UNIVERSITY OF WATERLOO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE-124 LAB MANUAL

V3.0C

Course: ECE-124 Digital Circuits and Systems

Contents

LIST OF FIGURES:

1 Introduction and ECE-124 Labs Outline

ECE-124 is an introductory course on Digital Logic Design and Implementation

Each laboratory experiment has several parts:

1. A prelab section to prepare you for the Lab session.

2. A three-hour lab session which is used to develop your Lab design. Help is available.

3. A final lab report, one day (24-Hour) after your demo for each lab except for Lab1. You must regard the guidelines in the lab manual as the final reference. Late lab reports will lose marks of 1 mark per day.

There is a Late-Submission Drop Box.

4. Absolutely no food or drink is allowed in the laboratory. Do not leave the doors or windows open.

The room will be closed after hours if the rules cannot be followed.

5. Following the Lab Manual is important but please note that:

FURTHER DETAILS FOR THE LAB SESSION FPGA DESIGN WORK IS PROVIDED DURING THE LAB SESSION. ATTENDANCE IS THEREFORE REQUIRED.

Each workstation in the ECE 124 lab is equipped with:

1. University of Waterloo LogicalStep Board housing an Altera MAX10 Field Programmable Gate Array (FPGA) chip and various peripheral components
2. Intel (Altera) Quartus-Prime FPGA Design v15.1 Software

We are going to be briefly covering the design of the board platform that will be used to prove the implementation of the FPGA designs during the course. This board platform is the University of Waterloo - LogicalStep board. Figure 1 is a photo of the U of W LogicalStep board that will be used for your experiments in the course.

Figure 1 - LogicalStep Board

Figure 2 is a sketch of the external FPGA designer resources (peripherals) that can be used for various FPGA design projects.

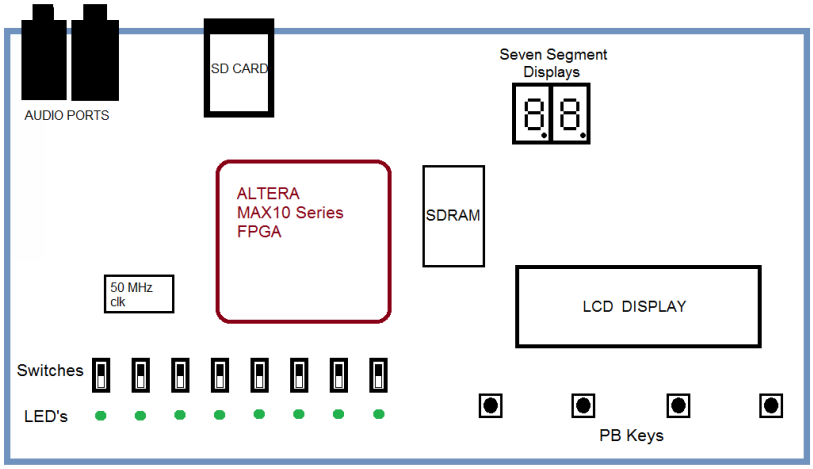


Figure 2 - LogicalStep Block Diagram

1.1 LogicalStep FPGA Board User Peripherals

The LogicalStep board is equipped with some peripherals that can be used to create various applications. These peripherals are things such as a clock source, switches and Push-Button Key inputs, output LED indicators, SDRAM, SD Card (like a mini-disk) and audio electronics etc.

For the ECE-124 labs we are only using the clock source, switches and push buttons for supplying logic inputs and for outputs we will use the LED's and the Seven-Segment Displays.

LEDs are electronic components which can emit light with much greater efficiency than incandescent lamps. Specific pins on the FPGA are connected on the LogicalStep board to drive these LED's.

Other LED indicators on the board are the Seven-Segment displays (so called because it is an arrangement of 7 bar segments). They are arranged in such a way that the numbers 0 to 9, and letters A to F, can be displayed. Seven segments is the minimum number which can uniquely display numbers and that is why they are used for many calculator or electronic displays. To drive these displays a Seven-Segment decoder (inside the FPGA) is usually used to converts a 4 bit binary number to a Seven-Segment LED pattern so that a person can see a number or letter instead of trying to interpret the original 4 bit binary number. There are also 8 slide switch inputs and 4 push-button key inputs. A 50 MHz clock is connected to the FPGA to drive logic processing.

1.2 Field Programmable Gate Array (FPGA) Technology

FPGA technology has been around since the mid 1980's. It has gradually grown in complexity and capability and is now a dominant technology used in the workplace. For a typical FPGA there are two prime components involved within its architecture. These are the Logic Elements (LE's) and the interconnect resources. Both of these items are completely configurable after the FPGA device is powered. The Logic Elements are used to implement gate-level logic and the interconnect section is used to connect the various LE's in some arrangement. The design files that are downloaded into the device are developed on an external software-based design platform (such as Altera Quartus Prime). When a designer has processed the design in Quartus, its download file can be sent into the appropriate FPGA for use with other technology on a board.

But one might ask "How does a FPGA download file become functioning hardware?".

In the sketch shown below all of the logic functions and interconnect that are in an FPGA device are controlled by configuration memory cells located in the FPGA. These memory cells directly control the activation or deactivation of switch transistors. These transistors are placed in the FPGA to control logic lookup-tables (inside the LE blocks) and interconnect path link connections.

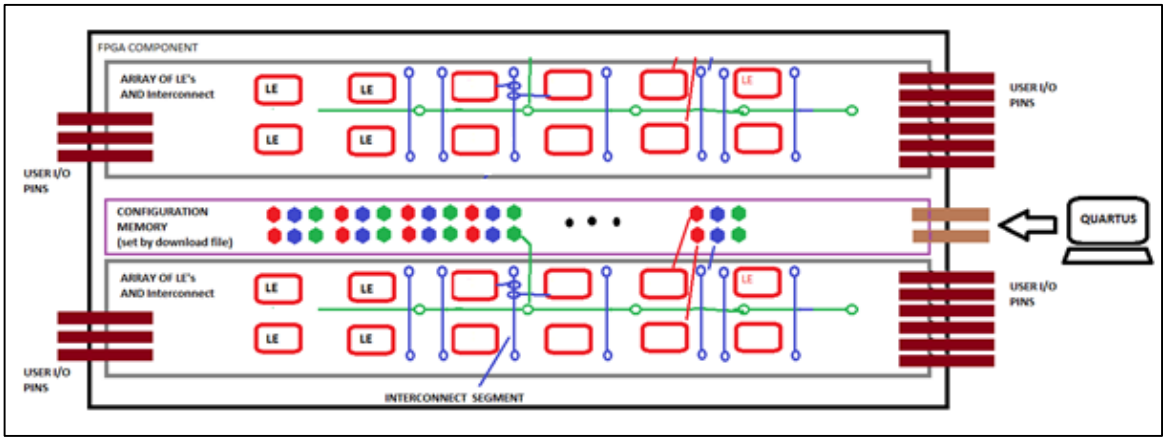


Figure 3 - Typical FPGA Configuration Memory

Thus a download file can be used to implement any logic hardware function or functions within the FPGA. Since the FPGA can be reconfigured, it can implement different hardware functions by simply having different download files sent to it. The external I/O connections are usually fixed on a board design however and they must be kept in mind when considering design new FPGA designs. FPGAs are different than microprocessors or microcontrollers because the designer is able to change the **hardware design** of the logic.

The FPGA device that is used on the LogicalStep board that you will be using is a current technology FPGA platform. Although it is one of the smaller FPGA's offered in the marketplace it can still pack a

lot of functionality into the device's 8K LE's and 144 pins. This FPGA offers many other kinds of internal resources (DSP', RAM, PLL's) but they are beyond the scope of the Lab course.

1.3 FPGA Design Software

This course will be using the Quartus Prime set of tools to develop the FPGA designs and it is a full FPGA design software suite from Altera (now Intel). It aids the designer through the different stages of describing the hardware design and targeting it for a certain Altera FPGA chip. The typical development process uses the following stages:

- Design Entry:
  - Schematic Entry: by connecting blocks of ranging complexity. It can be used to interconnect simple components such as simple logic gates or to interconnect previously created hardware blocks
  - Hardware Description Language: such as VHDL or Verilog (we use VHDL)

- Design Compilation Part 1:
  - Analysis and Synthesis: An HDL or schematic file is analyzed and is mapped into a number of logic gate equivalents and their connections. The synthesized file can be used for functional simulations.

- Circuit Simulation:
  - Functional Simulation: This is used to verify the logic functionality of the design before further processing steps are attempted. Any functional errors are fixed in the design

- Design Compilation Part 2:
  - Repeat of Analysis and Synthesis: The proven functional design is synthesized for further processing.
  - Place and Route: the synthesized logic is arranged (Placement) for an FPGA device. After the Placement phase a Routing algorithm determines the best way to connect the logic. The routing details yield timing information about the "placed and routed" design.

- Timing Analysis:
  - This is usually required for higher performance or higher density designs that have been placed and routed. It gives an accurate indication of how fast the circuit can run and how much timing margin is available for various operating conditions. Timing errors are fixed in the design.

- Design Compilation Part 3:
  - Repeat of Analysis / Synthesis / Placement / Route: The fully implemented design is processed.
  - Assembly: a load file is produced so that it can be downloaded to the FPGA chip.

- Programming the FPGA:
  - The circuit can be physically tested afterwards by applying inputs and observing the outputs.

1.4 ECE-124 Lab Sessions Outline:

Lab Sessions:

1. Session 1: Lab Procedures; Lab1- Design Entry Methods Using Intel (Altera) Quartus Prime Tools (3 hrs)
2. Session 2: Demo from Lab1; Lab2- VHDL for Combinational Circuits PART 1 (3 hrs) – Simple ALU Design
3. Session 3: Demo from Lab2; Lab3- VHDL for Combinational Circuits PART 2 (3 hrs) – Energy Monitor
4. Session 4: Demo from Lab3; Lab4- VHDL for Sequential Circuits (3 hrs)- Flip-flops/State Machines
5. Session 5: Demo from Lab4; Lab5- Wrap-up of Sequential Circuits (3 hrs) - Robotic Arm Controller

The Lab Stations are made for groups of two. Find a lab partner before or during the first Lab.

NOTE: in the Lab Manual there are numerous, highly detailed screen-shots. The fine print on those pictures may be readable by using the ZOOM feature in your document viewer.

2 Lab 1 – Design Entry Using Altera Quartus-II

The goal of this lab session is primarily to gain experience with the Quartus Prime FPGA Design Environment. Lab1 will go through two design entry methods and some simulation steps for design testing. Later the design will be processed for the programming of an FPGA on the LogicalStep board to observe how the logic circuit actually works in hardware. Since Lab1 starts so early in the term relative to the lecture material we will be doing some basic examination of some two-input gate functions.

2.1 Lab1 Intended Learning Outcomes

By the end of this lab students should be able to:

1) **CONSTRUCT** a digital circuit in both schematic and VHDL representations
2) **VERIFY** digital circuit representations using functional simulations.

2.2 Prelab

No prelab work is necessary for LAB1. However students should familiarize themselves with the ECE-124 Lab Manual Outline and FPGA technology sections.

2.3 Lab1 Outline:

Attendance will be taken. Your Team partnerships will be settled before or during LAB1 using Learn Groups.

Lab 1 is composed of the following main categories:

1. A brief introduction to the laboratory, its equipment and the student conduct expected during Lab sessions.
2. Learning two Design entry methods (Schematic and VHDL) for small digital circuits.
3. Running Synthesis and Simulation processes on the circuits to check that they operate as expected.
4. Compiling the circuits into files for programming the LogicalStep board to confirm that the hardware implementation functions as expected.
5. Modifying the circuit designs to provide new functionality and then test them.

2.4 Lab1 Activities

2.4.1 Starting Your Lab1 Project

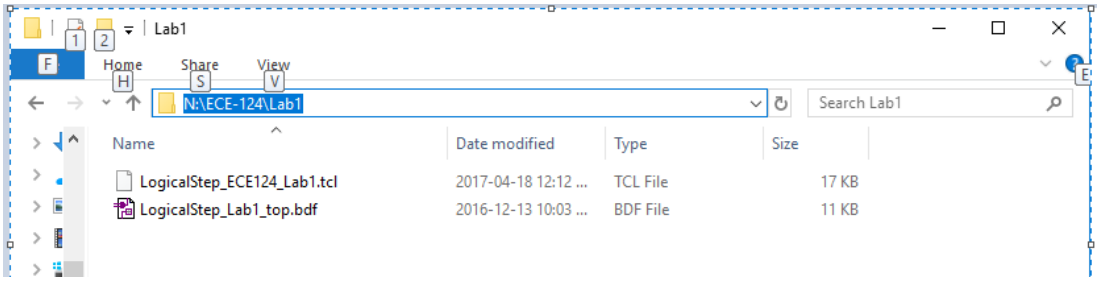
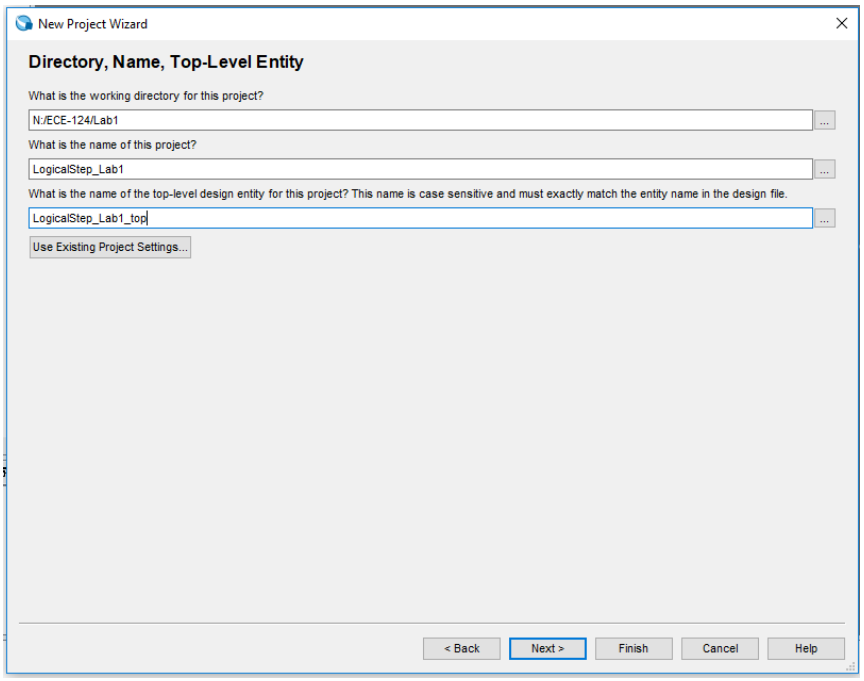


Figure 4 Lab1: Starting Lab1 Project Folder

To begin your Lab1 project use the Windows 10 File Explorer to browse the N: drive and create a folder called ECE-124. From LEARN download the Lab1 Zipped folder "Lab1" into the ECE-124 folder on the N: drive. Extract the contents into a new Lab1 project folder. The extracted files are as follows in Figure 4:



Start the Quartus Prime software to begin a new project. Go to the FILE tab.

SELECT FILE>New Project Wizard. Click NEXT to go to the second slide.
The project parameters will now be entered as in Figure 5.
Project Folder: **N:/ECE-124/Lab1**
Project Name: **LogicalStep_Lab1**
Project Top Level: **LogicalStep_Lab1_top**

Click FINISH on the Wizard Dialog Window.

After the setup has completed you should see the following in your project folder as in Figure 6:

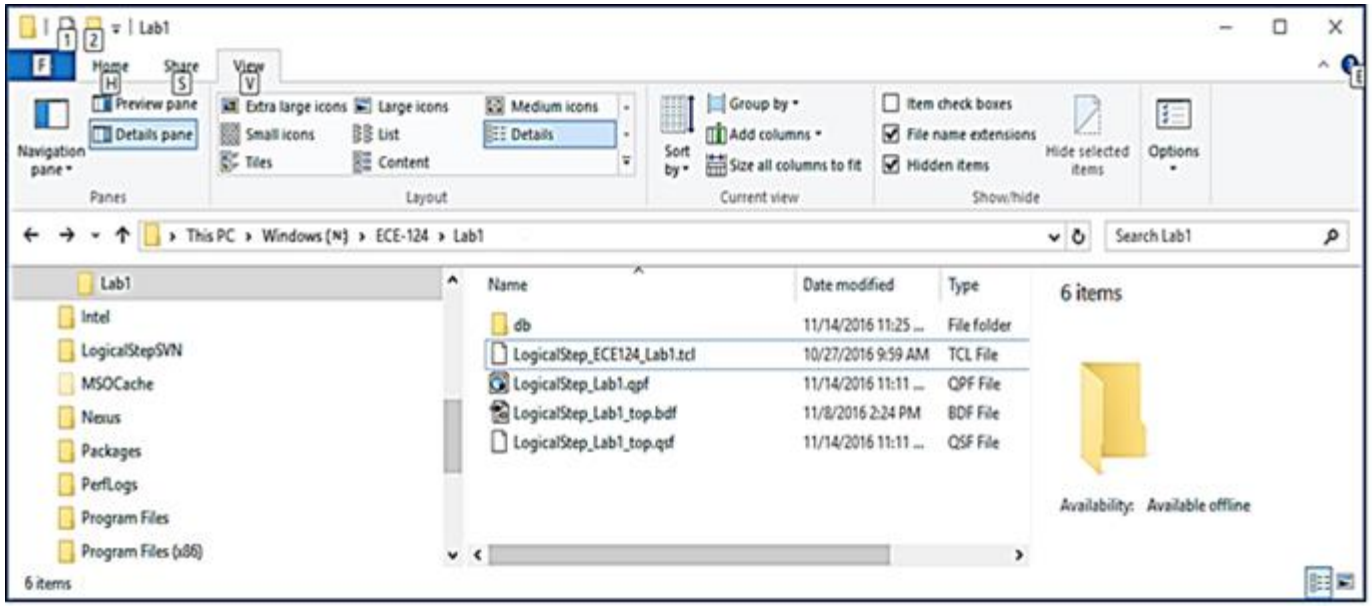


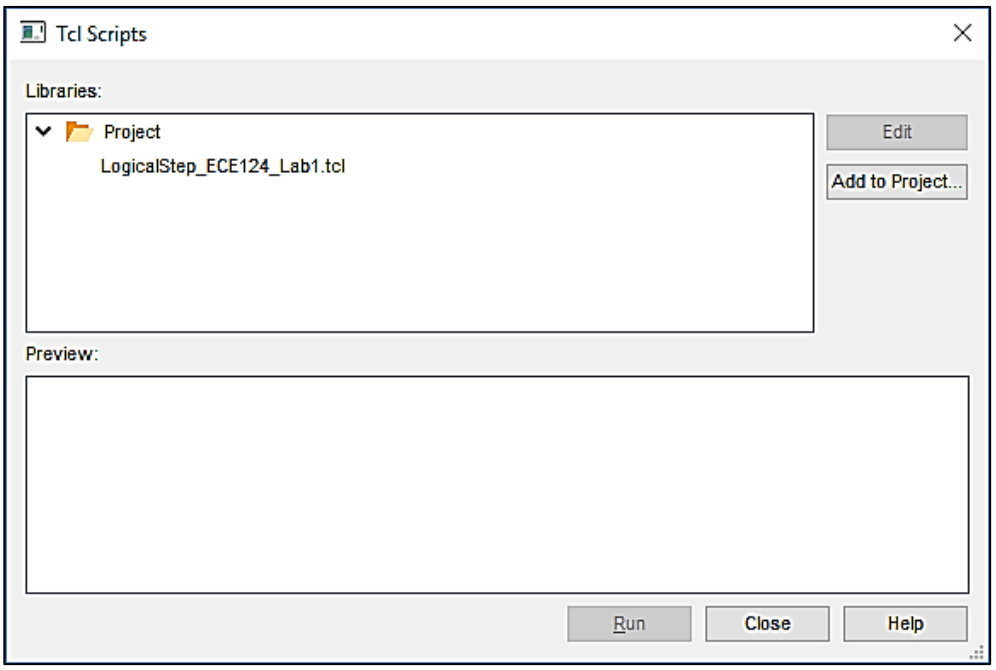
Figure 6 Lab1: Project Folder After Setup

Notice in the Project folder the file LogicalStep_Lab1.qpf (Quartus Project File or QPF).

In later FPGA design work you can do one of two procedures to get back into your FPGA project to run in Quartus for Lab1.

1) You can browse to the Project folder and "double-click on the QPF file. This will launch Quartus and will load your FPGA design that you saved previously.
2) Alternatively, you can invoke the Quartus Prime v15.1 tools and then go to the FILE>Open_Project tab and then browse to the QPF file in your project folder and select the QPF file.

Next, in Quartus, the TCL script must be run to assign the FPGA device type that is being used for this lab and then pin assignments for the FPGA that are reserved for the LogicStep FPGA and finally the project LogicalStep_Lab1 is opened.



Go to the Tools TAB and SELECT Tools>Tcl Scripts. The following dialog box (Figure 7) should appear:

SELECT the TCL (pronounced as “tickle”) file and then click on the RUN button.

Figure 7 Lab1: TCL Script Invocation

The Figure 8 window should appear when it is finished.

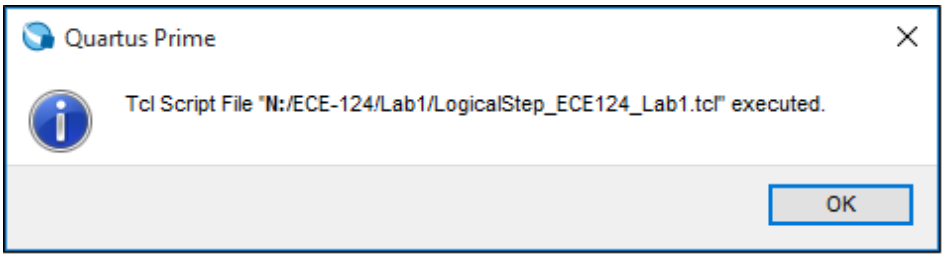


Figure 8 Lab1: TCL File Completed

Click OK. If it runs instantaneously then your project name is incorrect and you will be unable to program the FPGA later.

Then SELECT the CLOSE button on the TCL Script Dialog window.

NOTE that this TCL file will NOT have to be run again for the entire Lab1 project since the pin and FPGA Device assignments are established.

DEEP DIVE :

With the assignments made with the TCL file you can observe the signal pins that are used on the FPGA by calling up the Pin Planner utility. Go to the ASSIGNMENTS Tab and select the Pin Planner option. You should see something like the following:

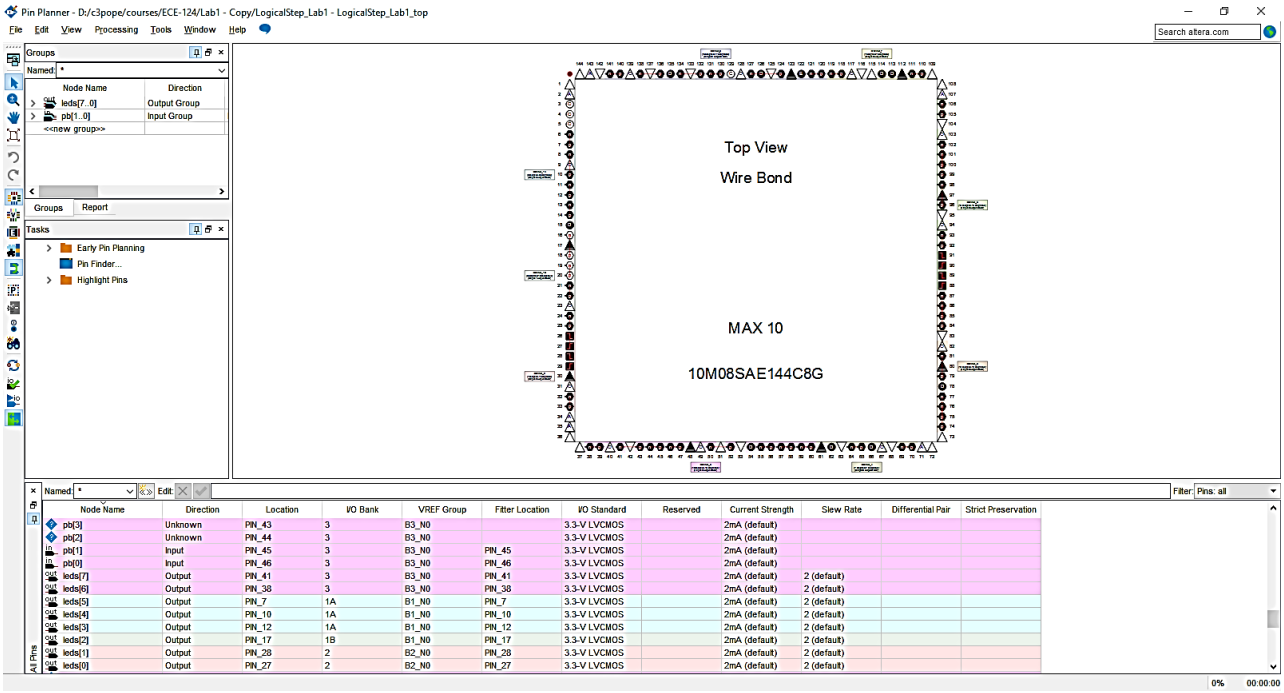


Figure 9 Lab1: FPGA PIN PLANNER

Earlier the top level file for this lab was downloaded from LEARN into the Lab1 project folder. The top level file is in schematic form (see Figure 10). Schematic entry methods for a simple set of gate-level functions will be the first part of this lab. Go to FILE Tab and SELECT File>Open and then browse to the LogicalStep_Lab1_top.bdf file (see Figure 6).

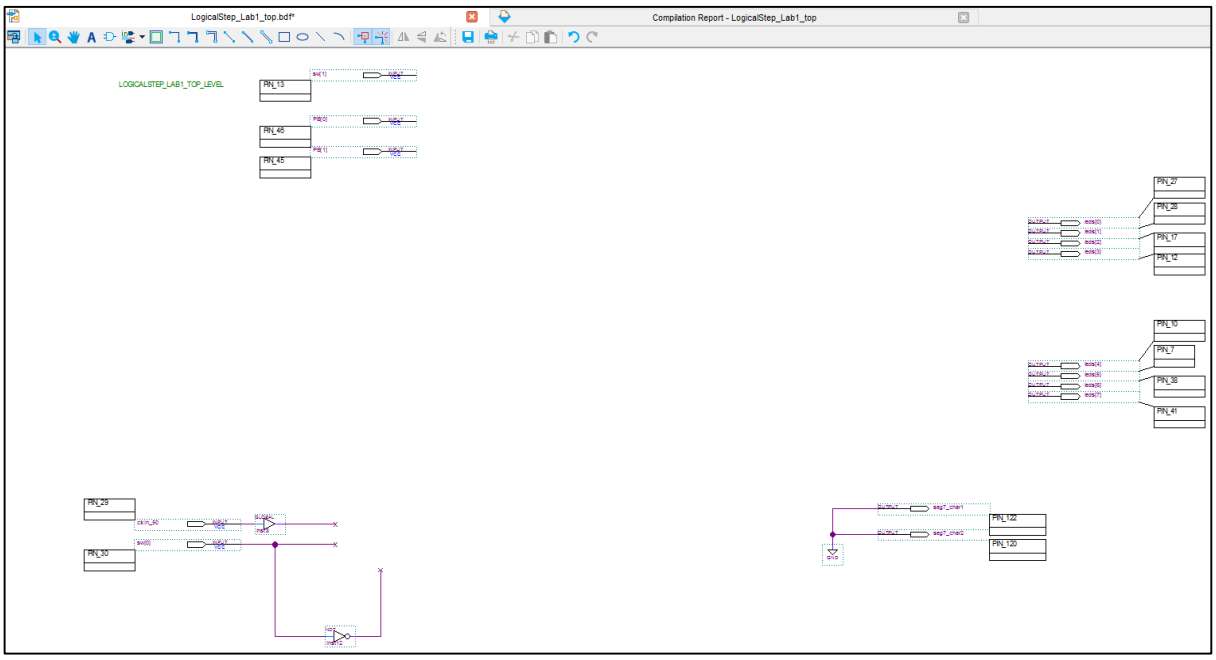
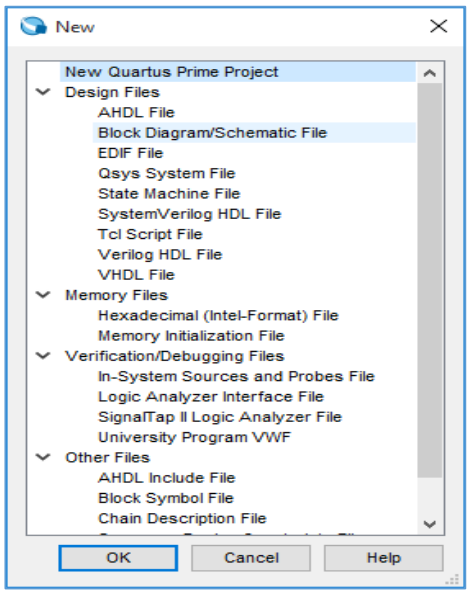


Figure 10 Lab1: Starting Top Level Schematic

There are two pins for Push-Button Key Inputs, two pins for Switch inputs, eight pins for LED Outputs and a pin for a Clock Input provided in the Lab1 design. The Clock Input will be used later in the lab session.



Now we can start adding some design hierarchy by creating functional blocks and then installing them at the top level design later. The first block that will be created will use the schematic entry format. Go to the FILE Tab and select File>New. The dialog box as shown in Figure 11 will open. SELECT the Block Diagram/Schematic File. A blank schematic window will then open in Quartus. After it opens save the schematic file as "schem_gates.bdf" by using the FILE Tab and selecting File>Save As.

Figure 11 Lab1: New Schematic Creation for schem_gates Block

To insert schematic symbols on the schem_gates sheet RIGHT-CLICK anywhere on the schematic sheet (as in Figure 12) and a dialog box should appear. SELECT the Insert>Symbol option.

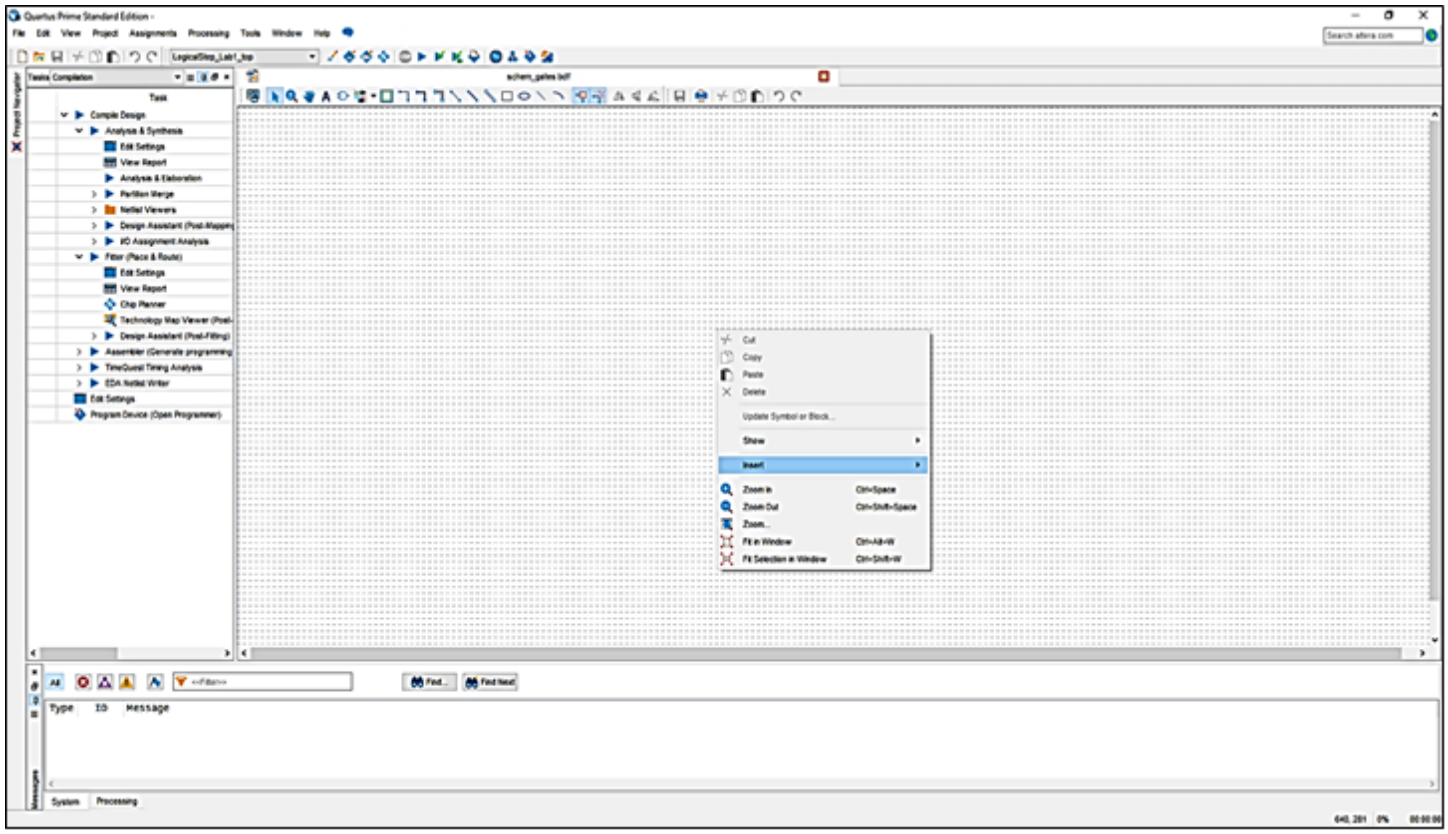


Figure 12 Lab1: Insertion of Library Symbols into schem_gates Schematic

To this schematic we will add INPUT pins on the left and OUTPUT pins on the right (typical for schematic convention for readability). So within the Symbols Dialog box that will appear browse to the altera/quartus libraries and then SELECT the Primitives/pin folder. Here you can select the pins as required and place them on the schematic sheet.

