Introduction to Xilinx Vivado and PYNQ Z2

Computer Architecture and Design – Practical 0

September 2019

1 Objectives

This practical is an introduction to logic design using Verilog HDL with Xilinx Vivado hardware design tools. Some new logic design concepts may be presented in this practical, depending on the order of the lectures for this course. The primary objective is that you become familiar with the tools and equipment that you will be using in the lab during the semester. There are two components, one software and one hardware:

- Xilinx's Vivado Design Suite software
- The PYNQ-Z2 FPGA board

Consider this practical a warm up for subsequent hardware design practicals, which become progressively more challenging. Please read this handout and work your way through all of the sections carefully, and take your time. If necessary, go back over the early sections a couple of times, and if you get stuck ask for help from teaching staff in the lab. As this is a tutorial introduction there is no summative assessment for this assignment. However, you are strongly advised to demonstrate your working design to an instructor at each stage of the tutorial. A sign-off sheet is provided for each stage of the tutorial, which you should ask an instructor to sign.

2 Bibliography

This practical draws on documents on the Xilinx website: http://www.xilinx.com, for details of the Vivado tools and the Zync 7020 FPGA chip that you will be using in the lab. The Zync 7020 chip is mounted on the PYNQ Z2 Board designed and manufactured by TUL. On the Xilinx website you can find Vivado tutorials, although this document should cover everything you need to know for the lab exercises you have been set. On the TUL website you can find the PYNQ Z2 Reference Manual here:

`https://d2m32eurp10079.cloudfront.net/Download/pynqz2_user_manual_v1_0.pdf`

We acknowledge the role of Xilinx and TUL in making this material available, and in particular we thank Xilinx for the donation of PYNQ boards for the Computer Architecture and Design lab.

3 Introduction to Xilinx Vivado 2018.3

Vivado is a tool that enables hardware designers to create a hardware design using a hardware description language (HDL) such as Verilog and map it to a Field Programmable Gate Array (FPGA). It also has facilities for simulating a design, in order to debug it before uploading it to an FPGA.

3.1 Setting up Your Environment

In order to run Vivado on a DICE machine you need to first set up a number of environment variables. These are packaged inside a shell script that you should source at the start of each lab session, or from within your default shell setup file. This can be done now by typing:

```
source /disk/scratch/Xilinx/Vivado/2018.3/settings64.sh
```

3.2 Creating a New Vivado Project

The design used initially in this practical is a simple two-input XOR. Two other designs will be introduced later in the practical. All designs will be described in Verilog.

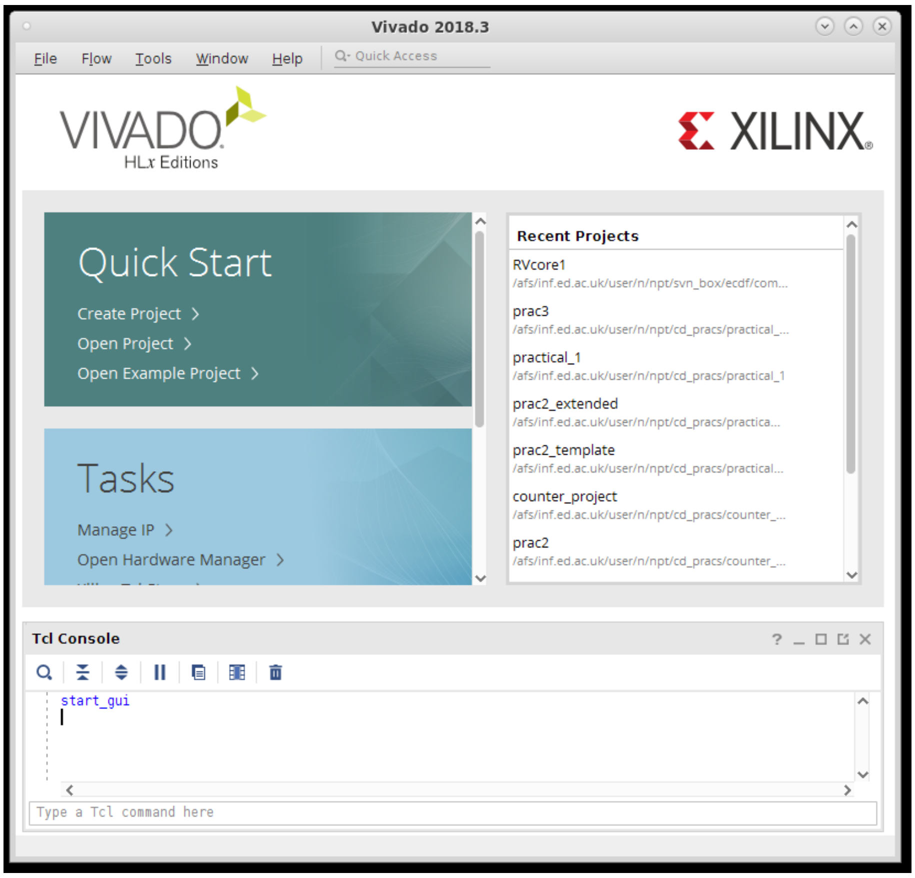
Firstly create a working directory for your project, *e.g.*, `$HOME/cd/prac0`, and `cd` to that directory.

Take a copy of the `pynq-z2_v1.0.xdc` file from the Learn page for this course, and rename it to `$HOME/cd/prac0/top.xdc`. You can find a link to this file on the page for **Practical 0**; the link is just after the link to this handout.

Then launch Vivado by simply typing:

```
vivado
```

Vivado provides a graphical user interface, so after a short delay you should see the opening screen appear. This is shown below:



Recent Projects

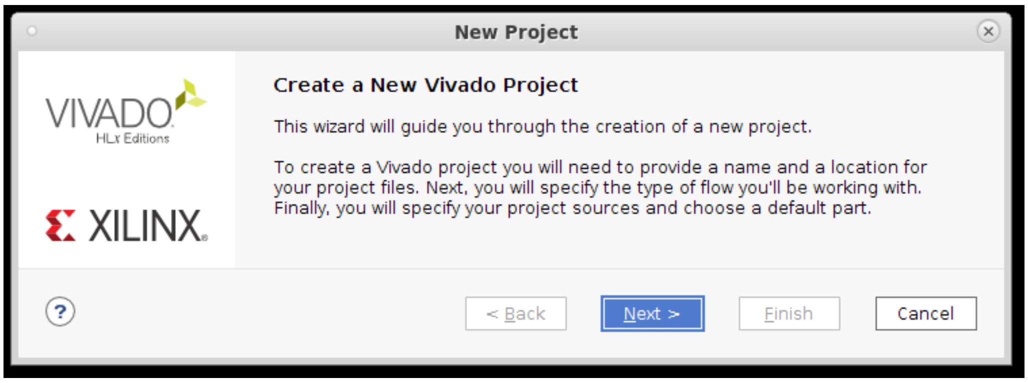
RVcore1
/afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/com...
prac3
/afs/inf.ed.ac.uk/user/n/npt/cd_pracs/practical_...
practical_1
/afs/inf.ed.ac.uk/user/n/npt/cd_pracs/practical_1
prac2_extended
/afs/inf.ed.ac.uk/user/n/npt/cd_pracs/practica...
prac2_template
/afs/inf.ed.ac.uk/user/n/npt/cd_pracs/practical...
counter_project
/afs/inf.ed.ac.uk/user/n/npt/cd_pracs/counter_...
prac2
/afs/inf.ed.ac.uk/user/n/npt/cd_pracs/counter_...

```
start_gui
```

Figure 1: Vivado opening screen

Here you can see some quick start options for creating a new project, opening an existing project, or opening an example project. If you have previously edited a project, it will appear in the right-hand list of Recent Projects. You can see one of mine in this example. To open a recent project just click on the project name.

The first time you run Vivado click on Create New Project, this opens up the new project wizard, as shown in Figure 2, and click Next to continue.



New Project

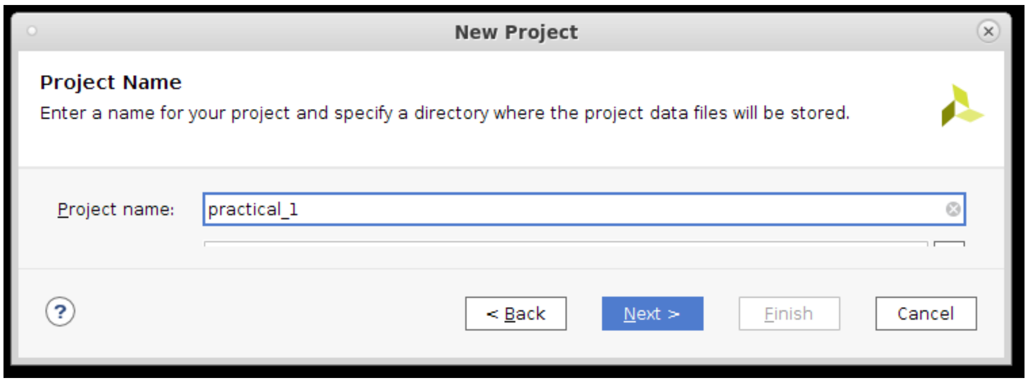
Create a New Vivado Project

This wizard will guide you through the creation of a new project.

To create a Vivado project you will need to provide a name and a location for your project files. Next, you will specify the type of flow you'll be working with. Finally, you will specify your project sources and choose a default part.

Figure 2: New project dialogue

Choose a suitable project name and a location to store the project, and enter them as shown in Figure 3. Click Next.



New Project

Project Name

Enter a name for your project and specify a directory where the project data files will be stored.

Project name: practical_1

Figure 3: Setting the new project name

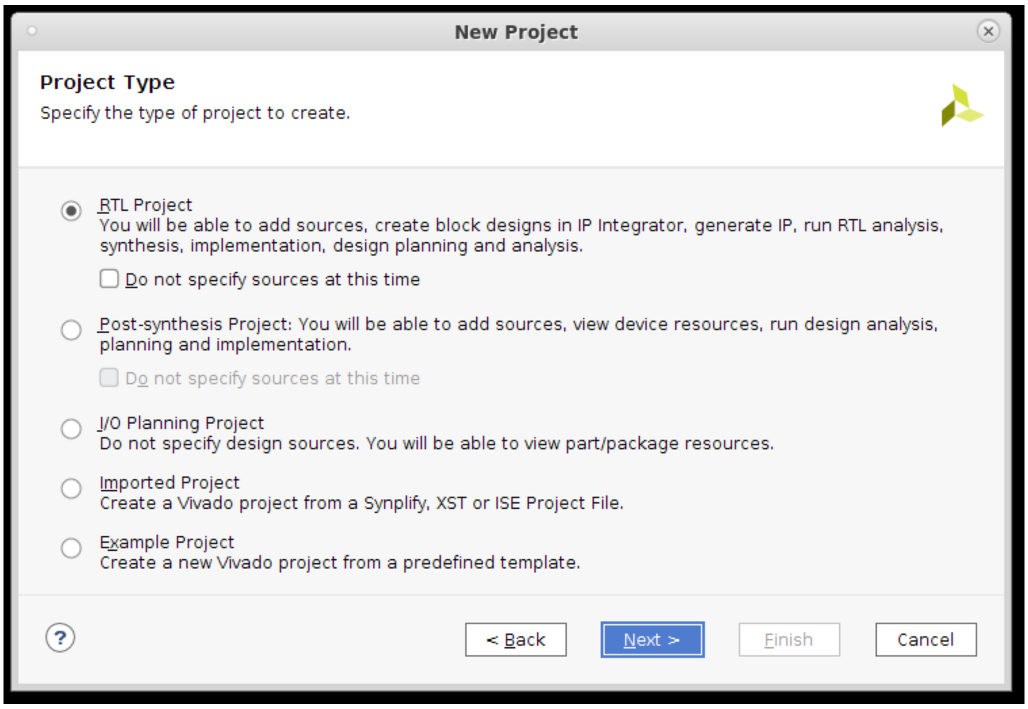
You will then be presented with the five options for the type of project to be created, as shown in Figure 4. Select an RTL project and click Next to continue.

