Project 1 A first project for the Boolean Board

With an introduction to Vivado and Verilog

Introduction

This first project introduces you to the Boolean board and the Xilinx Vivado design tool suite. It guides you through downloading and setting up your board and the tools, steps you through the process of creating a new design project, and demonstrates how to program your board. Many useful features of the Vivido design environment are introduced in this project, but it will take several projects to gain experience with all of the features.

The Vivado design tool lets you describe digital circuits using the Verilog or VHDL hardware description languages (we will use Verilog in this course). After you type in a Verilog circuit description, you can check the circuit's behavior using the built-in simulator, synthesize the circuit description into a form that can be programmed into the FPGA, and then program the FPGA and then interact with the design in the Boolean board. This last step, interacting with the actual design in hardware, is often the most important and revealing step in verifying overall design performance.

The ability to quickly and easily design, simulate and synthesize a digital circuit on your personal computer is hugely empowering. You can thoroughly check your understanding of any design, validate any given circuit's performance, probe behaviors that seem incorrect or mysterious, and gain a deeper understanding of circuits and design methods, all from your personal computer, and at your own pace. If you really want to learn and develop your skills as a digital design engineer, you will find no better learning environment than the Vivado tool, a Boolean board (or another Real Digital board), and your own desire and initiative.

When designing and implementing circuits in this course, the general flow is:

- understand the design goals and specifications;
- capture the design in an engineering formalism like a truth table, state diagram, and/or block diagram (this is usually a pencil-and-paper activity);
- type a Verilog description of the design into the Vivado tool;
- simulate the design to verify it works correctly;
- synthesize the design into a ".bit" file that contains a physical description of the circuit that can be programmed into the FPGA;
- transfer the .bit file to the FPGA using Vivado's "hardware manager" tool to configure/program the FPGA with your circuit;
- and finally, interact with the design to verify that it meets the original design specification.

With a little practice, you will find the tools, although complex, will not hinder your progress - you will be limited only by your knowledge base, your tenacity, and your creativity. The first few projects will introduce the Vivado tools a little at a time, but soon, you will be familiar with all the tools needed to perform all required steps.

You will be asked to demonstrate your work to the lab assistant. The assistant will assign scores during your demonstration and record them on the project submission forms. Remember that we will likely ask probing questions about your design during the submission.

Before you begin, you should:

- Have a Real Digital board (given in the lab);
- Have a PC running Linux or Windows with internet access;
- Have an hour or two to get your design environment up and running (you should have done this already).

After you're done, you should:

- Have the Xilinx Vivado tool installed and running;
- Know how to set up a project in Vivado;
- Know how to enter a basic Verilog circuit description;
- Know how to connect GPIO devices to your circuit.;
- Be able to configure an FGPA on a Real Digital platform from Vivado.

Background

This first project introduces the Xilinx Vivado design software that you will be using throughout the course. Vivado provides a complete, self-contained design environment that lets you create and simulate Verilog circuit descriptions, and then synthesize and download them to your board for hardware validation.

LAB TASKS:

☑ 1. Complete the first tutorial **PAGE 3**

This tutorial provides a first experience with the Vivado tool and the Boolean board. It steps through the entire circuit definition flow: creating a new Vivado project; entering a verilog hardware description; creating a constraints file to map inputs/outputs to device pins; synthesizing the design; creating a ".bit" programming file; and programming a device. In order to focus on the tool flow, you can copy-and-paste (and/or download) the prewritten Verilog and constraints files included in the tutorial. After completing the tutorial, your board will have been configured with the demo program, you will see various flashing patterns on the LEDs and seven-segment display.

☑ 2. Complete the second tutorial **PAGE 19**

In this tutorial, you will redo all the steps in the first tutorial, but this time, you will create your own circuit using Verilog, and create your own constraints file to map the input and output signals. After you program your device, SW0 should control LED0.

☑ 3. Modify your code to turn on all LEDs

Modify your code to connect all slide switches to all LEDs on your board, with each slide switch turning on the LED above it.

☑ 4. Reverse the order

Instead of turning on the LED above each switch, reverse the order so SW0 turns on LED15, SW1 turns on LED14, etc. Can you do this by modifying your Verilog source file? Or by modifying the xdc constraints file? Would either work? Try it! Try both!

☑ 5. Illuminate the seven-segment display

One of the topic documents above provides some useful background information on the seven-segment display (SSD) - you may want to read through that document before working on this requirement.

Connect the first eight slide switches to the seven segments and decimal point on the seven-segment display, and connect the four pushbuttons to the four anodes of DISP1 (the anodes act like "on/off" switches for the individual digits). Press the buttons and verify that segments illuminate according to the settings of the slide switches.

Introduction to Vivado

Creating a Vivado Project for the Boolean Board

This document will guide you through the creation of a first Vivado project for Boolean board. It covers:

- How to create a new project;
- How to edit a project and add required source files;
- How to Synthesize and Implement a project, and how to generate a bitstream;
- And finally, how to program your Real Digital board.

Looking ahead, you will probably want to create a new project every time you start a new design. You will need to repeat these same steps whenever you create a new project, so you may revisit this document several times.

Step 1: Create a Vivado Project

Vivado Projects

Vivado "projects" are directory structures that contain all the files needed by a particular design. Some of these files are user-created source files that describe and constrain the design, but many others are system files created by Vivado to manage the design, simulation, and implementation of projects. In a typical design, you will only be concerned with the user-created source files. But, in the future, if you need more information about your design, or if you need more precise control over certain implementation details, you can access the other files as well.

When setting up a project in Vivado, you must give the project a unique name, choose a location to store all the project files, specify the type of project you are creating, add any pre-existing source files or constraints files (you might add existing sources if you are modifying an earlier design, but if you are creating a new design from scratch, you won't add any existing files – you haven't written them yet), and finally, select which physical chip you are designing for. These steps are illustrated below.

Start Vivado

In Windows, you can start **Vivado** by clicking the shortcut on the desktop. In Linux, you can start **Vivado** using the command in the terminal as shown in Vivado Tutorial (in Stud.IP).

After **Vivado** is started, the window should look similar to the picture in figure 1.

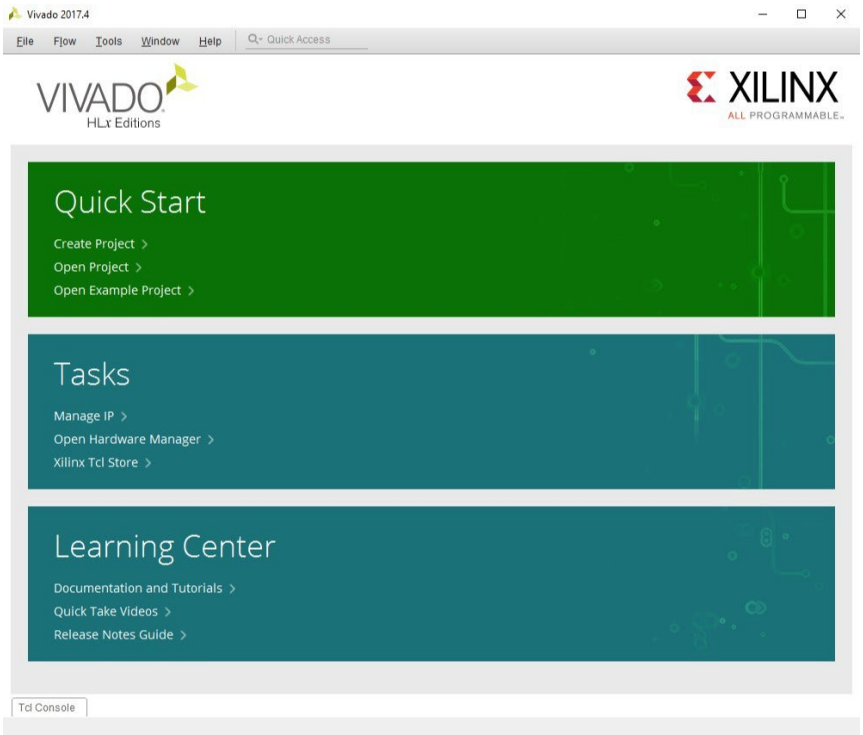


Figure 1. Vivado Start-Up Window

Open Create Project Dialog

Click on "Create Project" in the Quick Start panel. This will open the New Project dialog as shown in Figure 2. Click Next to continue.

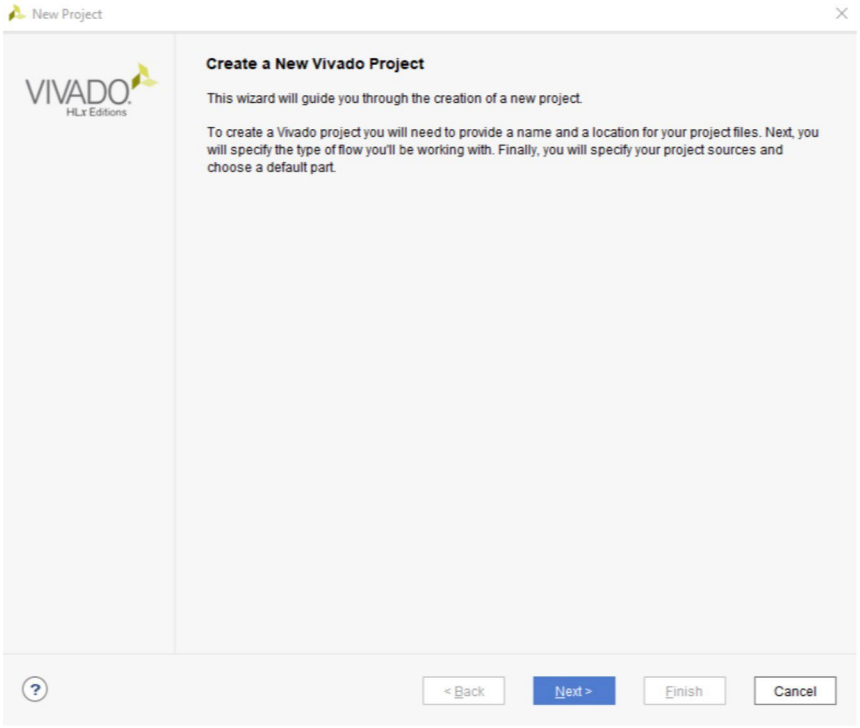


Figure 2. Create Project Dialog

Set Project Name and Location

Enter a name for the project. In the figure, the project name is "project_1", which isn't a particulary useful name. It's usually a good idea to make the project name more descriptive, so you can more readily identify your designs in the future. For example, if you design a seven-segment controller, you might call the project "seven segment controller". For projects related to coursework, you might include the course name and project number - for example, "ee214_project2". You should avoid having spaces in the project name or location, because spaces can cause certain tools to fail.

