE

MICROELECTRONIC AND IC DESIGN

LABORATORY

VHDL and FPGA laboratory



Hervé Pierre

E-MAIL: hpierre@uliege.be

Academic year 2019-2020

1 Introduction

1.1 Objectives

The purpose of this laboratory is to familiarize you with the software necessary for the creation of your VHDL project as part of the microelectronics and IC design course, and the development board DE0 nano.

The component used for this laboratory is a Cyclone IV EP4CE22F17C6N. Search in the datasheet the resources you have for your project. In particular, how many logic elements, how many inputs/outputs, etc. do you have?

1.2 Software needed

You will need the *Quartus* Prime software on your computer BEFORE the beginning of the lab session. When installing *Quartus*, pay attention to add the Cyclone IV series of FPGA and the *ModelSim* software. Also, let the software install the USB driver (essential to program the board). *Quartus* is available on Windows and Linux. The installation on Linux is rather simple, If you encounter issues, please contact the assistant.



Figure 1

2 Theoretical considerations

2.1 VHDL workflow

The conception of a VHDL project can be resumed by this diagram:

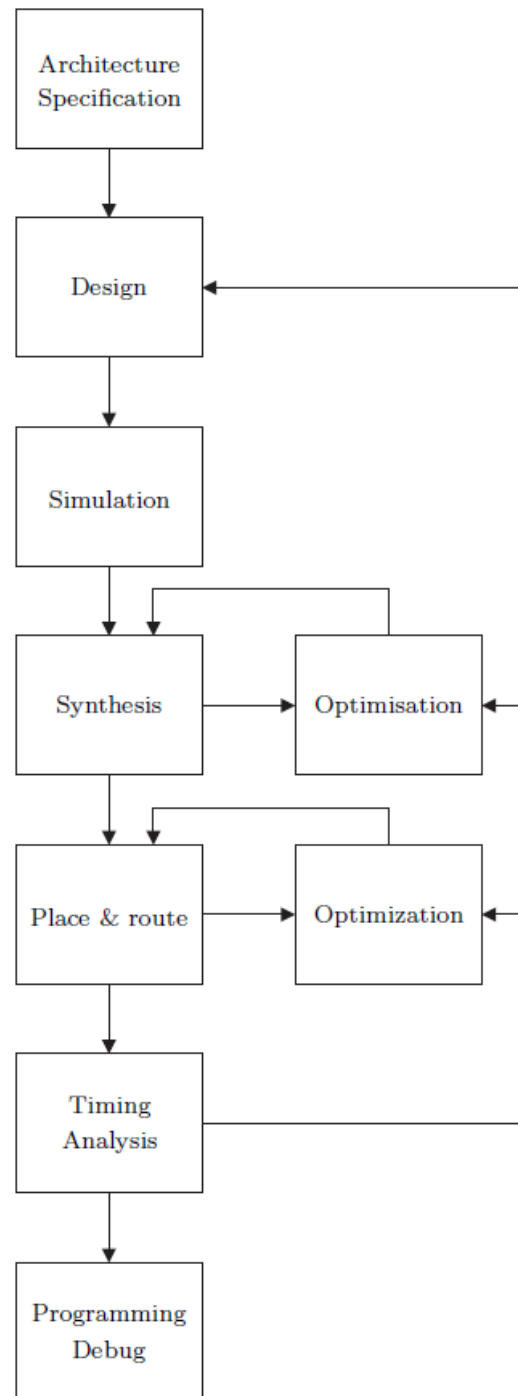


Figure 2: Diagram of the developpement of a VHDL project.

Just a quick review of this diagram:

- Architecture specification: This first step is the most important. It consists of writing the program specifications, or in other words, write down all of the functions that the program should be able to perform. This specification will be very useful for debugging, because it ultimately creates a set of test scenarios that the program will have to be able to perform without incident. The creation of the architecture consists in thinking about the organization of the code, and the resources required. In particular, we generally start with the creation of a state machine complete, as well as a list of inputs/outputs (I/O). The different signals and resources also need to be considered. At the end of this step,

it should be possible to have a first idea of all the necessary resources (pins, I/O, RAM, registers, etc.) to the implementation of the program. It is thanks to this reflection on the architecture of the program that it will be possible to determine its feasibility and/or determine the suitable one for the program. In this laboratory, and by extension of your project, the component is imposed on you. It is therefore your responsibility properly assess the resources required to determine if your program is feasible. If this is not the case, the specifications should be revised downwards.

- Design (program): This step resumes the actual programming phase. It's mandatory to scrupulously follow the architecture that you defined in the previous step, so as not to use unnecessary resources.
- Simulation: This step, although optional, allows you to debug your project, to eliminate design errors. At this stage, it is already possible to check if the program meets the specifications of your project, via a set of scenarios tests. The software used during this laboratory for this step is *ModelSim*.
- Synthesis: Your program created and simulated, you just have to physically implement it. The first step of this process is the synthesis, ie the translation of your VHDL code in Boolean equations that can be implemented in the component. Thanks to *Quartus II*, it is possible to perform this step in one click. At the end of the process, a report will be sent to you. presented with, among other things, the resources necessary for the implementation of the program. If these resources are too important for your component, it will be necessary to optimize your code to reduce consumption, see decreasing specifications.
- Simulation: This step, although optional, allows you to make a first debug of your project, to eliminate design errors. At this stage, it is already possible to check if the program meets the specifications of your project, via a set of scenarios test. The software used during this laboratory for this step is *ModelSim*.
- Place & Route: Your project has been translated into logic equations, but it is not yet under a programmable shape in the component. We still have to decide on the location different flip-flops used, control signals, etc. This is achieved, always by *Quartus* software, during this step. The analysis is generally relatively accurate and it is rare that a program that is analysable is not implementable. However, it is possible. It is then advisable to carry out some optimizations if such is the case.
- Timing analysis: For applications where speed is essential, you also need analyze the timings, i.e. the time taken for a signal to pass from the input of the component at its output (among others). However, this step will not apply to your programs as part of this course.
- Programming debug: Last step: place the program on the component and check that everything is going well. If the simulation has been successfully performed, the project should run from first attempt.

2.2 Writing clear code

Each programmer has his own style for coding, but there are a few things to keep in mind keys to keep a clean code. This helps to reduce errors, and to facilitate proofreading. Comments are also very important! Here is a non-exhaustive list of items to respect:

- Proper indentation of the code.
- Choice of a naming convention (for example: types start with a lowercase and variables with a capital letter, etc.).

- Choice of consistent and readable names.
- Comment for each element a bit complicated.

Here is an example of readable code. It corresponds to our nomenclature choices, but nothing you prevents you from using your own.

Example below:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;    -- for the unsigned type

entity cardio is
  port
  (
    --Input ports
    clk      : in std_logic;
    heart    : in std_logic;
    --Input ports
    led_fast : buffer in std_logic;
    led_slow : buffer in std_logic;
  )
end entity cardio;

architecture cardio_arch of cardio is
  signal cnt      : integer range 0 to 127 := 0;
  signal heart_old : std_logic := '0';
begin
  counter : process(clk)
  begin
    if(rising_edge(clk)) then
      if(heart_old = '1' and heart = '0') then
        --Do something immediately !!
        cnt<=0;
      else
        if(cnt/=127) then
          cnt<=cnt+1;
        end if;
      end if;
      heart_old<=heart;
    end if;
  end process counter;
end architecture cardio_arch;
```

3 Displaying a square on a VGA screen

3.1 Specifications

This first code is very simple: it involves generating a valid VGA signal at a resolution of 800X600 pixels. It is also displaying a small red square in the center of the screen. So there is only one extremely simple test scenario: check that the system generates a good VGA output.

In addition, the only input to the system is the clock set to 50MHz. It will result in a very simple simulation.

3.2 How VGA works

The VGA protocol is an analog protocol, used for a long time to control the CRT (Cathode Ray Tube) screens. This aspect should be kept in mind to understand the needs of this interface: it is needed to send data for a region greater than that required for display, as well as vertical and horizontal synchronization control signals, which are used to return the beam to the initial position.

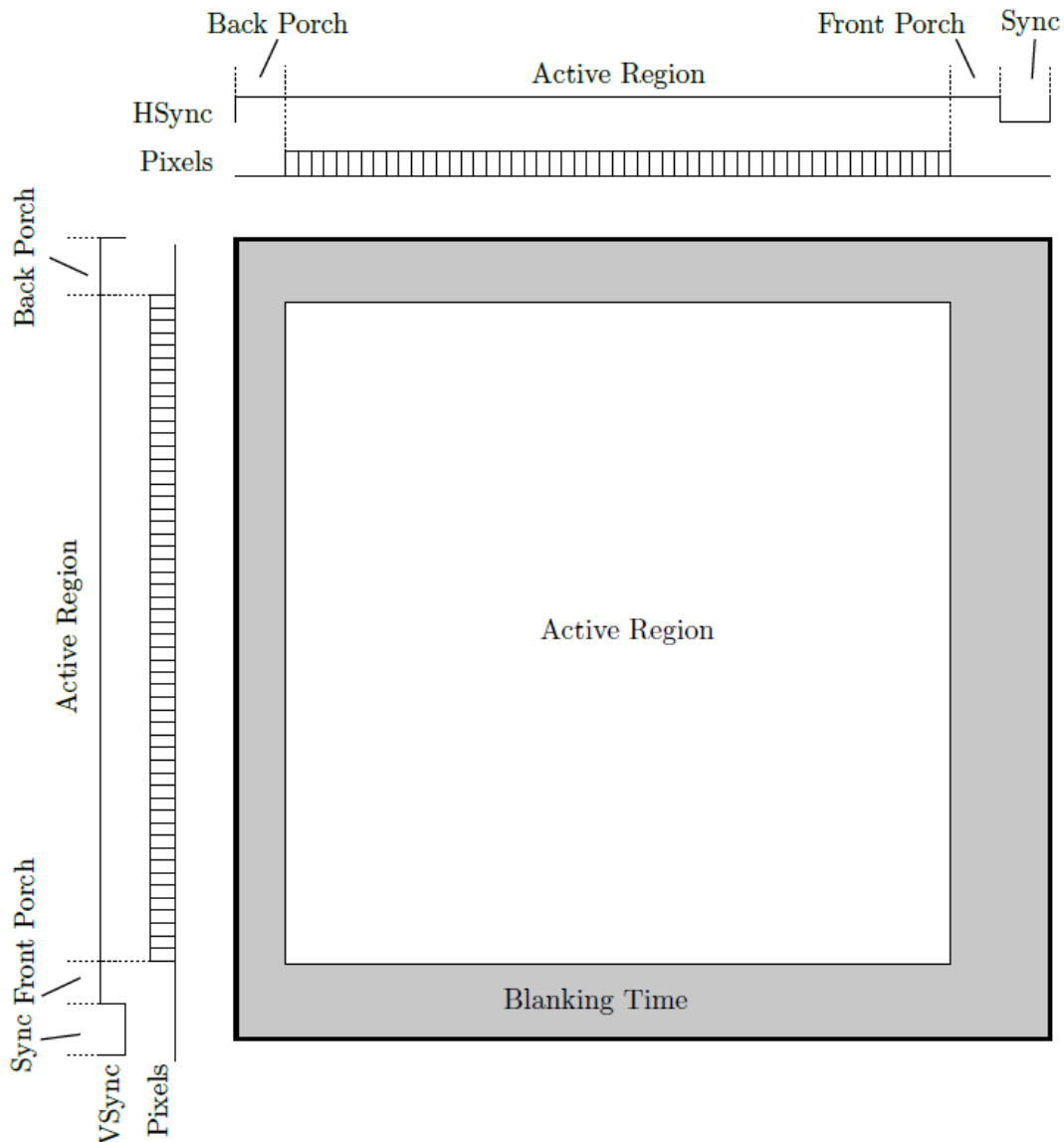


Figure 3: Simple figure showing synchronization signals, active and inactive region.

These signals must respect strict timings. For example, for a resolution of 800x600 @ 72 [Hz]:

- General specifications:
 - Pixel clock: 50MHz

- Line frequency: 48077Hz
- Image frequency: 72Hz
- Timings for a line:
 - Front porch: 56 pixels
 - Sync pulse: 120 pixels
 - Back porch: 64 pixels
- Timings for the image:
 - Front porch: 37 lines
 - Sync pulse: 6 lines
 - Back porch: 23 lines

Each pixel is coded by three analog voltages, giving the intensities of the sub pixels red, green, and blue. The daughter board above the DE0-Nano has three 4bits DAC (one for each color). The DAC is a simple, R-2R DAC (see the board schematics for further information). The color value is coded by 12bits: the first four bits are for the red channel, the second four bits are for the green channel and the last four bits are for the blue channel. You have therefore 4096 different colors.

3.3 Memory considerations

In a completely generic way, an image coded in true colors (24bits) requires a size equal to $800 \times 600 \times 3 = 1.44\text{Mo}$. The FPGA has on board a little less than 600Kbits of memory, which is far from it to be sufficient. Although, it is also possible to create memory cells from LE (Logical element), this decreases the available computing power, and therefore a better solution has to be considered. On the other hand, the card itself embeds an SDRAM memory of 32Mo, thus allowing to store several images. In this first program, the situation is much simpler: since we create a square of uniform color, just detect when you enter the area of the square (looking at where the pixel being sent is located), and set the red output to 1111, the green and blue to 0000. Otherwise, everything is kept at 0.

For this realization, only one process will be necessary. It will be executed at 50MHz, and therefore at every rising edge of the clock. Counters will record the line number and the pixel number on the line, and from there the synchronization signals will be generated.

3.4 Basic square code

Below, a code drawing a square on the screen:

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_ARITH.ALL;
use ieee.std_logic_UNSIGNED.ALL;

entity vga is port
(
    CLOCK_50 : in std_logic;
    RED: out std_logic_vector(3 downto 0);
    GREEN: out std_logic_vector(3 downto 0);
    BLUE: out std_logic_vector(3 downto 0);
```

```

        SYNC: out std_logic_vector(1 downto 0)
    );
end vga;

architecture Behavioral of vga is

    --Sync Signals
    signal h_sync :      std_logic;
    signal v_sync : std_logic;
    --Video Enables
    signal video_en : std_logic;
    signal horizontal_en : std_logic;
    signal vertical_en : std_logic;
    --Color Signals
    signal color : std_logic_vector(11 downto 0) := (others => '0');
    --Sync Counters
    signal h_cnt : std_logic_vector(10 downto 0) := (others => '0');
    signal v_cnt : std_logic_vector (10 downto 0) := (others => '0');

begin

    video_en <= horizontal_en AND vertical_en;

process
begin

    wait until(CLOCK_50'EVENT) AND (CLOCK_50 = '1');

    --Generate Screen
    if (v_cnt >= 0) AND (v_cnt <= 799) then
        color <= "000000000000";
        --Generate square
        if (((v_cnt >= 100) AND (v_cnt <= 300)) AND ((h_cnt >= 300) ...
                                                    AND (h_cnt <= 500))) then
            color <= "111111111111";
        end if;
    end if;

    --Generate Horizontal Sync
    if (h_cnt <= 975) AND (h_cnt >= 855) then
        h_sync <= '0';
    else
        h_sync <= '1';
    end if;

    --Reset Horizontal Counter
    if (h_cnt = 1039) then
        h_cnt <= "000000000000";
    else
        h_cnt <= h_cnt + 1;
    end if;
end process;
end Behavioral;