The next step is to add a source file to the project, using the Add Sources screen of the New Project wizard shown in Figure 5. Press the **Create File** button and a dialogue entitled Create Source File will appear, as shown in Figure 6.

Set the file type to Verilog, set the file name to `top.v`, and leave the File Location as `<Local to Project>`, as shown in Figure 7.

When the first Verilog source file has been added to the project, the display should look like Figure 8.

Every Verilog design that you intend to run on an FPGA must have a **constraint file**. At this stage you should add a suitable constraint file to the project. Click on the **Next** button of the New Project dialog to move to the Add Constraints dialog. Click on the **Add Files** button, and set the file name to `top.xdc`. Make sure that you have copied this file into your project directory, as explained earlier. You will be presented with an Add Constraint File dialogue as shown in Figure 9.


New Project

Project Type

Specify the type of project to create.

[● RTL Project] You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

☐ Do not specify sources at this time

○ Post-synthesis Project: You will be able to add sources, view device resources, run design analysis, planning and implementation.

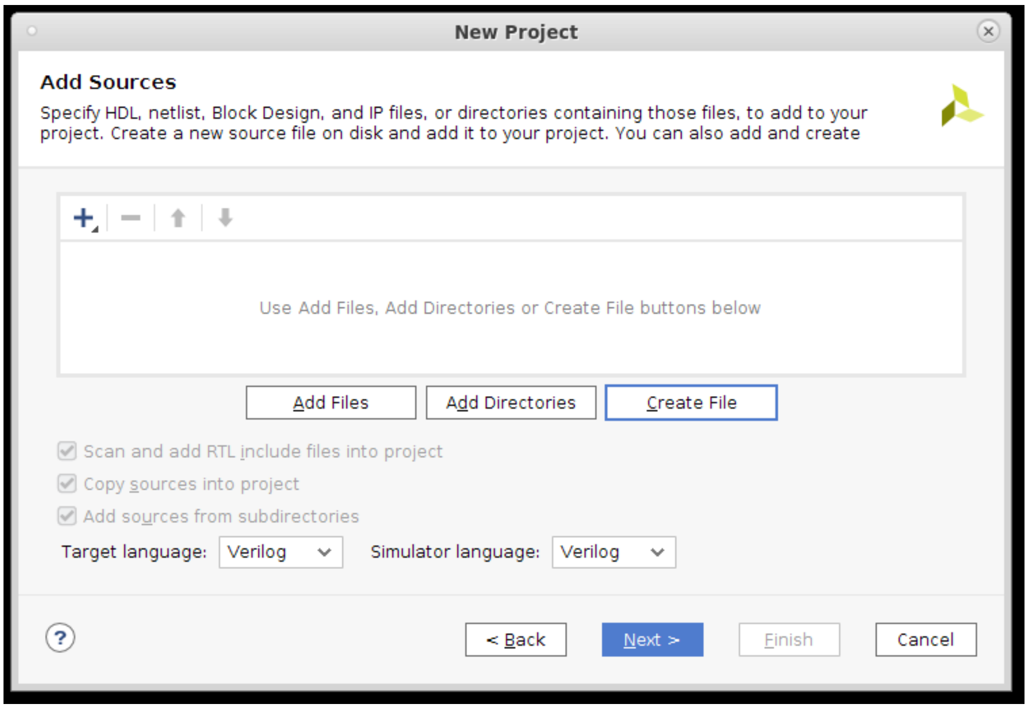
☐ Do not specify sources at this time

○ I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

○ Imported Project
Create a Vivado project from a Synplify, XST or ISE Project File.

○ Example Project
Create a new Vivado project from a predefined template.

Figure 4: Setting the new project type


New Project

Add Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those files, to add to your project. Create a new source file on disk and add it to your project. You can also add and create

Use Add Files, Add Directories or Create File buttons below

☑ Scan and add RTL include files into project

☑ Copy sources into project

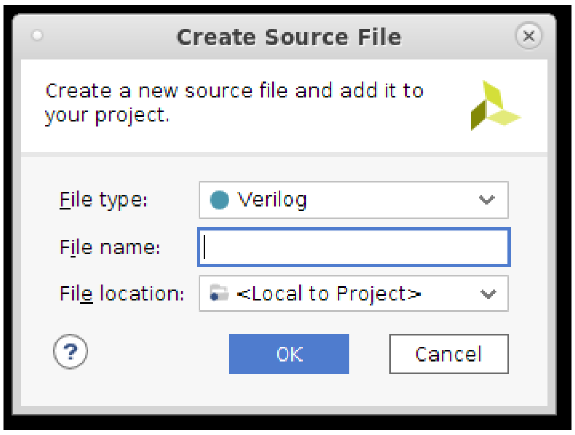
☑ Add sources from subdirectories

Target language: Verilog   Simulator language: Verilog

Figure 5: Adding a source file to the project

Navigate to your copy of top.xdc and select it. You will then see a preview of its contents. Click OK, and then click Next when you return to the New Project dialog.

You now have to select the type of hardware that you will be using. You will be presented with a **Default Part** dialog. Click on **Boards** and you will be presented with a selection of FPGA boards, as shown in Figure 10. Choose the **pynq-z2** option and click Next.


Create Source File

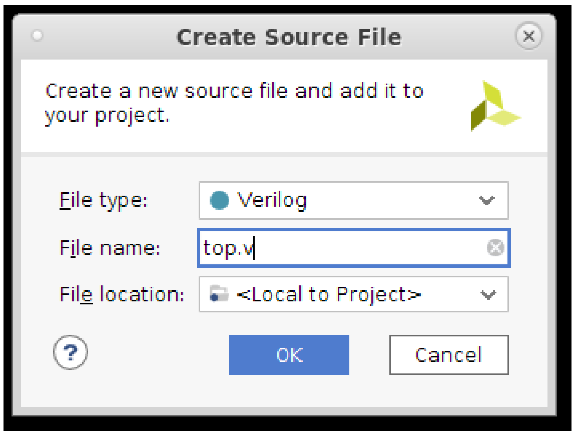
Create a new source file and add it to your project.

File type: Verilog

File name:

File location: <Local to Project>

Figure 6: Creating a source file


Create Source File

Create a new source file and add it to your project.

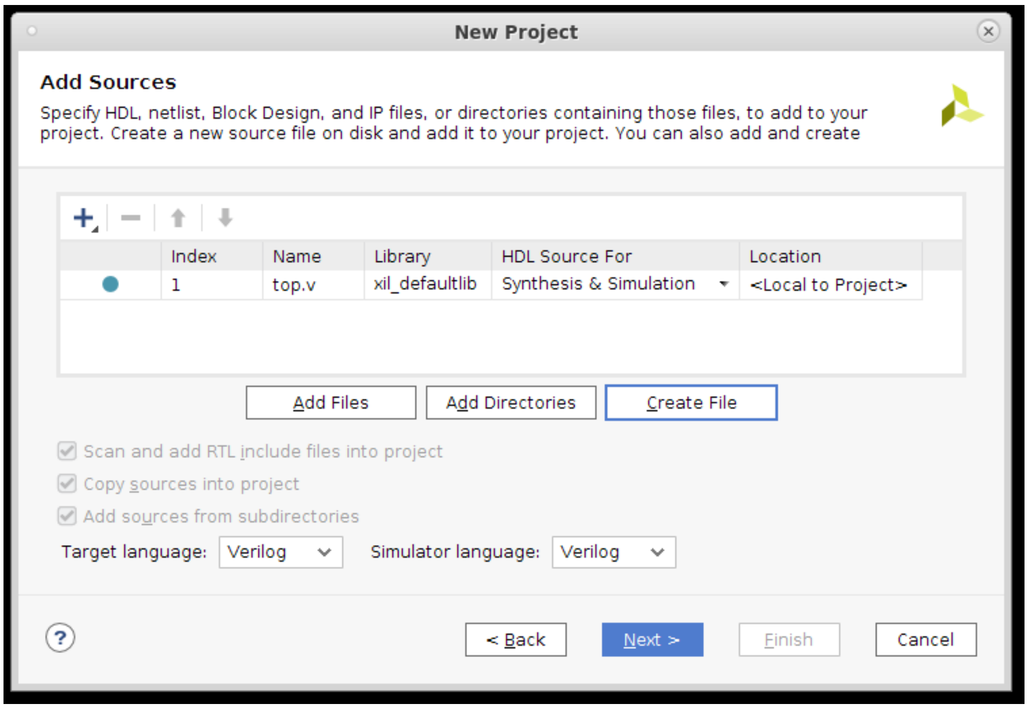
File type: Verilog

File name: top.v

File location: <Local to Project>

Figure 7: Source file after creation

You will be presented with a New Project Summary, which should look like Figure 11.


New Project

Add Sources

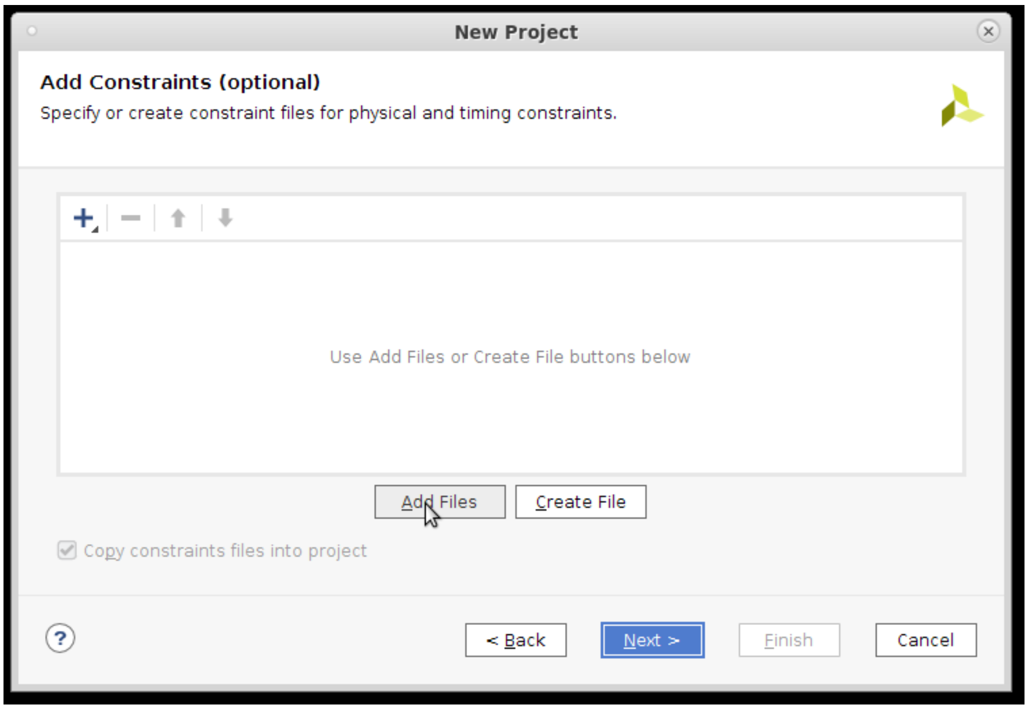
Specify HDL, netlist, Block Design, and IP files, or directories containing those files, to add to your project. Create a new source file on disk and add it to your project. You can also add and create

