HWP01

Programmable Hardware

Introduction Quartus II / ModelSim / Logic design with VHDL

Lab Work Handbook

Version History

Version	Date	Name	Modification
2.5	25/08/2020	LogEH	- Final assignments updated for 2020-2021
2.4	25/09/2019	LogEH	- New final assignments added, now 22 in total
2.3	10/09/2019	LogEH	- Several minor corrections
2.2e	27/08/2019	Roelj	- Updated for Quartus 18.0
2.2d	23/08/2017	Roelj	- Updated for Quartus 17.0
2.2	26/08/2016	WitRe	- Changed some information for 2016 labs.
2.1	27/08/2015	PelJH	- Changed tutorial to work with DE1-SoC kit instead of DE2.
2.0	29/08/2012	PelJH	- Changed the whole tutorial to first create projects with ModelSim and afterwards import that in Quartus.
1.2	29/11/2010	PelJH	- Added appendix A and Components to further reading. - Minor corrections after student feedback
1.1	22/11/2010	PelJH	- Minor corrections after student feedback - Added another final assignment.
1.0	13/09/2010	PelJH	- Initial version of HWP new style (now in English). - Translated some parts from the old document by LogEH into English. - Added ModelSim instead of waveform vectors. - Now using Quartus 10.1.

List of Abbreviations

ASIC	Application Specific Integrated Circuit
CPLD	Complex Programmable Logic Device
DC	Direct Current
DSP	Digital Signal Processor
EDA	Electronic Design Automation
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
PAR	Place And Route
PLD	Programmable Logic Device
RTL	Register-Transfer Level
SystemC	A HDL with a C-like syntax
TCL	Tool Command Language
USB	Universal Serial Bus
Verilog	A HDL language like VHDL
VHDL	VHSIC (Very High Speed Integrated Circuit Hardware Description Language)

Contents

1. Introduction

In the modern world of electronics, everything must be fast; the time-to-market must be short and performance must be high. Recently, the PLD (programmable logic device) market has grown tremendously (turnover doubles about every five years) because these devices have become more and more adapt to meet these criteria in many fields of electronics.

FPGAs (field-programmable gate arrays) and other reconfigurable hardware in general are becoming more popular every day. If you have a high-performance application without high power requirements but you don't have money or time to make an ASIC (application specific integrated circuit), reconfigurable hardware is often a good choice. The market for microcontrollers with a small piece of reconfigurable hardware embedded within the chip is growing as well.

HDLs (hardware description languages) capture the functionality of digital schematics in plain text. **In short; a HDL is used to "draw" digital logic with text.**
The most widely-used HDLs today are VHDL and Verilog, but there are many others (System C for example). In practise, HDLs are used to design full-scale production ICs, ASICs and especially FPGA configurations.

The power of HDLs is the high level of abstraction. Developing applications with HDLs and letting tools compile and place them in your PLD is faster than drawing schematics by hand. Next to that, the HDL code can often be reused in newer, bigger PLDs and is easily scalable. For example; a well coded 16-bit microprocessor core (captured in HDL) may be changed into a 32-bit microprocessor core by adjusting a few variables only.

The goal of this course is an introduction to VHDL, FPGAs and PLDs in general. The functional designs made by students are implemented into an FPGA that resides on the Altera DE2 (Development and Education) kit.

This document contains all the assignments for this course. The first assignment is a step by step introduction to designing logic with Quartus and simulating with ModelSim. The other assignments are only described briefly, with some tips & tricks added.

Good luck!

2. Purpose and prerequisites

The purpose of the first introductory assignment is as follows:

- To create a simple functional design with basic logic for a digital system consisting of LEDs and switches
- To simulate a digital system and thus verify its functionality with ModelSim
- To introduce you to working with the Quartus Prime software tool
- To reconfigure an FPGA with a digital schematic

The perquisites of this assignment are:

- The Altera DE1-SoC Development & Education Board is available.
- A 9V DC power supply
- USB-Cable to connect the USB Blaster to a computer
- Altera Quartus Prime (v18.1) Lite edition software has been installed. The v18.1 Lite edition does not need a license, so use this one! If you want to download the installation files, you do need to register. Alternatively, if you are on a HR PC, install Quartus using the Liquit Workspace.
- Mentor Graphics ModelSim Altera Starter Edition has been installed
- The USB Blaster driver software has been installed

Typically the USB Blaster driver software is installed when installing Altera Quartus, but if this is not the case: connect the DE1-SoC board to the computer using the USB cable. Use the USB cable next to the red on-off switch. A wizard will pop up to install a driver for the USB Blaster. Follow Figure 1 and Figure 2. The directory where the drivers can be found is **C:\intelFPGALite\18.1\quartus\drivers\usb-blaster** . Change the driver path according to your own installation settings if necessary.

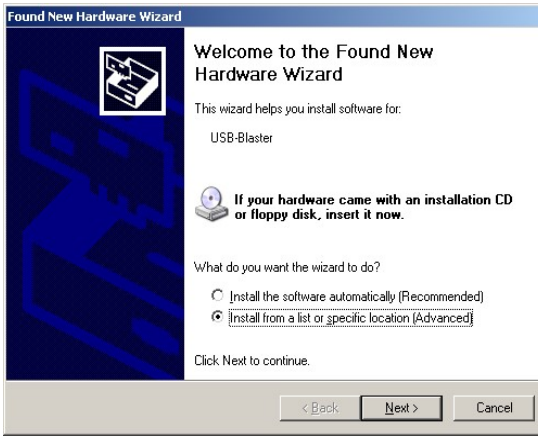


Figure 1

Installing the USB Blaster Driver (a)

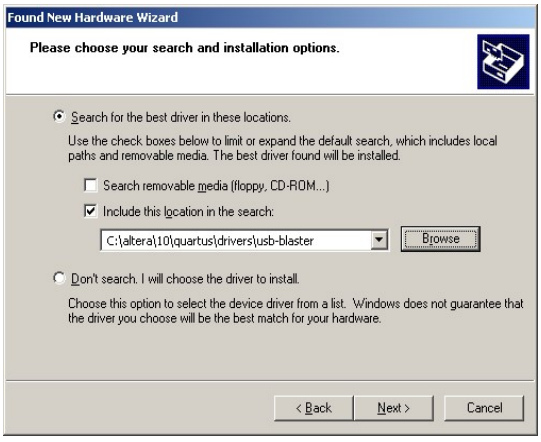


Figure 2

Installing the USB Blaster Driver (b)

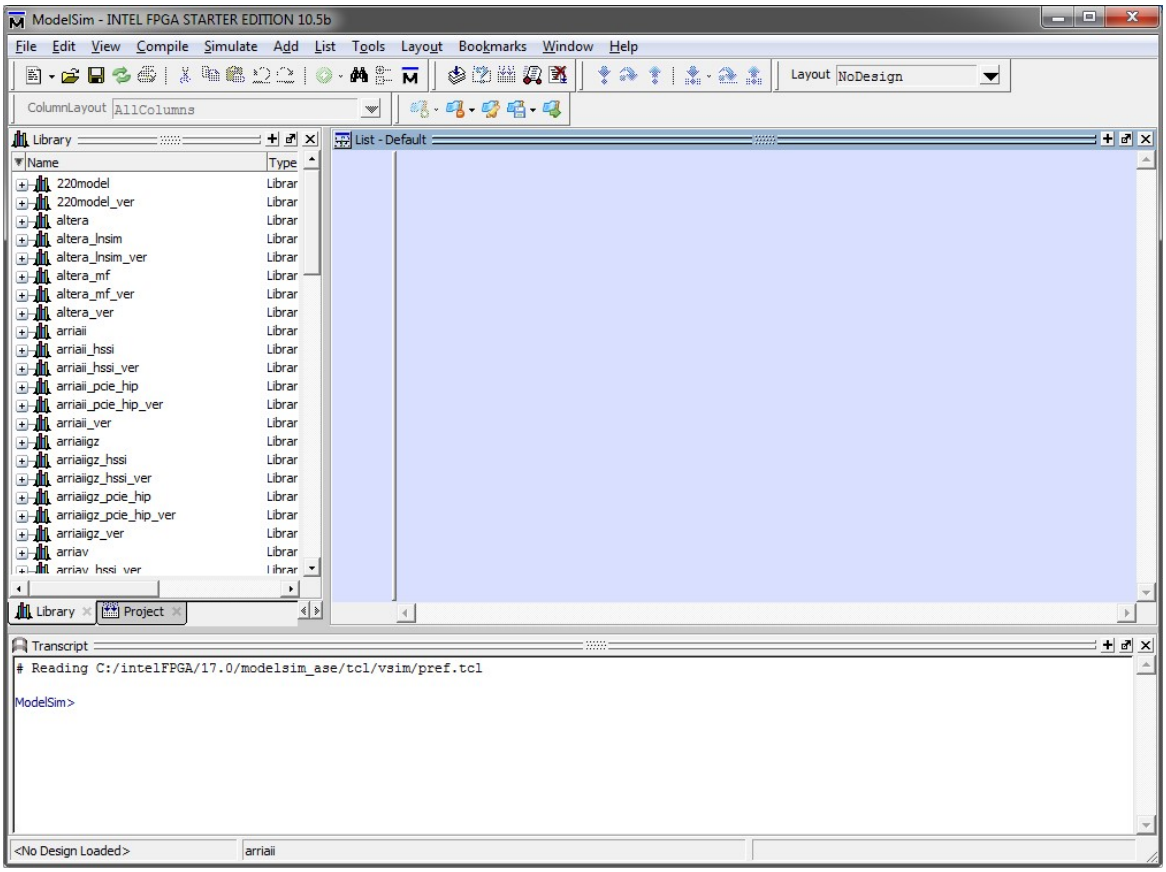
The first thing we will do is create a project in ModelSim to make functional simulations.

3. Creating a ModelSim Project

- Find and open the link to “ModelSim-Intel FPGA Starter Edition” on your desktop or in the start menu and start ModelSim.

After closing the “Important info” window, it should show the following window:

Figure 3



The library window shows all libraries. Libraries contain components that can be used in simulations. Most of the components in the ModelSim INTEL FPGA Starter edition shown in the list are hardware components inside the physical FPGA that can be used in your designs. There are also some other useful components that can be used for testing. For this course, we will not make use of any of them. Eventually, we will create our own library with our own components (later more on this).

We will first create a new project.

- Go to File → New → Project.

You should see the “create project” window.

- Fill it in as follows:

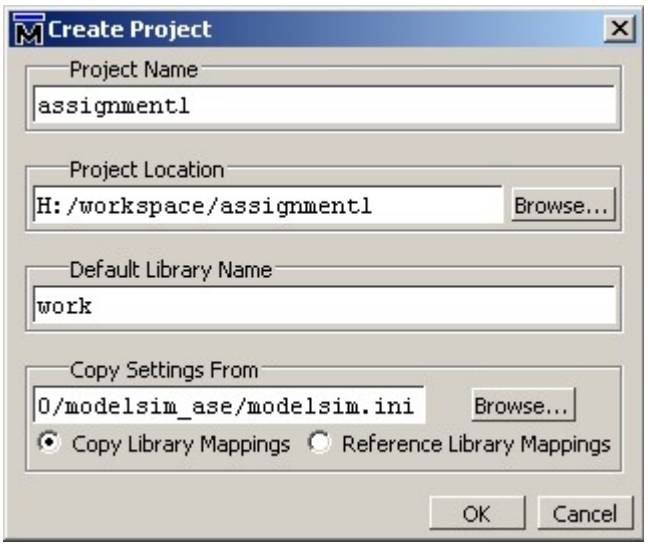


Figure 4

Create Project window

Warning!

Take care not to store your project files on the local hard-drive such as C:\, as other students or the ICT department might steal, corrupt or delete them!

- Press OK.

It will probably show a pop-up window that asks if you want to ModelSim to create a directory for your project.

- Let ModelSim create the new directory

A new window should pop up.

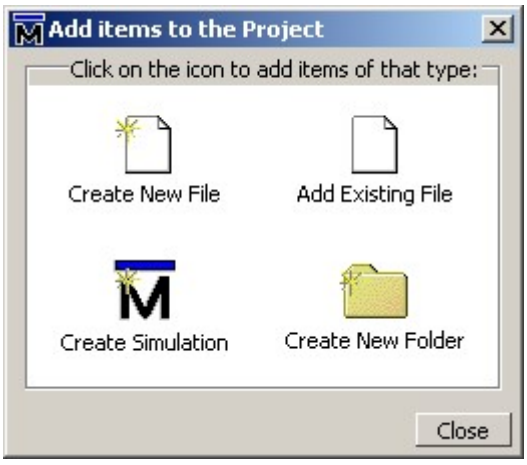


Figure 5

Add items to the Project window

- Click “Create New File”.

