Design Flow.

The Development Tools
Questa and Vivado

Contents

1   Design flow and tools

This cookbook will take you through the basic use of powerful tools used in FPGA design, such as:

1. Questa Advanced Simulator. VHDL/SystemVerilog/System C/System Verilog simulator from Mentor Graphics. The simulator was formerly known as ModelSim.

2. Vivado Design Suite. Synthesis and Place & Route/Device fitting tool from the FPGA manufacturer Xilinx.

3. Xilinx Zedboard. A Xilinx Zynq based prototyping board. The Zynq family is a series of SOC's (System On Chip) containing both an embedded ARM processor and traditional FPGA fabric.

This "coolbook"-document is written as an aid to get through the design process from A to Z for the first time, without too much digression in the use of programs. Using this cookbook requires access to a ZedBoard developer board. Documentation for the ZedBoard can be obtained following this link: http://zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf

http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf
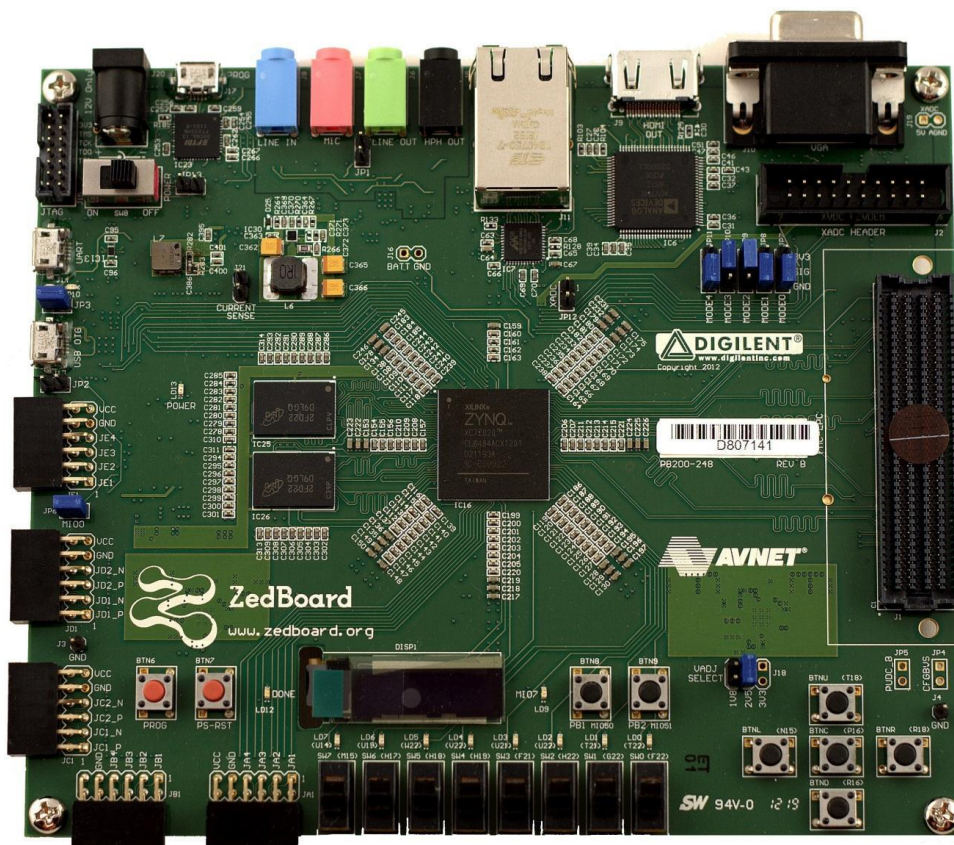

Figure 1. ZedBoard

1.1   Design flow for programmable logic

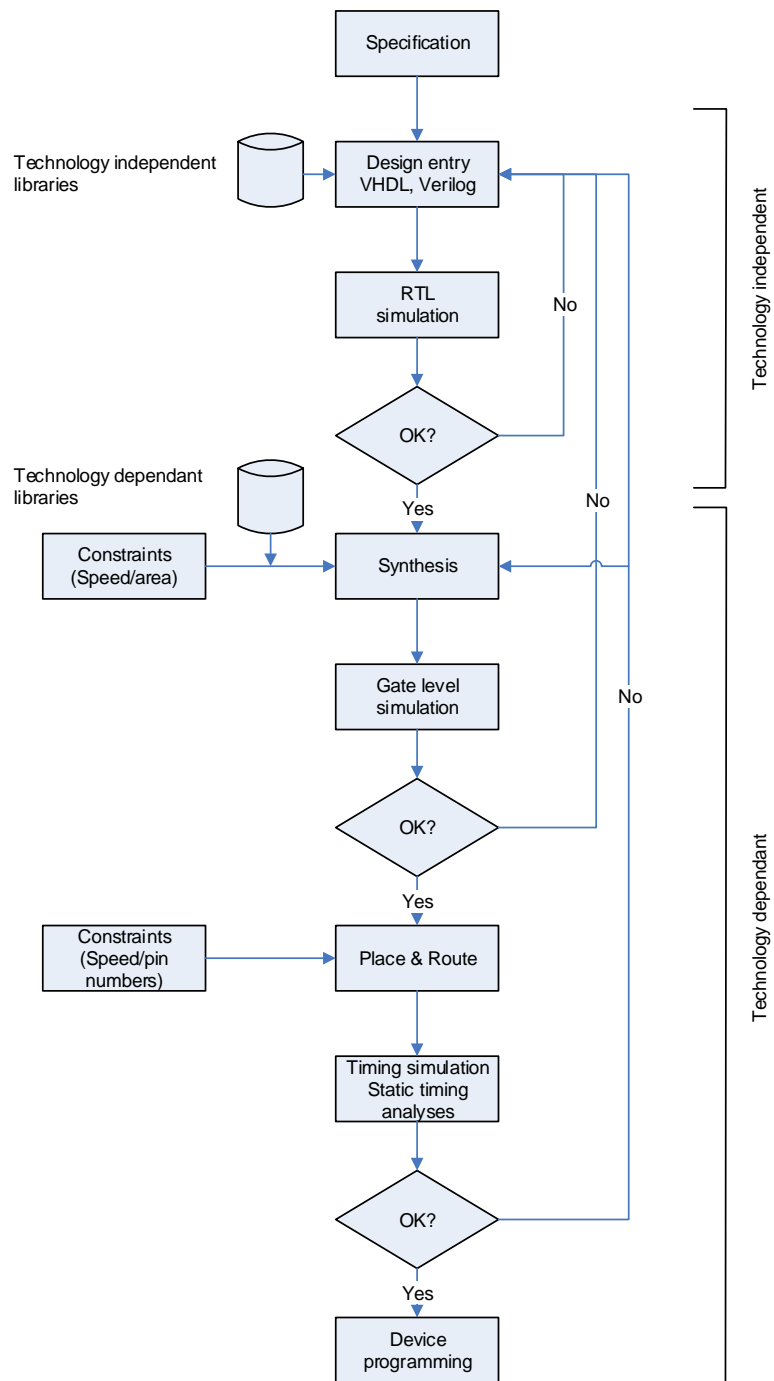The figure below shows the general design flow based on a HDL (Hardware Description Language) for an FPGA.

Figure 2. Design flow for FPGA

1.1.1 Design entry

There are several conceivable ways to describe a design. The oldest way is to draw a connection diagram (schematic) between components. This method has been abandoned for digital design, because it is a formidable job to draw-, scale- and maintain a diagram for a design that may perhaps contain several hundred thousand gates and flip-flops. Around the mid-1980s, a special programming language was developed to describe hardware, a so-called HDL (Hardware Description Language). VHDL was standardized in 1987, and has been refined several times later. It is much easier to maintain and expand a design that exists as text than one that exists as a graphic design. In this cookbook, we will be using VHDL for design entry.

1.1.2 Register Transfer Level (RTL) simulation

The first simulation in Figure 2 is a purely functional (logic) simulation of, HDL code. We must use a HDL simulator for this. This level of abstraction is called "register transfer level", RTL. Ideally, we do not need to consider which technology the design will be entered into at this level. Often, however, we must modify the code to take into account restrictions and limitations of the technology we have decided to use. We may have to take into account clock distribution, global reset, control of tri-state outputs, etc. When we have finished simulation and are satisfied, we can proceed to the synthesis (see 1.1.3). Most simulation and debugging should be performed at the RTL level because we have good observability there, and simulating RTL is much faster than it is simulating using technology specific data.

1.1.3 Synthesis

To perform synthesis, we have to decide on a technology (type of chip or chip family).

Synthesis is a process that transfers an RTL description to a gate level description. In an RTL description, all of the registers (flip-flops) and clocks are visible. The gate level refers to the connection of basic components (AND, OR, NOT, etc.) in a technology-dependent library. We can affect the result of the synthesis by creating *constraints*. These are requirements that we set in order to guide the tool in a desired direction. Typical synthesis constraints may be clock frequency and space constraints. Other constraints may be the type of encoding used for state vectors in Finite State Machines (FSMs). The result of the synthesis may also be Boolean equations if the design that can be loaded into a CPLD. Generally, it is possible to perform simulation after synthesis to verify that the result will work as intended.

1.1.4 Place and Route (Fitting) and timing simulation

If the simulation after the synthesis is correct, we can start the process in which the synthesized design is entered into the chosen technology. This process is called *Place and Route* for FPGAs and *Fitting* for CPLDs. The tools used in this process are made by the chip manufacturers. Assignment of pins and performance requirements (clock frequency) are important constraints in this part of the design process. After the design has gone through a Place and Route or Fitting process, we will usually want to verify both the functionality and timing of the design. It is possible to generate simulation files in various formats that can be used for this. If the simulation is in order, we are more or less finished.

1.1.5 Static timing analysis

Running time simulation of large designs can be time-consuming. An alternative that is often used is performing a static timing analysis. Special tools are available for this. A usual timing requirement is the maximum clock frequency. This is limited by delays in the combinational logic between the

registers. Other requirements may be the *clock to output delay*, input *set-up* and *hold* times. Static timing analysis tools identify the worst-case (critical) paths, to give an overview of the performance of the chip. Correct function can be verified based on a purely functional simulation of the *routed* design.

1.1.6   Device programming

The last step before testing the design physically is *device programming*, i.e. transferring the design into the desired chip. This can be done in very many different ways. For example, SRAM-based FPGAs require that we use an external PROM/EPROM/Flash EEPROM, etc. to create the chip configuration (design). Antifuse-based FPGAs can only be burned once, and it is not possible to make any changes. Antifuse is used more often in finished products (to reduce cost), than in a debugging phase. The programming file is downloaded to the chip from a PC via a programming-cable.

In a CPLD, the configuration is stored internally in the chip, and there are variations that are EPROM-based that are programmed electrically and can be deleted by UV light. There are also EPROM-based devices that can be programmed and deleted electrically. The most interesting variants are those that can be programmed and reprogrammed, while the chip is soldered to the finished board. They are called ISR (In System Reprogrammable) CPLDs.

1.2   Tools used in this cookbook

Since we will be using an FPGA from Xilinx with the associated tools, the design flow for this will be slightly different from the general flow. For Xilinx and other technologies, we can also use general synthesis tools. Examples of such tools are Precision from Mentor Graphics and Synplify from Synplicity. Common for these is that they can synthesize for many different target technologies, including ASICs (Application Specific Integrated Circuits). In this cookbook, we will use synthesis and place and route tools that are specific to Xilinx, and we will use the Questa Advanced Simulator from Mentor Graphics. The Questa simulator used to be called Modelsim, and the two simulators are identical for all practical purposes. They are completely general and independent of technology. Full versions of Xilinx Vivado and Questa will be installed in the laboratory. Free versions of both Vivado and Modelsim may be downloaded for personal use.

The free (device-limited) Vivado Webpack (requires registration) can be downloaded from

https://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html

A student version of ModelSim PE can be downloaded from
www.mentor.com/company/higher_ed/modelsim-student-edition

Note that if you use other versions than those what you find in the lab, you cannot assume that projects can be moved from machines with the latest version to machines with older versions. However, you can expect that source files and script can be used unchanged regardless of the program version[1].

---

[1] Note that VHDL 2008 has only been partially included in Vivado versions before 2019. In Questa and Modelsim there is one setting that toggles the support of VHDL 2008 on or off.

2 Getting started with Questa

Questa can be used alone or integrated with Vivado. First, we will look at how we can use Questa alone. The description below applies to Questa, and Modelsim PE and SE.

2.1 Creating a project

Project management is important in all CAD systems. The purpose of a project is to create an organized storage structure for all the files associated with a design.

2.2 Preparation for VHDL simulation with Questa

The description of Questa is based on a simple VHDL example, in which we model a 4-bit counter. The counter is reset asynchronously to 0 by the RESET signal and triggers on a positive edge of the clock CLK. In addition, the counter can be reset synchronously to the numeric value of the INP input by giving a positive pulse on the LOAD signal. When the counter reaches 15 (hexadecimal F), the MAX_COUNT signal becomes active.

First, we will look at how we can perform a functional VHDL simulation of the counter. This example is meant as an initial introduction to Questa and shows far from all the options we have there, just one possibility. After we have performed the first simulation, we will continue by using the same code in Vivado and synthesize and place the design onto a ZedBoard. After this we will run a timing simulation, and when this has verified that the chip works the way it should, we will program the chip.

