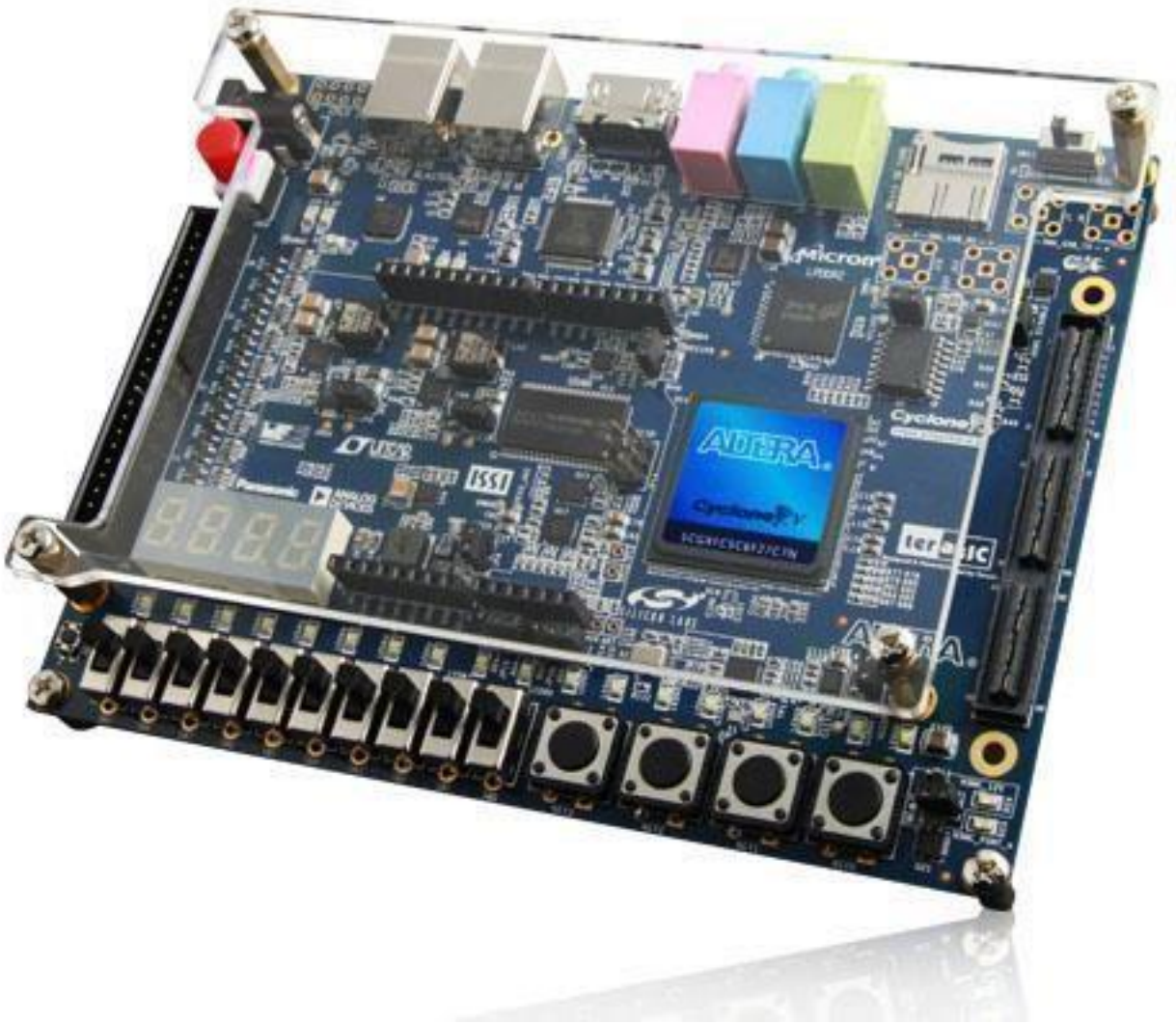
Introduction to Digital Logic and Altera FPGAs Using the Terasic Cyclone V GX Development Board



Contents

Lab 1: Obtaining the Quartus Prime Lite Design Tools

Background

FPGAs are digital semiconductors that infinitely configurable and used to build a huge variety of electronic functions including Data Center accelerators, wireless base stations, and industrial motor controllers to name but a few of their common applications. An FPGA is a special type of semiconductor that can be reconfigured to perform different digital hardware functions so it makes for a great learning platform.

To configure an FPGA you need to describe your digital electronics with either a Hardware Description Language (Verilog or VHDL are most common) or a schematic. Then you need to assign the "pins" of your FPGA based on how the printed circuit board connects the FPGA to various peripheral components on your board (switches, LEDS, memory devices and various connectors). Finally, you will "compile" your design and program the FPGA to perform the function you have specified in the Hardware Description Language or schematic.

The first step in this lab is to download the Altera design tools, called Quartus Prime Lite Edition. For the lower complexity Altera FPGA devices, the Quartus Prime Lite design tools are entirely free. This hardware development kit is $179. Purchase here: https://www.terasic.com.tw/cgibin/page/archive.pl?Language=English&CategoryNo=167&No=830 and there are numerous labs and online trainings you can take based on this full featured kit.

This training class assumes you have prerequisite knowledge of how computers and digital electronics work, but by no means do you need to be a degree electrical engineer to follow along this introductory course.

Installation

Quartus Prime is Altera's design tool suite. It serves a number of functions:

1. Design creation through the use of HDL languages or schematics
2. System creation through the Qsys graphical interface
3. Generation and editing of constraints: timing, pin locations, physical location on die, IO voltage levels
4. Synthesis of high level language into an FPGA netlist ("mapping" in FPGA terminology)
5. FPGA place and route ("fitting" in FPGA terminology)

6. Generation of design image (used to program FPGA, “assembly” in FPGA terminology) 7. Timing Analysis
8. Programming/download of design image into FPGA hardware
9. Debugging by insertion of debug logic (in-chip logic analyzer)
10. Interfaces to 3rd party tools such as simulators
11. Launching of Software Build Tools (Eclipse) for Nios II

To download Quartus Prime Lite, follow these instructions:

1. Visit this site: http://dl.altera.com/?edition=lite 2. Select version 16.0 and your PC’s operating system.
3. For the smallest installation, and quickest download time, enter only the entries shown below in Figure 1.

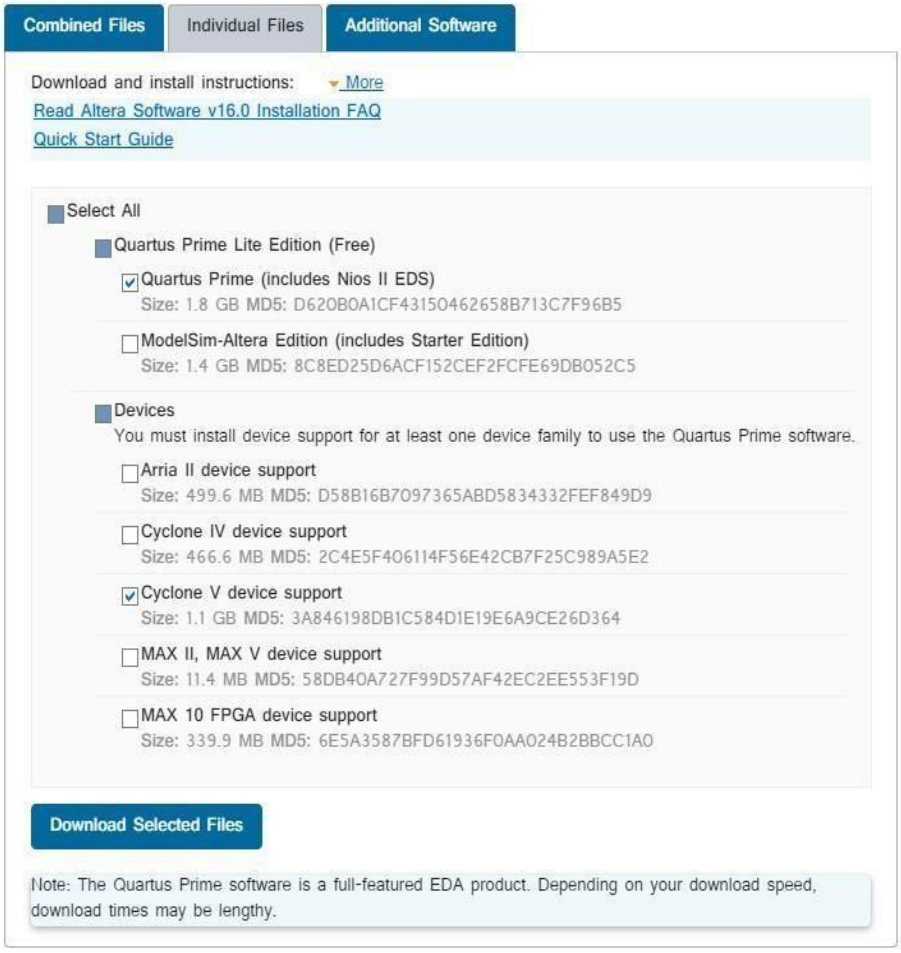


Figure 1: Quartus Prime Lite Minimum Required Files to Download

4. Follow the instructions to download the Design Tools and you will have the Quartus Prime Lite version 16.0 tools up and running on your PC.

Lab 2: New Project Wizard

Summary

This is a short lab that completes the basic project setup. At the end of this lab, you will be able to start a new project using New Project Wizard in Quartus Prime Software. There are other related tutorial links provided for you to learn more about the software.

Lab Instruction

2.0: Navigation of Quartus Prime

Open the tools by double clicking the “Quartus Prime” icon: 

You should now see something similar to the Figure 2 below. The function for each panel is listed below.