We will now create our first VHDL file. The extension for VHDL files is .vhd. **The name of the file must be the same of your components name!** Since the component will be named “my_first_component” we will name the file “my_first_component.vhd”

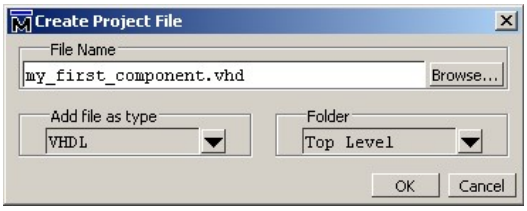


Figure 6

Create Project File

- Click OK and close the “Add items to project” window. Now ModelSim will show the project window:

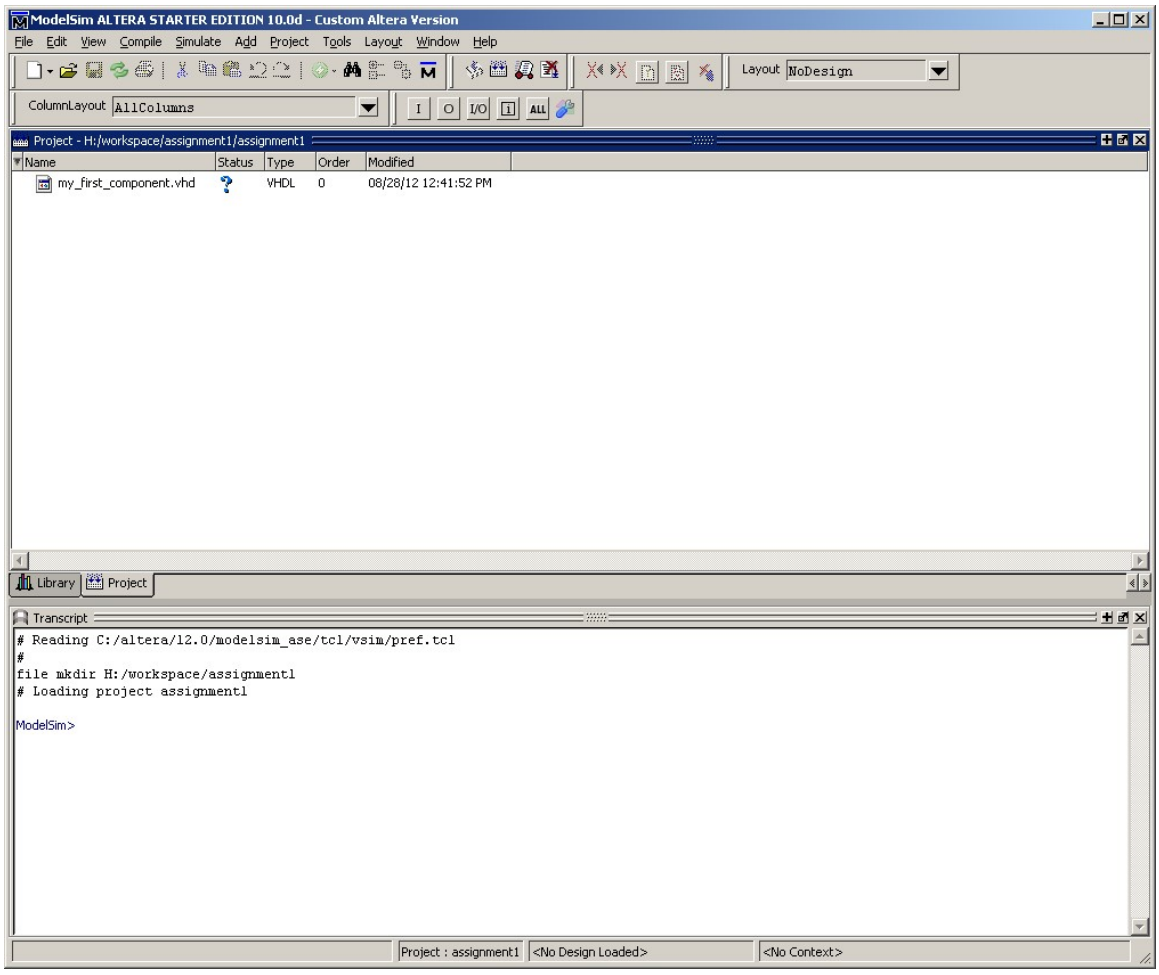


Figure 7

ModelSim Project window

We can see our VHDL file included in the project.

The status of the file, which is a question mark, means that the file is not compiled yet. We will do this later.

- Double click on the file to edit it. Add the following lines to the file:

```
library ieee;
use ieee.std_logic_1164.all;

entity my_first_component is
port (
   inputs:      in std_logic_vector(3 downto 0);
   outputs:     out std_logic_vector(6 downto 0)
);
end my_first_component;
```

The first two lines define the standard libraries that contain many different functions and data types in VHDL we want to use for this (and probably any other) project.

The ENTITY part defines the interface of our functional block. The PORT part inside the ENTITY defines which ports we want to see at the interface of our functional block. It does not define the implementation of the functionality of the block itself. This is done in the Architecture section.

Important Concept

This concept, the difference between and separation of the **interface** and the **implementation** of a (sub) system, is the key to system engineering in any field.

IN STD_LOGIC_VECTOR(3 DOWNTO 0) defines an input port (high impedance). STD_LOGIC is a basic digital connection. A vector makes it a group or bus of inputs. 3 DOWNTO 0 tells us that the bus is four bits wide. This is the VHDL notation for a bus. Note that we have written it in MSB-first style (Most Significant Bit first).

- Save the file.

We have now declared the interface of your component. What is missing is how it should behave. This is done in the architecture. However, only declaring the interface is enough to compile our component.

Right click on the file and select Compiler → Compile Selected.

You should now see the following:

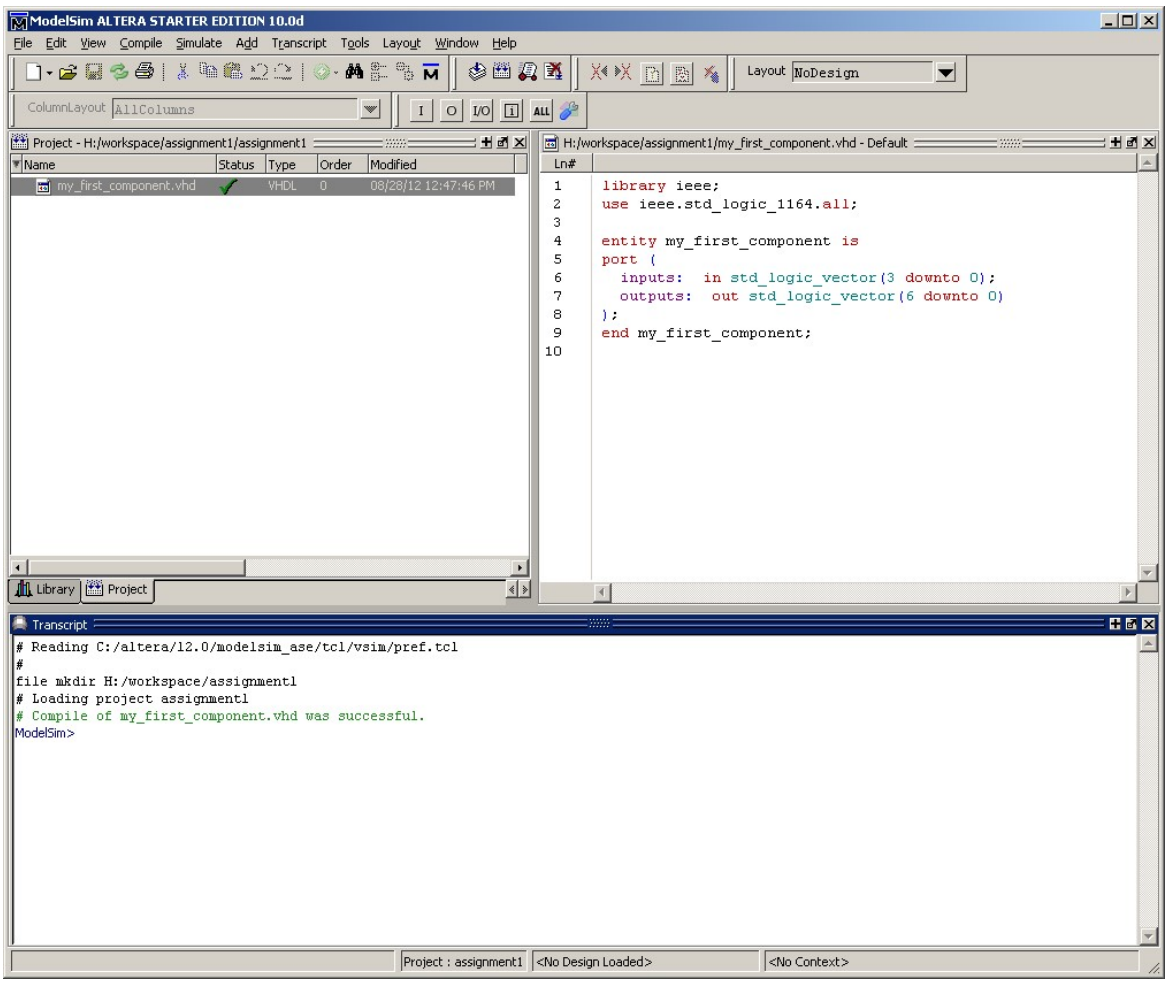


Figure 8

ModelSim Project window

Note that the status of the component has also changed to a green √. This means it compiled successfully. If it didn't compile successfully it would show a red X.
Even though we compiled the component, we cannot simulate it, because we have not declared how it should behave. This can be done by adding the architecture section to the code.

- Add the following lines to the file:

```
architecture implementation of my_first_component is
begin
    outputs(0) <= inputs(0);
    outputs(1) <= inputs(1);
    outputs(2) <= inputs(0) or  inputs(1);
    outputs(3) <= inputs(0) and inputs(1);
    outputs(4) <= inputs(0) and inputs(1) and inputs(2) and inputs(3);
    outputs(5) <= inputs(0) or  inputs(1) or inputs(2) or inputs(3);
    outputs(6) <= not(inputs(0) or inputs(1) or inputs(2) or inputs(3));
end implementation;
```

The outputs are now some combinational function of the inputs.

- Save and compile the file again.

If this is successful, we are ready to simulate our little design.

4. Simulating your design with ModelSim

- In the ModelSim menu, select Simulate → Start Simulation.

You should see the following dialog:

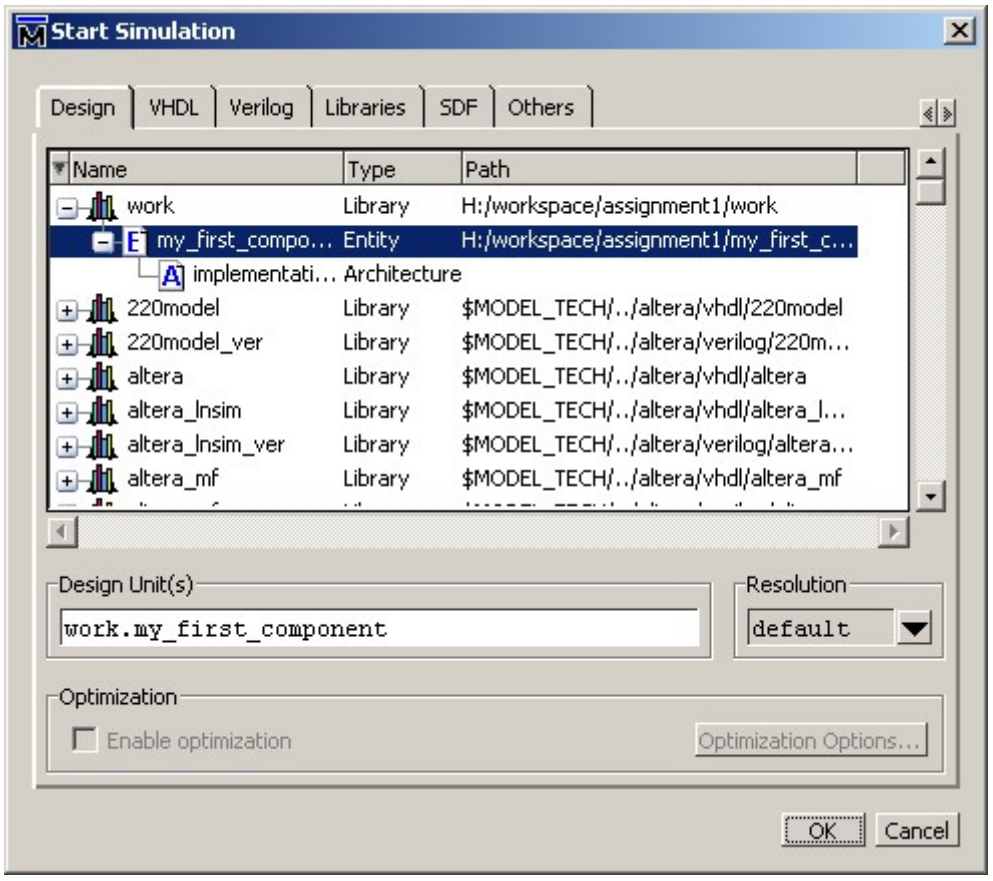


Figure 9

Starting a simulation

When we created the project, we called our default library “work”. This library now appears with all our compiled components in the dialog. Expand the library “work” and you can see “my_first_component” appearing there. You may even expand the component in the list to see which implementations are available. In our case, we have only one implementation which is called “implementation”.