2.3 Sample design file

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity FIRST is
  port
    (
      clk       : in  std_logic;          -- Clock signal from push button
      reset     : in  std_logic;          -- Global asynchronous reset
      load      : in  std_logic;          -- Synchronous load signal
      inp       : in  std_logic_vector(3 downto 0);  -- Start value
      count     : out std_logic_vector(3 downto 0);  -- Count value
      max_count : out std_logic           -- Indicates maximum count value
      );
end FIRST;

-- The architecture below describes a 4-bit up counter. When the counter
-- reaches its maximum value, the signal MAX_COUNT is activated.

architecture MY_FIRST_ARCH of FIRST is

  --  Area for declarations
  signal count_i : unsigned(3 downto 0);

begin
  --  The description starts here

  COUNTING :
  process (all)
  begin
    if load = '1' then
      count_i <= unsigned(inp);
    else
      count_i <= unsigned(count) + 1;
    end if;
  end process COUNTING;

  STORING:
  process (reset, clk)
  begin
    -- Asynchronous reset
    if(reset = '1') then
      count <= "0000";
    elsif rising_edge(CLK) then
      count <= std_logic_vector(count_i);
    end if;
  end process STORING;

  -- Concurrent signal assignment
  max_count <= '1' when count = "1111" else '0';

end MY_FIRST_ARCH;
```

Figure 3. Design file first.vhd

2.4 Sample test bench

Simulating an electronic design is based on impressing stimuli on inputs and checking that the chip gives a correct response on the outputs.

The most common and effective way to produce stimuli is to create a VHDL *testbench*. This means that we create all the stimuli input in VHDL. (Alternatively, simulation commands could be used). Use of a VHDL testbench offers many advantages, such as the fact that the simulation can be ported to another standard VHDL simulator, since it is not dependent on the simulator's command language. The use of VHDL testbenches is very flexible and effective, because the entire VHDL language is available. For example, a lot of the environment (other components, possibly an entire board) can be included in the testbench. The example below assumes that you have learned some VHDL, such as component

instantiation, but we have decided to include it here to illustrate a template for how a simple testbench can be created. We will return later to more advanced use of testbenches.

It is sensible to have naming conventions for file names. For example, we can call the testbench file tb_first.vhd, where tb_ indicates a testbench file.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity TEST_FIRST is
-- Empty entity of the testbench
end TEST_FIRST;

architecture TESTBENCH of TEST_FIRST is
  -- Area for declarations

  -- Component declaration
  component FIRST
    port
      (
        clk       : in  std_logic;        -- Clock signal from push button
        reset     : in  std_logic;        -- Global asynchronous reset
        load      : in  std_logic;        -- Synchronous reset
        inp       : in  std_logic_vector(3 downto 0);  -- Start value
        count     : out std_logic_vector(3 downto 0);  -- Count value
        max_count : out std_logic         -- Indicates maximum count value
        );
  end component;

  signal tb_clk       : std_logic                       := '0';
  signal tb_reset     : std_logic                       := '0';
  signal tb_load      : std_logic                       := '0';
  signal tb_inp       : std_logic_vector(3 downto 0) := "0000";
  signal tb_count     : std_logic_vector(3 downto 0);
  signal tb_max_count : std_logic;

  -- 50 Mhz clock frequency
  constant HALF_PERIOD : time := 10 ns;

begin
  -- Concurrent statements

  -- Instantiating the unit under test
  UUT : FIRST
    port map
    (
      clk       => tb_clk,
      reset     => tb_reset,
      load      => tb_load,
      inp       => tb_inp,
      count     => tb_count,
      max_count => tb_max_count
      );

  -- Generating the clock signal
  clk <= not clk after HALF_PERIOD;

  STIMULI :
  process
  begin
    tb_reset <= '1', '0' after 100 ns;
    tb_inp   <= "1010"   after HALF_PERIOD*12;
    wait for 2*Half_Period*16;
    tb_load  <= '1', '0' after 2*HALF_PERIOD;
    wait;
  end process;

end TESTBENCH;
```

Figure 4. Testbench tb_first.vhd

2.5 Creating a project in Questa

Start Questa from the start menu or the desktop. Select **File→New→Project** (possibly after a welcome greeting). Choose a name and a location for your project. Use a folder on your personal home folder (M:) for the location of the project. Leave the default library name with the value *work*.
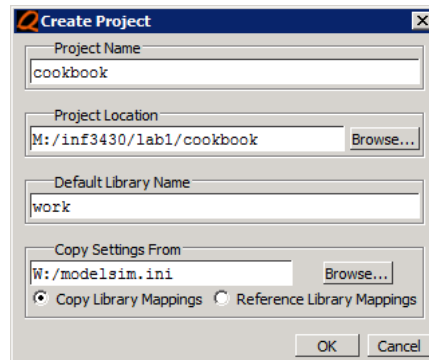

Figure 5. Creating a project in Questa

After you have created a project, you will be asked whether you want to add files to the project. This can also be done at any time. Usually, not all source files are ready during the initial simulation, so it may be necessary to add files later. Locate first.vhd and tb_first.vhd by selecting *Add Existing File* and then *Browse*, and select the files to be included by ctrl+click for each file to be added:
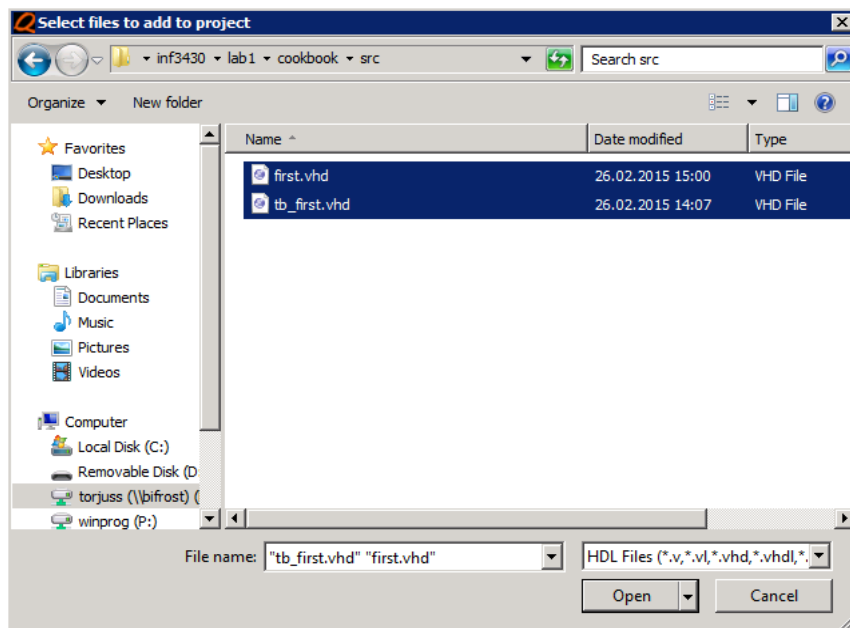
Figure 6. Add source files (1)

It is recommended to refer to the source files in their original location instead of letting Questa copy them into the project. It is good practice to keep the source files in a subfolder called *src*.

Note: By default, Questa will refer to the selected source files by their full absolute path (M:/inf3430/lab1/cookbook/src/first.vhd). In Figure 7 *the paths have been modified to a path relative to the project location (./src/first.vhd).* **This step is** *not* **necessary** *to make the project work but is useful* **if you ever decide to move the project***.*
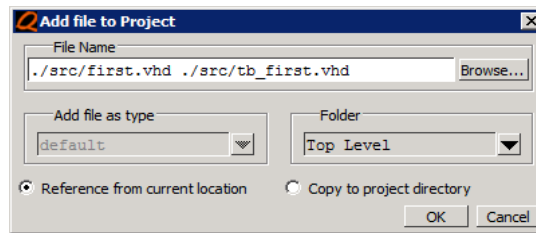
Figure 7. Add source files (2)

2.6 Compilation of source files

2.6.1 Working libraries in brief

A working library or *work* has a special meaning in VHDL. It refers to VHDL files that are under development or testing. It is very important in VHDL simulators and synthesis tools to know which library is working at any given time. We always <u>compile, simulate and synthesize</u> models that belong to *work*. We can have many user libraries, but only *one* library can be working. If we refer to work in a source file, it means that the simulator will expect to find the relevant design unit (more about design units later on in this course) in the library from which we load the design into the simulator. Vice versa, the library we load a design from is called the working library. This also means that VHDL requires that we actively keep track of libraries and to what libraries the design units are compiled. If we want to use design units from other libraries, we must refer to the logical name in the library. We will take a closer look at the use of libraries later on in this course.

2.6.2 Compilation of VHDL source files

To compile, you can select the desired files, **right click→Compile Selected.** If you want to compile all the files in the project, you can do so by **right click→Compile All**:
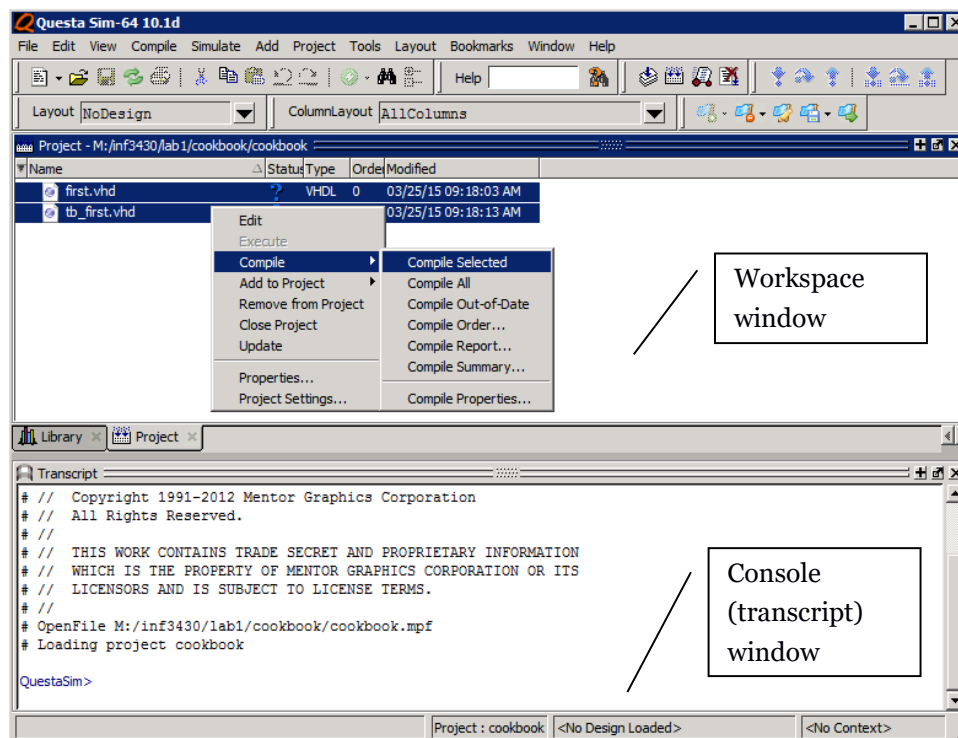

Figure 8. Compilation of VHDL source files

The probability of a syntax error during the first compilation is practically 100%. In Questa, as with other compilers, we have an *error parser* that shows us which line numbers in which files have syntax errors. There is a project option that controls whether the compiler output is displayed in the simulator's console window. It can be useful to have this option turned on. This is done by going to **Workspace→right click→Project Setting→check Display Compiler output**. Then you can click the error message, which will take you directly to the line with the error. Note that one small error often creates many consequential errors. In the example in Figure 9 below, a semicolon was missing and resulted in three error messages.

Some of the labs will require the sources to be compiled using VHDL-2008. Do the following to change VHDL version for compilation in Questa:

1. In the **Project** view, select all source files and right-click.
2. Select **Properties** from the drop down menu.
3. **Project Compiler Settings** is now open, go to the **VHDL** tab.
4. For **Language Syntax** select **Use 1076-2008**.
5. Press **OK**. The selected files will now be compiled with VHDL-2008.
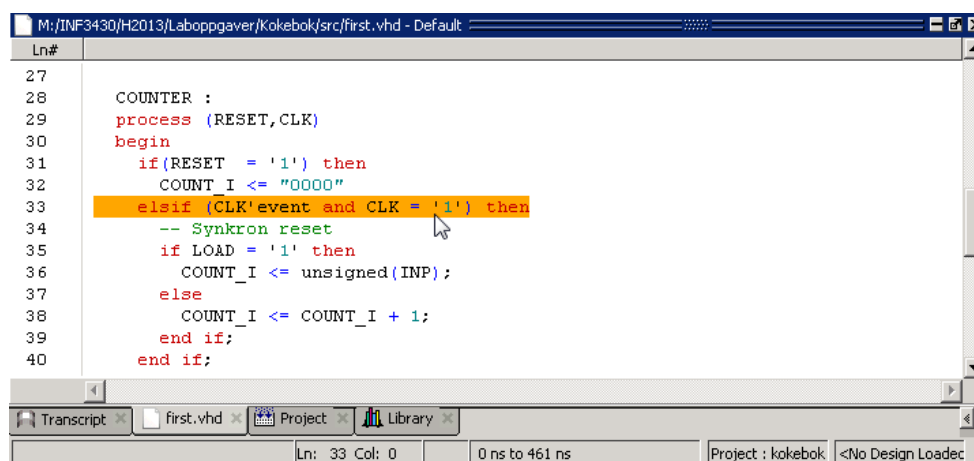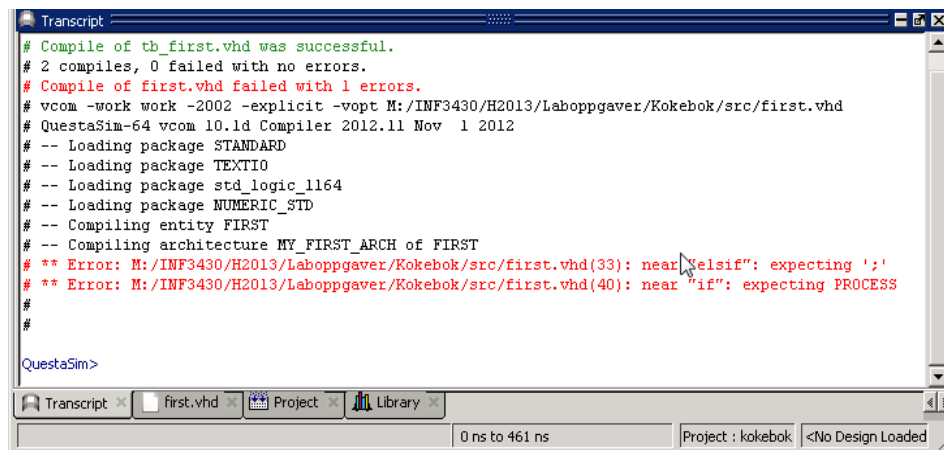

Figure 9. Error parser

2.7   Simulation

When the source files have been compiled, we are ready to simulate. Select the *Library* tab in the workspace, and go to the library named *work*, where each design unit has been compiled. To display

the design units in the various libraries, you can expand the library by clicking the [+] symbol before the library. We choose to load the entity *test_first*, which is our testbench entity. This represents the highest level in our design hierarchy. Select this entity→**right click**→**Simulate**.
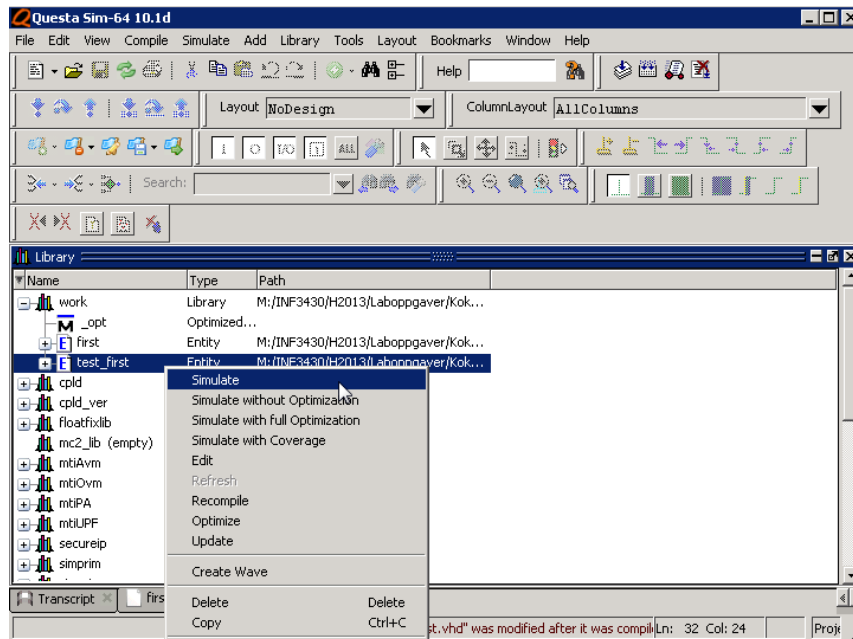