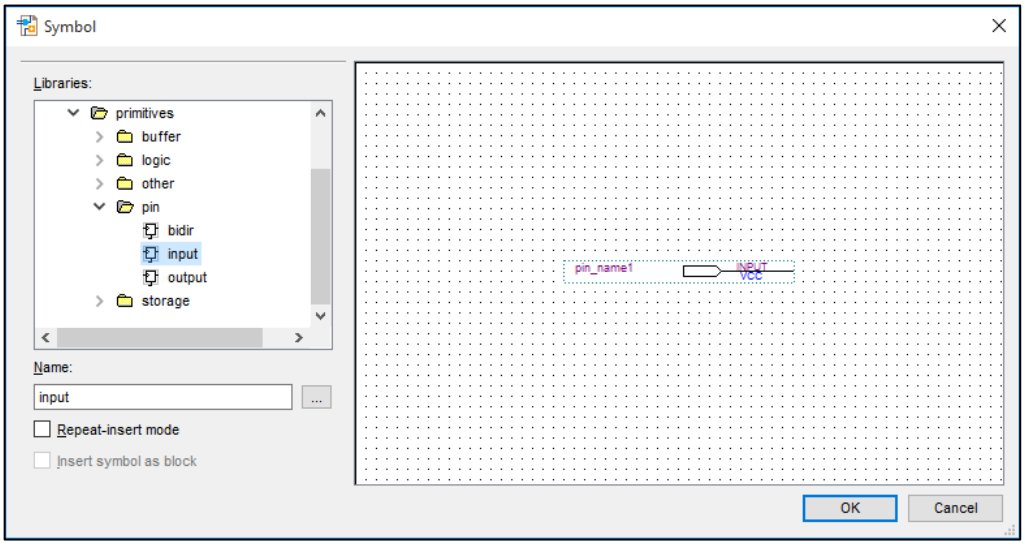


Figure 13 Lab1: Locating Pins in Symbol Library for schem_gates Block

Add Inputs and Outputs to the schematic sheet as shown below in Figure 14:

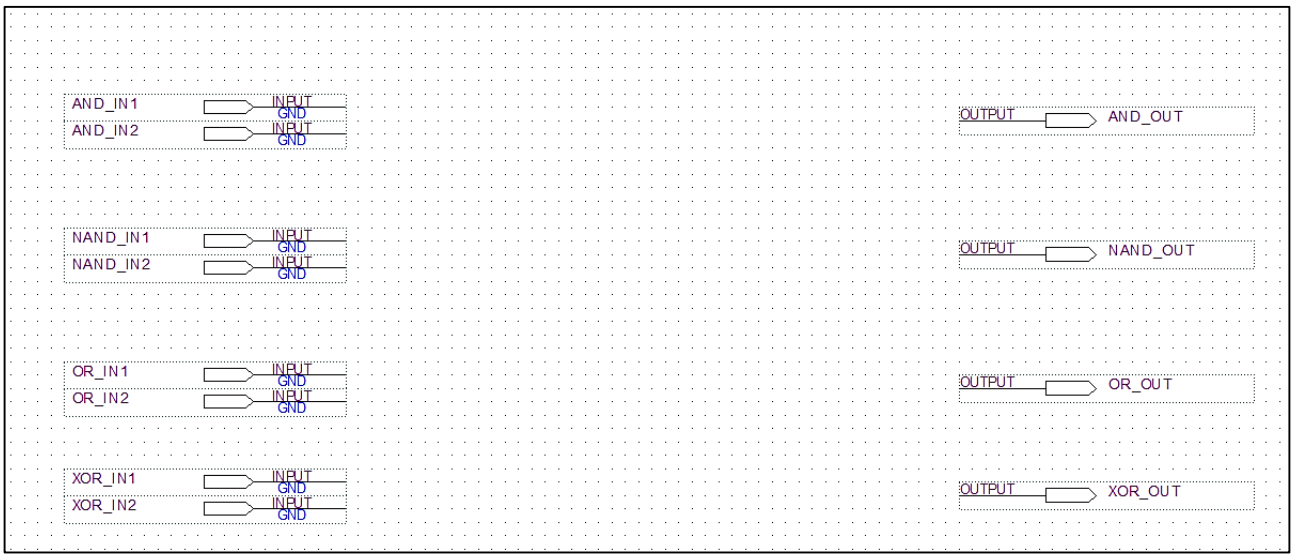


Figure 14 Lab1: Insertion of I/O Pins into schem_gates Schematic

List of inputs:
AND_IN1, AND_IN2, NAND_IN1, NAND_IN2, OR_IN1, OR_IN2, XOR_IN1, XOR_IN2
List of outputs:
AND_OUT, NAND_OUT, OR_OUT, XOR_OUT
Name each of the pins as in the lists above. (Double-click each pin and modify its Name property).

After placing and naming the pins on the schematic sheet return again to the symbol Libraries for logic gates in the Primitives/Logic folder.

We will only be using 2 input gates for this lab. Below is a truth table for the gates that are to be entered. Also notice how in the INPUTS that bit1 changes at half the rate of bit0.

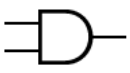
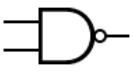
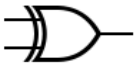
IN 1	IN 0	AND	NAND	OR	XOR
0	0	0	1	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	0	1	0
GATE SYMBOL →					

Figure 15 Lab1: Look-up Table for Gate Logic Functions

You must locate the basic 2 input gate functions (AND, NAND, OR, XOR) from the altera/quartus libraries and insert them into this schematic. Below in Figure 16 is an example of the two input AND gate.

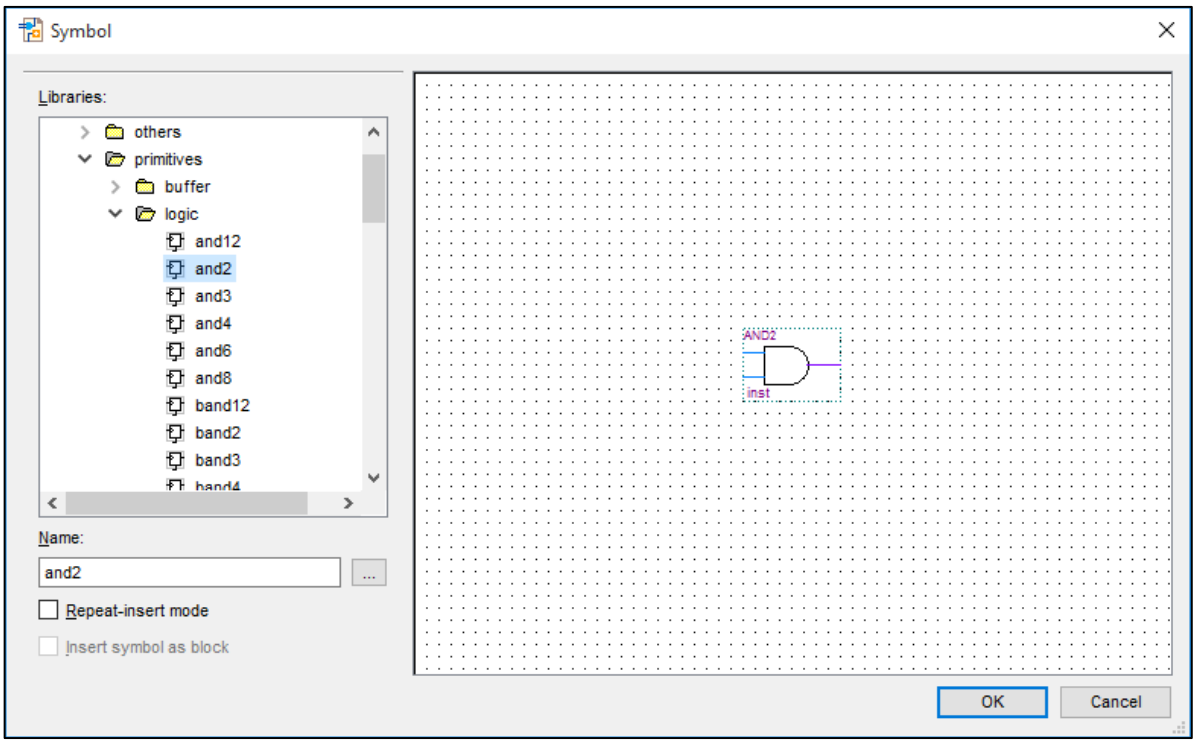


Figure 16 Lab1: Locating Gates in Symbol Library for schem_gates Block

Connect the input pins to the gate inputs and the output pins to the gate outputs as shown below in Figure 17. Use the Orthogonal Node Tool (highlighted below). After the schematic is drawn save the file. Go to the FILE Tab and select File>Save.

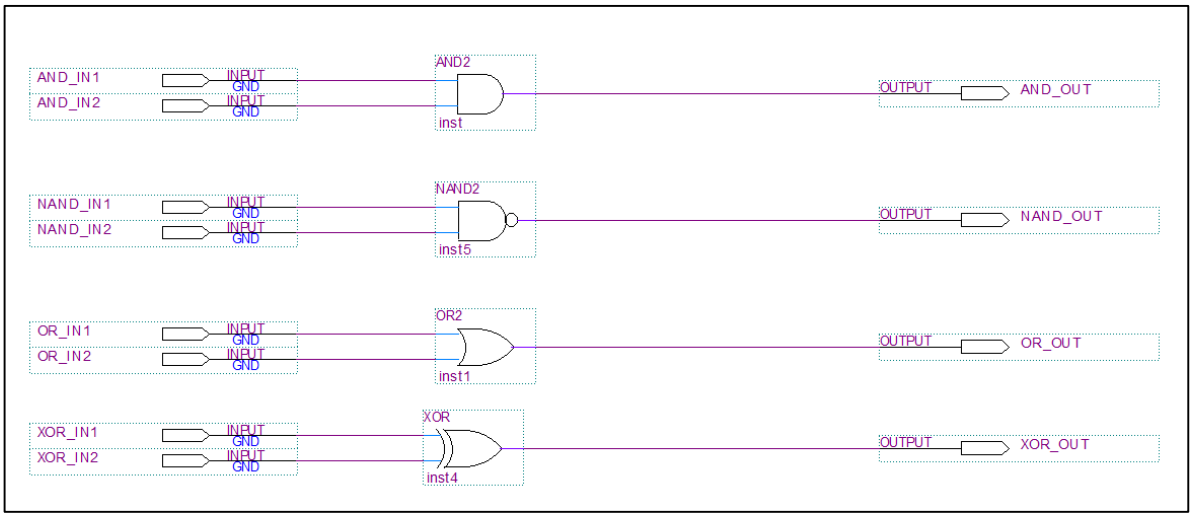


Figure 17 Lab1: Connecting Gates in schem_gates Block

Now that a schematic design file has been created we must make a symbol for it so that we can add its symbol to the top level schematic (LogicalStep_Lab1_top). Go to the FILE Tab.

SELECT the File>Create / Update> Create Symbol Files for Current File option. A Window like the one in Figure 18 appears. The symbol filename option for “schem_gates.bsf” should be visible.

Click Save.

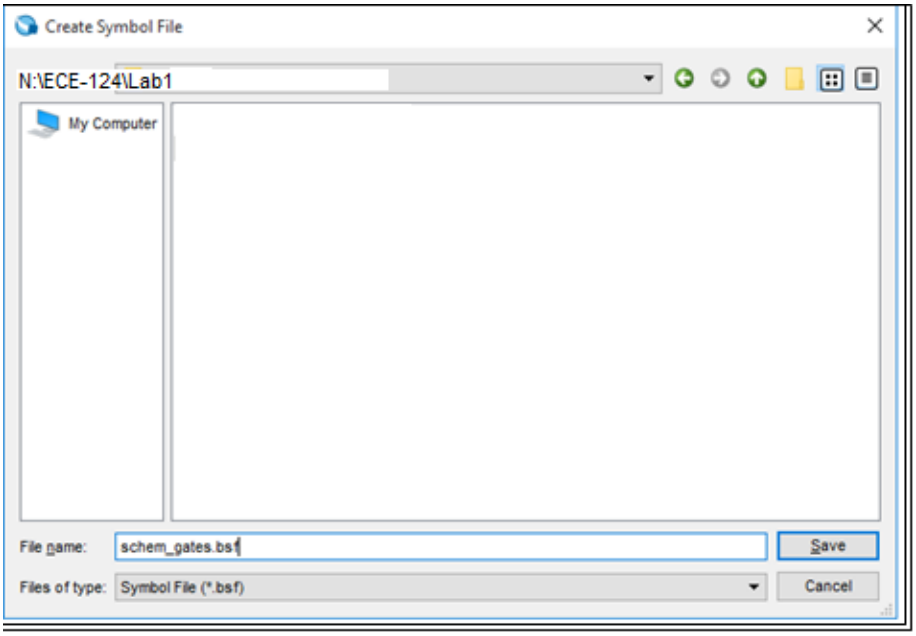


Figure 18 Lab1: Creating Symbol for schem_gates Block

Close the schem_gates design file. Return to the LocialStep_Lab1_top schematic as in Figure 19.

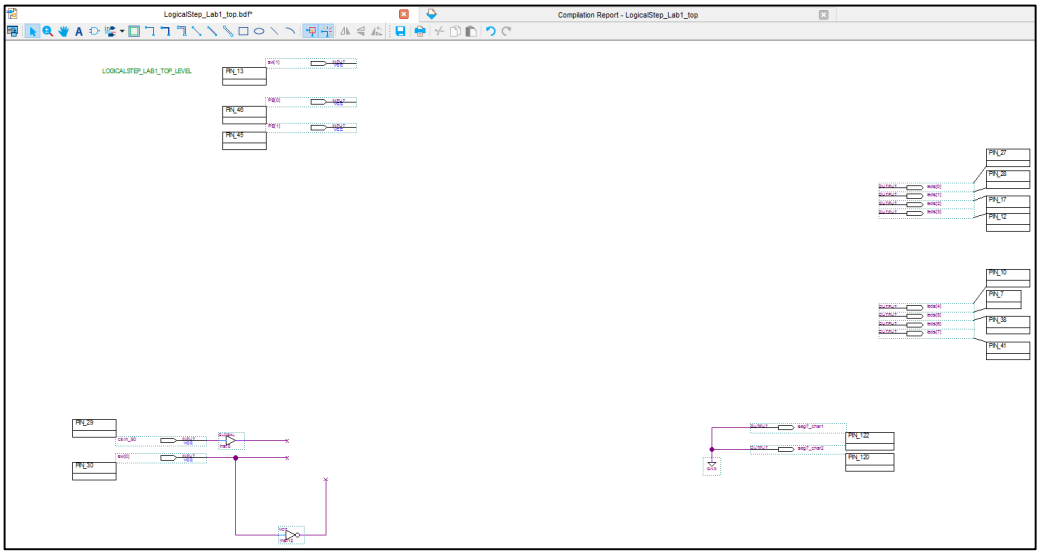


Figure 19 Lab1: Top Level Schematic Before Adding Symbols

On the LogicalStep_Lab1_top schematic design the new schem_gates.bsf symbol (block that was just created in the previous step) will be inserted. To insert a symbol RIGHT-CLICK anywhere on the top level schematic. SELECT the Insert>Symbol option.

The Symbol Dialog box appears (see Figure 20). Expand the Project folder and browse to the schem_gates file.

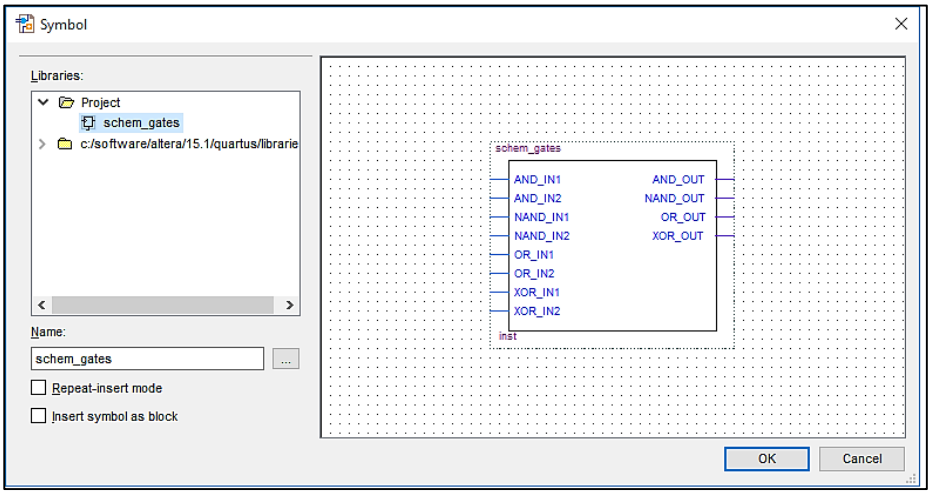
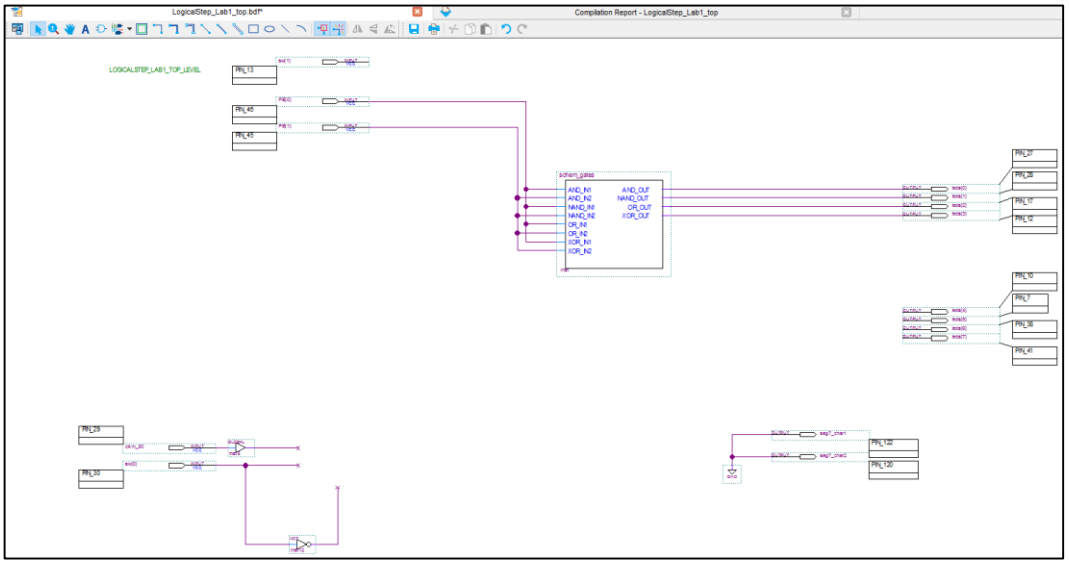


Figure 20 Lab1: Selecting the schem_gates Symbol for Insertion

Click the OK Button.

Place the symbol on the schematic.

After it is added to the top level schematic sheet connect its symbol pins to the Push-Button port pins and to the output pins that drive LED's on the LogicalStep board. Use the ORTHOGONAL NODE TOOL as before.



SELECT File>Save to save the top schematic design as in Figure 21.

Figure 21 Lab1: Hooking Up Pins to schem_gates Block

Now we are ready to do some functional testing by simulation.

2.4.2 Functional Simulations

2.4.2.1 Preparation for Functional Simulation – Analysis and Synthesis

Within Quartus go to the PROCESSING Tab. SELECT Start>Start Analysis and Synthesis" process to process the design into a gate level logic file for simulation purposes.

2.4.2.2 Opening a Simulation Window

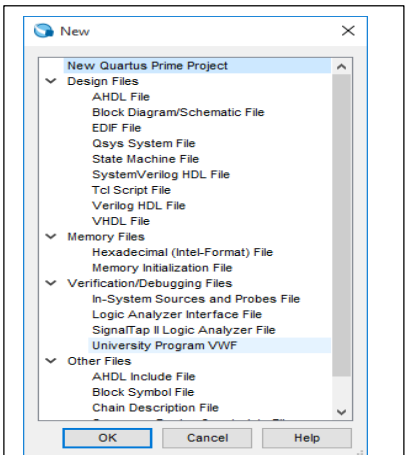


Figure 22 Lab1: Starting a New Simulation

Now under the FILE Tab SELECT the FILE>New and then SELECT the University Program VWF utility to open a Functional Simulation window (see Figure 22). This will be used to illustrate the gate-level functionality in a visual manner. Nodes (nets) will be inserted from the design (see below) for control and observation.

Click OK.

A new window will open like the one shown in Figure 23.

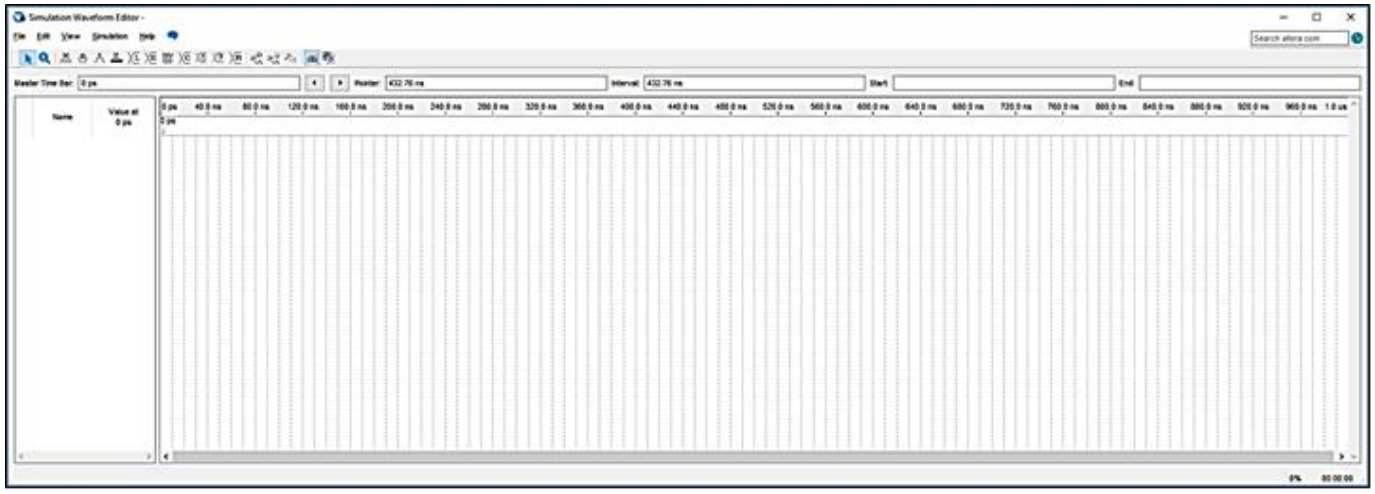
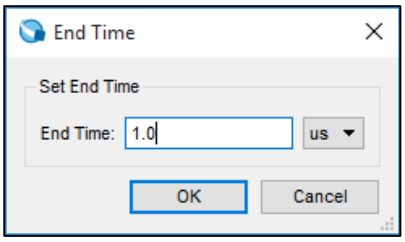


Figure 23 Lab1: Simulation Window

Set the Time scale by going to the Simulator Window EDIT Tab and SELECT the Edit >End Time option.

Then a window like the one shown in Figure 24 will appear.



Set it to 1 usec. Click the OK button.

Figure 24 Lab1: Setting Simulation End Time

2.4.2.3 Adding Nodes to the Simulation Window

For the simulation only the two Push-Button inputs and the first four LED outputs will be inserted into the simulator.

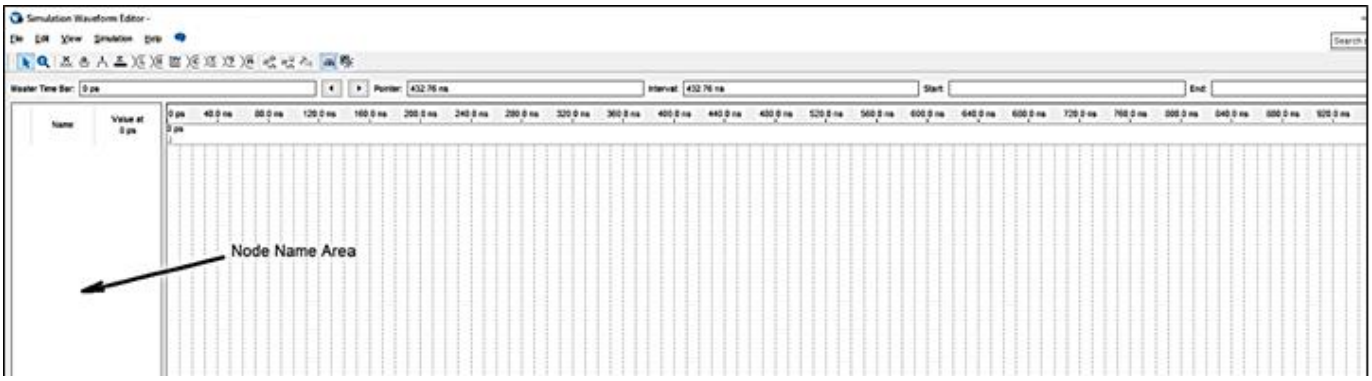
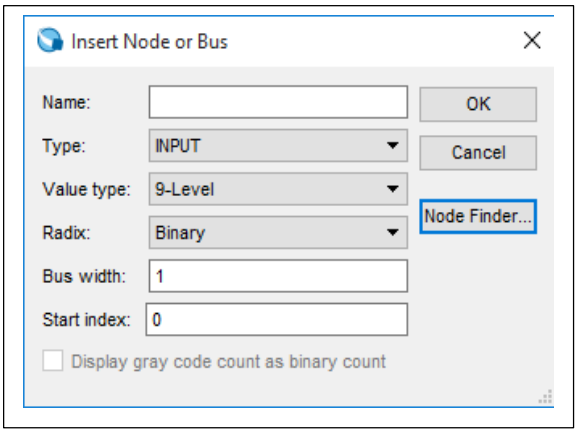


Figure 25 Lab1: Adding Nodes to Simulator Window

Double-Click the Node NAME area of the Simulator Window and the following Dialog window will appear as in Figure 26 below:



Click on the Node Finder Button for faster node identification and insertion. The Node Finder Dialog Box will appear. This will allow you to browse the synthesized design for nodes (nets) to probe for the Functional Simulation.

Figure 26 Lab1: Calling up Simulator Node Finder

With the node FILTER setting set to "Pins all" Click on the LIST button as shown in Figure 27. The list of pins from the design will appear as shown.

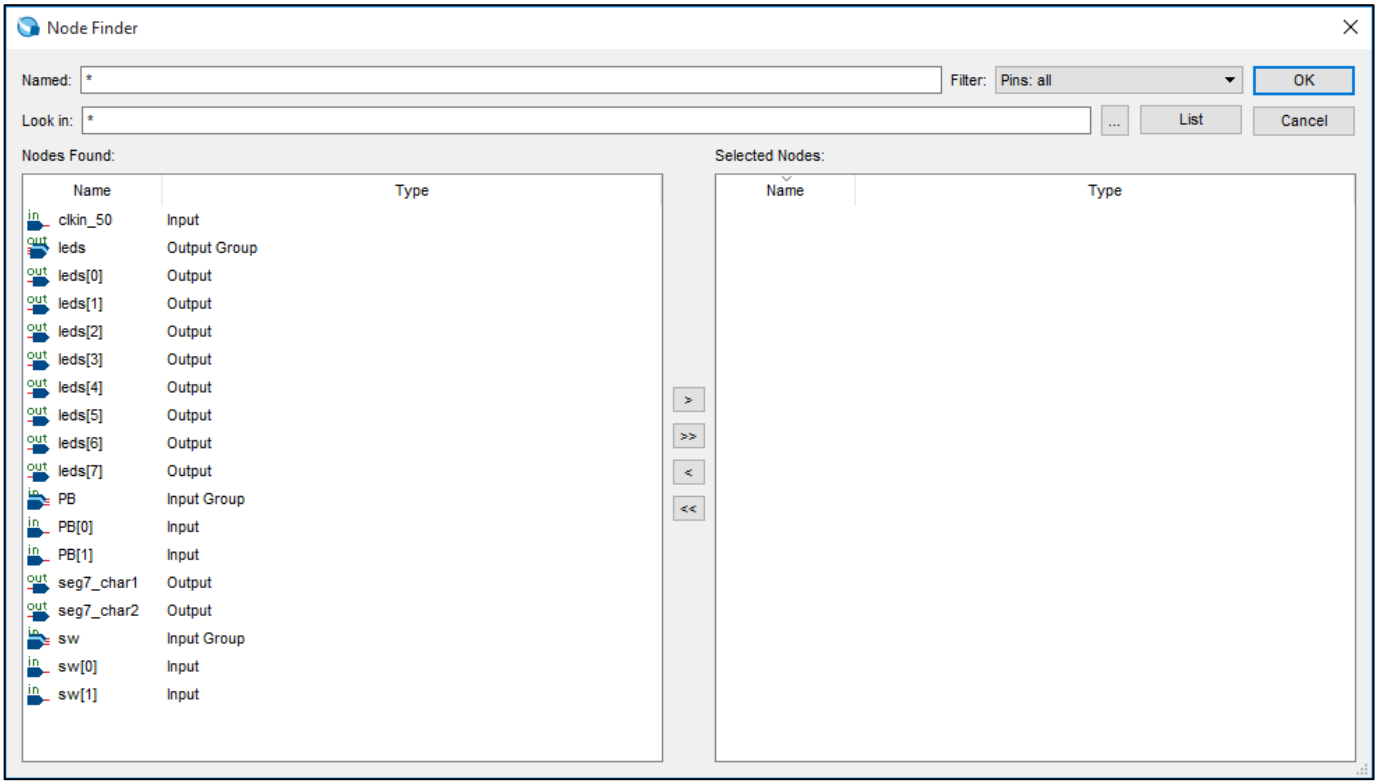


Figure 27 Lab1: Listing Pins with Node Finder

SELECT the following pins **<u>in the order</u>** specified: PB[0], PB[1], leds[0], leds[1], leds[2], leds[3], leds[4], leds[5], leds[6], leds[7]. After selection of the group click on the '>' button to copy them to the Selected Nodes window. Then click on the OK button and again Click on the OK button on the Node_Finder Dialog Box (Figure 26).

2.4.2.4 Adding Stimulus to the Input Nodes

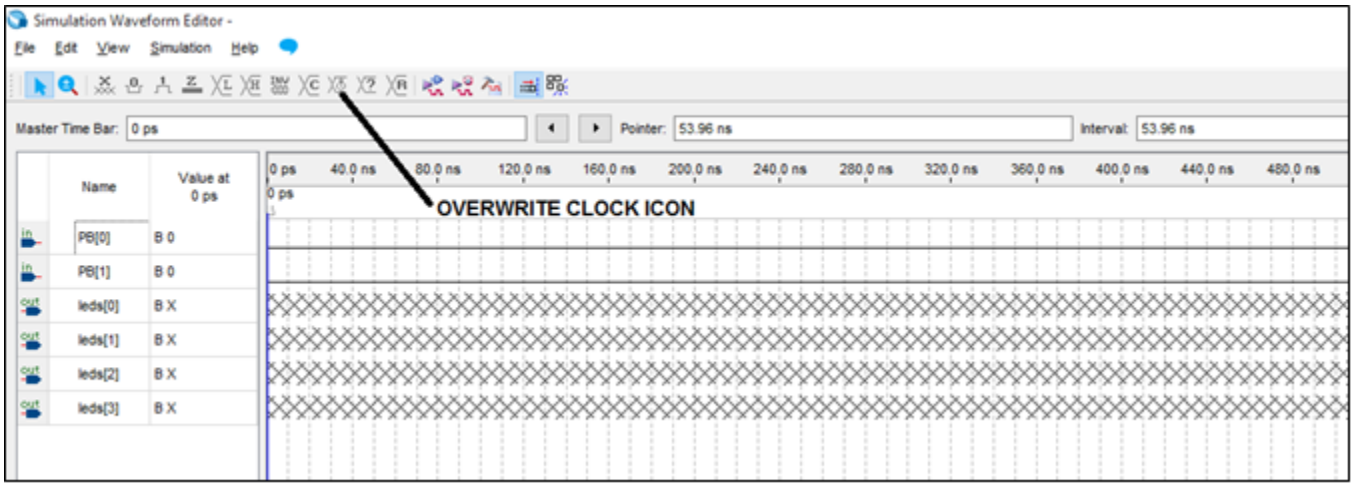


Figure 28 Lab1: Adding Node Stimulus

To provide the stimulus waveforms to the input pins SELECT input PB [0] in the NAME column and then Click on the OVERWRITE CLOCK button (shown above in Figure 28) and enter a period of 500 nanoseconds. Then similarly, for the PB[1] input SELECT the PB[1] in the NAME column and then click on the OVERWRITE CLOCK button (shown above in Figure 29) and enter 1000 nsec for the period value. These two entries should create waveforms for stimulus as shown in Figure 29.