	Index	Name	Library	HDL Source For	Location
●	1	top.v	xil_defaultlib	Synthesis & Simulation	<Local to Project>

- [✓] Scan and add RTL include files into project
- [✓] Copy sources into project
- [✓] Add sources from subdirectories

Target language: Verilog   Simulator language: Verilog

Figure 8: Screenshot of Vivado display when top.v has been added to the project


New Project

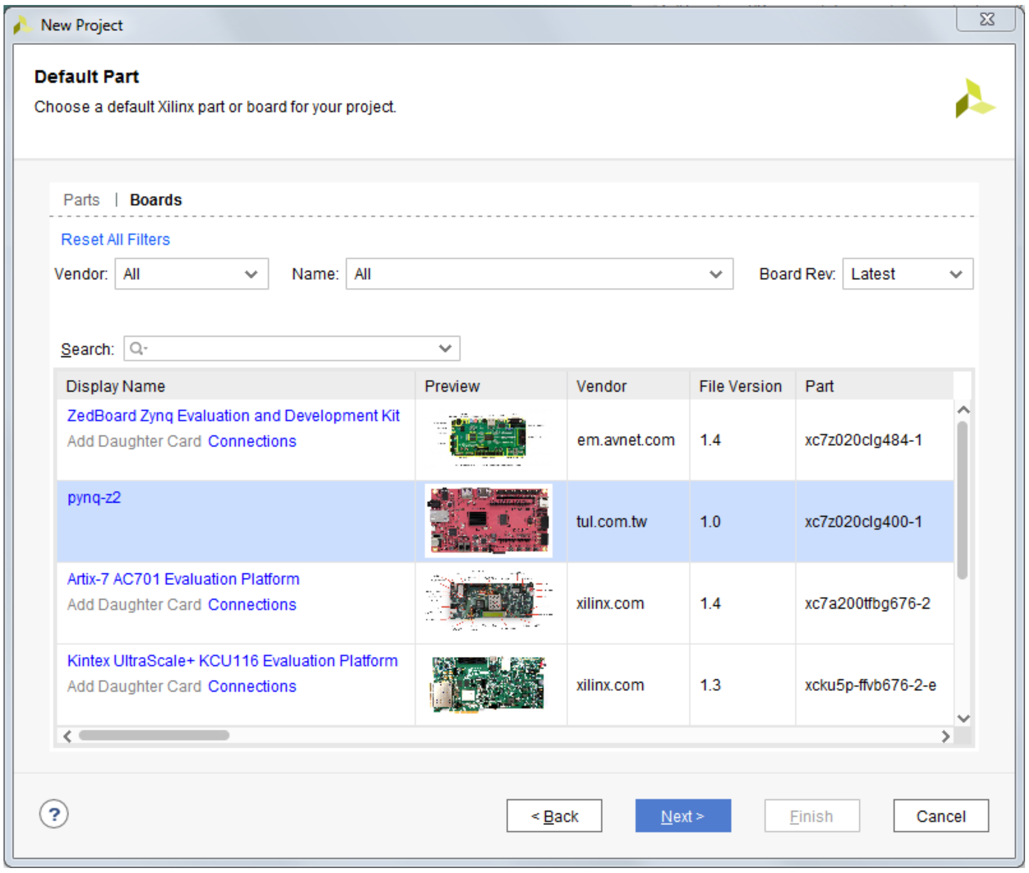
Add Constraints (optional)

Specify or create constraint files for physical and timing constraints.

Use Add Files or Create File buttons below

- [✓] Copy constraints files into project

Figure 9: Screenshot of dialog for adding a constraint file



New Project

Default Part

Choose a default Xilinx part or board for your project.

Parts | **Boards**

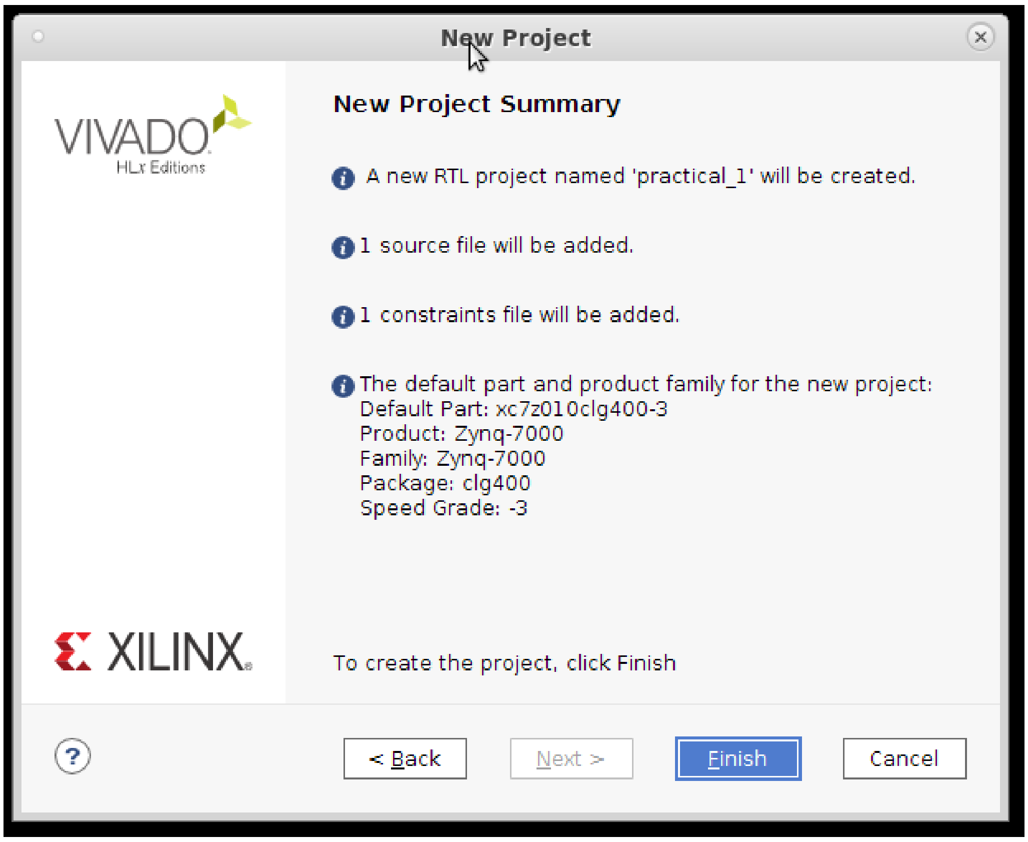
Reset All Filters

Vendor: All   Name: All   Board Rev: Latest

Search:

Display Name	Preview	Vendor	File Version	Part
ZedBoard Zynq Evaluation and Development Kit Add Daughter Card Connections		em.avnet.com	1.4	xc7z020clg484-1
pynq-z2		tul.com.tw	1.0	xc7z020clg400-1
Artix-7 AC701 Evaluation Platform Add Daughter Card Connections		xilinx.com	1.4	xc7a200tfbg676-2
Kintex UltraScale+ KCU116 Evaluation Platform Add Daughter Card Connections		xilinx.com	1.3	xcku5p-ffvb676-2-e

Figure 10: Screenshot showing selection of FPGA parts and boards



New Project Summary

A new RTL project named 'practical_1' will be created.

1 source file will be added.

1 constraints file will be added.

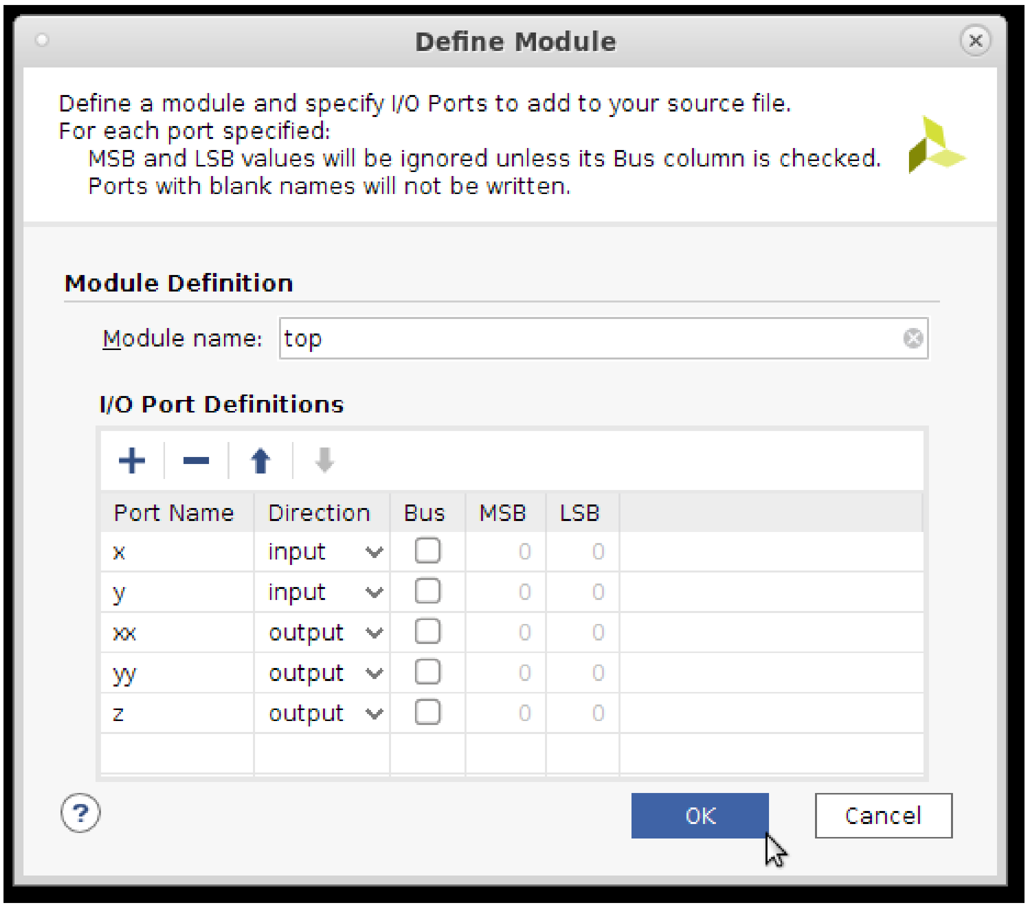
The default part and product family for the new project:
Default Part: xc7z010clg400-3
Product: Zynq-7000
Family: Zynq-7000
Package: clg400
Speed Grade: -3

To create the project, click Finish

Figure 11: Screenshot showing New Project Summary

When you are happy with the summary, click Finish, and the new project wizard will complete. The project manager window will open, and you will then be presented with another dialog through which you can define the Verilog module that forms the top-level design within the `top.v` source file that you have added to the project.

First you must define the input/output ports using the dialog shown below. Click in the Port Name field of the first empty row of the I/O Port Definition table and type the port name `x`, leaving it defined as an input. If you hit return, you can fill in entries for other I/O ports, creating as many as you need. For this practical you should create inputs `x` and `x`, and outputs `xx`, `yy`, and `z`, as shown in Figure 12.