Figure 10. Preparation for simulation

When the design has been loaded, an *Objects* window appears. If it does not appear, you can display it by selecting **View**→**Objects.** From the Objects window, you can choose what signals you want to look at during the simulation. By displaying the S*im* window in the workspace and clicking the various design units, different *objects* appear that you can choose to look at. Select what you want to look at (all of the objects in this case) →**right click**→**Add to Wave**→**Selected signals (alternatively, you can select Signals in Region)**:
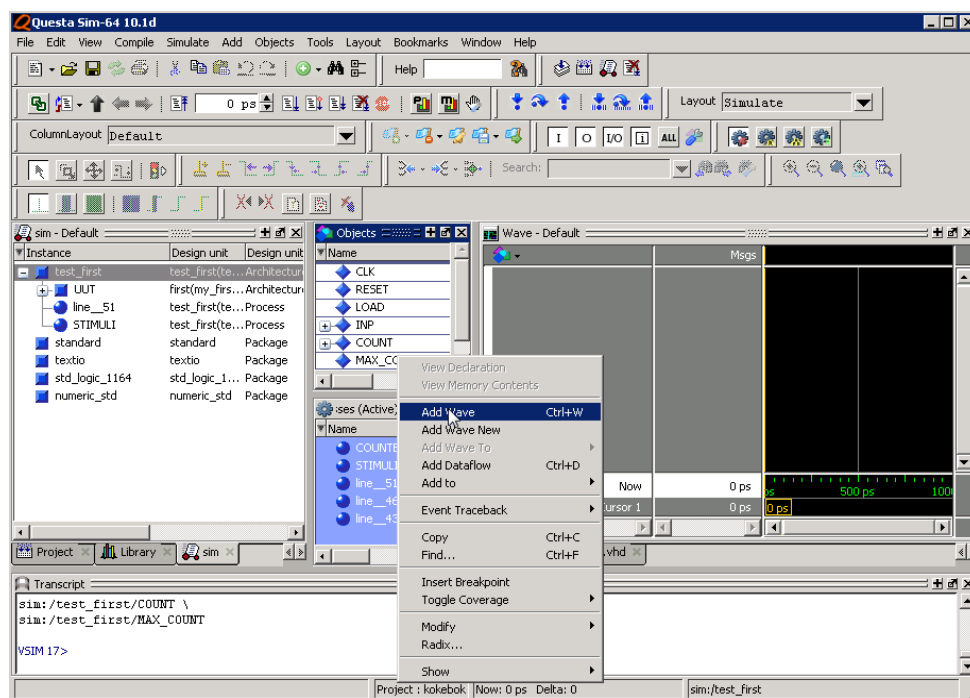

Figure 11. Selection of signals in the Waveform viewer

14

After you have selected the signals you want to observe, you can start a simulation. You can do this by entering the simulator command **run *<time>*** in the console (Transcript) window[2].

 For example, **run 1us** will simulate for 1 us. If you enter **run 1us**, you will simulate for an additional 1 us from the point in time you stopped after the last run command. If you enter **restart**, you will start from time 0 again, and all the *waveforms* will be reset to zero. Figure 12 below shows the signals from the counter after having simulated for 1 us.

Note: The wave window does not necessarily scale with the simulated time. To view the full time window of your simulation, *right-click in the wave window* and select **Zoom Full**[3].
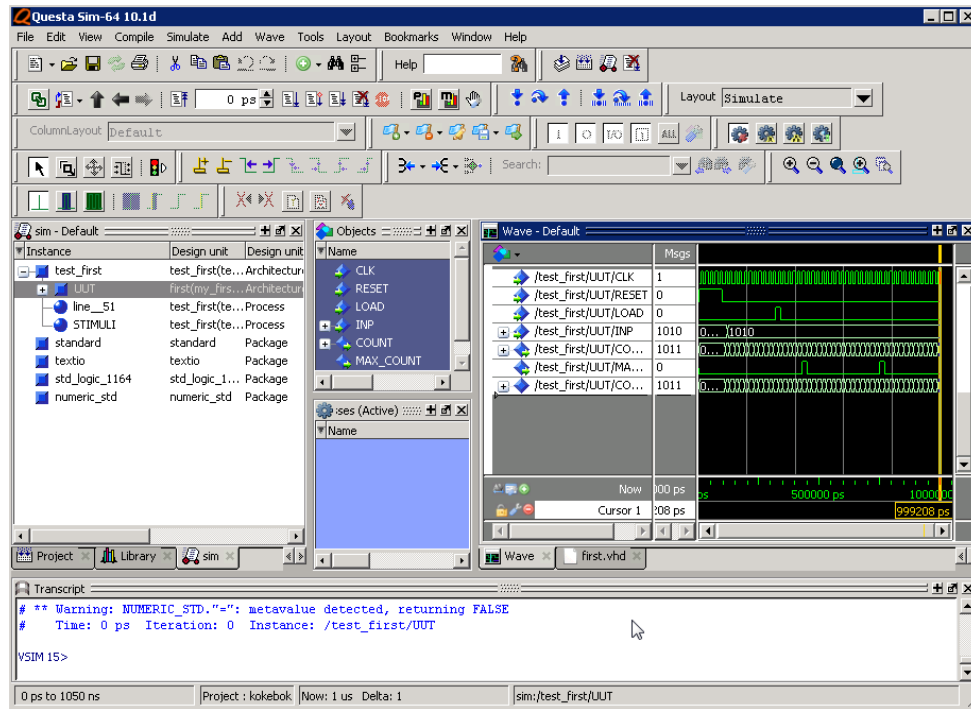
Figure 12. Waveform window

15

2.8   Macro files (do files)

Simulation is an iterative process. It may be sensible to automate large portions of the process. An alternative to entering or selecting simulation commands from menus is to combine commands in script or command files. In Questa you can create a script by means of the script language Tcl, or you can combine the simulation commands in a so-called do file. A do file often has the extension .do.

For example, it may be sensible to specify what signals you want to look at in the do file. Figure 13 is an example of a do file, sim_first.do. You can easily build up the contents of a do file by copying from the simulator's console window, where all the commands given are echoed. The first line loads the testbench entity (and the underlying design units) into the simulator. The next lines add signals to the *waveform viewer*, and the last line specifies that the simulation will last 1 us.

```
vsim -novopt work.test_first
add wave sim:/test_first/UUT/clk
add wave sim:/test_first/UUT/reset
add wave sim:/test_first/UUT/load
add wave sim:/test_first/UUT/inp
add wave sim:/test_first/UUT/count
add wave sim:/test_first/UUT/max_count
run 1 us
```

Figure 13. Do file sim_first.do

The do file in Figure 14 compiles the two files to the working library named work. Note that the full path is included in the file name here. If you replace your storage area or PC, you will therefore often need to update the do file. It is also possible to use a relative path, but then it is important that you are in the correct directory in Questa when you run the do file. You can find out where you are in Questa by entering **pwd** in the console window. The command **cd** also works to change the directory.

```
vcom -work work -2008 -explicit first.vhd
vcom -work work -2008 -explicit tb_first.vhd
```

Figure 14. Do file comp_first.do

The do files are run by selecting **Tools→Tcl→Execute Macro→Select do file**. Alternatively, the macro can be started by entering **do <filename>.do** in the console window. It is also possible to combine all the commands in a single do file, and it is possible for a do file to call up several other do files.

You are encouraged to experiment with the simulator. You can display many more debugging windows, we can set the breakpoints for code lines, and we can set breakpoints for signals when they have reached a certain value and much more. The Modelsim PE student edition is somewhat limited when compared to Questa.

Note that we have very good observability during RTL simulation. This is where we should spend time, because this is where most of the difficult errors can be detected.

3 Getting started with Vivado

Vivado Design Suite is Xilinx's tool for Synthesis, Place & Route and programming of their FPGAs. In each of these main task categories, there are a number of support tools. We will look at the most important in this summary. We will synthesize, assign pin numbers, create timing constraints, run place and route and program the chip.

Note: The dialogue windows may look slightly different, depending on the version in use.

3.1 Creating a project

Vivado also makes use of projects to organize files. We can use various types of projects in Vivado. For example, a design may be a mixture of diagrams and HDL modules, or it may be a pure HDL project. We will look at the latter type, having all VHDL source files.

Start Vivado from the start menu or from the desktop and choose *Create Project* from the greeting menu to start the New Project Wizard. Click *Next*.

When using an UiO based machine, make sure you store your files on your personal disk, rather than locally on the machine in use.
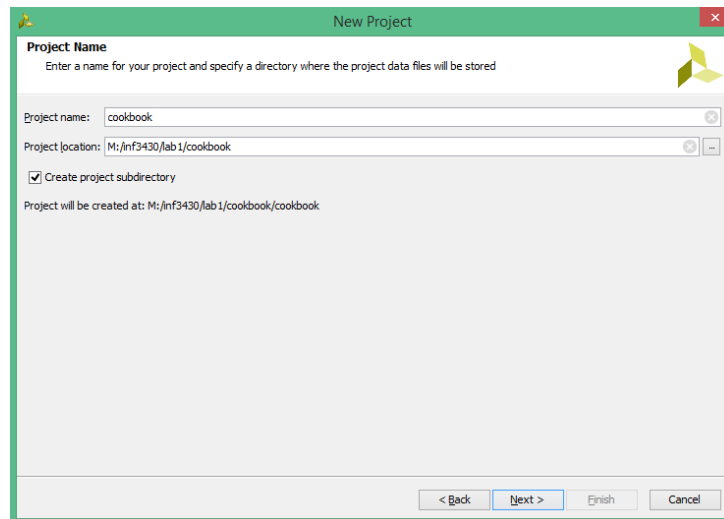
Figure 15. Project name. Add the project to the same directory as the VHDL files are located to avoid problems later on in the cookbook.

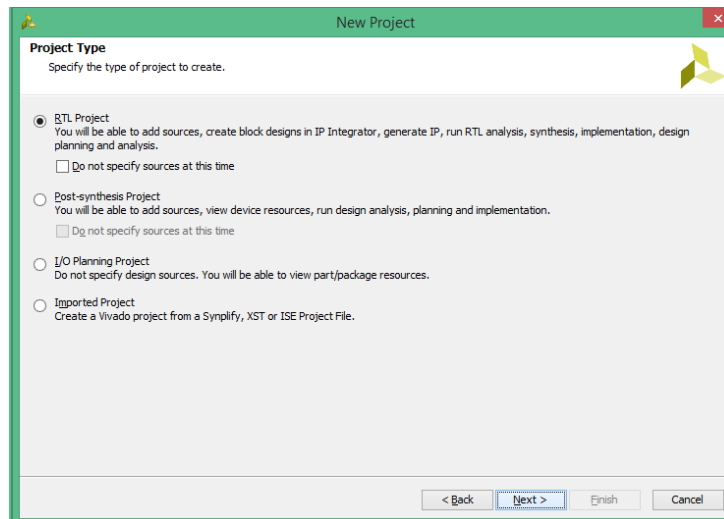Select *RTL Project* and click *Next*.

Figure 16. Project type

Click *Add Files*, locate the two files *first.vhd* and *tb_first.vhd*, select them and click *OK* to add the two source files to the project. In the column named *HDL Source For*, change the value for *tb_first.vhd* from *Synthesis & Simulation* to *Simulation only*. This is done because testbench files cannot be synthesized or programmed on the chip; they are only used for simulation. Uncheck *Copy sources into project* to let Vivado point to the source files in their original location instead of making a copy of the source file. It is also possible to add source files to the project later.

Make sure the *Target language* is set to VHDL. The simulator language is left as *Mixed*, indicating that it can be either VHDL or Verilog. Click *Next*.
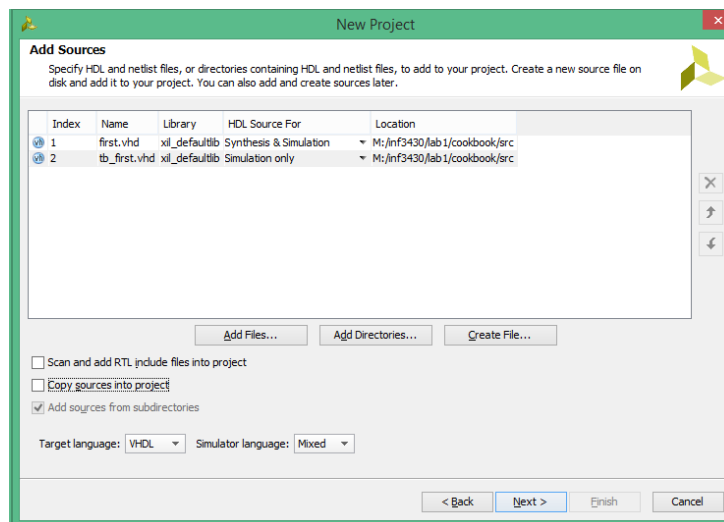
Figure 17. Add sources

Click *Next* when/if prompted for IP (Intellectual Property), as this will not be used in the current project.

Click *Next* when prompted for the location of constraints files, as we will be adding the constraints later in the project.

The choices made with respect to technology and tools can be made at any time during the design process. In this course, however, the choices have already been made. The board used is the ZedBoard

Zynq Evaluation and Development Kit, which uses a XC7Z020CLG484-1 FPGA. FPGAs are classified according to how fast they are, the higher the value of the speed grade the faster the chip is.

Use the values in Table 1 to select the right FPGA part number. Click *Next*.

Table 1. ZedBoard properties

Property	Value
Part	xc7z020clg484-1
Family	Zynq-7000
Sub-Family	Zynq-7000
Package	clg484
Speed grade	-1
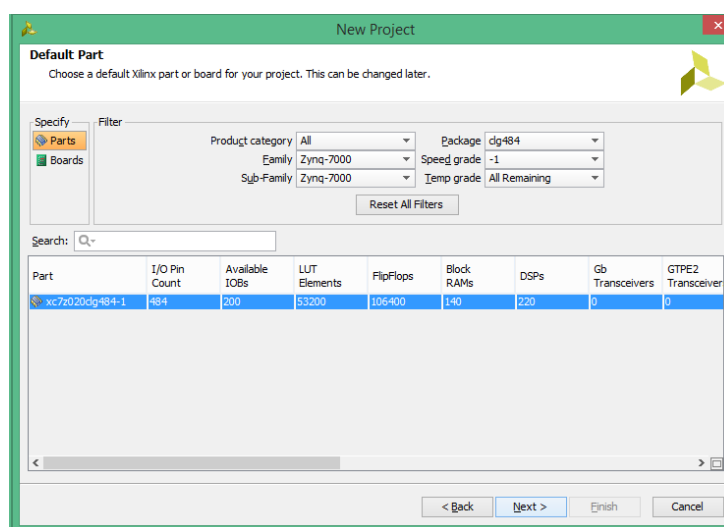
Figure 18. FPGA part number
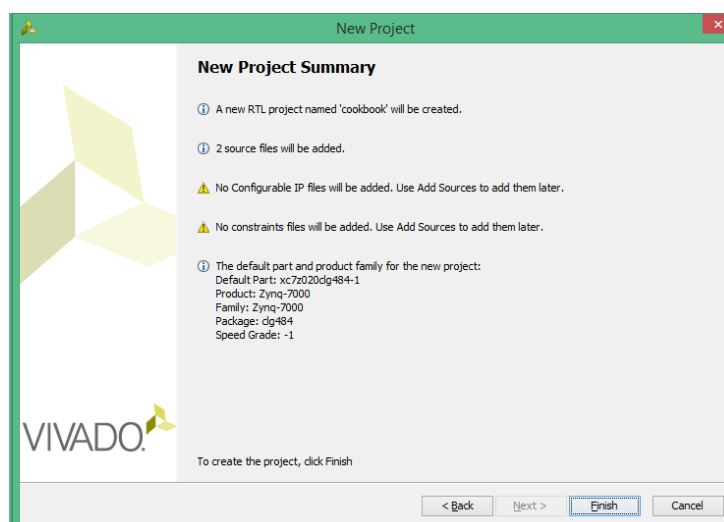
Figure 19. Project summary

3.2 Using Vivado

The *Flow Manager* is located on the left side of the Vivado environment. This is where the different processes of the project (synthesis, implementation and programming) are started.