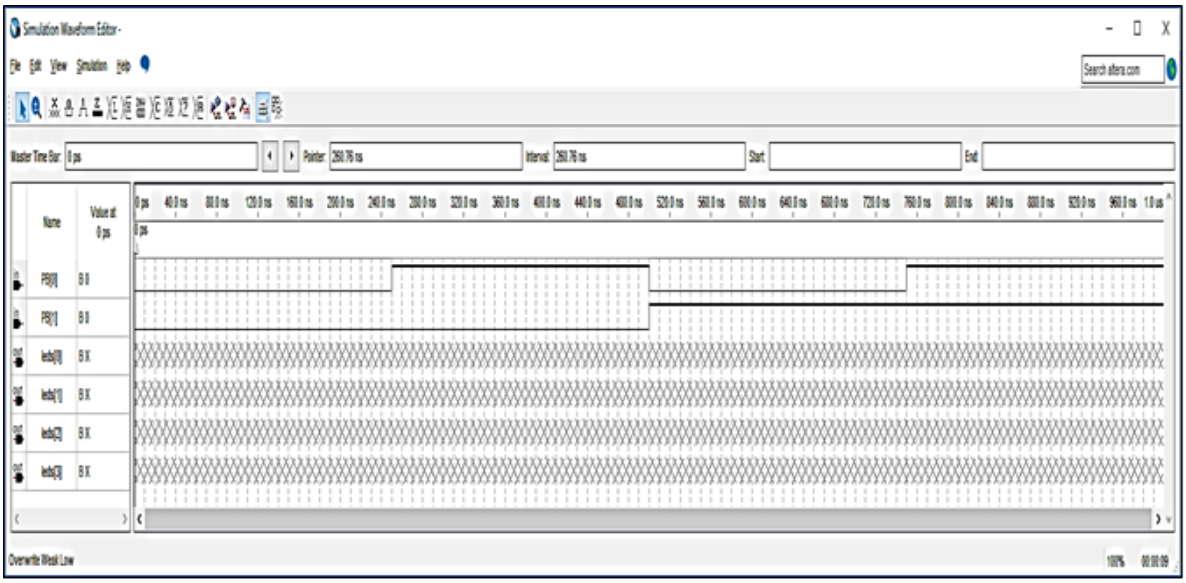


Figure 29 Lab1: Adding more Stimulus

The stimulus is now created. But the outputs are still undefined since the simulation has not yet been run. Save the Simulation file as waveform.vwf by going to the Simulator window FILE Tab and SELECT the File>Save option.

2.4.2.5 Running the Functional Simulation

On the Simulator window SIMULATION Tab and SELECT the Simulation>Run Functional Simulation option. The simulation results should look like Figure 30 when it has completed.

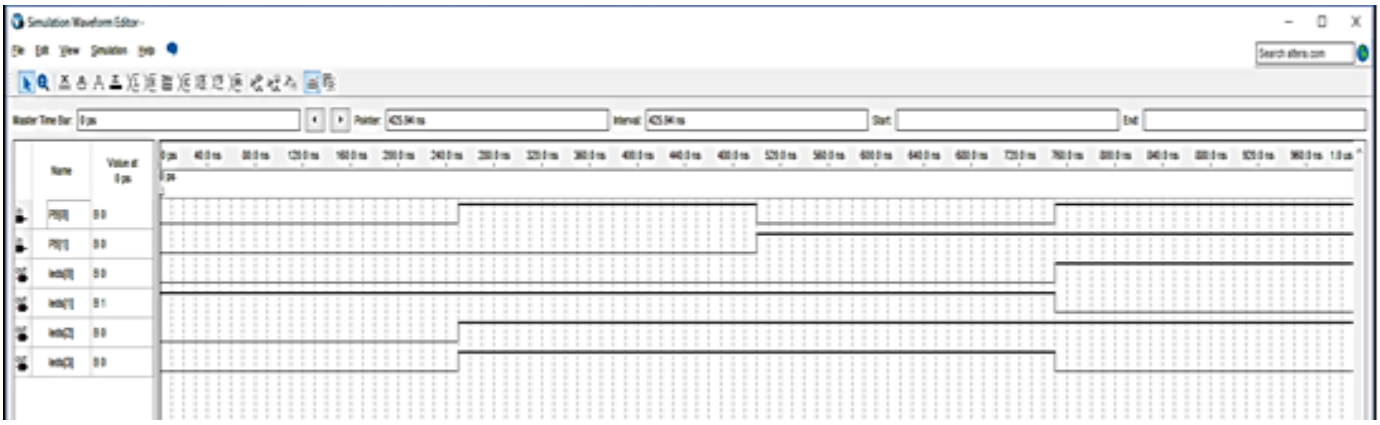


Figure 30 Lab1: Simulation Complete

Recall that the leds[0] pin is connected to the AND_OUT pin of schem_gates in the design. Similarly leds[1] is connected to NAND_OUT, leds[2] with OR_OUT and leds[3] with XOR_OUT. Confirm that the simulation waveforms follow the gate truth tables covered earlier.

2.4.3 Compensating for Active-LOW PB inputs

The functional simulation has proven the design functionality and we can now close the Simulator Windows.

For a more natural real world operation of the FPGA, on the LogicalStep board, there will have to be a small modification to the schematic to adjust for conditions external to the FPGA. The inputs to the logic blocks should match the logic levels that were defined in the simulation. On the LogicalStep board when each PB key is pressed the signal state is ‘0’ for a closed condition. But we want a logical ‘1’ to arrive at the appropriate schem_gates block input when the PB key is pressed.

Therefore we add inverters to PB[0] and PB[1] pin inputs as compensation. Go to the altera/quartus libraries again to insert the “not” gate from the Primitives/logic folder for each of the inverters. Insert and connect them as shown in Figure 31.

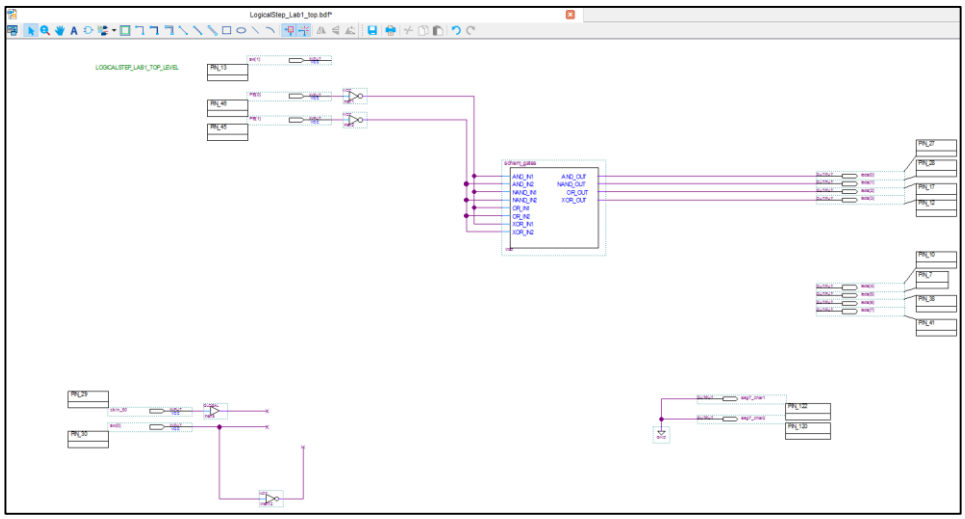


Figure 31 Lab1: Inserting Inverters after PB Key Inputs

2.4.4 FPGA Design Compilation and Download

Now the FPGA Compilation process will be executed. Go to the PROCESSING Tab and SELECT the Processing>Start Compilation option. When the FPGA compilation finishes, without compilation errors (you may ignore any warnings), then an FPGA load file can be downloaded into the FPGA.

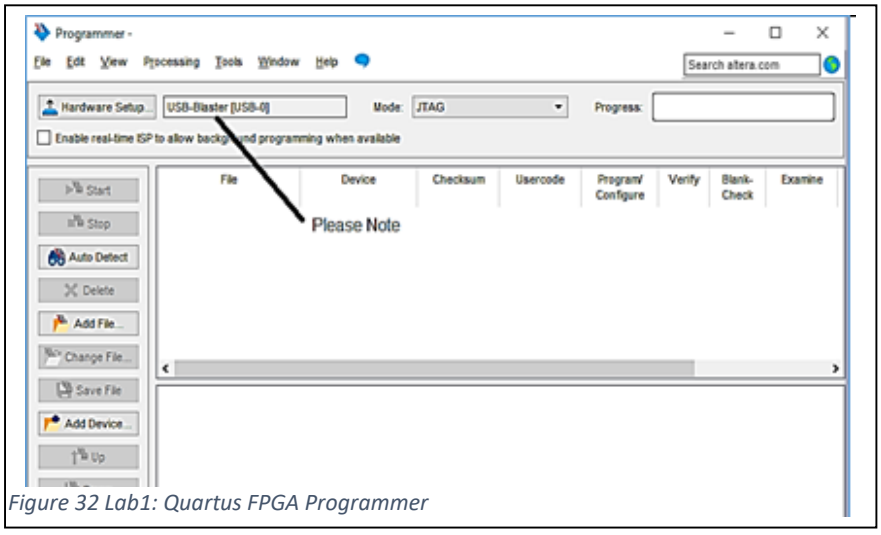


Figure 32 Lab1: Quartus FPGA Programmer

Use the Quartus Programmer to download your design into the LogicalStep board FPGA by going to the TOOLS Tab and SELECT the Tools>Programmer option.
A Programmer dialog window will open as shown in Figure 32.
Click the ADD File button.

NOTE: If the LogicalStep board is connected the **USB Blaster** should be seen beside the Hardware Setup field (Otherwise speak with the Instructor).

A Select Programming File window will open (Figure 33). Browse to the output_files folder.

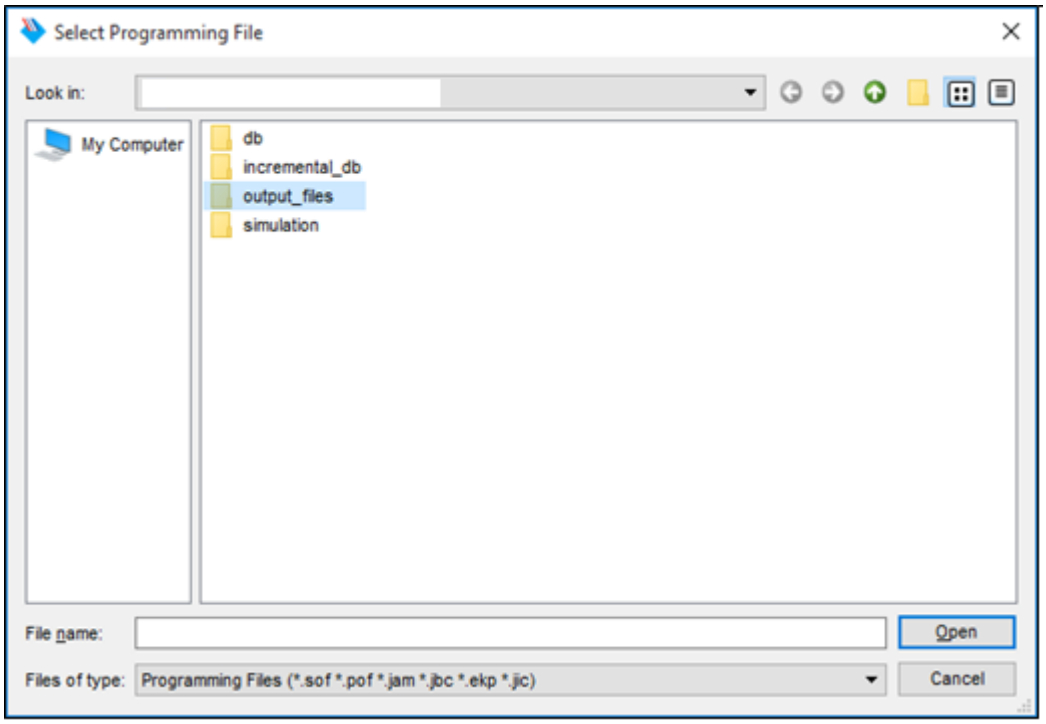
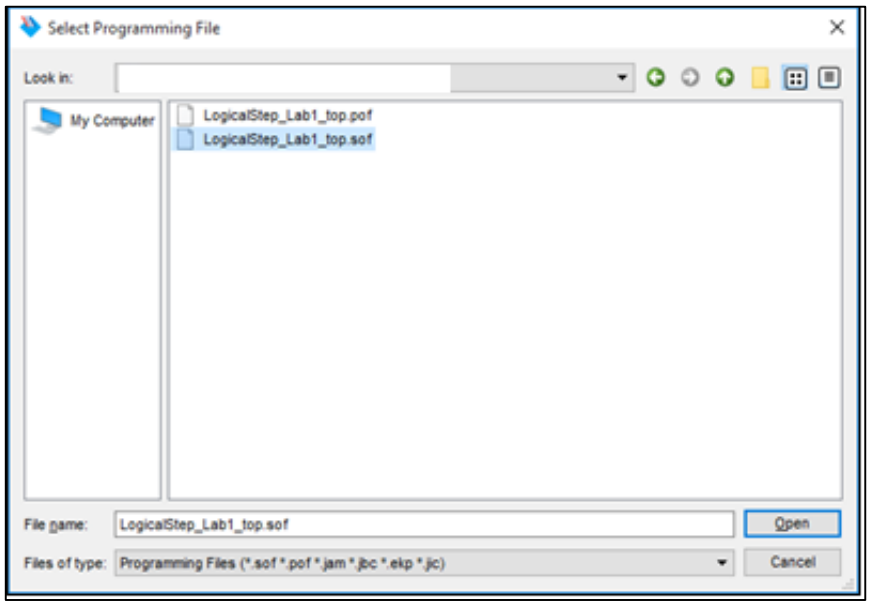


Figure 33 Lab1: Quartus FPGA Programming File Browser

Select the LogicalStep_Lab1_top.**sof file** as shown in Figure 34.



WATCH OUT!

(NOT the .pof file).

Click Open.

Then requested file will then show up in the Programmer window as shown below in Figure 35.

Figure 34 Lab1: Selecting the FPGA Programming File (.sof)

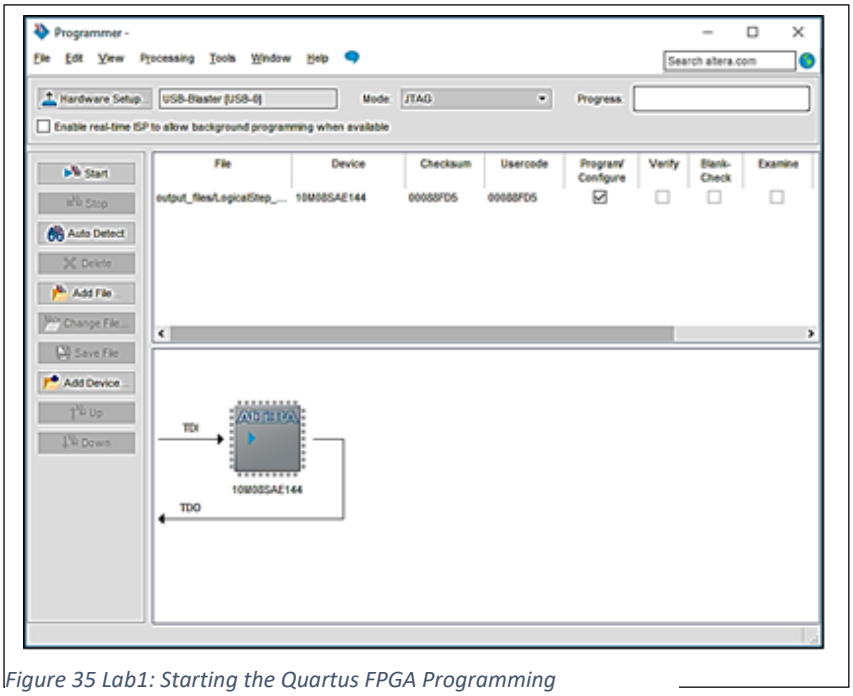


Figure 35 Lab1: Starting the Quartus FPGA Programming

Click on the START button (see Figure 35) to begin the FPGA download. The progress window should indicate 100% after the download is complete.

After downloading you can test your FPGA design by using the PB [1..0] Keys and observing the LED outputs according to your logic gate truth tables.

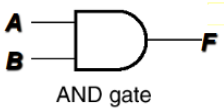
2.4.5 Basic VHDL Design Entry

The VHDL areas that will be covered regarding VHDL in this course are the VHDL design unit (hardware block) pars. The two main areas of focus are the ENTITY and ARCHITECTURE constructs. (Library declarations are also required, as shown in Figure 36 below but just a few variants of these library declarations will be provided to you for use):

1. ENTITY: declares the design unit name and the ports (the inputs and outputs of the entity or design unit) associated with it. Each port name, type (input or output) and width (number of bits) is declared in the entity.

2. ARCHITECTURE: specifies the actual functionality of the entity. Notice that the entity has no information about how the hardware block uses the inputs or how to produce the outputs - that is the role of the architecture associated with the entity.

   Figure 36 is an example of a complete VHDL unit for a two input AND gate:



AND gate

```
-- Library Declaration
library IEEE;
use IEEE.std_logic_1164;

-- Entity Declaration
entity AND2 is
  Port (A,B : in     std_logic;
        F   : out    std_logic);
end And2;

-- Architecture Declaration
architecture dataflow of and2 is
begin
    F <= A and B;
end architecture dataflow;
```

These are some of the standard VHDL libraries that are common to VHDL.

VHDL is case InSenSiTiVe. Here the VHDL file unit is declared to have the name AND2

The entity port names and directions for the inputs and outputs are declared.
std_logic is one kind of data type defined in the libraries above

NOTE how the semicolons are used to indicate the end of statements

The name of the architecture section here is called dataflow but any name could be used.

The Architecture section must be referenced to the entity section named and2.

In the ARCHITECTURE statement the function is defined. F is assigned the function of A and B.

Figure 36 Lab1: VHDL Example for a Simple AND Gate

Within the Architectural section there are three styles generally used to describe the functionality of a VHDL design unit. These are:

1. Dataflow: where the relation between input and output is declared using logical equations.
2. Structural: where you can use previously created entities in your design unit as components. For example if you built an adder unit you can use it as a component in designing a microprocessor.
3. Behavioral – where the VHDL design is described at a high level of abstraction (ie: algorithmic).

For Lab1 we will be using the DATAFLOW style. The Structural and Behavioral varieties will be covered later in the course.

The VHDL design entry method within Quartus will now be covered. Similar to how we created the design for the schem_gates block we will now create a VHDL design block.

Returning to the LogicalStep_Lab1_top design in Quartus go to the FILE Tab and Select File>New.

The dialog box shown below in Figure 37 will open:

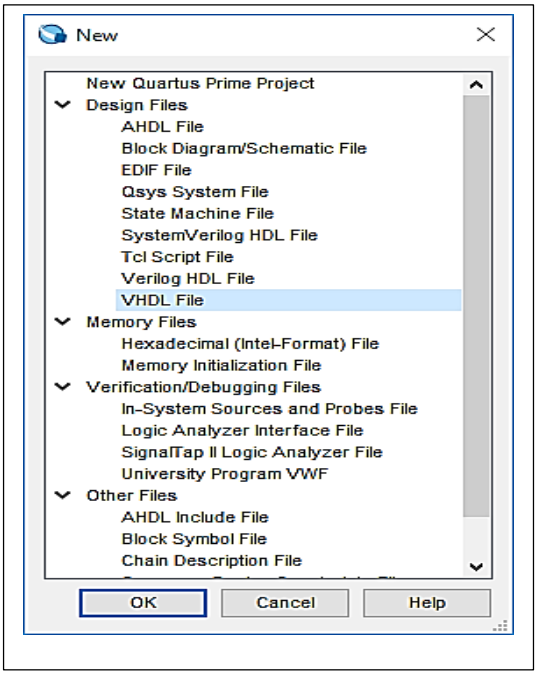


Figure 37 Lab1: Starting a VHDL Design Entry File

SELECT the VHDL File option. A blank VHDL window will then open in Quartus.

Save this VHDL file as "VHDL_gates.vhd" by going to the FILE Tab and SELECTING the File> Save As option.

This VHDL design file is to be an exact functional replica of the schem_gates circuit that was done earlier (for easier comparison during the demo) but entered with VHDL coding.

You must enter the all of the VHDL code shown below in Figure 38 and then fill in the VHDL coding in the ARCHITECTURE section for the remaining gate function types (same as "schematic_gates").

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY VHDL_gates IS
        PORT (
                AND_IN1, AND_IN2, NAND_IN1, NAND_IN2, OR_IN1, OR_IN2, XOR_IN1, XOR_IN2 : IN STD_LOGIC;
                AND_OUT, NAND_OUT, OR_OUT, XOR_OUT : OUT STD_LOGIC
                );
END VHDL_gates;

ARCHITECTURE simple_gates OF VHDL_gates IS
BEGIN

AND_OUT <= AND_IN1 AND AND_IN2;
NAND_OUT <=  _______;
OR_OUT <=  _______;
XOR_OUT <=  _______;

END simple_gates;
```

Figure 38 Lab1: Initial VHDL_gates File (Dataflow style)

Save the VHDL file by browsing to your **Lab1 project folder** and Save the VHDL_gates.vhd file and just leave the file active (current). Create a schematic block symbol for the VHDL_gates design.file using File>Create / Update>Create Symbol files for Current File. Save the VHDL_gates.bsf symbol.

Go back to the top level schematic design to insert ("instantiate") the new VHDL_gates symbol as in Figure 39. Recall that to select the symbol RIGHT-CLICK anywhere on the top level schematic again and SELECT the Insert>Symbol option. Browse to the Project folder in the Symbol Window and Select the VHDL_gates symbol and Click OK.

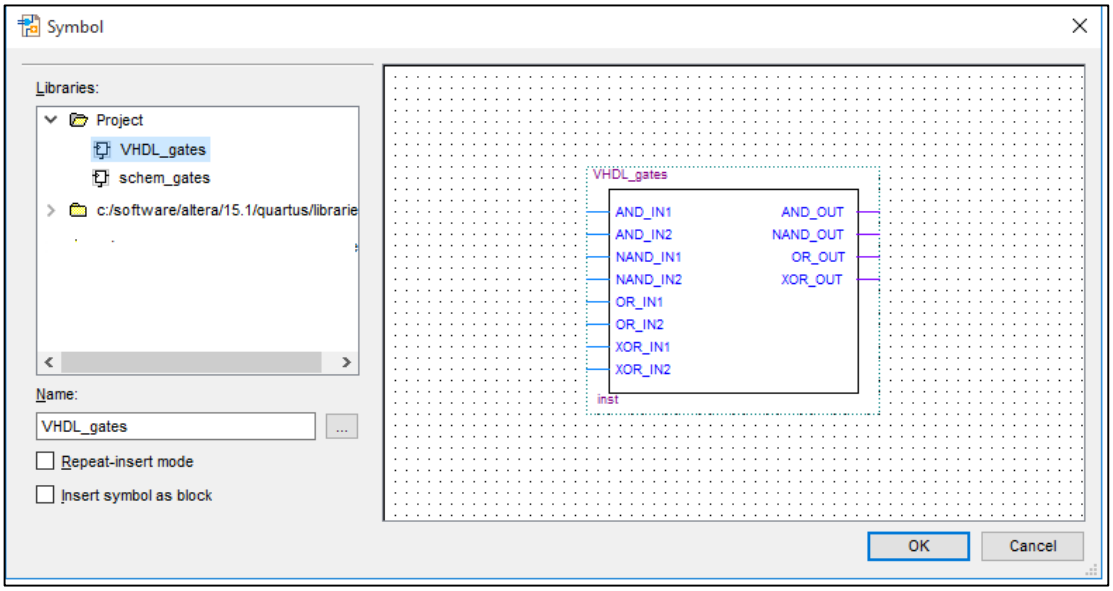


Figure 39 Lab1: Selecting the VHDL_gates Symbol for Insertion

Connect the VHDL block input to the same input connections and connect the VHDL block outputs to the other remaining LogicalStep board LEDs as shown in Figure 40 by using the ORTHOGONAL NODE TOOL as before.

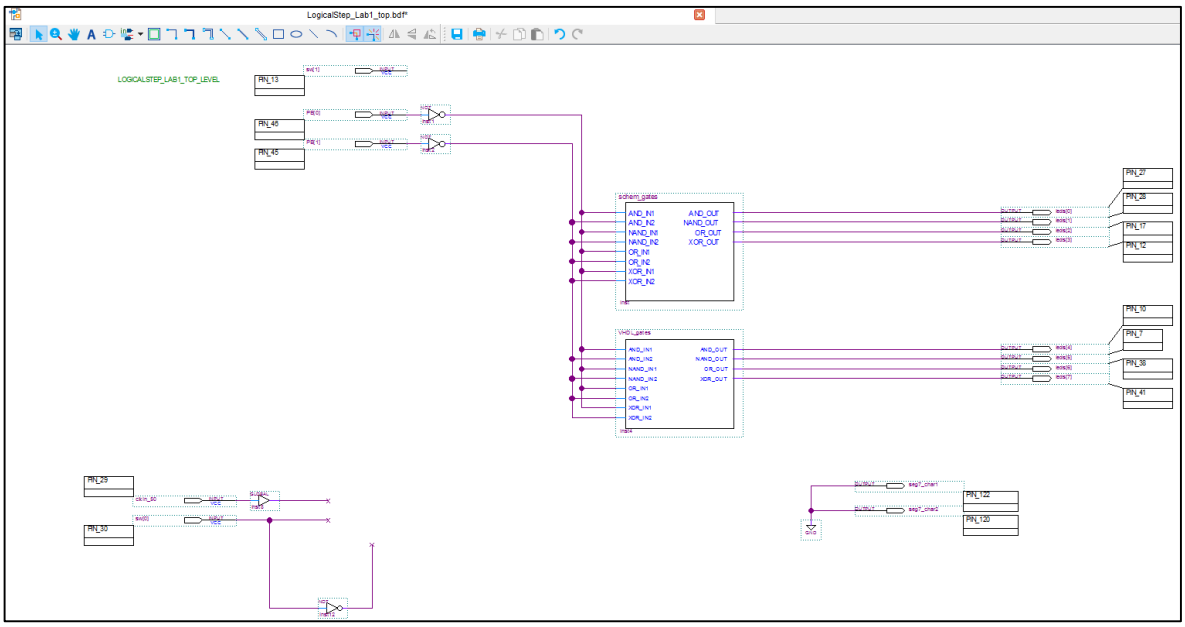


Figure 40 Lab1: Adding Connections to VHDL_gates

Save the design and run a FULL compile of the new FPGA design with Processing>Start Compilation. Download the new FPGA design to the FPGA with the programmer and then test with the PB[1..0] keys. Confirm that both the schematic and VHDL implementations work the same by observing the patterns on the two sets of LED outputs.

2.4.6 Adding Some Automation

As a next step to the LogicalStep_Lab1_top design add an LPM_counter from the library in (altera/quartus/megafunctions/arithmetic/lpm_counter). The counter will use the 50MHz clkin_50 input pin signal to increment. The counter is being added to automate the operation of the PB keys and to slow down the logic activity of the schem_gates and VHDL gates blocks so that you can actually see them switching. The parameters for the counter can be observed in Figure 41 below.

Double-Click the LPM_Parameter block and modify its properties.
LPM_MODULUS: 260000000
LPM_DIRECTION: “UP” (include quotes)
LPM_WIDTH: 28
LPM_PORT_UPDOWN: “PORT_UNUSED” (include quotes)

Connect the LPM_Counter clock input (pin with a “I>” on the left side of the symbol) to the GLOBAL buffer that is used by the CLKIN_50 input pin using the Orthogonal Node Tool as before.

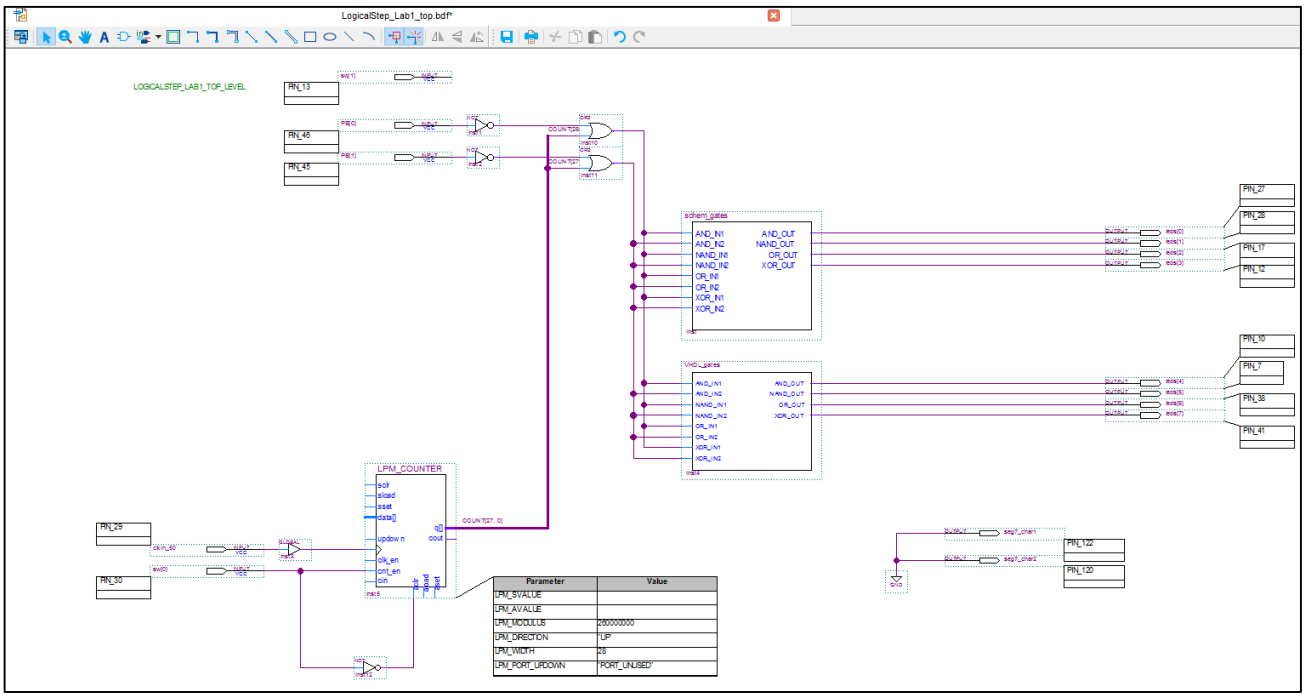
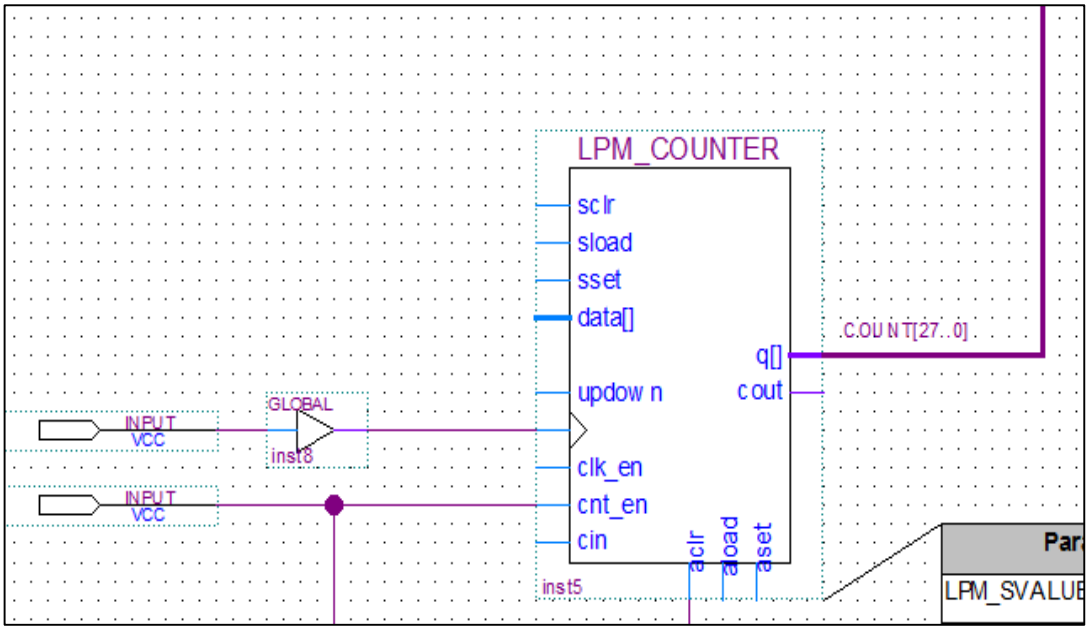


Figure 41 Lab1: Adding Automation to LogicalStep_Lab1_top Design

Next connect the counter “cnt_en” inputs to the sw[0] input pin. Also make sure to connect the sw[0] INVERTER to the counter “aclr” pin. These two connections will turn the counter on and off.

Disconnect the two PB inverter outputs from the schem_gates and VHDL_gates block inputs. Insert and connect a single, two-input OR gate to each of those inverter outputs. Then connect the OR gate outputs back to the wires connected to the schem_gates and VHDL_gates block inputs.



Using the Orthogonal **BUS** tool (icon is located beside the Orthogonal NODE Tool) connect a bus (a thick wire) to the LPM_Counter “q[]” output (See Figure 42). Draw it to up close to the OR gates. Select this bus and Right-click to change its properties. Label this bus as COUNT[27..0].

Figure 42 Lab 1: Creating a 28 Bit Signal Bus for the Counter Output

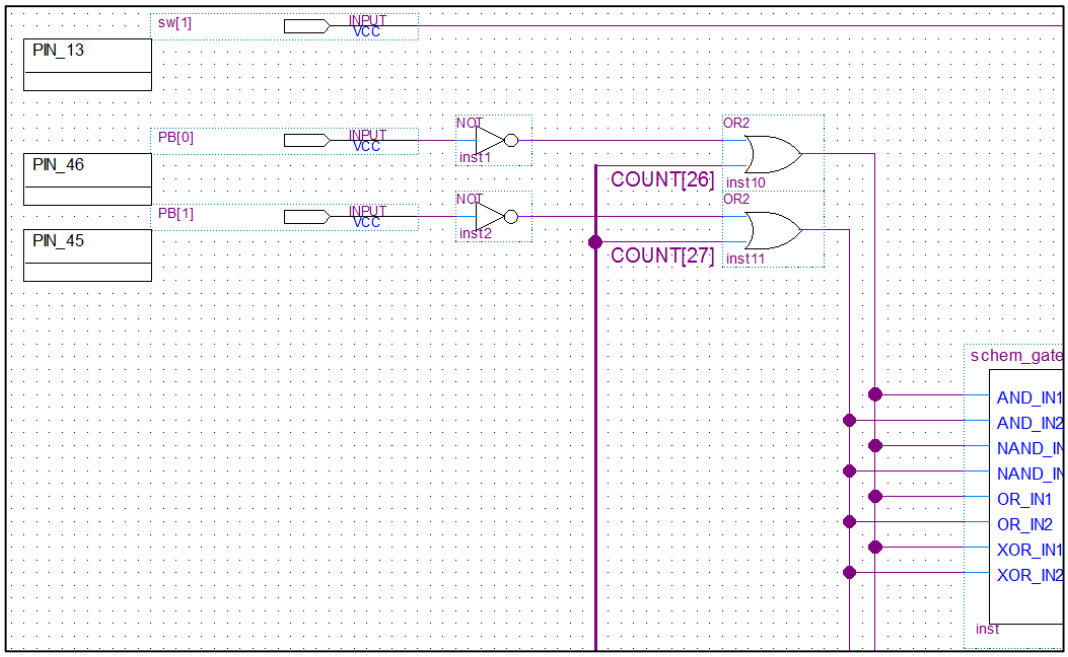


Figure 43 Lab1: Using Bits from the Counter Signal Bus

Using the Orthogonal **NODE** tool draw two thin wires from the new bus (COUNT[27..0]) to the OR gate inputs as shown in Figure 43. Select each of these thin wires and change their properties to label them as COUNT[26] and COUNT[27] as shown.

Why do you think we connect to the two highest COUNT bits from the counter??

We will NOT be discussing the internal functionality of the counter during this lab. We will only be using it as a "generic engine" to automate driving the PB inputs to the design.

Other schematic-based functions can be viewed in the library for your future reference.

So the way this automation should work is:

1) If sw[0] is OFF then the counter does not count and the PB inputs can be used as before.
2) If sw[0] is ON then the counter is enabled and the PB inputs are not required to be manually operated.

Recompile the design (Processing>Start Compilation) and then download the load file (using Tools>Programmer) into the FPGA and then confirm the automatic operation of the blocks on the LogicalStep LEDs.

Again, the VHDL design driven LED's should match the operation of the schematic design driven LEDs.

2.5 POST - Lab1 Activities

For your **DEMO** design in the next Lab session you must add another two blocks (one in schematic form and one VHDL design form) that allows **sw**[1] to be used as an output polarity change control on each of the output pins. Further, a new block function must be entered to use a **single 2**

pin gate per path. (One pin will be connected to the Polarity Control and the second pin will be connected to the upstream schem_gates or VHDL_gates output).

Hint: Draw the schematic circuit first and create the gate truth tables for various 2 input gates in the library to determine what type of 2 input gate to use. The connections are shown below in Figure 44 but with the gates missing.

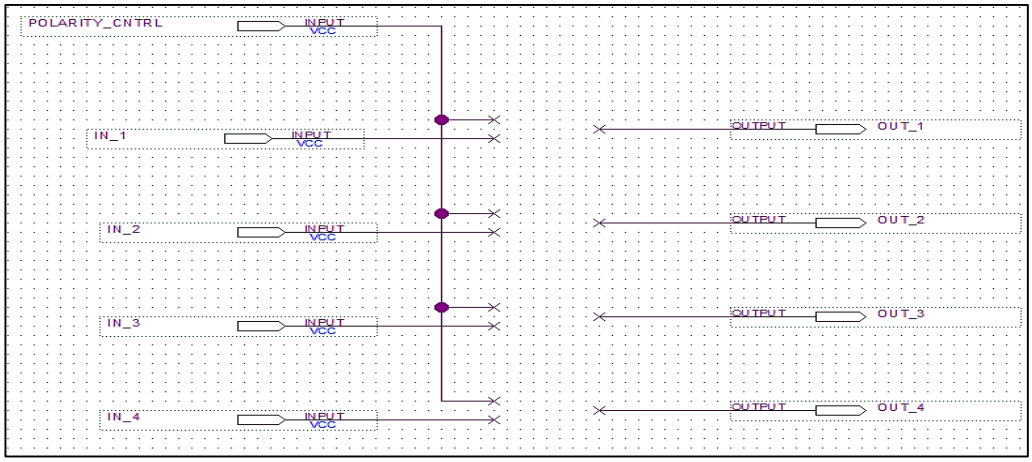


Figure 44 Lab1: Initial Schematic Version of Polarity Control

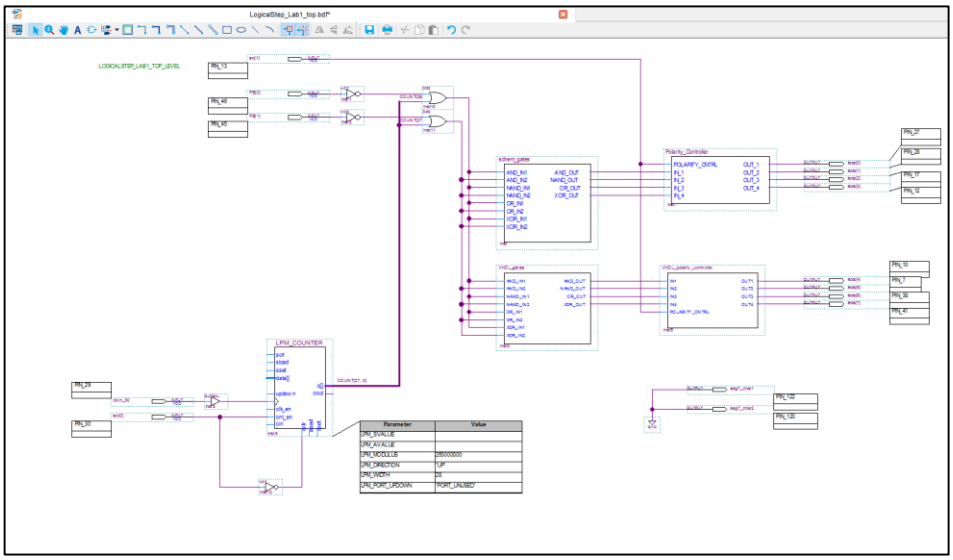
For the VHDL version you may use the following info below in Figure 45:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY _______ IS
        PORT (   -- put your input and output signals here
                 _____, _____, _____, _____, _____,  : IN STD_LOGIC;
                 _____, _____, _____, _____, : OUT STD_LOGIC
                 );
END ENTITY ___________________ ;

ARCHITECTURE _____   OF _______ IS

BEGIN
    ■ Complete this section to implement the required functionality
    ■ using only ONE Boolean gate per output signal. The single 2 pin gate must be
    ■ the same as was used in the Post-Lab schematic for the Polarity Control.
    ■ Refer to the VHDL_gates.vhd file for clues on VHDL syntax.

END ARCHITECTURE _______;
```