- Select “my_first_component” and press OK to start the simulation.

You should now see the following window:

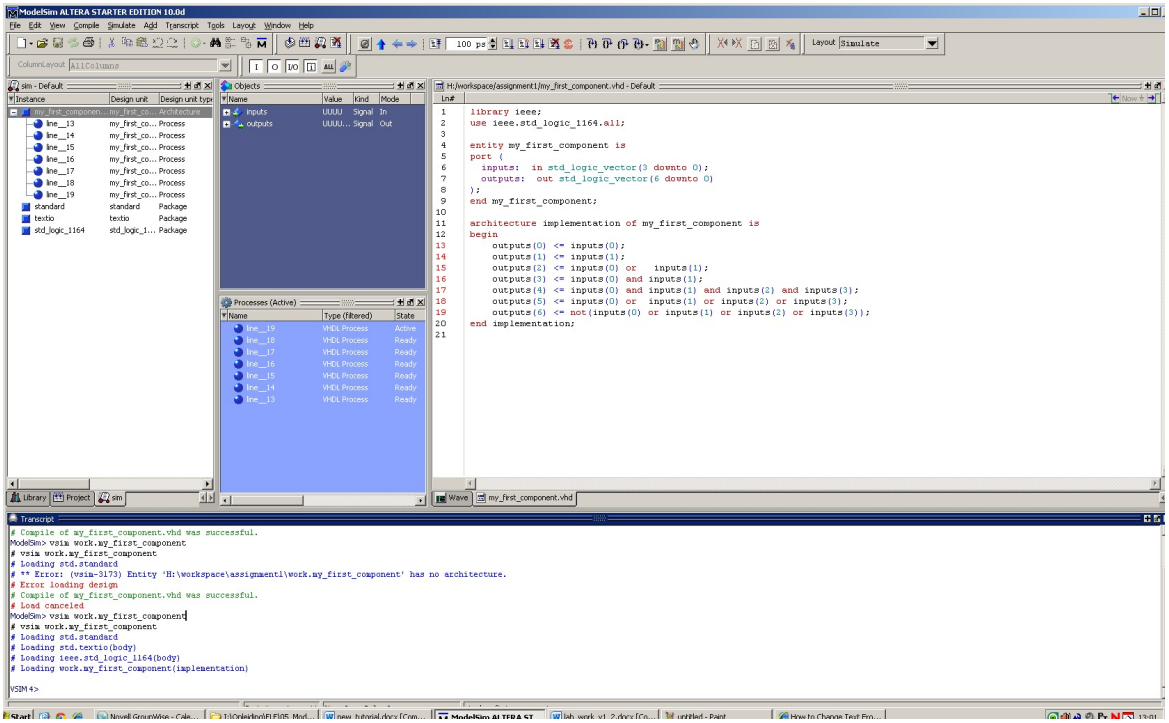


Figure 10

ModelSim simulation setup

This is the simulation setup of ModelSim. In the “Instance” window we can see all components and their underlying architectures and processes from which we can select objects to include in the simulation. All objects that can be included are shown in the “Objects” window.

We can clearly see our signals “inputs” and “outputs” here.

Now, if we want to simulate our design with all the signals that are in it do the following:

- Right click on *inputs* or *outputs* signal names in the “Objects” window.
- Click Add → To Wave → Signals in design.

All signals are now added to the “Wave” window. We can now see our signals in the wave window. Their values are all U’s. This means the values are unknown. To set the value of a signal, right click it in the wave window and select Force. Of course, we should only force signals that are inputs.

- Open the force window on the signal “inputs”.

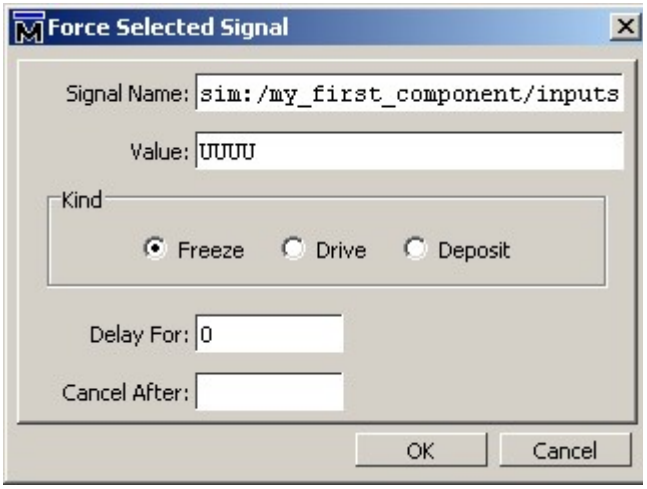


Figure 11

Forcing signal values

The value here is shown as UUUU. Remember that the inputs signal is a vector of four values. The inputs are currently all unknown (U).

- Change the value to 0000.

This means that input(0) will get the value 0. Input(1) will get the value 0 as well, and so forth.

- Click OK to apply the changes.

You may not see any changes right away in the wave window, but that is because the simulation hasn't started yet.

- To run the simulation for 100 timesteps, go to menu. Click Simulate → Run → Run 100.

The Wave window should now show: (to enlarge hold the Ctrl button down while scrolling with the mouse)

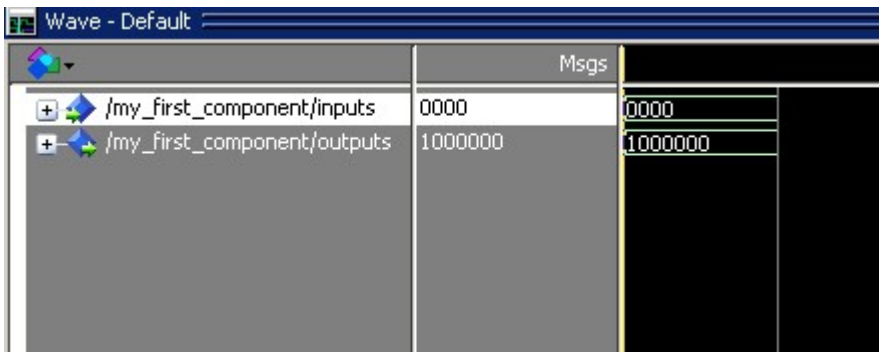


Figure 12

Simulated signals (1)

You can also expand the signals that are vectors to show their individual elements. This should look like this:

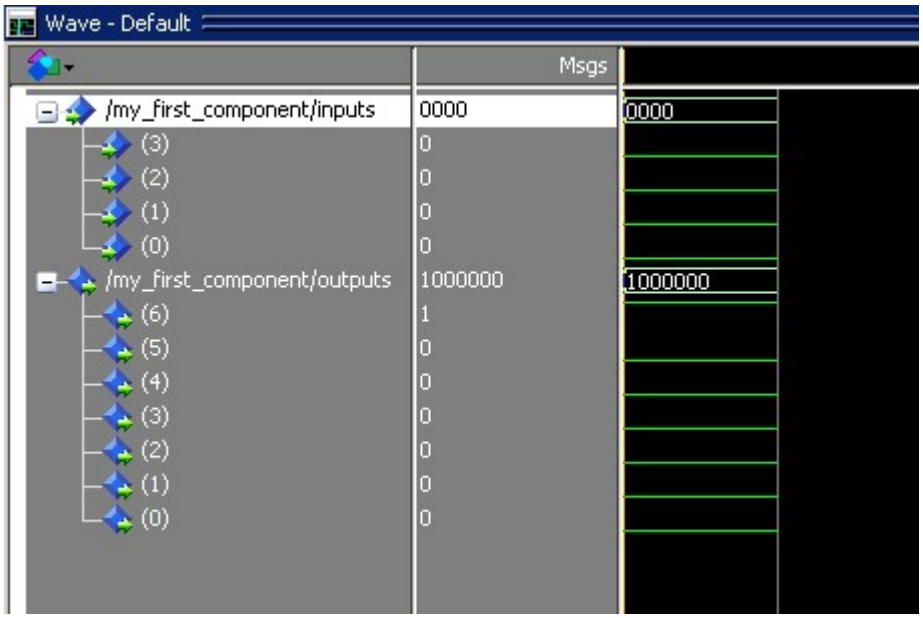


Figure 13

Simulated signals (2)

Verify that the outputs have the correct values (corresponding to your code).

- Now force the inputs to 0001 and repeat until you have tested all possible inputs (assume the values of each individual input can only be 0 or 1).

If you are sure your design works properly, call your instructor to verify your results.

5. Creating a Quartus project

Now the design is ready to be exported to Quartus so we may continue to implement it with the FPGA.

- Start “Quartus Standard Edition 18.0 (64-Bit)” from the desktop or from the start menu.
- Start the “New Project Wizard”.
- Skip the introduction of the Wizard.
- In the Directory, Name, etc… page of the wizard, select the same directory as your ModelSim project as the working directory.
- Name the project assignment1 as well.
- The top-level entity is our component called “my_first_component”. If you decide to add a different top level later, it is easy to change after you’ve completed the wizard.

Eventually, the current page should look as follows:

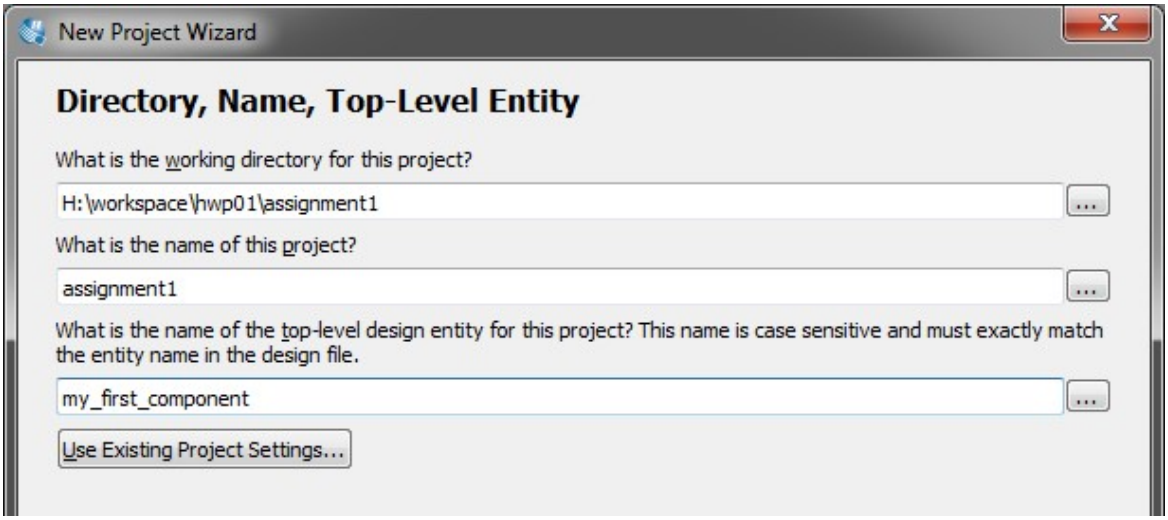


Figure 14

New Project Wizard in Quartus (1)

- Click Next.
- Set Project Type to Empty Project and click Next.

On the next page (Add Files), since we’ve already created a VHDL file, we can easily include it in the project by pressing Add All. The page should now look like this:

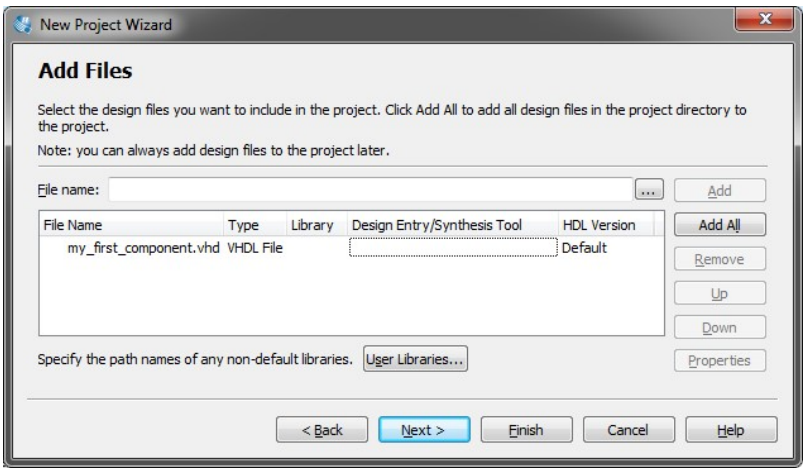


Figure 15

New Project Wizard in Quartus (2)

- Click Next.