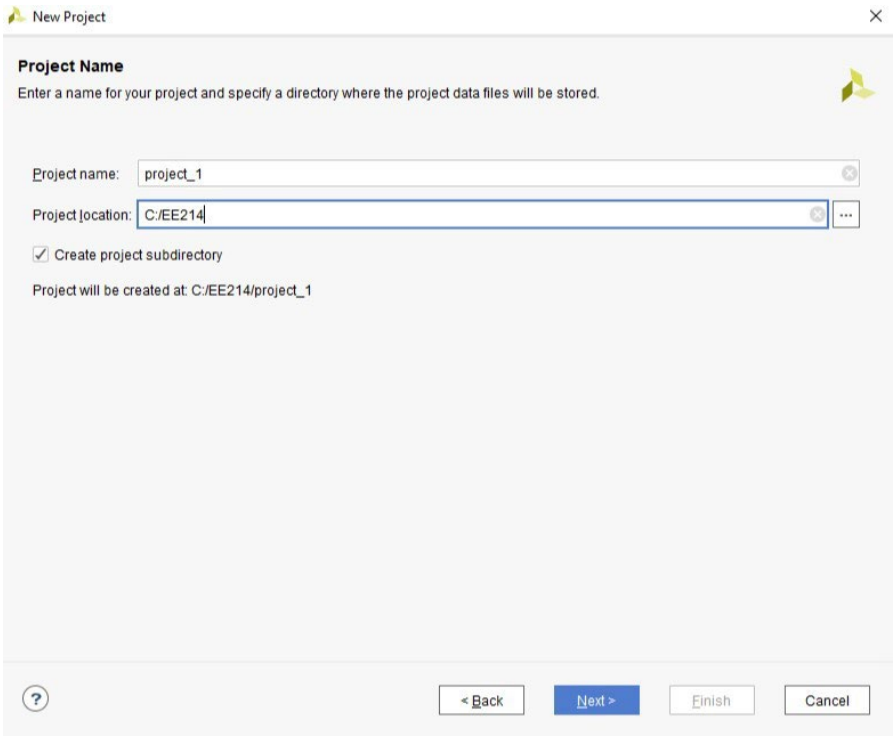


Figure 3. Enter Project Name

Select Project Type

The “project type” configures certain design tools and the IDE appearance based on the type of project you are intending to create. Most of the time, and for all Real Digital courses, you will choose “RTL Project” to configure the tools for the creation of a new design. (RTL stands for Register Transfer Language, which is a term sometimes used to mean a hardware design language like Verilog).

Make sure RTL project is selected, and click Next .

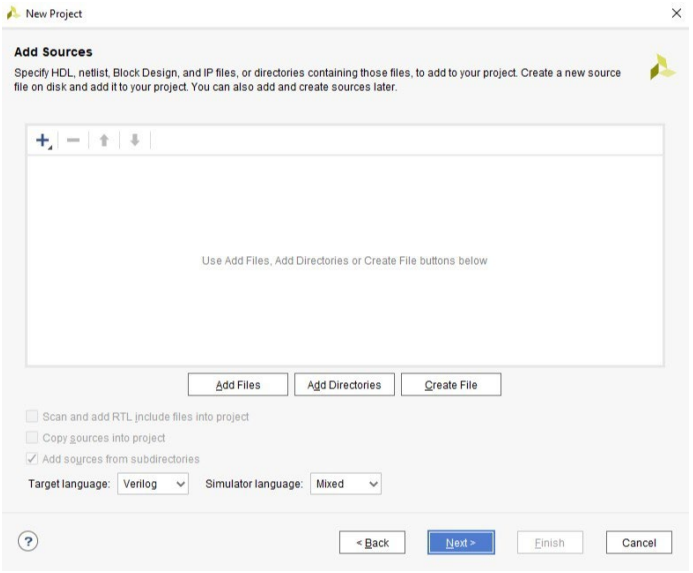


Figure 4. Select Project Type

Add Existing Sources

In a typical new or early-stage design, you won’t add any existing sources because you haven’t created them yet. But as you complete more designs and build up a library of previously completed and known good designs, you may elect to add sources and them use them in a new design.

For now, there are no existing sources to add, so just click Next .

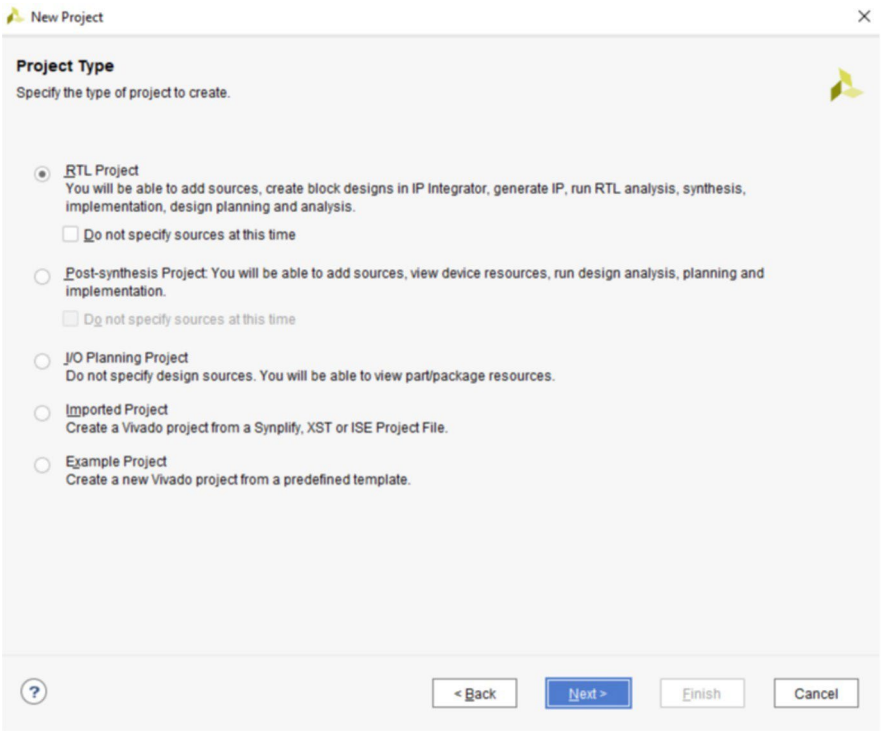


Figure 5. Add Sources

Add Constraints

Constraint files provide information about the physical implementation of the design. They are created by the user, and used by the synthesizer. Constraints are parameters that specify certain details about the design. As examples, some constraints identify which physical pins on the chip are to be connected to which named circuit nodes in your design; some constraints setup various physical attributes of the chip, like I/O pin drive strength (high or low current); and some constraints identify physical locations of certain circuit components.

The Xilinx Design Constraints (.xdc filetpye) is the file format used for describing design constraints, and you need to create an .xdc file in order to synthesize your designs for a Real Digital board. Later in this tutorial, you will create a constraints file to identify which named circuit nodes must be connected to which physical pins. But for now, you have no existing constraints file to add, so you can simply click Next .

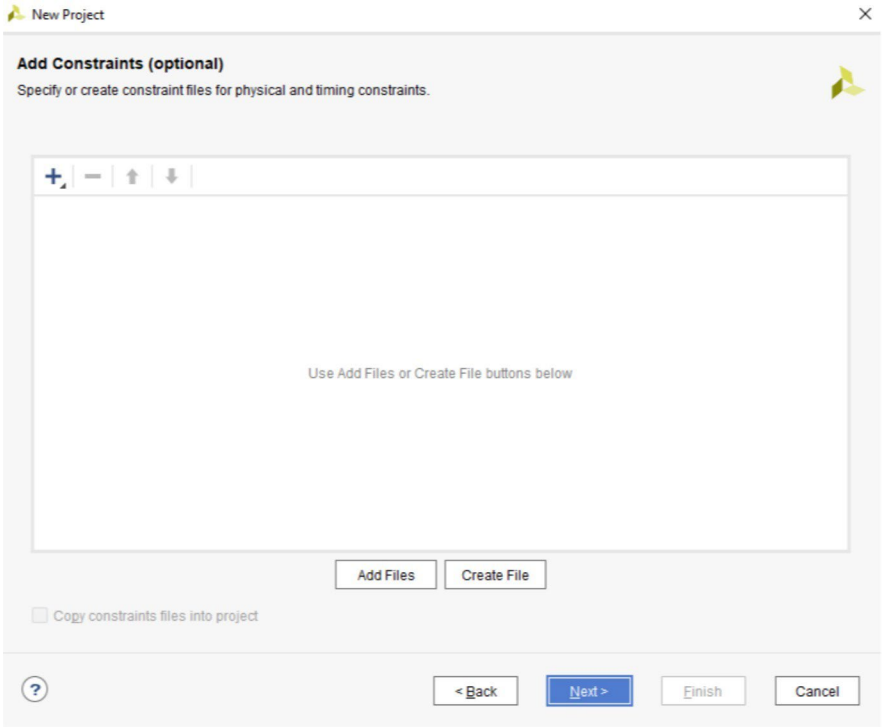


Figure 6. Add Constraint Files

Select Parts

Xilinx produces many different parts, and the synthesizer needs to know exactly what part you are using so it can produce the correct programming file. To specify the correct part, you need to know the device family and package, and less critically, the speed and temperature grades (the speed and temperature grades only affect special-purpose simulation results, and they have no effect on the synthesizer's ability to produce accurate circuits). You must choose the appropriate part for the device installed on your board.

The Boolean uses a **xc7s50csga324-1** Spartan-7 device with the following attributes:

Part Number	**xc7s50csga324-1**
Family	**Spartan-7**
Package	**ccsga324**
Speed Grade	**-1**
Temperature Grade	**C**

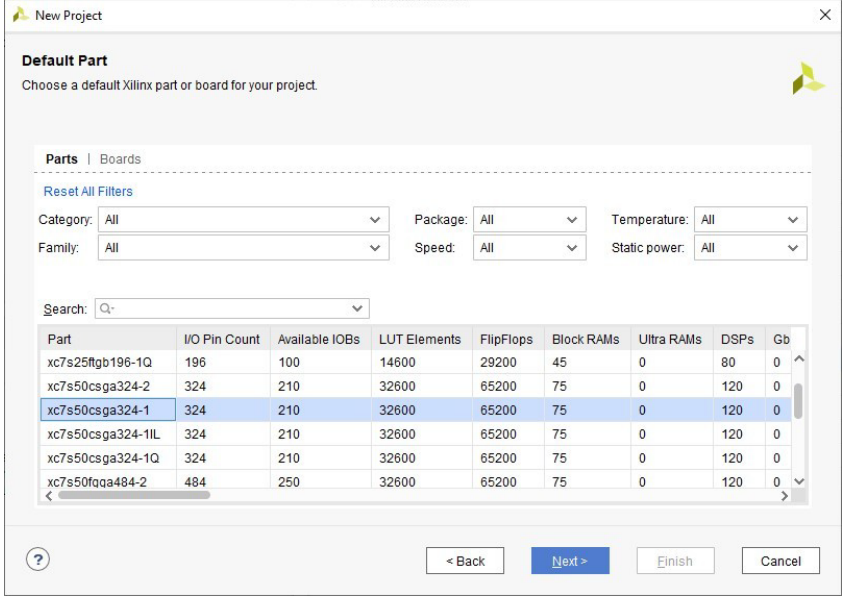


Figure 7. Select Zynq 7000 Part

How to find your Zynq 7000 SoC Parts

You can find the details about your xilinx part from the markings on the IC, in the reference manual for your board, on the page where you purchased the board, or in the board's schematic.