Figure 45 Lab1: Initial VHDL File of Polarity Control

For the VHDL version make sure that you save that file with the same name as the declared ENTITY name used.

Create, save and add a schematic block symbol for EACH of your new blocks as before (using File>Create / Update>Create Symbol files for Current File) and add it to the LogicalStep_Lab1_top schematic with the required connections mentioned above.



Your final schematic design will look something like the following in Figure 46.

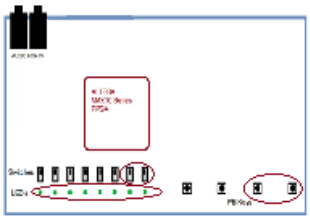


Figure 46 Lab1: Demo Design

Complete the work for the Lab1 Demo design. ie: compile and test it on the LogicalStep board as it will be required at the next lab session. Make sure everything is saved to project folder ("Lab1") on your N: drive. Select the FILE>Close Project option.

There will be no report for this lab but you will need to answer some Lab1 related questions listed on the Lab1 Submission form during your Lab1 DEMO. After the demo, take a picture of the completed Submission form and upload it to the Learn Lab 1 Demo Dropbox.

IN THE NEXT LAB SESSION: We will get into a more challenging logic designing where we learn and build functions that are used in simple logic designs today.

2.6 LAB1 SUBMISSION FORM

Table 1 - Lab1: Submission Form

ECE-124 Lab-1 Submission Form			
GROUP NUMBER:	Lab1 Demo	Lab1 Quiz	**TA:**
LAB SECTION: 20_	Out of 5	Out of 5	

I am submitting this report for grading. I certify that this report, including any code, descriptions, flowcharts as part of the submission were written by the team member(s) below and there has not been any use of prior academic credit at this university or any other institution. The penalty for plagiarism or submission without signature(s) will be a grade of zero

NAME: (Print)	**UW User ID (not Student ID)**	**Signature**
Partner A:		
Partner B:		

LAB1 DESIGN DEMO	**Marks Allotted**	**A**	**B**
With BOTH sw[0], sw[1] OFF and using PB[1..0] verify AND, NAND, OR, XOR of schem/VHDL gates driven LEDs	1		
Verify with sw[1] ON, sw[0] OFF that the LED's invert when using the previous step of AND, NAND, OR, XOR	1		
With sw[1] OFF and sw[0] ON verify the LED SEQUENCE on LEDS[3..0] together with LED's[7..4]: 0010, 1110, 1110, 0101	2		
With sw[1] ON and sw[0] ON verify the LED SEQUENCE on LEDS[3..0] together with LED's[7..4]: 1101, 0001, 0001, 1010	1		
LAB1 DEMO MARK	**Out of 5**		

LAB1 QUIZ	**Marks Allotted**	**A**	**B**
Why were inverters added after the PB[1..0] inputs?	1		
Name one typical development process used in an FPGA design.	1		
What are the two main components of a VHDL design file?	1		
What style of coding was used in the Lab1 VHDL Architecture section?	1		
For your Polarity Control block what kind of gate was used?	1		
THINK 1:…… For automation we added a counter to the 50 MHz clock input. The highest bits (COUNT[27..26]) of the counter selected to drive the logic block operations. Why? (no marks)			
THINK 2:…… What change(s) could make the LED's flash faster? (no marks)			
LAB1 QUIZ MARK	**Total** out of 5		

3 Lab 2 – VHDL - Combinational Circuits 1– Simple ALU (Dataflow / Structural VHDL)

The primary goal of this lab session is to gain more experience with VHDL for combinational logic design. Some new VHDL components will be introduced along with their associated data format requirements. There are some logic errors "planted" into the Seven Segment decoder that must be identified during simulation and then corrected to meet the Lab2 project requirements. The final design will be demonstrated during the next Lab Session and a report submitted on LEARN within 24 hours of your Lab2 DEMO.

3.1 Lab2 Intended Learning Outcomes

By the end of this lab students should be able to:

1) **IMPLEMENT** VHDL design units (entities,components and architectures)
2) **APPLY** VHDL Components into new Structural VHDL designs
3) **UNDERSTAND** basic VHDL Dataflow (Decoder and Multiplexer) designs
4) **DEBUG** a digital design using simulation

3.2 Prelab

1. Review the Lab1 processes used for entering, testing and implementing FPGA designs.
2. Review the Lab1 Submission form from LEARN for the Demo during the Lab2 session.
3. Have your Lab1 design available for demonstration.

3.3 Lab2 Outline

Attendance will be taken.

The lab starts with a brief review of the design entry methods used in Lab1 and some VHDL. The following new topics will be presented:

1 Recalling some parts of a VHDL Design
2 Design Re-use in VHDL – with Structural coding style
3 Project Setup for Lab2
4 New VHDL Component - What is a Hex to Seven Segment Decoder?
5 Lab2 Part A. – Hunting for "Bugs".
6 New VHDL Component - What is a Multiplexer or MUX function?
7 Lab2 Part B. – Using the Seven Segment Display
8 Lab2-Part C- Project Brief for Lab2 Demo

3.4 Lab2 Activities

3.4.1 Recall from Lab1:

In Lab1 a basic gate circuit was programmed into the FPGA. We briefly explored some of the tools and utilities available within the Quartus FPGA development environment. The top level design was schematic based. A subordinate block in schematic form was developed and added into the hierarchy. Then the design was "synthesized" to logical gates representation so that a functional simulation could be done. A functional simulation STIMULUS was and the simulation results were compared against the truth tables of the functionality implemented.

Having completed a functional verification of the schematic design entry with simulation, a full design COMPILATION was run so that a SOF file could be created for loading into the LogicalStep board FPGA. It was confirmed by observing the LED patterns that the schematic entry design worked in hardware.

A functionally equivalent VHDL design was created and added to the top level of the design. The larger FPGA design was compiled and downloaded to the LogicalStep board FPGA. Using the LEDs it was confirmed that the VHDL and schematic design versions behaved identically.

Later a type of hardware-based automation was added so that manual exercising of the hardware inputs was not required. As a final design addition for the DEMO, an output polarity control was added (one in schematic and one in VHDL form) so that the output operation could be run in two different modes based on an external Push-Button Key input.

3.4.2 Recalling Some Parts of a VHDL Design

The VHDL language uses two main parts to describe a design unit (hardware block):

1. Entity declaration: declares the design unit name and the ports
2. Architecture description: implements the actual functionality of the entity.

The Entity portion is used to define the inputs and outputs and the signal types.

The Architecture portion has further details presented below.

3.4.3 Lab2 VHDL – Architecture Styles

For Lab2 there are just two VHDL coding styles being used in the Architecture section:

a) Dataflow: where the relation between inputs and outputs are declared using logical equations
b) Structural: where you use previously created entities in your design as components

Lab1 was simple in scope and it used the simpler DATAFLOW style of VHDL coding with basic Boolean equations with two-input gates. Lab2 will use that and add the STRUCTURAL style of VHDL.

Quite often a VHDL design is constituted from smaller VHDL blocks connected together to form a more complex VHDL function. Using a hierarchy of VHDL blocks in this manner is the STRUCTURAL VHDL style.

Before we begin to use Structural VHDL we first have to understand the Component declaration. It looks like an Entity declaration (see Figure 47). See an example Entity syntax below in the VHDL file called VHDL_gates. A companion Component declaration in another file that could use the VHDL_gates file. They are very close in syntax content and the port names in the Component declaration must match those defined in the Entity declarations of the VHDL file being used.

```
ENTITY VHDL_gates IS
PORT
   (
      AND_IN1, AND_IN2,NAND_IN1, NAND_IN2, OR_IN1,OR_IN2, XOR_IN1, XOR_IN2 :  IN  BIT;
      AND_OUT, NAND_OUT, OR_OUT, XOR_OUT :  OUT  BIT
   );
END VHDL_gates;
```

```
component VHDL_gates
port (
      AND_IN1, AND_IN2,NAND_IN1, NAND_IN2, OR_IN1,OR_IN2, XOR_IN1, XOR_IN2 :  IN  BIT;
      AND_OUT, NAND_OUT, OR_OUT, XOR_OUT :  OUT  BIT
      );
end component;
```

Figure 47 Lab2: Component Structure are Similar to Entity Structures

After a Component is declared inside a VHDL architecture there is still the signal hook-up to its interfaces to be done. For example if we were to write the LogicalStep_Lab1_top as a VHDL file it could use the previously designed VHDL_gates file as a component. New signal declarations would be required for the internal VHDL component linking.

The LogicalStep_Lab1_top could look something like that shown in Figure 48 with the Component INSTANCES added in the bottom section. Note the “signal” declaration for the pb_bar internal signals.

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

Entity LogicalStep_Lab1_top is
   port (
         pb : in std_logic_vector(1 downto 0);
         leds : out std_logic_vector(7 downto 0);
         seg7_char1,seg7_char2 : out std_logic
         );
end LogicalStep_Lab1_top;

Architecture Structural_VHDL_Example of LogicalStep_Lab1_top is

component VHDL_gates
   port (
         AND_IN1, and_in2, nand_in1, nand_in2, or_in1, or_in2, xor_in1, xor_in2 : in std_logic;
         and_out, NAND_OUT, OR_out, XOR_out : out std_logic
         );
   end component;

   -- add internal signal declarations
signal pb_bar : std_logic_vector(1 downto 0);

begin

pb_bar <= NOT(pb); -- inverters added for PB Key active-low compensation

seg7_char1 <= '0'; -- used for some external signal "disables"
seg7_char2 <= '0'; --

inst1: VHDL_gates port map (
                            pb_bar(0), pb_bar(1), pb_bar(0), pb_bar(1),
                            pb_bar(0), pb_bar(1), pb_bar(0), pb_bar(1),
                            leds(0), leds(1), leds(2), leds(3)
                           );

inst2: VHDL_gates port map (
                            pb_bar(0), pb_bar(1), pb_bar(0), pb_bar(1),
                            pb_bar(0), pb_bar(1), pb_bar(0), pb_bar(1),
                            leds(4), leds(5), leds(6), leds(7)
                           );

end Structural_VHDL_Example;
```

Figure 48 Lab2: VHDL Example of Using Components for LogicalStep_Lab1_top Design

This can be used as a reference for the component instantiation exercises in this lab session.

The signal hook-up is in the bottom section (between the “begin and end” statements). Notice how two signals pb_bar(0), pb_bar(1) were added to do the PB key active-low compensation via adding the inverter function (NOT).

3.4.4 Project Setup for Lab2

Start the LAB2 as was done in Lab1 by creating a new project folder on your N: drive using the Windows File Explorer. Download the Lab2 Zipped folder "Lab2" from LEARN into your ECE-124 folder. Extract the contents into the Lab2 project folder. The new files are shown below in Figure 49.

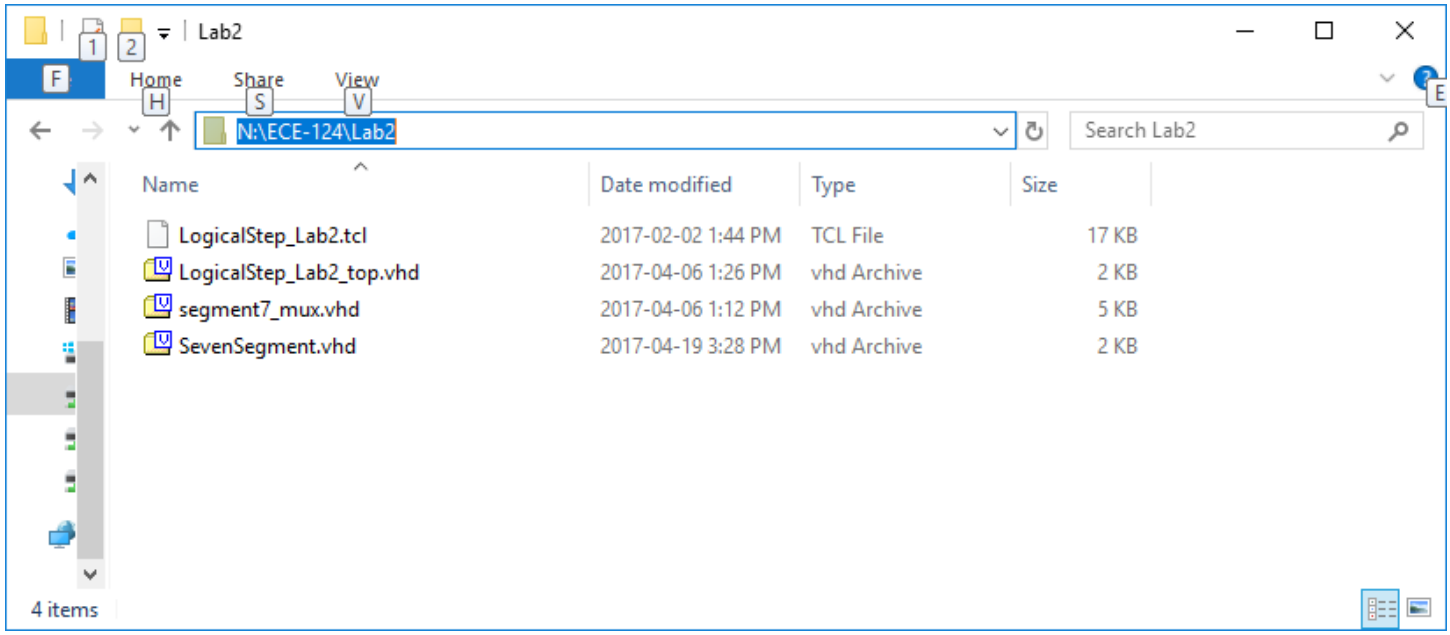


Figure 49 Lab2: Project Folder after Setup

Start up the Quartus Prime platform and begin a new project (Using FILE>New Project Wizard).

Click NEXT to go to the second slide.

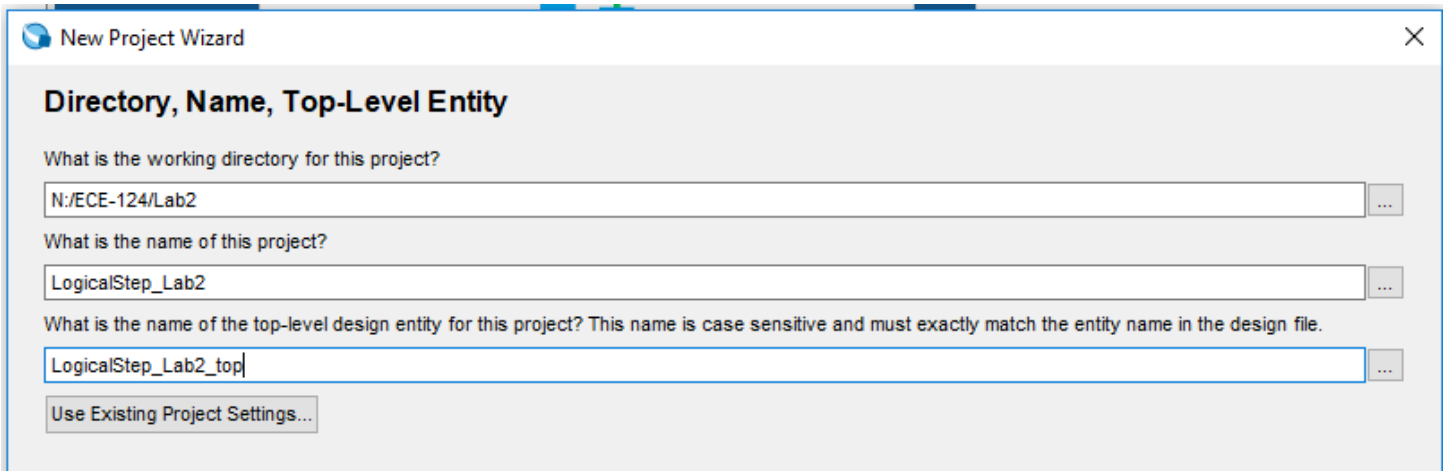


Figure 50 Lab2: FPGA Project Setup

The project parameters will now be entered.

Project Folder: **N:/ECE-124/Lab2**
Project Name: **LogicalStep_Lab2**
Project Top Level: **LogicalStep_Lab2_top**

Click FINISH on the New Project Wizard Dialog Window.
(see Figure 50)

Next, in Quartus, the Lab2 TCL script must be run to assign the FPGA device type, the FPGA pin assignments for the FPGA that are reserved for the LogicalStep FPGA and finally the project LogicalStep_Lab2 is created and opened.

Go to the Tools tab and SELECT the Tcl Scripts option. The dialog box in Figure 51 should appear:

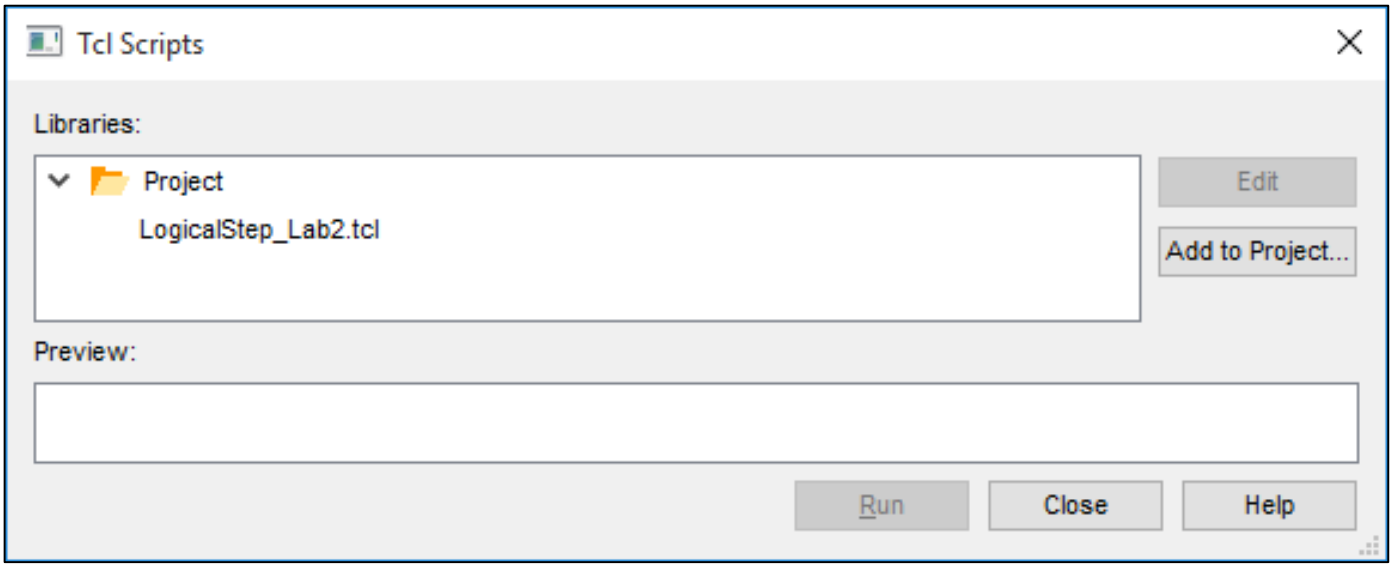


Figure 51 Lab2: TCL Script Invocation

SELECT the TCL file "LogicalStep_Lab2.tcl" and then click on the RUN button as in Figure 51.

The following in Figure 52 should appear when it is finished.

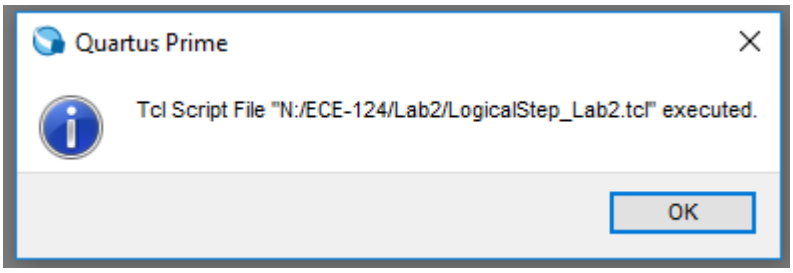
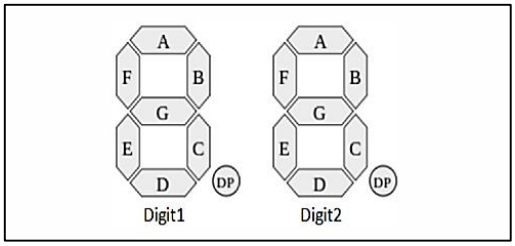


Figure 52 Lab2: TCL File Completed

Click the OK Button and close the TCL Scripts window.

3.4.5 NEW VHDL Component - What is a Seven Segment Decoder?



The LogicalStep board has two seven segment displays available. See Figure 53.

Figure 53 Lab2: LogicalStep Board Seven Segment Displays

To drive each display we typically use a hex to seven-segment decoder. Hex input values (4 bits) are used to represent hex variable values and the decoder converts the 4-bit hex values (the 3210 column below), to a pattern of 7 bits to drive the seven LEDs or segments. A VHDL example of this function is shown in Figure 54 below.

```
begin
  with dataIn select                -- g fedcba          3210    -- D
    7SegmentsOut(6 downto 0) <=     "0111111" when "0000",   -- [0]
                                    "0000110" when "0001",   -- [1]
                                    "1011011" when "0010",   -- [2]    +---- a----+
                                    "1001111" when "0011",   -- [3]    |          |
                                    "1100110" when "0100",   -- [4]    |          |
                                    "1101101" when "0101",   -- [5]    f          b
                                    "1111101" when "0110",   -- [6]    |          |
                                    "0000111" when "0111",   -- [7]    |          |
                                    "1111111" when "1000",   -- [8]    +---- g ----+
                                    "1101111" when "1001",   -- [9]    |          |
                                    "1110111" when "1010",   -- [A]    |          |
                                    "1111100" when "1011",   -- [b]    e          c
                                    "1011000" when "1100",   -- [c]    |          |
                                    "1011110" when "1101",   -- [d]    |          |
                                    "1111001" when "1110",   -- [E]    +---- d ----+
                                    "1110001" when "1111",   -- [F]
                                     "0000000" when others;   -- [ ]
end
```

Figure 54 Lab2: VHDL file Seven Segment Decoder

Question: For this example how hard would it be to build the function with schematic gates?

TAKE-AWAY:
➔A Hardware Description Language (HDL) is often much more efficient than the schematic design entry method.

Earlier the top level design file (LogicalStep_Lab2_top.vhd) was downloaded from LEARN into the Lab2 project folder. This time the top level design file is a VHDL file. Notice in the screen-shot below in Figure 55 that the pins are declared in the Entity section.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5
6   entity LogicalStep_Lab2_top is port (
7      clkin_50    : in  std_logic;
8      pb          : in  std_logic_vector(3 downto 0);
9      sw          : in  std_logic_vector(7 downto 0); -- The switch inputs
10     leds        : out std_logic_vector(7 downto 0); -- for displaying the switch content
11     seg7_data   : out std_logic_vector(6 downto 0); -- 7-bit outputs to a 7-segment
12     seg7_char1  : out std_logic;                    -- seg7 digi selectors
13     seg7_char2  : out std_logic                     -- seg7 digi selectors
14
15  );
16  end LogicalStep_Lab2_top;
17
18  architecture SimpleCircuit of LogicalStep_Lab2_top is
19  --
20  -- Components Used
21  ------------------------------------------------------------------
22     component SevenSegment port (
23
24     hex      :  in  std_logic_vector(3 downto 0);   -- The 4 bit data to be displayed
25
26     sevenseg :  out std_logic_vector(6 downto 0)    -- 7-bit outputs to a 7-segment
27     );
28     end component;
29
30  --
31  ------------------------------------------------------------------
32
33
34  -- Create any signals, or temporary variables to be used
35  --
36  -- Note that there are two basic types and mixing them is difficult
37  --  unsigned is a signal which can be used to perform math operations such as +, -, *
38  --  std_logic_vector is a signal which can be used for logic operations such as OR, AND, NOT, XOR
39  --
40     signal seg7_A           : std_logic_vector(6 downto 0);
41     signal hex_A            : std_logic_vector(3 downto 0);
42
43
44  -- Here the circuit begins
45
46   begin
47
48    hex_A <= sw(3 downto 0);
49    seg7_data <= seg7_A;
50
51
52  --COMPONENT HOOKUP
53  --
54  -- generate the  seven segment coding
55  --
56     INST1: SevenSegment port map(hex_A, seg7_A);
57
58  end SimpleCircuit;
59
60
```

Figure 55 Lab2: Initial VHDL Design of LogicalStep_Lab2_top

In the above example the component for the SevenSegment decoder is already declared as well as two signal busses (hex_A and Seg7_A). Observe (between the “begin” and “end” statements) how the

busses (signal groupings) are connected and how the instantiation of the component is done. YOU must add the above VHDL code to your LogicalStep_Lab2_top.vhd file.