There is an overview of the sources of the project located in the top middle of the window. The sources are grouped into *Design sources*, *Constraints*, and *Simulation Sources*.

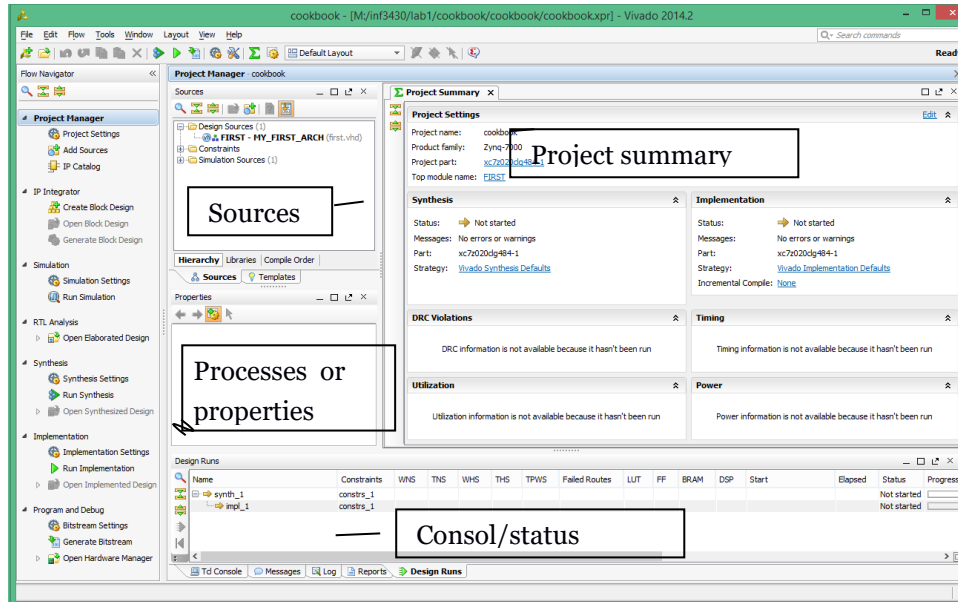A console window for entering Tcl commands is available by selecting the *Tcl Console* tab at the bottom.

Figure 20. Vivado environment

In the *Sources* window, check that the *Design source* is your design file, and not your testbench file. If the testbench is listed as design source, click on it, and go to the *Source file properties* (below) and uncheck *used in→Synthesis*. Check that the design file is selected as design source. When selected in the *Sources* window, the design file shall be *used in* both *synthesis* and *simulation*.

3.3   Constraints

3.3.1   Physical constraints (assigning pins)

Ideally, it should be possible start with synthesis and then the implementation. However, we must deal with the fact that *the design must be implemented in a chip that is mounted on a finished board. This means that all the pin locations are fixed*. The fact that the pin locations are defined in advance is often the situation in real life, because it is often desired that the printing of circuit boards and development of the internal workings of the FPGAs shall take place as parallel activities to save time in projects. Unfavorable pin locations may have a negative effect on performance. Since the pin location is already defined for us, we must verify this. This is done by using a Xilinx Design Constraints file with the file extension `.xdc`. In the XDC file, you can assign the top-level entity the right pin numbers, for example. The XDC file is a text file that can be edited directly in the text editor in Vivado.

If we want to create *pin number constraints* or *area constraints*, we can edit these graphically within Vivado. In our case, we will assign the signals of the entity FIRST to the pin numbers shown in Table 2. To do so, select *Open Elaborated Design* under *RTL Analysis* in the *Flow Navigator* on the left side of Vivado. After the design has been elaborated, select the *I/O Planning* layout from the dropdown menu on the toolbar.

NOTE: If you get messages of the type "[Synth 8-2757] this construct is only supported in VHDL 1076-2008", or "[Synth 8-1779] cannot read from 'out' object count: use 'buffer' or 'inout' "- VHDL 2008 has not been enabled. To do so: In the *Sources* window, select the design source *FIRST(MY_FIRST_ARCH)(first.vhd)* then go to the window below *Source File properties,* and click the '...' –icon next to *Type:,* and select "*VHDL 2008*", not simply VHDL. Repeat this process for the testbench (*Simulation sources/sim_1/TEST_FIRST...).*

Table 2. Pin assignment

Signal	Pin number	Signal name (on the board)
COUNT[3]	U21	LD3
COUNT[2]	U22	LD2
COUNT[1]	T21	LD1
COUNT[0]	T22	LD0
INP[3]	F21	SW3
INP[2]	H22	SW2
INP[1]	G22	SW1
INP[0]	F22	SW0
CLK	T18	BTNU
RESET	P16	BTNC
LOAD	M15	SW7
MAX_COUNT	U14	LD7

Expand the ports in the *I/O Ports* tab in the control window. (If these are not shown, try re-elaborate the design). Drag and drop the port from the *I/O* Ports to the selected pin in the *Package* window.
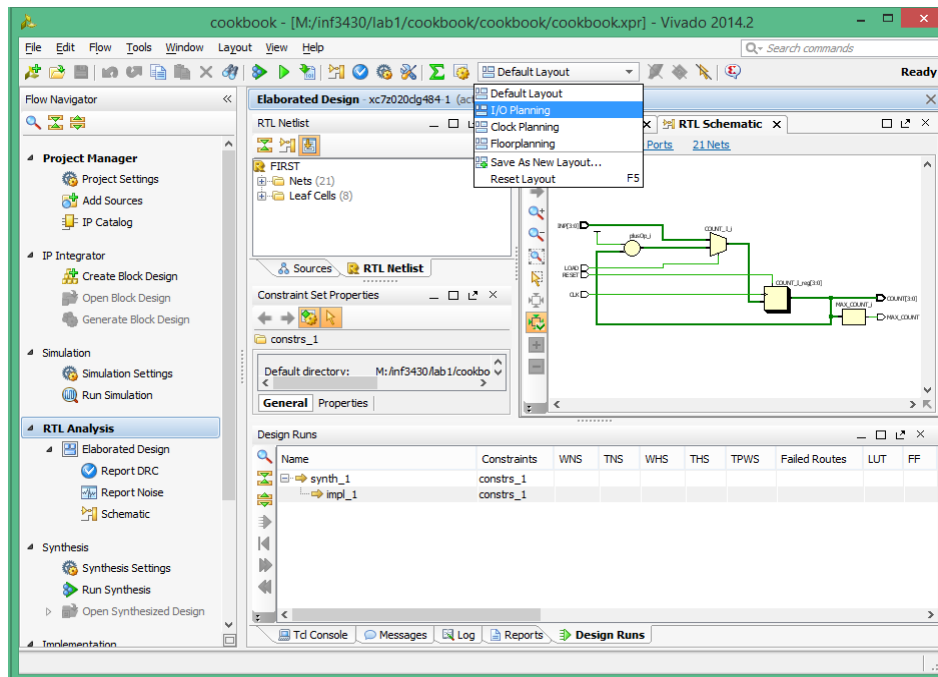
Figure 21. Selecting I/O planning layout

In the *I/O Planning* view, enter the physical pin constraints in the *Package Pin* column of the *I/O Ports* window at the bottom of Vivado. The assigned pins show up as orange rectangles in the *Package* window, the floor plan overview of the pins of the FPGA. To avoid future errors, the value in the I/O Std column must also be given. Change the I/O Std value for each of the pins from *default (LVCMOS18)* to *LVCMOS33* to define the voltage as 3.3 V.

Note: Although VHDL is not case sensitive, Vivado is (along with other HDLs such as Verilog). This is important when modifying the constraints file; constraints are compared case sensitive to the text in the VHDL entities. If you are in doubt, you should use the graphical user interface to get your constraints right, rather than modifying the .xdc file in the text editor.
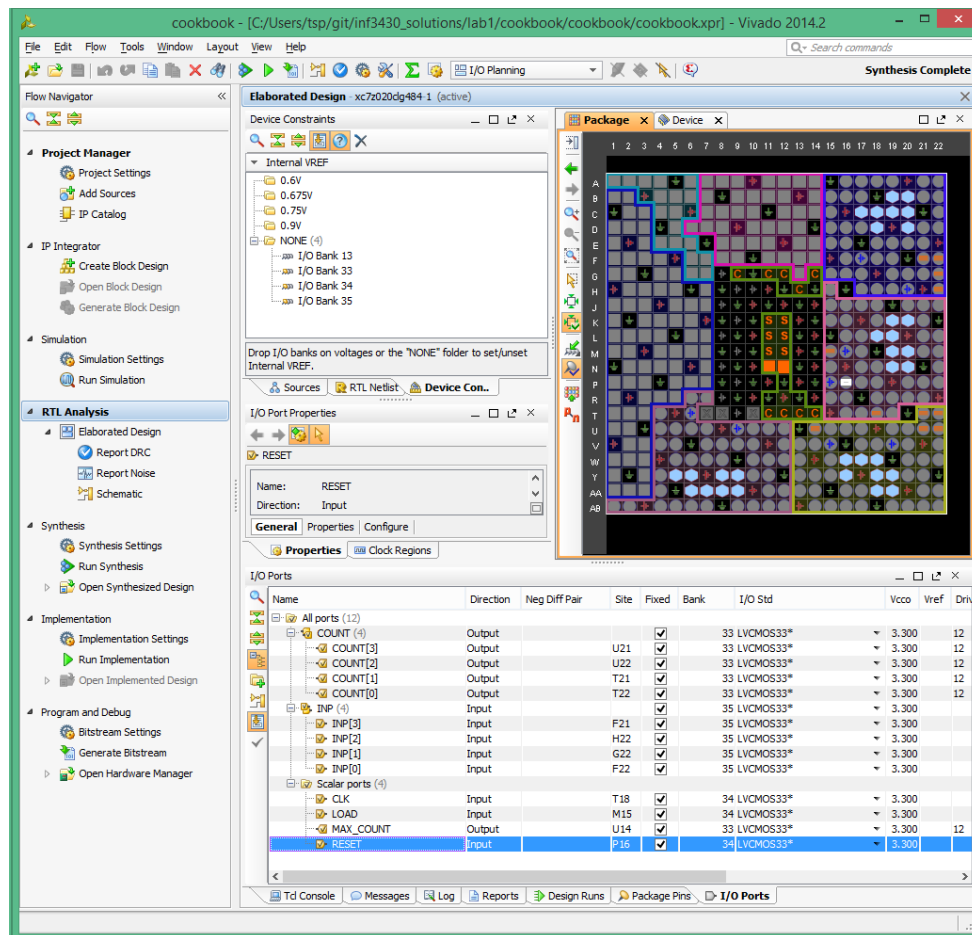
Figure 22. Physical pin layout

When you have finished entering the pin number constraints, select **File→Save Constraints**. You will be prompted to create a new XDC file. Call the file **constraints.xdc**.
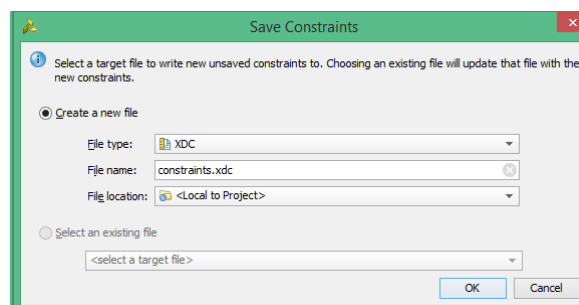
Figure 23. Save constraints

The constraint file can now be found and opened from the *Sources* tab in the *Sources* window.

Notice that we have assigned the clock to a push button. This pin is not a global clock input and thus it is not optimal for use as a clock input. Vivado gives an error message for this. To demote this error message to a warning, we must add the following line to the constraints file constraints.xdc:

```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk]
```

After modifying the constraints file, Vivado notifies us that the elaborated design is out of date, and prompts us to reload the constraints file. Click **Reload** to reload the constraints file.
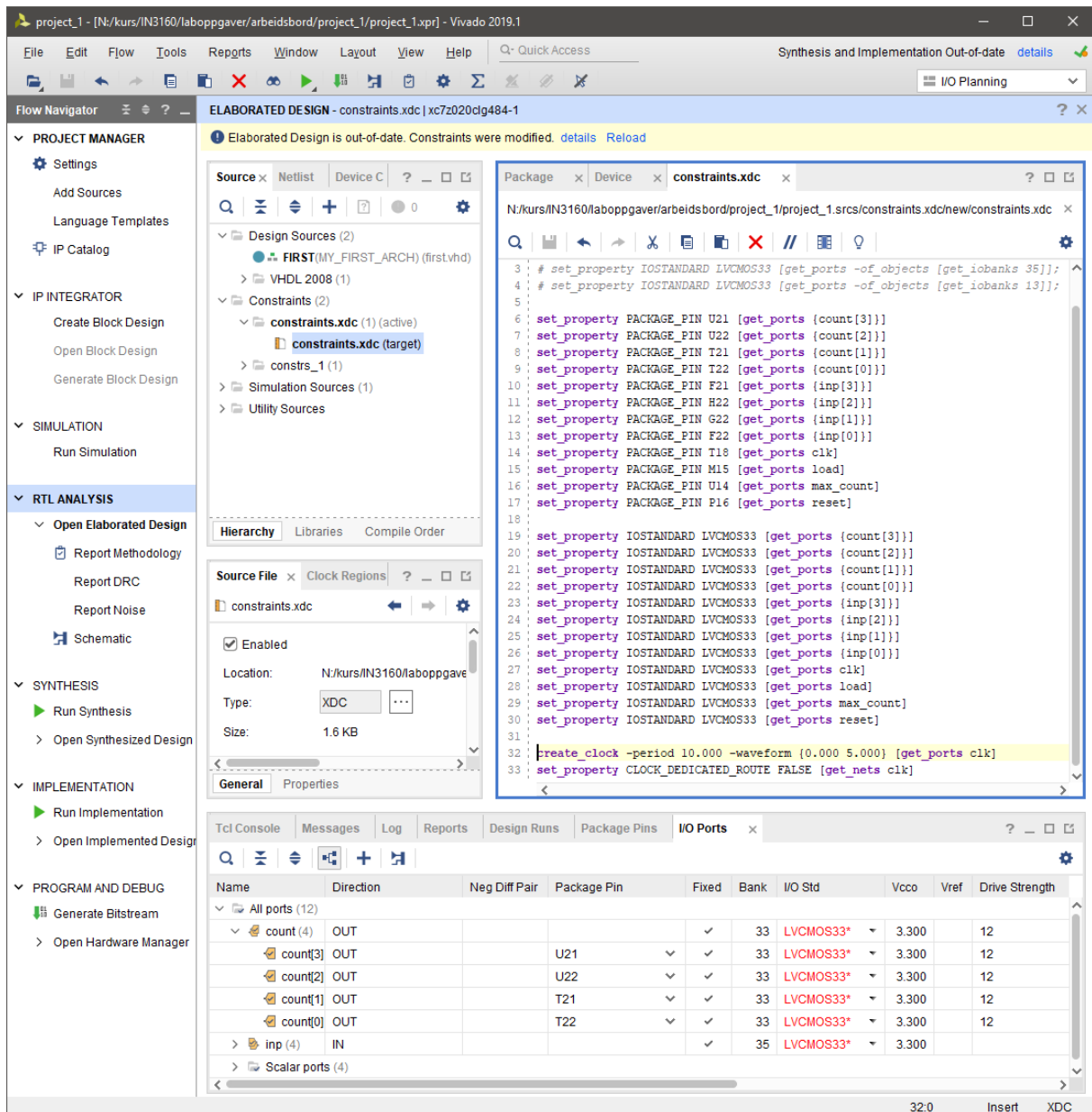
Figure 24. Reload constraints

In subsequent exercises, we will use a 100 MHz crystal oscillator as a clock. That is connected to a global clock input.