From Identification Marking on IC

Usually, you can find the part number directly from the marking on the surface of the chip. In the picture below, you can see the part number for the Spartan-7 device found on the Boolean board.

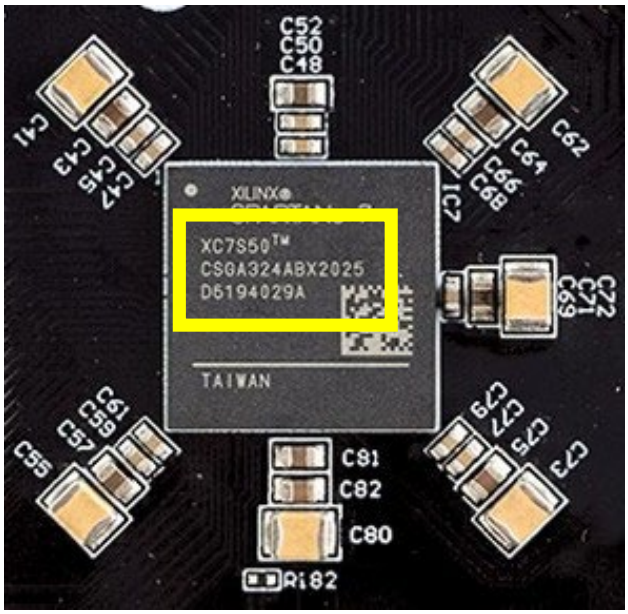


Figure 8. Boolean board IC Marking

From Board Schematic

You can also find the part number from your board's schematic. The figure below shows the Boolean board's FPGA as identified in it's schematic.

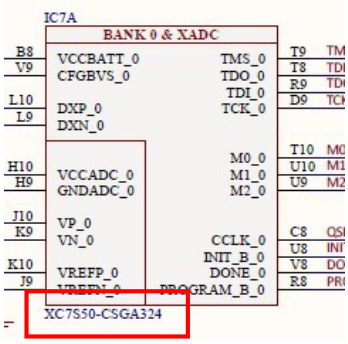


Figure 9. Boolean board part number in schematic

Check Project Configuration Summary

The last page of the Create Project Wizard shows a summary of the project configuration is shown. Verify all the information in the summary is correct, and in particular make sure the correct FPGA part is selected. If anything is incorrect, click back and fix it; otherwise, click **Finish** to finish creating an empty project.

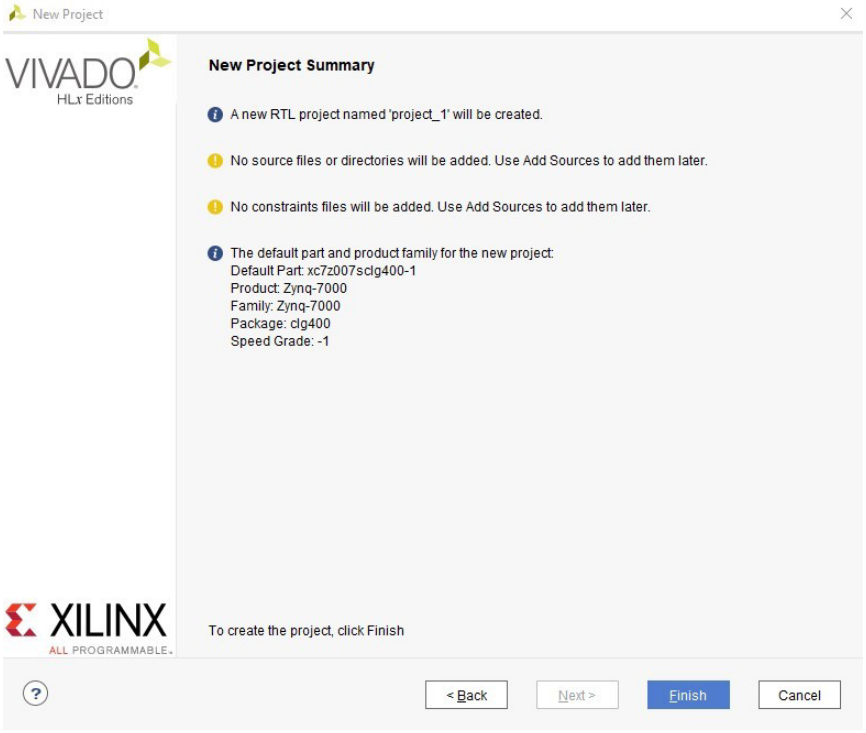


Figure 10. Create Project Summary

Vivado Project Window

After you have finished with the Create Project Wizard, the main IDE window will be displayed. This is the main "working" window where you enter and simulate your Verilog code, launch the synthesizer, and program your board. The left-most pane is the flow navigator that shows all the current files in the project, and the processes you can run on those files. To the right of the flow navigator is the project manager window where you enter source code, view simulation data, and interact with your design. The console window across the bottom shows a running status log. Over the next few projects, you will interact with all of the panels.

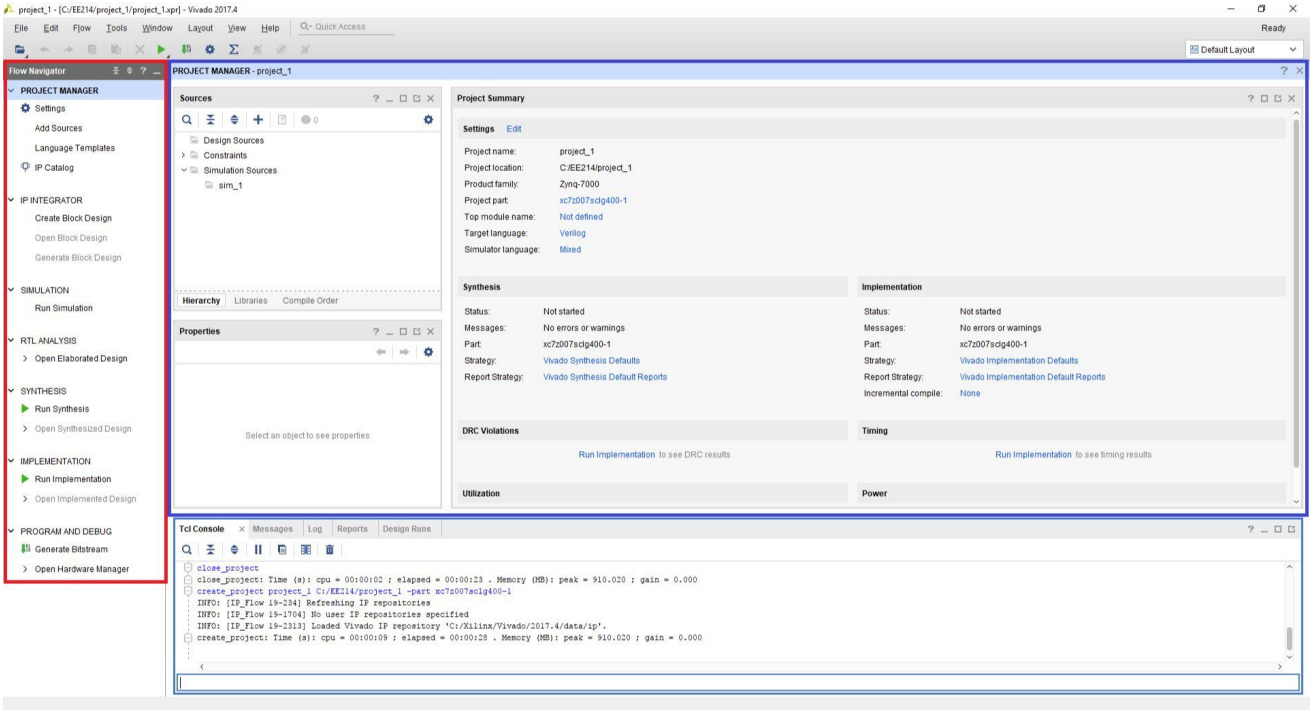


Figure 11. Vivado Project Window

Step 2: Edit The Project - Create source files

All projects require at least two types of source files – an HDL file (Verilog or VHDL) to describe the circuit, and a constraints file to provide the synthesizer with the information it needs to map the circuit into the target chip. In the following exercise, we will:

- create a Verilog source file to define an example circuit's behavior (for this exercise, an example file is provided for you to copy);
- create a constraints file to define how the Verilog circuit is mapped into the Xiling logic device (also provided);
- synthesize the source and constraint file into a ".bit" file that can be programmed onto your board;
- configure the Boolean board with the circuit.

After the Verilog source file is created, it can be directly simulated. Simulation (discussed in more detail in a later project) lets you work with a computer model of a circuit, so you can check its behavior before taking the time to implement it in a physical device. The simulator lets you drive all the circuit inputs with varying patterns over time, and lets you examine the outputs to verify whether they behave as expected under all conditions.

After the constraint file is created, the design can be synthesized. The synthesis process translates Verilog source code into logical operations, and it uses the constraints file to map the logical operations into a given chip. In particular, the constraints file defines which Verilog circuit nodes are attached to which pins on the Xilinx chip package, and therefore which physical devices on your board. The synthesis process creates a ".bit" file that can be directly programmed into the Xilinx chip.

In this exercise, the Verilog and constraint source files are provided, so you can copy them into the project instead of typing them (in later designs, you will create these files yourself).