3.4.6 Lab2-Part A – Hunting for "BUGS"

In the Architecture section instantiate a SevenSegment instance and call it INST1. Then connect the four switch inputs (sw[3..0]) to the SevenSegment instance INST1 hex inputs and connect the seg7_data pins to the SevenSegment instance outputs. Refer to Figure 48 in the "Design Re-use within VHDL" section in the Lab Manual as an example. Next run an ANALYSIS and SYNTHESIZE compilation process to allow a functional simulation to be executed. This can be done by going to the Processing TAB and then selecting "Processing>Start>Analysis and Synthesis" option.
WE WILL NOT BE DOWNLOADING THIS DESIGN DUE TO PIN PROPERTY CONSTRAINTS (pin drive settings) AT THIS STAGE OF THE LAB.

There are three "logic errors planted" into a part of the provided SevenSegment.vhd file that must be discovered during functional simulations and then corrected to meet the Lab2 project requirements.

Recall from Lab1 how functional Simulations are set up within Quartus (see section 2.3.2 and Figures 22 to 26). In the Node Finder window in Figure 56 SELECT the following pins **in the order** specified: sw[3], sw[2], sw[1], sw[0], seg7_data[6], seg7_data[5], seg7_data[4], seg7_data[3], seg7_data[2], seg7_data[1], seg7_data[0]

After selection of the group click on the '>' button to copy them to the Selected Nodes window.

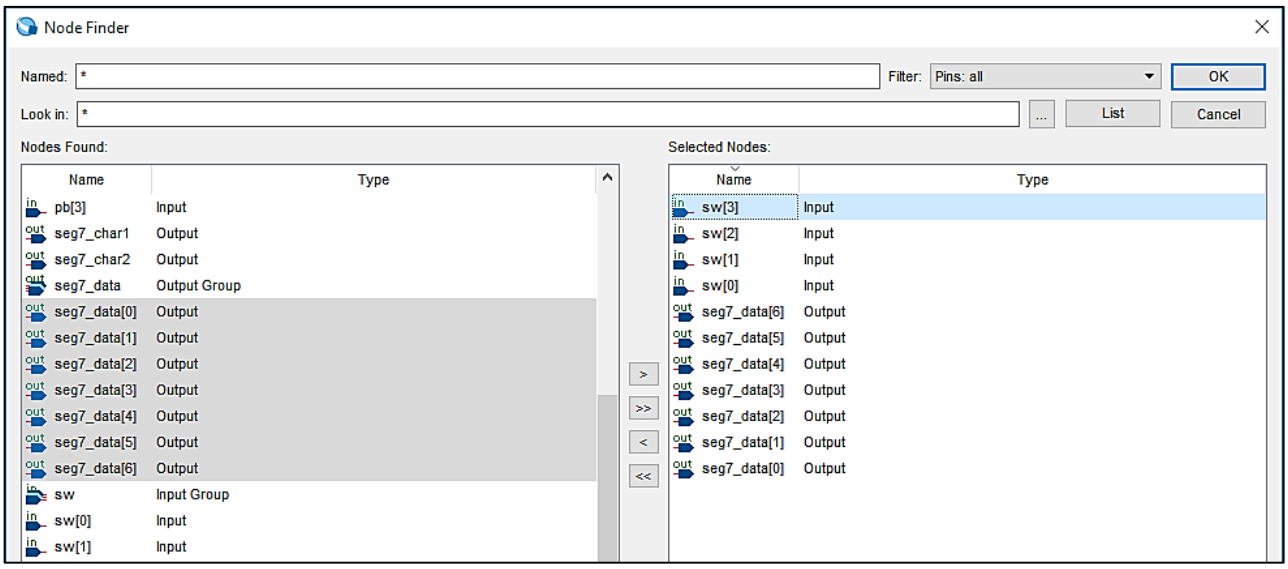
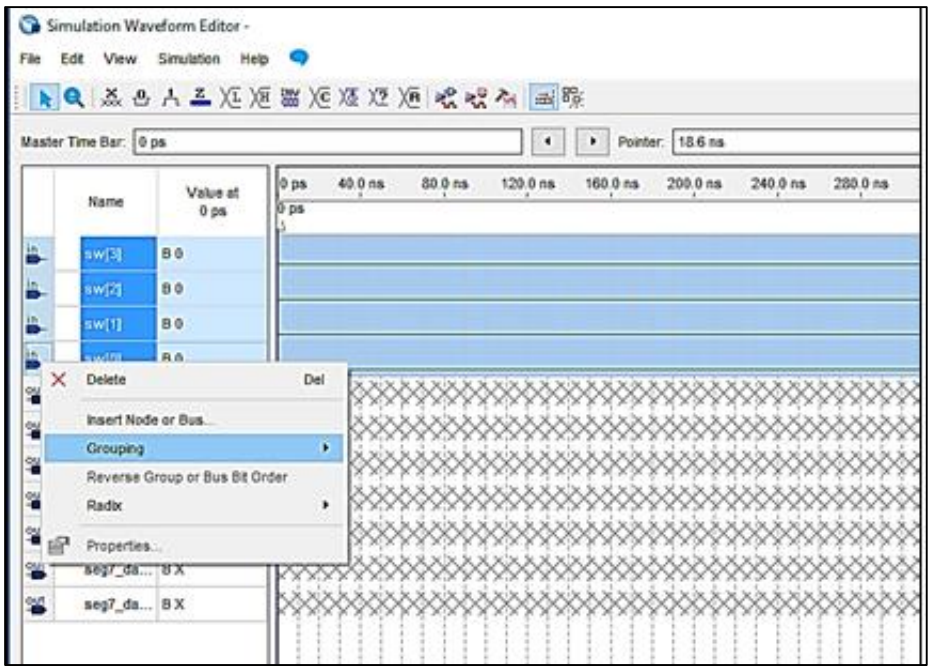


Figure 56 Lab2: Adding Nodes for Functional Simulation

Then click on the OK button. Then Click on the OK button on the Node Finder Dialog Box.



Within the simulation Window one can group the individual nodes into “groups” or “buses”. This can often save interpretation time of the simulation results. Start with the sw[3..0] nodes. SELECT the nodes in the following order with the Control Key continually pressed:

sw[3], sw[2], sw[1], sw[0].

With all of these signals highlighted RIGHT-CLICK over the names column and some options appear.

SELECT the Grouping option as in Figure 57.

Figure 57 Lab2: Grouping Nodes for Hexadecimal format

A new window will appear for the group of nodes as shown in Figure 58. Leave the name “sw” but set the RADIX to Hexadecimal. Click OK.

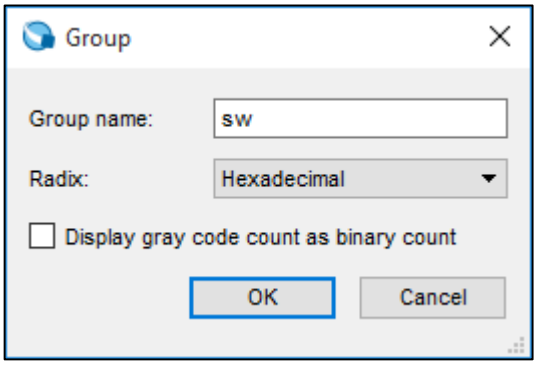


Figure 58 Lab2: Setting the Group Radix

Below in Figure 59 one can now see that the representation of the four sw nodes is replaced with a single BUS group called sw and its data is represented in Hexadecimal format.

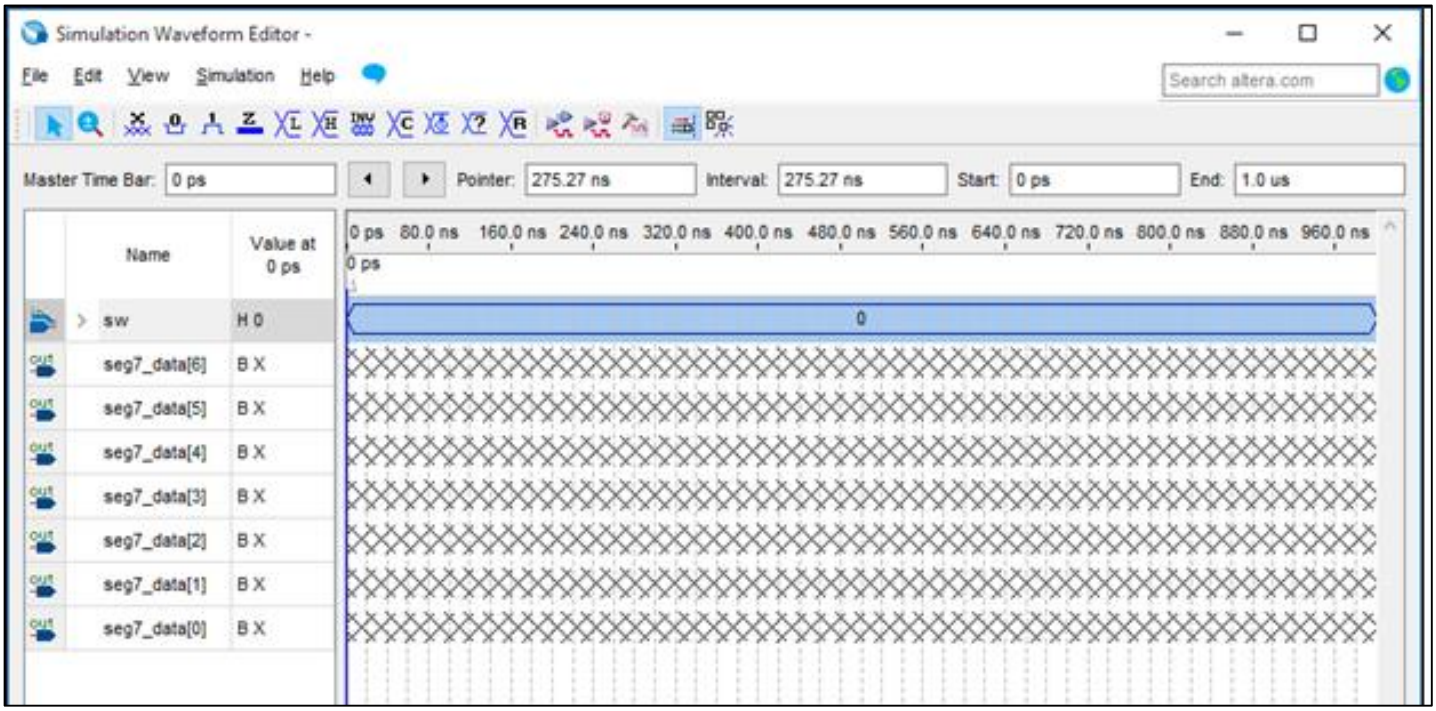
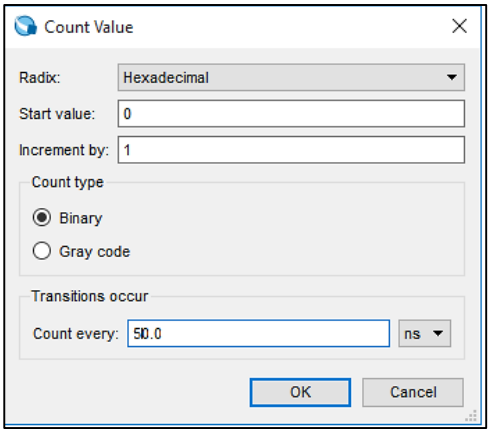


Figure 59 Lab2: Group Hex Value Shown in Simulator

Now we must add some STIMULUS to represent counting in hex. With the sw bus still selected Click on the COUNT VALUE button ( ). A window like that shown in Figure 60 will appear. Set the counting to increment by 1 every 50 nsec.



Click OK.

Figure 60 Lab2: Stimulus Counting Increment Setup

Now you should see the stimulus like the screen-shot below Figure 61:

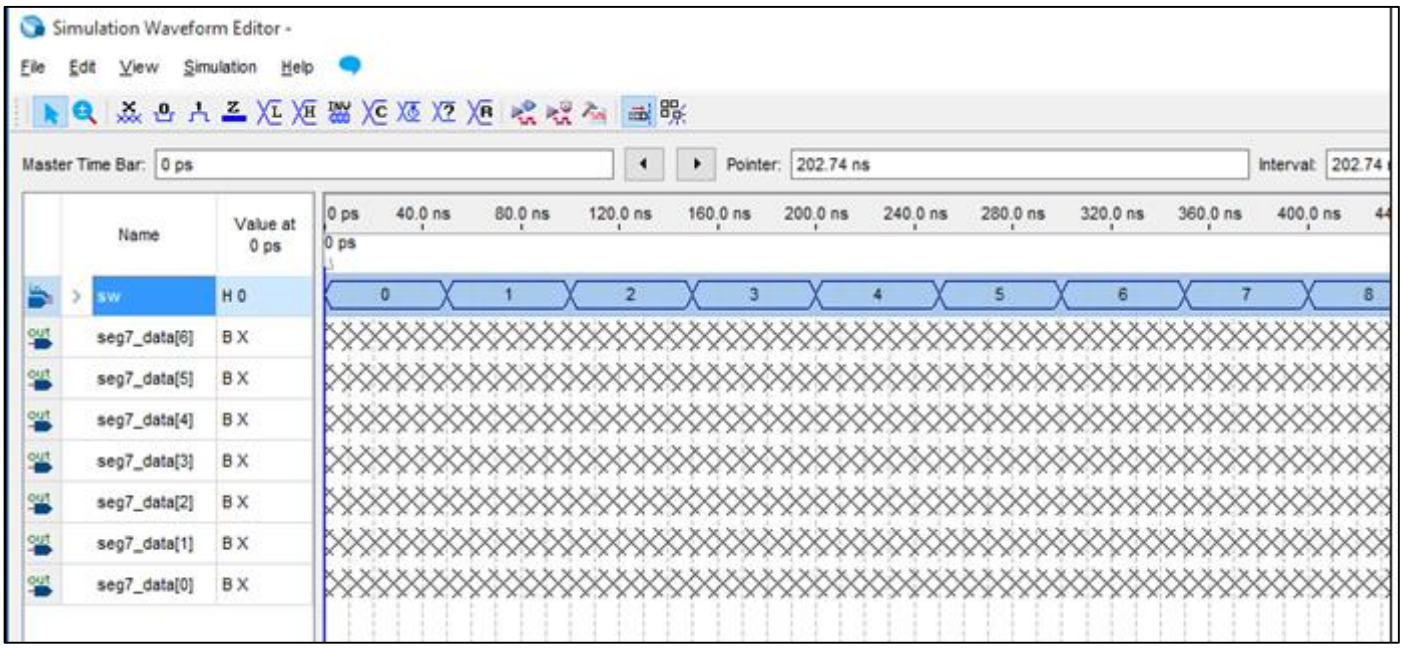


Figure 61 Lab2: Counting Input Stimulus

Save the file as waveform.vwf.

Now run the Functional simulation with the sw bus incrementing HEX values (0 – F). Refer to the reference simulation in Figure 62. Notice that for each HEX value in Figure 62 there is a set of column segment bit values. Compare your simulation results with those in Figure 62. Take note of any mismatched sets of column segment values per HEX input value in your simulation as compared to the reference column waveforms shown in Figure 62.

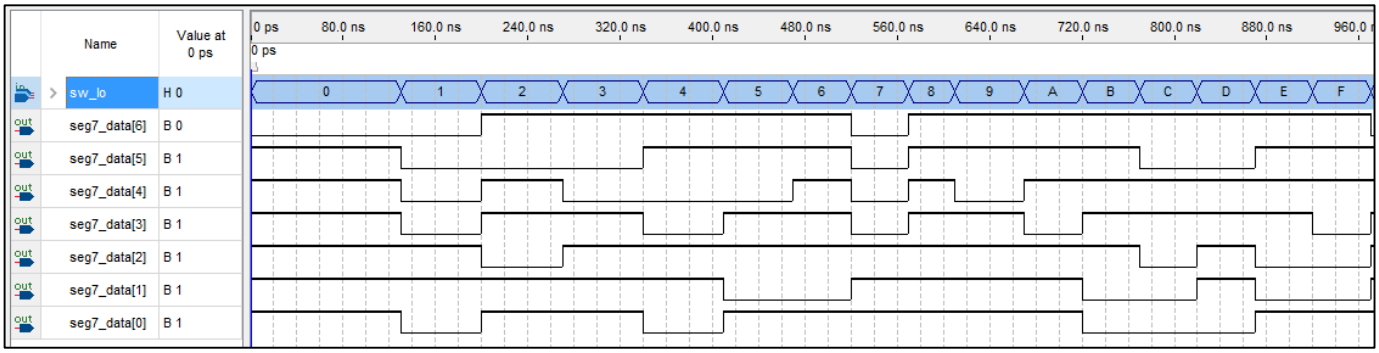


Figure 62 Lab2: Seven Segment Decoder Reference Operation

After noting the simulation differences for the HEX values open the SevenSegment.vhd file (inside Lab2 project directory) to correct the appropriate set of **row** segment values. The seg7_data[0] bits are in the segment "**A**", seg7_data[1]bits are in the segment "**B**", etc. Make the changes and then save the file in the Lab2 project folder. Then re-synthesize the design and run the simulation again to confirm the correct functionality as in Figure 62.

3.4.7 NEW VHDL Component - What is a Multiplexer or MUX function?

Multiplexers are used to select different data sources of input to a downstream function input or process. The selection is controlled by the state of the SELECT control inputs (see Figure 63).

Multiplexers can be found in a number of input/output ratios (e.g.: 2 to 1, 4 to 1, 8 to 1 …)

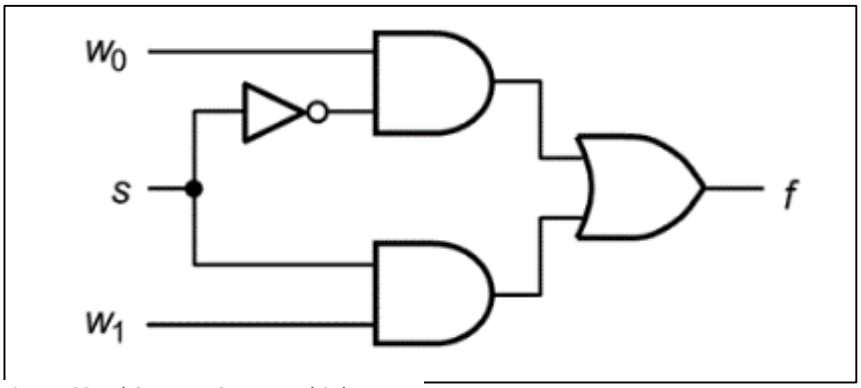


Figure 63 Lab2: VHDL 2 to 1 Multiplexer

Figure 63 is a simple 2 to 1 multiplexer or MUX function. Its output function "f" will pass thru the w0 input value when the "select" control input "s" is in a LOW state (or a "0"). When "s" is HIGH (or "1") the output function "f" will equal the value from the w1 input.

A graphical representation example of a QUAD-bit 4 to 1 multiplexer is shown below for your reference in Figure 64. A VHDL companion is shown in Figure 65. All busses are 4 bits wide.

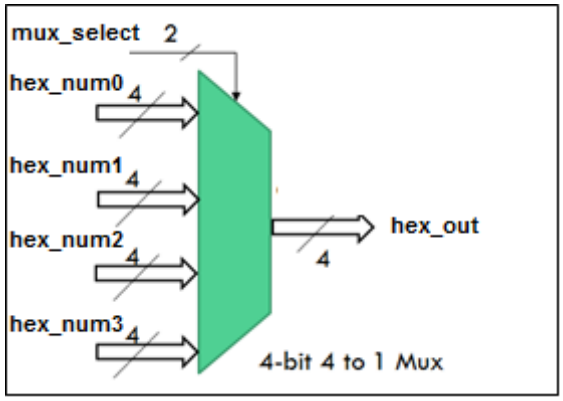


Figure 64 Lab2: Quad Port 4 bit Multiplexer

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4
5
6   entity hex_mux is
7   port (
8      hex_num3,hex_num2,hex_num1,hex_num0 : in  std_logic_vector(3 downto 0);
9      mux_select                          : in  std_logic_vector(1 downto 0);
10     hex_out                             : out  std_logic_vector(3 downto 0) -- The hex output
11  );
12
13  end entity hex_mux
14
15  architecture mux_logic of hex_mux is
16
17  begin
18
19   -- for the multiplexing of four hex input busses
20   with mux_select(1 downto 0) select
21   hex_out <= hex_num0  when "00",
22              hex_num1  when "01",
23              hex_num2  when "10",
24              hex_num3  when "11";
25
26  end mux_logic;
27
28
```

Figure 65 Lab2: VHDL Code for a Quad-Bit 4 to 1 Multiplexer

3.4.8 Lab2-Part B – Using the Seven Segment Displays

With the SevenSegment design debugged we will now use it. There are two seven segment displays on the LogicalStep board so two SevenSegment decoders will be required. So there must be a second **instance** of the SevenSegment decoder added to the next version of the LogicalStep_Lab2_top design.
Disconnect the SevenSegment decoder outputs (INST1) from the seg7_data pins that were used in Part A. This can be done by removing the **"seg7_data <= seg7_A"** line from the LogicalStep_Lab2_top file (added in Part A).

Add a second instance of the **SevenSegment** decoder component and name it as INST2.
For example:

INST2: SevenSegment port map (……);

Now connect the sw[7..4] switch inputs to a new signal bus called hex_B. This signal bus will have the same width (4 bits) as the hex_A signal bus when declared. Connect the other end of the hex_B bus to the INST2 SevenSegment decoder inputs. The output port of INST2 will connect to a new signal bus to be called seg7_B. This new bus will have the same width as the seg7_A bus.

The LogicalStep FPGA has just enough pins for the external peripherals. The FPGA pin-out has a common external seven-segment bus driving the Digit1 and Digit2 displays. Please refer again to Figure 53 described earlier for the orientation of Digit1 and Digit2 on the board.

So your FPGA design will need to use the provided seven segment multiplexer function.
A unique (14 input to 7 output) MUX will be added to our design for this purpose on the LogicalStep board. It's like having seven 2-to-1 muxes in parallel with a common SELECT input. This mux **is not suitable** for other purposes.
Add the VHDL function **segment7_mux** as a component declaration (Figure 66) to the LogicalStep_Lab2_top design.

```
component segment7_mux port (
        clk         : in  std_logic := '0';
        DIN2        : in  std_logic_vector(6 downto 0);
        DIN1        : in  std_logic_vector(6 downto 0);
        DOUT        : out std_logic_vector(6 downto 0);
        DIG2        : out std_logic;
        DIG1        : out std_logic
);
end component;
```

Figure 66 Lab2: VHDL Component Declaration for seg7_mux

Also instantiate the **segment7_mux** function instance as INST3. For example:

INST3: segment7_mux port map (……);

The internal signal buses (**hex_A, hex_B**) are connected to the INST1 and INST2 **inputs** mentioned earlier. The INST1 and INST2 **outputs (seg7_A, seg7_B)** must be connected to the INST3 **inputs** (for DIN2, DIN1 resp.).There are three other signals to connect to INST3. These are the clk input to the **clkin_50** pin; the output to the **seg7_char1** output pin and the output to the **seg7_char2** output pin. These signals are generated internally in the segment7_mux block. An example block diagram for these instance interconnects is shown below in Figure 67:

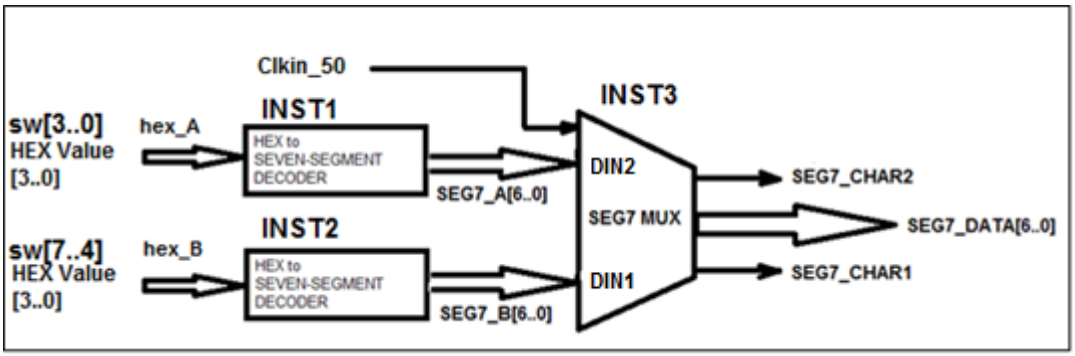


Figure 67 Lab2: Some Components Used in Lab2 Part B

With the clock input the "mux selector" is generated internally in the block and it is automatically alternated between the two SevenSegment data buses. The two outputs (seg7_char1 and seg7_char 2) are sent to the external display to direct the mux output to the appropriate seven segment display on the LogicalStep board.

Compile the design and download it to the FPGA. The Dual seven-Segment display should follow the two sets of HEX inputs from the switches.

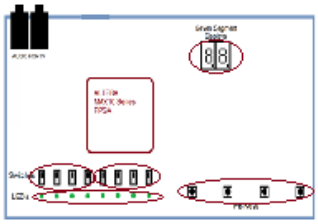
3.4.9 Lab2-Part C- Project Brief for Lab2 Demo

The next part of the lab will be used to develop a simple ALU with the LogicalStep_Lab2_top design. An ALU is an Arithmetic Logic Unit which is a fundamental part of computer. The functional requirements for our ALU are described in the following table:

SIGNAL TYPE:	SIGNAL NAME:	ASSIGNED PORT(s):	Comment
INPUTS	Operand1[3..0]	SW[3..0]	hex_A could be used
	Operand2[3..0]	SW[7..4]	hex_B could be used
	OPERATOR[3..0]	PB[3..0]	
OUTPUTS	LOGICAL RESULT[3..0]	LEDS[3..0], LEDS[7..4]	Logical operations of OP1,OP2 on LEDS[3..0], LEDS[7..4] are off
	ARITHMETIC RESULT[6..0]	SEVENSEG[6..0]. LEDS[7..0]	Addition of OP1, OP2 shown in HEX on Seven Segment Digit1 & Digit2 and in BINARY on the LEDS[7..0]

The functions are to be directed by the four PB Keys (perhaps use them as mux select lines). The PB keys are Active-low so you should either add inverters or modify your VHDL code to suit.

PB[3..0] Pins	Inverted PB[3..0]	OPERATOR	DESCRIPTION	DISPLAY
1111	0000	none	Operands displayed on seven-segment digits	Operand1 on seg7 DIGIT2, Operand2 on seg7 DIGIT1 LEDs are off
1110	0001	AND	Logical AND of operands	Result on LEDs[3..0]
1101	0010	OR	Logical OR of operands	Result on LEDs[3..0]
1011	0100	XOR	Logical XOR of operands	Result on LEDs[3..0]
0111	1000	ADD	Binary ADD of operands	Addition of OP1, OP2 shown on 7-segment displays in HEX and in BINARY on the LEDS[7..0]
Others	Others	None	Error condition	Display 88 and all LEDs on



Consider breaking the overall design into sections for your design.

Section 1: The hex to SevenSegment decoder and seg7_mux components from Lab2 part B may be used. But the SevenSegment Decoders will have different input signals. The 4 bit inputs will come from two halves of an eight-bit multiplexer output as shown in Figure 68. This output will be multiplexed between the 8 bit SUM (when PB(3) is active) and the two operands (e.g.: hex_A, hex_B) when PB(3) is NOT active.

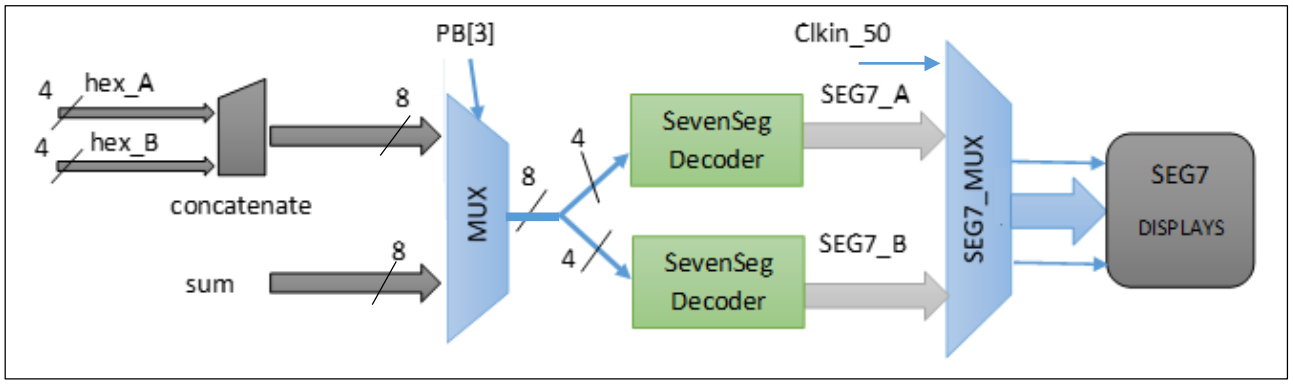


Figure 68 Lab2: Part C (Project) Multiplexing for Seven Segment Displays

When PB(3) is in the OFF state the two operands should be displayed on the two Seven Segment displays. However when PB(3) is activated then both of the Seven Segment displays must display the arithmetic sum. Add "signal" declarations as required for any intermediate connections between VHDL instances. To build a multiplexer you can use the reference example shown in Figure 65.

Section 2: The ALU logical results (Figure 69) should be displayed on the leds[3..0] but ONLY when a logical operation is selected from the table above and not for arithmetic. When the ADDITION mode is selected (related to pb(3)) the leds must show the ARITHMETIC result. Since the arithmetic result could be bigger than 4 bits the arithmetic result should be

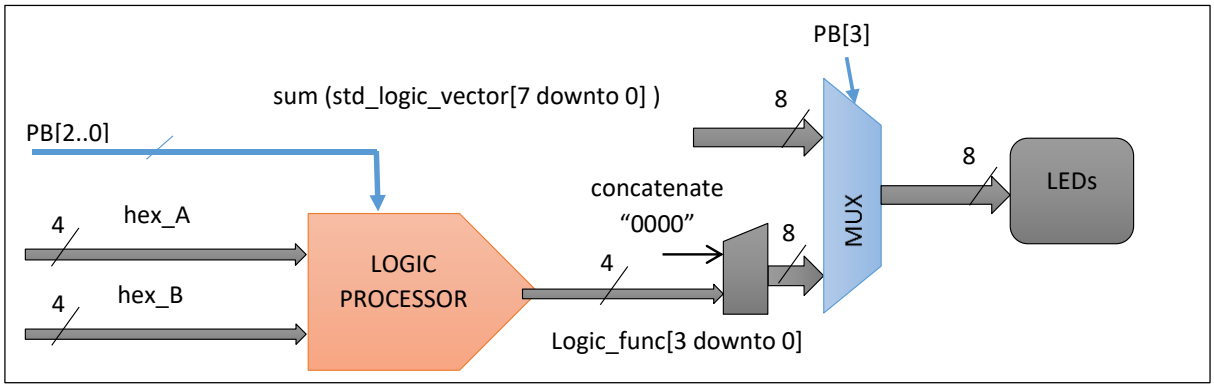


Figure 69 Lab2: Part C (Project) Logic Processor and Multiplexing for LED's

displayed on leds[7..0] when the ADDITION mode is requested.

Section 3: The hardest part of Lab2 Part C to understand will likely be the recasting of the data types. The logic signals must use the **std_logic_vector[..]** data type and they must be recast into the **unsigned** data type (like an Integer) for an arithmetic operation. When the Arithmetic result is computed the **unsigned** variable may be recast back to the **std_logic_vector[..]** for connection to the LEDs or the SevenSegment decoders etc. See Figure 70.

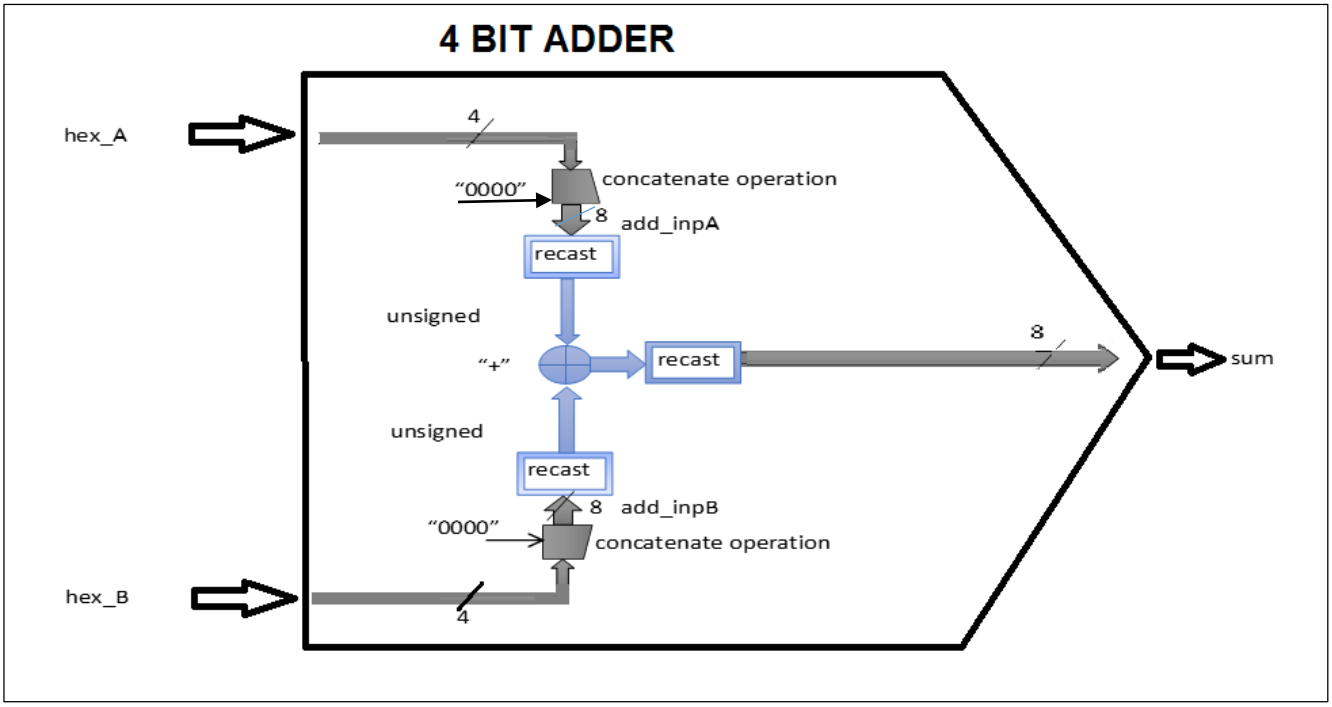


Figure 70 Lab2: Part C (Project) Four Bit Inputs and Adder Circuit

You may use a VHDL statement like the form below to do the add function and signal recasting (in light blue in Figure 70):