```



```

--Reset Vertical Counter
if (v_cnt >= 665) AND (h_cnt >= 1039) then
    v_cnt <= "0000000000";
elsif (h_cnt = 1039) then
    v_cnt <= v_cnt + 1;
end if;

--Generate Vertical Sync
if (v_cnt <= 642) AND (v_cnt >= 636) then
    v_sync <= '0';
else
    v_sync <= '1';
end if;

--Generate Horizontal Data
if (h_cnt <= 799) then
    horizontal_en <= '1';
else
    horizontal_en <= '0';
end if;

--Generate Vertical Data
if (v_cnt <= 599) then
    vertical_en <= '1';
else
    vertical_en <= '0';
end if;

--Assign pins to color VGA
RED(0) <= color(8) AND video_en;    --Red LSB
RED(1) <= color(9) AND video_en;
RED(2) <= color(10) AND video_en;
RED(3) <= color(11) AND video_en;   --Red MSB
GREEN(0) <= color(4) AND video_en;  --Green LSB
GREEN(1) <= color(5) AND video_en;
GREEN(2) <= color(6) AND video_en;
GREEN(3) <= color(7) AND video_en;  --Green MSB
BLUE(0) <= color(0) AND video_en;   --Blue LSB
BLUE(1) <= color(1) AND video_en;
BLUE(2) <= color(2) AND video_en;
BLUE(3) <= color(3) AND video_en;   --Blue MSB

--Synchro
SYNC(1) <= h_sync;
SYNC(0) <= v_sync;

end process;

end Behavioral;

```

4 Creating, simulating and programming

4.1 Discovering *Quartus* Prime

Quartus is a software suite developed by *Intel*. It will allow you to realize, within a single interface, all possible tasks on a programmable logic component of (here an FPGA):

- Writing programs not only via VHDL or Verilog, but also via state machines, logic schematics or block diagrams, etc.
- Generating logic functions from multiples sources.
- Generating reports.
- Programming components (CPLD, FPGA, etc).
- Various and varied analyzes: consumption, propagation time, resources, etc.

In addition, many blocks are available: the NIOS processor (codable in C), allowing you to implement a conventional processor on FPGA, different blocks like PLL's, DSP functions, various interfaces, etc. Also keep in minds that we are talking about the free version of the *Quartus* software. The commercial one as many other features, used for example in IC designs.

4.1.1 *Quartus* interface

This software works with the concept of Project (like pretty much all IDE). A project includes all the source codes, programming files, constraint files, etc., necessary for programming a component. You will therefore have only one project for your laboratory. Creating a project is rather simple:

Click on File → New Project Wizard

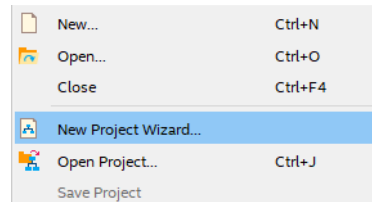


Figure 4: *Quartus* prime file menu.

A new window opens. By clicking on next, you can configure the different variables of your project, starting with the replacement of the files, and the name (put here vga). Then you can skip the step of adding files (we will create one later). Now, go to the configuration of the component, and select the EP4CE22F17C6 (which is the FPGA model reference on the DE0-Nano):

New Project Wizard

Family, Device & Board Settings

Device Board

Select the family and device you want to target for compilation.
You can install additional device support with the Install Devices command on the Tools menu.

To determine the version of the Quartus Prime software in which your target device is supported, refer to the [Device Support List](#) webpage.

Device family

Family: Cyclone IV E

Device: All

Target device

☐ Auto device selected by the Fitter

☒ Specific device selected in 'Available devices' list

☐ Other: n/a

Show in 'Available devices' list

Package: Any

Pin count: Any

Core speed grade: Any

Name filter:

☒ Show advanced devices

Available devices:

Name	Core Voltage	LEs	Total I/Os	GPIOs	Memory Bits	Embedded multiplier 9-bi
EP4CE22E22C8	1.2V	22320	80	80	608256	132
EP4CE22E22C8L	1.0V	22320	80	80	608256	132
EP4CE22E22C9L	1.0V	22320	80	80	608256	132
EP4CE22E22I7	1.2V	22320	80	80	608256	132
EP4CE22E22I8L	1.0V	22320	80	80	608256	132
EP4CE22F17A7	1.2V	22320	154	154	608256	132
EP4CE22F17C6	1.2V	22320	154	154	608256	132
EP4CE22F17C7	1.2V	22320	154	154	608256	132
EP4CE22F17C8	1.2V	22320	154	154	608256	132
EP4CE22F17C8L	1.0V	22320	154	154	608256	132
EP4CE22F17C9L	1.0V	22320	154	154	608256	132
EP4CE22F17I7	1.2V	22320	154	154	608256	132
EP4CE22F17I8L	1.0V	22320	154	154	608256	132

< Back Next > Finish Cancel Help

Figure 5: FPGA selection window.

Then, choose the right simulation system:

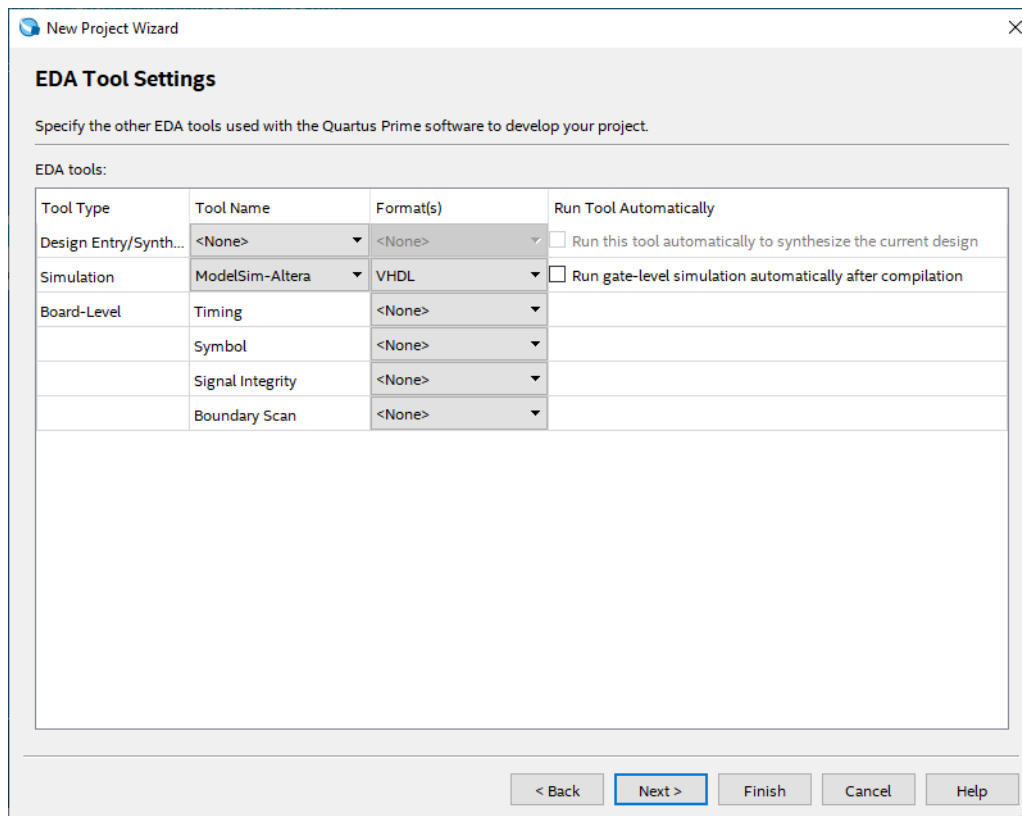


Figure 6: Tools settings and selection window.

The project has been created. Now you have access to the *Quartus* main window. This window is structured the same way as multiple IDE, you will easily find the info you are looking for.

4.2 Discovering *ModelSim*

4.2.1 *ModelSim* user interface

In order to verify the functioning of the system, it is possible to simulate it. To do this, go on Tools → Run Simulation Tools → RTL Simulation, which will launch the *ModelSim* program with all the pre-loaded parameters. All as for *Quartus*, the work-space is divided into different areas, the most important being these:

- Library: All the necessary booksellers are listed in this window. In particular, your program can be found in the Work library. Double-click on the name of your program to start the simulation.

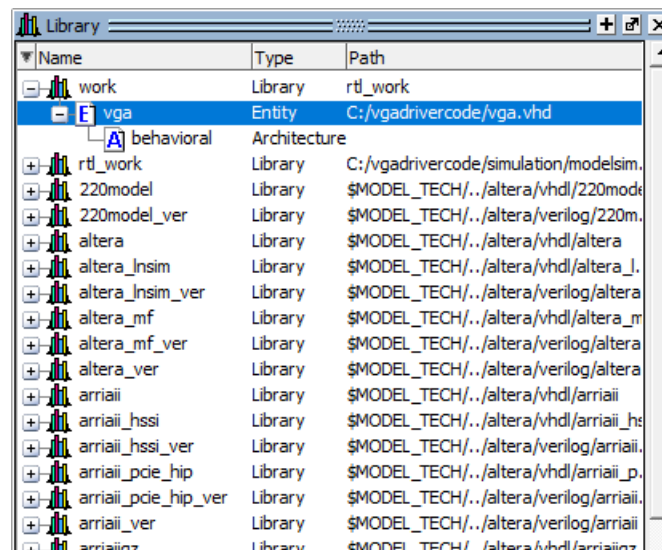


Figure 7: Click on work then on the entity you want to simulate.

- **Objects:** All available signals will appear in this window. You can, with the mouse, drag and drop them into the Wave window to display the result of the simulation of these signals.
- **Waves:** All the requested simulation results are available in this window. Here, the object CLOCK_50 has been moved (If the window does not appear, press View → Wave).
- **Console:** It is possible to control the software using a console (by entering a series of commands). Messages of the same type as those from *Quartus* are displayed here.

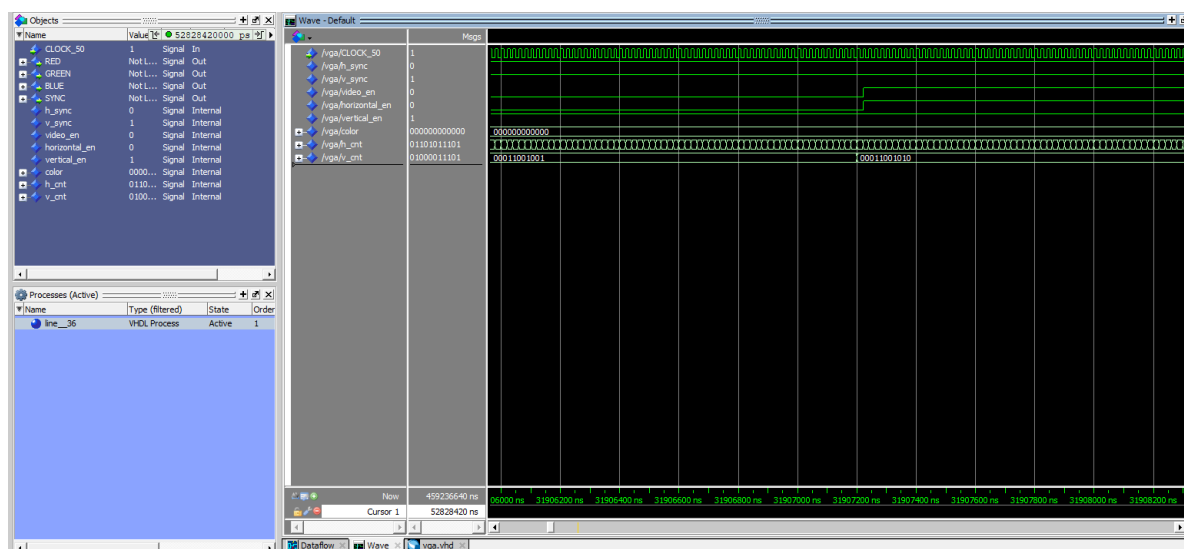


Figure 8: *ModelSim* interface with the Waves on the right and the Objects on the left.