Design Sources

There are many ways to define a logic circuit, and many types of source files including VHDL, Verilog, EDIF and NGC netlists, DCP checkpoint files, TCL scripts, System C files, and many others. We will use the Verilog language in this course, and introduce it gradually over the first several projects. For now, you can get some basic familiarity with basic Verilog concepts here:

VERILOG HDL: A FIRST EXAMPLE

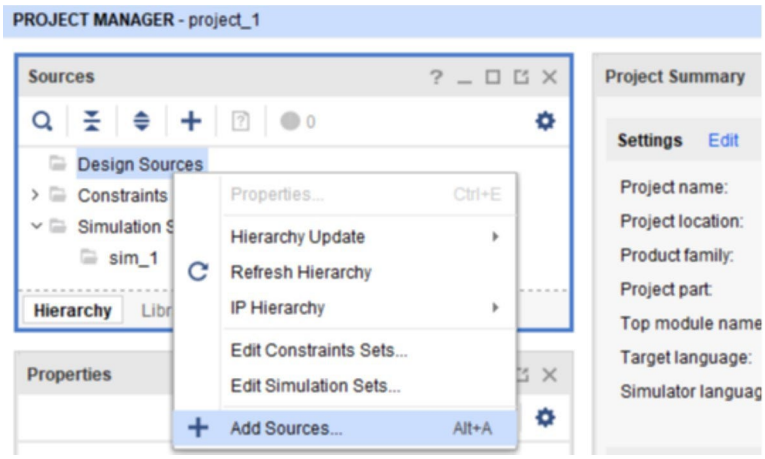


Figure 1. Add Design Sources

To create a Verilog source file for your project, right-click on "Design Sources" in the Sources panel, and select Add Sources (or equivalently, click on "Add Sources" in the Flow Navigator window under Project Manager). The Add Sources dialog box will appear as shown – select "Add or create design sources" and click **Next** .

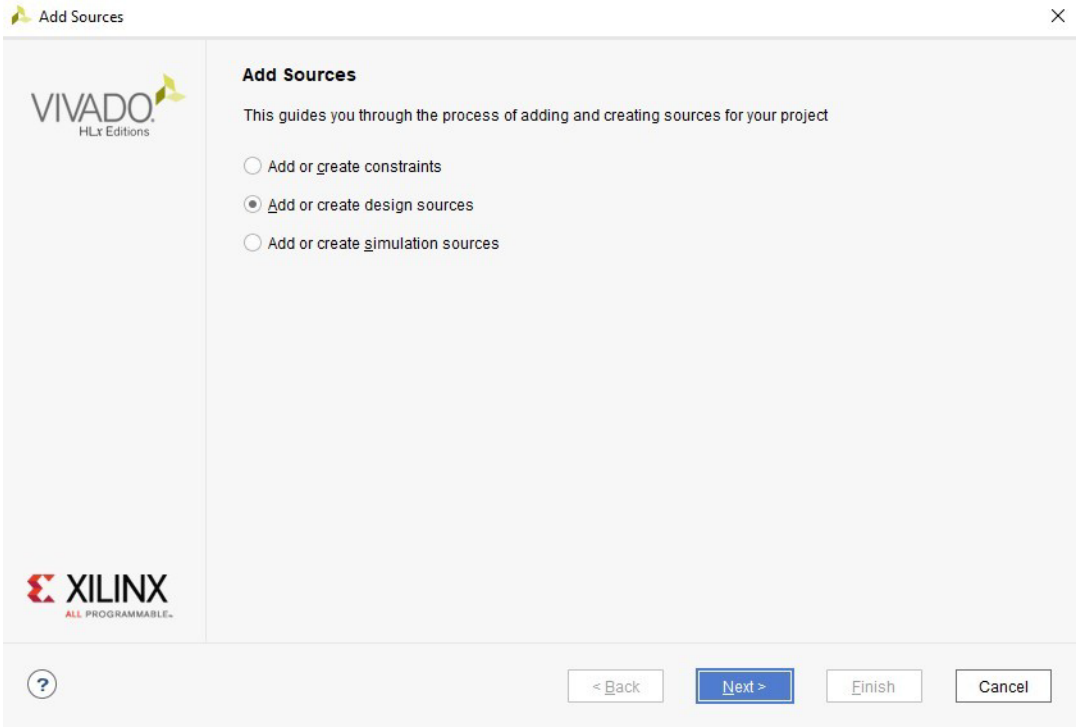


Figure 2. Add or create design sources using Add Source Dialog

In the Add or Create Design Sources dialog, click on **Create File** , enter `project1_demo` as filename, and click **OK** . The newly created file will appear in the list as shown. Click **Finish** to move to the next step.

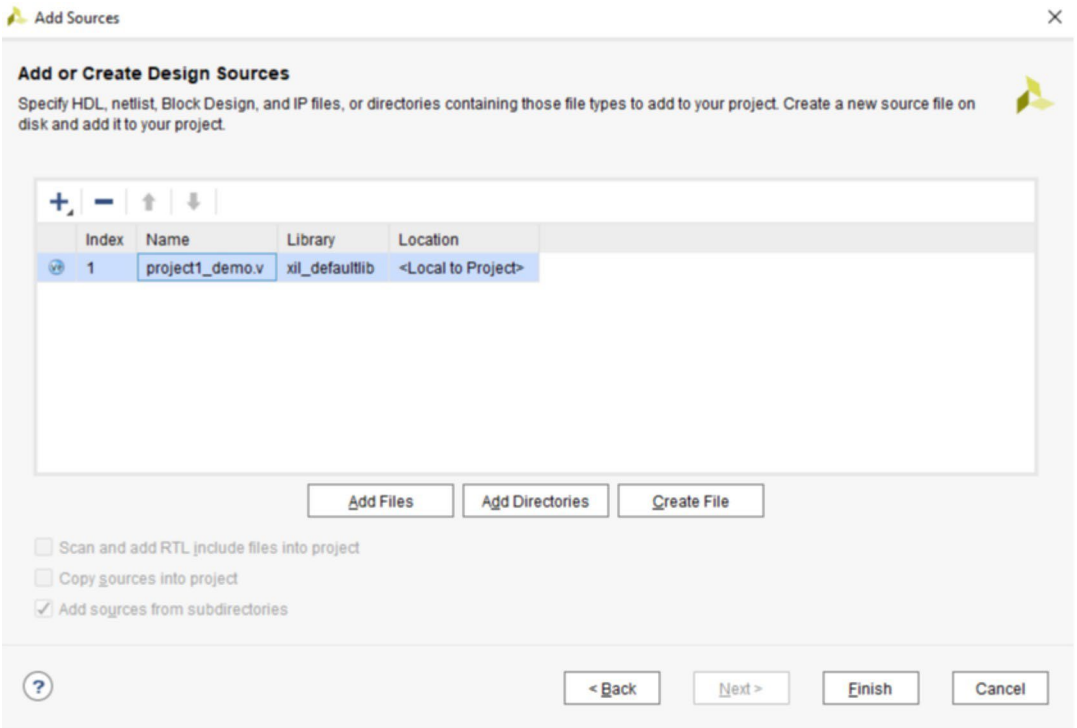


Figure 3. Create Design Source File

A “define module” dialog box opens and presents an area where you can define input and output ports for your circuit. In later projects, if you enter port names here, Vivado will create a Verilog source file with those ports already defined. For this exercise, the complete source file is provided, so you can skip this step by clicking OK (and then YES) to continue.

You will now see `project1_demo` listed underneath Design Sources folder in the Sources panel. Double click `project1_demo` to open the file, and replace the contents (copy and paste) with the code below.

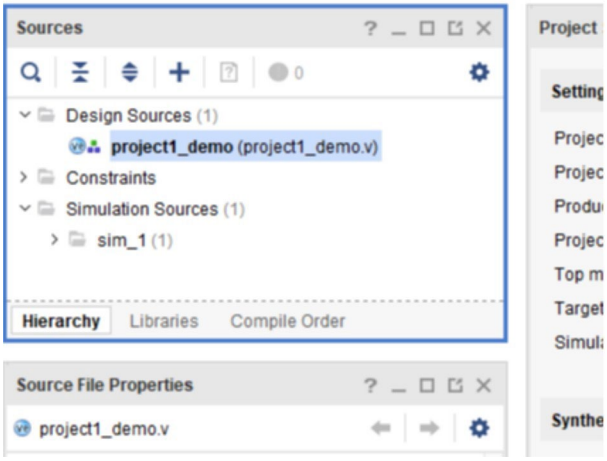


Figure 4. project1_demo appears in design sources

```
`timescale 1ns / 1ps

    module project1_demo(
        input mclk,
        input [15:0] sw,
        output [15:0] led,
        output [7:0] D0_seg, D1_seg,
        output [3:0] D0_a, D1_a
    );

    reg [28:0] cnt29 = 29'd0;
    reg [15:0] led_reg;
    reg [7:0] seg_reg;

    always @(posedge mclk) cnt29 <= cnt29 + 1;

    always @(posedge cnt29[24])
    begin
        if (cnt29[28:25] == 4'b0) begin
            led_reg <= 16'hFFFE;
            seg_reg <= 8'hFE;
        end
        else begin
            seg_reg <= {seg_reg[6:0], seg_reg[7]};
            led_reg <= {led_reg[14:0], led_reg[15]};
        end
    end

    assign led = sw | led_reg;
    assign D0_a = 4'h0;
    assign D1_a = 4'h0;
    assign D0_seg = seg_reg;
    assign D1_seg = seg_reg;

    endmodule
```

Design Constraints

Verilog source files only describe circuit behavior. You must also provide a constraints file to map your design into the physical chip and board you are working with.

To create a constraint file, expand the Constraints heading in the Sources panel, right-click on constrs_1, and select Add Sources (or equivalently, click on "Add Sources" in the Flow Navigator window under Project Manager).

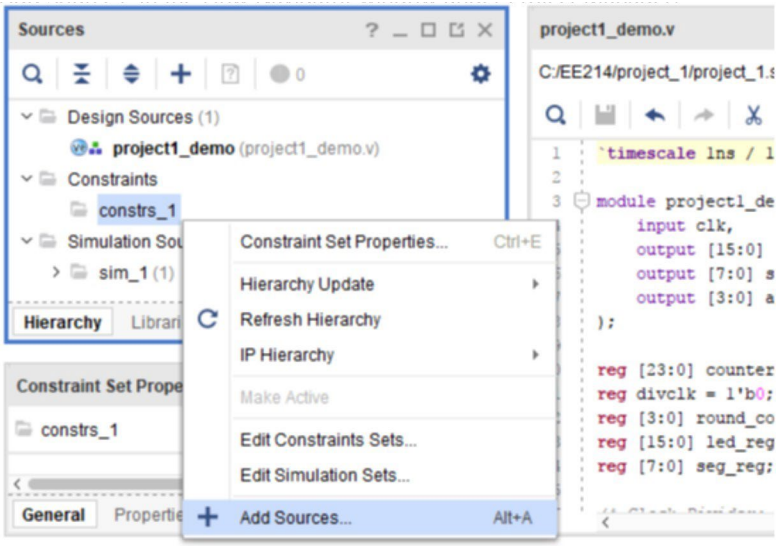


Figure 5. Add Source to Design Constraints

An Add Sources dialog will appear as shown. Select Add or Create Constraints and click Next to cause the "Add or Create Constraints" dialog box to appear.