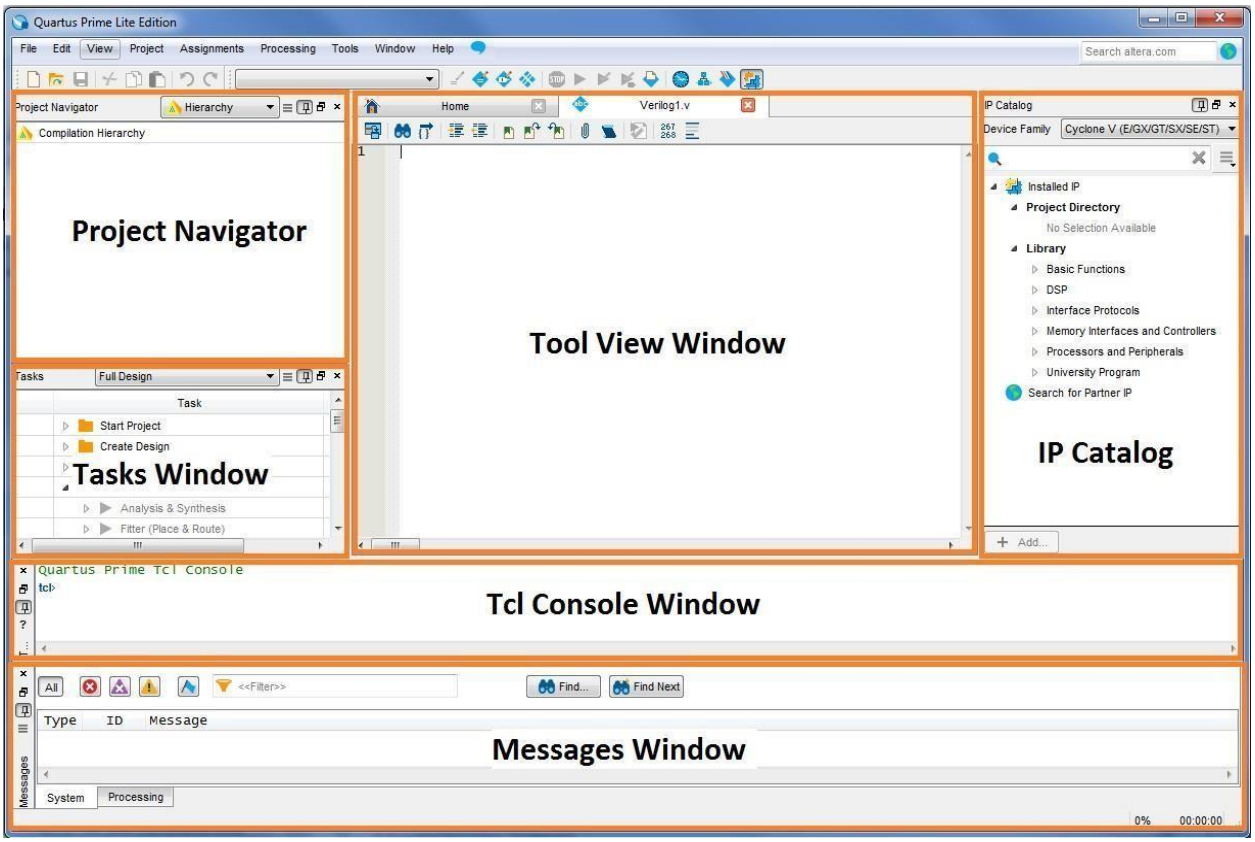


Figure 2: Quartus Prime main window

2.1: Making a new Project with the "New Project Wizard"

In the main toolbar of Quartus, navigate to the "**File**" drop down menu and "**New Project Wizard**".

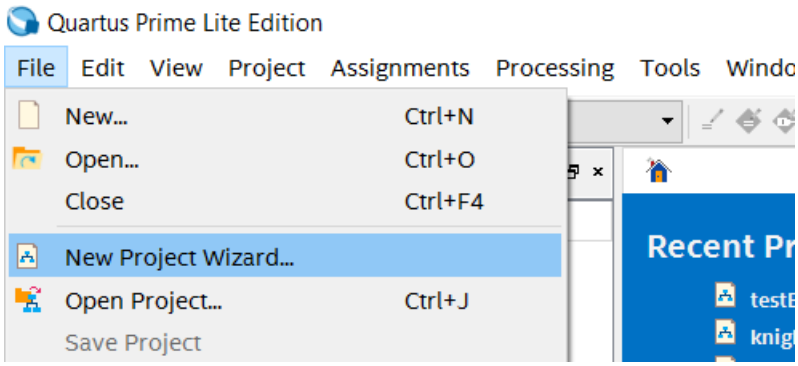


Figure 3: Quartus prime file menu

Pane 1: Basics. Fill in with a directory of your choice. It is recommended to be a personal directory, and not a directory under the Quartus installation which is the default.

Call the project Lab and the top level entity "**Switch_to_LED**" See Figure 3 below for a completed Pane 1.

Note that the screen shots will have a different directory than what you will use for your project. This is fine.

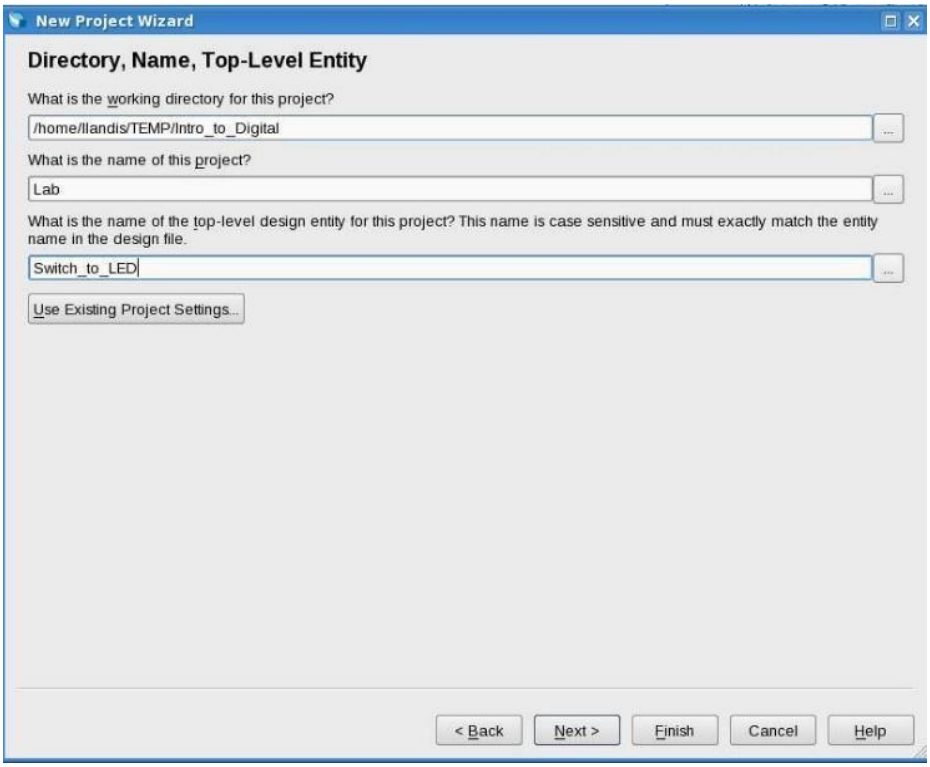


Figure 4: Pane 1 of the New Project Wizard correctly filled out

Pane 2: Project Type. Select "**Empty project**"
Pane 3: Source Files. Click Next as we will add project source files later.
Pane 4: Family, Device, and Board Settings

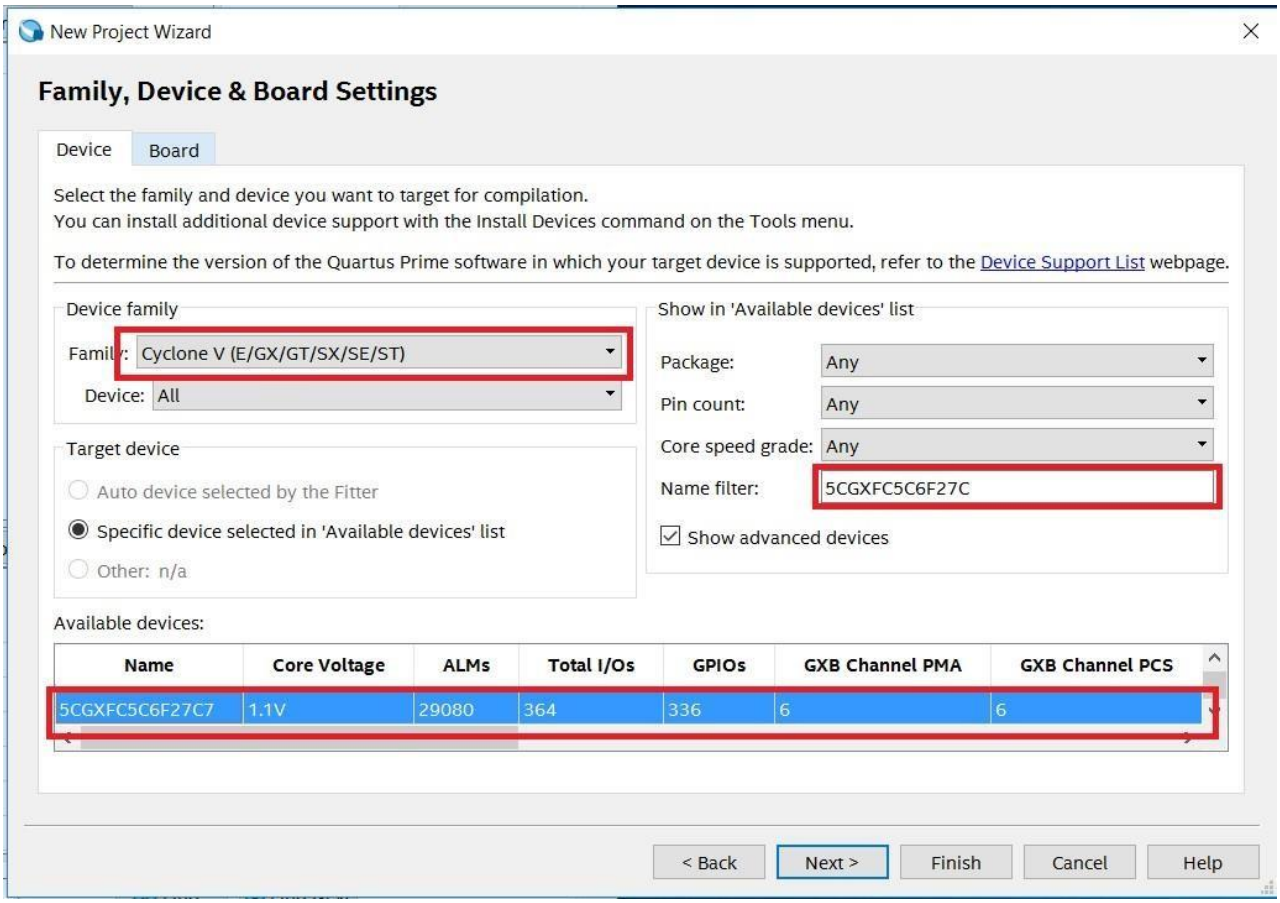


Figure 5: Filled-out family, device, and board settings

First take a look at your development board, and you should notice a part number on the Cyclone V E FPGA Chip. This number is VERY difficult to see. If you are having trouble reading the part number on your FPGA chip, try using your phone flashlight to illuminate the FPGA and better see the part number. In this workshop, the device we are looking for is 5CGXFC5C6F27C7N. You do not need to type in the 'N' character at the end of the part number.

Make sure the tab is set to Device. Type this part number in the Name filter and choose the device in the Available devices panel. Refer to Figure 5 above for a correctly filled out Pane 4.

Pane 5: EDA Tool Settings
Skip this section and click NEXT. This section is only needed if you are using other development software besides Quartus Prime.

Pane 6: Summary

Pane 6 should look similar to the image seen in Figure 6 below.