When you double-clicked on your program in the Libraries window, you have started a simulation. In reality, your simulation has been initialized, and is at time 0, but nothing has happened yet. In fact, you must first give the software stimuli, that's basically determining system inputs. With *ModelSim*, there are several ways to work. Here are three ways:

- Start the simulation without any information, then give the stimuli as you go.

- Start the simulation with graphically generated stimuli in advance.
- Start the simulation with a TestBench file, which contains all the stimuli.

In this lab, we will see the first and the last possibility. Indeed, the file of TestBench is actually much shorter and easier to write than creating stimuli by hand. The first possibility can be practical to debug a strange situation: we send a few stimuli, we stop the simulation, we look at what is happening, then we impose other stimuli, we relaunch the simulation, etc.

4.2.2 Launching a simulation

First of all, we will have to define the length of the simulation. Set the time in the following window. A length of 20ms seem appropriate for a first simulation.

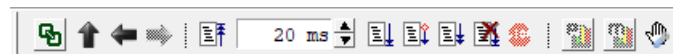


Figure 9: Setting the length of the simulation.

You can also choose the grid size by right-clicking on the time-scale:

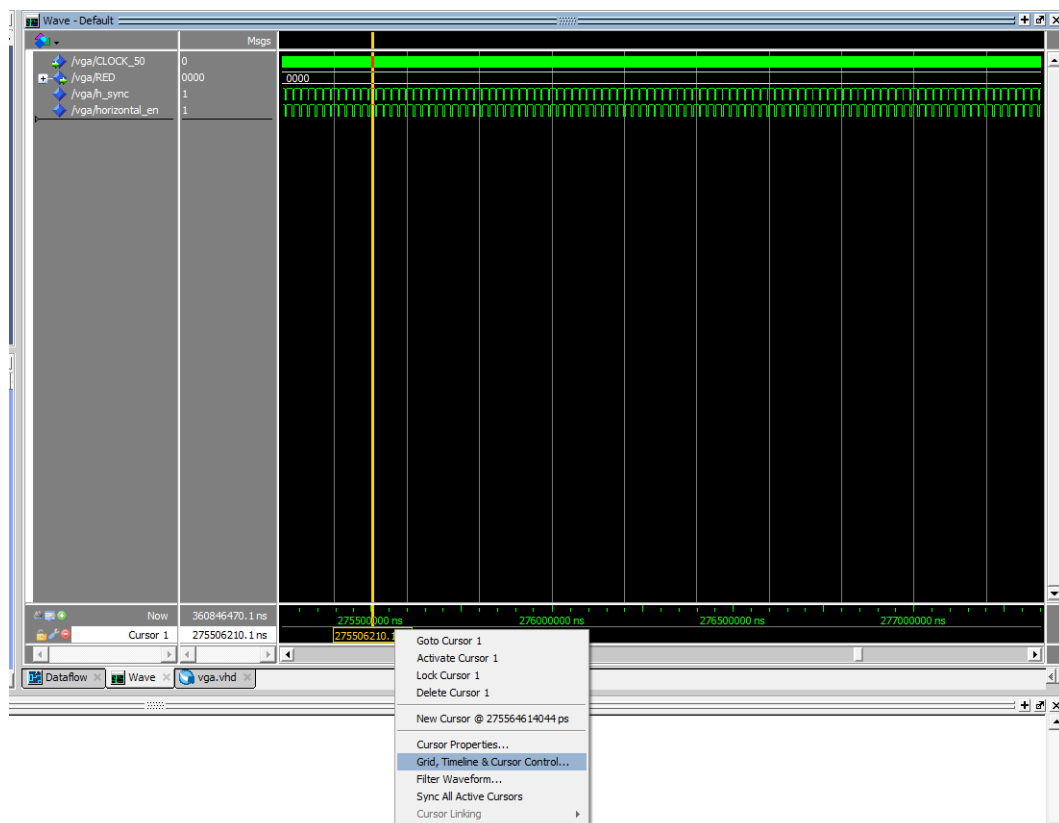


Figure 10: Right-clicking on the time-scale.

Select Grid, Timeline & Cursor Control... A new windows opens:

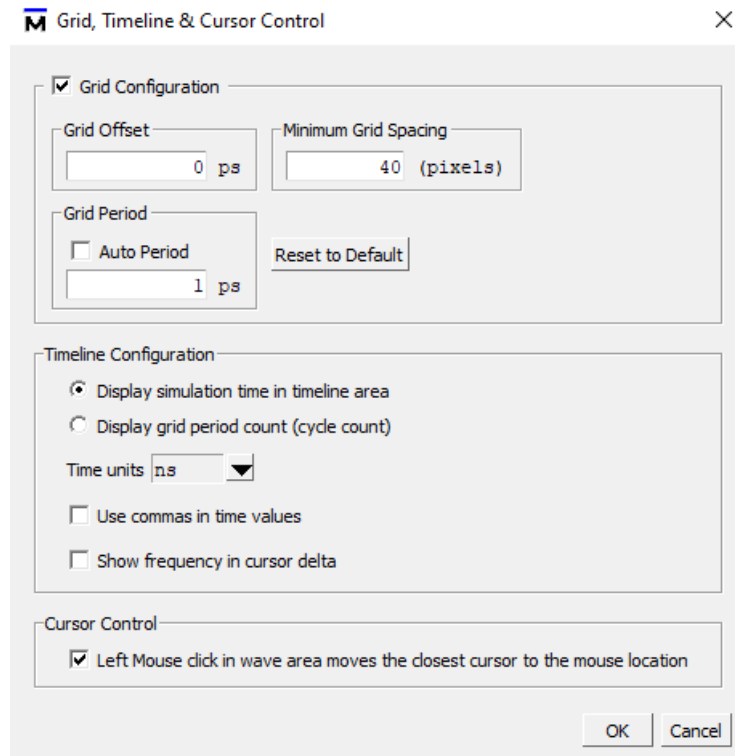


Figure 11: Grid, Timeline & Cursor Control window.

Select the parameters you want!

Now you have two options for your simulation:

- Clock stimulus.
- Stimulation by forcing signals values.

Once your simulation has started, first start by moving all the objects in the Wave window, which will allow you to view all the signals. Note that the entries, outputs, but also internal signals are available, which is very convenient for debugging a problematic situation.

In this simulation, we will use the clock stimulation. Right click on the CLOCK_50 wave:

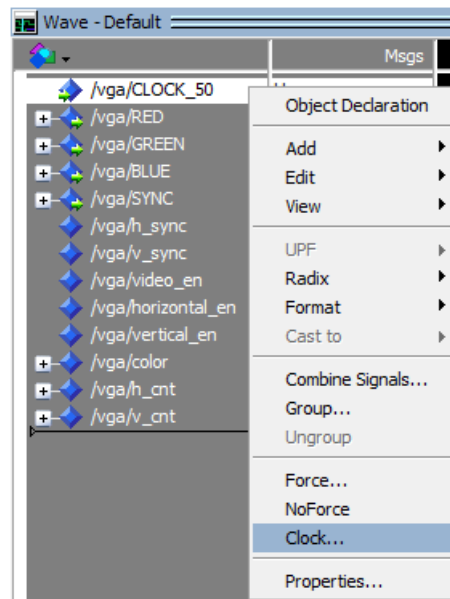


Figure 12: Right clicking waves.

Now click on Clock...:

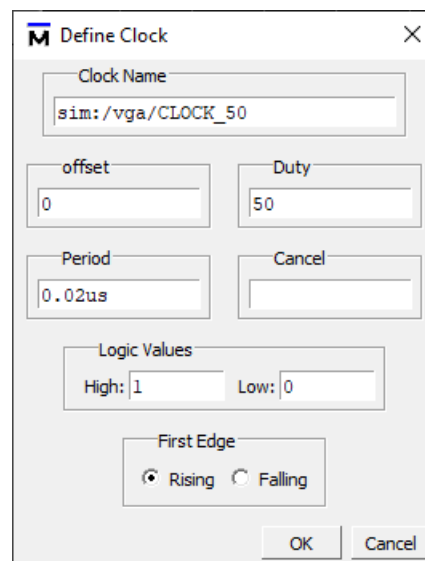


Figure 13: Define clock window.

Define the period, the frequency of the clock is 50MHz so the period will be $0.02\mu\text{s}$. Go to Simulate → Run → Run -All. Depending on your simulation and the performance of your computer, the simulation can take time to achieve. If the simulation take too long you can pause it by clicking Simulate → Break. You can see the results on the main window:

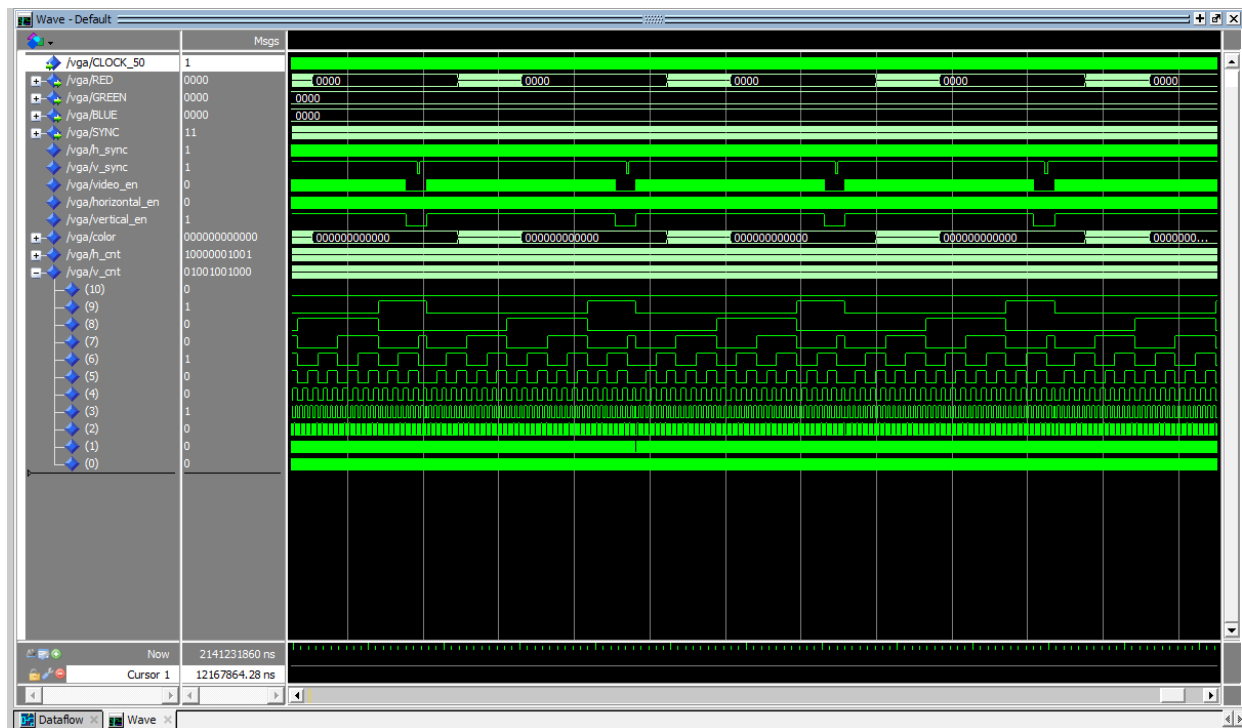


Figure 14: Results of a simulation.

4.3 Timing constraints analysis : *Timing Analyzer*

In some projects, it is important that timing deadlines are met. It will therefore be necessary to verify that the path delay between an entry and an exit is limited to a maximum value, or although the time difference between a valid output and the clock remains within a certain limit, or although the clock can be quite fast. Deadline analysis can also influence compilation: imposing very strict deadlines on certain paths will bring them together (to lower the propagation time), for example, I / O. With this in mind, *Quartus* provided the *Timing Analyzer* tool, allowing many analyzes to be made.