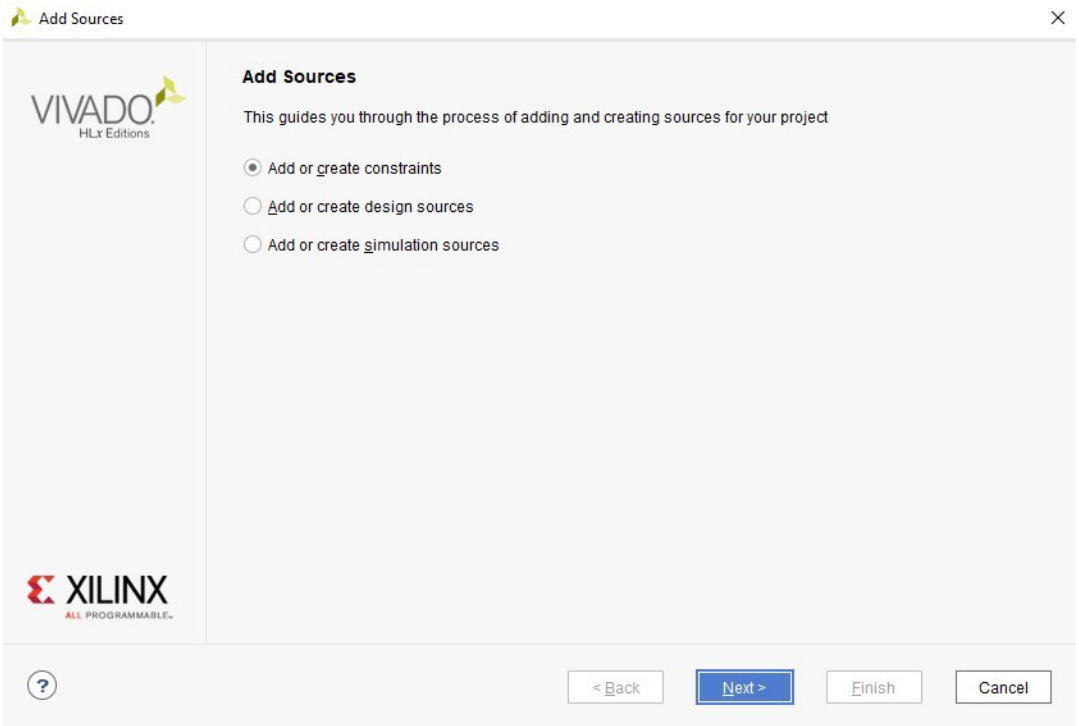


Figure 6. Add or create design constraints using Add Source Dialog

Click on **Create File**, enter `project1` for the filename and click **OK**. The newly created file will appear in the list as shown. Click **Finish** to move to the next step.

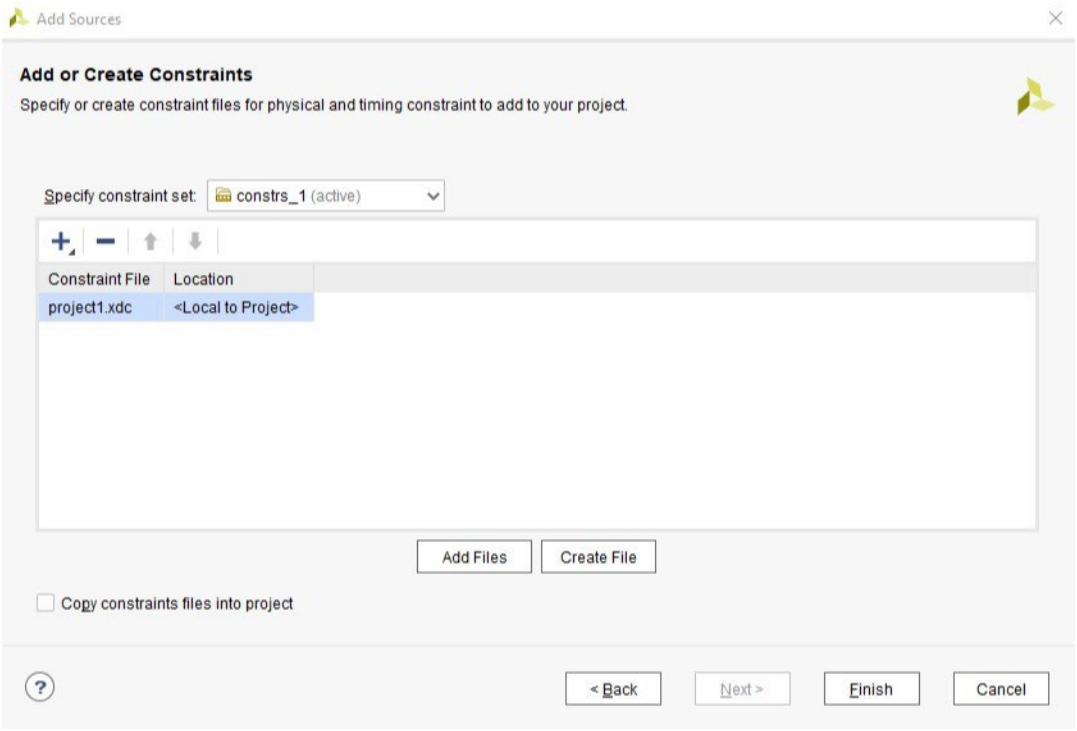


Figure 7. Add or create design constraints using Add Source Dialog

You will now see `project1.xdc` listed underneath `Constraints/constrs_1` folder in the Sources panel. Double click `project1.xdc` to open the file, and replace the contents with the code below.