In Figure 24 the property IOSTANDARD is set for each pin individually. The pins are arranged in different banks on the FPGA, and the IOSTANDARD may *optionally* be set for a whole bank at a time. What banks the individual pins belong to may be explored in the I/O Planning view. To set the bank voltage of bank 33 you would need to type the following into the XDC file:

```
set_property IOSTANDARD LVCMOS33 [get_ports -of_objects [get_iobanks 33]];
```

This line should be typed *after* all PACKAGE_PIN constraints within a target bank have been evaluated.

3.3.2   Timing constraints

After we have finished with the pin assignment, we can add other constraints, such as timing constraints. We will now limit ourselves to adding a timing constraint, namely the period of CLK. There is a 100 MHz clock oscillator of on the board, so it is natural to select a period time of 10 (ns).

Run **Synthesis→ Run Synthesis** (This may take some time to complete)

Select **Synthesis → Synthesized Design → Edit Timing Constraints**. A timing constraints editor appears. (The timing constraints editor can also be reached under **Implementation**.)
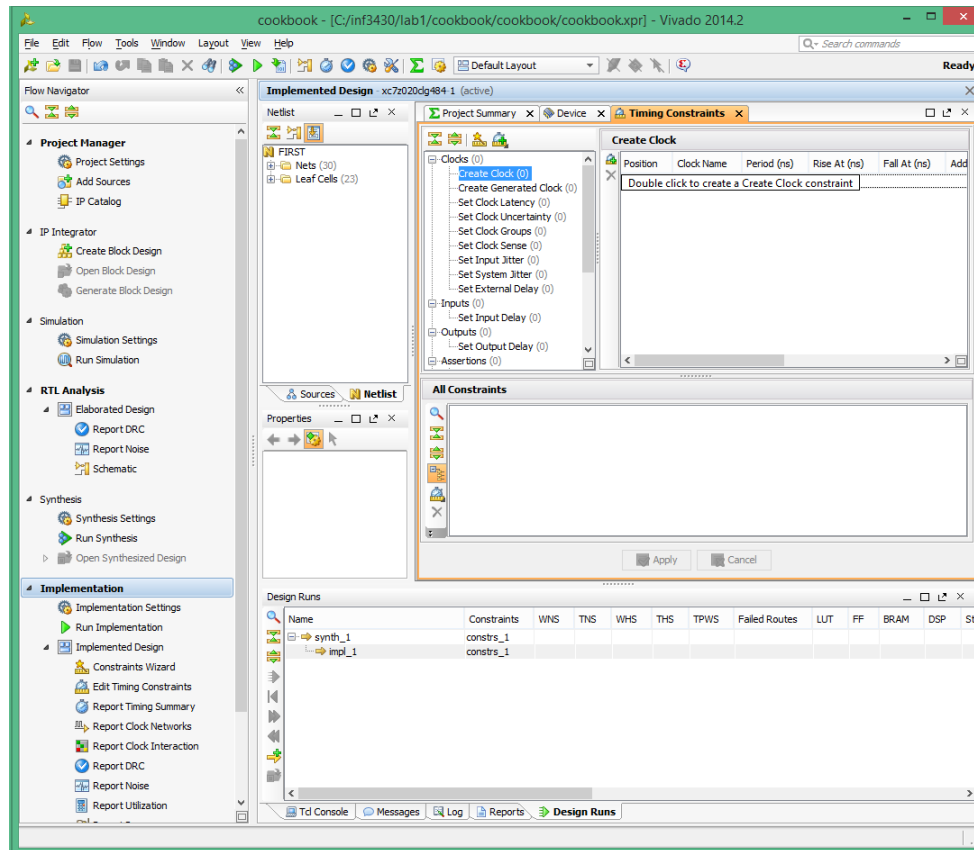
Figure 25. Edit timing constraints

Double click in the timing constraints editor to create a Create Clock constraint. Click the browse [...] icon to the right of the **Source objects** field. Click **Find** in the window that opens. Choose the CLK port, and click **SET** (**OK)** to close the window.
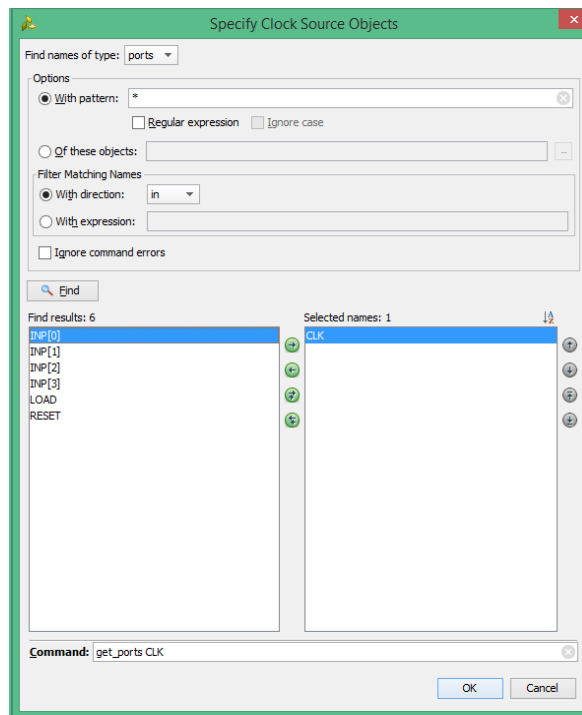
Figure 26. Specify clock source

Enter 10 ns for the period, and leave the rest of the fields as default values. Click **OK**.
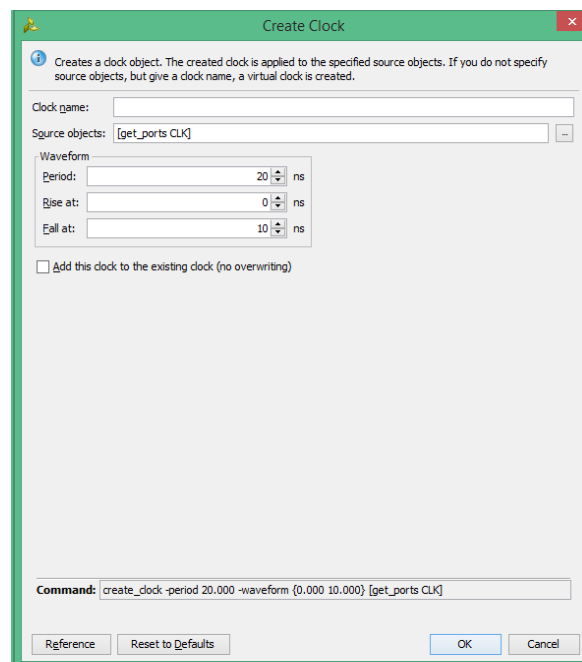
Figure 27. Clock period

Save the timing constraints. The file constraints.xdc is updated with the new timing constraints. The synthesis and implementation tools will now try to satisfy the constraints we have added.

When you have become more experienced, you can just as easily edit the XDC files in a text editor, using a XDC file you have created earlier as a template.

To edit the constraints file in Vivado's internal text editor, go to **Constraints → constrs_1** in the source file overview tree and double-click on **constraints.xdc**.
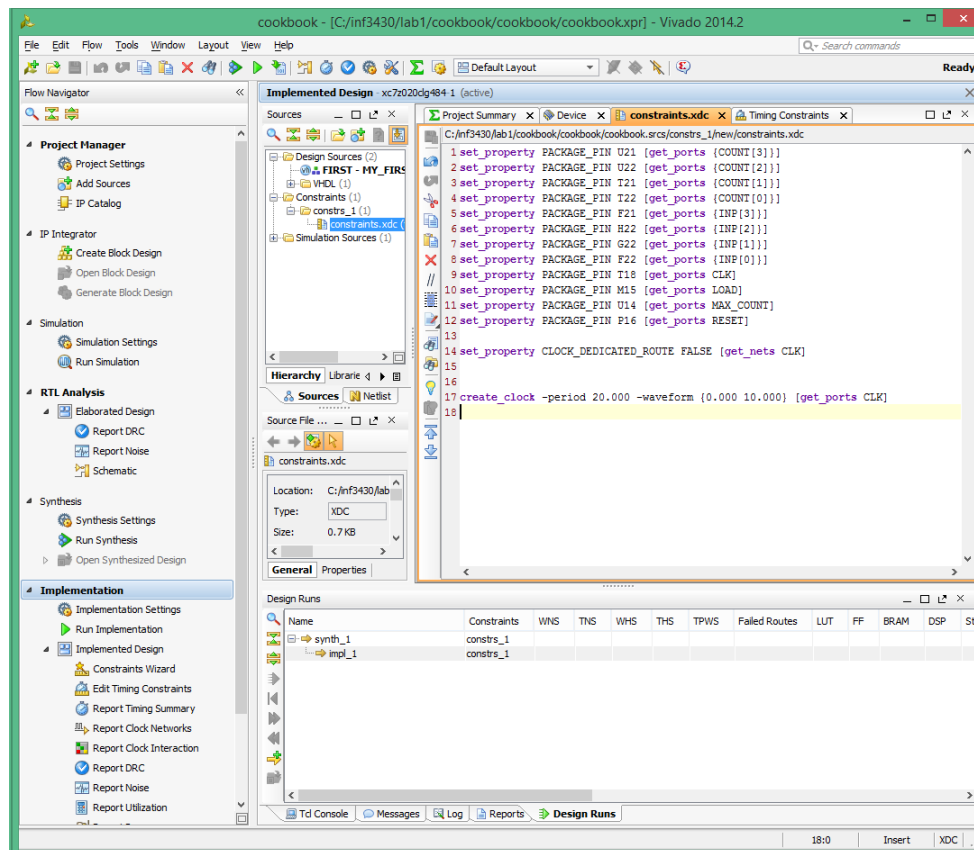
Figure 28. Editing the XDC file

3.3.3 More on timing constraints (*not needed for first.vhd*)

Often a design has more than one clock and one clock domain. Signals and data may then move between clock domains. Vivado will then require that the designer add some constraints to define the relationship between the different clock domains.

The previous section explained how to add a clock constraint for a primary clock to the constraint file. In this section, we will add generated clocks to the constraint file. We will also add constraints that defines the relationship between clock domains and use properties to define synchronization registers.

A generated clock is very often the primary clock divided or multiplied (or both) with a fixed value, but it can also have a different phase and/or waveform.

The primary clock is defined in the previous section. We add the generated clock to the constraint file by using the Timing Constraint view (**Synthesis → Synthesized Design → Edit Timing Constraints**). Under **Clocks** select **Create Generated Clocks** and then press the plus icon to add a new constraint.
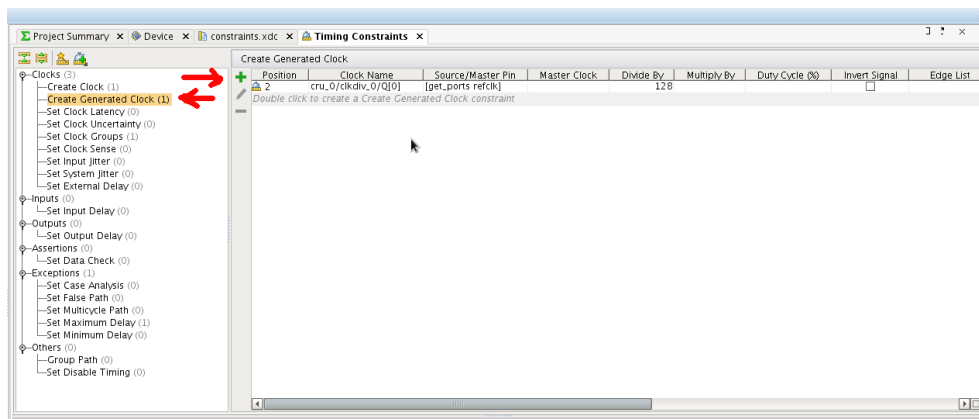
Fill in the following in the window that appears:

- **Clock name** is the optional name for the clock you are constraining.
- You have two options when it comes to clock source. You can use a clock input pin as a source (**Master pin**) or an already defined clock (**Master clock**). The easiest is to press **...** on the **Master clock** and find the primary clock you are using as source for the generated clock. Enter **clk** in **Pattern** and press **Find**, *refclk* should appear in the list of results. Select the wanted clock and press **OK**.
- Set the values for how much the primary clock shall be divided. In this case it is 128.
- **Source objects** is the source of the generated clock. This is usually the output pin of a register or primitive. Push **...** and search for the name of the wanted source. Then select it from results and press **OK**.
- Press **OK**. Save. The generated clock is now constrained.