Within the framework of this laboratory, we will limit ourselves to imposing a maximum delay between the input clock and the availability of a correct value on our output. Compile your entire program. When finished, click on to Tools → Timing Analyzer. A new window appears:

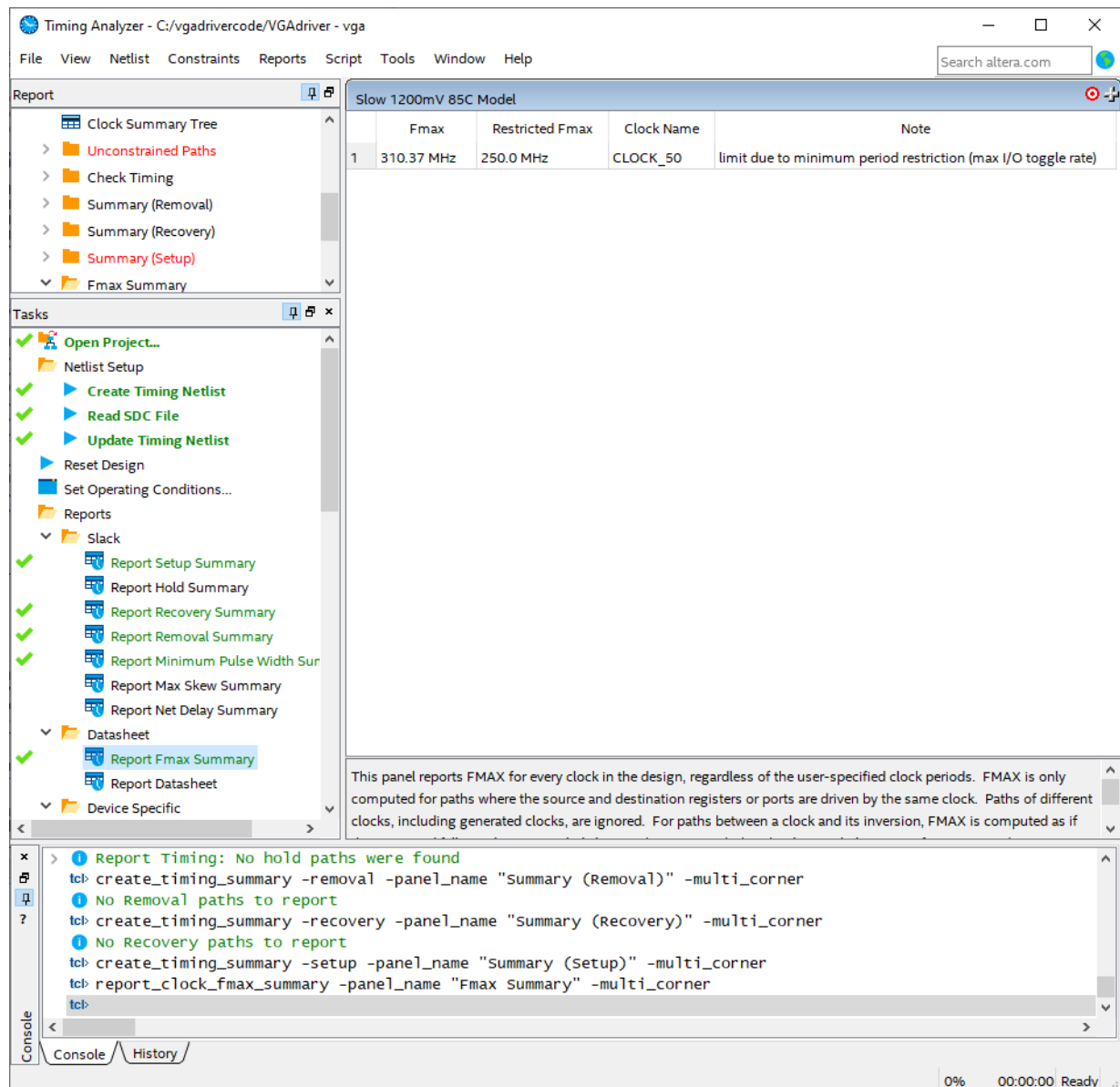
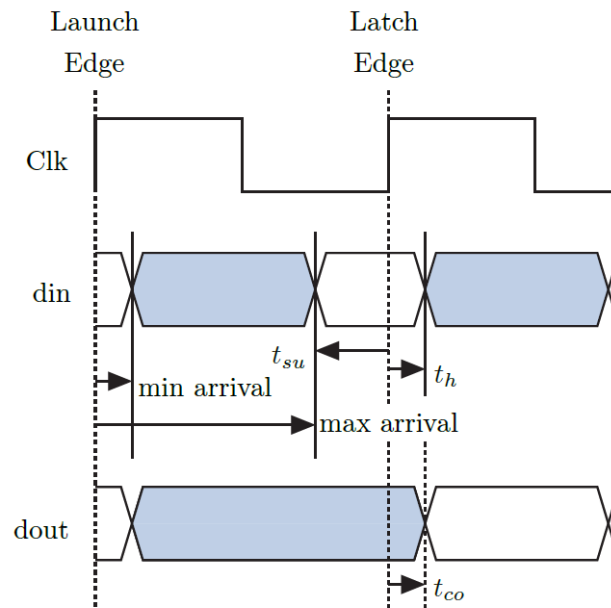


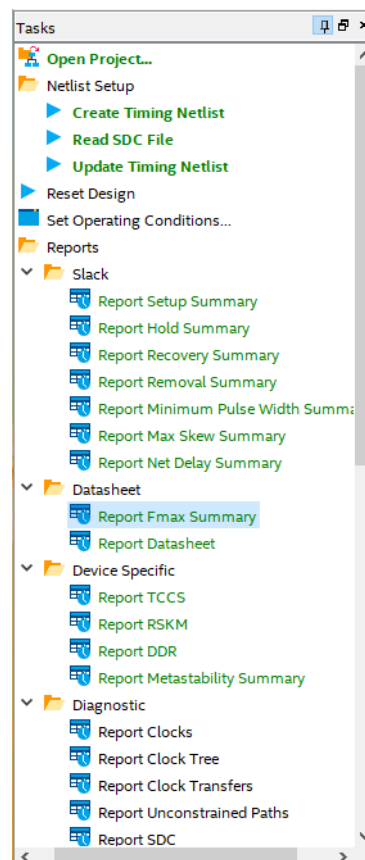
Figure 15: Timing analyzer main window.

Before setting the timing analyzer, a little reminder about timings in digital electronics. The different types of delays we will find are listed below:

- T_{su} : setup time. The minimum time during which the entry must be stable before the rising edge. This time delay constrains the latest arrival of a signal at the entry of a register, and so the longest combinatorial path.
- T_h : hold time. The minimum time during which the entry must be stable after the rising edge. This time constrains the arrival of the signal as quickly as possible at the entry of a register.
- T_{co} : clock to output. The time between rising edge and availability of output data.
- T_{pd} : propagation delay. Propagation time of a purely combinatorial logic path.

Figure 16: T_{su} , T_h and T_{co}

To start your timing analysis, you have to select the simulations conditions. First click on Netlist → Set Operating Conditions... and then select *MIN-fast-1200mv-0c*. Now to go Constraints → Create Clock... and Then go to the vertical task bar:

Figure 17: Task menu of the *Timing Analyzer*

Click on Create Timing Netlist, Read SDC File, then on Update Timing Netlist. You can

check the different reports below. Those reports can give you plenty of information:

- The maximum frequency you can use in your project.
- The net delays (useful for further length matching when making a PCB).
- Slack histogram (the difference in times between the expected arrival of a signal and the actual arrival of a signal).

During this lab session, you can play a bit with the Timing Analyzer

5 Analysis, placing and routing

5.1 Analysis

During this step, the software will analyze your VHDL file and translate it into logical elements implementable in the FPGA. To start the analysis, just click on Analysis Synthesis in the task window.

Once the file has been analyzed, a lot of information about what has been achieved is now available. You can access the compilation report, if it is not already open, with a right click on Analysis & Synthesis then view the Report:

The screenshot shows the 'Compilation Report - vga' window. The left pane displays a 'Table of Contents' with a tree structure. The right pane shows the 'Analysis & Synthesis Summary' with a search filter and a table of project details.

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Wed Feb 05 17:07:45 2020
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	vga
Top-level Entity Name	vga
Family	Cyclone IV E
Total logic elements	81
Total registers	33
Total pins	15
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0

Figure 18: Compilation report window.

This window shows important information:

- The number of I/O of your system.
- The number of logic elements used by your design.
- The total embedded memory used.

These two numbers must of course be compatible with the equipment you have! Otherwise you will have to modify your code to use fewer resources. You can also, by curiosity, see the other parts of the report, like the Resource Usage Summary.

Among the other information available, it is possible to see the physical implementation of the system. Go to Tools Netlist Viewers → RTL Viewer:

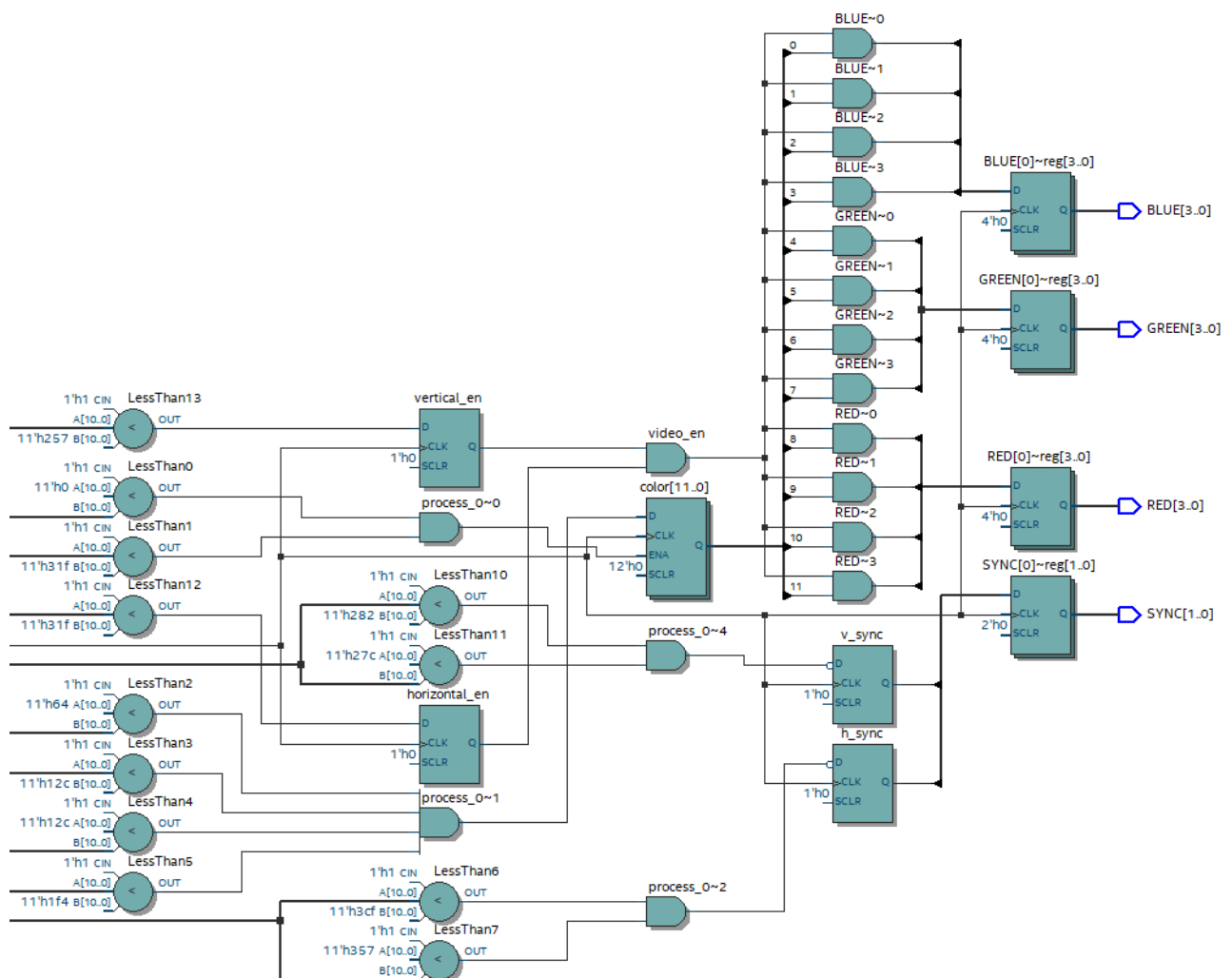


Figure 19: RTL schematics.

This schematic can be extremely useful to better understand the structure of a VHDL code! You can find which code line correspond to a RTL symbol.

5.2 Placing

The last step is to assign the different signals of your system to inputs/outputs components. Only certain FPGA I/O are accessible on the DE0. Therefore, you will have to make sure that all your signals are accessible! You will also have to tell the software what to do with unused

inputs/outputs, and finally indicate what voltage use.

In this project, there are two types of signals: classical signals, and clock signals. A signal is considered as a clock signal by *Quartus* automatically when you use the `rising_edge()` function, among other things. In addition, it is important to know that it can only have two clock signals per component. You must be able, from the code you have written, to determine which signal is a clock signal.