Define Module

Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

Module Definition

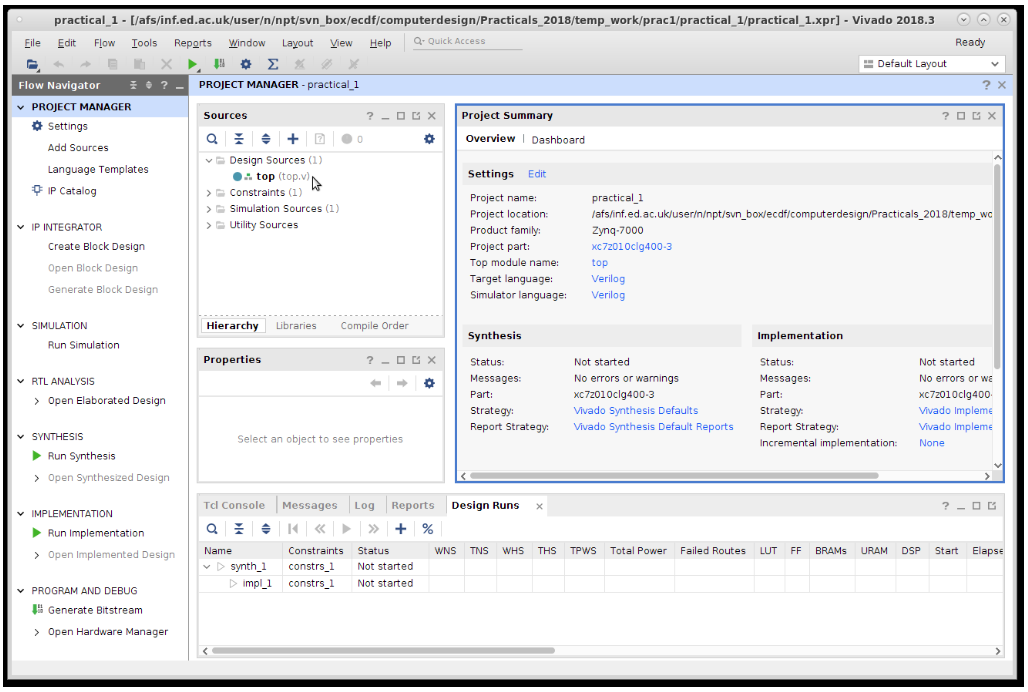
Module name: top

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
x	input	☐	0	0
y	input	☐	0	0
xx	output	☐	0	0
yy	output	☐	0	0
z	output	☐	0	0

Figure 12: Screenshot showing the definition of input and output ports

You will then see the top-level view for the newly created project, as shown in Figure 13. On the left-hand side you will see the Flow Navigator panel. At the top is the Project Manager, where you can setup the project. Below that you will see six phases of the design flow, starting with *IP Integrator* and ending with *Program and Debug*. In general, we work through these phases from top to bottom, although in this lab we may skip some steps that are not needed. For each step there is normally an option to change the control settings for that step, a button to initiate that step, and sometimes a button to open up or view the results of running that step. The final step, *Open Hardware Manager*, allows you to upload your design to the FPGA and run it on the PYNQ Z2 board.



Settings Edit

Project name:	practical_1
Project location:	/afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/computerdesign/Practicals_2018/temp_wo
Product family:	Zynq-7000
Project part:	xc7z010clg400-3
Top module name:	top
Target language:	Verilog
Simulator language:	Verilog

Synthesis

Status:	Not started
Messages:	No errors or warnings
Part:	xc7z010clg400-3
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports

Implementation

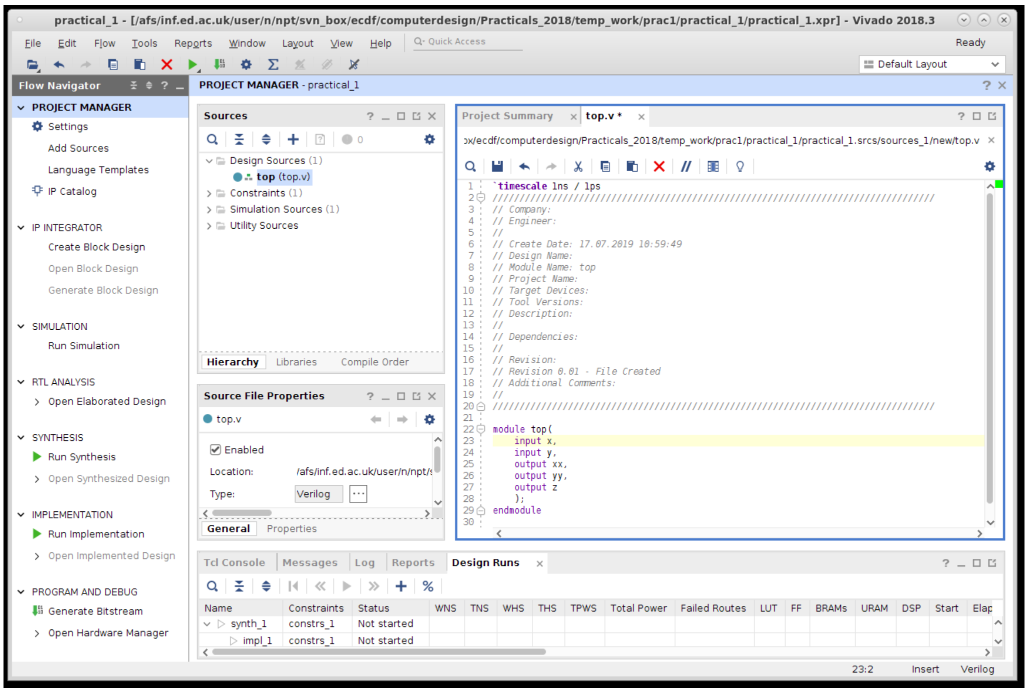
Status:	Not started
Messages:	No errors or wa
Part:	xc7z010clg400-
Strategy:	Vivado Impleme
Report Strategy:	Vivado Impleme
Incremental implementation:	None

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	Start	Elapse
synth_1	constrs_1	Not started														
impl_1	constrs_1	Not started														

Figure 13: Screenshot showing the newly-created project

4 Writing the Verilog Code

To edit the Verilog source code, double-click on the icon for the top.v file in the Sources window of the Project Manager. This opens an editor showing the template for top.v that was created using the I/O port definitions you entered in the Create Project wizard. You will be presented with an editor in the right-hand pane, as shown in Figure 14.



practical_1 - [/afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/computerdesign/Practicals_2018/temp_work/prac1/practical_1/practical_1.xpr] - Vivado 2018.3
File Edit Flow Tools Reports Window Layout View Help
Quick Access
Ready
Default Layout
Flow Navigator
PROJECT MANAGER - practical_1
Sources
Design Sources (1)
top (top.v)
Constraints (1)
Simulation Sources (1)
Utility Sources
Hierarchy
Libraries
Compile Order
Source File Properties
top.v
Enabled
Location: /afs/inf.ed.ac.uk/user/n/npt/
Type: Verilog
General
Properties
Project Summary
top.v *
ɔx/ecdf/computerdesign/Practicals_2018/temp_work/prac1/practical_1/practical_1.srcs/sources_1/new/top.v
Tcl Console
Messages
Log
Reports
Design Runs
23:2
Insert
Verilog

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 17.07.2019 10:59:49
// Design Name:
// Module Name: top
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module top(
    input x,
    input y,
    output xx,
    output yy,
    output z
    );
endmodule

```

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	Start	Elap
synth_1	constrs_1	Not started														
impl_1	constrs_1	Not started														

Figure 14: Screenshot showing the editing of a Verilog source file

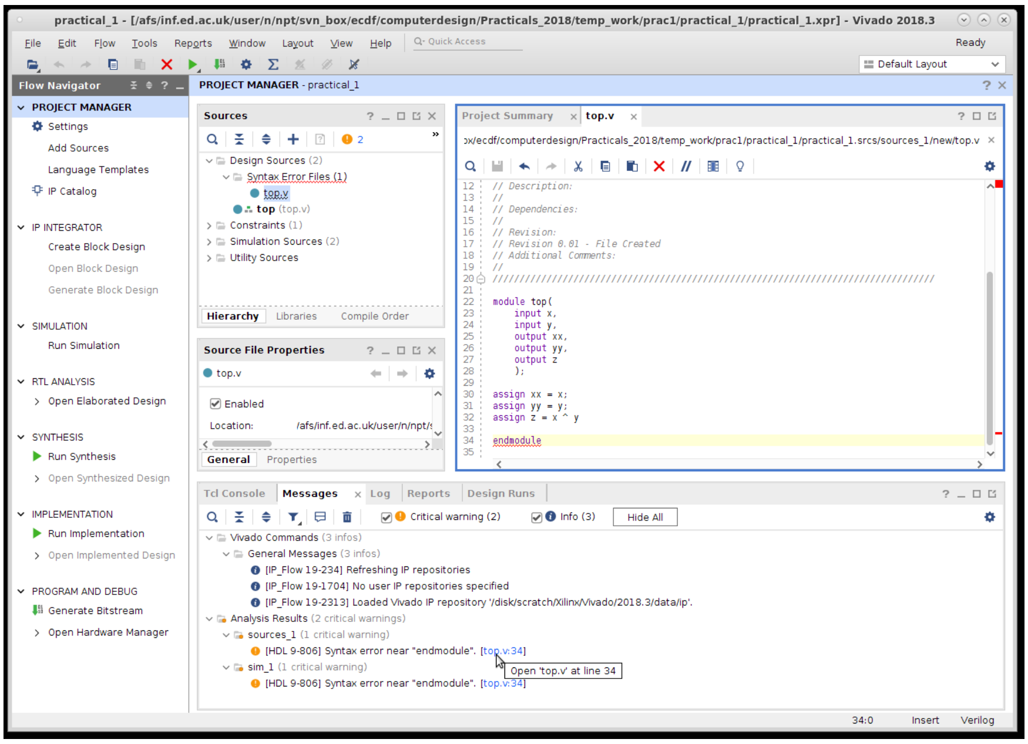
In this very simple example, our aim is to connect the two inputs to two push-buttons on the PYNQ Z2 board, and to connect the three outputs to three LEDs on the PYNQ Z2 board. Two of the outputs will simply display the two inputs, and the third will display the XOR of the two inputs. This requires three `assign` statements:

```
assign xx = x;
assign yy = y;
assign z  = x ^ y;
```

Add this code to top.v and save your changes.

5 Checking the Syntax of your code

When you save a Verilog source file, the Vivado tool automatically checks for syntax errors. For example, if I accidentally omit a semi-colon at the end of a statement, the Sources window creates a folder containing any source files that have critical syntax errors. Click on the small orange circle in the Sources header row, or select the Message tab in the lower half of the display, and the errors are displayed in the Messages tab in the lower pane of the display, as shown in Figure 15