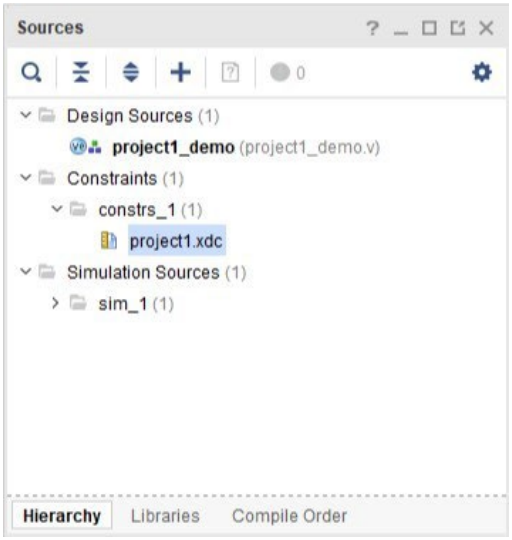


Figure 8. Double Click to Edit project1.xdc

```
# mclk is from the 100 MHz oscillator on Boolean board
set_property -dict {PACKAGE_PIN F14 IOSTANDARD LVCMOS33} [get_ports mclk]
# On-board Slide Switches
set_property -dict {PACKAGE_PIN V2 IOSTANDARD LVCMOS33} [get_ports {sw[0]}]
set_property -dict {PACKAGE_PIN U2 IOSTANDARD LVCMOS33} [get_ports {sw[1]}]
set_property -dict {PACKAGE_PIN U1 IOSTANDARD LVCMOS33} [get_ports {sw[2]}]
set_property -dict {PACKAGE_PIN T2 IOSTANDARD LVCMOS33} [get_ports {sw[3]}]
set_property -dict {PACKAGE_PIN T1 IOSTANDARD LVCMOS33} [get_ports {sw[4]}]
set_property -dict {PACKAGE_PIN R2 IOSTANDARD LVCMOS33} [get_ports {sw[5]}]
set_property -dict {PACKAGE_PIN R1 IOSTANDARD LVCMOS33} [get_ports {sw[6]}]
set_property -dict {PACKAGE_PIN P2 IOSTANDARD LVCMOS33} [get_ports {sw[7]}]
set_property -dict {PACKAGE_PIN P1 IOSTANDARD LVCMOS33} [get_ports {sw[8]}]
set_property -dict {PACKAGE_PIN N2 IOSTANDARD LVCMOS33} [get_ports {sw[9]}]
set_property -dict {PACKAGE_PIN N1 IOSTANDARD LVCMOS33} [get_ports {sw[10]}]
set_property -dict {PACKAGE_PIN M2 IOSTANDARD LVCMOS33} [get_ports {sw[11]}]
set_property -dict {PACKAGE_PIN M1 IOSTANDARD LVCMOS33} [get_ports {sw[12]}]
set_property -dict {PACKAGE_PIN L1 IOSTANDARD LVCMOS33} [get_ports {sw[13]}]
set_property -dict {PACKAGE_PIN K2 IOSTANDARD LVCMOS33} [get_ports {sw[14]}]
set_property -dict {PACKAGE_PIN K1 IOSTANDARD LVCMOS33} [get_ports {sw[15]}]

# On-board LEDs
set_property -dict {PACKAGE_PIN G1 IOSTANDARD LVCMOS33} [get_ports {led[0]}]
set_property -dict {PACKAGE_PIN G2 IOSTANDARD LVCMOS33} [get_ports {led[1]}]
set_property -dict {PACKAGE_PIN F1 IOSTANDARD LVCMOS33} [get_ports {led[2]}]
set_property -dict {PACKAGE_PIN F2 IOSTANDARD LVCMOS33} [get_ports {led[3]}]
set_property -dict {PACKAGE_PIN E1 IOSTANDARD LVCMOS33} [get_ports {led[4]}]
set_property -dict {PACKAGE_PIN E2 IOSTANDARD LVCMOS33} [get_ports {led[5]}]
set_property -dict {PACKAGE_PIN E3 IOSTANDARD LVCMOS33} [get_ports {led[6]}]
set_property -dict {PACKAGE_PIN E5 IOSTANDARD LVCMOS33} [get_ports {led[7]}]
set_property -dict {PACKAGE_PIN E6 IOSTANDARD LVCMOS33} [get_ports {led[8]}]
set_property -dict {PACKAGE_PIN C3 IOSTANDARD LVCMOS33} [get_ports {led[9]}]
set_property -dict {PACKAGE_PIN B2 IOSTANDARD LVCMOS33} [get_ports {led[10]}]
set_property -dict {PACKAGE_PIN A2 IOSTANDARD LVCMOS33} [get_ports {led[11]}]
set_property -dict {PACKAGE_PIN B3 IOSTANDARD LVCMOS33} [get_ports {led[12]}]
set_property -dict {PACKAGE_PIN A3 IOSTANDARD LVCMOS33} [get_ports {led[13]}]
set_property -dict {PACKAGE_PIN B4 IOSTANDARD LVCMOS33} [get_ports {led[14]}]
set_property -dict {PACKAGE_PIN A4 IOSTANDARD LVCMOS33} [get_ports {led[15]}]

# On-board Buttons
set_property -dict {PACKAGE_PIN J2 IOSTANDARD LVCMOS33} [get_ports {btn[0]}]
set_property -dict {PACKAGE_PIN J5 IOSTANDARD LVCMOS33} [get_ports {btn[1]}]
set_property -dict {PACKAGE_PIN H2 IOSTANDARD LVCMOS33} [get_ports {btn[2]}]
set_property -dict {PACKAGE_PIN J1 IOSTANDARD LVCMOS33} [get_ports {btn[3]}]

# On-board color LEDs
set_property -dict {PACKAGE_PIN V6 IOSTANDARD LVCMOS33} [get_ports {RGB0[0]}];   # RBG0_R
set_property -dict {PACKAGE_PIN V4 IOSTANDARD LVCMOS33} [get_ports {RGB0[1]}];   # RBG0_G
set_property -dict {PACKAGE_PIN U6 IOSTANDARD LVCMOS33} [get_ports {RGB0[2]}];   # RBG0_B
set_property -dict {PACKAGE_PIN U3 IOSTANDARD LVCMOS33} [get_ports {RGB1[0]}];   # RBG1_R
set_property -dict {PACKAGE_PIN V3 IOSTANDARD LVCMOS33} [get_ports {RGB1[1]}];   # RBG1_G
set_property -dict {PACKAGE_PIN V5 IOSTANDARD LVCMOS33} [get_ports {RGB1[2]}];   # RBG1_B

# On-board 7-Segment display 0
set_property -dict {PACKAGE_PIN D5 IOSTANDARD LVCMOS33} [get_ports {D0_a[0]}]
set_property -dict {PACKAGE_PIN C4 IOSTANDARD LVCMOS33} [get_ports {D0_a[1]}]
set_property -dict {PACKAGE_PIN C7 IOSTANDARD LVCMOS33} [get_ports {D0_a[2]}]
set_property -dict {PACKAGE_PIN A8 IOSTANDARD LVCMOS33} [get_ports {D0_a[3]}]
set_property -dict {PACKAGE_PIN D7 IOSTANDARD LVCMOS33} [get_ports {D0_seg[0]}]
set_property -dict {PACKAGE_PIN C5 IOSTANDARD LVCMOS33} [get_ports {D0_seg[1]}]
set_property -dict {PACKAGE_PIN A5 IOSTANDARD LVCMOS33} [get_ports {D0_seg[2]}]
```

```
set_property -dict {PACKAGE_PIN B7 IOSTANDARD LVCMOS33} [get_ports {D0_seg[3]}]
set_property -dict {PACKAGE_PIN A7 IOSTANDARD LVCMOS33} [get_ports {D0_seg[4]}]
set_property -dict {PACKAGE_PIN D6 IOSTANDARD LVCMOS33} [get_ports {D0_seg[5]}]
set_property -dict {PACKAGE_PIN B5 IOSTANDARD LVCMOS33} [get_ports {D0_seg[6]}]
set_property -dict {PACKAGE_PIN A6 IOSTANDARD LVCMOS33} [get_ports {D0_seg[7]}]

# On-board 7-Segment display 1
set_property -dict {PACKAGE_PIN H3 IOSTANDARD LVCMOS33} [get_ports {D1_a[0]}]
set_property -dict {PACKAGE_PIN J4 IOSTANDARD LVCMOS33} [get_ports {D1_a[1]}]
set_property -dict {PACKAGE_PIN F3 IOSTANDARD LVCMOS33} [get_ports {D1_a[2]}]
set_property -dict {PACKAGE_PIN E4 IOSTANDARD LVCMOS33} [get_ports {D1_a[3]}]
set_property -dict {PACKAGE_PIN F4 IOSTANDARD LVCMOS33} [get_ports {D1_seg[0]}]
set_property -dict {PACKAGE_PIN J3 IOSTANDARD LVCMOS33} [get_ports {D1_seg[1]}]
set_property -dict {PACKAGE_PIN D2 IOSTANDARD LVCMOS33} [get_ports {D1_seg[2]}]
set_property -dict {PACKAGE_PIN C2 IOSTANDARD LVCMOS33} [get_ports {D1_seg[3]}]
set_property -dict {PACKAGE_PIN B1 IOSTANDARD LVCMOS33} [get_ports {D1_seg[4]}]
set_property -dict {PACKAGE_PIN H4 IOSTANDARD LVCMOS33} [get_ports {D1_seg[5]}]
set_property -dict {PACKAGE_PIN D1 IOSTANDARD LVCMOS33} [get_ports {D1_seg[6]}]
set_property -dict {PACKAGE_PIN C1 IOSTANDARD LVCMOS33} [get_ports {D1_seg[7]}]
```

Step 3: Synthesize, Implement, and Generate Bitstream

Synthesis

After your Verilog and constraint files are complete, you can Synthesize the design project. In the synthesis process, Verilog code is translated into a "netlist" that defines all the required circuit components needed by the design (these components are the programmable parts of the targeted logic device - more on that later). You can start the Synthesize process by clicking on Run Synthesis button in the Flow Navigator panel as shown.

When synthesis is running, you can select the log panel located at the bottom of Project Manager to see a log of the currently running processes. Any errors that occur during the synthesis process will be described in the log.

Note: At the end of the synthesis process, a dialog box opens to ask what you want to do next. This is just a convenience - it shows the same choices that are available in the Flow Navigator window. You can simply close the box, or select "Implement Design" which is the next step (see below).

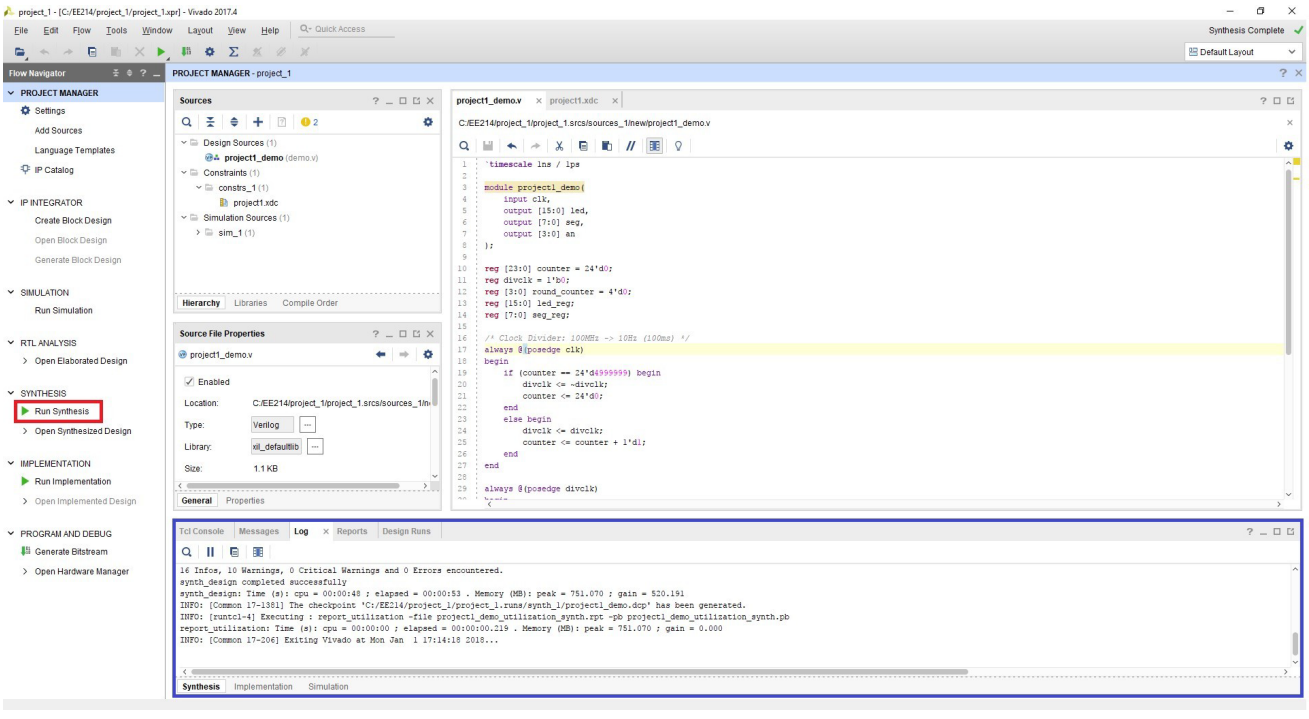


Figure 1. Start Synthesis process and monitor the systhesis log

Implementation

After the design is synthesized, you must run the Implementation process. The implementation process maps the synthesized design onto the Xilinx chip targeted by the design. Click the Run Implementation button in the Flow Navigator panel as shown.

When the implementation process is running, the log panel at the bottom of Project Manager will show details about any errors that occurr.

Note: At the end of the implementation process, a dialog box opens to ask what you want to do next. This is just a convenience - it shows the same choices that are available in the Flow Navigator window. You can simply close the box, or select "Generate Bitstream" which is the next step (see below).