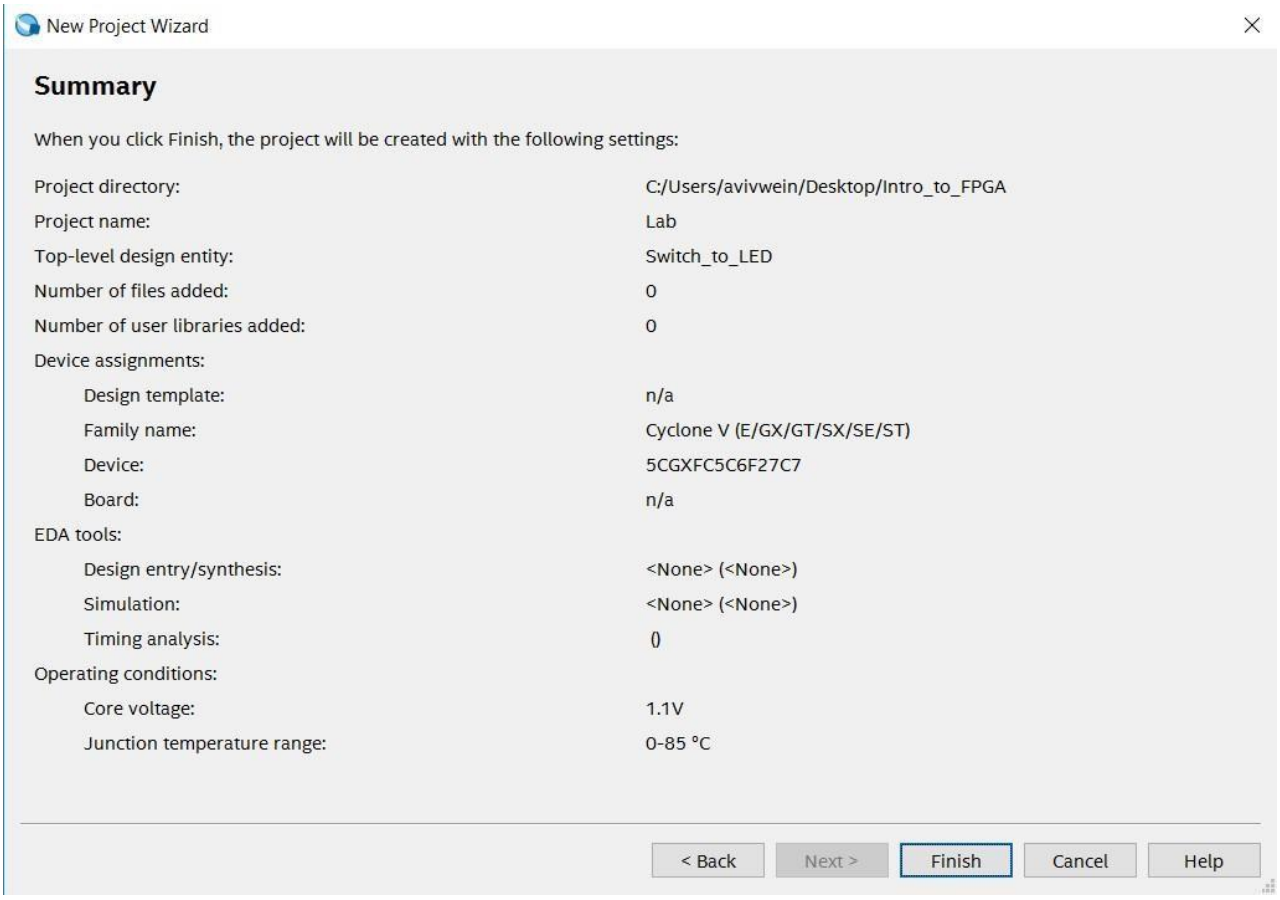


Figure 6: Summary page generated by the New Project Wizard

Notice that you have a project called Lab, and top-level entity called "Switch_to_LED", no files selected (yet) and are using a Cyclone V E FPGA device. With these settings, you are now ready to start designing your FPGA project.

Click "Finish"

If you navigate in Windows Explorer to your project directory, you will see some files and directories created by the New Project Wizard as part of the setup process.

Lab 3: Making Assignments

Overview:

This lab will step you through the process of a simple design from generating your first Verilog file to synthesize and compile. Synthesis converts your Verilog language file to an FPGA specific "netlist" that programs the programmable FPGA lookup tables into your desired function. Compilation figures out the location of the lookup table cells used in the FPGA and generates a programming image that is downloaded to your Altera Development kit. At the end of this lab, you will be able to test the functionality of the example digital electronic circuits by toggling the switches and observing the LEDs for proper circuit operation.

Instructions:

3.1: Creating a new file

Create the Verilog HDL file. Go the "**File**" dropdown menu and select **"New"**. A window, shown in Figure 7, should popup. Click on "**Verilog HDL File**" and then "**Ok**"

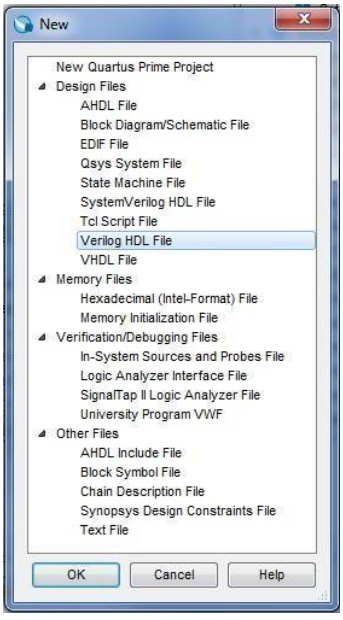


Figure 7: New File Window

3.2: Adding Verilog Code

Create a simple module in your Verilog HDL file by typing in, copy and pasting, or downloading the following code. If there is a problem you can copy the code from the link here. (NOTE click on the date of the most recent revision)

```
module Switch_to_LED(SW, LEDR);  //create module Switch_to_LED

      input [9:0] SW;             // input declarations: 10 switches
      output      [9:0] LEDR;     // output declarations: 10 red LEDs
      assign      LEDR = SW;      // connect switches to LEDs

endmodule
```

Check your syntax carefully! Can you explain what this circuit does?

Next you will run Analysis and Elaboration. Analysis and Elaboration checks the syntax of your Verilog code, resolves references to other modules and maps to FPGA logic. If you see any errors during the Analysis and Elaboration step, carefully review your Verilog code for syntax errors and re-run this step.

To run Analysis and Elaboration, click the play button with a green check mark. Shown in Figure 7 below.

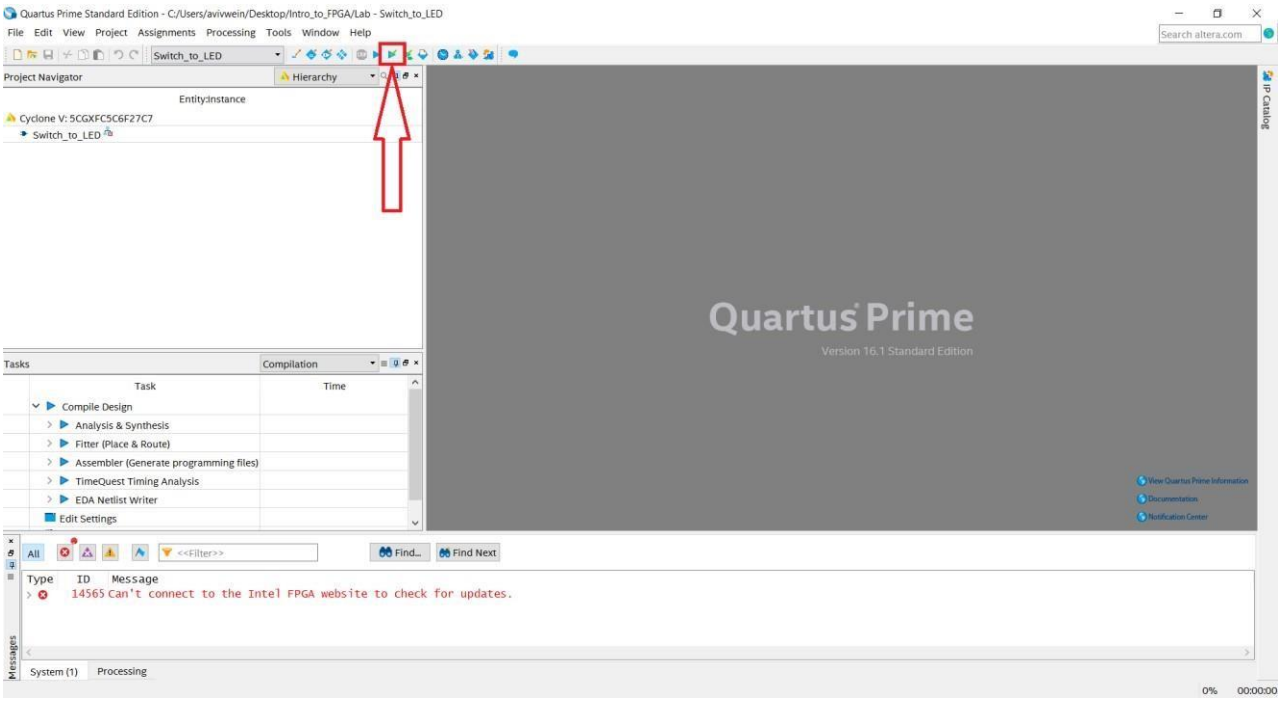


Figure 8: Main Quartus Window Highlighting the Location of the Analysis& Elaboration Button

3.3: Assigning Pins

By default, Quartus Prime does not know how the FPGA pins are connected on the Cyclone V GX development board to the Switches and LEDs used in this circuit. Because our FPGA is already on a PCB we need to tell Quartus what pins to use. The next step will assign the Switches and LED signals in your code to the appropriate pins, using the main toolbar at the top of the Quartus window, navigate to the "**Assignments**" drop down menu in the main Quartus toolbar. Click on **"Pin Planner"** and a window similar to the image in Figure 10 should open.