In the Family and Device Settings page, we have to select which FPGA we're using. Our programming tools are well capable of selecting the proper device automatically (this is done through the JTAG chain), but let us select the right FPGA from the start.

The right device is the **Cyclone V 5CSEMA5F31C6**

Select this device (Assignments -> Device…):

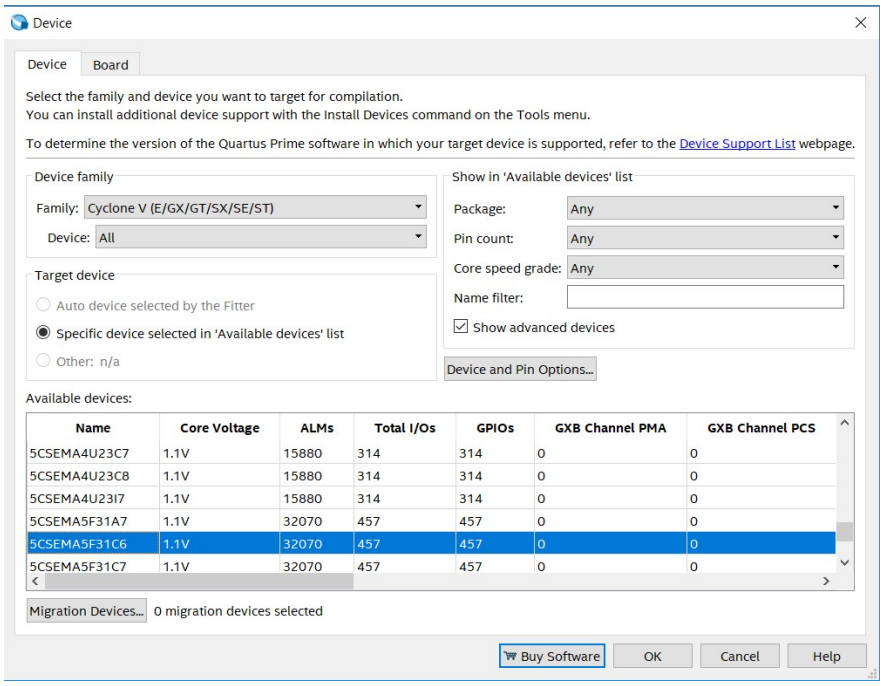


Figure 16a

New Project Wizard in Quartus (3a)

Alternatively, you can select the Board tab and select the DE1-SoC board (fig. 16b):

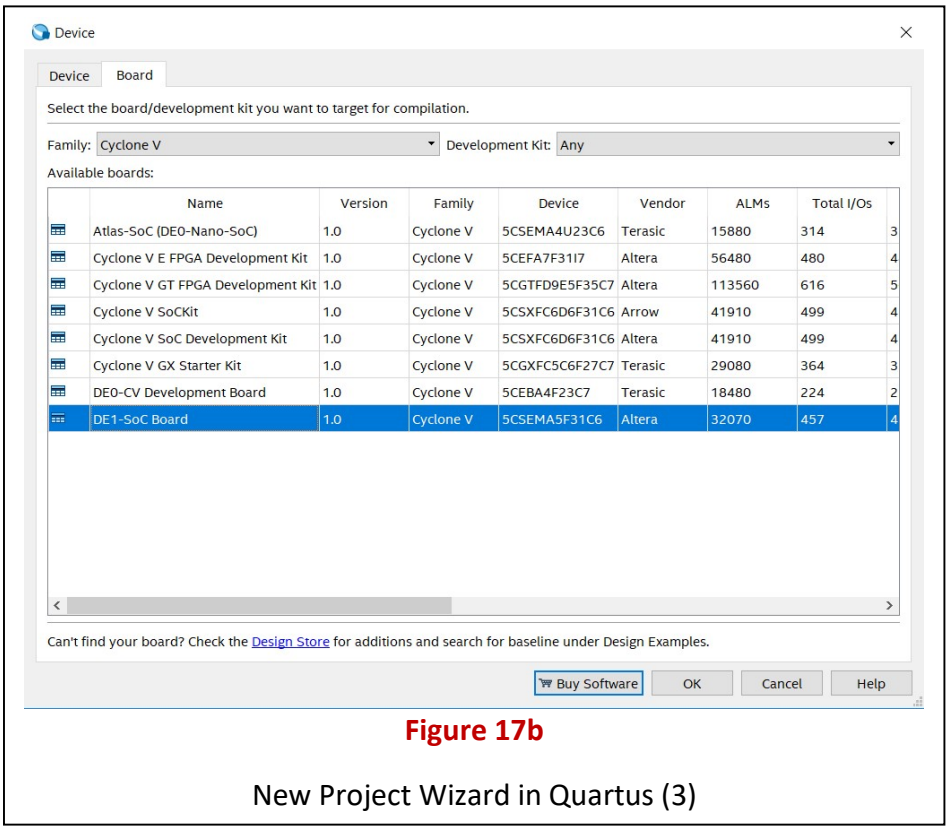


Figure 17b

New Project Wizard in Quartus (3)

- Now click Next until you can Finish the wizard.

We can now see in the Project Navigator pane on the left of the Quartus II screen. We want to implement my_first_component on the Cyclone V FPGA that is on the DE1-SoC development and education kit.

You may double click on “my_first_component” on the left side at the tab “Files” to show its source. This can be a schematic, a VHDL file, a Verilog file or anything else that Quartus can handle. In our case, we know it is our VHDL file we created in ModelSim, so double clicking it should result in the following:

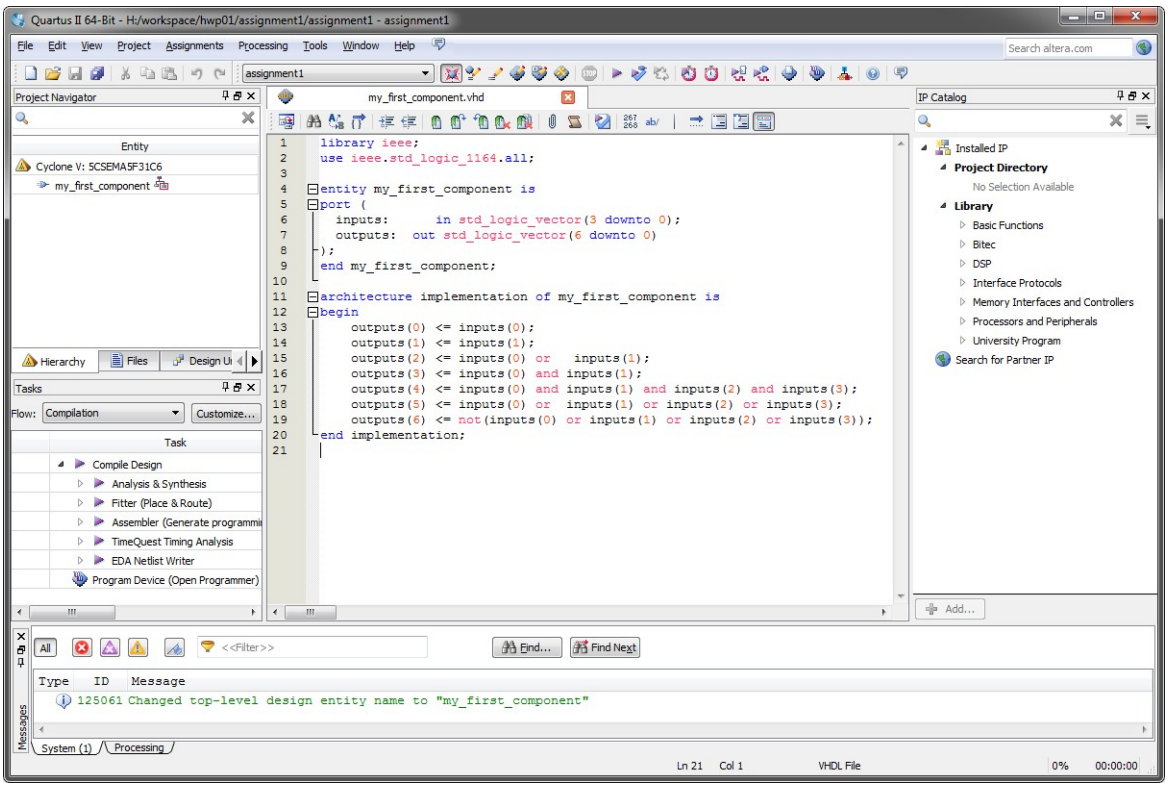


Figure 18

Quartus main screen

6. Assigning signals to physical pins

The first thing to do now, is to compile the design in Quartus, to let Quartus know which signals we are using and which signals have to go to the outside world.

- In the Tasks pane, double click “Analysis & Synthesis”.

If Quartus is done, it will show a green check mark next to the task you performed (if everything went OK). If there are any errors or warnings investigate them and determine if they need to be resolved.

Quartus should show this:

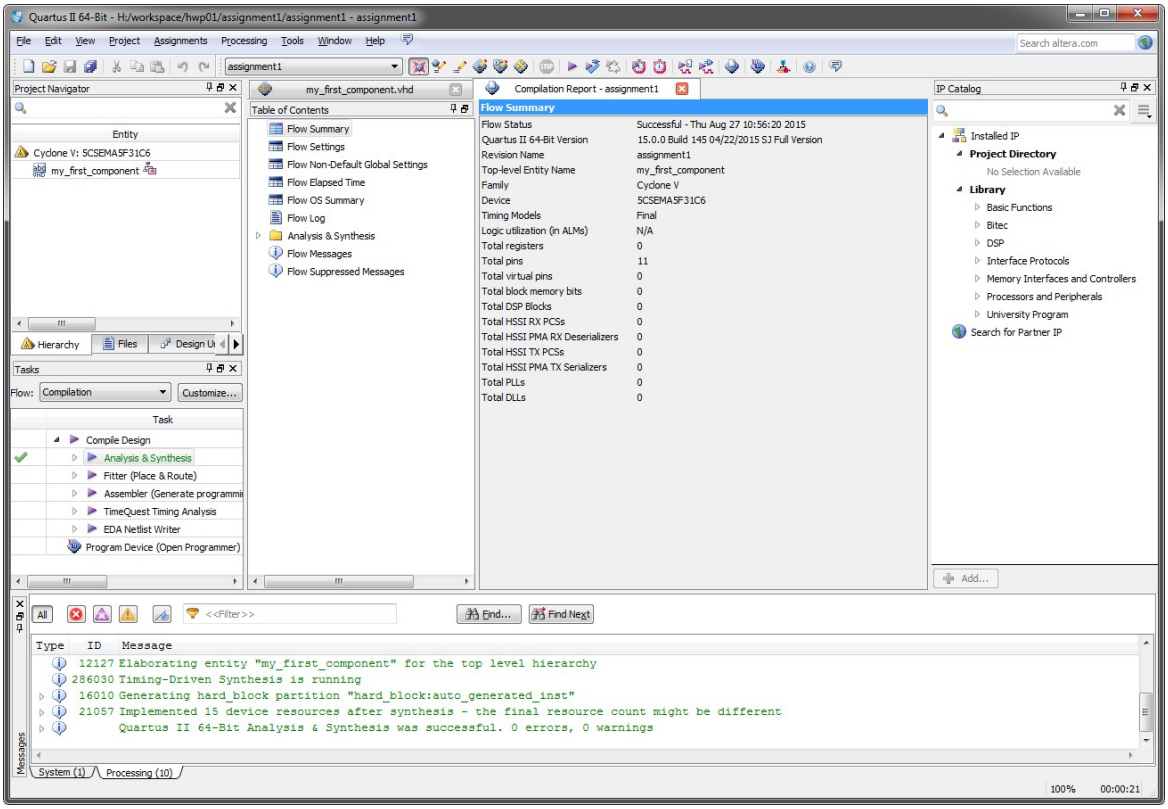


Figure 19

Compilation Report

In the flow summary, we can see that our design uses 11 pins. The number of pins is correct, since we have 4 inputs and 7 outputs. This means Quartus properly detected which signals we want to connect to the outside world.

We will now assign real world pins of the FPGA to our signals.

First we **have** to configure the FPGA’s voltage settings and what to do with unused pins.

- In the menu click Assignments → Device.
- Click on the “Device & Pin Options” button.
- In the Device & Pin Options menu, open the Unused Pins Category.

We have to change all unused pins to tri-stated (high impedance) inputs to prevent a short-circuit that can potentially damage the board!

- Change “Reserve all unused pins” to “As input tri-stated”.