If we have two clocks in a design and we want to tell the tool to view the two clock domains as asynchronous, do the following:

- Open the timing constraint window and select **Set clock groups**. Push the plus icon.
- A window named **Set Clock Groups** pops up. Give the group a name in the **Group Name** field.
- Now you should add groups. Press **...** behind **Group 1**. Search for your primary clock and add it to the group.
- Back in **Set Clock Groups**: Press the plus icon behind **Group 1** to add another group.
- Do the same as you did for **Group 1** for **Group 2**, but with a different clock
- **The specified clocks are**: select asynchronous.
- Press **OK**. Save.
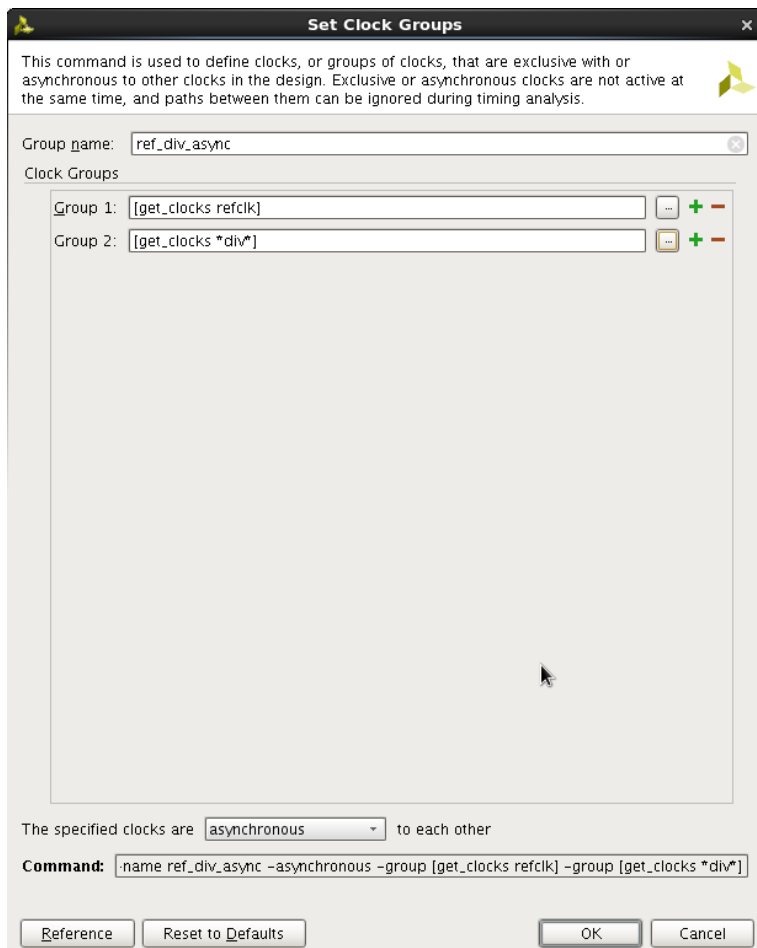
 For more information read the constraint user guide **UG903**.

3.3.4   Property for clock domain crossing (*not needed for first.vhd*)

Signals crossing clock domains shall be synchronized by using double flip-flops to bring the signal into the new clock domain. It is highly recommended to have these registers placed close. Vivado has a property constraint that should be used to tell the tool which registers are used for synchronization and be placed close.

By giving the signals used for clock domain crossing names which includes *s1* and *s2*, we can use a search pattern to assign signals containing those strings with an **ASYNC_REG** property. As an example:

To add the property constraint, do the following:

* In Vivado: Open the constraint file for editing. In a previous section we called it: **constraints.xdc**.
* Go to the bottom and add the following two lines:
* *set_property ASYNC_REG TRUE [get_cells -hierarchical *_s1*]*
* *set_property ASYNC_REG TRUE [get_cells -hierarchical *_s2*]*
* Save and close the file.

It is possible to use other naming conventions than *s1/s2*. The registers are found with *get_cells –hierarchical <some pattern>*. It is possible to try the *get_cells* command in the Vivado Tcl Console to view which registers that are returned by the search pattern.

Properties are further explained in **UG912 Vivado Properties**.

3.4 Synthesis

Clicking **Synthesis → Run Synthesis** will start the synthesizing process. The result of the synthesis is a gate level netlist, i.e. a connection of basic elements. The result of the synthesis is summed up in a separate report file. The *Synthesis Report* can be accessed from the *Report* tab at the bottom of Vivado. The report contains statistics about which types of basic elements that are used, and an estimate of performance. A schematic of the synthesized design can be viewed by clicking **Synthesis → Synthesized design → Schematic**.

3.5 Design implementation

After the synthesis, we can proceed to implementing the design in the selected chip. This implementation consists of three sub stages:

- Optimization
- Placement
- Routing

3.5.1 Optimization

This stage performs a logic optimization in preparation for placement and routing. The objective of optimization is to simplify the logic design before committing to physical resources on the target part. Logic optimization performs a netlist connectivity check to warn of potential design problems such as nets with multiple drivers and un-driven inputs.

3.5.2 Placement

During the placement stage a general placement of the overall design is performed. Timing and pin constraints (and especially area constraints) are important so that the result of this process contributes to fulfillment of the constraints. The placer engine positions cells from the netlist onto specific sites in the target chip.

3.5.3   Routing

After the placement process, we go over to the routing phase, in which the various basic elements are connected together. Timing constraints are used a great deal here to ensure that time delays in the connections do not exceed the constraints.

Implementation is started by selecting **Implementation → Run Implementation**. All sub-processes will be performed provided there are no errors during the process. After completion of the implementation process, a number of reports are generated. The reports are available from the **Reports** tab at the bottom of Vivado. (IO report is found under Place Design in the reports tab)

3.6   Device programming

To program the FPGA, we first need to generate a bitstream file which can be loaded into the FPGA. To generate the bitstream, go to **Program and Debug → Generate Bitstream**. This creates a file called FIRST.bit which will be used to program the FPGA.

3.6.1   Connections

For testing and debugging it may be appropriate to load the design directly into the chip without having to program external memory. All Xilinx FPGAs and CPLDs can be programmed via JTAG. JTAG, or IEEE1149.1 Boundary Scan, is actually a standard that has been developed to test connections on a circuit board by shifting bits through a long shift register, which goes through several chips. JTAG has gradually become a common programming interface for FPGAs, CPLDs and microcontrollers. We will take a closer look at JTAG later on in this course. See also http://www.jtag.com.

3.6.2   Downloading the configuration directly to the FPGA

Connect a USB cable from the computer running Vivado to the programming USB port of the ZedBoard. This is the USB port which is placed next to the board's power connector, and is labeled *PROG*. Make sure power is connected to the ZedBoard, and that the power switch is switched *ON*.

Select **Program and debug → Open Hardware Manager → Open Target → Open New Target** to start the *Open New Hardware Target Wizard*. Click **Next**. Select **Local server** in the **Connect to** field and click **Next**.
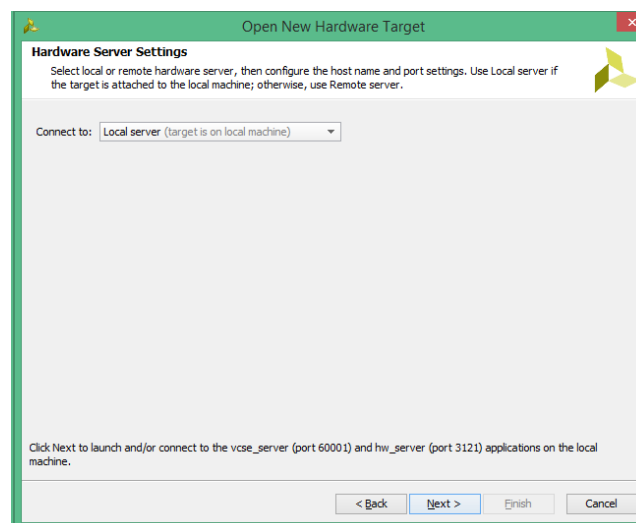

Figure 29. Hardware server settings

If ZedBoard is connected by the programming USB port, the next screen will show the hardware target to be programmed, as shown in Figure 30. Leave the fields as default values and click **Next** twice.
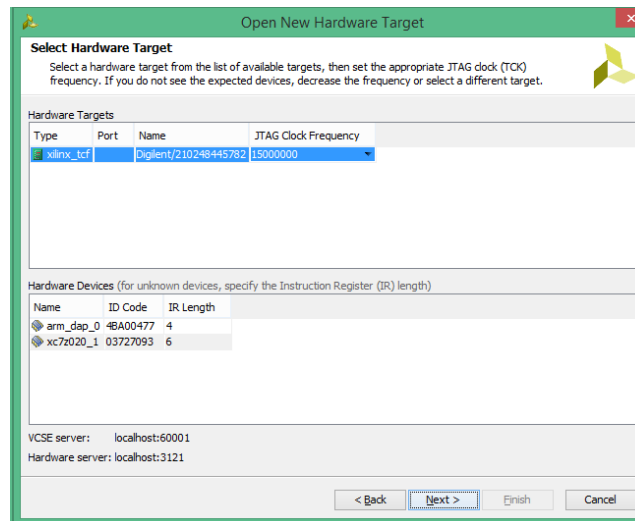

Figure 30. Select hardware target

To program the FPGA, select **Program and Debug → Hardware Manager → Program Device → xc7z020_1**. In the *Program* Device dialog box that opens, the fields are populated with the paths to the FIRST.bit file and another debug probes file. Leave the fields as they are, and click **Program**.
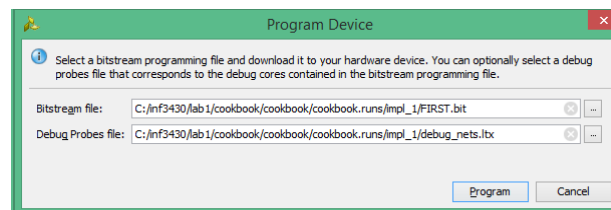

Figure 31. Program device

The board is programmed, and a blue LED labeled *DONE* lights up. You can experiment with the design after it has been loaded onto the FPGA. We have a counter that counts up to 15. At the next flank, it will count round and restart at 0. When LOAD is at 1, the counter will be set synchronously to the number on the INP inputs when you give a positive flank on CLK. Note that when we use a push button as a clock, we may experience false clock pulses at times. This is bouncing, i.e. the metal in the switch goes between the on and off state before it stabilizes in one position.

Note: In the Zedboard_HW_UG_v1_1.pdf, the pins are listed in chapter 2.7 User IO.
The pin numbers from Table2 and your constraint file may refer to these. Ie at the Zedboard programming first.vhd without modifications, we have out clock signal on BTNU. Try pushing this, and see if the LEDs will show the counting process, and if you are able to use LOAD.

4 Editing VHDL code

This section covers useful information concerning the editing of VHDL code and other text files in general. This is information that should be used when you want to edit files in the laboratory exercises. Test these tips on a VHDL source file, and learn the keyboard shortcuts.

4.1 Choice of editor

The text editors that are integrated in Vivado and Questa are somewhat cumbersome to use. We recommend that you use an external editor to improve the flow and control over the editing. In the lab, we have installed Notepad++ (http://notepad-plus-plus.org), which a powerful, free text editor. It supports, for example, selecting blocks and commenting on multiple lines of VHDL code. To get this to run automatically from Vivado, go to **Tools - Options → General → Text editor → choose Notepad++**.

4.2 Indenting

Indenting is moving the code different distances from the margin to make it more readable. It is not necessary in order for the VHDL code to be understood by the computer, but it is completely necessary so that other people can understand the code that has been written. We can find a lot of examples of how this is done in the VHDL book. All code that is submitted in this course shall be indented well.

1.1.1 Example of un-indented and well-indented code:

```
COUNTER :                                  COUNTER :
process (RESET,CLK)                        process (RESET,CLK)
begin                                      begin
if(RESET  = '1') then                        if(RESET  = '1') then
COUNT <= "0000";                               COUNT <= "0000";
elsif (CLK'event and CLK = '1') then         elsif (CLK'event and CLK = '1') then
if LOAD = '1' then                             if LOAD = '1' then
COUNT <= INP;                                    COUNT <= INP;
else                                           else
COUNT <= COUNT + 1;                              COUNT <= COUNT + 1;
end if;                                        end if;
end if;                                      end if;
end process COUNTER;                       end process COUNTER;
```

Poor (no) indenting   **Good indenting**

4.2.1 Tab/space for indenting

If you use the tab key to indent program lines, you just insert a tab code in the document, not a certain number of spaces (as is done if you press the space bar). How the program is displayed then is up to the various editors. The use of tab has both advantages and disadvantages, but one thing certain is that it can quickly look very strange in other editors if a program is written with both tabs and spaces indiscriminately. *Therefore, it is very important that you do not mix these forms of indentation.*

In this course, we have decided to establish a standard that only spaces are to be used for indenting, and that 2 spaces should be used for each level. The tab key can be set to insert a certain number of spaces instead of the tab character in the document. This is how you can avoid pressing the space bar twice. In Notepad++ you can enable this in **settings → preference → edit components → Tab size** and selecting **replace by space**. **Tab size** should be set then at 2. In Notepad++ you can easily check whether tab or space has been used in the document by selecting **view → show whitespace and tab**.

```
26   ·
27   ·  COUNTER · :
28 ⊟ ·  process · (RESET,CLK)
29   ·  begin
30 ⊟ · · ·  if(RESET · = · '1') · then
31 |——→——→COUNT · <= · "0000";
32 ⊟ · · ·  elsif · (CLK'event · and · CLK · = · '1') · then
33   · · · · |· · ·  -- · Synkron · reset
34 ⊟——→if · LOAD · = · '1' · then
35   · · · · |· · · · · COUNT · <= · INP;
36 ⊟ · · · · |· ·  else · · ·
37 |——→ ——→COUNT · <= · COUNT · + · 1;
38   · · · · |· ·  end · if;
39   · · · · |  end · if;
40   ·  end · process · COUNTER;
41   · ·
```

Figure 32: Here we see that both space and tab have been used (arrows show the tabs and dots show the spaces). This should be avoided.

4.3   Comments in the code and variable names

The use of comments is important to make the code more readable. This is important with a view to cooperation projects and reuse. It is not always easy to remember what one-year-old code does, even if you have written it yourself. Then some comments can be helpful.

However, the use of comments is no excuse for creating cryptic variable/signal names. Try to give descriptive names to the input/output signals and other elements in the code. Very short names should only be used when the scope is short, for example, within a simple process. If the code is self-explanatory with good variable names, comments everywhere are not required.

4.4   Use of keyboard

It is strongly recommended that you familiarize yourself with the use of the keyboard and shortcut keys for editing. This will enable you to edit faster, more easily and more ergonomically. The use of a mouse to copy and paste a line is cumbersome. Notepad++ follows the standard shortcut keys for editing in Windows, so it is not a waste of time to learn these in any case.

4.4.1   Keys and combinations you must know

Keys:

- **home**, **end**, **page up/down**
- **insert**, **delete**
- **tab**

Combinations:

- **ctrl** + **arrow keys/home/end** for faster navigation
- **shift** + **arrow keys** to select text (**ctrl** and **home/end** also work here)
- **ctrl** + **x/c/v** for cut/copy/paste
- **ctrl** + **s** to quickly save the document

For example, if you want to copy a line with the keyboard, you can go first to the beginning of the line (**home**), and then select the line by pressing **shift+down arrow**. Then press **ctrl+c** to copy it. Go then to where you want to copy it, and press **ctrl+v**.

4.4.2   Special tricks for Notepad++

- If you want to comment away lines, you can select them and then press **ctrl+q**. Do the same again to remove the comments.

- If you want to select columns of text, you can hold **alt** down, while selecting with shift and the arrow keys.

- **Ctrl+mouse wheel** zooms in/out in text.