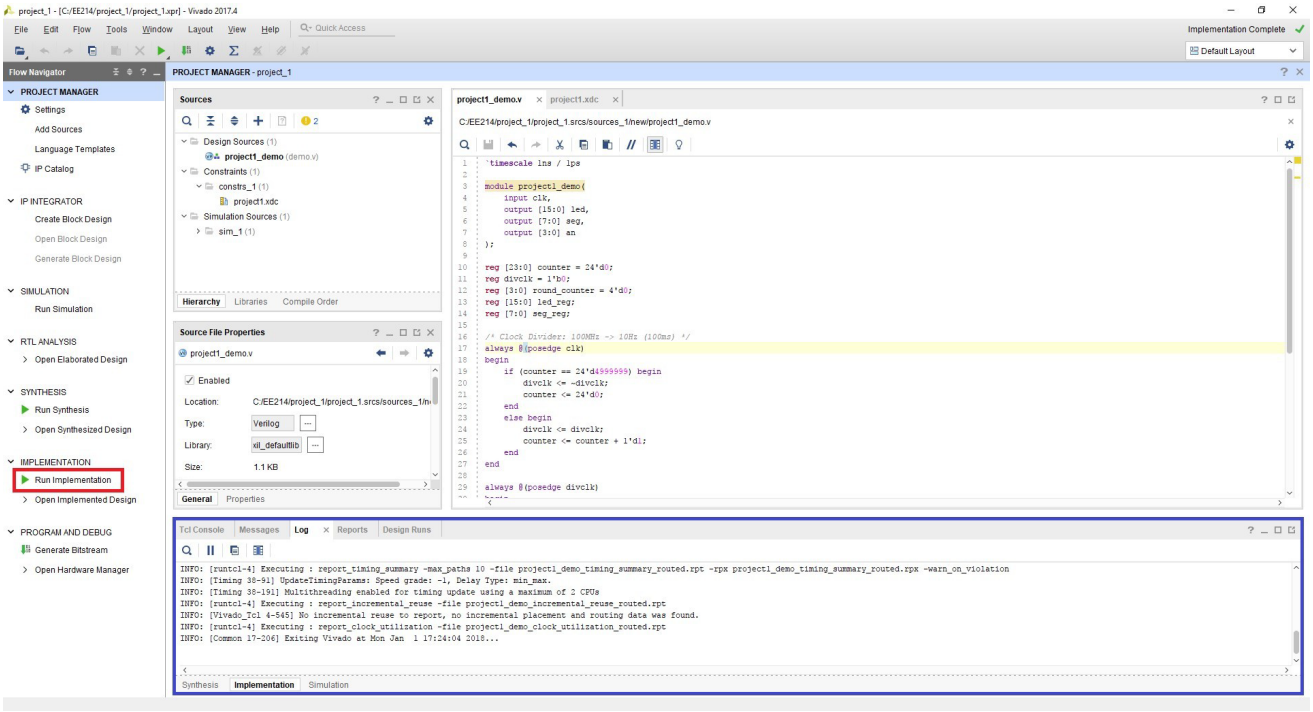


Figure 2. Start Implementation process and monitor the implementation log

Generate Bitstream

After the design is successfully implemented, you can create a .bit file by clicking on the Generate Bitstream process located in the Flow Navigator panel as shown. The process translates the implemented design into a bitstream which can be directly programmed into your board's device.

Note: At the end of the Bitstream generation process, a dialog box opens to ask what you want to do next. This is just a convenience - it shows the same choices that are available in the Flow Navigator window. You can simply close the box, or select "Open Hardware Manager" which is the next step (see below).

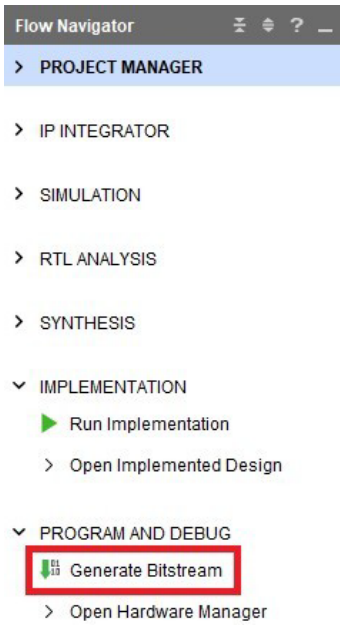


Figure 3. Generate Bitstream

Step 4: Download Bitstream

Open Hardware Manager

After the bitstream is successfully generated, you can program your board using the Hardware Manager. Click Open Hardware Manager located at the bottom of Flow Navigator panel, as highlighted in red in the Figure.

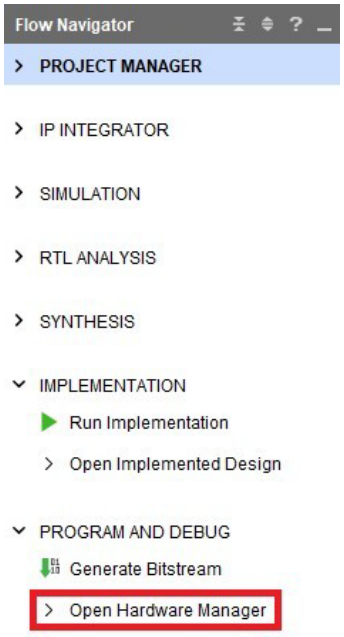


Figure 9. Open Hardware Manager

Connecting Your Board via USB

Connect your Boolean board to your Computer with a micro-USB cable. Make sure you connect the micro-USB cable to the port labeled "**PROG UART**". Turn on your board by moving the switch in the top-left corner to the on position. You'll see a Green LED by the switch when it powers on. If your board doesn't power on, check that the blue jumper by the port labeled "**EXTP**" is set to "**USB**".

Connecting your Board to Vivado

Click on the Open target link underneath Hardware Manager. Select Auto Connect to automatically identify your board.

If you're having trouble connecting in Linux you may need to install cable drivers. Follow the tutorial:

TUTORIAL: INSTALLING LINUX CABLE DRIVERS

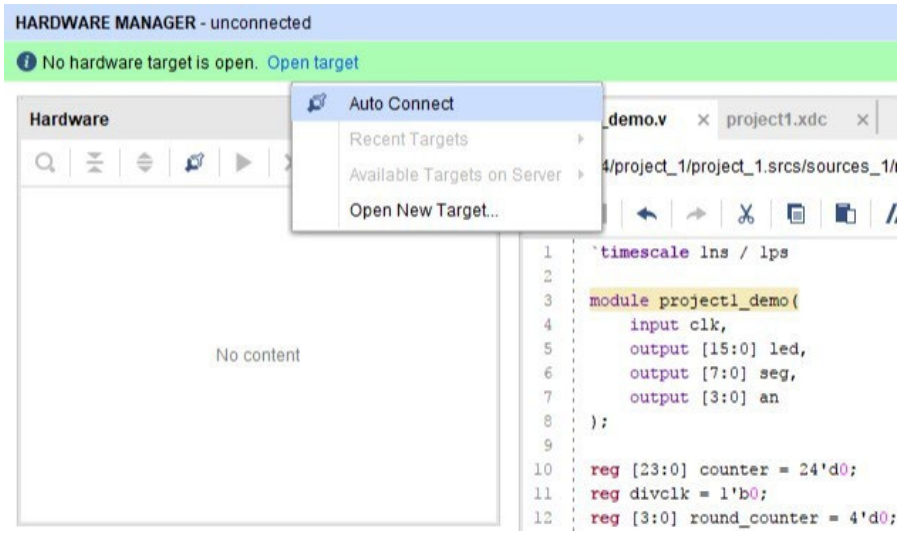


Figure 10. Auto Connect Target

Verify that Your Board is Identified

If Vivado successfully detects your board, the **Hardware** panel (located at the top left corner of **Hardware Manager**) will show the board's logic device part number (For the Boolean board this will be xc7s50).

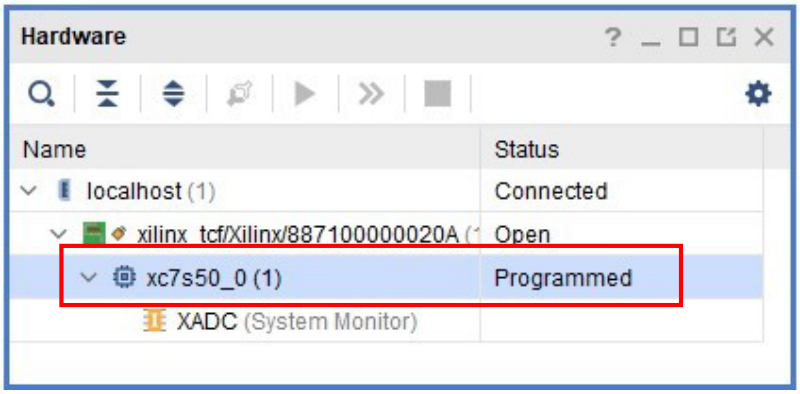


Figure 11. Vivado successfully connects to the Spartan 7 device

Download Bitstream

Select the device you want to program, right click and select **Program Device** . A **Program Device** pop-up dialog window will appear, with the generated bit file selected in the text box. Click on **Program** to download the bitstream to your board.

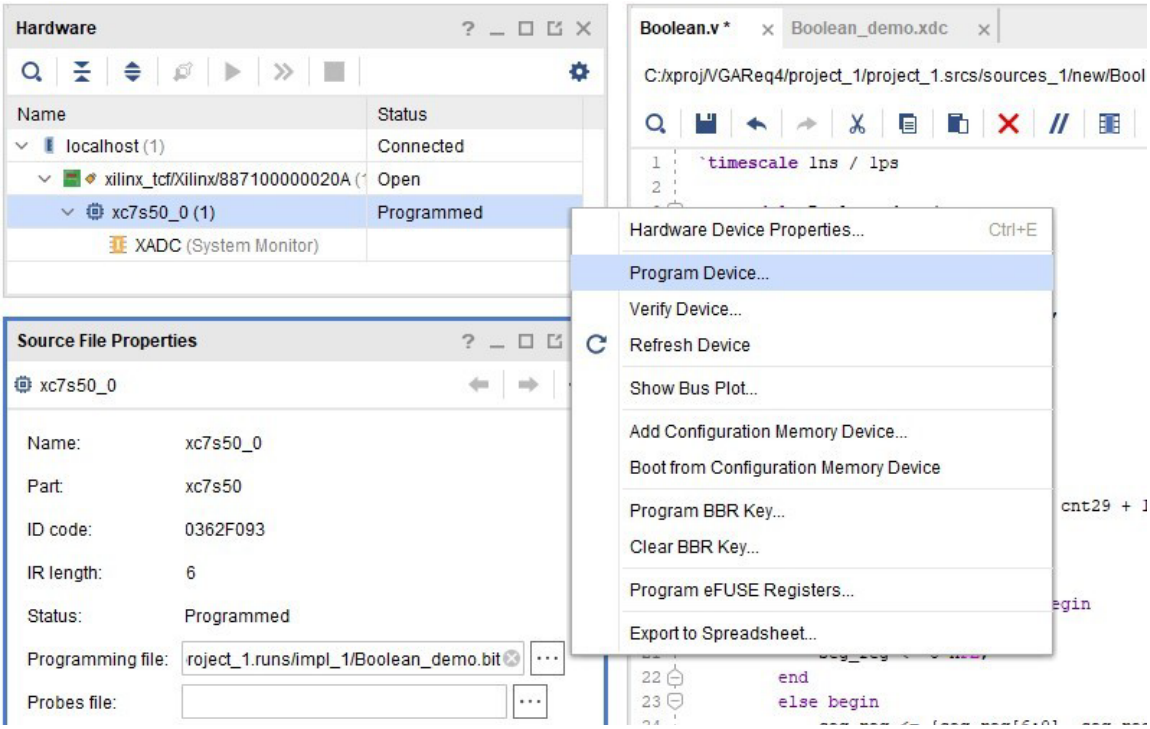


Figure 12. Program Device

Controlling LEDs using Slide Switches

A second experience using Vivado with the Boolean board

This second tutorial demonstrates how to describe a basic digital circuit using Verilog, and in particular, how to connect circuit inputs and outputs to physical devices on the Boolean board. In this tutorial, slide switch inputs are connected through the FPGA to the LED outputs, without any logic in between. This simple exercise is intended to demonstrate how input and output devices are connected to your circuits – in the next exercise, you will add logic circuits between the inputs and outputs.
Digital circuits receive information from the "outside world" in the form of digital signals. Recall digital signals are voltage signals that are constrained to be at the circuits "high" voltage (typically 3.3V, also called a '1'), or the circuits "low" voltage (or ground, also called a '0'). Although signals are wires that transport voltage, we tend to think of digital signals as transporting information (a '0' or a '1').