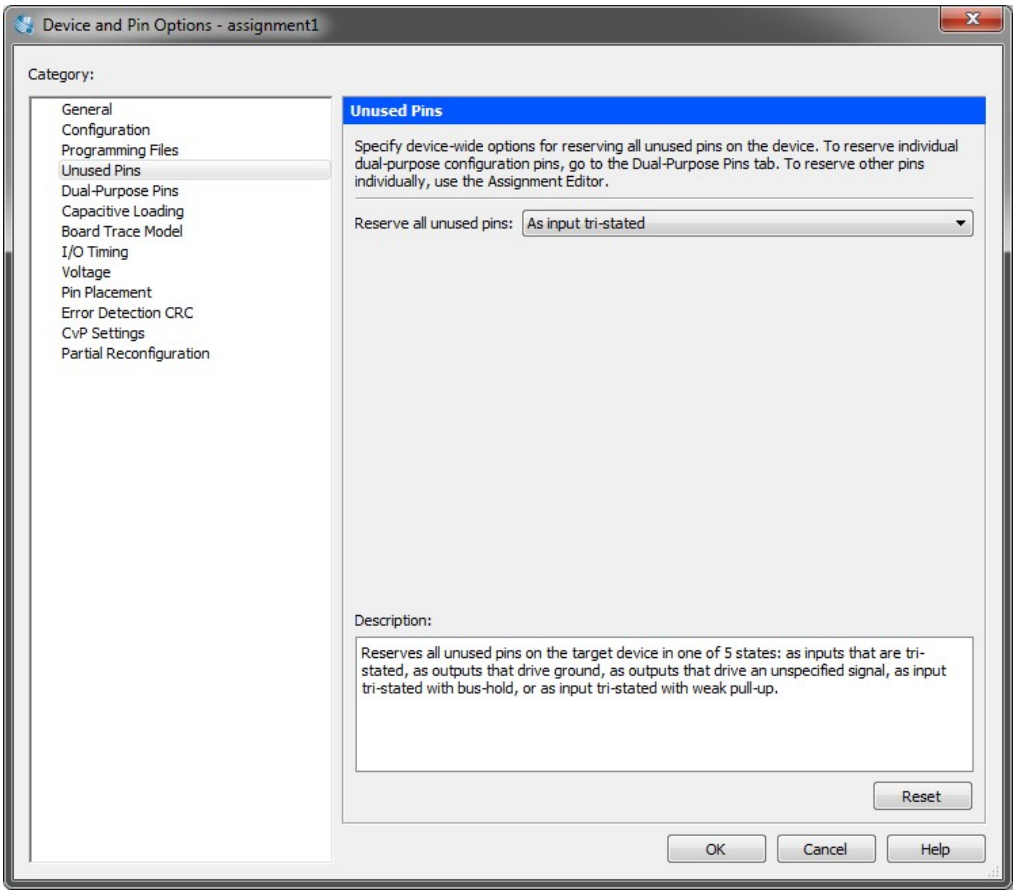


Figure 20

Device and Pin options

The FPGA settings should be OK now. If you are unsure, ask your instructor, for it is very important these settings are correct!

Now it is time to connect the signals of our design to real world pins.

- In the menu, select Assignments → Pin Planner.

You will now get the following window:

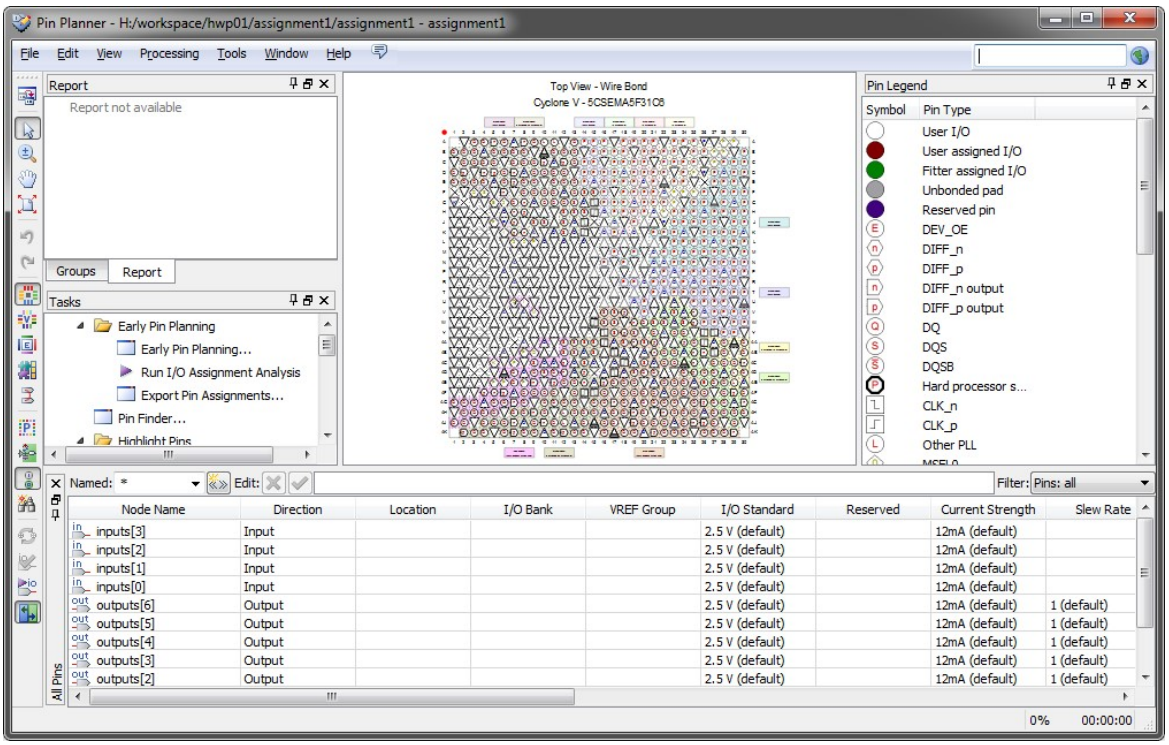


Figure 21

Pin Planner

We see the BGA side of the FPGA chip that we've selected. Our goal is to connect our signals to one of the pins of the chip. We cannot just arbitrarily select some pins because our FPGA resides on a given circuit board. Luckily we have the schematics and a user manual for the board.

We see a list of so called "nodes" which are actually our signals that have to get a physical connection to the outside world.

Our goal is to connect inputs 0 to 3 to keys 0 to 3 on the DE1-SoC board. Open the user manual (it resides in the DE1-SoC\ folder in this course's folder) and find out at which location those keys are connected.

Do the same for the outputs 0 to 6. We want to connect these to the red LEDs 0 to 6.
Once you know all the pin locations for the keys and the LEDs, fill in those locations in the column locations next to their respective nodes. You may just type the pin location instead of searching for it in the dropdown menu.

An example is shown for the first input and output in the list:

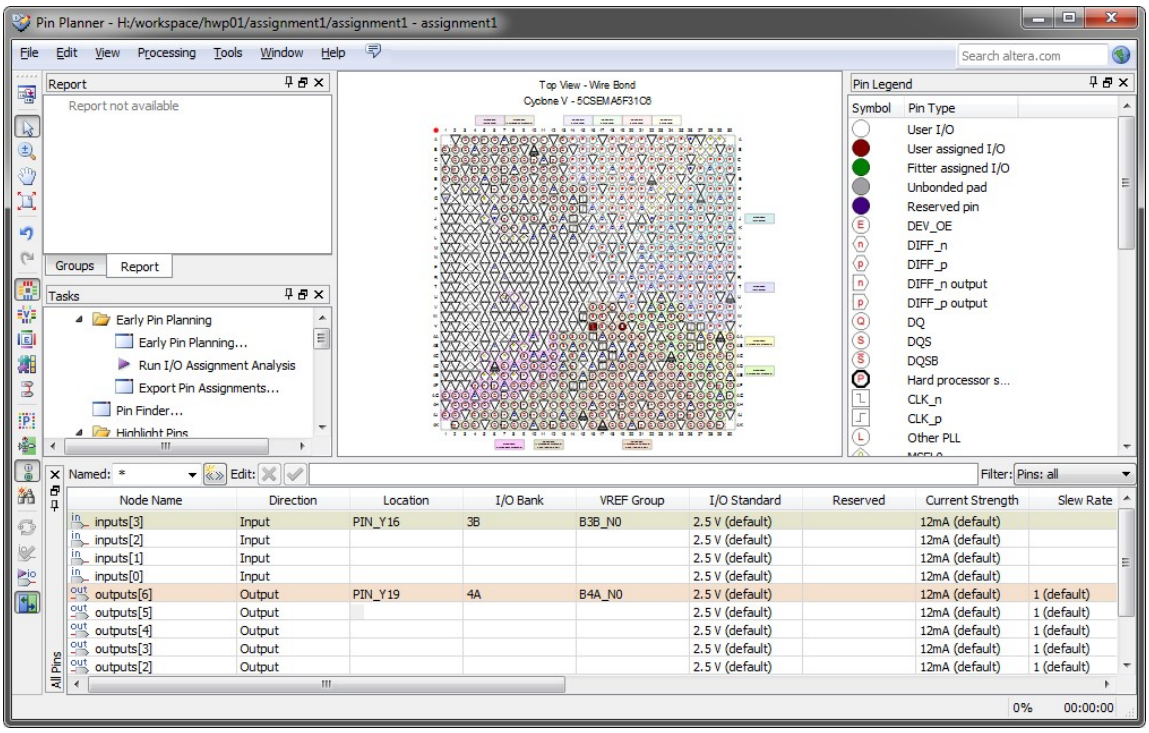


Figure 22

Some nodes assigned to some pins

Once you are done connecting all nodes, you may close the pin planner.

7. Place and route the design for FPGA implementation

Now that all nodes are connected to some physical pin, we may let Quartus place and route our new design in the FPGA. This will generate a programming file with which the FPGA will be configured to work as we want.

- To do this, double click on the Assembler task:

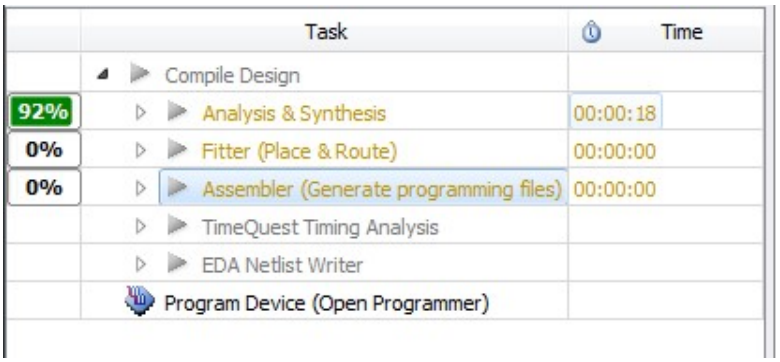


Figure 23

Assembler task

This step usually takes a relatively long time, compared to compiling a small piece of software, for example. This is another reason why we want to simulate our design before implementing it into the FPGA.

If everything went OK, it should eventually display a pop-up that says the compilation is successful. Ignore any warnings for now.

Some interesting things to check is the RTL schematic that Quartus has synthesized from your code. You may find it here:

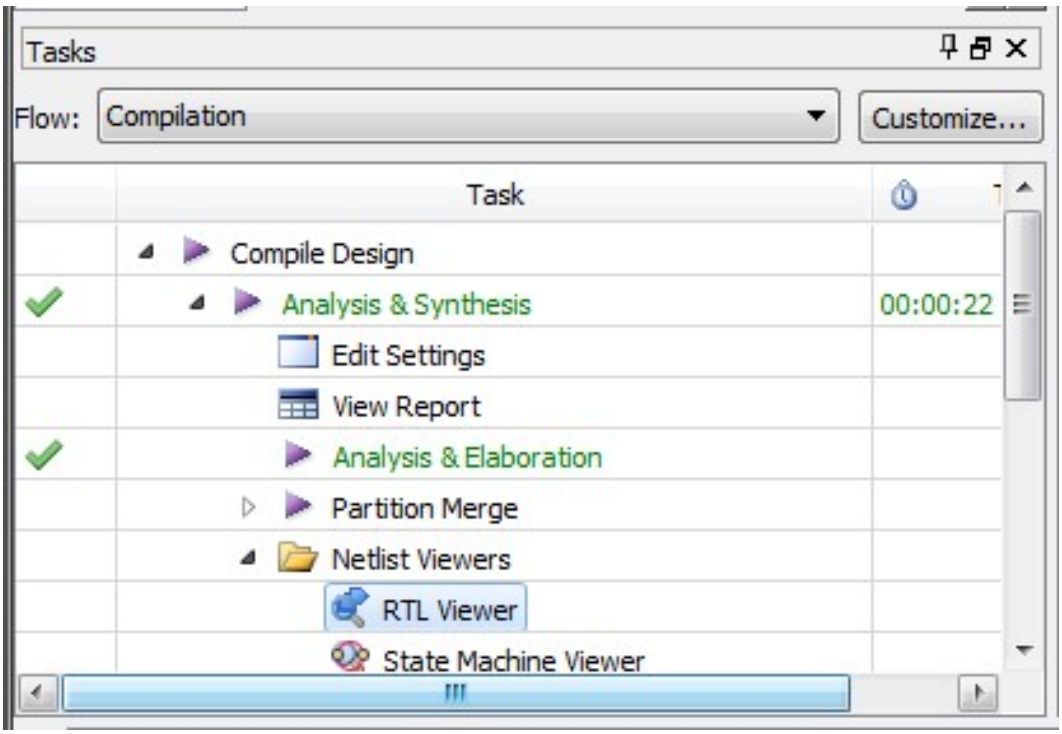


Figure 24

RTL Viewer task

The RTL viewer will show the register transfer level schematic of your design.