```
sum (7 downto 0) <= std_logic_vector(unsigned(add_inpA) + unsigned(add_inpB));
```

(where: **sum** is declared of type std_logic_vector[7 .. 0], **add_inpA**, **add_inpB** are ALL declared to be of type std_logic_vector[7 .. 0]. Note then that the add_inpA and add_inpB buses are 8 bits. But the input hex operands sw[3..0], sw[7..4] are only 4 bits each. Use the Concatenate operator "&" in VHDL to add your leading zeroes with each of the input operands (hex_A, hex_B)to modify add_inpA[], add_inpB[] into 8 bit vectors etc.

If more than ONE PB is pressed at the same time then ALL GREEN LED's should be activated to show an "ERROR" condition and the two SEVEN-SEGMENT displays should show "8".

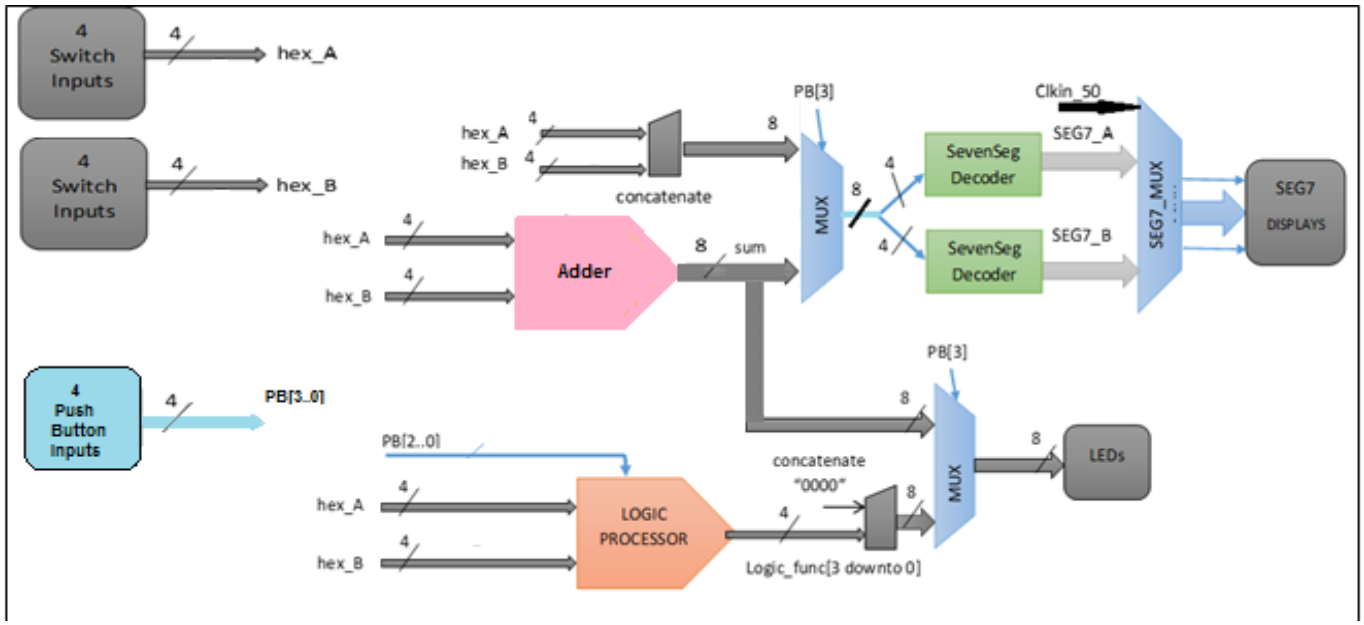


Figure 71 Lab2: Part C (Project) Sections Integrated

3.5 POST- Lab2 Activities

1. Complete the work for the Lab2 Demo design since it will be required for the Lab 3 session.

2. After the Lab 2 Demo, take a picture of the completed Submission form and upload it to the Learn Lab 1 Demo Dropbox.

3. A report on the Lab2 design is due within 24 hours (after your Lab2 Demo).

Submit your completed project report, as a PDF, to LEARN ECE-124 Lab2 Reports Dropbox folder. Make sure that your Session Number and Group Number are included in your report title (see Figure 72) and filename.

Labx GRPy SESSz REPORT

(the entire report must be a single PDF file)

(the filename must be the same as title with lab / group / session numbers)

(it is due within 24 hrs after the demo in Learn Dropbox folder)

Late submissions will be docked by 1 mark, 10%, per day late

First) LogicalStep_Labx_top.vhd file: - Show all VHDL code for this file (it must use the STRUCTURAL VHDL style) - Try to organize the code into a logical flow for readability - Add comments preceded by '—' to describe the VHDL code
Second) Subordinate VHDL files: - Show all of the other project VHDL code files - Try to organize the code into a logical flow for readability - Add comments preceded by '—' to describe the VHDL coding
Third) Supporting documentation requested for in the Lab Submission Form: - Simulations, RTL diagrams, Compile Report info as requested - Mark up simulations with comments to highlight interesting points - **Your comments should show the Marker you understand what is being displayed**

Figure 72 Lab2: Project Report Format

NEXT LAB SESSION: You will be designing a Magnitude Comparator (from scratch) that will be used in subsequent labs. An Energy Monitor Logic function will be developed in Lab3.

3.6 LAB2 SUBMISSION FORM

Table 2 - Lab2: Submission Form

ECE-124 Lab-2 Submission Form			
GROUP NUMBER:	Lab2 Demo:	Lab2 Report	**TA:**
LAB SECTION: 20_	Out of 10	Out of 10	

I am submitting this report for grading. I certify that this report, including any code, descriptions, flowcharts as part of the submission were written by the team member(s) below and there has not been any use of prior academic credit at this university or any other institution. The penalty for plagiarism or submission without signature(s) will be a grade of zero

NAME: (Print)	**UW User ID (not Student ID)**	**Signature**
Partner A:		
Partner B:		

LAB2 DESIGN DEMO	**Marks Allotted**	**A**	**B**
Seven Segment Display bugs (quantity 3) are corrected?	1		
Operands appear on Digit1 & Digit2 when PB's are OFF?	1		
Logical Results shown correctly on LEDs[3..0] when PB[2..0] ON?	1		
Arithmetic results shown on Digits and LED's when PB(3) ON?	2		
LEDs[7..4] OFF when Arithmetic result Less than or Equal to 1111	2		
ALL LED's ACTIVATED if more than ONE PB is PRESSED AT A TIME and BOTH Seven Segment LED's show '8'.	1		
DISCUSSION: Describe how you implemented the VHDL coding.	2		
LAB2 DEMO MARK	**Total out of 10**		

LAB2 DESIGN REPORT (see Lab Report info. In the Lab. Manual for details)	**Marks Allotted**	**TEAM**
Structural VHDL Used in top level VHDL design	2	
Sub-block VHDL files with good Coding Style	2	
Simulation of Logic functions showing the AND,OR,XOR modes	2	
Simulation of Arithmetic functions showing the ADD mode	2	
Total Design Logic Elements Used from Compilation Report	2	
Delay in Report Submission (-1 per day) x number of days:		
LAB2 Report MARK	**Total Out of 10**	

4 LAB3: VHDL for Combinational Circuits 2 – Energy Monitor (DATAFLOW/STRUCTURAL/BEHAVIORAL VHDL)

In this lab a new project for combinational logic will be designed in the Dataflow/Structural VHDL styles. A demonstration and lab report are expected as deliverables. The project for this lab session is a simple Energy Monitoring Controller that could be used in a home. There will be two ways to test the design. The first will be with a Simulation environment and the second will employ a "Testbench" approach. For the Testbench a BEHAVIORAL style of VHDL coding will be utilized. The final design must be demonstrated during the next Lab session. A Lab3 report must be submitted on LEARN within 24 hours after the Lab3 DEMO is completed.

4.1 Lab3 Intended Learning Outcomes

By the end of this lab students should be able to:

1) **CREATE** a 4-bit Magnitude Comparator with Dataflow/Structural VHDL coding
2) **DEVELOP** and **APPLY** a Functional Simulation to test the Magnitude Comparator
3) **UNDERSTAND** BEHAVIORAL VHDL coding style via VHDL Processes
4) **APPLY** a Testbench technique to test a digital design
5) **APPLY** components in a hierarchy (Structural VHDL) for a Home Energy Monitor

4.2 Prelab

1. Review the Lab2 procedures used for entering, testing and implementing an FPGA design.
2. Review the Lab2 Submission form for the Demo during the Lab3 session.
3. Be ready to have your Lab2 Demo design available for demonstration.

4.3 Lab3 Outline

Attendance will be taken.

The following topics will be presented:

1. Review of Lab2
2. New VHDL Component - What is a Magnitude Comparator?
3. Project Setup for Lab3
4. Lab3 Part A. – Creating a 4 Bit Magnitude Comparator in Dataflow and Structural VHDL
5. Functional Simulation of a 4-Bit Magnitude Comparator
6. An Alternative way of testing – Testbench (Behavioral VHDL)
7. Lab3 Part B – Creating a Home Energy Monitor with an on-board Testbench

4.4 Lab3 Activities

4.4.1 Recall from Lab2:

During Lab2 we developed some VHDL knowledge on using the VHDL "Structural" style of coding. We learned how to re-use VHDL previously developed files as components (Seven Segment Decoder and Multiplexers) and instantiate a number of copies of these components in the LogicalStep_Lab2_top VHDL design file. The INSTANCES could be then "wired" together using declared signals (busses) and port mapping.

We also learned to "re-cast" signal data types into different types so that other types of processing could be accomplished.

4.4.2 Project Setup for Lab3

Create a new Lab3 project as was done previously. Using the Windows File Explorer go to the ECE-124 folder directory. From LEARN download the Zipped folder "Lab3" into the ECE-124 folder. Extract the zipped folder contents to create your Lab3 project folder. It should have the LogicalStep_Lab3_top.vhd, LogicalStep_Lab3.tcl and segment7_mux.vhd files inside it.

Start up the Quartus Prime software. Begin the new project by using

FILE>New Project Wizard). Enter the new Project parameters:

Project Folder: **N:/ECE-124/Lab3**

Project Name: **LogicalStep_Lab3**
Project Top Level: **LogicalStep_Lab3_top**

Click FINISH on the New Project Wizard Dialog Window.

Copy the sevensegment.vhd file from your Lab2 folder into your Lab3 Project folder.

Then run the Lab3 TCL script to assign the FPGA device type, pin assignments subordinate VHDL files for the LogicalStep_Lab3 project.

4.4.3 New VHDL Component – What is a Magnitude Comparator?

A magnitude comparator accepts two input variables and determines how they compare to each other. There are three outputs: input A is greater than input B; input A is equal to input B; and input A is less than input B.

4.4.4 Creating a 4-Bit Magnitude Comparator by DATAFLOW / STRUCTURAL VHDL Design

For Lab3 a magnitude comparator is to be designed with **GATES** using two levels of hierarchy. The bottom level (DATAFLOW) will just compare the magnitude of two single bits and then pass along its three results (A>B, A=B, A<B) to the upper level. The upper level will use multiple instances of the lower level (component). This level will be largely STRUCTURAL in VHDL coding. But alongside the Structural portion will be another DATAFLOW implementation to create the second level VHDL outputs. The outputs at this level are the overall A>B, A=B, A<B results for the four-bit operands of A and B.

The logic that is to be built in Lab3 project requires a 4-bit Magnitude Comparator to determine whether more energy is required for heating or cooling a home.

So….you guessed it… we must make up one for the project. See Figure 73.

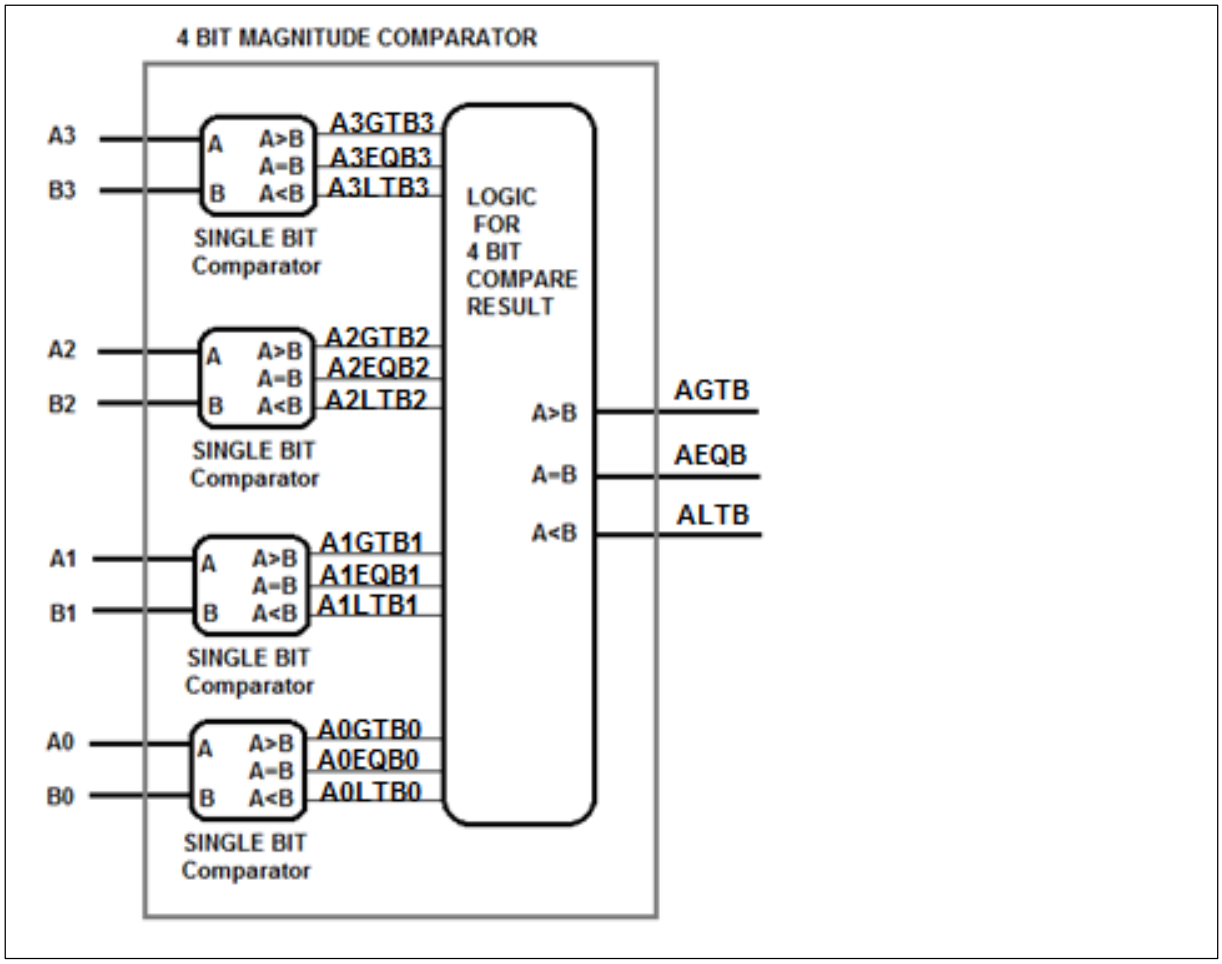


Figure 73 Lab3: Levels of Comparator Logic

Start by creating a Single bit Magnitude Comparator Truth Table with single bit inputs for A and B. Then create 3 output columns (one for each of A>B, A=B, A<B) and then use logical reasoning to determines which conditions force these outputs to activate.

Create a Single-bit comparator Dataflow VHDL file (Compx1.vhd) with the above information in its design using Boolean equations for the outputs. Then create a higher-level Structural VHDL file (Compx4.vhd) which includes the Declaration of the Compx1 as a component and also 4 instances of the Compx1 in your Compx4 architecture section.

In the Compx4 VHDL file some arrangement of logic has to be made to make use of all of the individual comparisons coming from the instances of Compx1. This is to be in BOOLEAN form as well.

Start with the highest-order bit in the group of bits being compared. For example:

If A3 > B3 is TRUE then the lower bits of A2, B2, A1, B1, A0, B0 don't need to be considered further for the 4 bit A > B equation. Use the table below (X means "Don't Care").

Same thing if B3 > A3 (which is of course the same thing as A3<B3).

However if A3=B3 is TRUE then you have to look to the comparison of the lower bits (A2>B2) and so on. Remember the A>B, A=B, A<B outputs are all mutually exclusive.

Comparison Inputs from 1-Bit Comparators												4-Bit Comparator Outputs		
A3<B3	A3=B3	A3>B3	A2<B2	A2=B2	A2>B2	A1<B1	A1=B1	A1>B1	A0<B0	A0=B0	A0>B0	A<B	A=B	A>B
X	X	1	X	X	X	X	X	X	X	X	X	0	0	1
1	X	X	X	X	X	X	X	X	X	X	X	1	0	0
X	1	X	X	X	1	X	X	X	X	X	X	0	0	1
X	1	X				X	X	X	X	X	X			
X	1	X							X	X	X			
X	1	X							X	X	X			
X	1	X												
X	1	X												
X	1	X	X	1	X	X	1	X	X	1	X	0	1	0

Your gate-tree design for the 4-bit Comparator (Compx4) will be based on that logical approach. Fill in the blanks for the inputs and outputs for the 4 bit Comparator function table above. Then write Boolean equations for the three Comparator outputs inside your Compx4 file based on the inputs.

After the Compx4 design is completed; open the LogicalStep_Lab3_top.vhd file and declare the Compx4 in it as a component and also instantiate one instance in the Architecture section in LogicalStep_Lab3_top.vhd. Connect a 4-bit signal group to the sw[3..0] inputs for input A and a second 4-bit signal group to sw[7..4] bits for input B. Connect the 3 magnitude comparison outputs of the Compx4 to the leds[2..0] output pins of LogicalStep_Lab3_top.vhd.

A set of Functional Simulations will be used to prove whether the 4 bit Magnitude Comparator was designed correctly.

4.4.4.1 Functional Simulation of the Magnitude Comparator

4.4.4.1.1 Adding Nodes to the Simulator

Begin this phase of Lab3 by running the Analysis and Synthesis process.

Open a Functional Simulation window (section 2.3.2 of Lab1) and insert the nodes of interest

(inputA ←sw[3..0], input B←sw[7..4], magnitude comparison outputs ←leds[2..0])

As before, group the individual nets (sometimes called busses or vectors). For example, if you select the signals one at a time for sw[3], sw[2], sw[1], sw[0] you can group them into a hexadecimal format and name the group as Input_A. Similarly this can be done for the sw[7], sw[6], sw[5], sw[4] becoming a hexadecimal formatted Input_B vector.

4.4.4.1.2 Adding Stimulus to the Inputs

To stimulate this design it is best to make one input vector change while the other is held static. Observe the behavior of the A>B, A=B and A<B outputs. Simulate with 4 different stimulus sets.

For example, for simulation 1 set Input_A to a static value of hex 1. Then run Input_B through all hex values between 0x0 and 0xF. Save that stimulus file and then run that simulation and save it.

For simulation 2 set Input_A to hex 2 and repeat the sequence on Input_B as before. Run the simulation and save it to a different file.

For simulation 3 set Input_A to hex 5 and repeat the sequence on Input_B. Run the simulation and save it to a different file.

Finally for simulation 4 set Input_A to hex 8 and repeat the sequence on Input_B. Run the simulation and save it to a different file.

Save the four stimulus files for your report.

4.4.5 An Alternative Way of Testing - Testbenches (Behavioral VHDL)

Functional simulations are very useful during the development of digital designs. But after the design is in production a FASTER means of testing must be used due to the high production volume.

To achieve that a “testbench” may be employed in the digital design. This is “dormant” in the typical product application but gets used in the production environment for manufacturing tests.

Testbenches can vary in complexity, performance and scope. Quite often they are designed with a high level of abstraction because it may not be critical what logic is used to implement the testbench design. Thus a well-suited VHDL coding style for testbench design is the BEHAVIORAL coding style. This style resembles a sort of “algorithmic” approach to coding.

Before we get to the BEHAVIORAL coding style we must discuss a new VHDL construct called a PROCESS. VHDL Processes are by far the most often used constructs for BEHAVIORAL coding.

VHDL Process structures have the following syntax:

```
Label:  process (sensitivity list) is
        begin
                ..
                Sequential statement
                …
                Sequential statement
                …
        end process;
```

where:

LABEL: is used to identify the process in the design,

SENSITIVITY LIST: lists inputs to the process. For combinational Logic ALL input signals must be present in this list.

SEQUENTIAL STATEMENTS: put the process steps in a specific sequence. Sequential statements bring a new set of capabilities to logic design. These include statements such as “IF-THEN-ELSE; CASE statements etc.

SIGNALs (must be defined outside of the process construct) used within the process get their values assigned their values AFTER the process sequence steps are completed.

Please note that the Process statement itself is just another way to describe logic behavior. There is NO time consumed creating the values for the outputs for combinational logic processes. The Process statement is CONCURRENT with other concurrent statements in a VHDL design.

A simple example follows below. The PROCESS IS EVALUATED IF AND ONLY IF THERE IS A CHANGE TO THE ELEMENTS IN THE SENSITIVITY LIST.

```
signal Zout  : out  std_logic_vector(2 downto 0); -- NOTE: signals are declared outside of a process statement

…

begin

…

 Process1: Process (A,B,C) is

begin

        if (A)  then
                Zout <= 3'b001;
        elsif (B) then
                Zout <= 3'b010;
        elsif (C) then
                Zout <= 3'b100;
        else
                Zout <= 3'b000;
        end if;
end;
```

CBA	SIGNAL Zout (AFTER PROCESS IS FINSHED)
000	000
001	001
010	010
011	001
100	100
101	001
110	010
111	001

The IF THEN ELSE sequential statement brings “priority levels” into the process. The conditions listed earliest have the highest priority.

For the above process the table describes the assignment to externally declared signal Zout AFTER the process is evaluated.

Another sequential statement that can be used inside a Process statement is the CASE statement.

```
signal Zout  :       std_logic_vector(2 downto 0); -- signals are declared outside of a process statement

…
begin
…

Process2:process(A,B,C) is

        variable inp : std_logic_vector(2 downto 0);

        begin

         inp := C & B & A;      -- variable inp is assigned the concatenated inputs of C, B, A

        case inp is

                when  000  =>     Zout <= "000" ;

                when  001  =>     Zout <= "001" ;

                when  010  =>     Zout <= "010" ;

                when  011  =>     Zout <= "001" ;

                when  100  =>     Zout <= "100" ;

                when  101  =>     Zout <= "001" ;

                when  110  =>     Zout <= "010" ;

                when  111  =>     Zout <= "001" ;
        end case;
end process;
```

The above example will result with the same values for “Zout” as in the previous example. All variants of the CBA bits in a CASE statement are given the SAME PRIORITY and each possible value of the CBA inputs creates an explicit value for Zout. A “when others” case is needed for a real simulator as “0” and “1” are only 2 of the 9 possible states for each bit (1, 0, Z (high impedance), W (weak, neither 0 nor 1), L (weak 0), H (weak 1), - (don’t care), U (uninitialized), X (unknown, multiple drivers).

An alternative CASE statement below could shorten the list of values allowed for Zout when we bring in the “X” state value for B and C values. Note that if not all possible variants of CBA are included in the CASE statement then a “when OTHERS” clause must be added to catch all of the remaining assignments for Zout.

```
signal Zout  :  std_logic_vector(2 downto 0); -- signals are declared outside of a process statement
```

```
…
begin
…
Process2X: Process (A,B,C) is
variable inp: std_logic_vector(2 downto 0); -- variable declared
begin
 inp := C & B & A;      -- variable inp is assigned the concatenated inputs of C, B, A
case (inp) is
begin
        when XX1 =>:
                Zout <= 3'b001;
        when X10 =>
                Zout <= 3'b010;
        when 100 =>
                Zout <= 3'b100;
        when others =>
                Zout <= 3'b000;
end case;
end;
```

For all of the above Process examples Zout is handled in a CONCURRENT fashion with the other VHDL structures outside of the Process which also may use Zout (like an implied wide-ORing for Zout). Zout is declared as a SIGNAL and must be declared OUTSIDE of Process constructs.

Only Constants and Variables are allowed to be declared within a Process construct (not signals).

There are other properties for Process statements that will be covered in the next section on Sequential Logic (Lab4).

KEY Things to Remember:

1) Signals that are used in a Process construct are assigned values at the END of the process.
2) Variables declared and used in a process get their values assigned (:=) immediately as the steps proceed.

4.4.5.1 Creating a Simple Testbench with Behavioral VHDL for the Magnitude Comparator

A Process statement will be used to check the Magnitude Comparator output against the Input_A and Input B values described in the **Functional Simulation of the Magnitude Comparator** section. This process will create combinational logic. It is added below as a separate section (Testbench1) of the VHDL code that you can enter into your Lab3 design.

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

entity LogicalStep3_top is port (
   clkin_50    : in  std_logic;
   pb          : in  std_logic_vector(3 downto 0);
   sw          : in  std_logic_vector(7 downto 0); -- The switch inputs
   leds        : out std_logic_vector(7 downto 0); -- for displaying the switch content
   seg7_data   : out std_logic_vector(6 downto 0); -- 7-bit outputs to a 7-segment
   seg7_char1  : out std_logic                     -- seg7 digi selectors
   seg7_char2  : out std_logic                     -- seg7 digi selectors

);
end LogicalStep3_top;

architecture Energy_Monitor of LogicalStep3_top is

/*
add any components here
*/

signal AEQB, ALEB,AGEB  : std_logic; -- Outputs from Magnitude Comparator

signal TEST_PASS :  std_logic;
/*
add any other signals here
*/

begin

/*
Lab3 Project (Energy Monitor) goes in here
*/

Testbench1:
PROCESS (sw, AEQB, AGEB, ALEB, pb(2)) is

variable EQ_PASS, GE_PASS, LE_PASS  : std_logic := '0';

begin
      IF ((sw(3 downto 0) = sw(7 downto 4)) AND (AEQB = '1')) THEN
      EQ_PASS := '1';
      GE_PASS := '0';
      LE_PASS := '0';

      ELSIF ((sw(3 downto 0) >= sw(7 downto 4)) AND (AGEB = '1')) THEN
      GE_PASS := '1';
      EQ_PASS := '0';
      LE_PASS := '0';

      ELSIF ((sw(3 downto 0) <= sw(7 downto 4)) AND (ALEB = '1')) THEN
      LE_PASS := '1';
      GE_PASS := '0';
      EQ_PASS := '0';

      ELSE
      EQ_PASS := '0';
      GE_PASS := '0';
      LE_PASS := '0';

      END IF;

      TEST_PASS <= pb(2) AND (EQ_PASS OR GE_PASS OR LE_PASS);
      leds(6) <= TEST_PASS;
end process;

end Energy_Monitor;
```

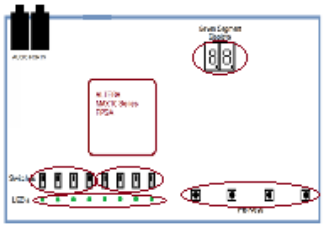
Figure 74: LAB3: Behavioral VHDL Used in a Process Construct

This testbench design can be applied to your Lab3 Project discussed in the next section. In a production setting this simple testbench could be a consistent point of reference for a test “status” result for numerous switch input settings used by a production line operator.

4.4.6 Lab3 Project – Creating a Home Energy Monitor

The project for this lab session is another VHDL-based FPGA design for implementing a simple Energy Monitoring Controller that could be used in a home. The final design must be demonstrated during the Lab3 DEMO time. The Testbench1 must be included for the DEMO.

SIGNAL TYPE:	SIGNAL NAME:	ASSIGNED PORT(s):	Comment
INPUTS	Current_Temp[3..0]	SW[3..0]	Set somewhere is mid-range
	Desired_Temp[3..0]	SW[7..4]	Various settings
	Vacation Mode, MC_TESTMODE, Window Open, Door Open	PB[3..0]	V/C MODE-PB3, MC_TESTMODE-PB2, Window-PB1, Door-PB0
OUTPUTS	FURNACE_ON	LEDS[0]	ON when BELOW temp and with all windows/Doors closed
	SYSTEM AT TEMP	LEDS[1]	ON when AT TEMP (even if Doors/windows are OPEN)
	A/C ON	LEDS[2]	ON when ABOVE temp and with all windows/Doors closed
	BLOWER ON	LEDS[3]	ON when A/C or Furnace are ON and with all windows/Doors closed
	Window Open, Door Open, in alignment with PB[1..0]	LEDS[5..4]	Window-led5, Door-led4
	TEST_PASS	LEDS[6]	Mag-Comp TEST_PASS
	VACATION MODE	LEDS[7]	When PB3 pressed
	Current Temperature[3..0]	DIGIT 2	Current Temp (HEX)
	Desired Temperature[3..0]	DIGIT 1	Desired Temp (HEX) or Vacation Mode Setting



With the door and window CLOSED the Furnace or A/C will be activated whenever there is a difference between the Current and Desired temperature values. When either of these turn ON the Blower must also turn ON.

The AT_TEMP indicator turns ON when DesiredTemp = Current Temp regardless if a door or window is OPEN.

The appropriate status indicators must be illuminated when any door or window is OPEN.

The Current and Desired temperatures are to be displayed on the two seven-segment digits.

Reuse the seven_segment decoders and multiplexers components from Lab2. You can use your newly developed 4-bit comparator design to control the Blower and determine whether the A/C or Furnace is to be activated.

Implement a Vacation Mode (via PB(3)) that, when pressed, provides an alternate FIXED setting (Vacation Mode Value set to “0100” in binary inside your VHDL file) instead of the Desired Temp value. When PB(3) is pressed LED7 is to go ON and the Comparator is forced to compare the Current Value against the Vacation Mode Value and the Furnace/AC operations should activate accordingly. The Vacation Mode value should be displayed on DIGIT1. Please keep the Vacation Mode Value and its logic separate from your Comparator design.

4.4.7 Lab3 Project – Adding the test bench to the Home Energy Monitor for Production Testing

Activating PB(2) at any time will cause the Testbench1 logic to do a functional test of the internal Magnitude Comparator. The TEST_PASS LED (LEDS(6)) will activate if the test bench comparisons of the Input A and Input B magnitudes match with the Magnitude Comparator outputs.

To force a mismatch between the test bench comparisons with the Magnitude Comparator there is a way to do this using the Vacation Mode, Input_B and the MCTest Mode together. Be ready to explain and demonstrate during your DEMO.

4.5 POST - Lab3 Activities

1. Complete the work for the Lab3 Demo design since parts of it will be required at the next lab session in Lab 4. Different settings of Current Temp and Desired temp monitoring and control must be demonstrated; Doors/Windows Open detection, MC_TestMode, Vacation Mode demonstrated.

2. After the demo, upload a picture of the completed Submission form to the Learn Lab 3 Demo Dropbox.

3. A report of the Lab3 design, as a PDF, is due within 24 hours after your Lab Demo and its contents are outlined in the Lab3 Submission Form.

Submit your completed project report to LEARN ECE-124 Lab3 Report Dropbox folder. Make sure that your Session Number and Group Number are included in your report title (see Figure 72 in the Lab2 section) and filename.

NEXT LAB SESSION: You will be getting into the really cool stuff in digital logic design. You will be learning about storage elements (flip-flops) and state machines that do the logic processing over time (sequential logic).