```
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module top(
    input x,
    input y,
    output xx,
    output yy,
    output z
    );

assign xx = x;
assign yy = y;
assign z = x ^ y

endmodule
```

```
Vivado Commands (3 infos)
  General Messages (3 infos)
    [IP_Flow 19-234] Refreshing IP repositories
    [IP_Flow 19-1704] No user IP repositories specified
    [IP_Flow 19-2313] Loaded Vivado IP repository '/disk/scratch/Xilinx/Vivado/2018.3/data/ip'.
Analysis Results (2 critical warnings)
  sources_1 (1 critical warning)
    [HDL 9-806] Syntax error near "endmodule". [top.v:34]
  sim_1 (1 critical warning)
    [HDL 9-806] Syntax error near "endmodule". [top.v:34]
```

Figure 15: Screenshot showing compilation error messages

You can click on the link to the file and line number of each error and the file display will go to that file and that line. Correct any errors, and then re-save your top.v source file. The error messages will go away (hopefully!), and the folder of erroneous files will disappear from the Sources window. At this point your code is syntactically correct.

6 Connecting Ports to Pins

So far, the project has no way to associate the I/O ports of your top module with the push buttons and LEDs on the PYNQ board. This is achieved by setting *constraints* in the Xilinx Design

Constraints file `top.xdc` that you added to the project earlier.

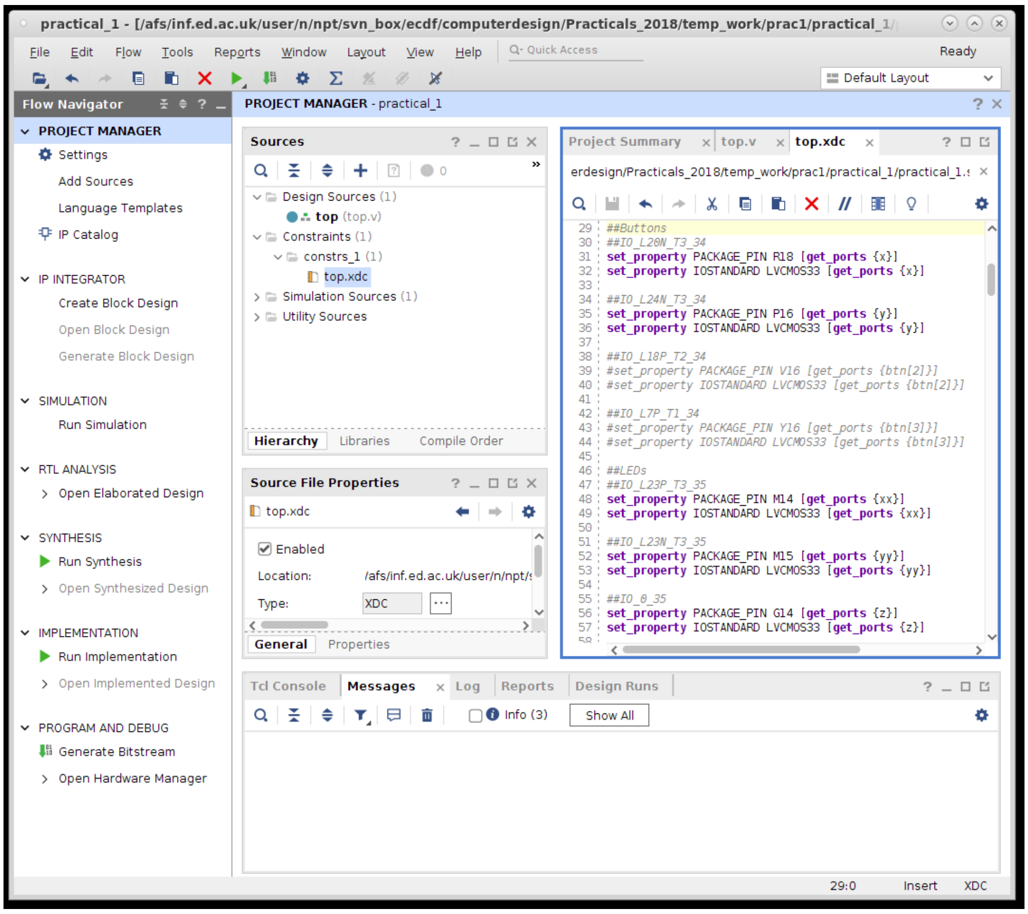
In the Sources window, open the constraints hierarchy to show Constraints, constrs_1 and `top.xdc`. Double-click on `top.xdc` to open an editor window so that you can make the necessary changes to the port assignments.

The `top.xdc` file contains two lines for each pin of the Zynq-7020 chip package, one to define the connection between the pin and the top-level ports of your design, and the other to define the I/O standard for that signal. The I/O standards should not be altered, and should be left as LVCMOS33 (low-voltage CMOS, at 3.3V).

Each package pin is identified by a unique coordinate comprising a letter and a number. There are four push-buttons on the PYNQ board, with coordinates D19, D20, L20 and L19, and their entries appear in lines 27–30 of the `top.xdc` file.

To connect the push buttons to your two inputs `x` and `y`, uncomment the two `set_property` lines for D19 and D20, and replace `btn[0]` with `x`, and likewise replace `btn[1]` with `y`.

To connect your three output ports to three of the available LEDs, make similar changes to the ports on which you find `led[0]`, `led[1]` and `led[2]`. These should be connected to `xx`, `yy` and `z`. When this is done save your changes to `top.xdc`, which should look like Figure 16.

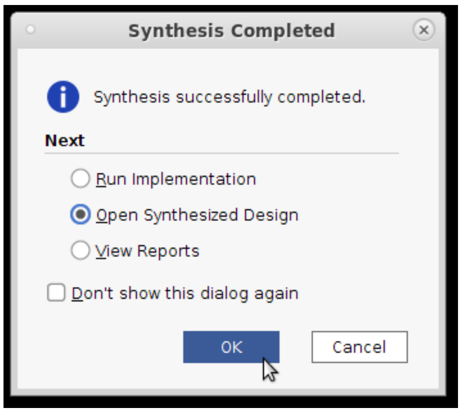


```
##Buttons
##IO_L20N_T3_34
set_property PACKAGE_PIN R18 [get_ports {x}]
set_property IOSTANDARD LVCMOS33 [get_ports {x}]

##IO_L24N_T3_34
set_property PACKAGE_PIN P16 [get_ports {y}]
set_property IOSTANDARD LVCMOS33 [get_ports {y}]

##IO_L18P_T2_34
#set_property PACKAGE_PIN V16 [get_ports {btn[2]}]
#set_property IOSTANDARD LVCMOS33 [get_ports {btn[2]}]

##IO_L7P_T1_34
#set_property PACKAGE_PIN Y16 [get_ports {btn[3]}]
#set_property IOSTANDARD LVCMOS33 [get_ports {btn[3]}]

##LEDs
##IO_L23P_T3_35
set_property PACKAGE_PIN M14 [get_ports {xx}]
set_property IOSTANDARD LVCMOS33 [get_ports {xx}]

##IO_L23N_T3_35
set_property PACKAGE_PIN M15 [get_ports {yy}]
set_property IOSTANDARD LVCMOS33 [get_ports {yy}]

##IO_0_35
set_property PACKAGE_PIN G14 [get_ports {z}]
set_property IOSTANDARD LVCMOS33 [get_ports {z}]
```

Figure 16: Screenshot showing .xdc constraint file contents

7 Synthesis Phase

In the Synthesis section of the Flow Navigator click on the *Run Synthesis* button, and hit OK to accept the default synthesis options when prompted. This will initiate the synthesis phase, which for this simple design should take less than a minute. If it completes successfully it will pop up a dialog box as shown in Figure 17.


Synthesis Completed

Synthesis successfully completed.

Next

( ) Run Implementation

(●) Open Synthesized Design

( ) View Reports

☐ Don't show this dialog again

Figure 17: Screenshot showing completed synthesis

At this point you can choose any of the three suggested options for the next stage in the flow. Try opening the synthesized design. You will see a graphical representation of the internal resources within the FPGA and how they are used by your design. Click on the **top** icon in the Netlist pane and the Netlist Properties pane will be populated. This shows how many LUTs (Look Up Tables) and I/O ports are used by your design. Click on the second icon from the left, just below the Sources tab label, to display a gate-level schematic of your synthesized design. It should look like Figure 18.

Notice the rectangular component labeled LUT2, with two inputs and one output. A LUT is a Look Up Table, and it is these components that implement the combinational logic functions that are produced when your Verilog code is synthesized. Unsurprisingly, this LUT implements a single 2-input XOR gate. Close the synthesized design pane to return to the previous view of the project manager.

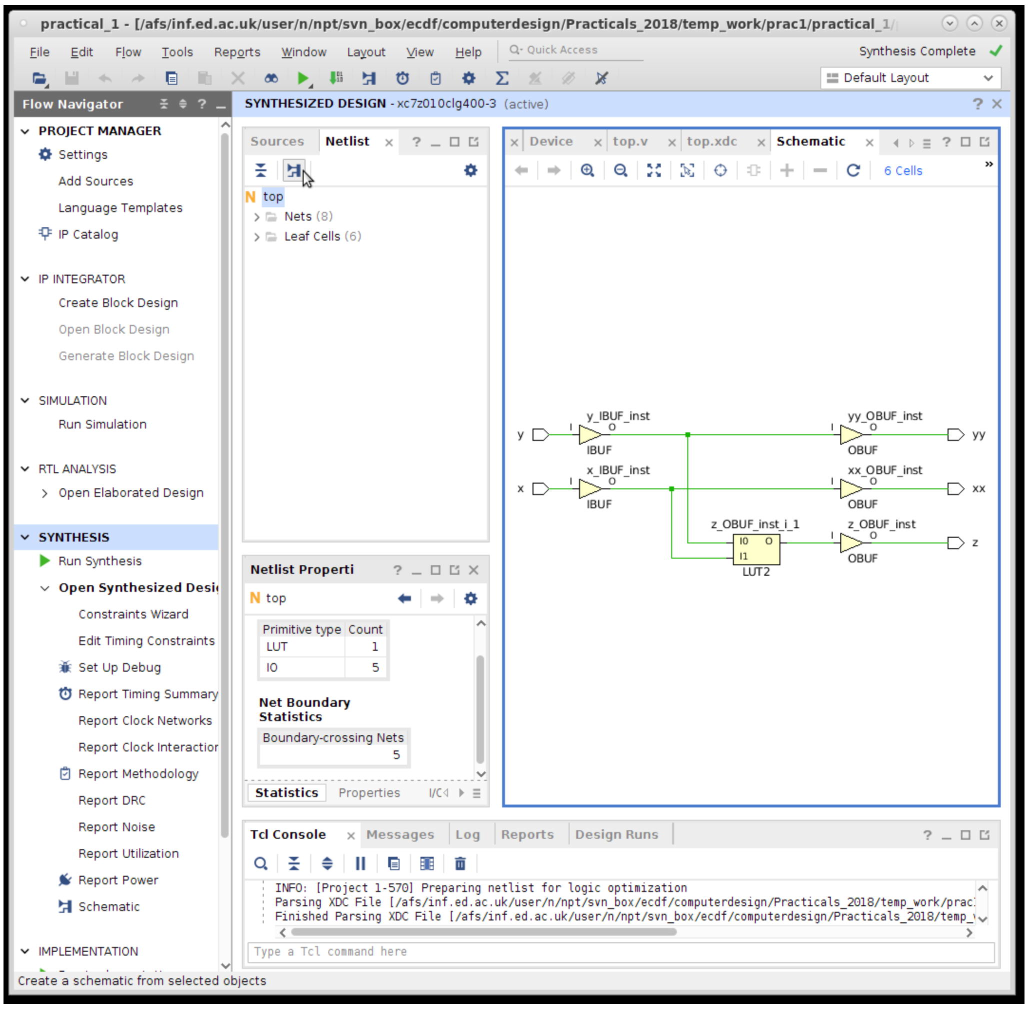
8 Implementation Phase

When synthesis has been completed successfully, you can move to the implementation phase, where the design is prepared and optimized for running on the FPGA. Click the *Run Implementation* button under the Implementation phase, and hit OK to accept the default implementation options. This normally takes longer than synthesis, but for this simple design should take no more than a couple of minutes. When it finishes successfully, you should see a further dialog box suggesting three options, as shown in Figure 19.

Select the option to *Generate Bitstream* and click OK, then ckick OK again when presented with the default options for generating a bitstream. This produces an output file containing the configuration information for the FPGA, and which is used to program the FPGA with your design. When the bitstream has been generated Vivado will open another dialog box with three suggested options, as shown in Figure 20.

Select the option to *Open Hardware Manager* and click OK. You will see that the Hardware Manager opens in the right-hand pane, and indicates that it is unconnected. At this point you must connect a PYNQ Z2 board to your computer using one of the USB cables provided in the lab. The USB cable connects to the PYNQ Z2 board on the left-hand side, near to the power switch. Turn on the PYNQ Z2 board and you should see a red LED illuminate.

Once your PYNQ Z2 board is connected and powered up, click on the link to *Open Target*, and select the option to *Auto-connect*. If the connection to the FPGA succeeds you will see an


practical_1 - [/afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/computerdesign/Practicals_2018/temp_work/prac1/practical_1/

SYNTHESIZED DESIGN - xc7z010clg400-3 (active)

Primitive type	Count
LUT	1
IO	5

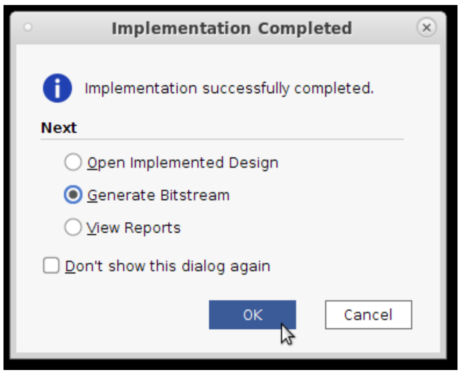
Net Boundary Statistics

Boundary-crossing Nets
5