4.5   Naming guidelines for VHDL design files

To identify VHDL source files, it is a good idea to have certain naming conventions and rules for what the various files should contain. In all of the designs you create starting with part 2 of the exercise, we will save the entity and architecture in different files and follow the naming rules given in the following table:

Table 1. Naming guidlines for VHDL source files

File content	File name
Entity and architecture	`<design_unit_name>.vhd`
Standalone entity	`<design_unit_name>_ent.vhd`
Standalone architecture[4]	`<design_unit_name>_arch.vhd`
Package	`<packagename>_pkg.vhd`
Test bench	`tb_<design_unit_name>.vhd`
Behavioral architecture (simulation model)	`<design_unit_name>_beh.vhd`

4.6   Type conversions for number formats

Type conversions are frequently used in VHDL. Working with arithmetic operations there will normally be times where you will want to go between std_logic_vector and some type of number format. At that point you should consider whether you are working with signed or unsigned numbers. For signed arithmetic, use `signed` and `integer`, and for unsigned arithmetic `unsigned` and `natural`. Unless you have a specific need for signed arithmetic, stick to unsigned to avoid complexity.

When dealing with arithmetic, we use `IEEE.numeric_std.all` in addition to `std_logic_1164`.

`numeric_std` defines `unsigned` which can be converted from and to `std_logic_vector` using the functions `unsigned(<std_logic_vector>)` and `std_logic_vector(<unsigned>)`. Another handy conversion at bit level can be `resize()` which can be used to truncate or pad with 0's to achieve the desired length of an `unsigned` used in a function. Browsing the package on internet may be useful.

Here is an example of a return statement in a function, where want to return an `unsigned` with the size of the `unsigned` "count", even though the sum may be one bit larger:

```
return resize(count + 1, count'length);
```

---

[4] In designs having multiple architectures for one entity, architecture names should be `<entityname_architecturename>_arch.vhd`

5 Getting Started with Block Design, AXI, and Xilinx SDK

(This section is no longer in use, as oblig 10 uses Xilinx tutorials. It is however kept, as a source of information, not as a part of the course syllabus).

In this section we will explain how to use the block design feature of Vivado, how to integrate the chip Processing System (PS) in your design, and how to communicate between the PS and the Programmable Logic (PL) using AXI interface.

5.1 Create Zynq Block Design and AXI interconnect

1.1.2 Create Vivado Project

In Vivado, start by creating a new project

Select File > New Project > Next

You can choose your project name and its location. You cannot have spaces in the names. You can use underscore ("_") as space.

When done, select

Next > RTL Project > Next

Change the Target Language depending on your needs.

We are not going to use any hdl files yet.

Select *Next > Next > Next*

Under "*select : Boards*" choose "ZedBoard Zynq Evaluation and Development Kit" as target board.
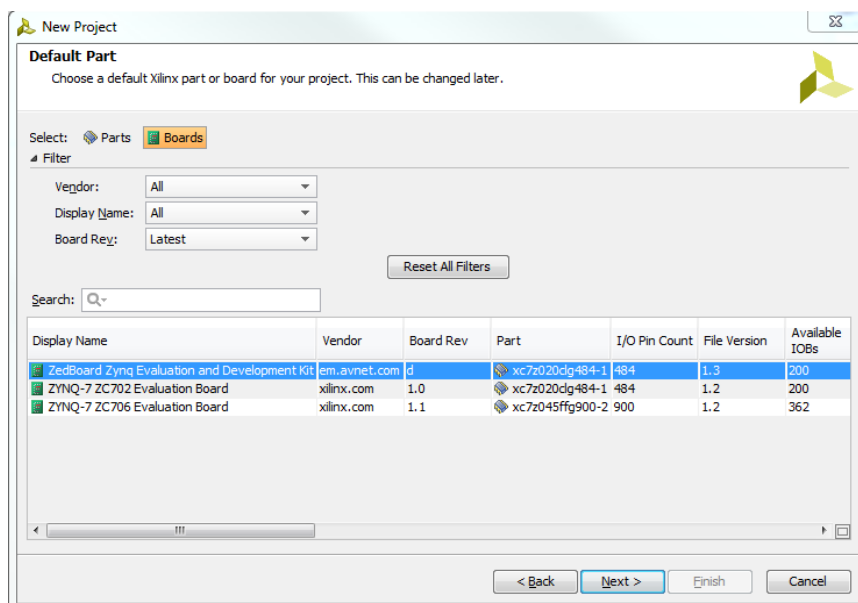
Then, complete the project creation by hitting *Next* and *Finish*.

5.1.1 Create Block Design

To create a block design, you need a dedicated page: under *IP Integrator* click on *Create Block Design* and *Ok*.

A new window will show up and you can add the blocks in the Diagram window.

Add IPs

To add blocks, you can right-click on the Diagram window and choose *Add IP* or use or ctrl + I. A drop down menu with available IPs will be displayed and you can choose among them:
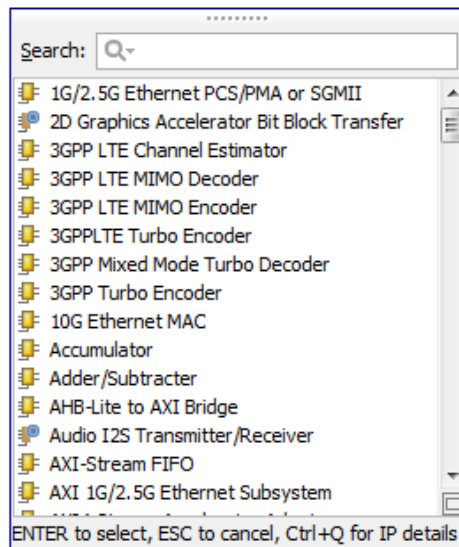

Search for **ZYNQ7 Processing System** and double click to add it. Do the same for the **Processor System Reset** and **AXI Interconnect**.

Configure IPs

Double-click on the AXI Interconnect IP and configure the *Number of Slave Interfaces* and *Number of Master Interfaces* to "1". Click *OK* to save the changes.

Connect the Zynq PS, the System Reset, and the AXI interconnect IPs in the following way:
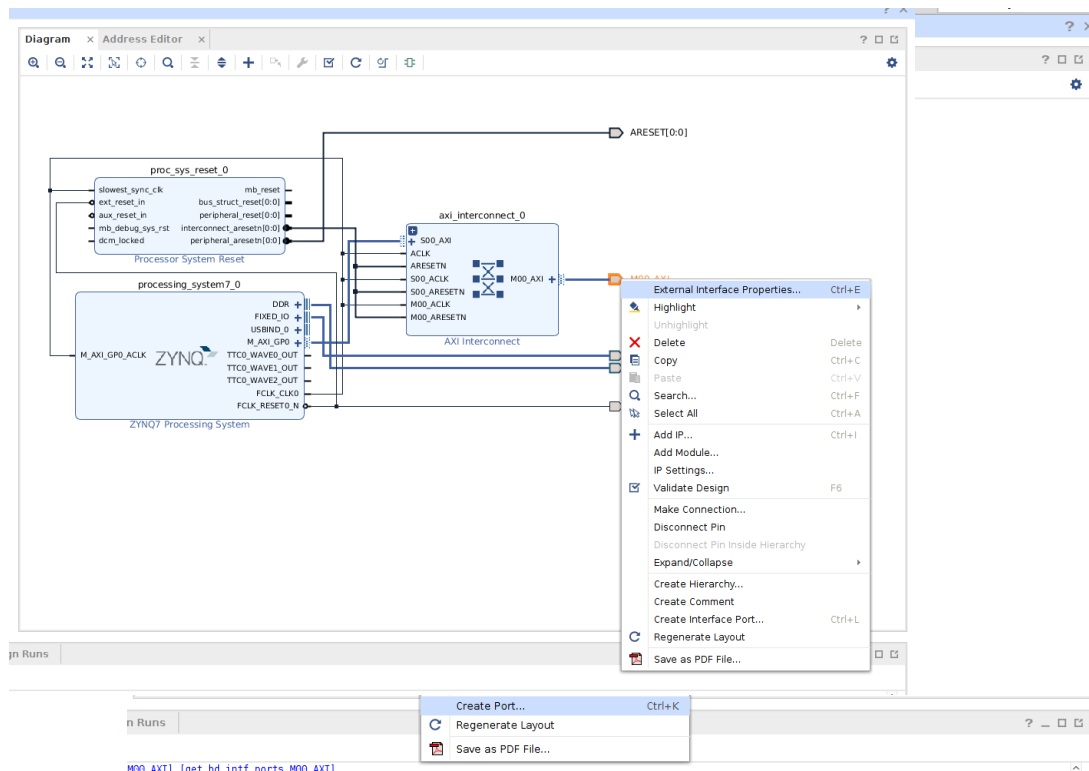
Click
Run
Block
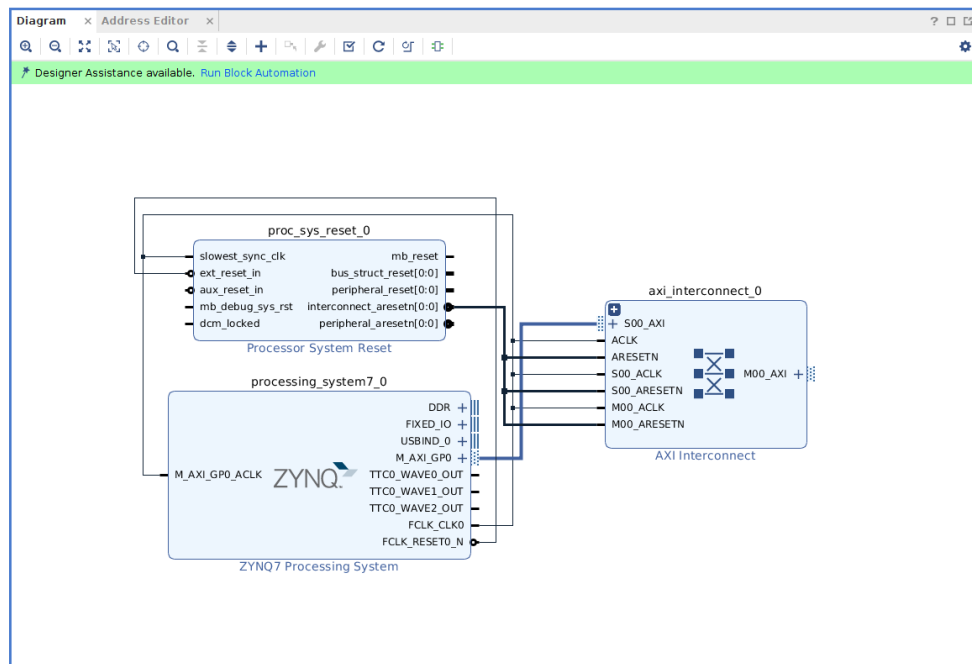
Automation (on the top of the panel) and click *OK*. This process will create FIXED_IO and DDR external outputs from the PS.
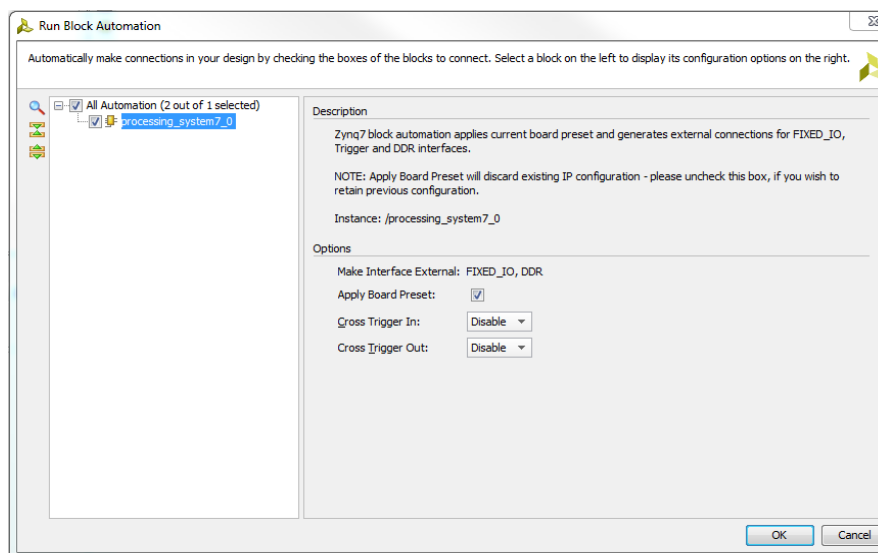
Right-click the **M_AXI** output port and select make it external by clicking on *Make External*.
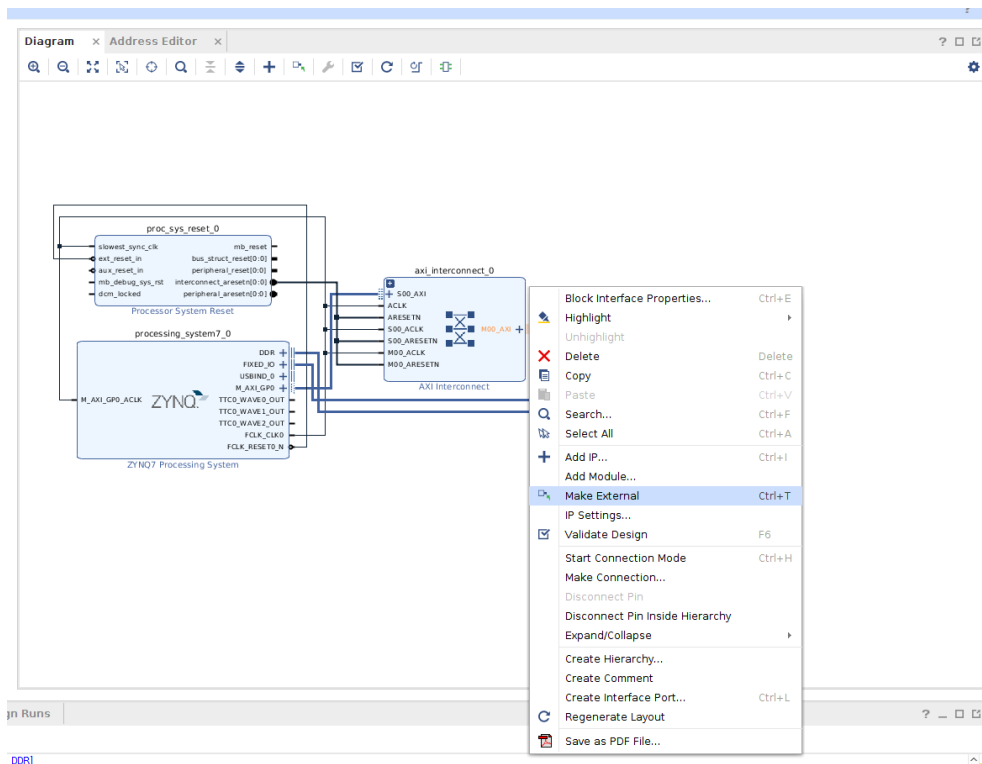
Right-click on peripheral_aresetn port of the system reset module and select *Create Port*.
Change the Port name to ARESET. Do the same with the FCLK_CLK0 of the Zynq PS and call it ACLK.
The purpose of this operation is to provide the AXI Interconnect with a clock from the Zynq PS since
the clock and reset are not part of a bundled AXI interface.