It should show the following:

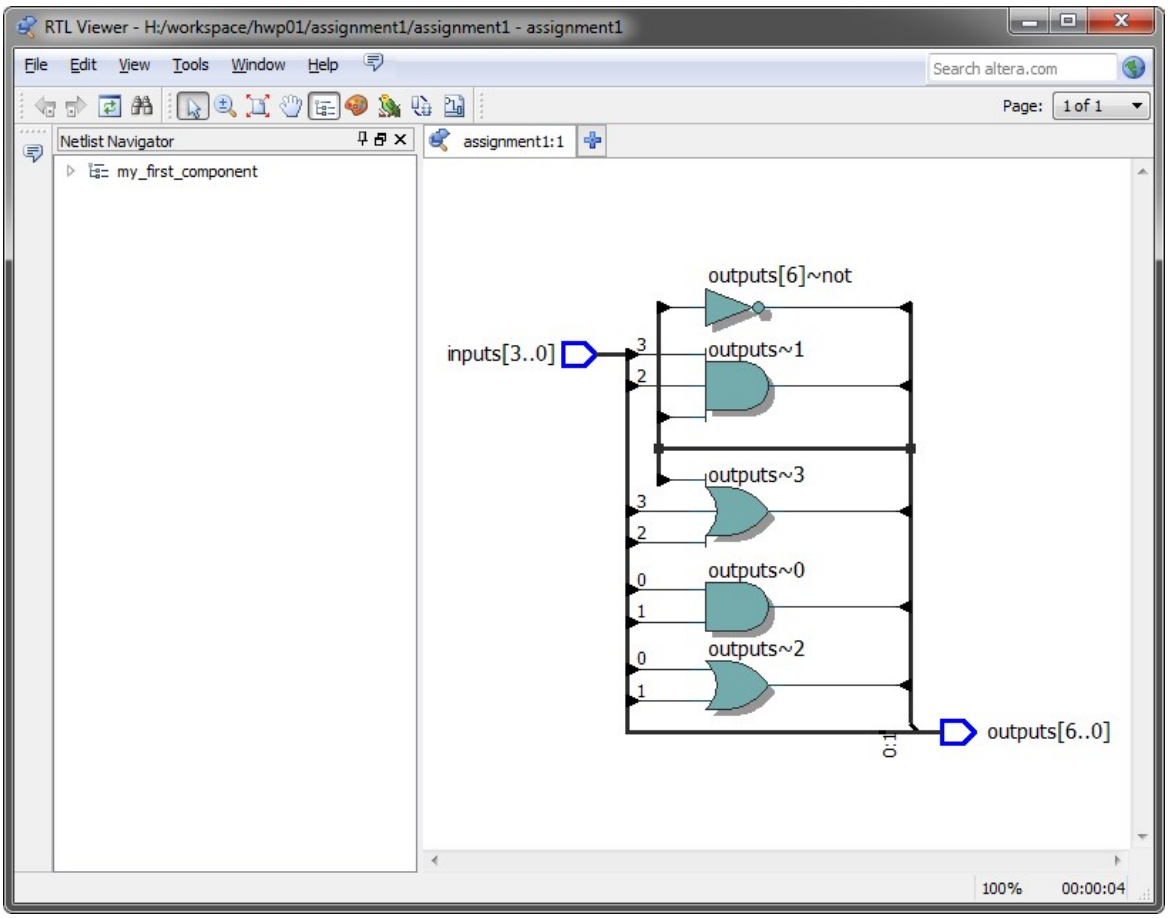


Figure 25

RTL Viewer

Here we may see the logic function that our architecture implements with the given inputs and outputs.

Eventually, writing VHDL code is about thinking how this schematic will look when you write a certain piece of code. (Usually you will think about your schematic in a more macroscopic way, and not as detailed as this.)

Another interesting view is the Post-Fitting Technology Map viewer. Here you can see how the logic elements and other hardware components in the FPGA are used to create your schematic. It can be found here:

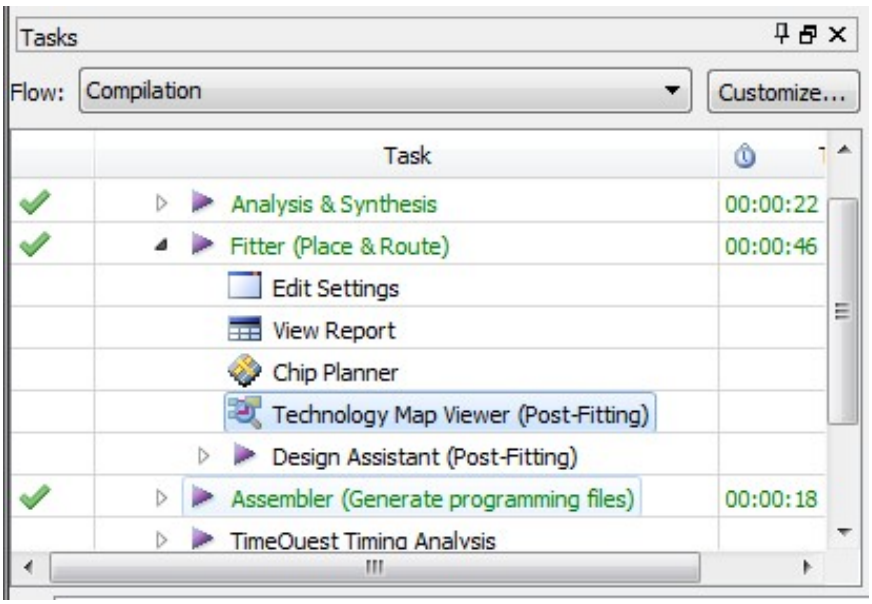


Figure 26

Technology Map Viewer (Post Fitting) task

We can see the following:

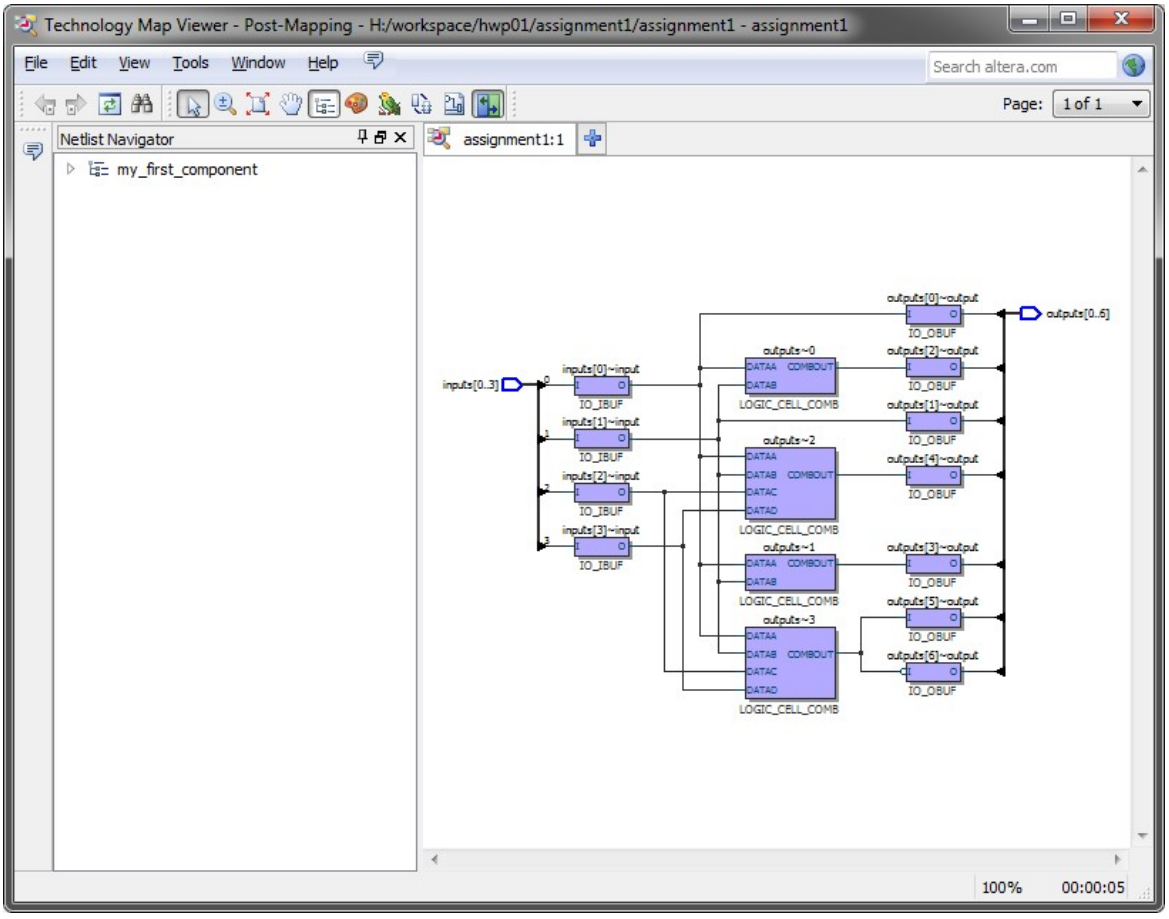


Figure 27

Technology Map Viewer

Here we see that it uses four logic elements. You may even double click them to expand them and see which kind of function the logic element implements. If you close the technology map viewer and look at the compilation report, you may even see that four logic elements are only a tiny fraction of the number of logic elements available (over thirty thousand).

Another interesting thing to look at is the Chip Planner which can be found above the Technology Map Viewer task.

Here we can see the following:

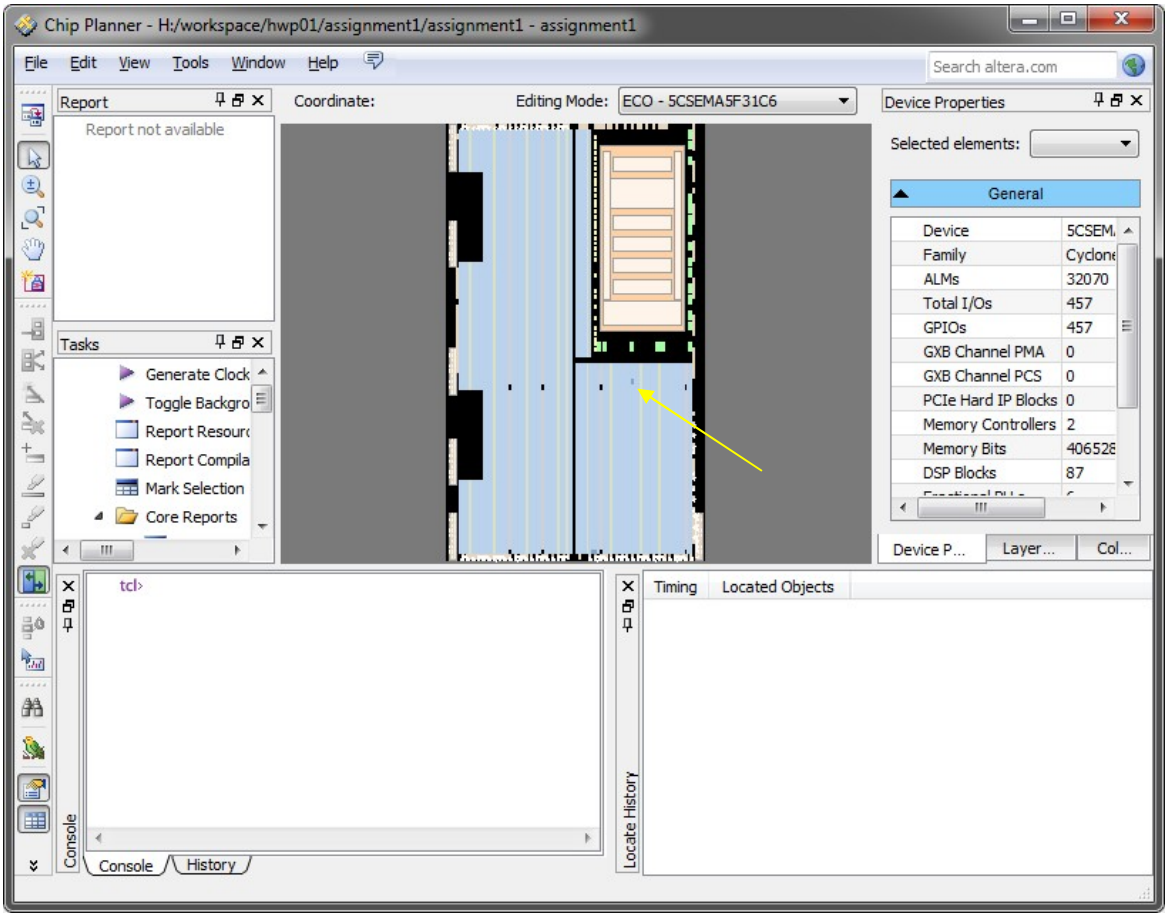


Figure 28

Chip Planner (1)

This is a schematic view of the FPGA chip itself, and it also shows which elements of the chips are used. Only one region is used, and most of the time the unused regions are shut down to save power. The bigger your design is, the more power it will therefore use.