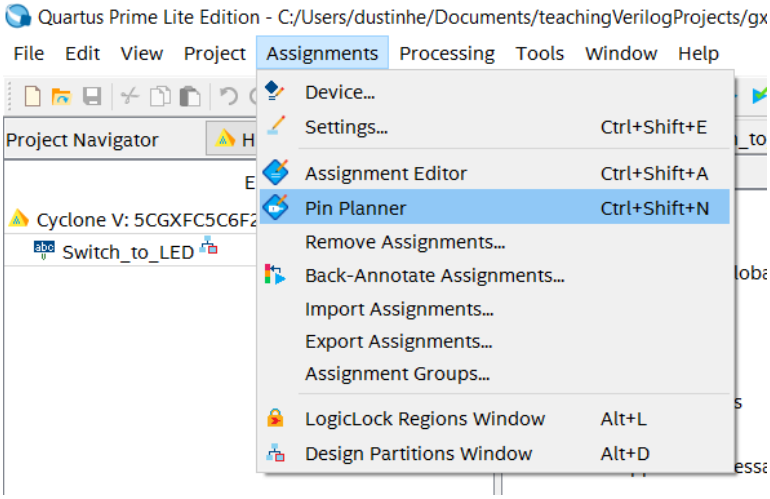


Figure 9: Quartus assignment editor menu

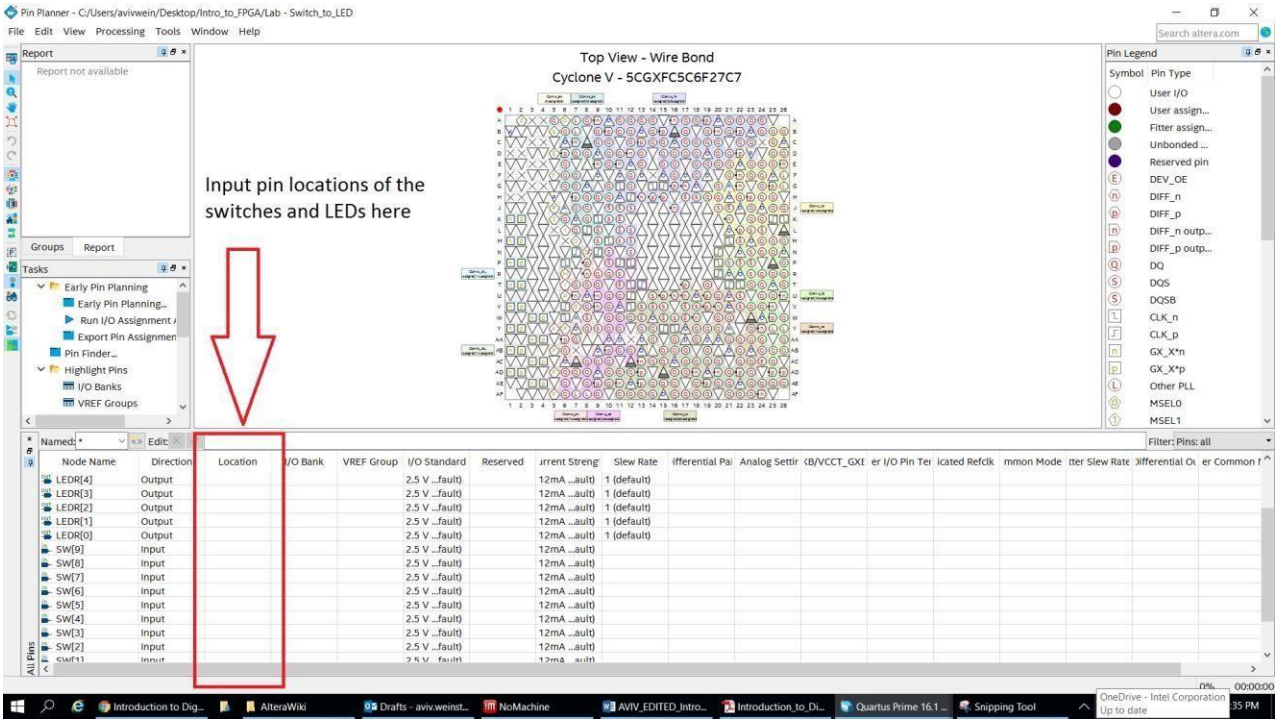


Figure 10: Pin Planner Assignment Window

We can see the I/O pins have not been assigned to any locations yet. To make the right pin assignments for this project, we have to look it up in development board vendor's manual.

In this case, the vendor is Terasic and the board is the Cyclone V GX Board. In most cases you would go to Terasic's Cyclone V GX Development Kit website and download the Cyclone V GX User Manual. For convenience, we include a table, Table 1 on the following page, with the relevant should you choose to skip downloading the manual.

Open the user guide or refer to Table 1, on the following page, and we can find out the pin assignments for the switches and Red LED lights. Assign the first 2 LEDR pins using the instructions on page 14. **(You do not need to assign a clock for the first lab. Clock information is included for other labs.)**

Signal Name	FPGA Pin No.	Description
SW0	PIN_AC9	Slide Switch [0]
SW1	PIN_AE10	Slide Switch [1]
SW2	PIN_AD13	Slide Switch [2]
SW3	PIN_AC8	Slide Switch [3]
SW4	PIN_W11	Slide Switch [4]
SW5	PIN_AB10	Slide Switch [5]
SW6	PIN_V10	Slide Switch [6]
SW7	PIN_AC10	Slide Switch [7]
SW8	PIN_Y11	Slide Switch [8]
SW9	PIN_AE19	Slide Switch [9]
Signal Name	**FPGA Pin No.**	**Description**
LEDR0	PIN_F7	LED [0]
LEDR1	PIN_F6	LED [1]
LEDR2	PIN_G6	LED [2]
LEDR3	PIN_G7	LED [3]
LEDR4	PIN_J8	LED [4]
LEDR5	PIN_J7	LED [5]
LEDR6	PIN_K10	LED [6]
LEDR7	PIN_K8	LED [7]
LEDR8	PIN_H7	LED [8]
LEDR9	PIN_J10	LED [9]
Signal Name	**FPGA Pin No.**	**Description**
CLOCK_50	PIN_R20	50MHz Clock Input (Bank 5B)
CLOCK2_50	PIN_N20	50MHz Clock Input (Bank 6A)
CLOCK3_50	PIN_H12	50MHz Clock Input (Bank 7A)
CLOCK4_50	PIN_M10	50MHz Clock Input (Bank 8A)

Table 1: Pin assignments for the dev. kit

Match the "**Signal Name**" (1st column) with the "**FPGA Pin No**." (2nd column) section in the manual or table above. Assign LEDR[9] to PIN_J10 using this method.
Note that the signal names in your code and names in the manual don't have to match, as long as you connect to the proper pin location, your design will be connected properly.

An alternate method is to left click on the Node Name in the Pin Planner and drag the pin on top of the ball grid map location that is assigned in the table. Release the pin on the proper location. Hit the escape key and move to the next pin. Assign LEDR[8] to PIN_H7 using this method.

When you finish, you can just close the window – the Planner does not have a Save button, but it will save anyways. **This SWITCH to LED lab does not require the CLOCK signals so you can ignore these for the time being.**

Last, we will assign the remaining pins using a TCL script. Click the new file button or press Ctrl+N. Select "Tcl Script File" and click ok.

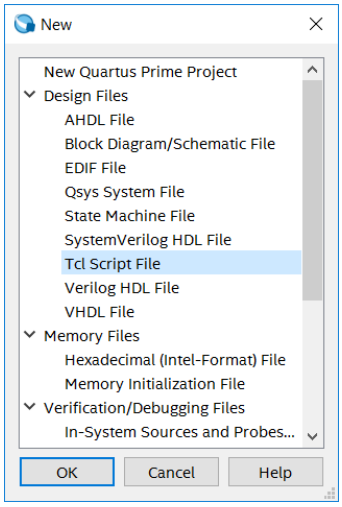


Figure 11: Quartus new file menu

Copy and paste the code from the next page or from the link here into the new file.

```
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDG[0]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDG[1]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDG[2]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDG[3]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDG[4]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDG[5]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDG[6]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDG[7]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDR[0]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDR[1]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDR[2]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDR[3]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDR[4]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDR[5]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDR[6]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDR[7]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDR[8]
set_instance_assignment -name IO_STANDARD "2.5 V" -to LEDR[9]
set_instance_assignment -name IO_STANDARD "1.2 V" -to SW[0]
set_instance_assignment -name IO_STANDARD "1.2 V" -to SW[1]
set_instance_assignment -name IO_STANDARD "1.2 V" -to SW[2]
set_instance_assignment -name IO_STANDARD "1.2 V" -to SW[3]
set_instance_assignment -name IO_STANDARD "1.2 V" -to SW[4]
set_instance_assignment -name IO_STANDARD "1.2 V" -to SW[5]
set_instance_assignment -name IO_STANDARD "1.2 V" -to SW[6]
set_instance_assignment -name IO_STANDARD "1.2 V" -to SW[7]
set_instance_assignment -name IO_STANDARD "1.2 V" -to SW[8]
set_instance_assignment -name IO_STANDARD "1.2 V" -to SW[9]
set_location_assignment PIN_L7 -to LEDG[0]
set_location_assignment PIN_K6 -to LEDG[1]
set_location_assignment PIN_D8 -to LEDG[2]
set_location_assignment PIN_E9 -to LEDG[3]
set_location_assignment PIN_A5 -to LEDG[4]
set_location_assignment PIN_B6 -to LEDG[5]
set_location_assignment PIN_H8 -to LEDG[6]
set_location_assignment PIN_H9 -to LEDG[7]
set_location_assignment PIN_F7 -to LEDR[0]
set_location_assignment PIN_F6 -to LEDR[1]
set_location_assignment PIN_G6 -to LEDR[2]
set_location_assignment PIN_G7 -to LEDR[3]
set_location_assignment PIN_J8 -to LEDR[4]
set_location_assignment PIN_J7 -to LEDR[5]
set_location_assignment PIN_K10 -to LEDR[6]
set_location_assignment PIN_K8 -to LEDR[7]
set_location_assignment PIN_H7 -to LEDR[8]
set_location_assignment PIN_J10 -to LEDR[9]
set_location_assignment PIN_AC9 -to SW[0]
set_location_assignment PIN_AE10 -to SW[1]
set_location_assignment PIN_AD13 -to SW[2]
set_location_assignment PIN_AC8 -to SW[3]
set_location_assignment PIN_W11 -to SW[4]
set_location_assignment PIN_AB10 -to SW[5]
set_location_assignment PIN_V10 -to SW[6]
set_location_assignment PIN_AC10 -to SW[7]
set_location_assignment PIN_Y11 -to SW[8]
set_location_assignment PIN_AE19 -to SW[9]
```

Press the save button or hit Ctrl+S and name the file pins.tcl. Open the TCL menu by clicking tools>Tcl Scripts…

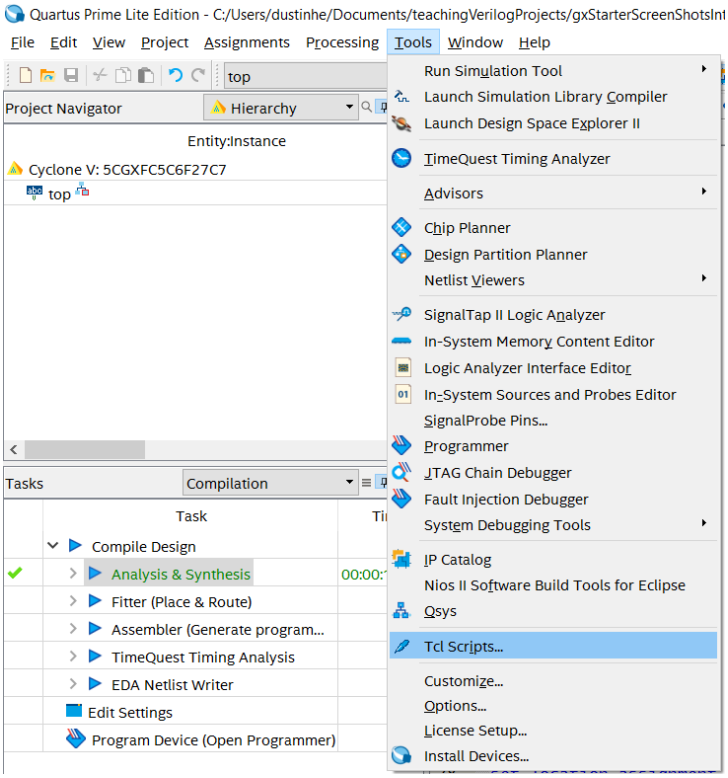


Figure 12: Navigate to the Quartus TCL Scripts menu

A menu like the one in figure 13 should open. Click on pins.tcl to highlight it then click RUN.