A digital circuit combines and manipulates information transported by input signals ('0' or '1') using various logical constructs like AND'ing or OR'ing, and produces one or more output signals that return information to the "outside world". Within the digital circuit, there are typically several signals that transport information between devices inside the circuit, and that never see the outside world; and there are several signals called **ports** that communicate with the outside world. In the example circuit shown below, the input port signals A, B, and C are connected to pushbuttons, and the output ports are connected to LEDs.

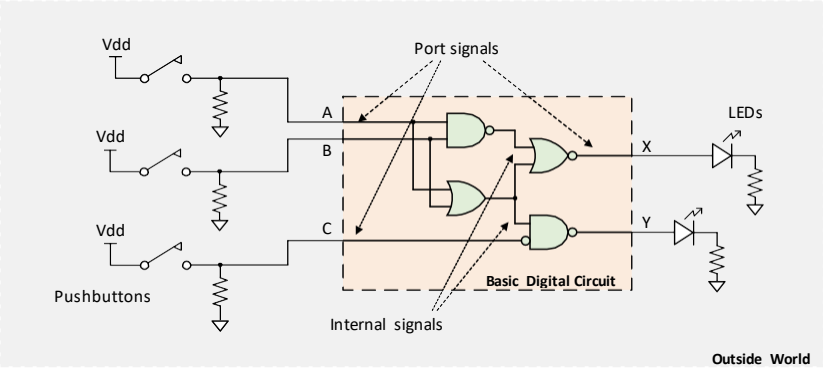


Figure 1. A simple/basic digital circuit

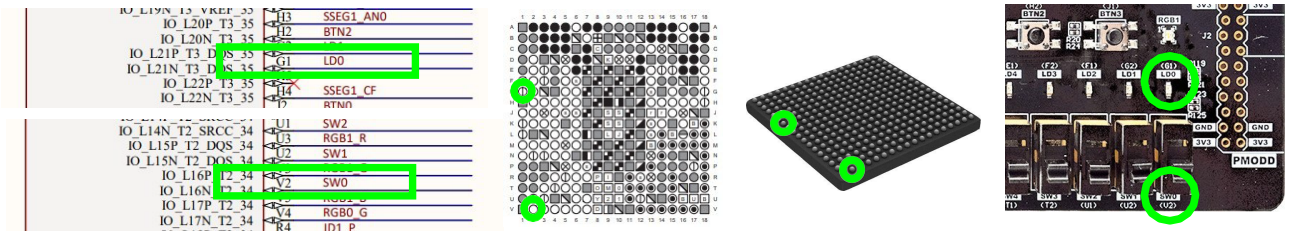
There are many devices and sources that can drive input port signals, including pushbuttons, slide switches, keyboards, touch panels, other electronic devices. Likewise, there are many devices that might receive output port signals, like LEDs, speakers, actuators, other electronic devices, etc. Your Real Digital board contains several input and output devices that can produce and consume I/O port signals. In this tutorial, a slide switch is used as an input device, and an LED is used as an output.

A Simple Circuit

We will build a simple circuit called **led_sw** that passes a signal through the FPGA, from an input slide switch to an output LED, without using any other components. The input port is called "sw", and the output port is called "led". Only a single **assign** statement is needed in the Verilog code to connect the "sw" input to the "led" output, and this is shown below. But recall the Verilog description is only part of the solution - we must also connect the "logical" circuit names in our design (i.e., sw and led) to the physical pins on the Xilinx device – in this case, to the pins connected to the switch and LED we wish to use.

To connect logical signal names in a Verilog source file to physical pins on the FPGA, Xilinx requires "pin mapping" statements in an **.xdc** file (xdc stands for "Xilinx Design Constraints"). Part of the Boolean board schematic is shown below, together with a view of a FPGA chip and the Boolean board. The green boxes in the schematic show LD0 is connected to FPGA pin G1, and SW0 to pin V2. Those same two pins are shown on the FPGA pinout graphic, and also on a photo of the bottom side of the FPGA. When the FPGA is soldered to the circuit board, small copper wires on the board electrically connect pin G1 to LD0, and pin V2 to SW0. Since LD0 is connected to pin G1, any Verilog signal name in your design tied to pin G1 will drive the LED. Likewise, any Verilog signal name in your design tied to pin V2 will be driven to a 1 or 0 depending on the position of SW0. All that remains is to associate your named Verilog signals with physical FPGA pins, and that's what the .xdc file does. A little later in this tutorial, we'l

associate your named Verilog signals with physical FPGA pins, and that's what the .xdc file does. A little later in this tutorial, we'll show you how to create the required entries in the .xdc file.



Boolean board schematic: Note FPGA pin connections: G1 to LD0 and V2 to SW0

FPGA pinout: Physical pin locations G1 and V2

FPGA Package: Solder balls for G1 and V2

Boolean Board: LD0 connected to G0; SWO to V2

Figure 2. Boolean board FPGA pin connections

Step 1: Create a new project

Create an empty project named `requirement_2` (or something similar) following the procedure introduced in the previous tutorial.

Step 2: Design the circuit in Verilog HDL

Create and add a Verilog source file named "led_sw" to your project. If you can't remember how, refer to the previous tutorial. Note that when defining a new Verilog source file, Vivado opens a "Define Module" dialog box after you choose a filename. This dialog allows you to enter input and output signal names, and then those signals will be automatically added to your new Verilog file. This is entirely optional – you can always skip this step by clicking OK, and then type the signal names into the source file. For now, you can skip this step - in later projects, you may wish to type signal names into this dialog box.

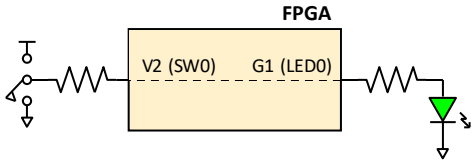


Figure 3. A block diagram for the led_sw circuit

Begin the Verilog file by typing the module statement. In any Verilog source file, the first statement is the "module" statement that provides a name for the module and declares the input and output port signals. The format shown below is generic. The keyword "module" is followed by an alpha-numeric text string that provides the module's name, and a list of port signals and their direction (in parenthesis). The keywords "input" and "output" are followed by the names of input and output signals, in any order. Any number of input and output signals can be declared. In this example, you can name the module "led_sw", and declare one output signal named "led" and one input signal named "sw".

```
module led_sw(
    output led,
    input sw
);
```

The input and output signals "sw" and "led" are recognized as single wires. You can also declare groups of related signals that are organized as "buses". The statements below declare 16 inputs grouped together as a bus named "sw", and 16 output signals grouped together as a bus named "led". This defines sixteen distinct input signals named sw(15), sw(15), etc., and sixteen distinct output signals named led(15), led(14), etc. using one-line declarations. Note you can access individual signals on a bus by using the root name and the signal index in parenthesis (so, for example, sw(4) is the name of the 5th least significant signal on the sw bus).

```
input [15:0] sw
output [15:0] led;
```

Following the module statement, you can write any number of "assign" statements to define combinational logic circuits. The keyword "assign" is used in Verilog any time you want to assign a value to a wire. Simple assignments, like "assign led = sw", can be used to map one signal onto another. More complex assignments can assign the result of logic operations to a wire, like "assign X = A | B & C" – these types of assignment will be used in the next exercise.

```
assign led = sw;
```

Assignment statements can work on individual signals or buses. For example, if "led" and "sw" are both declared as wires, then the statement "assign led = sw" will map a single signal. But if "led" and "sw" are declared as busses, then the same assign statement will assign multiple signals. The assignments happen bit by bit, starting with the right-most (least significant) input bus signal and assigning it to the right-most (least significant) output bus signal, then the next-most significant bit, and so on until all input bus signals are assigned. If there are more input bus signals than output bus signals, the extra input bus signals have no effect. If there are more output bus signals than input bus signals, the extra output signals are left unchanged.

When all required assign statements have been added to the Verilog file, the module description is ended with the "endmodule" key word.

Verilog Source File

For this first exercise, one LED will be driven from one switch. When you have completed the previous steps, your Verilog source file should look like this:

```
`timescale 1ns/1ps
module led_sw(
    output led,
    input sw
);

assign led = sw;

endmodule
```

Step 3: Add Constraints (XDC)

Recall that before a circuit can be implemented on your Xilinx device, all the external pin connections must be defined in a constraints file. Create and add an empty constraints file to your project. If you can't remember how, refer to the previous tutorial. Add two entries into your xdc constraints file – one for the input switch, and one for the output LED as shown below. In a later module, the syntax and meaning of the .xdc entries will be explained more fully.

The .xdc file format to define a pin mapping is shown below - these two lines can be added to your .xdc file.

```
set_property -dict { PACKAGE_PIN V2   IOSTANDARD LVCMOS33 } [get_ports { sw }];
set_property -dict { PACKAGE_PIN G1   IOSTANDARD LVCMOS33 } [get_ports { led }];
```

Note: When mapping signals defined as a bus (for example, sw[15:0]), each individual wire needs a corresponding pin-mapping statement in the XDC file. Individual signals in a bus are identified by the bus root name followed by the signal number in square brackets. So, for example, the signal name of the first switch on the bus is sw[0], and the fourth switch on the bus is sw[3].

Typing in all the individual pin constraints can be time consuming. Instead, you can download and add the corresponding "master" .xdc file for the Boolean board to your project (or you can create an .xdc file and cut-and-paste the attached file contents into it). The master .xdc file defines all pin connections to all external devices (switches, LEDs, etc). Any "extra" signals defined in the master .xdc file but not referenced in your design will be ignored.

BOOLEAN .XDC FILE

Step 4: Generate a .bit file, and test your design

Run synthesis, implementation, and generate bitstream as shown in the previous tutorial. Download your circuit to your board using the Hardware Manager.