Right-click the M00_AXI port and select External Interface Properties.

Select the *General* tab on the left panel; choose "ACLK" as Clock Port. This is required to enable the IP Integrator to understand the clock frequency of the external interface. Otherwise, a *Critical Warning* similar to the following will occur:
[BD 41-968] AXI interface port /M00_AXI is not associated to any clock port. It may not work correctly. Please update ASSOCIATED_BUSIF parameter of a clock port to include this interface port.

Select the *Properties* tab, expand the *CONFIG* list, and set the *PROTOCOL* to "AXI4LITE." While this is set on the external interface port, the AXI Interconnect will query this setting and automatically use a protocol_converter block to convert the AXI3 interface of M_AXI_GP0.

Assign the address for M00_AXI in *Address Editor* tab. Right-click on M00_AXI and choose Assign Address. Then assign the following address: 0x**40000000**. This will be the address range for accesses that will be routed to the external AXI4LITE peripheral.

Output products generation

Navigate back to Vivado source window and right-click your block design (.bd) and choose *Generate Output Products* to save the project.
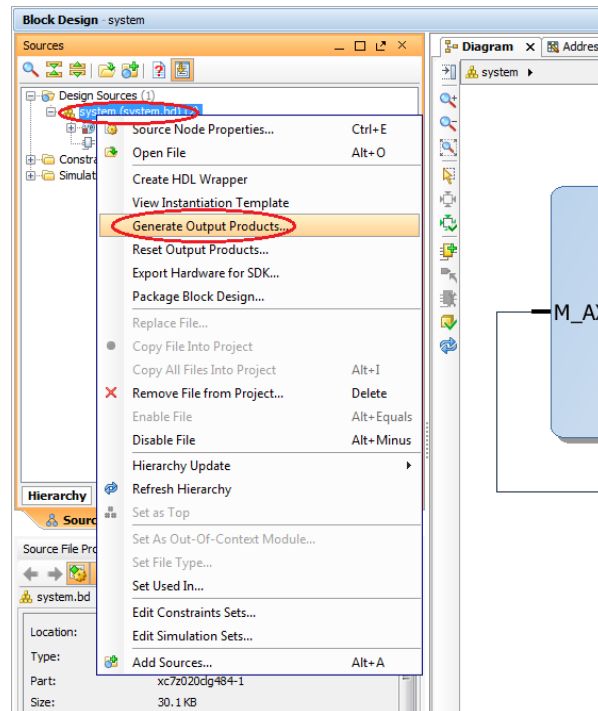
Note: There is a known issue in Vivado 2013.2. If you are using a later version, skip this note. The AXI4Lite ports will not be updated in the external wrapper unless the BD (block diagram) is closed and reopened. Close the block design then double-click it to reopen it. Right-click your block design and select Generate Output Products to regenerate the Output Products.
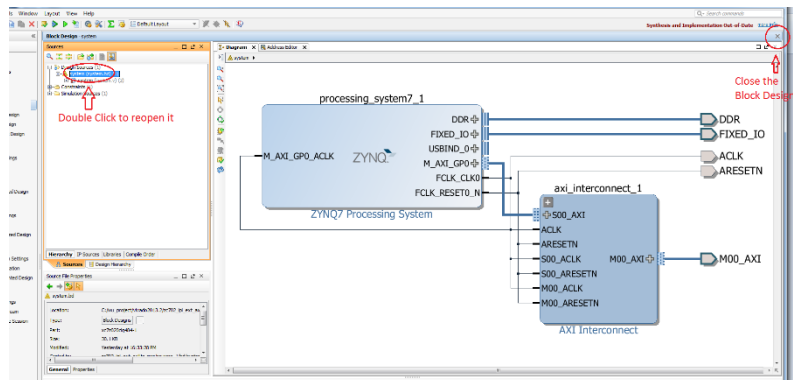

Now that the Block Design for the Zynq PS has been created, the next step is to create a HDL wrapper containing input and ouput ports so that we can interface it with other HDL sources.

Right-click your block design and select *Create HDL Wrapper*.

Open the resulting file and notice the AXI external master signals that were exposed.

The wrapper file can be imported as an HDL file into another project and the input/output ports can be connected to other components in the *top* file.
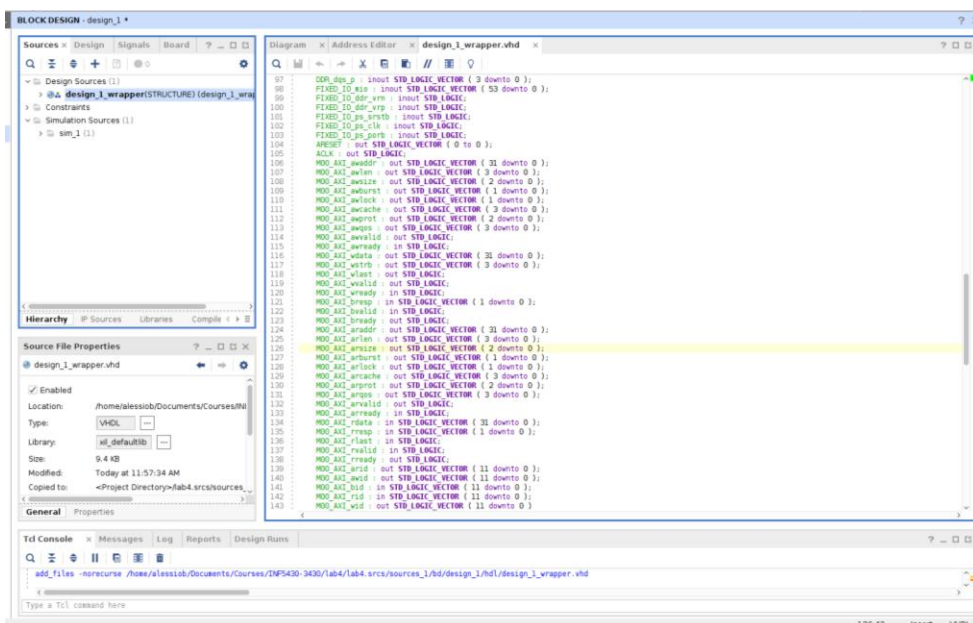
When you have finished your top and connected all modules, run the synthesis, the implementation and the generation of the bitstream.

Later, you will need the bitstream to program the FPGA in Xilinx SDK. In order to export the hardware for the SDK use:

Select File > Export > Export Hardware
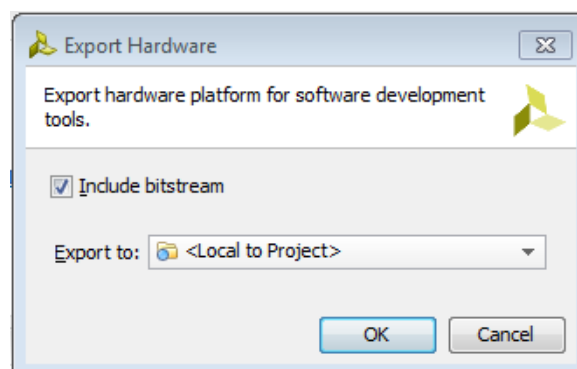
Be sure to check "*Include bitstream*".

The next step is to open the Xilinx SDK and write a c program for the PS.

Select File > Launch SDK > OK

5.2   Using the Xilinx SDK

5.2.1   Create a new project

The SDK should have opened after the conclusion of 5.1. If it did not open, you can open the SDK by navigating to *Start > All Programs > Xilinx Design Tools > SDK 2015.2 > Xilinx SDK 2015.2* and specifying the workspace.

To create a new project, select *File > New > Application Project* from the menu bar.
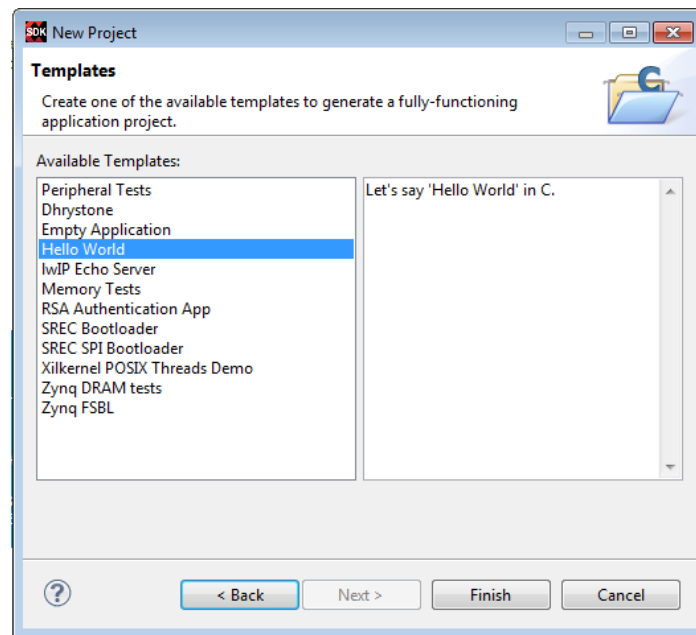
The *New Project* dialogue window will open. Enter the name of your project. You can keep all other options with the default settings and click next.

Select *Hello World* as template among the *Available Templates* and click finish.

Note: The new project should open automatically. If it does not, you may need to close the welcome tab in order to view the project.

In the *Project Explorer* panel, when you expand the *project_name* and *src* directory, you will find the "helloworld.c" file, which contains the main function of the program.

5.2.2 Board Support Package

There is a *bsp* directory created with our project: in this, you will find variables and functions that will help you creating your program.

You will find the parameters extracted from the exported hardware in the "xparameters.h". For example, here you will find the addresses mapped to the AXI interconnect M00. (*XPAR_M00_AXI_BASEADDR* to *XPAR_M00_AXI_HIGHADDR*).

```
/* Definitions for interface M00_AXI */
#define XPAR_M00_AXI_BASEADDR 0x43C00000
#define XPAR_M00_AXI_HIGHADDR 0x43C0FFFF

#include "xparameters_ps.h"

#define STDIN_BASEADDRESS 0xE0001000
#define STDOUT_BASEADDRESS 0xE0001000
```

If you want to write something at this address, you can use xil_out32 function you can find in the "xil_io.h".

```
#define Xil_Out32LE(Addr, Value) Xil_Out32((Addr), (Value))
```

For example:

```
Xil_Out32(XPAR_M00_AXI_BASEADDR, 0xffffff01);
```

will write 0xffffff01 at the starting address of the AXI.

There are also functions for 8-bit and 16-bit access using the same format:

```
Xil_Out16(XPAR_M00_AXI_BASEADDR, 0xff01);
```

```
Xil_Out8(XPAR_M00_AXI_BASEADDR, 0x01);
```

Sometimes it can be required to add delay between transactions. One way to create delay is to use "sleep".

At the top of the code add the following line:

```
#include <sleep.h>
```

Then use the following function to add delay:

```
sleep(2);
```

This will delay the program with 2 seconds. Any positive integer value can be used.

5.2.3   Run Program

Once your program is finished, you can build it just by saving it with
Project > Build (All or Project) or by pressing ctrl+B.

Verify that your board is powered up and connected to the computer with the JTAG port.

You will now download the bitstream to the Zynq PL by selecting

Xilinx Tools > Program FPGA from the menu bar or pressing .

The Program FPGA window will appear. The bitstream field should already be populated with the correct bitstream, if not you will need to find it.

With the Zynq PL successfully configured with the bitstream file, you can now launch your software application on the Zynq PS.

Select *project_name* in Project Explorer, right-click on it and select *Run As > Launch on Hardware (GDB)*.

The program will be uploaded to the Zynq PS and run on the board.

5.3   Creating managed IP project

It is possible to create a project for managing IPs in Vivado. The project location will then be the place where you put all of the configured and compiled IPs used in the design. One important thing to note is that the IP modules will be generated and synthesized Out Of Context, saving timing during synthesis of designs that uses a lot of IPs.

For Lab 4 you will create an IP project to manage the RAM and ILA module. An IP project is created in the following way:

1. Start Vivado
2. In Vivado select: **File->New IP Location**
3. Select the correct board, simulator and language.
4. Put the IP project in a folder named **IP** under the project root.
5. Press finished.
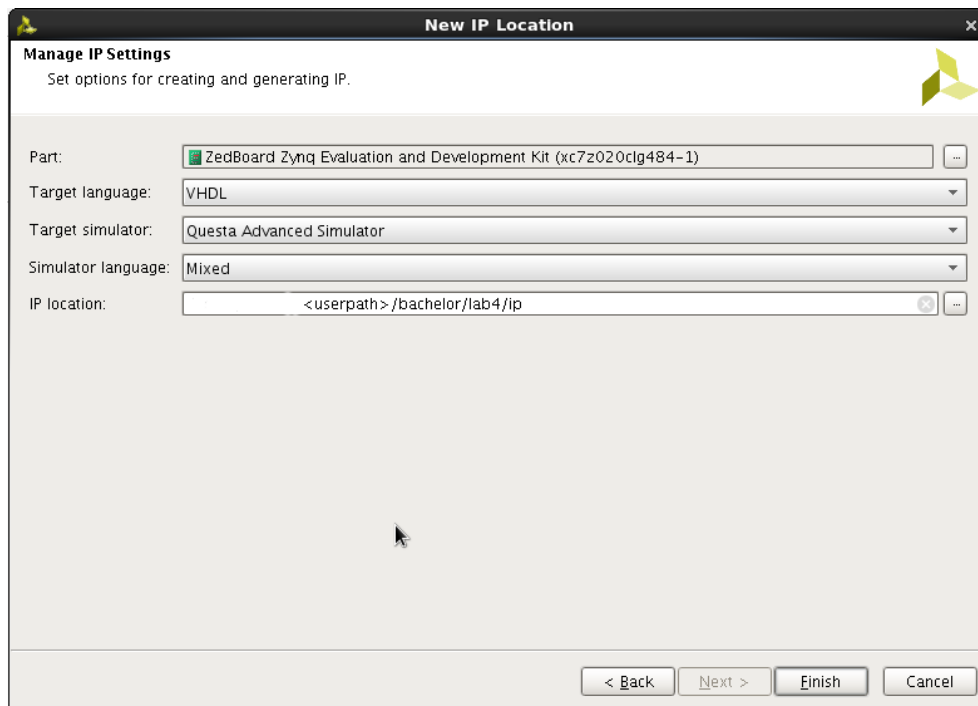
The RAM IP is created by doing the following steps:

1. Search in the IP Catalog for **Block Memory Generator**.
2. Select **Block Memory Generator** and press **Customize IP**
3. Customize the IP with name and settings, then press **OK**.
4. In the **Generate Output Products** window (which should appear) select **Out of context per IP** and press **Generate**.
5. It is also possible to trigger **Generate Output Products** by left clicking the IP in the **Sources** window.

You now have an IP that can be instantiated in your design. In the folder generated by **Generate Output Products** there should be a **.vho** file with a template for instantiation.

The ILA IP (or any other IP) can be created in much the same steps as the RAM.