Now you will have to assign those signals to physical pins. Therefor, go to Assignments → Assignments Editor. A new window will appear:

Pin Planner - C:/Users/Hervé/Downloads/G-sensor/G-sensor/gsensor_vga - Gsensor_vga

File Edit View Processing Tools Window Help

Tasks

- Early Pin Planning
 - Early Pin Planning...
 - Run I/O Assignment Analysis
 - Export Pin Assignments...
- Highlight Pins
 - I/O Banks
 - VREF Groups
 - Edges
- Clock Pins
 - Clock
 - PLL/DLL Input
 - PLL/DLL Output
 - Clock Region Input
- Memory Pins
 - DQ/DQS
 - Differential Pins
 - OCT Pins

Top View - Wire Bond
Cyclone IV E - EP4CE22F17C6

Pin Legend

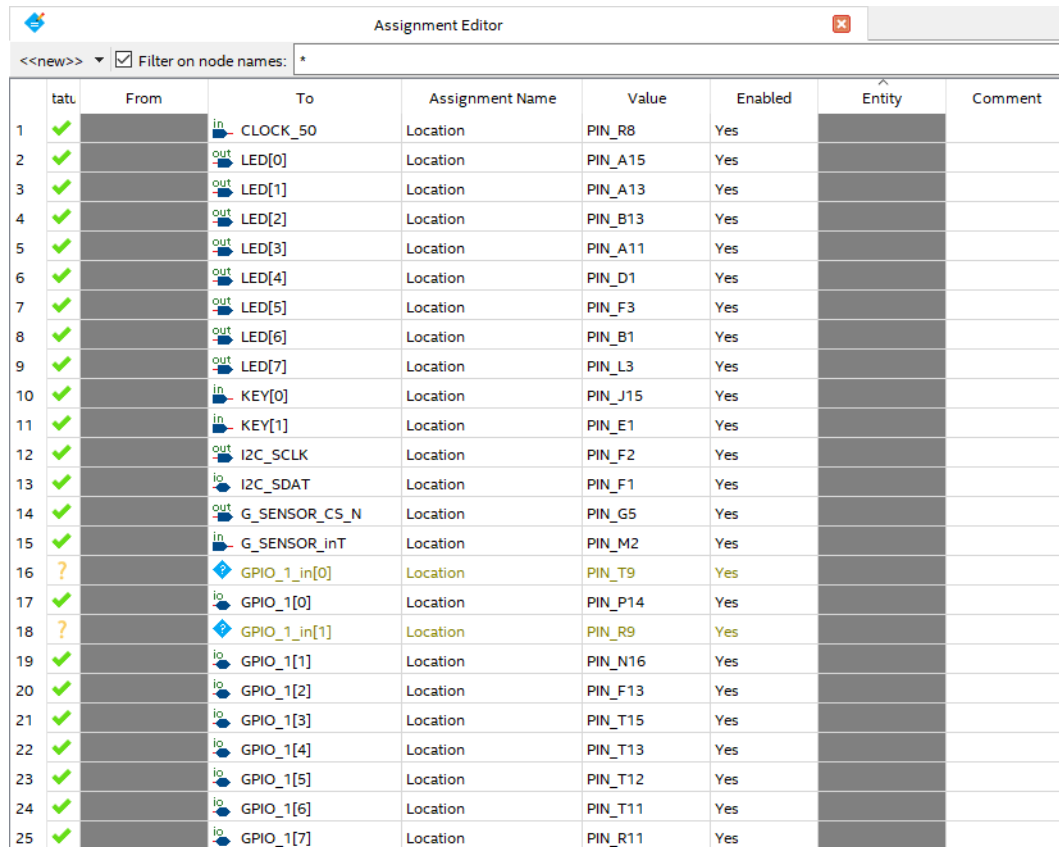
- Symbol Pin Type
- User I/O
- User assigned I...
- Fitter assigned I...
- Unbonded pad
- Reserved pin
- Other configura...
- DEV_OE
- DEV_CLR
- DIFF_n
- DIFF_p
- DQ
- DQS
- CLK_n
- CLK_p
- Other PLL
- Other dual purp...
- MSEL0
- MSEL1
- MSEL2
- CONF_DONE
- nCE
- nCONFIG
- TDI
- TCK
- TMS
- TDO
- nSTATUS

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential
CLOCK_50	Input	PIN_R8	3	B3_NO	PIN_R8	3.3-V LVTTTL		8mA (default)		
G_SENSOR_CS_N	Output	PIN_G5	1	B1_NO	PIN_G5	3.3-V LVTTTL		8mA (default)	2 (default)	
G_SENSOR_inT	Input	PIN_M2	2	B2_NO	PIN_M2	3.3-V LVTTTL		8mA (default)		
GPIO_1[33]	Bidir				PIN_L7	3.3-V LVTTTL		8mA (default)	2 (default)	
GPIO_1[32]	Bidir				PIN_B11	3.3-V LVTTTL		8mA (default)	2 (default)	
GPIO_1[31]	Bidir				PIN_R16	3.3-V LVTTTL		8mA (default)	2 (default)	
GPIO_1[30]	Bidir				PIN_P15	3.3-V LVTTTL		8mA (default)	2 (default)	
GPIO_1[29]	Bidir				PIN_P6	3.3-V LVTTTL		8mA (default)	2 (default)	
GPIO_1[28]	Bidir				PIN_R7	3.3-V LVTTTL		8mA (default)	2 (default)	
GPIO_1[27]	Bidir				PIN_T5	3.3-V LVTTTL		8mA (default)	2 (default)	
GPIO_1[26]	Bidir				PIN_M8	3.3-V LVTTTL		8mA (default)	2 (default)	
GPIO_1[25]	Bidir				PIN_E8	3.3-V LVTTTL		8mA (default)	2 (default)	
GPIO_1[24]	Bidir				PIN_A4	3.3-V LVTTTL		8mA (default)	2 (default)	

Figure 20: Pin planner window.

You can browse this window in order to discover all of it's features! At the middle, you can see the FPGA pinout. Below, you can assign each of your designs signals to a pin of the FPGA but before check the schematics of the DE0 and VGA board!

If you go to Assignments → Assignments Editor, a new window opens:



	tatu	From	To	Assignment Name	Value	Enabled	Entity	Comment
1	✓		in CLOCK_50	Location	PIN_R8	Yes		
2	✓		out LED[0]	Location	PIN_A15	Yes		
3	✓		out LED[1]	Location	PIN_A13	Yes		
4	✓		out LED[2]	Location	PIN_B13	Yes		
5	✓		out LED[3]	Location	PIN_A11	Yes		
6	✓		out LED[4]	Location	PIN_D1	Yes		
7	✓		out LED[5]	Location	PIN_F3	Yes		
8	✓		out LED[6]	Location	PIN_B1	Yes		
9	✓		out LED[7]	Location	PIN_L3	Yes		
10	✓		in KEY[0]	Location	PIN_J15	Yes		
11	✓		in KEY[1]	Location	PIN_E1	Yes		
12	✓		out I2C_SCLK	Location	PIN_F2	Yes		
13	✓		io I2C_SDAT	Location	PIN_F1	Yes		
14	✓		out G_SENSOR_CS_N	Location	PIN_G5	Yes		
15	✓		in G_SENSOR_inT	Location	PIN_M2	Yes		
16	?		io GPIO_1_in[0]	Location	PIN_T9	Yes		
17	✓		io GPIO_1[0]	Location	PIN_P14	Yes		
18	?		io GPIO_1_in[1]	Location	PIN_R9	Yes		
19	✓		io GPIO_1[1]	Location	PIN_N16	Yes		
20	✓		io GPIO_1[2]	Location	PIN_F13	Yes		
21	✓		io GPIO_1[3]	Location	PIN_T15	Yes		
22	✓		io GPIO_1[4]	Location	PIN_T13	Yes		
23	✓		io GPIO_1[5]	Location	PIN_T12	Yes		
24	✓		io GPIO_1[6]	Location	PIN_T11	Yes		
25	✓		io GPIO_1[7]	Location	PIN_R11	Yes		

Figure 21: Signal list as seen in the pin planner window.

The assignment editor is an other method of pin planning.

In order to choose correctly your pins, please check the DE0 nano manual provided. You will see figure like this:

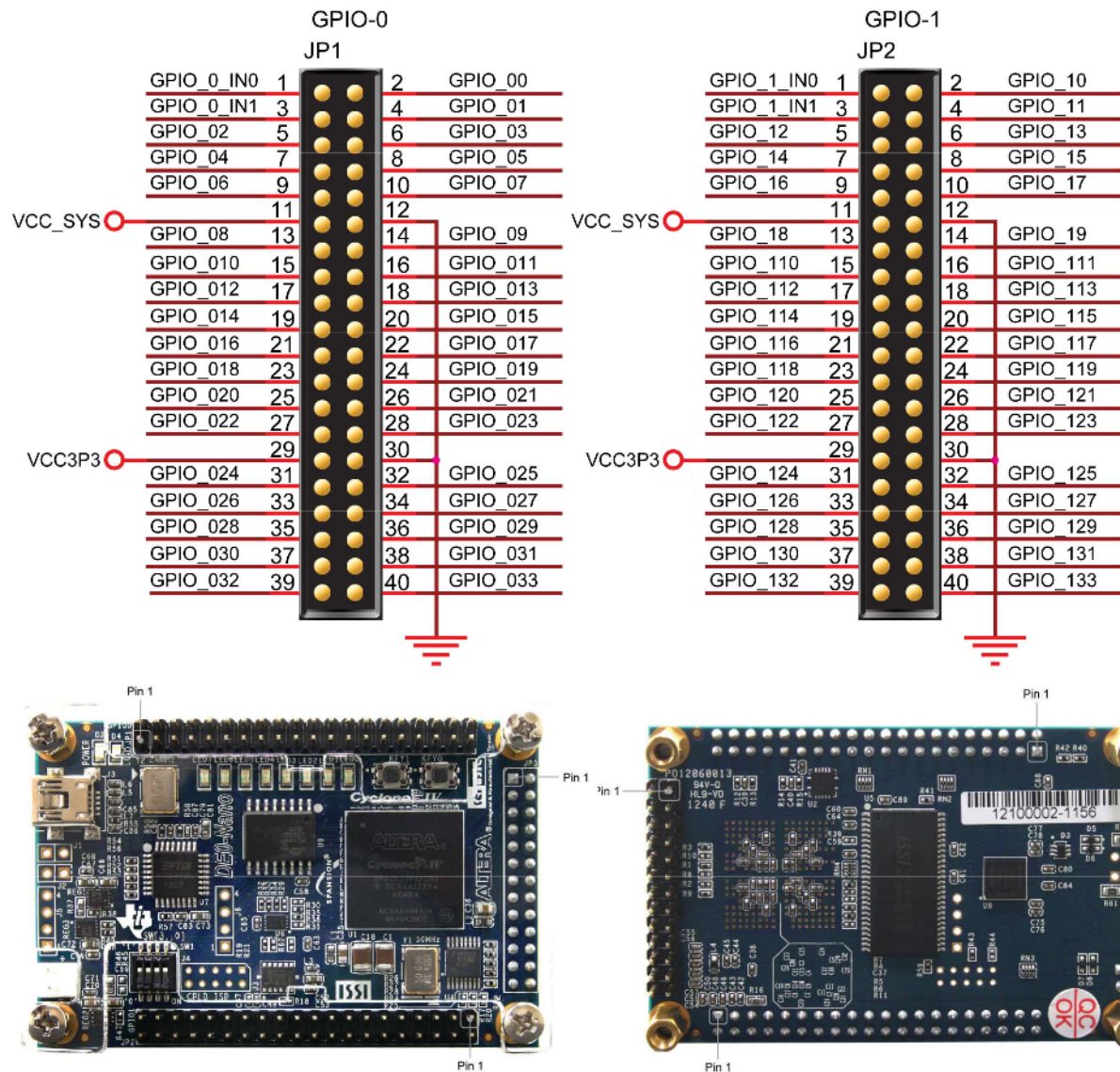


Figure 3-9 Pin1 locations of the GPIO expansion headers

Table 3-6 GPIO-0 Pin Assignments

Signal Name	FPGA Pin No.	Description	I/O Standard
GPIO_0_IN0	PIN_A8	GPIO Connection DATA	3.3V
GPIO_00	PIN_D3	GPIO Connection DATA	3.3V
GPIO_0_IN1	PIN_B8	GPIO Connection DATA	3.3V
GPIO_01	PIN_C3	GPIO Connection DATA	3.3V

Figure 22: DE0-nano pin assignment.

This figure shows the link between the DE0 nano GPIO and the FPGA pins used in the pin planner. Also, do not forget to check on the VGA board schematics which color channel is connected to which pin of the DE0 nano and so to which pin of the FPGA.

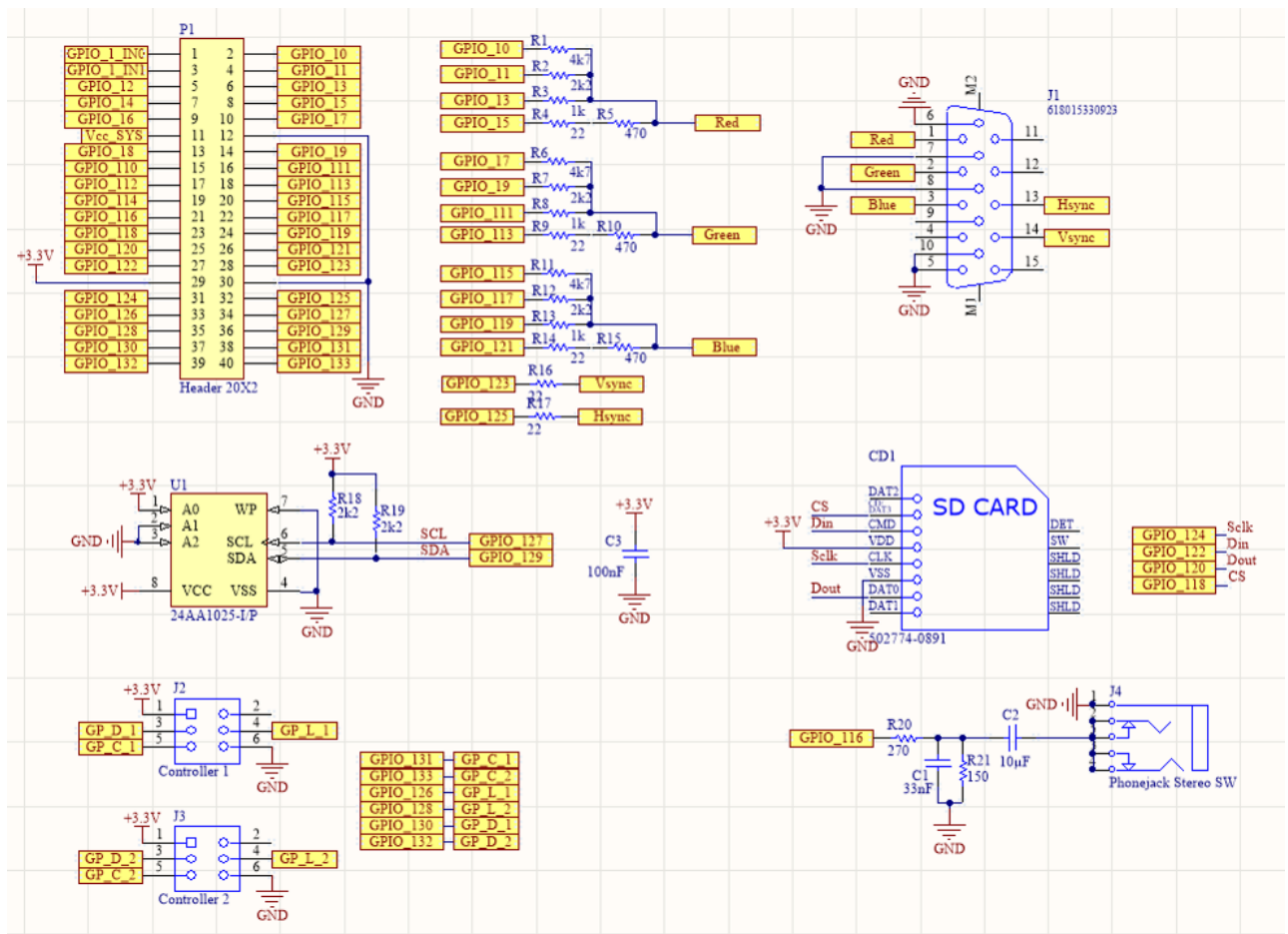


Figure 23: Custom made VGA daughter board.