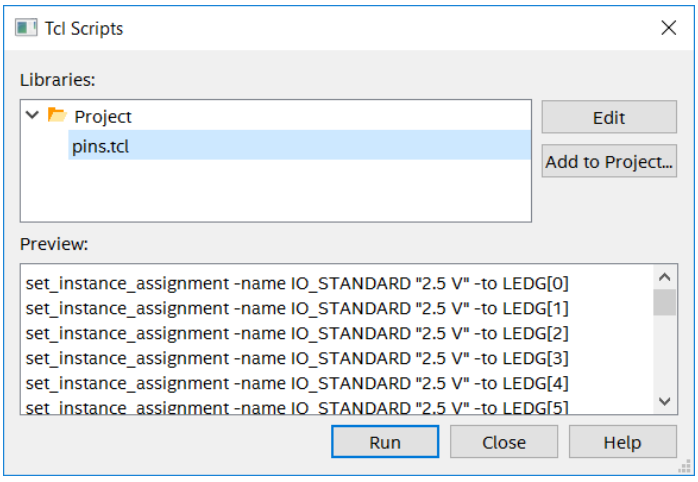


Figure 13: Quartus TCL scripts menu

Now the remaining pins have been assigned for you by the script

3.4: Compiling your Code

Click  ,located at the top of the main Quartus window, to start the full compilation of your code. You can also go to: "**Processing" → "Start" → "Compilation".**

After roughly 1 to 2 minutes, (depending on your machine type and amount of RAM), the compilation should complete and there should be 0 errors (you can ignore warnings).

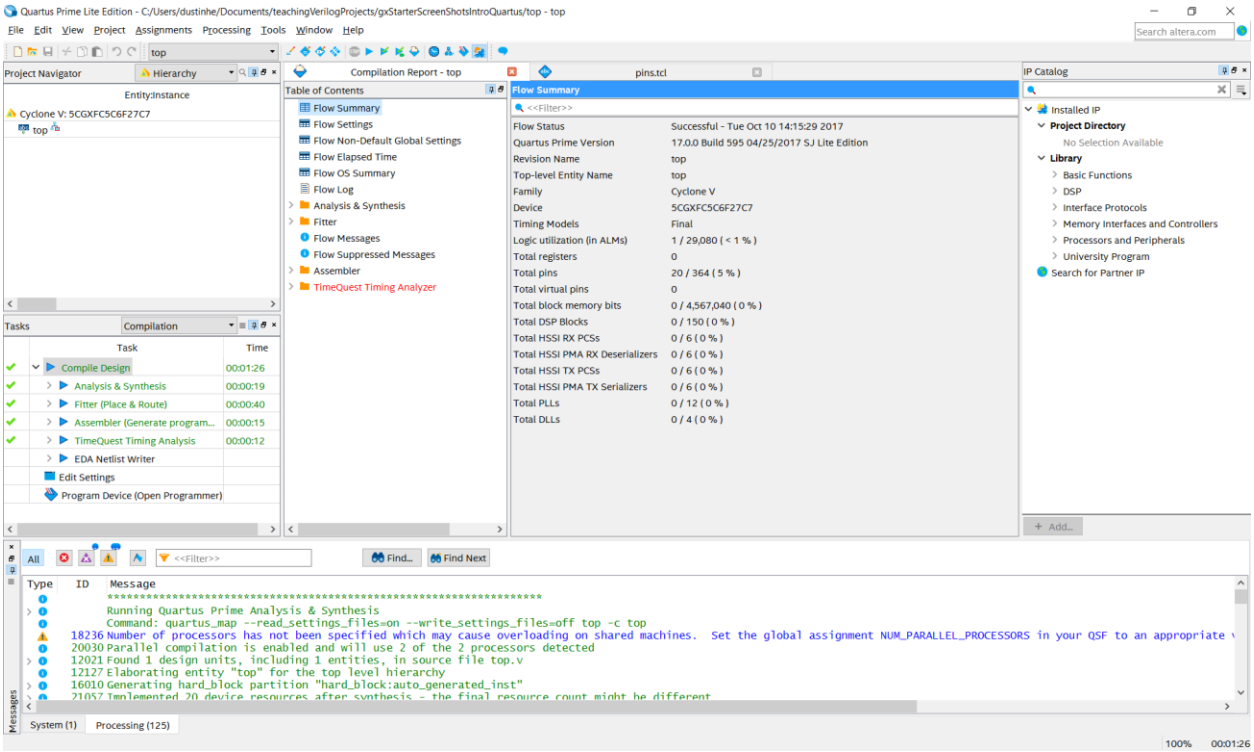


Figure 14: Quartus Window Showing Successful Compilation

3.5: Installing and Using the USB Blaster to Download Your Design to the FPGA

To download your completed FPGA design into the device, connect the USB Blaster cable between your PC USB port and the USB Blaster port on your development kit. Don't forget to also plug the kit into power using the wall adapter. Upon plugging in your device, you should see flashing LEDs and 7-segment displays counting in hexadecimal, or other lit up LEDs and 7-segments depending on previous projects that have been downloaded to the development kit.

To use the USB Blaster to program your device, you need to install the USB Blaster driver. Follow the following steps to install the USB Blaster driver.

To begin, open your device manager, by hitting the windows key 

, typing "**device manager**", and clicking on the device manager tile that appears. Navigate to the "other devices"

section of the device manager and expand the section. Figure 11 below depicts what the Device Manager will look like when the USB Blaster Drivers are not installed.

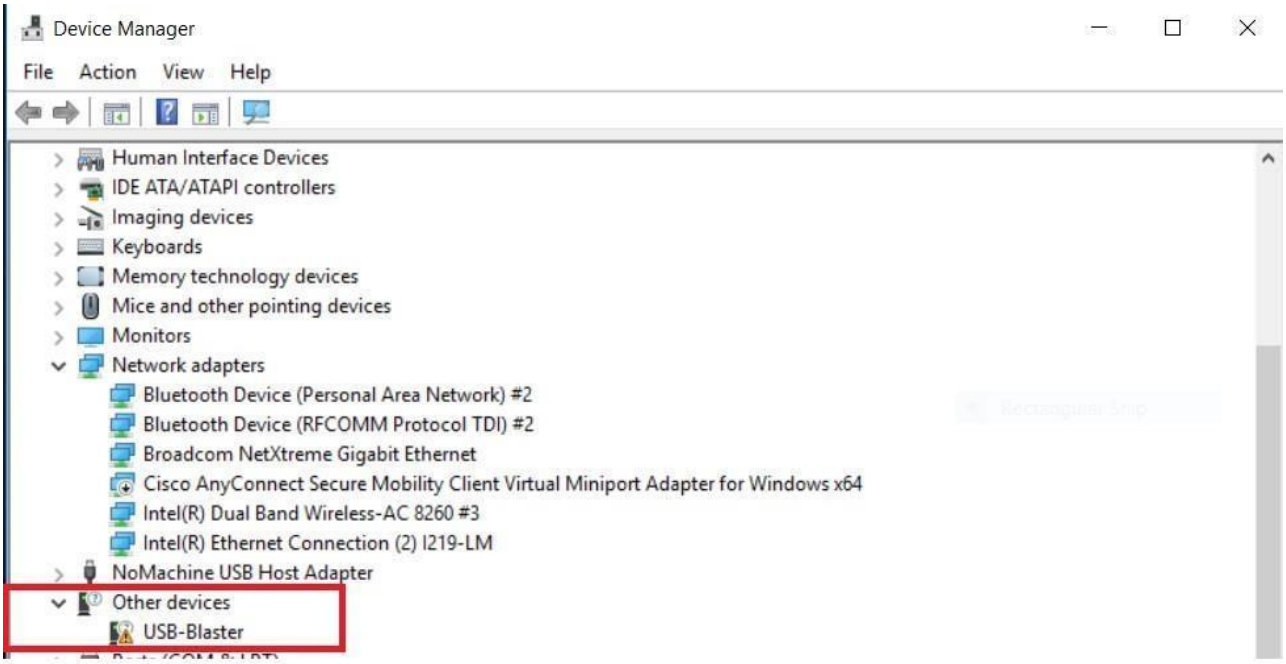


Figure 15: Device Manager Showing USB Blaster Drivers not Installed

Right click the USB-Blaster device and select "**Update Driver Software**". Choose to browse your computer for driver software and navigate to the path shown on the following page in Figure 12.

Figure 16: Directory Containing USB Blaster Drivers

Once you have the proper file path selected, click on "**Next**" and the driver for the USB Blaster should be installed.

3.6: Programming your Design into the FPGA

The next step will take the FPGA "image" and download it to your Cyclone V GX development kit.

Return to Quartus Prime and click  , a button located at the top of the main Quartus window to open the Programmer. A screen similar to the one seen in Figure 13 below should appear.

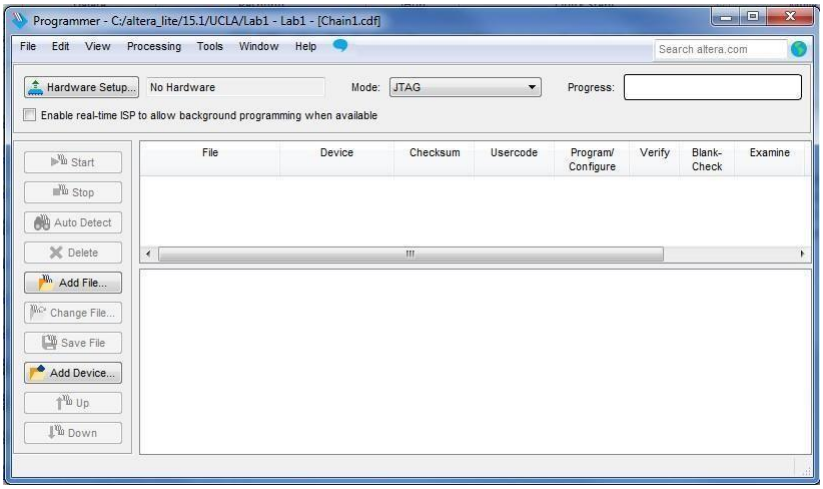


Figure 13: The Programmer Window

Click "**Hardware Setup**" up left corner and choose USB-Blaster (if the driver is properly installed) and close. Click on "**Add File**" on the left panel.