```
INFO: [Project 1-570] Preparing netlist for logic optimization
Parsing XDC File [/afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/computerdesign/Practicals_2018/temp_work/prac
Finished Parsing XDC File [/afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/computerdesign/Practicals_2018/temp_
```

Create a schematic from selected objects

Figure 18: Screenshot showing the resulting gate-level implementation


Implementation Completed

Implementation successfully completed.

Next

( ) Open Implemented Design

[✓ Generate Bitstream]

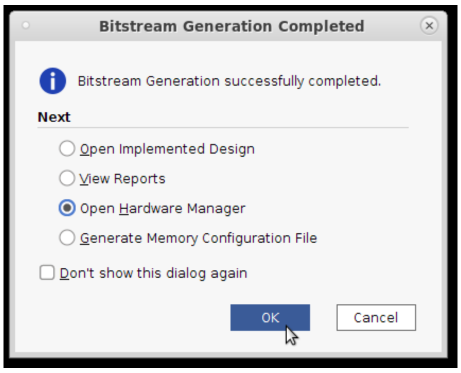
( ) View Reports

[ ] Don't show this dialog again

Figure 19: Screenshot showing implementation options

updated Hardware pane as shown in Figure 21.

Then click on the link to *Program Device*. This pops up the dialog shown in Figure 22, where you can change the bitstream file and/or select a file that identifies internal signals that you want to probe for debugging purposes. Don't change anything, just click *Program*. When the upload completes, in a few seconds at most, you should find that the buttons on the PYNQ Z2 board will illuminate the LEDs as you would expect. Congratulations, you have programmed the FPGA, and


Bitstream Generation Completed

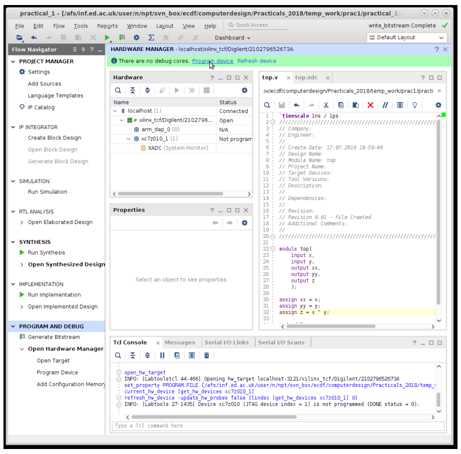
Bitstream Generation successfully completed.

Next

- Open Implemented Design
- View Reports
- [✓ Open Hardware Manager]
- Generate Memory Configuration File

☐ Don't show this dialog again

Figure 20: Screenshot showing further options when bitstream generation is complete


HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/210279652673A

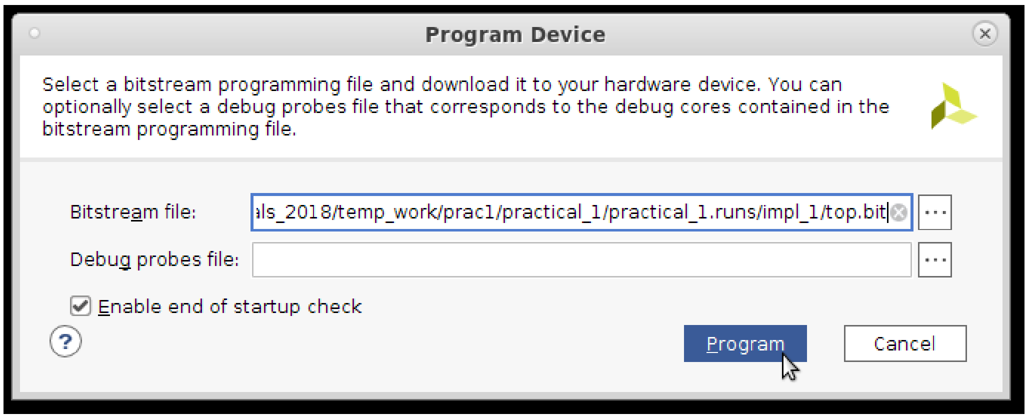
There are no debug cores. Program device Refresh device

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 17.07.2019 10:59:49
// Design Name:
// Module Name: top
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////

module top(
    input x,
    input y,
    output xx,
    output yy,
    output z
    );

assign xx = x;
assign yy = y;
assign z = x ^ y;
```

```
open_hw_target
INFO: [Labtoolstcl 44-466] Opening hw_target localhost:3121/xilinx_tcf/Digilent/210279652673A
set_property PROGRAM.FILE {/afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/computerdesign/Practicals_2018/temp_
current_hw_device [get_hw_devices xc7z010_1]
refresh_hw_device -update_hw_probes false [lindex [get_hw_devices xc7z010_1] 0]
INFO: [Labtools 27-1435] Device xc7z010 (JTAG device index = 1) is not programmed (DONE status = 0).
```

Figure 21: Screenshot showing a PYNQ Z2 board ready for programming

completed the first part of this exercise!


Program Device

Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.

Bitstream file: als_2018/temp_work/prac1/practical_1/practical_1.runs/impl_1/top.bit

Debug probes file:

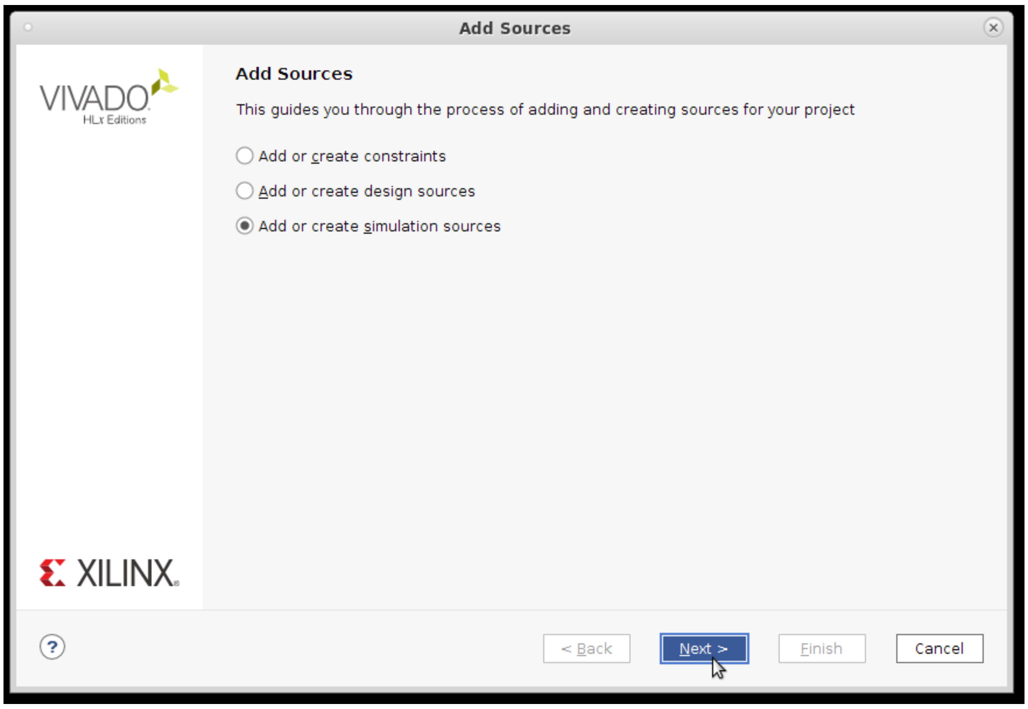
☑ Enable end of startup check

Figure 22: Screenshot for programming a device

8.1 Design Simulation

Verifying functionality using behavioural simulation is typically carried out before a design is synthesized, to verify that the logic design is correct. This allows a designer to find and fix any bugs in the design before spending time with subsequent steps. Of course, for the simple designs we will work with on this course, simulation is not really necessary since the design can be downloaded to the FPGA and tested in hardware. However, when a design is mapped to an FPGA there is much less visibility of the signals, and less control over the inputs, than can be achieved in a simulation environment. It is easy enough to return to the design, edit it and reprogram the FPGA. However, for a more complete familiarization with the tools, it is worth running through the simulation of even this simple design. In order to simulate the design, a *test bench* (TB) is required to drive the *design under test* (DUT). The TB is normally a top-level module that instantiates the DUT and therefore controls all inputs to the DUT. A *self-checking* TB will compare the outputs produced by the DUT against the expected behaviour, and will exercise the DUT sufficiently to obtain adequate *coverage* of the design.

Create a new source file for the test bench. Under the Project Manager options select Add Sources, as you did when creating `top.v`. The first of the Add Source dialogue boxes will appear, as shown in Figure 23. Select *Add or create simulation sources* to indicate you are creating a Verilog test bench module. Click Next to move on to the next set of options.