4.6 LAB3 SUBMISSION FORM

Table 3 - Lab3: Submission Form

ECE-124 Lab-3 Submission Form			
GROUP NUMBER:	Lab3 Demo	Lab3 Report	**TA:**
LAB SECTION: 20_	Out of 10	Out of 10	

I am submitting this report for grading. I certify that this report, including any code, descriptions, flowcharts as part of the submission were written by the team member(s) below and there has not been any use of prior academic credit at this university or any other institution. The penalty for plagiarism or submission without signature(s) will be a grade of zero

NAME: (Print)	**UW User ID (not Student ID)**	**Signature**
Partner A:		
Partner B:		

LAB3 DESIGN DEMO	**Marks Allotted**	**A**	**B**
Desired Temp (sw[7..4]) is displayed on Digit1?	**1**		
Current Temp (sw[3..0]) is displayed on Digit2?	**1**		
MCTest_Mode, Door/Wndw (PB[2..0]) displayed on LEDs[6..4]?	**1**		
Furnace, Blower, System At Temp, A/C, Blower Indicators LEDs?	**1**		
A/C ON & Blower ON when Current Temp > Desired Temp?	**1**		
Furnace ON & Blower ON when Current Temp < Desired Temp?	**1**		
A/C, Furnace, Blower turn OFF when Doors/Windows Open?	**1**		
VACATION MODE Operation: LED7 goes ON with PB3 and Furnace, A/C use internal “0100” temp value	**1**		
DISCUSSION: Comment on your VHDL Implementation?	**2**		
LAB3 DEMO MARK	**Total out of 10**		

LAB3 DESIGN REPORT (see rubric on LEARN for details)	**Marks Allotted**	**TEAM**
All VHDL files (not the sevenseg_decoder or seg7_mux files). Structural VHDL design must be used at the Top Level. Comparator must have a **Boolean equation** per each output.	2	
Truth Table for 4-Bit Comparator Design with all entries inserted	2	
Part A Simulations of Comparator showing A>B, A=B, A<B	2	
RTL View of the Logic design (just of the Top Level)	2	
Total Design Logic Elements Used: /8064	2	
Delay in Report Submission (-1 per day) x number of days:		
LAB3 REPORT MARK	**Total Out of 10**	

5 LAB4: VHDL for Sequential Circuits – Flip-flops & State-Machines

Lab4 will be the first one that uses Sequential Logic. Background information on sequential logic will be briefly covered to suit the Lab4 requirements. Design elements from previous labs will be used for Lab4. Both Moore and Mealy State Machine designs will be new parts for the design goals for this lab.

5.1 Lab4 Intended Learning Outcomes

By the end of this lab students should be able to:

1) **UNDERSTAND** Sequential Logic concepts
2) **APPLY** "storage elements" called flip-flops (registers) used in Sequential Logic designs
3) **IMPLEMENT** registers using a Process Construct
4) **UNDERSTAND** simple Shift Register designs
5) **APPLY** State Machine design concepts, including Moore and Mealy forms, to implement State Machines to control a two-dimensional Robotic Arm Controller (RAC)

5.2 Prelab

1. Review the Lab3 steps used for entering, testing and implementing FPGA designs.
2. Review the Lab3 Submission form from LEARN for the Lab4 session.
3. Be ready to have your Lab3 Demo design available for demonstration.

5.3 Lab4 Outline

Attendance will be taken.

The following new topics will be presented:

1. Review of Lab3
2. Project Setup for Lab4
3. Brief Discussion on Sequential Processing
4. New VHDL Component - What is a Flip-Flop? How are they created in VHDL?
5. Creating Some Simple Flip-Flop Register Designs for Sequential Logic
6. New VHDL Component – What are State Machines?
7. Project Brief for Lab4 Demo

5.4 Lab4 Activities

5.4.1 Recall from Lab3:

Last time in the lab we developed a 4-bit Magnitude Comparator. The comparison process was implemented in two levels. This technique often simplifies the creation phase of a design. This Magnitude Comparator function was used in an Energy Monitoring Controller to determine current and desired temperature differences. Also in Lab3 the topic of Behavioral VHDL was introduced and a simple “testbench” design was added to test the Lab3 project Magnitude Comparator with a Process construct. The Process used in Lab 3 was formed for Combinational Logic only.

5.4.2 Initial Project Setup for Lab4

Start the Lab4 project like what was done in earlier Labs. Go to LEARN and download the Lab4 Zipped folder “Lab4” into the ECE-124 folder on your N: Drive. Extract the contents to create the new Lab4 project folder and its contents therein.

Start up the Altera Quartus Prime platform and begin a new project with the New Project Wizard. Enter the new Project parameters:

Project Folder: Lab4
Project Name: LogicalStep_Lab4
Project Top Level: LogicalStep_Lab4_top

Click FINISH on the New Project Wizard Dialog Window.
Run the Lab4 TCL script to assign the FPGA device type and pins for the LogicalStep FPGA.

This part of the lab session will require some of the same components that were used in the previous lab such as SevenSegment, seg7_mux, Compx4 and Compx1 VHDL files. Copy these VHDL files from your Lab3 Project folder into your Lab4 Project folder.

5.4.3 Brief Discussion on Sequential Logic Processing

Sequential logic is used for event-driven processing (Please don't confuse Sequential Logic with Sequential Statements in Process constructs). You can probably think of things in life that require a gradual process to complete tasks. These time-driven processes usually run in a step by step sequence.

A trivial example could be something that is done every morning perhaps before getting here for your lab work:

Process: Getting Simple Breakfast

Step 1: Get bread from cupboard
Step 2: Place bread in toaster
Step 3: Get plate
Step 4: Wait for Toast cycle completion
Step 5: Put Toast on plate
Step 6: Butter the Toast
Step 7: Add Jam
Step 8: Process done…..--> Enjoy.

These tasks must be done in the step by step sequence described for the process to be completed successfully.

In the next section a new logical element will be described that can "hold" the logical value for any sequential step in a process.

5.4.4 New VHDL Component – What is a Flip-Flop? How are they created in VHDL?

Well this is not a reference to a type of footwear. Nope.

In digital logic a basic storage element that is used in Sequential Logic is the Flip-Flop. This function records a sample presented at its input and presents a copy of that input value to its output when a clock signal is received. If there are no further clocks then the Flip-Flop should hold its output value indefinitely (as long as there is power). Another name, perhaps more professional sounding, for this kind of device is a Register.

Flip-Flops can come in many flavors (R/S, J/K, T, D) but most casual references to flip-flops are referring to the D-type variety as shown below in Figure 75:

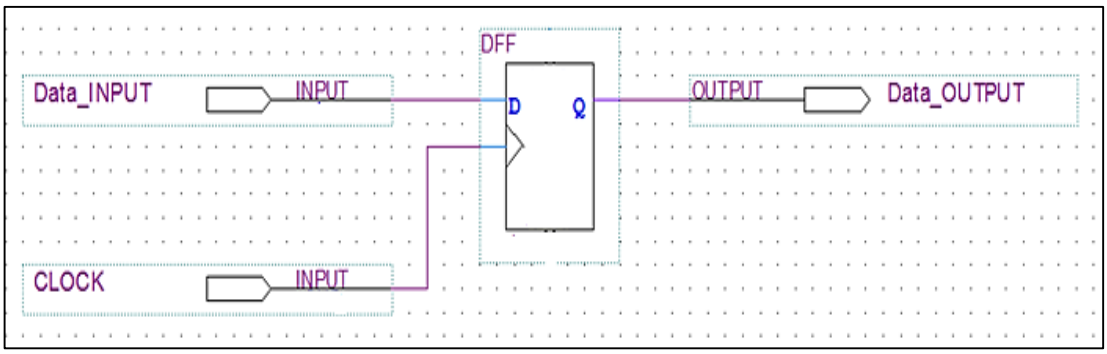


Figure 75 Lab4: D-Type Flip-Flop

There are usually other inputs to a standard flip flop (such as Asynchronous Preset, Asynchronous Clear, Clock Enable inputs) but for now let's just focus on the simpler functions.

Registers can be used in all kinds of higher-order functions such as counters, shift registers, state machines, accumulators etc. In Lab1 we used a 28-bit counter to automate the Lab1 digital logic. It would have used 28 "registered" outputs.

Registers can be created using the VHDL Process constructs for Sequential Logic.

Like what was introduced in Lab3 the VHDL Process structures have the following syntax:

```
Label:  process (sensitivity list) is
        begin
        ..
        end process;
```

The Label can be any convenient value for code readability etc. The <sensitivity list> is the list of inputs that the process is directed to observe to determine when a process is to be updated. For Sequential Logic such inputs usually include only a "clock" and "reset" signals for example. For Combinational Logic the Sensitivity List contains all inputs used by the process.

For clocks there is usually a kind "filtering" added within the Process construct to constrain process changes to occur on a desired clock EDGE. For example one could use the "IF (rising_edge (clock)) THEN…" in the construct to evaluate whether any state machine change will occur but only in association with the rising edge of each clock cycle.

So a Flip-Flop or Register (Figure 75) can be created in VHDL using a PROCESS. See the example below:

```
process (<clock_signal>) is
begin
        if (rising_edge(<clock_signal>)) then
                register <= <data>;
        end if;
end process;
```

The Register will accept the “data” value on the rising edge of <clock signal>. When the next clock rising edge arrives the input <data> is again captured and the previous captured value is OVERWRITTEN. But what if we want to capture the data and KEEP IT for a while? What do we need to do since the clock may always be running? For this situation we need to add some new Flip-Flop functionality. In Figure 76 we add a new capability. This is the Clock Enable or just “ENABLE”.

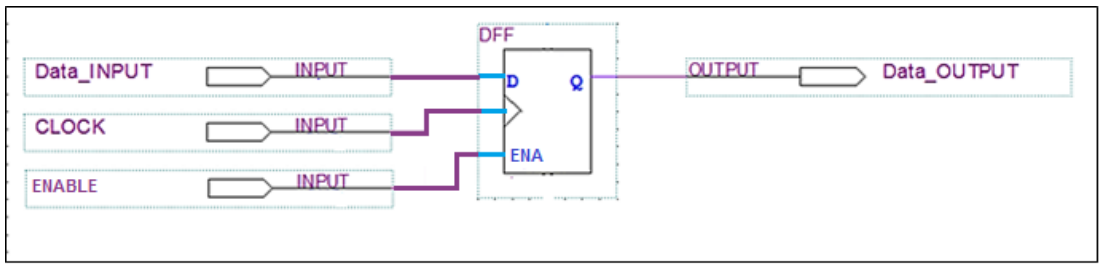


Figure 76 Lab4: Adding a Clock Enable To the Flip-Flop

This function makes the Flip-Flop update to a new value at the Flip-Flop Data_INPUT when the next clock edge arrives and only IF the ENABLE is active. If the Enable is OFF during any rising_edge of the clock signal then the flip-flop implicitly retains its current value (inferred by VHDL).

```
process (<clock_signal>) is
begin
        if (rising_edge(<clock_signal>) then
                if (ENABLE) then
                        register <= <data>;
                end if;
        end if;
end process;
```

The ENABLE can be activated whenever a change is required for a Flip-Flop at the next clock cycle. Now imagine the case where we want to put a string of these Flip-Flops registers together as shown in Figure 77. One can see that the separate registers are activated by when the various ENABLE inputs activate AND with the preceding upstream stage being ACTIVE. The process sequence is DONE when the last stage (Data_Output) is activated in this example.

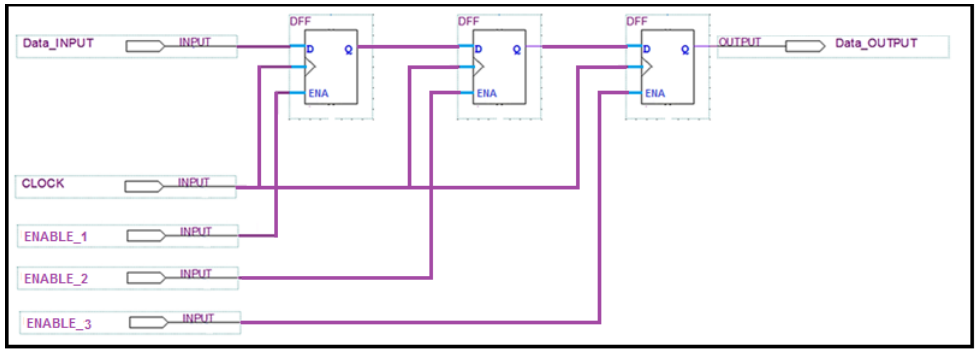


Figure 77 Lab4: Serial String of Flip-Flops

5.4.5 Lab4 Part A – Creating Some Simple Flip-Flop Register Designs

In the first part of this lab we will develop a simple bidirectional shift register and an up/down binary counter. For each of these the total number of register stages used will be eight. The register stages will also be output to the LED's.

5.4.5.1 First Add Some Clock Source Flexibility

We want to be able to run this designs in either a physical, real-world scenario (with a download to the LogicalStep board) OR in a simulation environment. A flexible, high-level control of your design will select the "clocking source" for these two situations.

Within the new top-level design file (LogicalStep_Lab4_top.vhd) near the beginning of the file VHDL code (as shown in Figure 78 below) the design uses the 50MHz clock input (clkin_50) and some new signals are created for flexibility of the clocking network. The first process "**BINCLK**" is a clock divider.

This process is merely a counter (similar to what was used in Lab1) to slow down the shift register clock to something where the register changes can be observed by eye. At the high end of the divider counter outputs is Bit23 and it will output a clock that is 2**24th of the input clock frequency (50 MHz or 50,000,000 Hz). After division Bit23 will change at just under a 4 Hz clock rate. This will be used to drive the signal "**Main_Clk**" for the physical FPGA download version of the LogicalStep_Lab4_top design on the LogicalStep_Board for visual reasons.

However for simulation purposes later on you must change the source for Main_Clk to one that is driven by STIMULUS for the clkin_50 pin **directly** (otherwise you will be waiting for a LOOONNNGGG time for each **Main_Clk** cycle to occur with the 2**24 division of the clock divider output). To make things easier to switch between FPGA-design versions some flexibility should be added to the design.

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

ENTITY LogicalStep_Lab4_top IS
    PORT
    (
    clkin_50    : in  std_logic;
    rst_n       : in  std_logic;
    pb          : in  std_logic_vector(3 downto 0);
    sw          : in  std_logic_vector(7 downto 0); -- The switch inputs
    leds        : out std_logic_vector(7 downto 0); -- for displaying the switch content
    seg7_data   : out std_logic_vector(6 downto 0); -- 7-bit outputs to a 7-segment
    seg7_char1  : out std_logic                     -- seg7 digi selectors
    seg7_char2  : out std_logic                     -- seg7 digi selectors
    );
END LogicalStep_Lab4_top;

ARCHITECTURE SimpleCircuit OF LogicalStep_Lab4_top IS


-----------------------------------------------------------------------------------------------
   CONSTANT SIM                     :  boolean := FALSE;    -- set to TRUE for simulation runs otherwise keep at 0.
   CONSTANT CLK_DIV_SIZE            :  INTEGER := 26;    -- size of vectors for the counters

   SIGNAL   Main_CLK                :  STD_LOGIC;        -- main clock to drive sequencing of State Machine

   SIGNAL   bin_counter             :  UNSIGNED(CLK_DIV_SIZE-1 downto 0); -- := to_unsigned(0,CLK_DIV_SIZE); -- reset binary counter to zero

   SIGNAL   sreg_bits               : std_logic_vector(7 downto 0);
   SIGNAL   Left0_Right1            : std_logic;

-----------------------------------------------------------------------------------------------
BEGIN

-- CLOCKING GENERATOR WHICH DIVIDES THE INPUT CLOCK DOWN TO A LOWER FREQUENCY

BinCLK: PROCESS(clkin_50, rst_n) is
   BEGIN
      IF (rising_edge(clkin_50)) THEN -- binary counter increments on rising clock edge
         bin_counter <= bin_counter + 1;
      END IF;
   END PROCESS;

Clock_Source:
            Main_Clk <=
            clkin_50 when sim = TRUE else          -- for simulations only
            std_logic(bin_counter(23));                     -- for real FPGA operation

-----------------------------------------------------------------------------------------------
```

Figure 78 Lab4: LogicalStep_Lab4_top with Clock Sourcing for FPGA

In Figure 78 a CONSTANT named “SIM” is ahead of the signal declarations in the Architecture Section and it can be used to change the source of Main_Clk. The Boolean constant SIM is used as a selector for a 2-to-1 mux. When SIM is TRUE the Main_Clk signal is driven directly by the Clkin_50 pin as above for **SIM**ulation. When SIM is set to a FALSE value then the Main_Clk signal is connected to the divider output Bit23 for use when running your design on the LogicalStep board.

5.4.5.2 Implementing a Bidirectional Shift Register Component

As mentioned earlier the shift register is to have 8 stages..

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use IEEE.numeric_std.all;
4
5   Entity Bidir_shift_reg is port
6      (
7            CLK            : in  std_logic := '0';
8            RESET_n        : in  std_logic := '0';
9            CLK_EN         : in  std_logic := '0';
10           LEFT0_RIGHT1   : in std_logic  :='0';
11           REG_BITS       : out std_logic_vector(7 downto 0)
12     );
13   end Entity;
14
15   ARCHITECTURE one OF Bidir_shift_reg IS
16
17   Signal sreg          : std_logic_vector(7 downto 0);
18
19
20  BEGIN
21
22  process (CLK, RESET_n) is
23  begin
24     if (RESET_n = '0') then
25              sreg <= "00000000";
26
27     elsif (rising_edge(CLK) AND (CLK_EN = '1')) then
28
29           if (LEFT0_RIGHT1 = '1') then  -- TRUE for RIGHT shift
30
31              sreg (7 downto 0) <= '1' & sreg(7 downto 1); -- right-shift of bits
32
33           elsif (LEFT0_RIGHT1 = '0') then
34
35              sreg (7 downto 0) <= sreg(6 downto 0) & '0'; -- left-shift of bits
36
37           end if;
38
39     end if;
40
41
42  end process;
43  REG_BITS <= sreg;
44
45  END one;
```

Figure 79 Lab4: VHDL for a Bidirectional Shift Register

The shift register will be clocked by the **Main_Clk** clock signal. The Shift Register will only change, be enabled, if the "CLK_EN" (PB(1)) is a '1'. Recall that registers are implemented inside a VHDL processe. Two externally defined control signals (or ports) that should be made "common" to all of the shift register registers will be the input called **Left0_Right1** and **CLK_EN**. When **Left0_Right1** is at '0' the shift register "shifts LEFT". When it is at "1" the shift register "shifts RIGHT". Enter the VHDL code (lines 1 to 45) in Figure 79 into a **VHDL COMPONENT** design and then declare it and Instantiate it at the top level. Connect its outputs to the LEDs, its **Left0_Right1** control input to PB(0) and its **CLK_EN** to PB(1).

5.4.5.3 Functional Simulation of the Bidirectional Shift Register

Up until now the Quartus Simulator has been used to display combinational logic activity using only the Input and Output pins to control and observe the design. This is a limitation with the Quartus Simulator for the simulation of Combinational Logic. That is why in earlier labs when it was desired to view internal signals steps were taken to connect them to the outside pins in the VHDL coding (eg: Lab2 PartA).

But the Quartus simulator also will display internal REGISTERS and this is the first lab session that illustrates that kind of access for observation purposes.

Before you recompile the LogicalStep_Lab4_top for simulation change the clock source for **Main_Clk** by setting the SIM constant to TRUE. Then do an Analysis & Synthesis type of compile for your design.

Besides adding the Input and Output Pins to the Node List in your Functional simulation you can now add the internal registers of the shift register. This can be done by changing the NodeFinder Filter to: Post-Synthesis. Then the register bits of the shift register can be selected.

Refer to Figure 80 below.

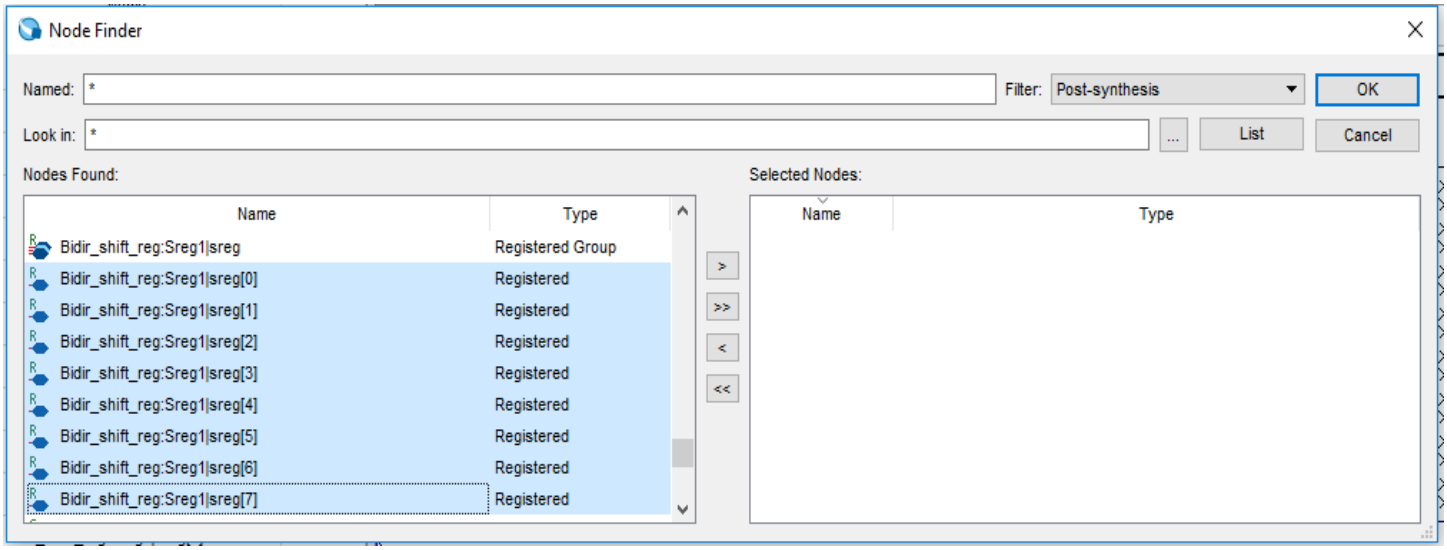


Figure 80 Lab4: Inserting Register Nodes for Simulation

For stimulus do the following:
Set the **Clkin_50** to a 20 nsec period.
Set each of the pb(0) and pb(1) inputs to a '1'.
Set the **rst_n** to a '1' (OFF state) for the whole simulation and then just force it to a '0' for, say, the first 50 ns.
Set the **rst_n** signal return to the '1' state. Halfway in the simulation set the pb(0) to a '0' state.
Sometime after pb(0) is changed try toggling the pb(1) to '0' and then back to a '1'.

Your simulation should look something like that shown in Figure 81.

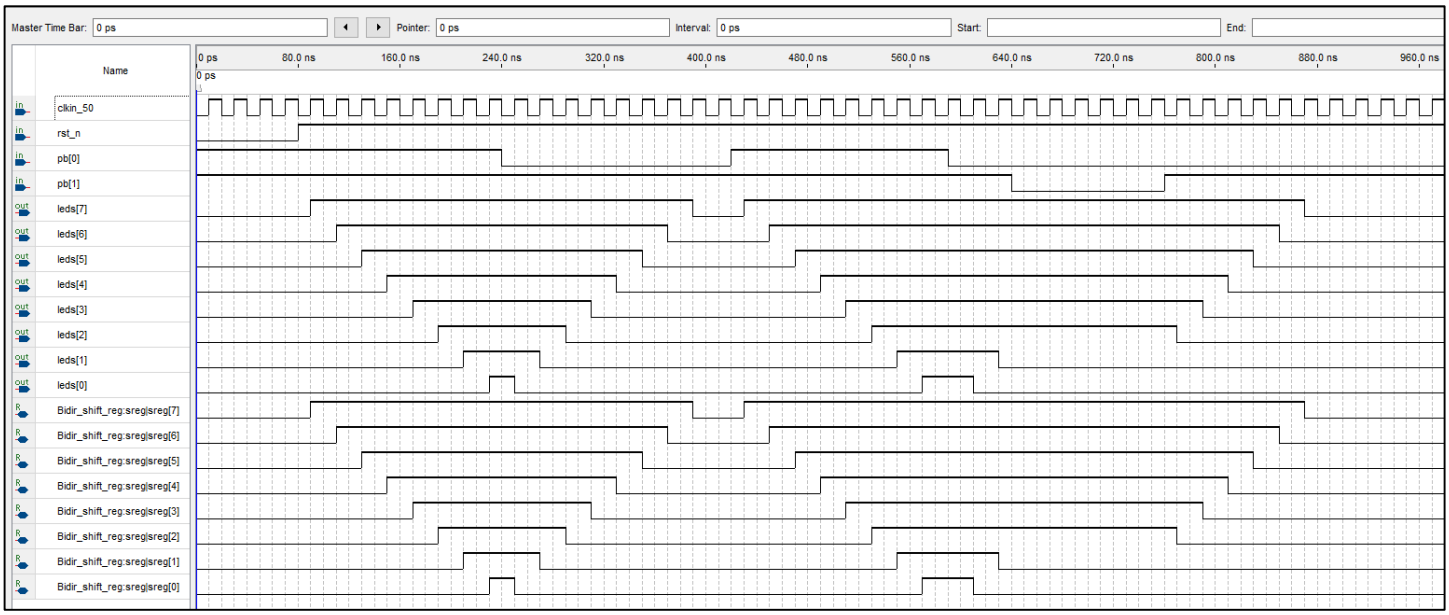


Figure 81 Lab4: Simulation of the Bidirectional Shift Register

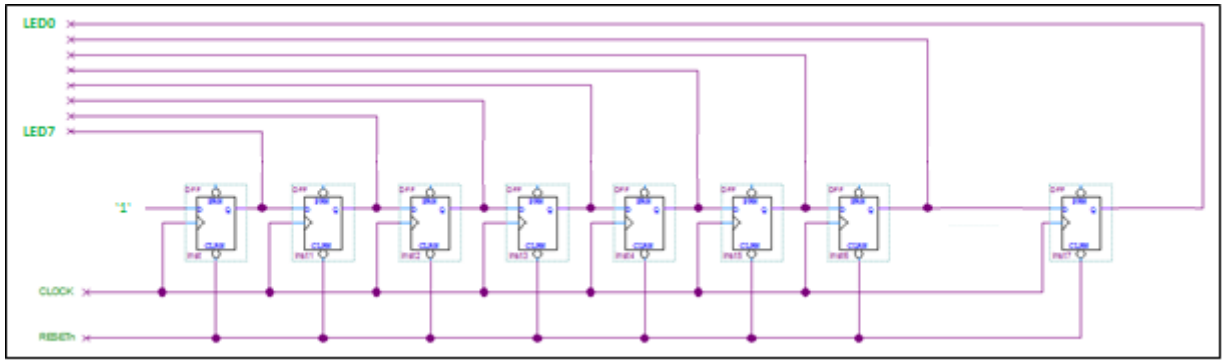


Figure 82: Lab4: Compiler Example of a Unidirectional Shift Register

For clarity purposes Figure 82 shows what the VHDL compiler creates for a simplified version of the shift register(unidirectional only). The flip-flops are only influenced by data from its downstream neighbor. There is no increase in connection complexity as number of shift register stages increases. This design will be compared to one in the next section.

5.4.5.4 Implementing a Simple Up/Down Binary Counter Component

You have already used some binary counters in this Lab course. A simple 8 bit up/down binary counter will now be implemented.

The binary counter will be clocked by the **Main_Clk** clock signal as was done for the shift register example. Recall that registers can only be implemented inside a VHDL process. Two externally defined control signals (or ports) that should be made "common" to all of the shift register registers will be the input called **Up1_Down0** and **clk_en**. When up1_down0 is at '1' the counter counts up. When it is at '0' the counter counts down. Enter the VHDL code (lines 1 to 43) in Figure 83 into a **VHDL COMPONENT** design and then declare it and Instantiate it in your LogicalStep_Lab4_top file. At the top level disconnect the LED's from the shift Register component and then connect them to the counter component outputs, its Up1_Down0 control input to pb(0) and its **clk_en** to pb(1).

The Binary Counter output will only change if the **clk_en** (pb(1)) is a '1' and the **Main_Clk** is running.