Navigate to the "output files" folder, choose "**Switch_to_LED.sof**" and then **"Open"**. This file is called a SRAM Object File and is the compiled image that programs your FPGA. Check the small "**Program Configure**" box in the middle of your screen.

Click "**Start**". Observe the progress meter complete to "100% (Successful)" Figure 14 below shows the programmer window after a successful .sof programming file has been uploaded to the FPGA

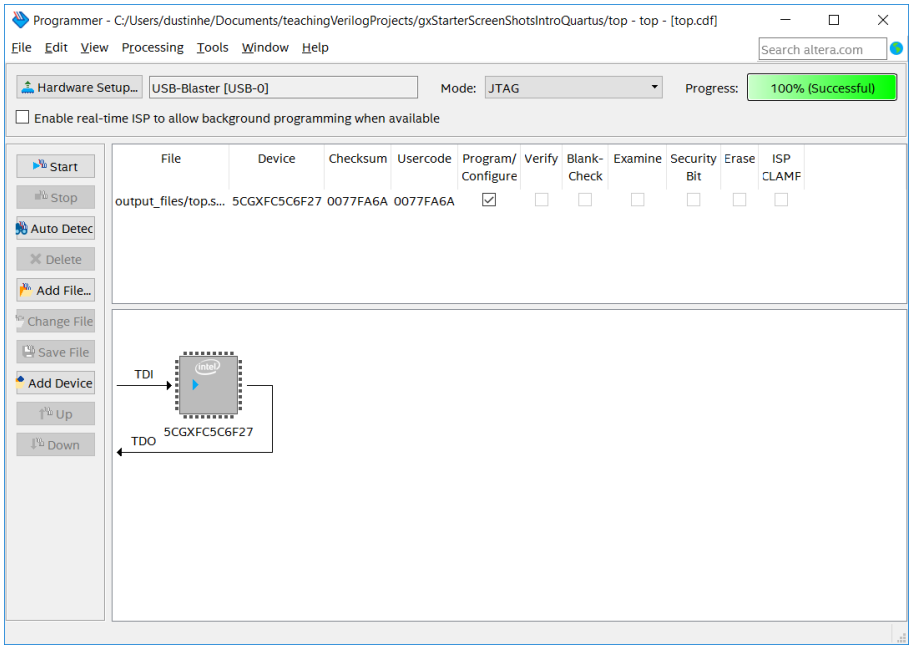


Figure 17: Successful Programming of the .sof File to the FPGA

3.7: Testing your Design

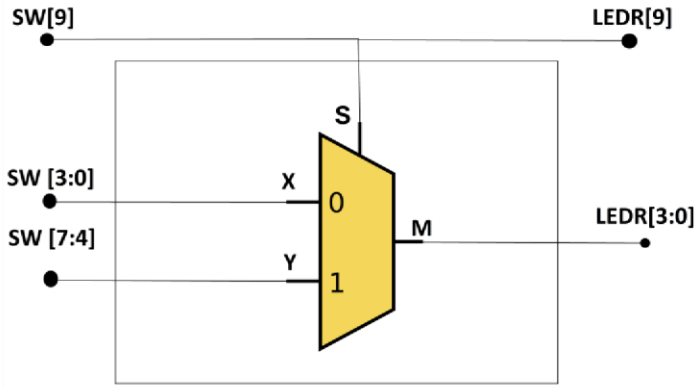
Check the functionality of the circuit by toggling the sliding switches (not the push buttons) and see the LEDs turn on and off.

Congratulations!

You have just completed the Switch_to_LED lab using the CV GX Development Board.

Close the programmer. If Quartus displays a dialogue box that asks you to save changes to your chain file, press the "**No**" selection.

Lab 4: 2 to 1 Multiplexer



Overview

Follow the steps from last lab and implement a 4 bit 2-to-1 multiplexer. A 2-1 multiplexer selects one of 2 data inputs. If the "S" pin is logic 0 M gets the value X, else M gets the value Y. Note this lab uses arrays. To define an array, refer back to section 3.2 which use a syntax such as input [3:0] X; to define the input signal

Instructions

4.1: Creating a Revision

In this lab we are going to use a handy feature in Quartus called revisions.
Using revisions will save you time since you can reuse the pin settings you made in the Pin Planner tool and carry them over to other projects. Launch the Revision tool by navigating to "**Project**" → "**Revisions**". Figure 15 below shows the revisions window that opens up.

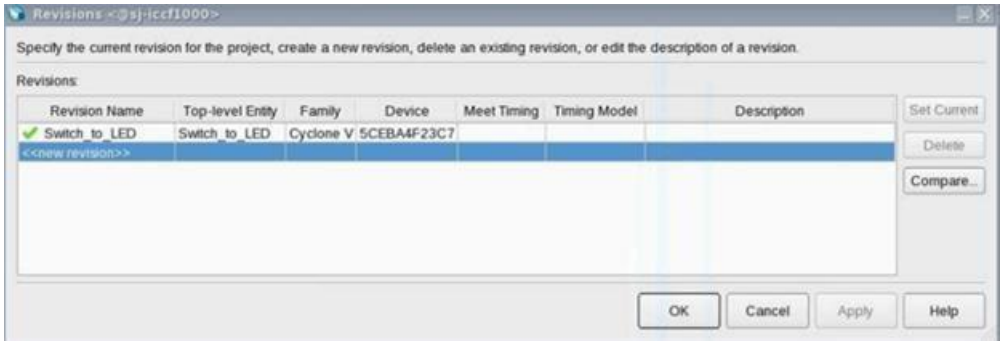


Figure 18: Quartus revisions window

Add a new revision by doubling clicking on the new revision selection and make the revision name "**Mux_2_to_1**". Figure 16 on the following page show the next window that should appear after pressing "**OK**".

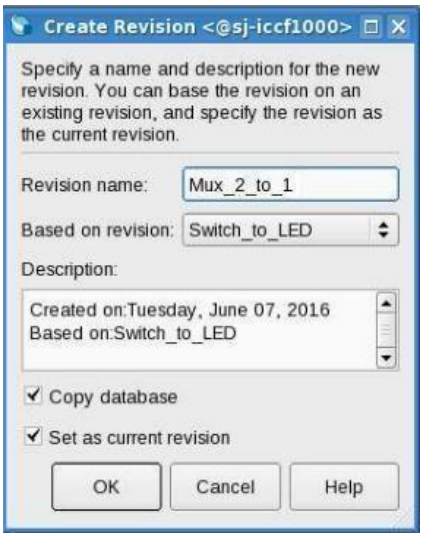


Figure 19: Final Revision Creation Window

It is important to note the difference between a project, revision, top level entity, and top level Verilog file of your overall project. A single project can have multiple revisions. The project name does not have to match the name of the top level entity of your design. Similarly, the Verilog file name might not have the same name as the top level entity and can indeed contain many modules (entities).

A common compilation error is a mismatch between what the top level module in your code is versus the one assigned in the Quartus settings. If you have a compilation error of this nature, check: **Assignments → Setting → General** and make sure the top level entity is indeed set to the one you think you are compiling in your Verilog source code.

4.2: 2-1 Mux Verilog Code

There are several approaches to this lab. If you are brand new to coding in Verilog you may copy and paste the code from section 4.6 (not the code snippets below). Should you choose this option, once you copy the code and save the Verilog file to the name Mux_2_to_1.v, you may skip to the next section of this lab manual.

The other option is to create a Verilog file from scratch for the 4-bit wide 2-to-1 multiplexer in your project. Take a look at section 3.2 on how to declare the ports on your module. This means to include the module statement and inputs/output definitions.

There are several ways to define a multiplexer in Verilog. Pick one of the 3 styles shown below. If time, try a couple of different coding styles for practice. Place these lines after the module definition and before the "**endmodule**" statement. You can also get the code from here.

Continuous Assignment:

```
assign M = (S==1) ? Y : X;  // if S then M = Y else M = X; All
                            // signals are of type wire.
```

Procedural Assignment "if" Statement:

```
 always @ (S or X or Y) begin // If any of the signals S, X or Y
                              // change state, execute this code.
                              // Note that signals to the left of an
                              // equal sign in an always block
                              // need to be declared of type reg so
                              // declare M as:
                              // output reg [3:0] M;
if(S == 1)
      M <= Y;    //Note the non-blocking operator '<='
else
      M <= X;
end
```

Procedural Assignment "case" Statement:

```
 always @ (S or X or Y) begin
   case (S):
   1'b0: M <= X;
   1'b1: M <= Y;
   endcase
 end
```

Also note that variables that are assigned to the left of an equal sign (= or <=) in an always block must be defined as **reg**. Other variables are defined as wire. If undeclared, variables default to a 1 bit wire.

Use switch **SW[9] as the 'S' input** (the selection bit of the multiplexer), switches **SW[3:0] as the 'X' input** and switches **SW[7:4] as the 'Y' input**. Display the value of the **input 'S' on LEDR[9]** (this can be done with an *assign* statement). **Assign 'M' to LEDR[3:0]**.

With the above port and signal assignments, we will see the output 'X' when the select input 'S' is low and we will see 'Y' when 'S' is high

4.3: Revision Control

Now you need to make sure you have the proper files included in your project. To the right of the Project Navigator Window, change hierarchy to Files. You will only be operating on the Mux_2_to_1.v so you will need to remove Switch_to_LED.v from your project.