Simple review of the VGA schematics:

- The labels of the P1 header fit the DE0 nano GPIO pins names.
- The I²C EEPROM can be removed from its socket, allowing the user to program it externally.
- J2 and J3 are the SNES gamepad connectors:
 - GPx-C: the clock input
 - GPx-L: the latch input
 - GPx-D: the data output
- The sound output is a simple line output without amplifier

5.3 Planning

After synthesizing, you can see the planning of your design. Click on Tools → Chip Planner, a new window appears:

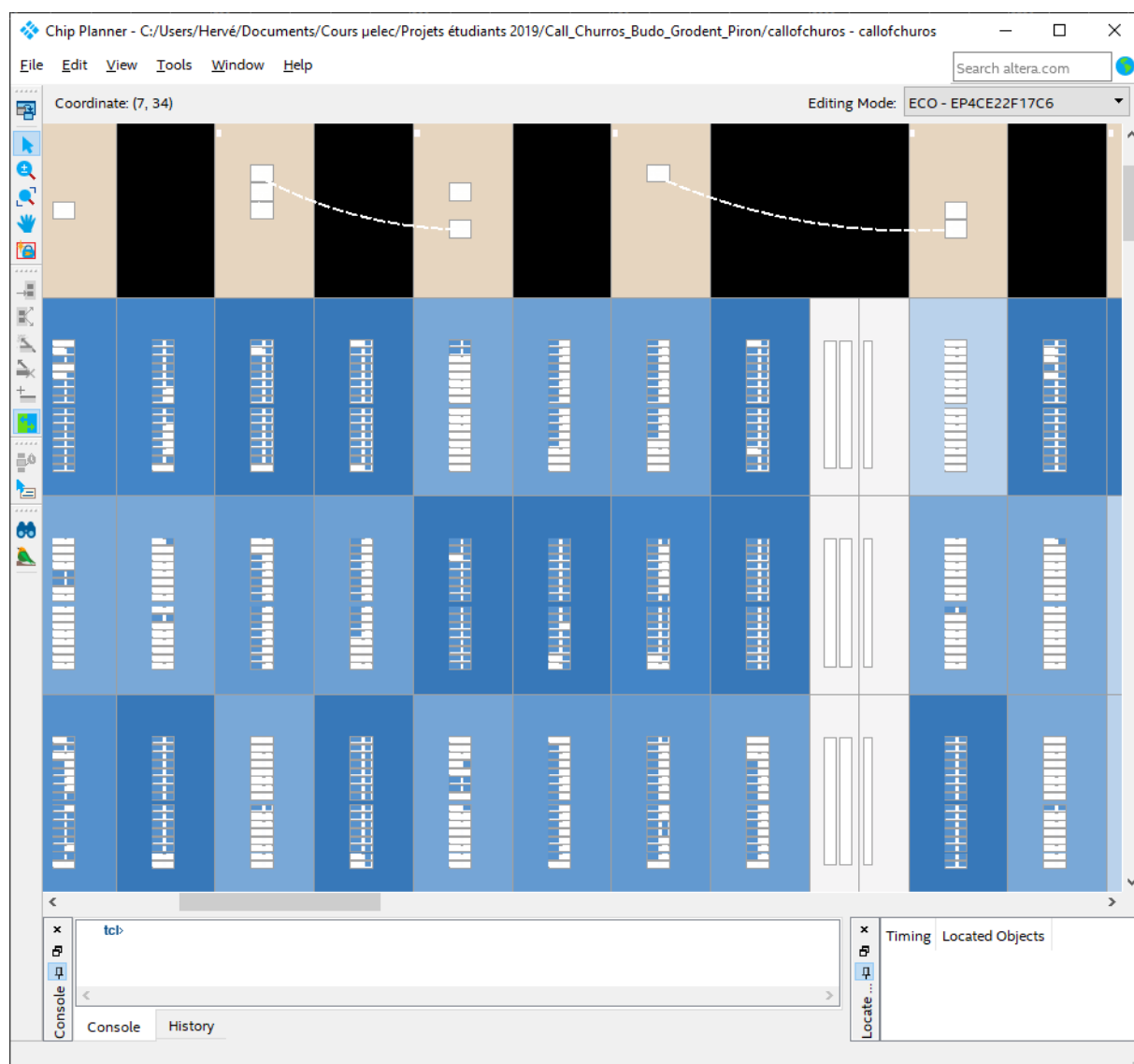


Figure 24

This feature is more used for ASIC design. The darker the block, the more it is used. You can move around in the program to see exactly how your system was implemented. You can for example see logic equations, registers, etc. Double-click on a block to open a new window with the details of the block. For example :

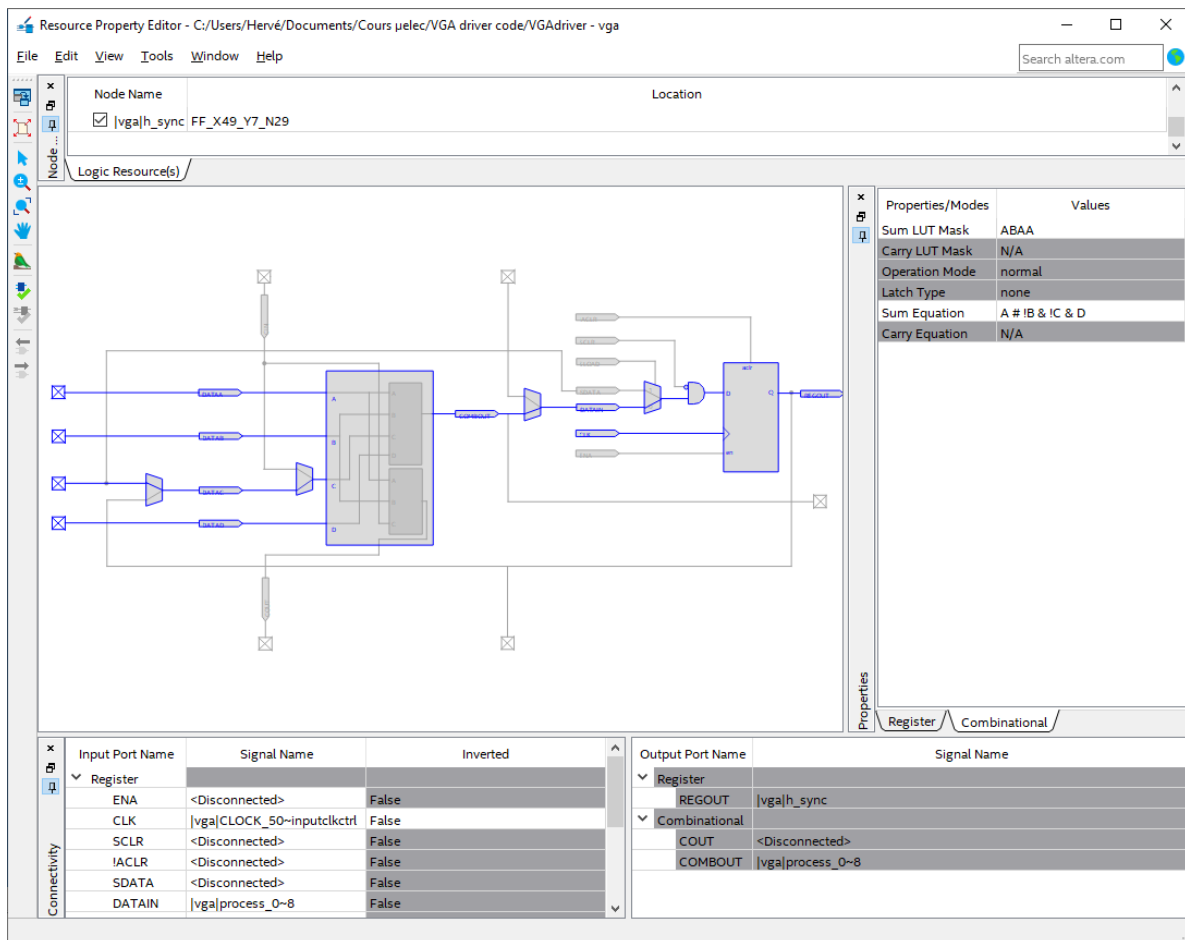


Figure 25: Ressource property editor window.

You can locate any element by right clicking and choosing Locate → Locate in Design File.

5.4 Programming the FPGA

The very last step is to program the FPGA. To do so, connect the programmer to your PC. On Windows, the installation of the drivers is not necessarily automatic: you will have to do this manually, by going to your device manager, and indicating the location of the drivers. These are in the Quartus/drivers directory of your installation of *Quartus*.

Open the programmer by double-clicking on Programmer, in the Tools part of *Quartus*. A new window opens:

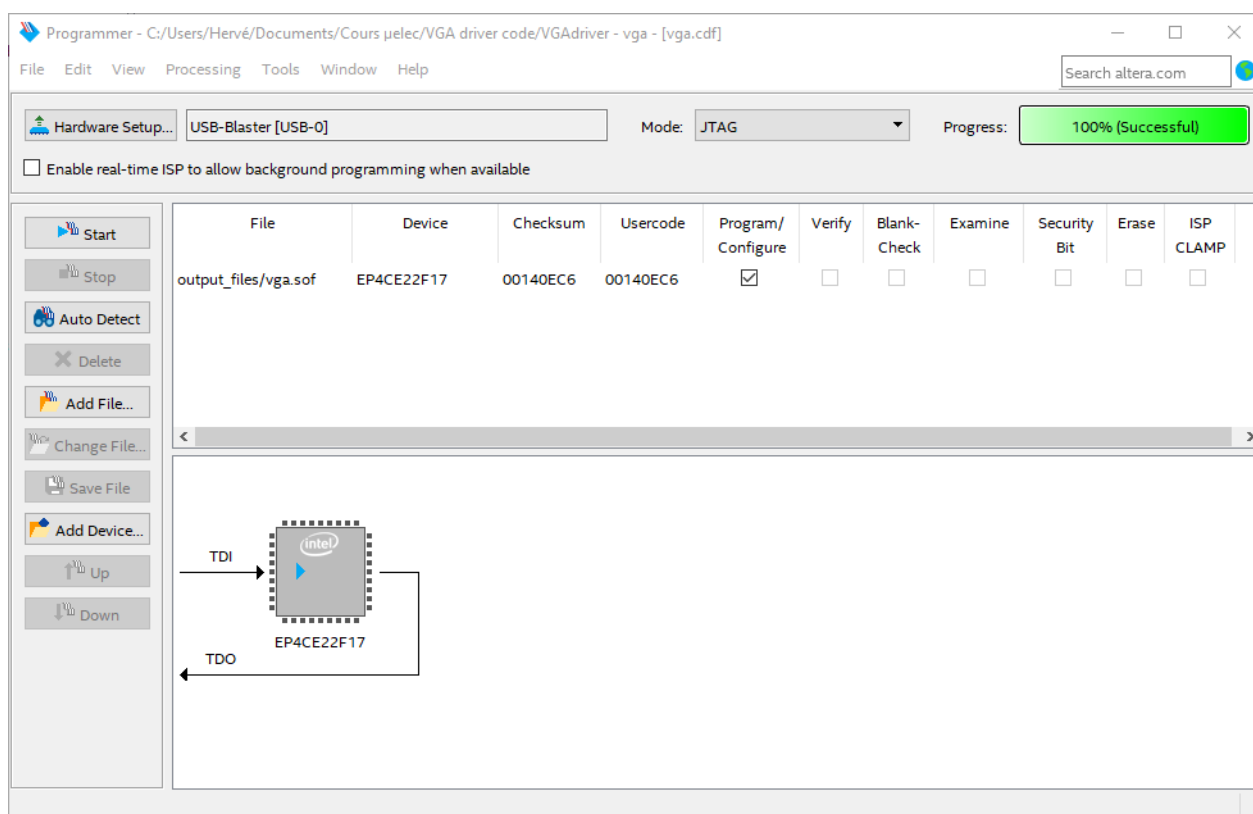


Figure 26: Programmer main window

Go to Hardware Setup, the following window will appear:

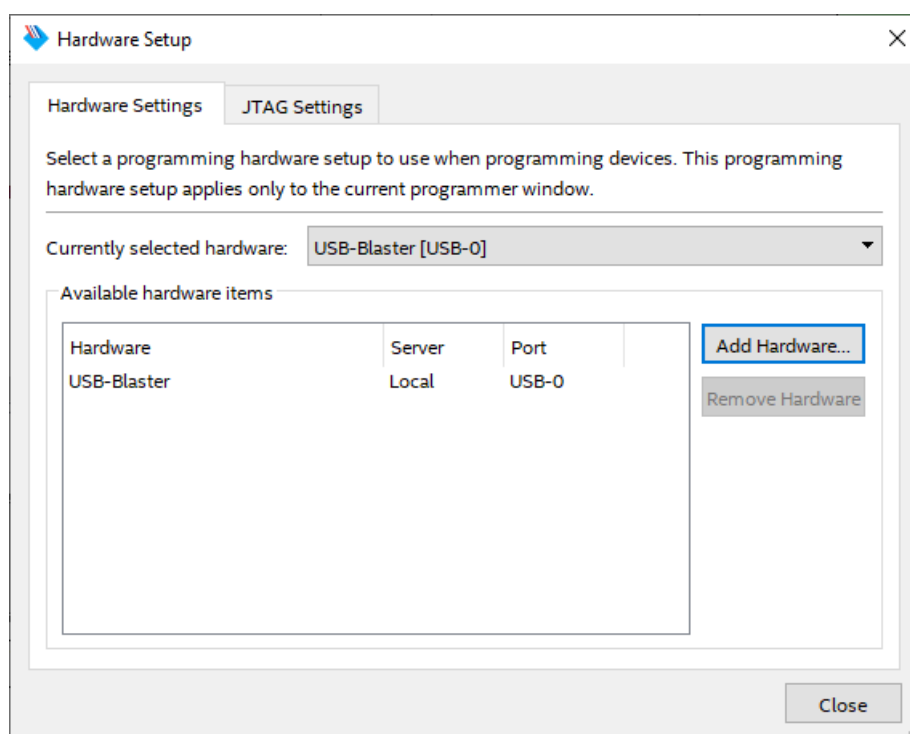


Figure 27: Hardware setup window.