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use IEEE.numeric_std.all;
4
5   Entity U_D_Bin_Counter8bit is port
6      (
7             CLK            : in  std_logic := '0';
8             RESET_n        : in  std_logic := '0';
9             CLK_EN         : in  std_logic := '0';
10            UP1_DOWN0      : in std_logic  :='0';
11            COUNTER_BITS   : out std_logic_vector(7 downto 0)
12     );
13    end Entity;
14
15    ARCHITECTURE one OF U_D_Bin_Counter8bit IS
16
17    Signal ud_bin_counter    : UNSIGNED(7 downto 0);
18
19
20  BEGIN
21
22  process (CLK, RESET_n) is
23  begin
24     if (RESET_n = '0') then
25              ud_bin_counter <= "00000000";
26
27     elsif (rising_edge(CLK)) then
28
29       if(( UP1_DOWN0 ='1') AND (CLK_EN ='1'))then
30           ud_bin_counter  <= (ud_bin_counter + 1);
31
32       elsif (( UP1_DOWN0 ='0') AND (CLK_EN ='1')) then
33           ud_bin_counter  <= (ud_bin_counter - 1);
34
35       end if;
36
37     end if;
38
39  end process;
40
41  COUNTER_BITS <= std_logic_vector(ud_bin_counter);
42
43  end;
```

Figure 83 Lab4: Simple Up/Down Binary Counter

5.4.5.5 Functional Simulation of the Up/Down Binary Counter

Similar to what was done for the Bidirectional Shift Register the Up/Down Binary Counter can be simulated. Using the same setup (SIM set to TRUE, Clkin_50 set to 20 nsec period etc.) run a simulation of the design. It should be similar to the 8 bit version that is shown in Figure 84.

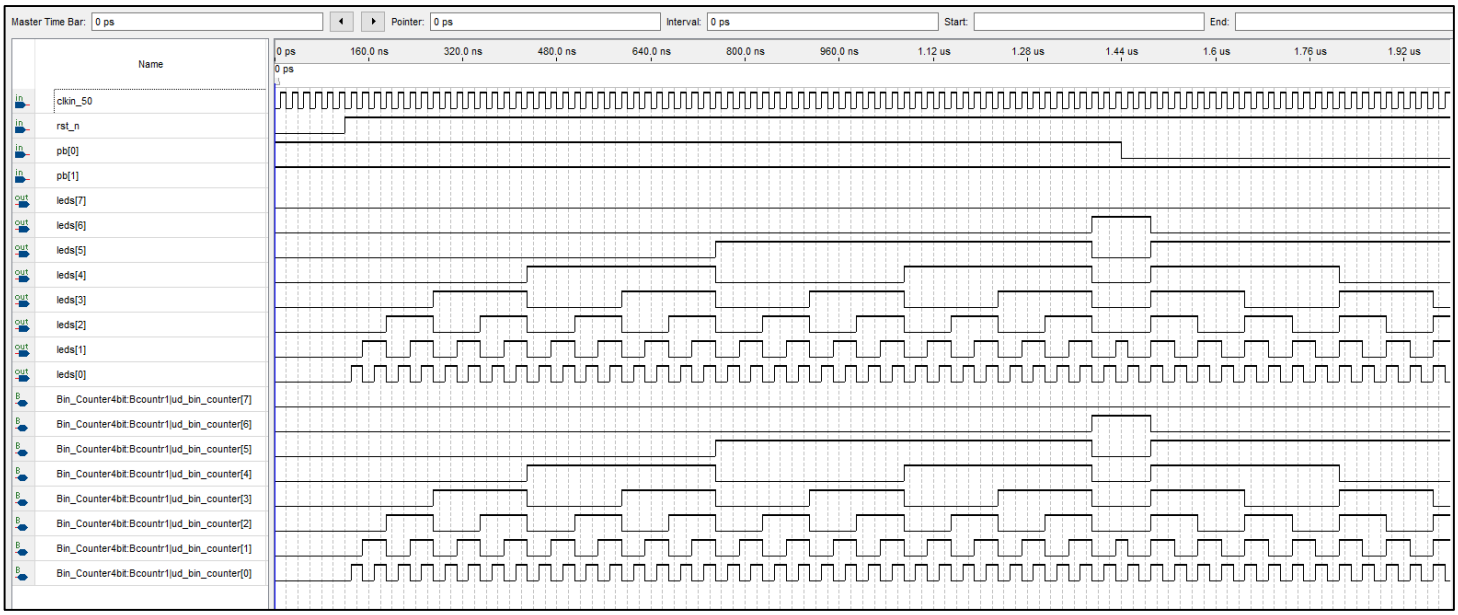


Figure 84 Lab4: Simulation of Up/Down Binary Counter

Side Note: Notice in Figure 84 how each higher-order bit in the binary counter toggles at half of the rate of the preceding lower-order bit (i.e.: bit 4 changes at half the rate of bit 3 which in turn changes at half the rate of bit2 and so on).

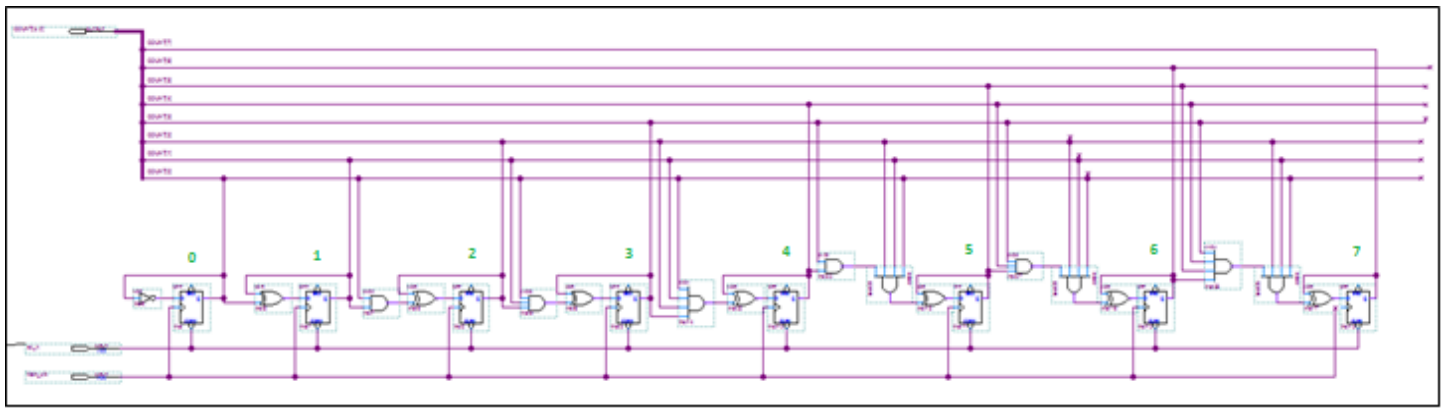


Figure 85 Lab4: VHDL Compiler generated Binary Counter (Up direction only)

For clarity purposes Figure 85 shows what the VHDL compiler creates for a simplified version of the Binary Counter where the clock enable and UP/Down logic are not included.

Notice how much more logic is required as the design gets towards the higher order bits in the counter. The logic “FAN-IN” gets greater and greater for the binary counter higher order register bits.

Each of the register outputs from all lower register bits in the chain come toward the higher order register inputs.

Whereas for the earlier shift register example the logic FAN-IN is maintained to be the same for each stage (the inputs to a register stage just comes from its immediate neighbor).

Each of these two kinds of register designs has an impact on how fast each of these register designs can be “clocked”.

Another aspect to examine in this comparison is with the amount required for register logic for the format choice (encoding format). The 8 bit binary counter has 8 Flip-flops which can represent 2**8 combinations (256 eight-bit binary values). To implement the equivalent number of combinations in the shift register it would require 256 serially connected flip-flops.

Some interesting things can be noted from these two design exercises in terms of design trade-offs:

1) Sequential logic encoding format choice can impact the amount of logic required (both combinational and sequential).
2) Sequential logic encoding format choice can impact the speed of operation.

The focus will now leave the simulation of these simple register designs. These designs will be revisited later on for the Lab4 Project Demo.

5.4.6 New VHDL Component – What is a State Machine?

Now that the topic of basic register functionality has been introduced a new powerful use for registers will be covered. Returning to the earlier trivial process for a Simple Breakfast an outline is shown in Figure 86 and a state diagram is shown in Figure 87.

STATE:	OUTPUTS:	TRANSITION EVENT TO NEXT STATE:
State 1: Get bread from cupboard		< Got Bread >
State 2: Placing bread in toaster	(DROP BREAD IN TOASTER)	< Bread in toaster>
State 3: Get plate	(START TOASTER)	< Got Plate >
State 4: Wait for Toast cycle completion		< Toast Pops Up >
State 5: Put Toast on plate	(GET TOAST FROM TOASTER)	< Toast on Plate >
State 6: Butter the Toast	(APPLY BUTTER)	< Toast Buttered >
State 7: Add Jam	(APPLY JAM)	< Jam Added >
State 8: Process done…..--> Enjoy.	(DONE)	<wait for REPEAT>

Figure 86 Lab4: Process: Getting a Simple Breakfast

Observe the left hand column in Figure 86 that shows the name or "state" steps involved in the process. In the right-hand column are the transition inputs to the process that would be from sensors that could signify events such as "Got_Bread, Bread_in_Toaster_and_Started, Got_Plate" etc.

The process outputs could be used to control the mechanics to "Drop Bread in Toaster", "Start Toaster",…."DONE" etc.

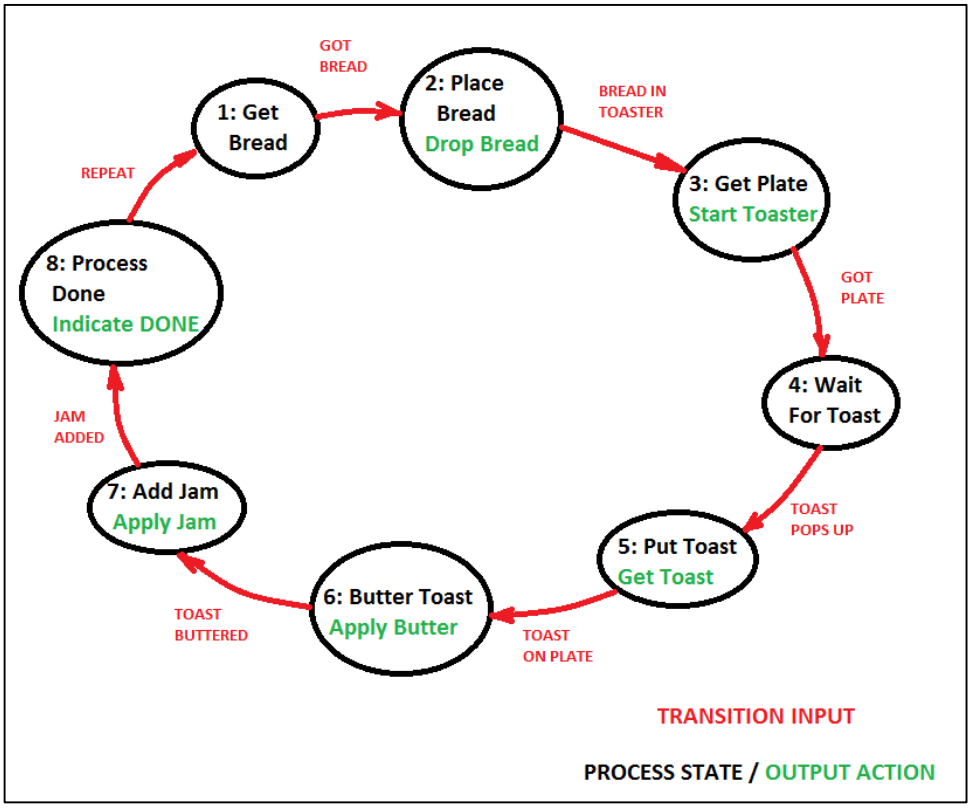
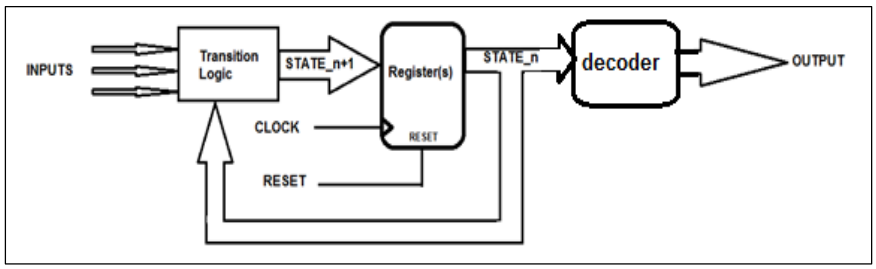


Figure 87 Lab4: State Diagram for Simple Breakfast process

To repeat the process above would require an additional input such as a "Repeat" signal. The above process could be implemented with a STATE MACHINE design.

In a state machine design, the conditions that make it change from the CURRENT state to the NEXT state are varied. Combinational logic is used to determine if or when those state transitions should occur. The state changes occur in synchronization with a state machine clock because registers are involved.

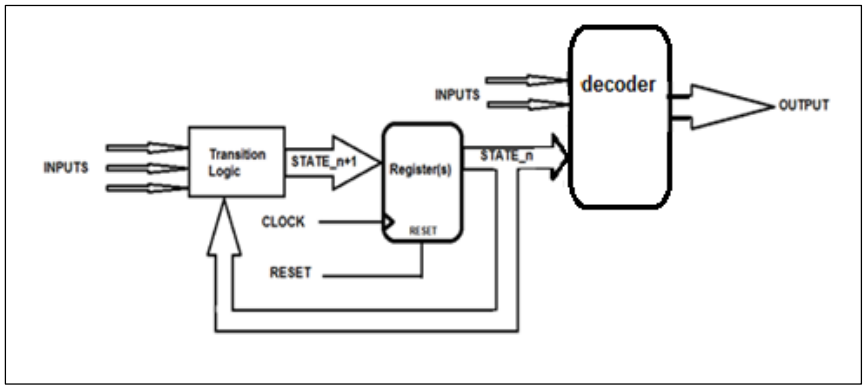
There are two primary classes of state machines used in sequential logic design: Moore and Mealy. To remember the names just think of two engineers sitting at a Thanksgiving table. One engineer politely says to the other "Would you like some Moore corn?" to which the second engineer replies "Only if it's not Mealy". (Groan!).



MOORE STATE MACHINE

Output is a function of present state ONLY (Figure 88).
Moore SM's outputs are only changed when the state machine is clocked.

Figure 88 Lab4: Moore State Machine



MEALY STATE MACHINE
Outputs are functions of present state AND inputs (Figure 89).

Figure 89 Lab4: Mealy State Machine

For this lab both Moore and Mealy state machines are used. Referring to Figures 88 and 89 there are generally just three sections in each state machine design. These are the Register, Transition and Decoder sections. Only the Register section is implemented as a Sequential Logic block. The Transition and Decoder sections are implemented as Combinational Logic block.

Provided in the Lab4 download files from Learn there is a VHDL file named State_Machine_Example. The Entity section for this file is shown in Figure 90. It consists of a clock input, a reset input, a few transition control inputs and two outputs to signal when the process is in specific states.

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

Entity State_Machine_Example IS Port
(
 clk_input, rst_n, I0, I1, I2              : IN std_logic;
 output1, output2                          : OUT std_logic
 );
END ENTITY;
```

Figure 90 Lab4: Entity Declaration for State Machine Example

Further in the VHDL file (shown in Figure 91) in the Architecture declaration section the details set up the list of states for a state machine with a TYPE statement named STATE_NAMES. This gives the state machine the set of state values for its processing.

```
Architecture SM of State_Machine_Example is

TYPE STATE_NAMES IS (S0, S1, S2, S3, S4, S5, S6, S7);   -- list all the STATE_NAMES values
SIGNAL current_state, next_state   :  STATE_NAMES;      -- signals of type STATE_NAMES

BEGIN
```

Figure 91 Lab4: State Machine States Defined by a VHDL TYPE Statement

Then two signals are declared to HOLD two different values of the TYPE STATE_NAMES to keep the "current" state and "next" state assignments for the state machine.

Now the three sections of the State Machine will be created using the VHDL Process construct.

The Register section uses the state machine clock input and usually some kind of RESET input signal in its Sensitivity List. These are used to advance the state machine to new states values in alignment with a particular clock edge or if a RESET signal is applied. This first state machine process is labelled in Figure 92 as "Register_Section". IT IS SEQUENTIAL LOGIC.

```
-------------------------------------------------------------------------------
--State Machine:
-------------------------------------------------------------------------------

-- REGISTER_LOGIC PROCESS:

Register_Section: PROCESS (clk_input, rst_n)  -- this process synchronizes the activity to a clock
BEGIN
   IF (rst_n = '0') THEN
      current_state <= S0;
   ELSIF(rising_edge(clk_input)) THEN
      current_state <= next_State;
   END IF;
END PROCESS;
```

Figure 92 Lab4: Process for the Register Section of State Machine Example

The Transition section (to be located in a second Process construct) uses the VHDL "CASE" statement. It uses a number of the State Machine inputs (not the clock or reset) and the CURRENT state in its Sensitivity List to evaluate what and when the NEXT state will be (including "no change"). A typical example of a CASE statement is shown below in Figure 93 and it must be implemented within a process. NOTE: All "constant_expression" options in a VHDL case statement must be constant and unique. Also, the case statement entries must include a "when others" clause at the end.

```
-- TRANSITION LOGIC PROCESS

Transition_Section: PROCESS (I0, I1, I2, current_state)

BEGIN
     CASE current_state IS

          WHEN S0 =>
             IF(I0='1') THEN
                next_state <= S1;
             ELSE
                next_state <= S0;
             END IF;

          WHEN S1 =>
                next_state <= S2;

          WHEN S2 =>
             IF(I0='1')) THEN
                next_state <= S6;
             ELSIF(I1='1') THEN
                next_state <= S3;
             ELSE
                next_state <= S2;
             END IF;

          WHEN S3 =>
             IF(I0='1') THEN
                next_state <= S4;
             ELSE
                next_state <= S3;
             END IF;

          WHEN S4 =>
                next_state <= S5;

          WHEN S5 =>
                next_state <= S6;

          WHEN S6 =>
             IF(I0='1') THEN
                next_state <= S7;
             ELSE
                next_state <= S6;
             END IF;

          WHEN S7 =>
             IF(I2='1') THEN
                next_state <= S0;
             ELSE
                next_state <= S7;
             END IF;

             WHEN OTHERS =>
                next_state <= S0;
      END CASE;
 END PROCESS;
```

```
case <expression> is
 when <constant_expression> =>
          -- Sequential Statement(s)
 when <constant_expression> =>
          -- Sequential Statement(s)
 when others =>
          -- Sequential Statement(s)
end case;
```

The CASE statement construct can specify a signal group for the <expression> and for each possible value of that signal group is stated in the <constant expression> fields.

An example of Transition section being implemented with a CASE statement is shown in Figure 93 with the process labelled as "Transition_Section". IT IS COMBINATIONAL LOGIC because each IF has an ELSE and there is no clock (RISING_EDGE) statement.
Note how specific inputs are observed by the Transition Logic only in specific states and also how inputs can cause the State Machine to "JUMP" states. For example if an "I0" input signal is active when the State Machine Example is in State "S2" the State Machine will take a jump in its state sequence to state "S6".

The example Decoder section shown in Figure 94 sets the output1 and output2 signals when the state machine state reaches specific states. The output signal levels must be defined for all current_state

values. Figure 94 would be a Decoder section for a Moore State Machine design since the outputs are dependent ONLY on the state machine current_state values. Figure 95 shows an abbreviated version. The Decoder section is COMBINATIONAL LOGIC because there is no clock and all cases are covered.

Figure 94 Lab4: Example of Decoder Logic for Moore State Machine

```
-- DECODER SECTION PROCESS for Moore
Moore_Decoder_Section: PROCESS (current_state)
BEGIN
    CASE current_state IS
        WHEN S0 =>
            output1 <= '1';
            output2 <= '0';
        WHEN S1 =>
            output1 <= '0';
            output2 <= '0';
        WHEN S2 =>
            output1 <= '0';
            output2 <= '0';
        WHEN S3 =>
            output1 <= '0';
            output2 <= '0';
        WHEN S4 =>
            output1 <= '0';
            output2 <= '0';
        WHEN S5 =>
            output1 <= '0';
            output2 <= '0';
        WHEN S6 =>
            output1 <= '0';
            output2 <= '1';
        WHEN S7 =>
            output1 <= '0';
            output2 <= '0';
        WHEN OTHERS =>
            output1 <= '0';
            output2 <= '0';
    END CASE;
END PROCESS;
```

Figure 95 Lab4: Shortened Version of Moore Decoder section

```
-- DECODER SECTION PROCESS for Moore
Moore_Decoder_Section: PROCESS (current_state)
BEGIN
    CASE current_state IS
        WHEN S0 =>
            output1 <= '1';
            output2 <= '0';
        WHEN S6 =>
            output1 <= '0';
            output2 <= '1';
        WHEN OTHERS =>
            output1 <= '0';
            output2 <= '0';
    END CASE;
END PROCESS;
```

For a Mealy State Machine the difference is that the outputs depend on the state value AND the inputs. The inputs may or may not affect a change of the current_state value. In essence the Mealy State Machine state is like a "gate control". If we were to convert the previous Moore State Machine into a Mealy State Machine then the following changes would happen to the Decoder section as shown in Figure 96. Again this section is COMBINATIONAL Logic.

Let's assume that the Transition Logic can remain the same as in the Moore example earlier.

```
--Decoder Section PROCESS for Mealy

Mealy_DECODER_SECTION: PROCESS(I0, current_state)

  BEGIN
    IF (current_state = S0) THEN
          output1 <= I0;

    ELSIF (current_state = S6) THEN
          output2 <= I0;
    ELSE
          output1 <= '0';
          output2 <= '0';
    END IF;
END PROCESS;
```

Figure 96 Lab4: Mealy State Machine Decoder Section with Extra Input and Output Added

From Figure 96 it can be deduced that output1 and output2 are combinational outputs driven by the input I0 but only when the state machine is in state S0 (for output1) or state S6 (for output2). If required the outputs can be driven during numerous different states with the appropriate inputs being considered.

Note that for Mealy State machines, with the outputs being driven by combinational inputs that the outputs can suffer from any kind of asynchronous behaviour of the inputs. For example, if I0 is "intermittent" then output1 and output2 will be "intermittent" as well during the enabling states. This is the main disadvantage of Mealy machines as compared to Moore machines. The advantage for Mealy State Machines is that often they can be designed with fewer states than their Moore machine counterparts. They can also change the outputs sooner since a state change may not have to occur.

Alternatively an example for the Moore machine case, the output1 would be driven by a current_state value of "S0"only. Outside of the S0 state the direct input connection from "I0" to the outputs would be moot. The advantage that Moore State Machines over their Mealy State Machine counterparts is that external inputs are prevented from influencing the State Machine outputs directly. The State Machine must change "state" first and then the outputs are driven by decoding the "current_state" value.

Which type of state machine is used depends on the requirements of the application.

5.4.7 Lab4 Part B - Project Brief for Lab4 Demo

The Lab4 Project will encompass all of the components that you have designed earlier this term and in addition to the components that were developed earlier in this lab session.

You will creating a Robotic Arm Controller or RAC (illustrated in Figure 96 and its Control System in Figure 97) that could be used for a positioning a robotic arm in 2 dimensions and employing an extender/grappler.

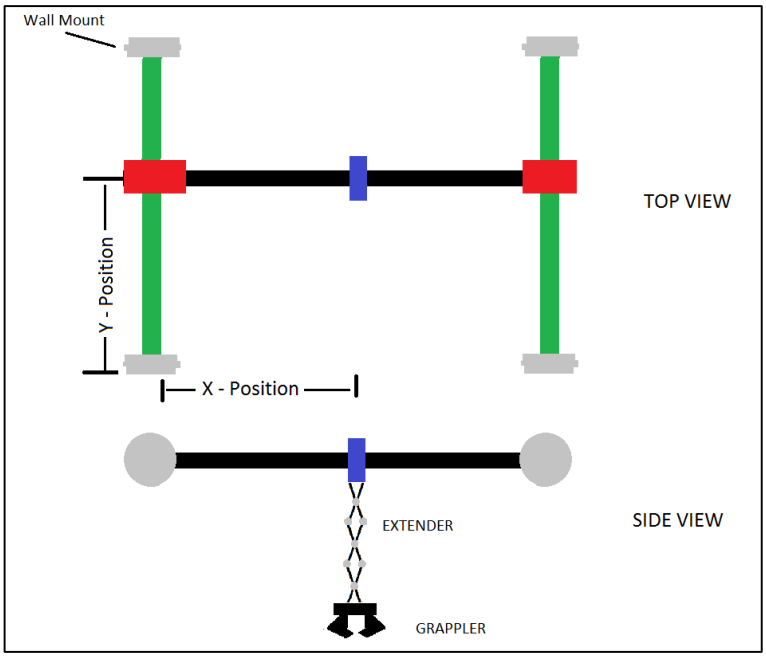


Figure 97 Lab4: Robotic Arm Project

Your Lab4 project will include the following declared as VHDL COMPONENTS in the LogicalStep_Lab4_top.vhd file:

1) 4-bit Magnitude Comparator (2 instances)

2) Seven-Segment Decoder (2 instances)

3) Seg7_Mux (1 instance)

4) Bidirectional 4-bit Shift Register (1 instance - changed from 8-bit version earlier)

5) Binary 4-bit up/down Counter (2 instances - changed from 8-bit version earlier)

6) Mealy State Machine (1 instance) to be used as the X/Y Position Controller

7) Moore State Machine (2 instances) one for Extender and one for Grappler

8) all multiplexers

The above VHDL components are to be declared and used in the LogicalStep_Lab4_top.vhd file **ONLY.**

There will also be some other I/O elements used in the Lab4 project:

1) 8 slide switch inputs:

SW[7:4] – Target X co-ordinate;

SW[3:0] – Target Y co-ordinate

2) 4 Push Buttons:

pb(3) – X Drive Enable;

pb(2) – Y Drive Enable;

pb(1) Extender Toggle (In/Out);

pb(0) Grappler Toggle (Open/Closed);

3) 2 seven segment displays:

Digit1 – Multiplexed between Target X-co-ordinate when pb(3) open and ; X-current position when pb(3) is pressed;

Digit2 – Multiplexed between Target Y-co-ordinate when pb(2) open; Y-current position when pb(2) is pressed;

4) 8 LED outputs:

leds[7:4] – Extender position;

leds[3] -- Grappler ON;

leds[2:1] -- for your use;

leds[0] -- System Error;

Figure 98 shows the control system block diagram. The Motor Drive electronics are external and will be driven by the X/Y Comparators. The comparator outputs (EQ, GT, LT) also are connected back into the X/Y Position Controller (Mealy SM). The Comparator inputs are from the 4 bit Binary Counters and the Slide Switches.

The slide switches will be used to set the desired X and Y TARGET co-ordinates for the Robotic Arm Controller (RAC). The hex-coded switches SW[7:4] value will represent the X TARGET co-ordinate and the hex-coded switches SW[3:0] value will represent the Y TARGET co-ordinate for the RAC.

The seven-segment displays are multiplexed between the Target and the “in-motion” current X/Y position of the RAC (X on Digit1 and Y on Digit2).

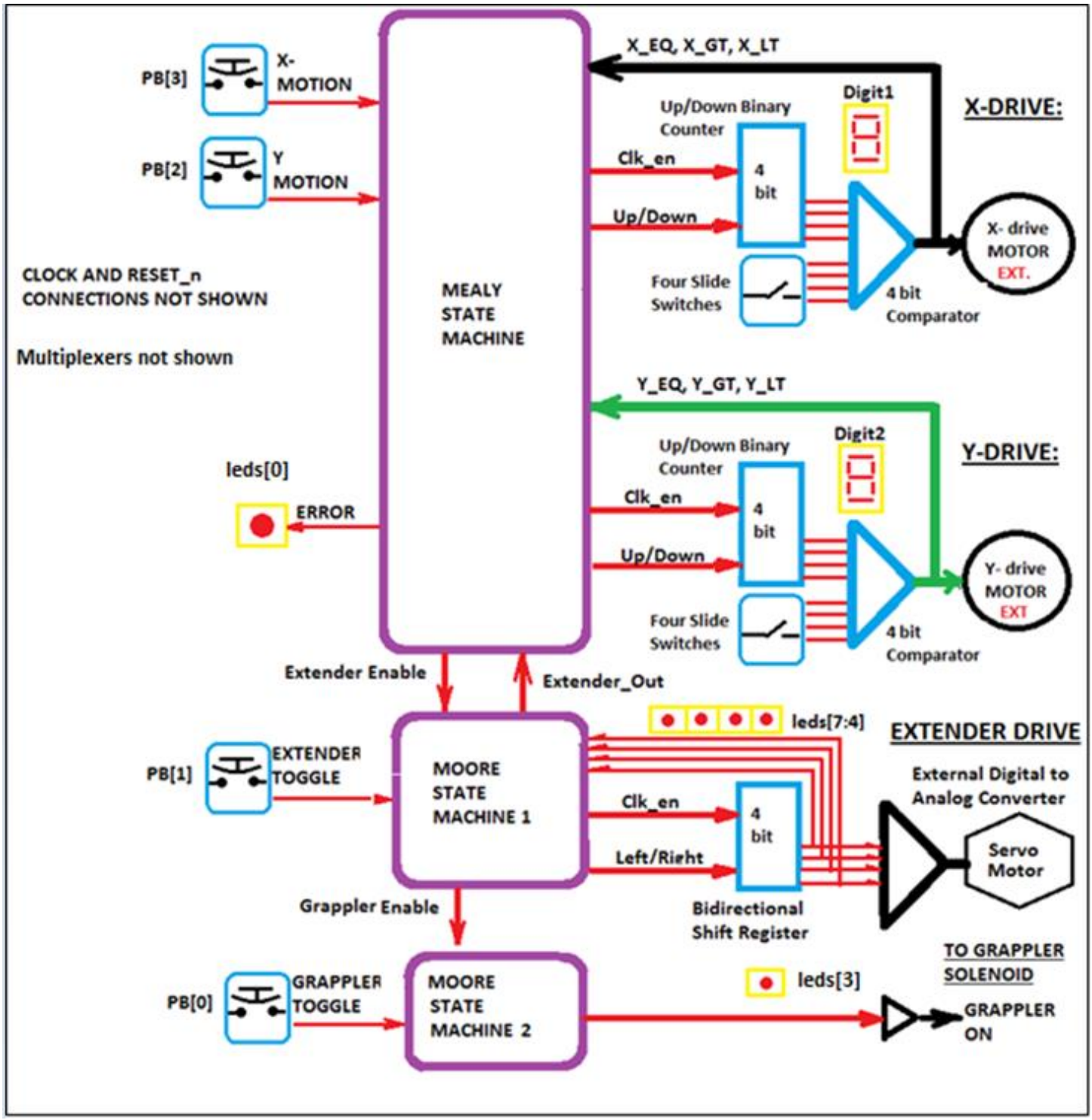


Figure 98 Lab4: Lab4 Project Block Diagram

The RAC will be considered “in X-motion” when PB[3] is pressed and RAC has not yet reached its TARGET X-location. The RAC will be considered “in Y-motion” when PB[2] is pressed and RAC has not yet reached its TARGET Y-location. If either pb(3) or pb(2) are not pressed before the Target Position is reached the appropriate binary counter will just retain its current value. The X-motion and Y-motion

are completely independent operations. They should be able to operate either sequentially or be able to operate in parallel.

When the RAC reaches its TARGET X/Y co-ordinates the Extender can be enabled for operation. When "enabled by the X/Y Position Controller the Extender Controller (Moore SM1) may extend the extender. A single bidirectional shift register is to represent the extender position. It is to be toggled either ON or OFF by push button pb(1). When activated the extender begins its extension process and continues until it reaches FULL extension. The extender position sequence is to be displayed on leds[7:4] as shown below:

Position:	leds[7:4]
Retracted:	0000
Extending1:	1000
Extending2:	1100
Extending3:	1110
Fully Extended:	1111

Anytime that the extender is NOT in its retracted position ("0000") the Moore SM1 status flag "Extender_Out" must be driven active. Only in the Fully Extended position ("1111") is the Grappler Controller (Moore SM2) to be enabled for Open/Close operations.

To Toggle the Open/Closed state of the Grappler the input pb(0) is used. The Grappler "Closed" state is displayed on leds[3].

NOTE: If any attempt is made to move the RAC to new Target co-ordinates while the Extender_Out flag is active the X/Y Position Controller must be PREVENTED from X or Y motion and a System Error must be indicated on leds[0]. The System Error will also be displayed on BOTH Digit1 and Digit2 by exhibiting a FLASHING condition. This Error Condition must remain active (locked) until the Extender is Retracted. When retracted the Moore SM1 Extender_Out flag is turned OFF, the System Error conditioned is turned off and then X/Y motion may continue.

5.4.7.1 Mealy State Machine Requirements for X/Y Position Control

The Mealy State Machine for the Lab4 Project will consist of 3 primary sections. These will be the Register, Transition and Decoder sections. The Register section will be clocked by the rising edge of the Main_Clk input. A reset input must also be included to clear the Register section back to its initial state.

The Transition section will allow the states to change states according to the inputs received from the Comparators, Push-Button switches pb[3:2] and the Extender Controller.

The Decoder section MUST be of the MEALY form. Outputs should be defaulted to '0' but when activated to a '1' level the outputs are NOT to be driven by the "current_state" alone but by inputs

during specific current_state values. Hybrid forms of decoding (mix of Moore and Mealy) will result in losing marks for Lab4.

5.4.7.2 Moore State Machine 1 and 2 Requirements

The Moore State Machine 1 (Extender Controller) for the Lab4 Project will consist of 3 primary sections. These will be the Register, Transition and Decoder sections. The Register section will be clocked by the rising edge of the Main_Clk input. A reset input must also be included to clear the Register section back to its initial state. The Moore State Machine 2(Grappler Controller) is provided.

For Moore SM1 the Transition section will allow the states to change states according to the inputs received from the bidirectional Shift Register, Extender_Enable and the Push-Button pb[1].

The Moore SM2 the Transition section (**provided to you on LEARN**) allows the states to change states according to the inputs received from the Grappler_Enable input and the Push-Button pb[0].

The Decoder section MUST be of the MOORE form. Outputs should be defined for each of the "current_state" values only.

Hybrid forms of decoding (mix of Moore and Mealy) will result in losing marks for Lab4.

5.4.7.3 Multiplexers

Any Multiplexers used in the design are to be declared and instantiated at the top level to make for easier readability etc. All functionality will be reviewed and marked.

5.4.7.4 State Diagrams

State diagrams for the MOORE SM1 and MEALY state machine can be created by the Quartus tools. Use the Tools>Netlist Viewers> State Machine Viewer to create them. Include these diagrams in your report.

5.4.7.5 Fitter Report -Resources Used by Entity Diagrams

The Fitter Report can be copied from the Quartus tools Compiler Report.

Include the Resources Utilization by Entity in your report for your design.

5.4.7.6 Lab4 Project Input / Output Definitions

SIGNAL TYPE:	SIGNAL NAME:	ASSIGNED PORT(s):	Comment
Inputs	Main_clk	n/a	Main clock driven by clock source logic
	rst_n	rst_n	RESET (Active_LOW)
	sw[7..4]	sw[7..4]	Target X- Co-ordinate (in hex)
	sw[3..0]	sw[3..0]	Target Y- Co-ordinate (in hex)
	pb[3]	pb[3]	X-Motion Enable
	pb[2]	pb[2]	Y-Motion Enable
	pb[1]	pb[1]	Extender Toggle
	pb[0]	pb[0]	Grappler Toggle
Outputs	seg7_data[6..0]	seg7_data[6..0]	Used to display TARGET values for X (on Digit1 and Y (on Digit2).
	seg7_char1, seg7_char2	seg7_char1, seg7_char2	Target X/Y seven segment display control by seg7_mux
	leds[7:4]	leds[7:4]	Shows extender position
	leds[3]	leds[3]	Shows Grappler activation
	leds[2:1]	leds[2:1]	User LEDs.
	leds[0]	leds[0]	Shows System Error

5.5 POST – Lab4 Activities

1. Demo's of the Lab4 designs will be required at the next lab session in Lab 5.

2. After the Lab 3 demo, take a picture of the completed Submission form and upload it to the Learn Lab 4 Demo Dropbox.

3. A report on the Lab4 design is due within 24 hours after your Lab Demo. Refer to the Lab4 Submission Form for its requirement details.

Submit your completed project report (pdf format) to LEARN ECE-124 Lab4 Reports Dropbox folder according to your Session and Group Number naming format.

5.6 LAB4 SUBMISSION FORM

Table 4 - Lab4: Submission Form

ECE-124 Lab-4 Submission Form			
GROUP NUMBER:	Lab4 Demo	Lab4 Report	**TA:**
LAB SECTION: 20_	Out of 10	Out of 10	

I am submitting this report for grading. I certify that this report, including any code, descriptions, flowcharts as part of the submission were written by the team member(s) below and there has not been any use of prior academic credit at this university or any other institution. The penalty for plagiarism or submission without signature(s) will be a grade of zero

NAME: (Print)	**UW User ID (not Student ID)**	**Signature**
Partner A:		
Partner B:		

LAB4 DESIGN DEMO	**Marks Allotted**	**A**	**B**
Target X value on Digit1 (pb3 OFF); Target Y value on Digit2 (pb2 OFF)	1		
X-Motion/Y-Motion has changing values on Digit1/Digit2	1		
Extender enabled only at Target co-ordinates	1		
Extender Position shown on leds[7:4]	1		
Grappler enabled only at Fully Extended Extender (Grappler- led[3])	1		
System Error when X/Y Motion with Extender not retracted	1		
System Error Cleared when Extender is retracted.	1		
DISCUSSION: Comment on your VHDL Implementation?	3		
LAB4 DEMO MARK	**Out of 10**		

LAB4 DESIGN REPORT (see rubric on LEARN for details)	**Marks Allotted**	**TEAM**
Structural VHDL for Top Level VHDL file (only instances and connections) – no gates except in instance input fields	2	
Simulation of 8bit Shift Register and 8 bit Binary Counter in both directions	2	
State Diagrams of Mealy SM, Moore SM1 machines	2	
Mealy Form for Mealy SM; Moore form for Moore SM1	2	
Fitter Report on Resources Utilization by Entity (Logic Cells each)	2	
Delay in Report Submission (-1 per day) x number of days:		
LAB4 REPORT MARK	**Out of 10**	