By, the way, the region that is used is not the black region. It is in the middle right, the bit more blueish rectangle. If you zoom in on this, you can see the logic elements that are used:

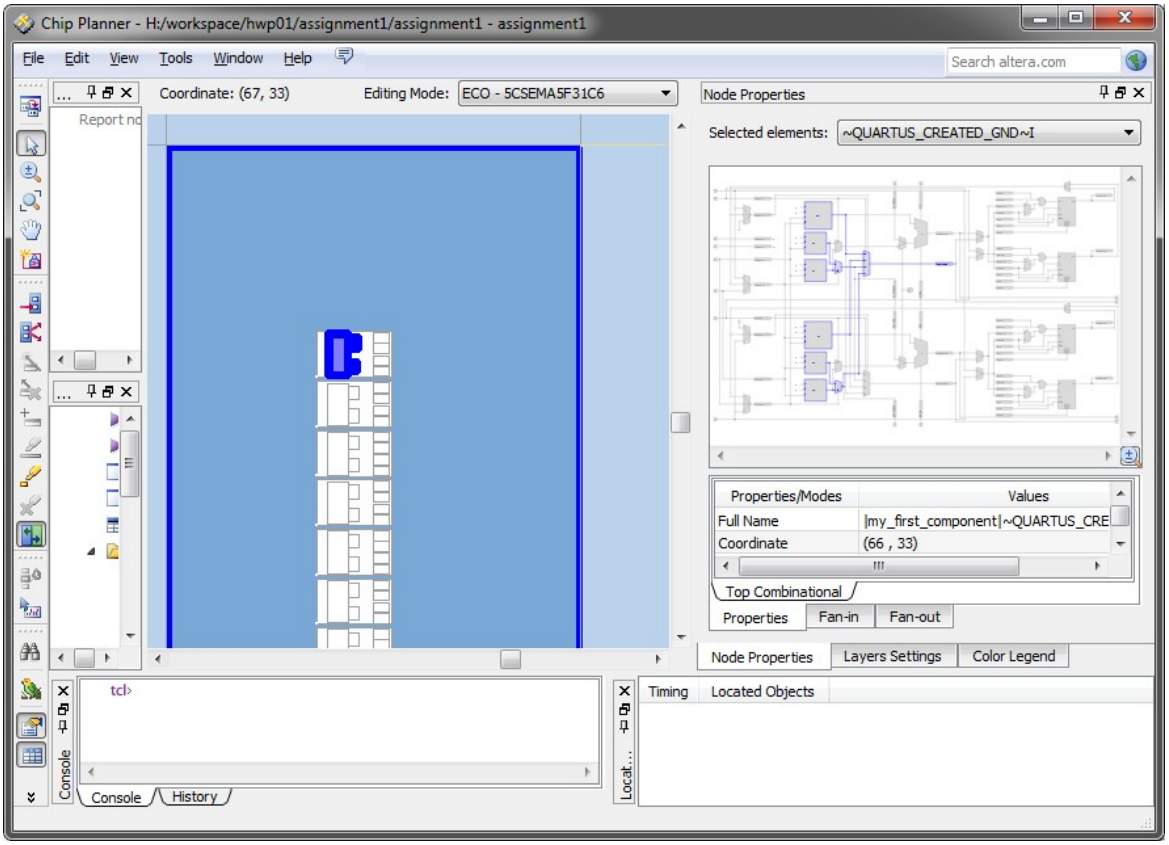


Figure 29

Chip Planner (2)

For specialized and high-speed designs, the tools which you have just seen are extremely useful, because you can manually implement designs in the FPGA if you wish, so to optimize your design. In this course, we will not use them much (except for the RTL viewer).

- On the bottom of the Tasks list, double click Program Device.

This will open the Programmer window.

- Connect the DE2 kit to the computer using the USB Blaster connection on the board, and also connect it to a power supply, and turn on the board.

Under the Hardware Setup button in the Programmer window, make sure the currently selected hardware is set to the USB Blaster.

- After you click Auto-Detect, the programmer window should look like this:

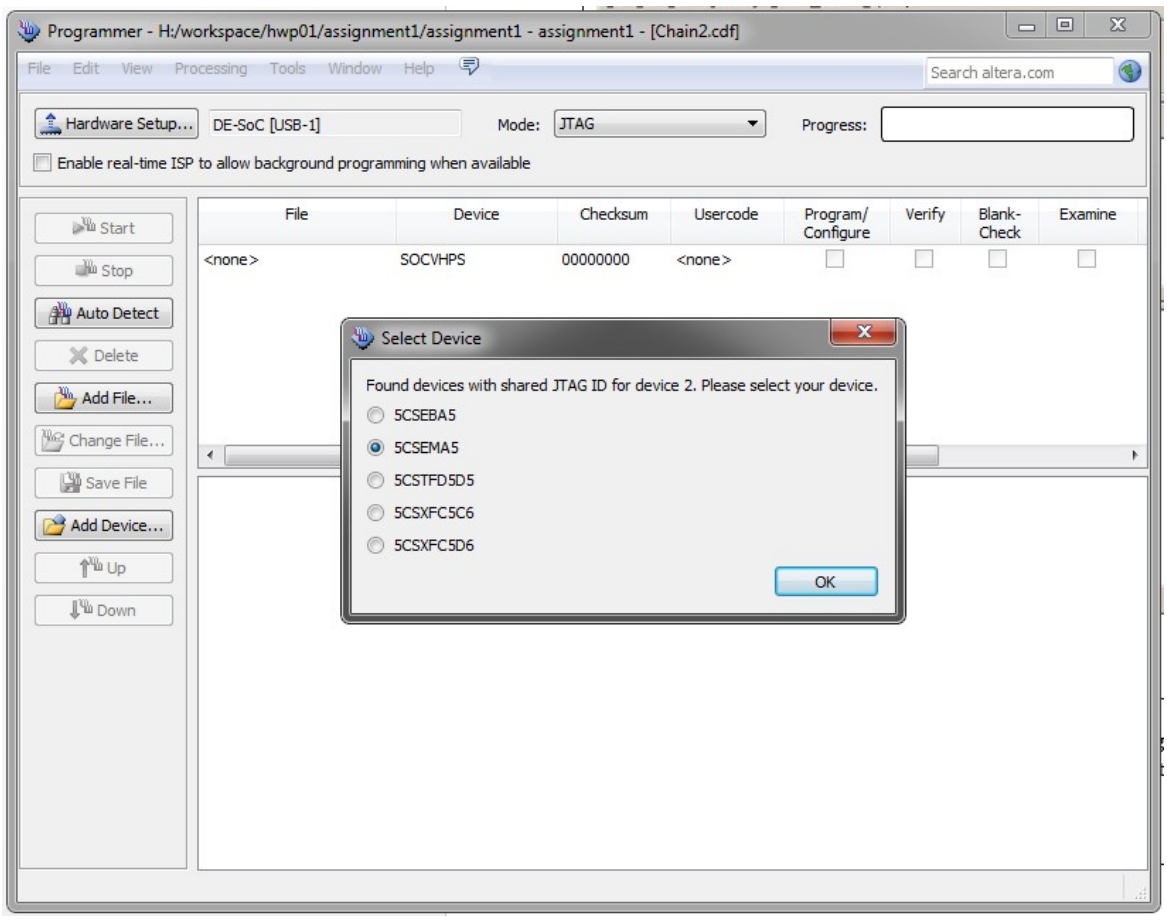


Figure 30

Quartus Programmer

- Select 5CSEMA5 and press OK.

- Double click under the “File” column next to device 5CSEMA5 and select assignment1.sof in the output_files/ folder of your project.

- Also check the Program/Configure checkbox next to the device.

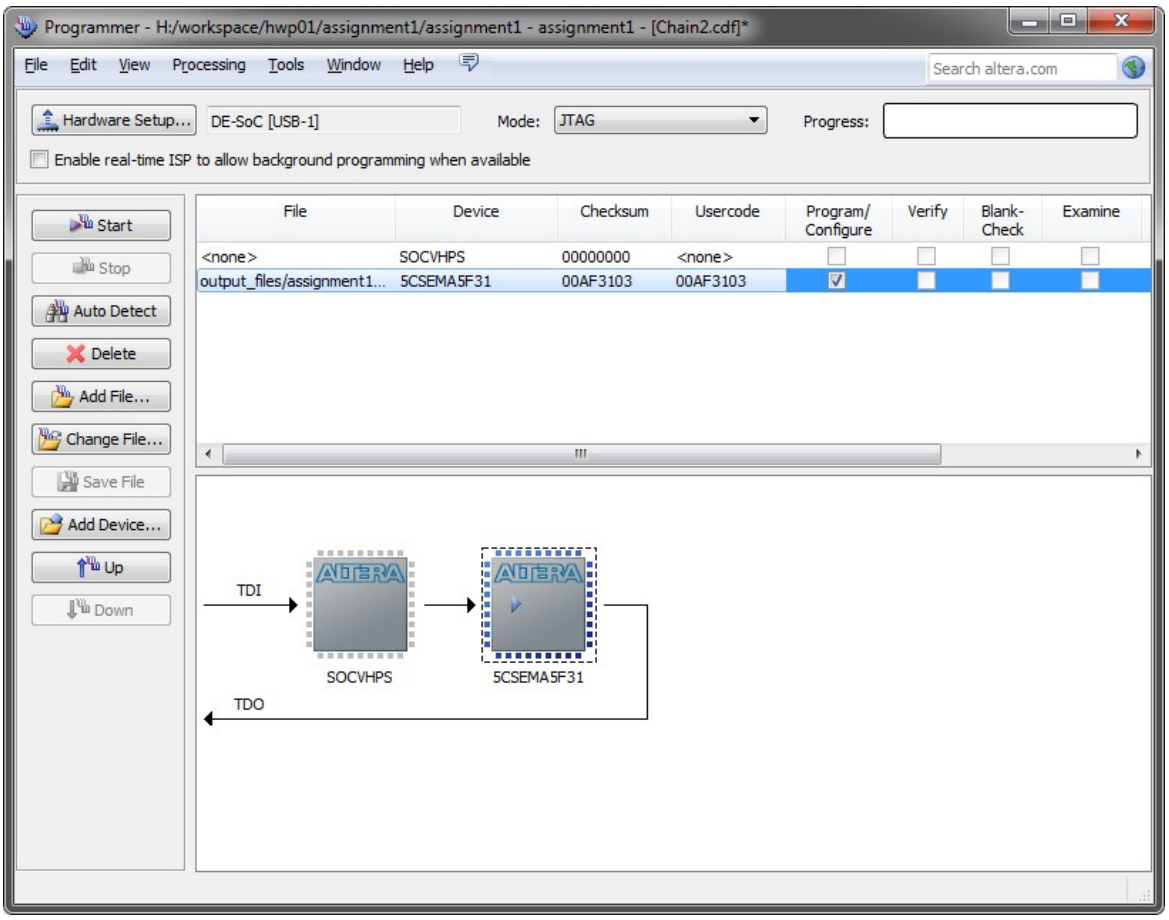


Figure 31

Quartus Programmer

By doing this, you are letting the programming interface know that the bitstream to be programmed in the FPGA is in assignment1.sof.

We can see the boundary scan chain that will be used to program our devices. In this case we will only program the FPGA chip, so only this one is seen in the chain.

- Look at the DE1-SoC kit while pressing start.
- While loading the new FPGA configuration, the JTAG TX LED should be on that shows a data transfer is in progress in the JTAG boundary scan chain.

Your new design will be programmed into the FPGA.

Notice that LEDR0 to LEDR5 are on. Look at your VHDL code and verify that this is correct by trying all the combinations with KEY0, KEY1, KEY2 and KEY3.

Congratulations!

You have designed, simulated and programmed (probably) your first FPGA design!

Call your instructor now to verify that the DE1-SoC board is programmed correctly. Explain your steps taken and convince him/her that you are ready to move on to the next assignment.

8. Summary

We have now finished our first assignment and our first design. We have done the following:

- Created a new project in ModelSim and Quartus
- We've made a VHDL design
- We've simulated our design with ModelSim
- We've connected the pins of our top-level design to the real FPGA pins
- We've generated programming files and programmed them into the FPGA

You are now ready to move on to the next assignments. You can use this manual as a reference for making new projects and simulations.

Good luck with the rest of the course!