Click on Add Hardware... and select your programmer (embedded onto the DE0-nano

board). If you cannot see your device, check your drivers. If you are using Linux, pay attention to run *Quartus* as a superuser.

Go to the current project folder, select the output folder, then choose the .sof (SRAM Object File). You can also use the Auto Detect button, then click on Start. Now, you successfully programmed the FPGA. If the VGA board is connected to the DE0-nano, you should see a square on the screen. At this step, the FPGA will lose its configuration when the power is down. Fortunately, the development board contains a flash memory, which therefore allows the FPGA to load its configuration at startup. However, some settings must be made to use this memory.

You will have to convert your output file to a .jic (JTAG Indirect Configuration File) file. In order to do so, click on Files → Convert Programming Files. A new window will open:

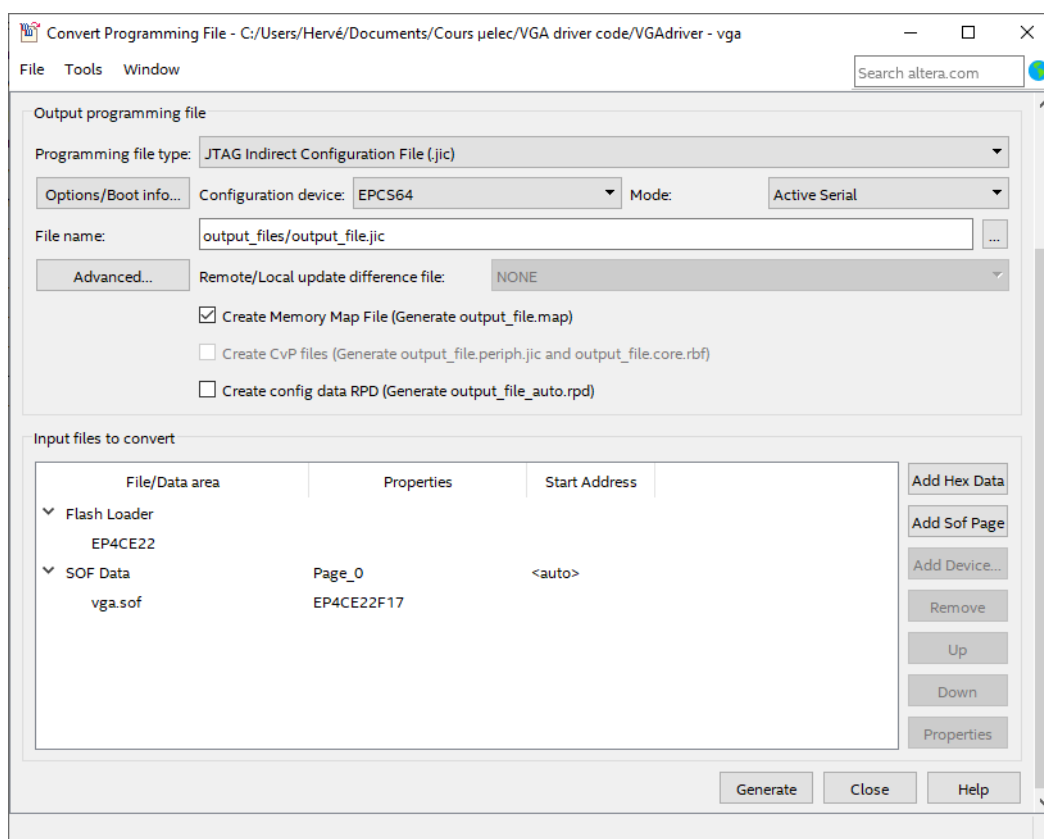


Figure 28: Programming file conversion window.

Note: EPCS stands for Enhanced Programmable Configuration Single (single refers to one output).

In Programming file type, choose JTAG Indirect Configuration File (.jic). Then, for configuration device, choose EPCS64. Once configured, click in the white part at the bottom of the window on Flash Loader, then on the right buttons, do Add Device. In the new window, choose the FPGA:

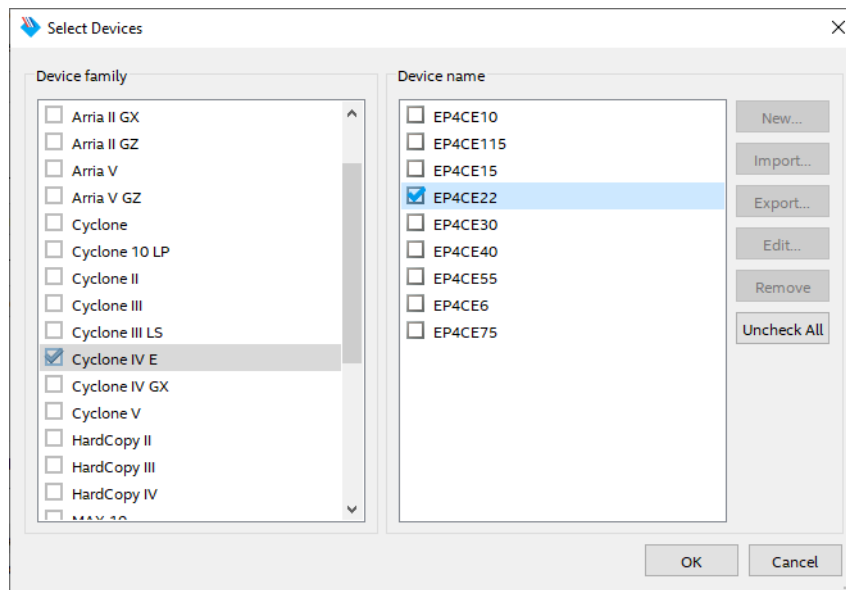


Figure 29: Device selection window.

First select Cyclone IV E, then EP4CE22. You will return on the programming file conversion window, then click on Add File:

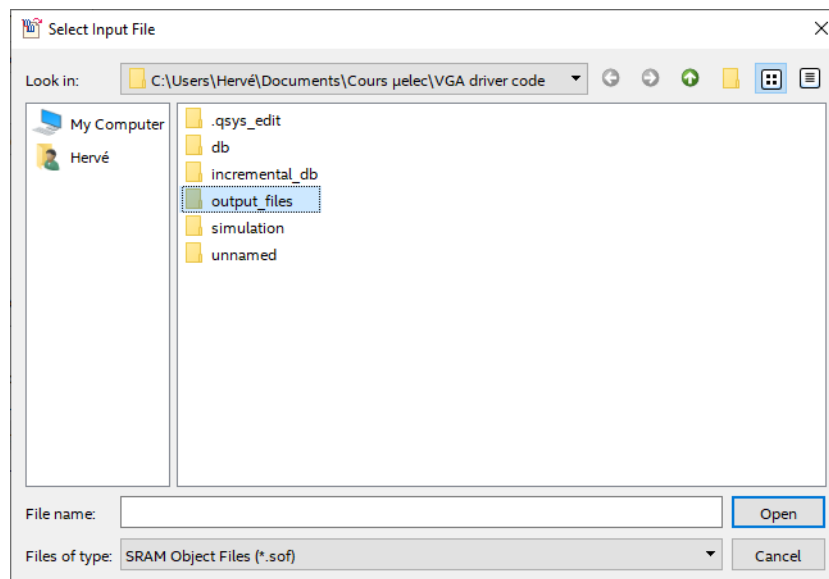


Figure 30: Input file selection window.

Go to the current project folder, select the output folder, then choose the .sof file. You will return on the programming file conversion window, then click on Generate.

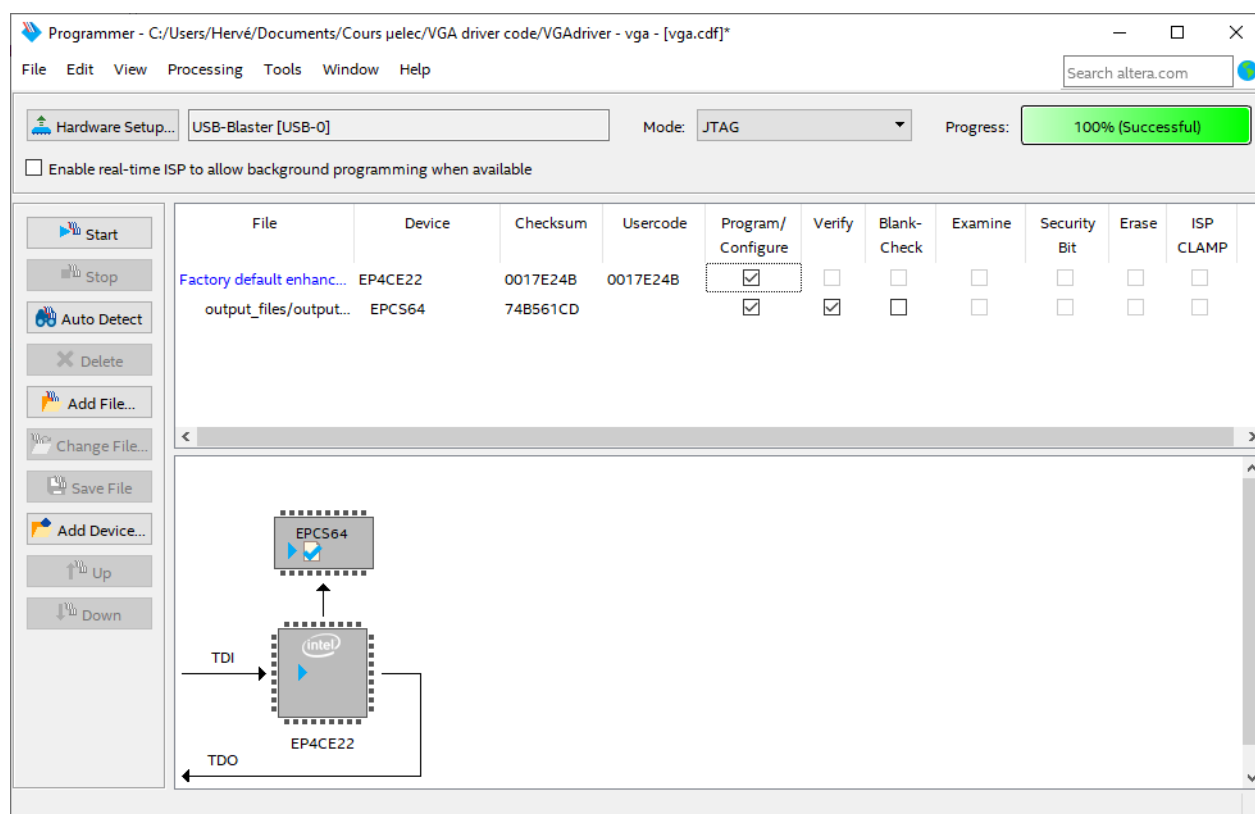


Figure 31: Programmer main window.

Reprogram the system. This time, the task should be a little longer. Unlike the previous times, the program should not start automatically. Remove the USB plug of the card and reconnect: your square appears!

6 Debugging a FPGA/VHDL project

6.1 Introduction

Sometimes it is very difficult to create a simulation, if not impossible. A frequent case is when using external devices, such as the accelerometer or the gamepad. The embedded ADXL345 accelerometer of the DE0-nano uses the SPI protocol, and therefore generate valid signals to simulate its responses via a TestBench is particularly delicate.

Another possible case is when using a third party code. This code is sometimes little not well commented (or not at all), or follows a logic that seems specific to the programmer. In this last part, we will therefore see how to analyze the signals generated in the FPGA after programming it, without using an oscilloscope or a logic analyzer. So it will be possible to observe each signal of a program, even if you have no idea at first glance of the usefulness of these signals.

6.2 Analysis of the signals generated within the FPGA: discovering *Signal Tap Logic Analyzer*

To carry out this task, *Quartus* has a very practical tool: *Signal Tap Logic Analyzer*. This tool uses the on-board SRAM memory of the FPGA to record the signals generated internally, then retransmits them on the USB port via the JTAG and displays them. It is also possible to define certain triggers to trigger only at times wanted. Of course, all of this comes at the cost

of an overhead on your program.