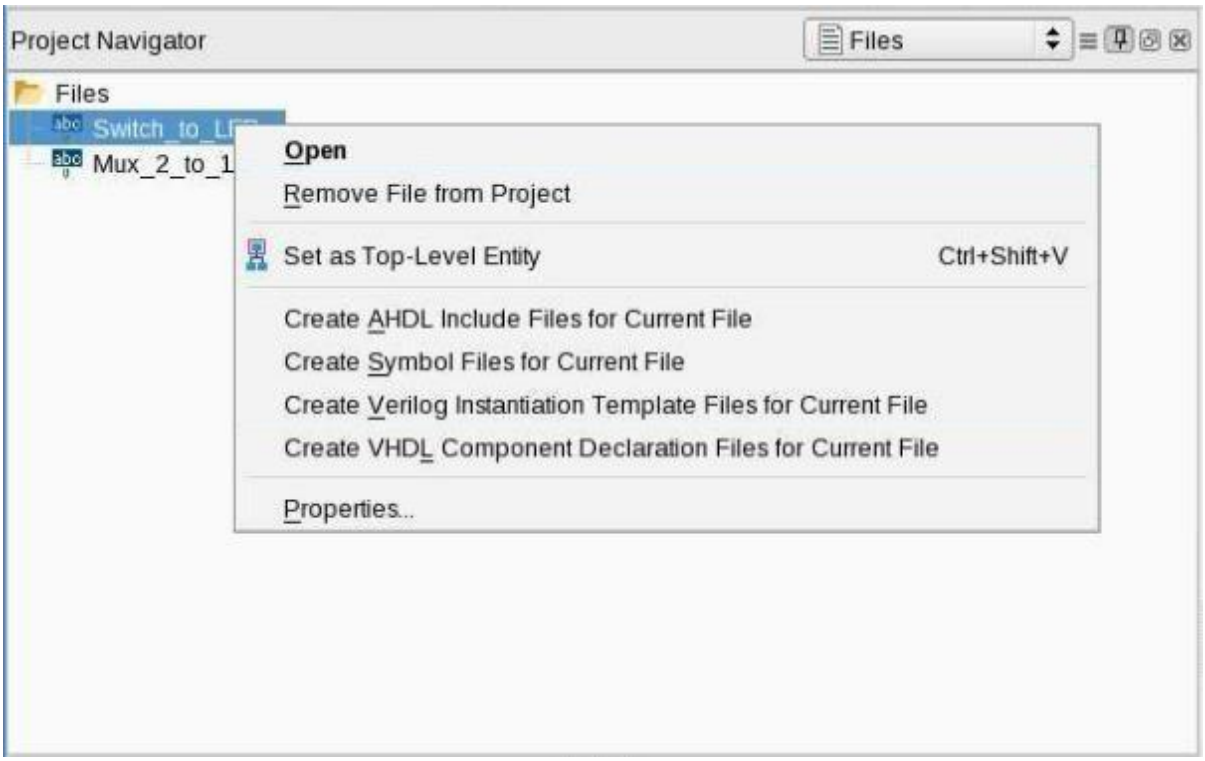


Figure 20: Changing Top Level Entity

Right click Switch_to_LED.v and remove this source file from your project. Next you need to change your top-level entity from Switch_to_LED to Mux_2_to_1. Select:
"**Assignments**"→"**Settings"**→"**General**". Change the Top-level entity to the Mux_2_to_1 design. Now your revision and your top level entity is Mux_2_to_1. Compile your design:

4.4: Checking Pin Locations and Editing (If Necessary)

In your project, the required pin assignments for your CV-GX Development board have carried over from the previous lab since the pin names are the same. Open up the Assignment Editor to make sure the pin names are indeed assigned to the appropriate pin locations.
"**Assignments**"→"**Assignment Editor**".

4.5: Downloading Your Design to Your Device

Once you have successfully compiled the project, download the resulting .sof file onto the FPGA chip as you did in section 3.6. Test the functionality of the 4-bit wide 2-to-1 multiplexer by toggling the switches and observing the LEDs.
Remember that we want the lights LEDR [3:0] to display the multiplexed result, LEDR[9] to display the switch SW[9] result (selection bit)

4.6: Working 2 to 1 MUX Verilog Code

In case you had problems getting things working, here is one complete implementation of the mux.

http://www.alterawiki.com/wiki/File:Mux_2_to_1.v

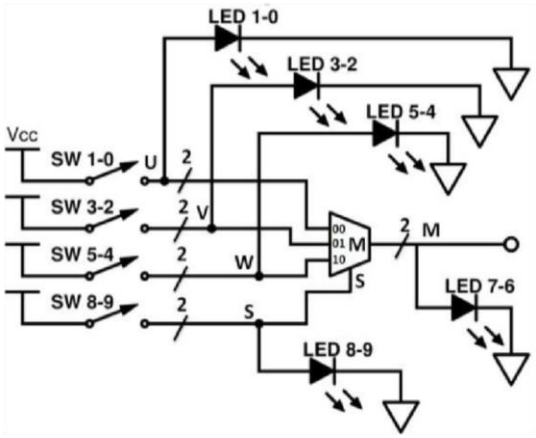
```
module Mux_2_to_1 (SW, LEDR);//Create module Mux_2_to_1

input [9:0]SW;    //Input Declarations: 10 slide switches
output[9:0]LEDR;  //Output Declarations: 10 red LED lights

wire S;                 //Declare the Select signal
wire [3:0] X, Y, M;     //Declare the inputs and outputs to the MUX

assign S = SW[9];    //Assigning input switches to internal signals
assign X = SW[3:0];
assign Y = SW[7:4];

assign LEDR[3:0] = M; //Assigning internal signals to output LEDs
assign LEDR[9] = SW[9];

assign M = (S == 0) ? X : Y;     //Mux Select Function

endmodule
```

Lab 5: 3-to-1 Multiplexer

Overview

Implement a 2-bit wide 3-to-1 multiplexer. This lab is similar to the previous lab. However, instead of a 2-to-1 mux, you will implement a 3-to-1. We will give you less hints on this part of the lab.

NOTE *If you are short on time, skip this lab and continue to lab 5*



Instructions

5.1 Create a Revision

Create a new Quartus revision for your circuit and call it Mux_3_to_1. Refer to step 4.1 if you do not remember the steps

5.2: 3-1 Verilog Module Multiplexer Code

Here is what to connect your ports to:

- **Input Ports:**
  - **'S' as SW[9:8]**
  - **'W' as SW[5:4]**
  - **'V' as SW[3:2]**
  - **'U' as SW[1:0]**
- **Output Ports**
  - **LEDR[9:8] as 'S'**
  - **LEDR[7:6] as 'M'**
  - **LEDR[5:4] as 'W'**
  - **LEDR[3:2] as 'V'**
  - **LEDR[1:0] as 'U'**

You are tasked with creating the correct internal signals and control logic for the 3-1 mux. If you get stuck, refer to section 5.5 for a complete and working solution.

5.3: Setting the Correct Top Level Entity and Compiling Your Design

Set up the proper files in the Project Navigator and make sure your top level entity setting is set appropriately to Mux_3_to_1. Compile the project. Note that if you observe the netlist viewer RTL, that it might not look exactly the diagram above, but it is functionally equivalent.

5.4: Downloading and Testing Your Design

Download the compiled design on to the development board. Test the functionality of the two-bit wide 3to-1 multiplexer by toggling the switches and observing the LEDs. Ensure that each of the inputs 'U', 'V' and 'W' are selected as the output 'M', and that the red LEDs next to the switches display the value on the switch.

Solution for 3 to 1 MUX

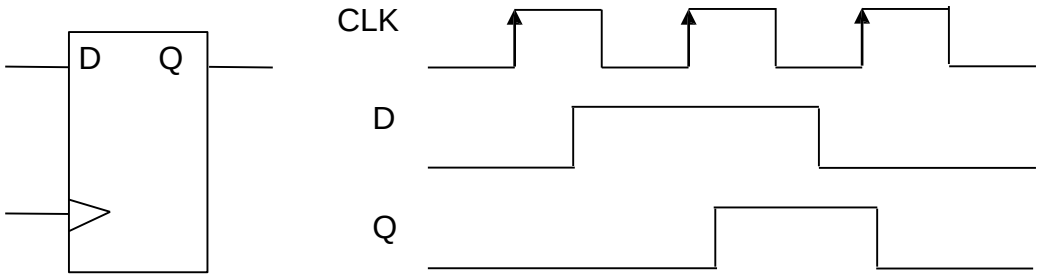
http://www.alterawiki.com/wiki/File:Mux_3_to_1.v

```
module Mux_3_to_1 (SW, LEDR); //Create module Mux_3_to_1
input [9:0] SW;           //Input Declarations: 10 slide switches
output [9:0] LEDR;        //Output Declarations: 10 red LED lights
wire [2:0] S, W, V, U, M; //Declare the Select signal and inputs and
                          //outputs of the MUX
reg [2:0] temp_M; //Temporary register for storing M due to using an
                  //always block
assign S = SW[9:8]; //Assigning input switches to internal signals
assign W = SW[5:4];
assign V = SW[3:2];
assign U = SW[1:0];
always @ (S or W or V or U) //3-1 MUX control logic.
begin //When S, W, V, or U change, the always block runs
  case(S) //Depending on the value of S, assign temp_M to a value
        2'b00: temp_M <= U;
        2'b01: temp_M <= V;
        2'b10: temp_M <= W;
        2'b11: temp_M <= 2'b00;
  endcase
end

assign M = temp_M; //Remove the value from register temp_M to M

assign LEDR[9:8] = SW[9:8];  //Assigning internal
                             //signals to output LEDs
assign LEDR[7:6] = M;
assign LEDR[5:4] = W;
assign LEDR[3:2] = V;
assign LEDR[1:0] = U;

endmodule
```

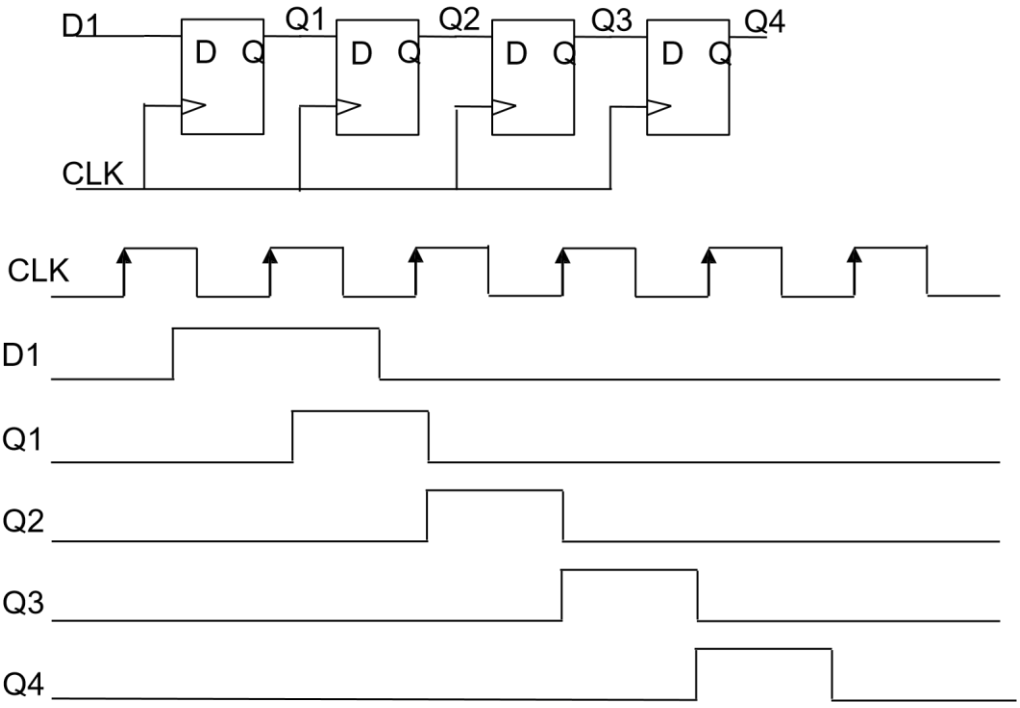
Lab 6: Knight Rider

Perhaps some of you have heard of or watched a TV show called Knight Rider that aired from 1982 to 1986 and starred David Hasselhoff. The premise of the show was David Hasselhoff was a high-tech crime fighter (at least high technology for 1982) and drove around an intelligent car named "KITT". The KITT car was a 1982 Pontiac Trans-Am sports car with all sorts of cool gadgets. The interesting gadget of interest for this lab were the headlights of KITT which consisted of a horizontal bar of lights that sequenced one at a time from left to right and back again at the rate of about 1/10th of a second per light. Check out this short YouTube[1] video for crime fighting and automotive lighting technology's finest moment. This lab will teach you a thing or two about sequential logic and flip-flops. Let's quickly review how flip-flops work.