9. Tips and further reading

Engineering tools are often very versatile and offer a lot of functionality. All those buttons and settings might confuse you a bit in the beginning. It is always handy to keep user manuals and tutorials at hand. Most tools have a user community with forums. On these you can often find questions that arise already asked and answered. If not, of course you can always post a new question yourself.

Quartus Help

In Quartus you can click Help → Search and type in anything you want to know about in Quartus.

ModelSim

In ModelSim you can click Help → PDF Documentation → User Manual to find out about ModelSim in general.

TCL Scripts (DO Files) for ModelSim

You can open the Reference manual to see what commands are available for the DO files (TCL scripting). Also you can see what syntax the commands we've used above use, so you can experiment with those. There are also examples given.

Hierarchical Designs

If you want to create hierarchical designs (this is recommended for most exercises) read chapter 10.3 of the book for more information on the COMPONENT keyword.

10. Assignments

This chapter contains the further assignments for the lab work of this course. There are some tips and hints given on a regular basis, so make use of those. Also there are some code snippets in the course folder on the network that you can use.

Completing an assignment takes three steps:

1. Brainstorm about what the best solution would be for the problem in the assignments and **sketch the chosen solution** (top level and sub level functional blocks that you think you will need).
2. Write the code, **simulate (WITH TESTBENCH)** and debug it.
3. Program the DE1-SoC board with your solution and **verify** its functionality.

After completing *every* step, call your instructor to make him/her verify that you are done with that step.

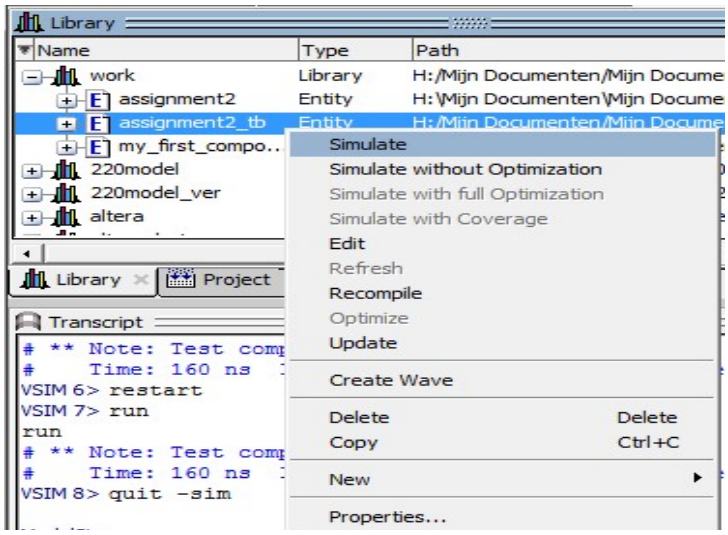
Assignment 2: Driving the 7-segment display

Program the FPGA in such a way that you can input a 4 bit binary number using the switches on the DE2 board (SW0 to SW3). The binary number must then be displayed in hexadecimal on a 7 segment display as 0, 1 ,2 ,3 , 4, 5, 6, 7, 8, 9, A, b, C, d, E or F. Also make the red LEDs above the switches light up if the corresponding switch is turned on. The LEDs must go out if the corresponding switches are turned off.

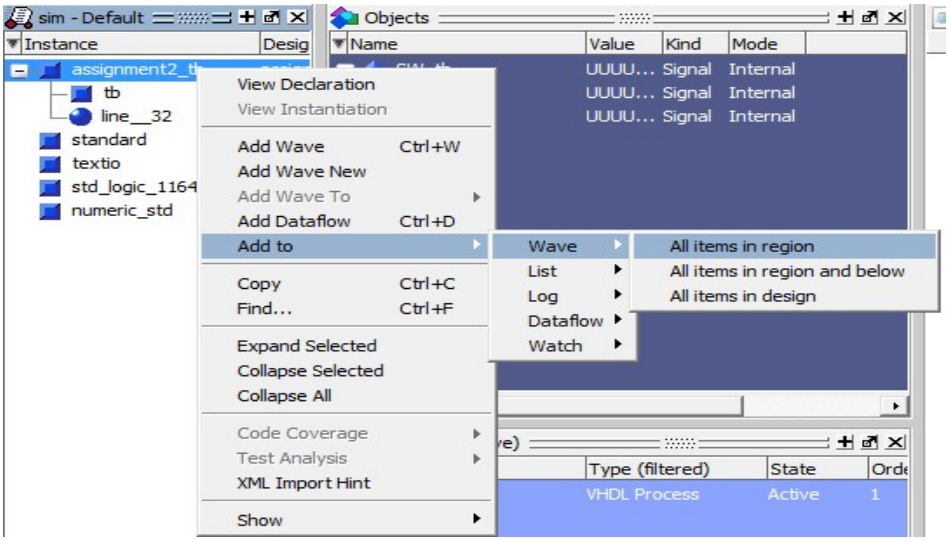
Functional testing of your design is necessary to verify your design. In Chapter 4 we have specified the inputs manually. However, for larger circuits this is not feasible and therefore testbenches are introduced. For this assignment we focus on functional simulation and its purpose is to check the functionality of the circuit. A testbench is written in VHDL and we use ModelSim to execute the testbench. Add a file to your project and name it **assignment2_tb.vhd**. Add the code found in the following Listing.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

-- The entity of your testbench. No ports declaration in this case.
ENTITY assignment2_tb IS
END ENTITY;

ARCHITECTURE testbench OF assignment2_tb IS
      -- The component declaration should match your entity.
      -- It is very important that the name of the component and the ports
      -- (remember direction of ports!) match your entity! Please notice that
      -- the code below probably does not work for your design without
      -- modifications.
      COMPONENT assignment2 IS
            PORT (
      SW    : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
      HEX0  : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
      LEDR  : IN STD_LOGIC_VECTOR(17 DOWNTO 0)
);
      END COMPONENT;
      -- Signal declaration. These signals are used to drive your inputs and
      -- store results (if required).
      SIGNAL SW_tb      : STD_LOGIC_VECTOR(17 DOWNTO 0);
      SIGNAL HEX0_tb    : STD_LOGIC_VECTOR(6 DOWNTO 0);
      SIGNAL LEDR_tb    : STD_LOGIC_VECTOR(17 DOWNTO 0);

      BEGIN
      -- A port map is in this case nothing more than a construction to
      -- connect your entity ports with your signals.
      tb: assignment2 PORT MAP (SW => SW_tb, HEX0 => HEX0_tb, LEDR => LEDR_tb);

      PROCESS
      BEGIN
      -- Initialize signals.
      SW_tb <= "000000000000000000";
      -- Loop through values.
      FOR I IN 0 TO 15 LOOP
            WAIT FOR 10 ns;
            -- Increment by one.
            SW_tb <= STD_LOGIC_VECTOR(UNSIGNED(SW_tb) + 1);
      END LOOP;

      REPORT "Test completed.";
      -- Wait forever.
      WAIT;

END PROCESS;

END ARCHITECTURE;
```

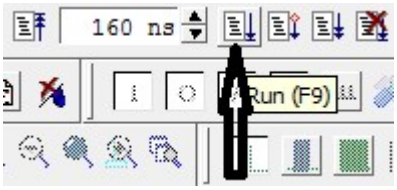
The code in the testbench should look very familiar. In the testbench you can use whatever VHDL construct you like, as you would normally do in your circuit designs. As you have read in the comments it is **very important** that the component declaration have the same port(s) as your entity declaration. Once you have added this file you can start simulation and test your circuit. See the following screenshot for instructions on starting the simulation.



Click on *Library* and then right click on your testbench. A similar window should pop up and click on *Simulate*. Now you add the signals from your testbench to your waveform.



Now you can run the simulation. Click on the button pointed to by the arrow in the Figure below.



Make sure you specify your run time in nanoseconds otherwise the testbench does not work as expected. For more information on testbenches see Chapter 10 in your textbook.

Assignment 3: Doing some calculations

Create a system where you can input a number in binary with the switches SW[5..9]. The number must be displayed on HEX6. Do the same with SW[3..0] and HEX4.

When you press KEY0, the sum of the two numbers must appear and stay on HEX1 and HEX0, even when releasing the button. For example if we insert 0xA+0xF and press KEY0, 0x19 appears, where the 1 is displayed on HEX1, and the 9 on HEX0. If the sum is smaller than 0x10 the leading 0 will not be displayed, and HEX1 must be dark (so for results 0x01, 0x02, … 0x0F, we don't see anything on HEX1, only on HEX0). Only if we press KEY0 again, a new calculation is made.

Assignment 4: Using the clock

Create a system where LEDR0 flashes every second with a duty cycle of 50%. If this works, add a counter going from 0…1…2…3…4…5…6…7…8…9...0…1…2…etc… every second over and over and display the value of the counter on a HEX display.

When you want to simulate, temporarily decrease the period in between counts, since in one second, using the 50 MHz clock, there are 50 million cycles. It will therefore take very long to simulate this.

Assignment 5: Implementing a simple Finite State Machine

Make a separate functional block for the flashing LED with an enable input and a speed_select input. When the enable input is high and the speed_select input is low the LED will start flashing every second with a duty cycle of 25%. When the enable input is high and the speed select input is high, the led will start flashing twice as fast with a duty cycle of 75%.

Also make a separate functional block to make a HEX display count up from 0 to 9. The block has a reset input that - when asserted - keeps the internal counter on 0 and the HEX display dark. If the reset is not asserted, then the block will start to count and a count_done signal will be asserted on the output of the block when it has reached 9.

Connect the flashing led enable and speed input to an output of a state machine. Also connect the reset input of the counter to the output of that state machine. Connect the count_done output of the counter to the input of the state machine. Also connect KEY0 as input and connect KEY3 as reset button of the state machine.

Use either a 7 segment display or the green LEDs to indicate in which state the FSM resides (for debugging).

Program the state machine to function like this:

In the initial state of the state machine (when reset is asserted), a LED must be flashing slowly. When you press and release KEY0, the HEX display will start counting from 0 to 9 (every second it will count up). While it is counting, the LED must be turned off. The HEX display will turn dark when it's done counting. Then, the LED will be flashing quickly. When you press reset everything must start over again.

Final Assignments

The final assignment is somewhat larger than the practice assignments. You may choose a final assignment as long as it is not picked by another student. You can also propose your own final assignment, but this has to be approved by the teacher.

For most assignments, sensors need to be connected to the DE1-SoC board. **Be careful, use ESD precautions, check voltage-levels and have the interface checked by a teacher!**

- **Lab 01**: Read the temperature from a **LM75** temperature sensor and display it on the 7-segment displays
- **Lab 02**: Read the temperature from a **MCP9808** temperature sensor and display it on the 7-segment displays
- **Lab 03**: Read the temperature from a **DHT22** sensor and display it on the 7-segment displays
- **Lab 04**: Read the Humidity from a **DHT22** temperature sensor and display it on the 7-segment displays
- **Lab 05**: Read the Temperature from a **DS18B20** temperature sensor and display it on the 7-segment displays
- **Lab 06**: Use a **LED-strip** and turn this into a reaction-meter. Every LED should correspond with a number of milliseconds. **Use a separate power-supply for the LEDs!**
- **Lab 07**: Use a **LED-strip** and turn this into a running-light. Use up/down buttons for faster/slower running and another button for direction switch (clockwise / counter clockwise). **Use a separate power-supply for the LEDs!**
- **Lab 08**: Use a **LED-strip** and turn this into a KITT-scanner. The speed of scanning is controlled by two switches. **Use a separate power-supply for the LEDs!**
- **Lab 09**: Use a **Neopixel ring** to build a kind of clock that displays seconds, minutes and hours. **Use a separate power-supply for the LEDs!**
- **Lab 10**: Use a **Neopixel ring** and turn this into a reaction-meter. Every LED should correspond with a number of milliseconds. **Use a separate power-supply for the LEDs!!**
- **Lab 11**: Use a **Neopixel ring** and turn this into a running-light. Use up/down buttons for faster/slower running and another button for direction switch (clockwise / counter clockwise). Use a separate power-supply for the LEDs!
- **Lab 12**: Read the barometric pressure from a **BMP280** sensor using the SPI-protocol and display it on the 7-segment displays
- **Lab 13**: Read the barometric pressure from a **BMP280** sensor using the I2C-protocol and display it on the 7-segment displays
- **Lab 14**: Read the temperature from a **BMP280** sensor using the SPI-protocol and display it on the 7-segment displays
- **Lab 15**: Read the temperature from a **BMP280** sensor using the I2C-protocol and display it on the 7-segment displays
- **Lab 16**: Read the code (button name) received by an **IR-receiver** and display it on the 7-segment displays
- **Lab 17**: Create a square on a **VGA-monitor** and have it running around by using switches.

- **Lab 18**: Read a **PS2-keyboard** and display the key pressed on the 7-segment displays.
- **Lab 19**: Read a **PS2-mouse** and display the coordinates on the 7-segment displays.