Go to Tools, then click on Signal Tap Logic Analyzer. A new window appears:

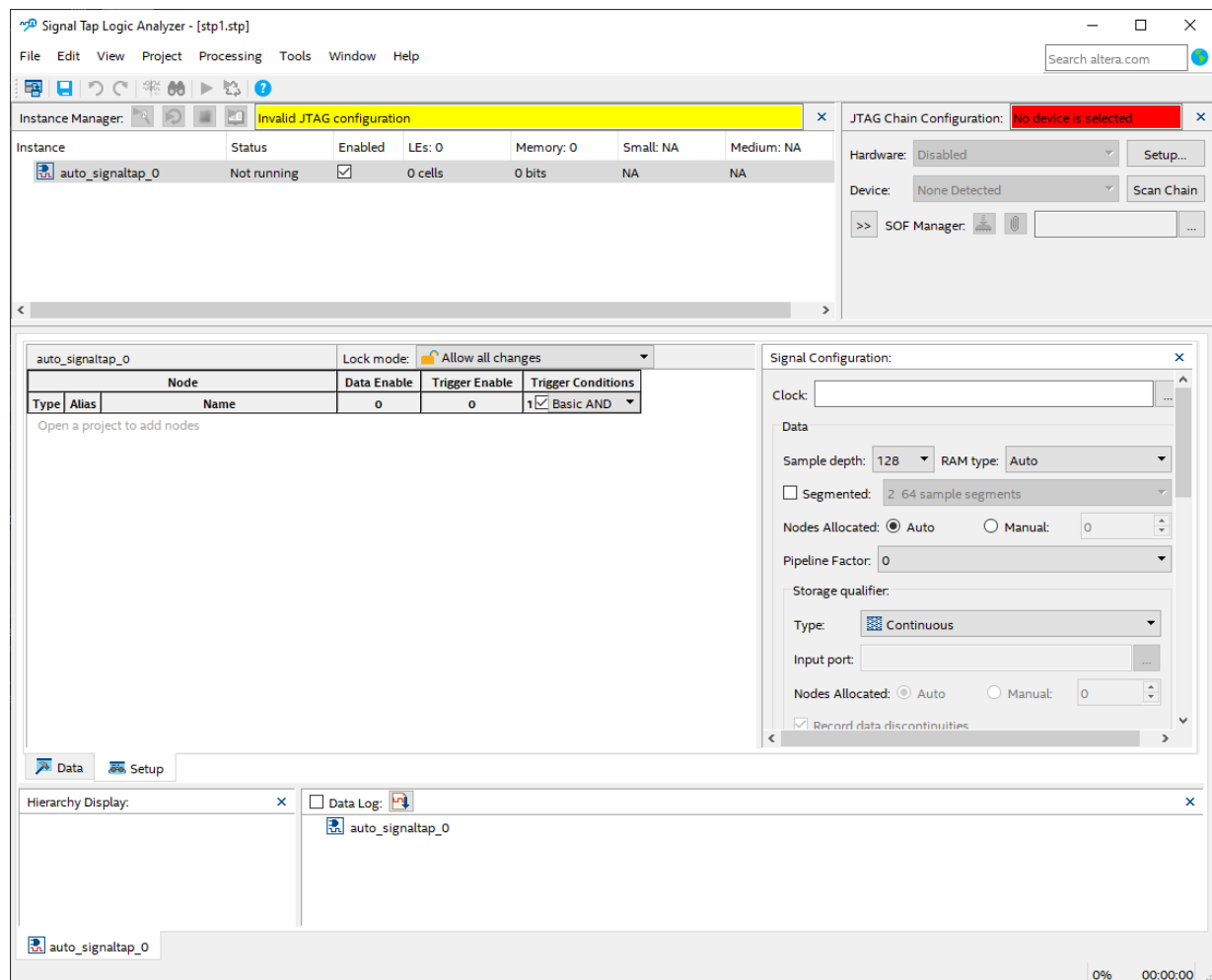


Figure 32: Global view of the *Signal Tap Logic Analyzer* interface.

The first thing to do is configure the programmer. In the right part of the window, click on Setup, and configure your USB-Blaster as when programming the component. You should see a message with a yellow background "Invalid JTAG configuration", indicating that the component is not properly configured. This is normal since we have not yet loaded any code compatible with *Signal Tap*.

In the upper white part of the window, right click, then Create Instance:

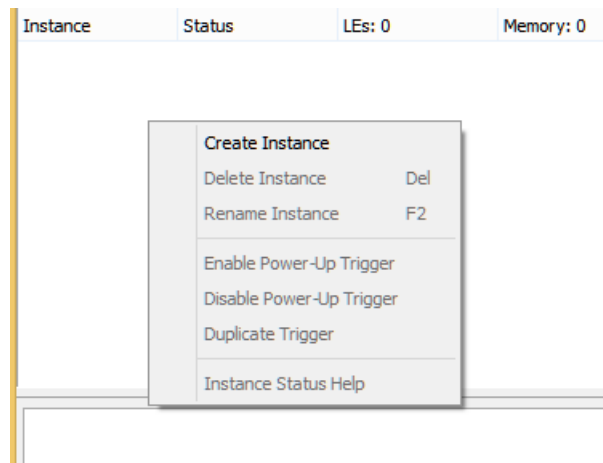


Figure 33: Creating a instance.

The window then changes and an instance is added. In Signal Configuration, configure the clock and indicate a Sample depth of 64K. If your code is too large choose a smaller Sample depth.

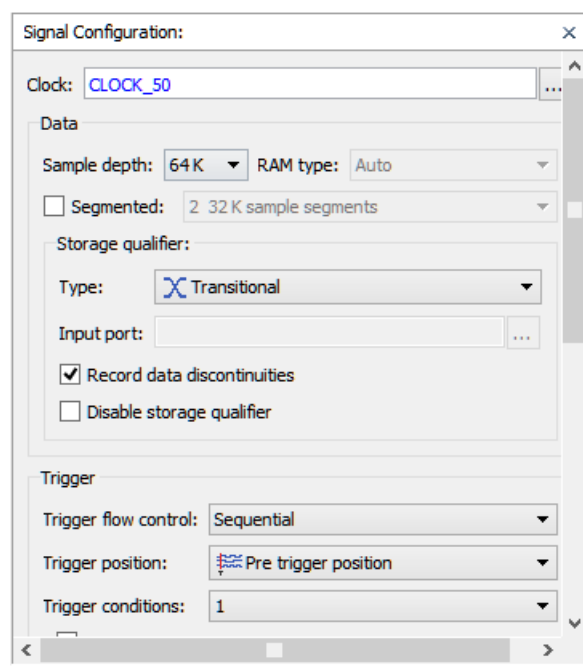


Figure 34: Setting up of the analysis.

Finally, on the left side of the window, right click to add signals, and add the following signals:

auto_sigtap_0			Lock mode: Allow all changes				
Node			Data Enable	Trigger Enable	Storage Enable	Storage Qualifier	Trigger Conditions
Type	Alias	Name	6	2	6	Transitional	1 Basic OR
		v_sync	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		red_signal	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		h_sync	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		video_en	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		vertical_en	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
		horizontal_en	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

Figure 35: Adding the signals to observe.

The more signals you add to observe, the higher the memory requirements, and therefore the more it will be necessary to decrease the Sample Depth. It is also possible to use a wide variety of triggers by clicking on the Trigger Conditions column. In this case, we will use a Basic OR, on the rising edge of the signal v'sync and h'sync.

Finally, still in the same window, select your instance, compile your program as usual. You will notice that it has increased in size in terms of resources used. Program the component without, however, programming the Flash, so that the program starts automatically, and return to *SignalTap*.

Press the button (the one with the triangle and the magnifying glass) to start the acquisition, which should give you something like this:

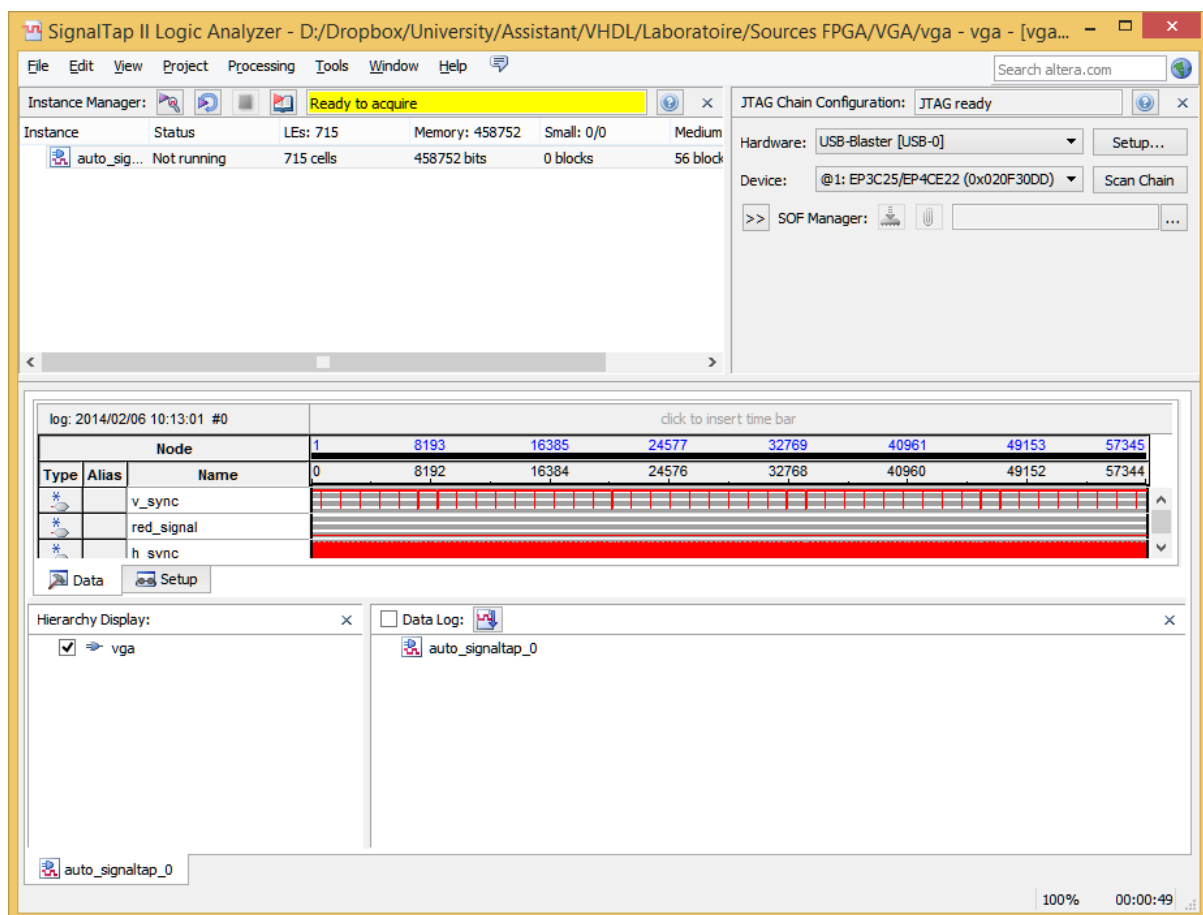


Figure 36: Displaying the results.

All of the indicated signals were recorded during execution and then transferred to the software. You can now zoom in and analyze the behavior of your system in detail.

6.3 Third party code analysis

Download the Gsensor project available on the website. Thanks to all the tools seen throughout this laboratory, try to understand how this project works. What is he doing ? How does he do it? What are the important variables? In particular, put comments back in the code. Start by displaying the RTL viewer to see the nesting of the different parts of the code. Next, read the datasheet of the card's accelerometer, an ADXL345. try to understand the code and then, through *Signal Tap Logic Analyzer*, verify that what you have understood corresponds to the observations: change a few pieces of code, and observe the reactions.

You don't have to time to do this during the lab session. If you get bored at home, you can do it.

7 Exercices

7.1 Modifying the accerelerator code

You have to modify the gsensor-vga project in the Gsensor-VGA folder in order to make your square move following the accelerometer. You will have to add the 12 bits color output and assign the different pins.

7.2 Moving a square using a SNES gamepad

Use the SNES controller direction pad in order to move the square.

7.2.1 SNES gamepad communication protocol

The SNES gamepad is based on two 74LS165 parallel 8 bit shift registers (simple parallel to serial shifter). It can serialize up to 16bits. The period of the clock signal is $12\mu s$. In order to communicate with the controller, the SNES sends a $12\mu s$ positive impulse on the latch. This latch indicates that the SNES will start sending 16 data clock periods to the gamepad. Each of the 16 data clock periods represent a different button, and the SNES will expect the output signal to toggle on those data clock periods which indicates a button on the gamepad is pressed.

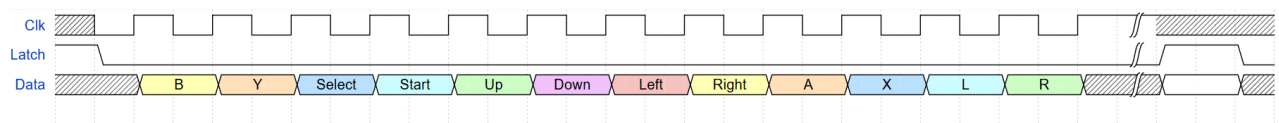


Figure 37: Timing diagram of the SNES gamepad communication protocol.

There are 12 buttons on the SNES gamepad, but the SNES sends 16 data clocks periods. Only the first 12 data clock periods are used, the following 4 are ignored by the SNES. The protocol was first designed for the NES console which had less buttons. *Nintendo* added new buttons on the SNES but didn't change the communication protocol as it could support more buttons. Potentially, this protocol could support up to 16 different buttons.