Add Sources

This guides you through the process of adding and creating sources for your project

- Add or create constraints
- Add or create design sources
- [✓ Add or create simulation sources]

Figure 23: Creating a simulation test-bench source file

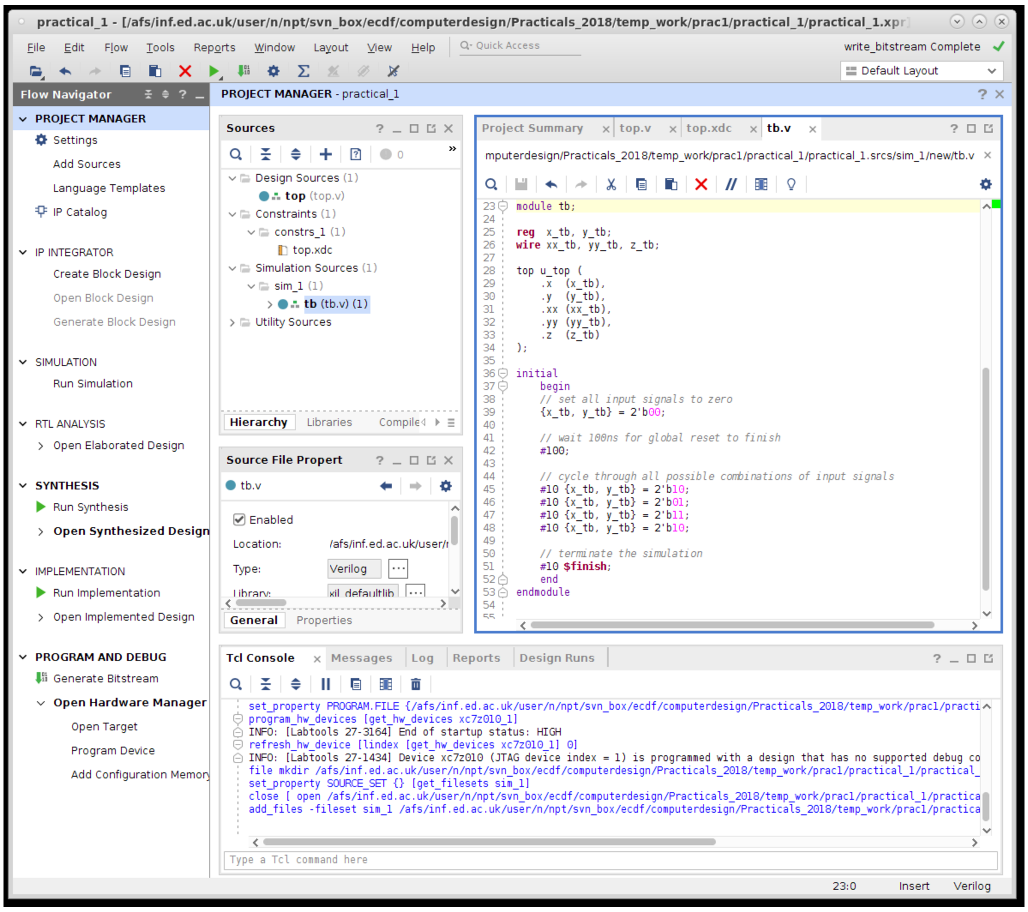
Click again on the **Create File** symbol, as you did when creating `top.v`. When the **Create Source File** dialog pops up set the file type to Verilog, and provide the file name `tb.v` in the same way you did for `top.v`.

As with the creation of the top.v file, you are now presented with a dialogue in which you can define the port names and directions. A standalone test-bench normally needs no ports, so you can click OK immediately. It will ask if you really meant to define an empty port list, which you should confirm.

The newly-created file will now appear in the *Simulation Sources* tree of the Sources window. When you double-click on the new file it will open in the right-hand editing pane, with an empty module definition, where you can then edit it to create the test-bench code. Before reading any further, think for a moment about what the TB needs to do, and consider the list of hints below. This is effectively a check-list for the contents of a simple test-bench for a purely combinational DUT, such as `top.v`.

1. Declare wires for each input and output of the DUT.
2. Instantiate the DUT, attaching the declared wires to the ports of the DUT.
3. Create an initial block, containing the following:
   (a) Set to 1’b0 all wires that drive DUT inputs.
   (b) Wait for 100ns to allow global reset to be completed.
   (c) At 10ns intervals, change the input signals so that eventually all possible input combinations have been presented to the DUT.
4. Terminate the simulation, using the `$finish` directive.

Figure 13 shows the code required to perform the steps identified in the check-list, which you should now add to your `tb.v` test-bench file. Note the `#10` statements before the assignments, these are important as they each specify a time delay that will precede the next statement.



```
module tb;

reg  x_tb, y_tb;
wire xx_tb, yy_tb, z_tb;

top u_top (
    .x  (x_tb),
    .y  (y_tb),
    .xx (xx_tb),
    .yy (yy_tb),
    .z  (z_tb)
);

initial
    begin
    // set all input signals to zero
    {x_tb, y_tb} = 2'b00;

    // wait 100ns for global reset to finish
    #100;

    // cycle through all possible combinations of input signals
    #10 {x_tb, y_tb} = 2'b10;
    #10 {x_tb, y_tb} = 2'b01;
    #10 {x_tb, y_tb} = 2'b11;
    #10 {x_tb, y_tb} = 2'b10;

    // terminate the simulation
    #10 $finish;
    end
endmodule
```

```
set_property PROGRAM.FILE {/afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/computerdesign/Practicals_2018/temp_work/pracl/practi
program_hw_devices [get_hw_devices xc7z010_1]
INFO: [Labtools 27-3164] End of startup status: HIGH
refresh_hw_device [lindex [get_hw_devices xc7z010_1] 0]
INFO: [Labtools 27-1434] Device xc7z010 (JTAG device index = 1) is programmed with a design that has no supported debug co
file mkdir /afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/computerdesign/Practicals_2018/temp_work/pracl/practical_1/practical_
set_property SOURCE_SET {} [get_filesets sim_1]
close [ open /afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/computerdesign/Practicals_2018/temp_work/pracl/practical_1/practica
add_files -fileset sim_1 /afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/computerdesign/Practicals_2018/temp_work/pracl/practica
```

Figure 24: Contents of the tb.v test-bench file. Note the property pane, showing that the Synthesis and Implementation properties have been deselected for tb.v.

Simulate your design as follows:

1. Click on the Run Simulation button in the left-hand Project Manager pane
2. Select the first option, to Run Behavioural Simulation

You should see that the main working area changes to show the *Behavioural Simulation* pane. This contains the structure of the simulated system on the left, then the objects of the simulated system in the centre, and the source code and waveform display tabs on the right. Check that the *Tcl Console* at the bottom of the display reports *Xsim completed.* If it does not, then look through the log and message tabs to diagnose the problem (ask an instructor if you can not figure out the problem).

When the simulation completes successfully, you will see a tab entitled *Untitled:1*. Select that tab and you will see a waveform display, which is effectively the a time trace of all signals in the test-bench module. You should see a display similar to the one shown below in Figure 14.

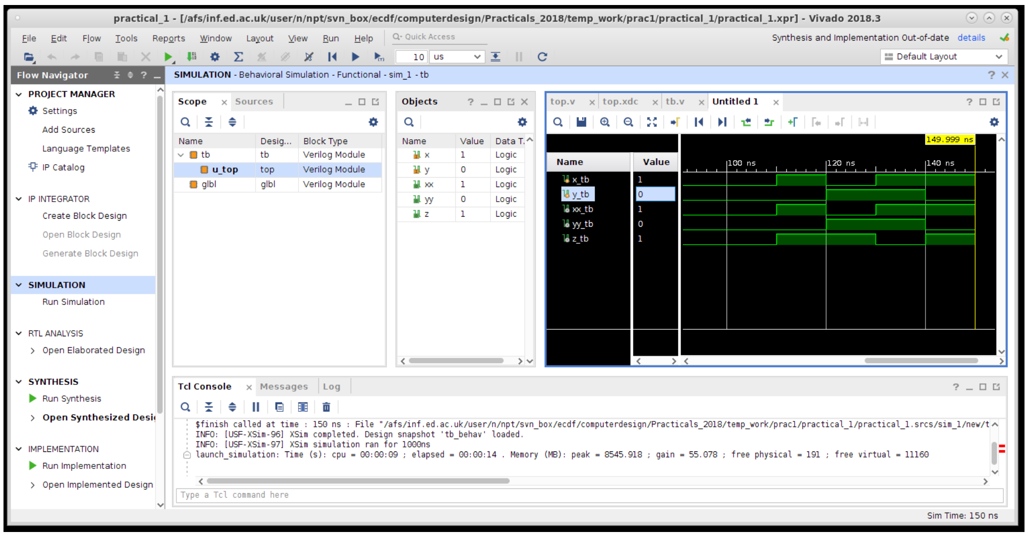

```
$finish called at time : 150 ns : File "/afs/inf.ed.ac.uk/user/n/npt/svn_box/ecdf/computerdesign/Practicals_2018/temp_work/prac1/practical_1/practical_1.srcs/sim_1/new/t
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:09 ; elapsed = 00:00:14 . Memory (MB): peak = 8545.918 ; gain = 55.078 ; free physical = 191 ; free virtual = 11160
```

Figure 25: Waveform display, after simulating the test-bench and the DUT together.

You may need to adjust the window dimensions, and select the period of time that you want to view, in order to see the behaviour of the system from 100ns to 150ns. This can be achieved using the zoom buttons (one of which is highlighted in Figure 14). If you firstly zoom to show the entire time window, you can then click and drag within the waveform display to select a period of time to zoom into. Use these zoom facilities to obtain a similar display to the one shown here, and familiarize yourself with the waveform display in general. This will prove useful for future exercises. You have now completed the functional simulation of your first design using the Vivado simulator.

IMPORTANT: You have now finished the first part of the practical and before you do anything more, call an instructor over to check your working XOR function programmed on the board. The demonstrator may wish to cycle power to make sure that the XOR function is only programmed in the FPGA.

After having shown the demonstrator your working XOR gate, continue with the full-adder below.

9 Further Familiarization

This part of the practical uses the work flow that you have learned from the simple XOR module, but with a slightly larger and more interesting design. Please refer to the previous exercise for information how to construct and simulate your design, and to program the FPGA, as this is not covered in detail again.

9.1 Designing a Full-adder

A full-adder is a logic block that has three inputs (A, B and C_{in}) and two outputs (Sum and C_{out}). It's function is, as its name suggests, that of adding numbers in a given binary representation. We will work with what most consider the normal binary representation, which is 2s complement.

The truth table of a 2s complement full-adder is shown below. You may think of this as adding up the number of 1s in the three inputs and delivering the result as a 2-bit binary value with Sum as bit 0 and C_{out} as bit 1.

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

9.2 Hardware Design

There are several ways to describe the logic of a full-adder in Verilog. At the highest level of abstraction we can use the '+' operator, and write the entire functionality of a full adder in the very convenient form shown below:

```
{C_out, Sum} = A + B + C_in;
```

However, this is **not** the way you will be writing it for this practical exercise! The goal here is for you to exercise your logic design skills and create a minimized boolean expression for the Sum and C_{out} signals by hand.

To implement this design manually you first need to draw up the Karnaugh maps for each of the two outputs, and then code these up as Verilog assignment statements, in a new Verilog file. You do not have to create a new project for this, but you may wish to do so; whether you do or not, your full-adder module should be in a separate file.

The number of inputs and outputs to the top level of the design is different, and you therefore need to change the constraints file (`top.xdc`) accordingly. Keep the outputs connected to the LED ports, but now connect the three inputs to three of the four slider switches. This allows you to set them to a 1 or 0 and then leave them set at that value for as long as you wish. Obviously, you will be using one more input and one less output LED (there is no need to display the inputs, just the Sum and C_{out}). To enable instructors to test your design, please use the assignment of switches to the inputs, and LEDs to the outputs, as shown in Table 2 below:

It is strongly recommended that you create a new Verilog module for your full-adder, and instantiate this module in the top level module of the project. The reason is that later exercises assume you either have a full-adder module or can make one with no trouble. You already know how to create a new Verilog source file; just follow this procedure again to create a full-adder source file containing your full-adder module.

After creating the module, you need to fill it in with two *assign* statements, one for each of the outputs, which should have the logic you have deduced from the Karnaugh maps.