Flip-flops are basic storage elements in digital electronics. In their simplest form, they have 3 pins: D, Q, and Clock. The diagram of voltage versus time (often referred to as a waveform) for a flip-flop is shown below. Flip-flops capture the value of the "D" pin when the clock pin (the one with the triangle at its input transitions from low to high). This value of D then shows up at the Q output of the flip-flop a very short time later.



When you connect several flip-flops together serially you get what is known as a shift register and that circuit serves as the basis for the Knight Rider LED circuit that we will study in this lab. Note how we clock in a 1 for a single cycle and it "shifts" through the circuit. If that "1" is driving an LED each successive LED will light up for 1/10 of a second.



6.1: Knight Rider Verilog Code

http://www.alterawiki.com/wiki/File:Knight_rider.v

The following Verilog code is the starting point for your Knight Rider design but there are some bugs. Start a new revision of the project "**lab**" and call it knight_rider with similar settings as the previous labs. **The code intentionally has errors. See if you can find them all.**

```
module knight_rider(
	input wire CLOCK_50,
	output wire [9:0] LEDR
);
	wire slow_clock;

	reg [3:0] count;
	reg count_up;

	clock_divider u0 (.fast_clock(CLOCK_50),.slow_clock(slow_clock));

	always @ (posedge slow_clock)
	begin
		if (count_up)
			count <= count + 1'b1
		else
			count <= count - 1'b1;
	end

	always @ (posedge clk)
	begin
		if (count == 9)
			count_up <= 1'b0;
		else if (count == 0)
			count_up <= 1'b1;
		else
			count_up <= count_up;
	end

	assign LEDR[9:0] = (1'b1 << count);

endmodule


module clock_divider(
	input fast_clock,
	output slow_clock
);

	parameter COUNTER_SIZE = 5;
	parameter COUNTER_MAX_COUNT = (2 ** COUNTER_SIZE) - 1;

	reg [COUNTER_SIZE-1:0] count;

	always @(posedge fast_clock)
	begin
		if(count==COUNTER_MAX_COUNT)
			count <= 0;
		else
			count<=count + 1'b1;
	end

	assign slow_clock = count[COUNTER_SIZE-1];

endmodule
```

6.2: Creating "knight_rider.v"

Open a new Verilog file called knight_rider.v and copy and paste the source code above into knight_rider.v. Make sure your top level entity is called knight_rider and the source file is knight_rider.v.

In the upper left Project Navigator window, you should see something similar to this:

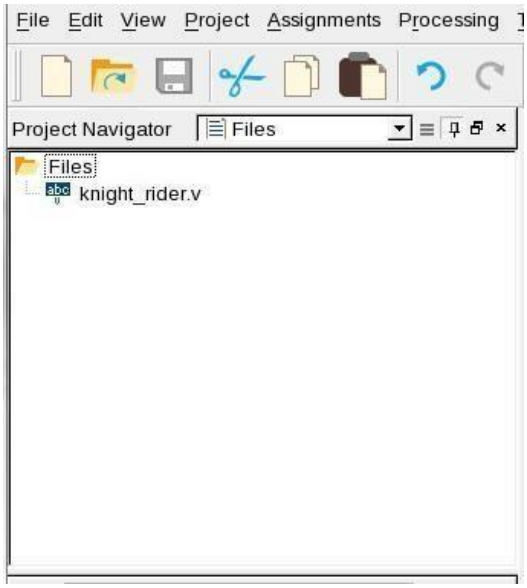


Figure 21: Example of knight_rider.v in Quartus Window

6.3: Debugging Code

Click on the Play  button and run analysis and elaboration**. This source code has several syntax bugs.**

Look at the transcript window on the bottom and observe the errors that are flagged with  the symbol. Carefully look at the source code and fix the errors and continue to recompile until the compilation steps run to completion.

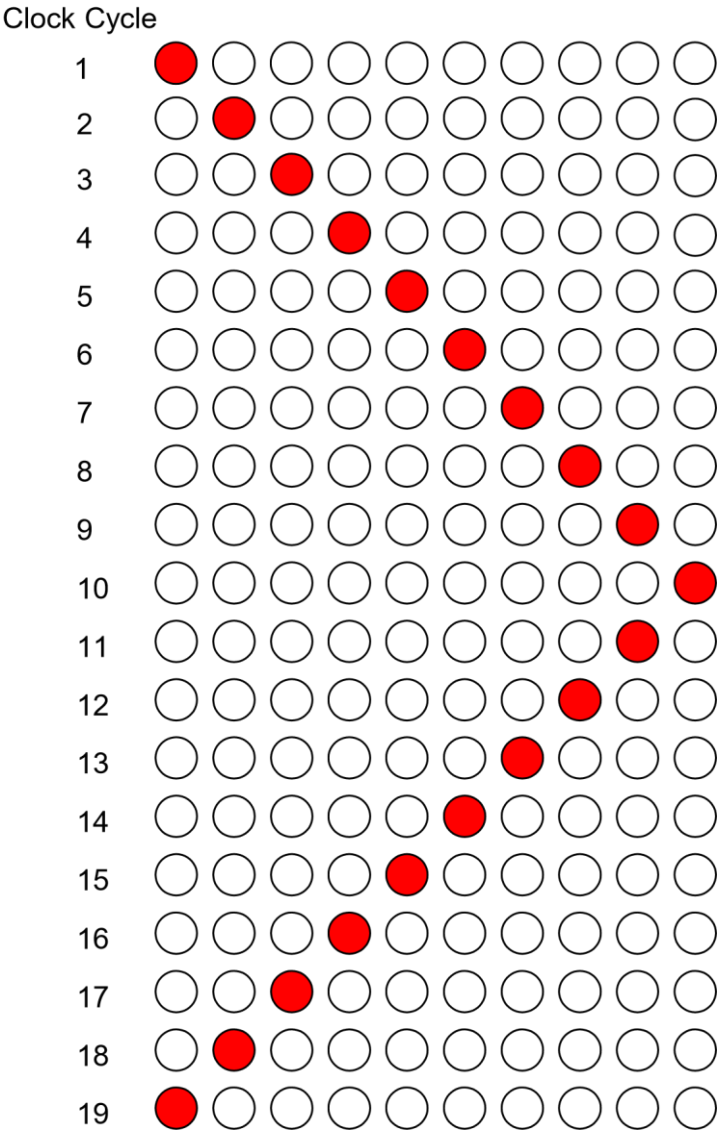
6.4: Assigning Pins with the Pin Planner Tool

Next you need to make sure the pins are in the right place. Open up the Pin Planner:  . Check the pins and make sure the LEDs are assigned to the same locations as the first lab. Note that there is an additional pin name called **CLOCK_50** in this design that needs to be assigned. The clock should be connected to **PIN_R20** (50 MHz). Note that the pin planner will use default locations that don't match your development kit unless specifically instructed.

Hit compile after changing the pinout.

6.5: Downloading Your Code to Your Device

By now you should have the hang of how to program the FPGA image into the CV GX Board. Go ahead and try it out. Do you see the infamous Knight Rider pattern? When working properly, you should see something like the following:



What do you see? **(If it does not work look at the next section)**

6.6: More Debugging

You knew we weren't going to make it that easy, did you? How come the lights don't sequence? Here is a portion of the explanation. The selected clock frequency of the CV GX Board is 50 MHz. That means the clock changes 50 million times per second. If you change the LEDs at that rate, you cannot view them with the naked eye. When you go through the code you will see a module in

your code called clock_divider. You want the output clock to toggle at around 10 Hz (10x per second). This clock_divider module takes the 50 MHz clock and divides down the clock to a slower frequency. Your lab instructor goofed and did not calculate the right divide ratio to slow the 50 MHz clock down to 10 Hz. You need to do a bit of math (including the log function!) to determine how to derive the proper size of the counter to divide 50 MHz to roughly 10 Hz. Basically, think about a divide ratio that is 2^N where N is the width of the counter. Adjust the parameter to COUNTER_SIZE to the appropriate ratio and recompile and reprogram the FPGA. Work out N based on the following equation: 10 = 50,000,000 / 2^N. Round N up
to the nearest integer to discover the proper WIDTH parameter setting. Recompile and program the CV GX development board.

6.7: Even More Debugging!

Is the knight rider sequence working properly? Does each LED stay on for ~1/10 second? If not, redo your math to find the right WIDTH parameter. Look at the sequencing carefully. Does each LED illuminate once and proceed to its neighboring LED? As you will observe, the LED[0] and LED[9] blink twice.

Dang – that lab instructor created another error in the design! Look at the source code in the knight_rider.v code and see if you can find the error. Change the code, recompile and reprogram your CV GX development kit until your Knight Rider LEDs are sequencing properly.

Thanks for taking time learning how to develop Altera FPGA products. We hope you found this lab informative. Long live David Hasselhoff[2]!

1. https://www.youtube.com/watch?v=Mo8Qls0HnWo
2. https://www.youtube.com/watch?v=PJQVlVHsFF8

Revision History

3/21/2016	L. Landis	Initial Release
4/13/2016	L. Landis	Remove Modelsim from download package
5/24/2016	P. Mayer	Fixed broken links, updated for Quartus 16.0, added a few extra assignments
6/3/2016	L. Landis	Added solution to 2:1 Mux lab
6/7/2016	L. Landis	Added revision for copying pin assignments
7/20/2016	L. Landis	Clarify Mux_2_to_1 copy and paste code
10/3/2016	L. Landis	Clarify no driver image; typos
3/16/2017	A. Weinstein	Added USB Blaster driver installation instruction. Added table of figures and figure numbers. Made instructions clearer w.r.t. revision control and when writing Verilog code for labs. Added a solution for the 3-1 MUX lab.
10/10/17	D. Henderson	General document formatting and clean up. Additionally, updated wiki links and some screen shots. Last, added TCL script instructions for assigning pins.

Table 2: revision control history