I/O Signal	I/O Device	Zync Pin
A	sw[0]	G15
B	sw[1]	P15
C_{in}	sw[2]	W13
Sum	led[0]	M14
C_{out}	led[1]	M15

Table 2: Required I/O device usage for the full-adder exercise

You then need to instantiate this full-adder module in your top level module. You will have seen module instantiation already, when instantiating the `top` module in the test-bench during simulation of the XOR gate. A typical instantiation of your full-adder module (called `fulladder`) may look like this:

```
  fulladder u_fulladder (
    .A    (in1),
    .B    (in2),
    .Cin  (in3),
    .O    (out1),
    .Cout(out2)
  );
```

Listing 1: Module instantiation example

This instantiation inserts the `fulladder` module into the enclosing module, names this instance of it `u_fulladder`, and connects the named ports (A, B, and so on) of the `fulladder` module to the wires of the current module given as actual arguments. Note, an input argument can be a `reg` variable, but all output arguments must be `wire` variables.

If you think of modules as classes, in the C++ or Java sense, and instantiated modules as an object of that class, you are not too far off.

After instantiating your fulladder module, simulate the design, correct any errors, and synthesize it. Load it into the FPGA to test your full-adder.

Recommendation: At this point you should show your design and your Karnaugh maps to an instructor so that you can receive feedback on the second part of the exercise.

10 Synchronous Processes

The third and final part of this assignment involves the use of *synchronous processes*. So far, all circuits have been *combinational*, meaning that all outputs can be represented as boolean functions of the input signals. Synchronous processes introduce a clock signal into the design, which is then used to define the points at which register values are updated. Those register values may define outputs, or may be used internally within a module. This section of the practical assignment introduces the standard approach to describing synchronous elements within a Verilog design. As you might expect, any register value that is updated on the transition of a clock signal can be implemented as an edge-triggered D-type flip-flop.

10.1 Memory elements

Memory elements, or flip-flops, can be made from discrete gates. For example, a simple R-S latch can be constructed from two NAND gates with cross-connected feedback paths. We have also seen (or will shortly cover) in lectures, the gate-level design of more sophisticated flip-flops, such as D-type latches and edge-triggered D-type flip-flops. These may also have a *clear* or *preset* input to force the flip-flop to zero or one, respectively. The edge-triggered D-type flip-flop (or simply flip-flop from hereon) is an extremely useful abstraction for logic designers, and is used almost exclusively in digital logic design as the memory element of choice. However, instead of instantiating a gate-level representation of a flip-flop every time we need a 1-bit memory element, we rely on Verilog language constructs to *infer* flip-flops by using a synchronous process. A simple example of such an inference is shown below.

```
wire clock;
wire D;
reg  Q;

always @(posedge clock)
begin
  Q <= D;
end
```

Listing 2: D-type flip-flop

You will note several new things about this snippet of Verilog; first of all, the declaration of `Q` using a `reg` statement, the `always` statement, the `begin` and `end` statements, and finally, the <= assignment operator. If you do not understand these elements, a brief explanation follows below; it is also recommended that you consult the lecture slides, or an online Verilog tutorial (and there are many) if you are still unsure.

- The declaration of a variable as a `reg` type is necessary for any variable that is assigned within a Verilog process (or task), and indicates that the value of the variable will persist after each assignment has taken place.

- The always block executes every time anything listed in its *sensitivity list* changes; here, we're limiting that to every time there is a positive edge on the clock wire. You may recall from lectures that we should always use non-blocking assignment, *i.e.* the <= assignment operator, when inside a synchronous process. In contrast, only blocking assignment *i.e.* the = assignment operator, should be used inside a combinational process. A synchronous process mentions `posedge` (or sometimes `negedge`) when specifying the signals in its sensitivity list. Conversely, a combinational process simply lists the signals on which the evaluation of the process depends, or in Verilog 2000 we may simply write `(*)` to tell the Verilog compiler to figure out what signals it ought to be sensitive to.

- The begin / end statements are Verilog's equivalent of {/} in Java or C, and delimits the body of the process.

- The <= operator is a non-blocking assignment. This differs from a blocking assignment (=) in that it doesn't take effect until AFTER the always block has finished executing. The difference between blocking and non-blocking assignments is one of the most important distinctions in Verilog, and also unfailingly confuse those new to the language. You would do well to read up on the differences between them.

10.2 D-type with reset

The use of a reset signal is as important as the use of a clock signal, as it allows all flip-flops to start in a known state before the first clock edge. If we were to leave the reset signal out of a design, we would not be able to guarantee correct behaviour as each flip-flops would begin with an undefined value. In simulation, an undefined value is represented as the value 1'bx, and this means a simulation cannot run if flip-flops are not given a defined value (*i.e.*, 1'b0, or exceptionally 1'b1). The Verilog code for a D-type flip-flop with reset is shown below

```verilog
module Dtype(
  input  clock,
  input  reset,
  input  D,
  output reg Q
);

always @(posedge clock or posedge reset)
begin
    if (reset == 1'b1)
        Q <= 1'b0;
    else
        Q <= D;
end
endmodule
```

Listing 3: D-type flip-flop with reset

This is the standard template you should always use when inferring a flip-flop. Note that the process is sensitive to the rising edge of `reset`, and that in the body of the process the `Q` register is cleared when `reset` goes to a logic 1. We therefore say that *reset is active high*. Furthermore, the clearing of `Q` not only happens on the rising edge of `reset`, but `Q` remains held in that state while `reset` is asserted. This means that any positive edge on the `clock` signal will be ignored when `reset` is asserted.

When `reset` is rescinded, i.e. when it transitions to logic 0, the flip-flop once again becomes sensitive to a positive edge on the `clock` signal (or another assertion of the `reset` signal). You may, at this point, wonder what happens if the `reset` signal is rescinded at exactly the same time as the next rising edge of the `clock` signal. This is a topic for a later lecture, where we look at the problem of *synchronizing* signals to a clock. For now, you may assume that the `reset` signal can be asserted at any time, but will always be rescinded some short time after the rising edge of the `clock` signal, and never coincidentally with the `clock` signal. We refer to this reset strategy as *asynchronously asserted and synchronously rescinded*. This is the reset strategy that you should adopt in your designs.

10.3 Implementation

To demonstrate the use of a D-type flip-flop, we are going to use our full-adder together with a D-type with reset.

You need to first add the `clock` and `reset` input signals to the port list of your top-level design from the previous part, so that you have these signals available in your design. Connect these top-level ports to pins of the FPGA that are wired to the push buttons on the PYNQ Z2 board. Your top level module should then have five inputs and two outputs, with the I/O pin assignments in your `top.xdc` file as shown in Table 3 below:

Next connect one D-type flip-flop to each output from the full-adder. You will need to declare two new `wires` that internally connect the full-adder and the D-types. Connect the outputs from these flip-flops to the LED outputs the full-adder was originally connected to. The end effect is that the flip-flops are interposed between the full-adder and the top-level module outputs.

IMPORTANT: There is also an extra constraint you will need to add in your `.xdc` file, because you are using an external input signal as a clock within your design. The routing tool does not like using just any physical input as a clock, as it does not know how to constrain the speed of the logic in relation to the period of that clock. Furthermore, the routing of the clock input

I/O Signal	I/O Device	Zync Pin
A	sw[0]	G15
B	sw[1]	P15
C_{in}	sw[2]	W13
Sum	led[0]	M14
C_{out}	led[1]	M15
$clock$	btn[0]	R18
$reset$	btn[1]	P16

Table 3: Required I/O device usage for the full-adder exercise

is typically considered by the router to be sub-optimal. For the purposes of this exercise that does not matter, but we need to tell Vivado to override this normal restriction, by setting an additional property on the clock input. If this is not done, you will get errors `Place 30-574` and `30-99`. To do this, edit the `.xdc` file again, and add a new property for the `clock` net, after the two other lines that you already have for defining the `PACKAGE_PIN` and `IOSTANDARD` properties, thus:

```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets {clock}]
```

You will now get a warning instead of an error, but you are allowed to continue the synthesis process. In HDL terminology, a *net* is simply a wire.

After doing this you should be able to synthesize and implement the design, and then load it into the FPGA. Now, the LEDs should not change until you press the `clock` button, and the LEDs should similarly be low (off) after having pressed the `reset` button until the `clock` button is pressed again, at which point they may change.

You should now be able to simulate, test, and synthesize this design with no trouble, as well as test it on the PYNQ Z2 board. If you look in the synthesis report, you will see that it indicates your design uses 2 flip-flops.

Show your working full-adder with registered outputs to an instructor.

11 Further Remarks

You might have realized by now that Verilog provides a number of different ways of describing combinational logic. Firstly, we can use an `assign` statement, provided the value we are assigning is declared as a `wire` (and unless otherwise stated, an output port is also assumed to be declared this way). Secondly, if the value we want to assign is declared as a `reg`, then we can assign a value to it within either a combinational process or a synchronous process.

If we assign a value to a `reg` from within a combinational process, then it is re-assigned every time the process is triggered by a signal in its sensitivity list. Note, such a process will never be triggered, at least not in simulation, if there are no inputs!

If we assign a value to a `reg` from within a combinational process, but not every time the process is triggered, then the corresponding `reg` variable will create an *inferred latch*. This is almost always a coding error, and most code quality (linting) tools for Verilog will object when they see an inferred latch. Essentially, an inferred latch is extremely difficult to test during the post-production testing, if the inferred latch is put into a chip. And, rather importantly, it does not have a defined state on reset. It can also complicate the timing analysis during synthesis and layout, so it is always to be avoided.

So let's look again at the various ways of coding the adder. We have seen already how it can be coded using optimized boolean expressions for the Sum and C_{out} signals, and we have seen a one-line assignment statement for use within a combinational process. Within a combinational process we can also define the logic in the form of a truth table.

```
module fulladder(
  input      A,
  input      B,
  input      C_in,
  output reg Sum,
  output reg C_out
);

always @(*)
begin
  case ( {A, B, C_in} )
    3'b000: {C_out, Sum} = 2'b00;
    3'b001: {C_out, Sum} = 2'b01;
    3'b010: {C_out, Sum} = 2'b01;
    3'b011: {C_out, Sum} = 2'b10;
    3'b100: {C_out, Sum} = 2'b01;
    3'b101: {C_out, Sum} = 2'b10;
    3'b110: {C_out, Sum} = 2'b10;
    3'b111: {C_out, Sum} = 2'b11;
  endcase
end
endmodule
```

Listing 4: Adder implemented as a truth table

This approach is useful if the relationship between inputs and outputs cannot be represented using a higher-level mathematical operator, such as '+', or if you do not trust your manual logic minimization skills (or want to save design time). In general, it is always best to code your designs using the highest possible level of abstraction, as this maximizes design productivity.

However, suppose that I used the truth table approach shown above, but I accidentally omitted one of the case elements. Can you deduce what would happen in this case? There are two ways to catch these cases; one is to use a `default` case element, which assigns a default value to $\{C_{out}, Sum\}$, and the other is to assign a set of default values to $\{C_{out}, Sum\}$ before the case statement. This means that usually there will be two assignments to these variables, but this simply means that the final assignment overrides the earlier assignment(s). This is a common and accepted practice for combinational processes, where blocking assignments are used.

12 Conclusion

After you have completed all three parts of this exercise, you have a better understanding of the Xilinx Vivado toolset as well as some further understanding of Verilog. There is no report to write for this first lab, but the lab sheet which your lab demonstrator has signed should be handed in.